ARM9E-S

(Rev 1)

Technical Reference Manual

Copyright © 1999, 2000 ARM Limited. All rights reserved.

ARM DDI 0165B

ARM9E-S
Technical Reference Manual

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Release Information

Change history

Date	Issue	Change
16th December 1999	A	First release.
12th September 2000	B	Second release.

Proprietary Notice

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

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, PrimeCell, ARM7TDMI, ARM7TDMI-S, ARM9TDMI, ARM9E-S, ARM946E-S, ARM966E-S, ETM7, ETM9, TDMI, and STRONG are trademarks of ARM Limited.

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

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

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

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

Figure C-2 on page C-4 reprinted with permission IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture Copyright 2000, by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM9E-S Technical Reference Manual

Preface

About this document ... xvi
Further reading ... xix
Feedback ... xx

Chapter 1 Introduction

1.1 About the ARM9E-S ... 1-2
1.2 ARM9E-S architecture ... 1-5
1.3 ARM9E-S block, core, and interface diagrams ... 1-7
1.4 ARM9E-S instruction set summary ... 1-10

Chapter 2 Programmer's Model

2.1 About the programmer's model ... 2-2
2.2 Processor operating states ... 2-3
2.3 Memory formats ... 2-4
2.4 Instruction length ... 2-6
2.5 Data types ... 2-7
2.6 Operating modes ... 2-8
2.7 Registers ... 2-9
2.8 The program status registers ... 2-16
2.9 Exceptions ... 2-20

 Copyright © 2000 ARM Limited. All rights reserved.

Chapter 3 Device Reset

3.1 About device reset 3-2
3.2 Reset modes 3-3
3.3 ARM9E-S behavior on exit from reset 3-5

Chapter 4 Memory Interface

4.1 About the memory interface 4-2
4.2 Instruction interface 4-3
4.3 Instruction interface addressing signals 4-4
4.4 Instruction interface data timed signals 4-6
4.5 Endian effects for instruction fetches 4-7
4.6 Instruction interface cycle types 4-8
4.7 Data interface 4-13
4.8 Data interface addressing signals 4-15
4.9 Data interface data timed signals 4-18
4.10 Data interface cycle types 4-24
4.11 Endian effects for data transfers 4-30
4.12 Use of CLKEN to control bus cycles 4-31

Chapter 5 Interrupts

5.1 About interrupts 5-2
5.2 Hardware interface 5-3
5.3 Maximum interrupt latency 5-7
5.4 Minimum interrupt latency 5-8

Chapter 6 ARM9E-S Coprocessor Interface

6.1 About the coprocessor interface 6-2
6.2 LDC/STC 6-4
6.3 MCR/MRC 6-8
6.4 MCRR/MRRC 6-10
6.5 Interlocked MCR 6-12
6.6 Interlocked MCRR 6-13
6.7 CDP 6-14
6.8 Privileged instructions 6-16
6.9 Busy-waiting and interrupts 6-17
6.10 Coprocessor 15 MCRs 6-18
6.11 Connecting coprocessors 6-19

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 7 Debug Interface and EmbeddedICE-RT

7.1 About the debug interface 7-2
7.2 Debug systems 7-3
7.3 About EmbeddedICE-RT 7-6
7.4 Disabling EmbeddedICE-RT 7-8
7.5 Debug interface signals 7-9
7.6 ARM9E-S core clock domains 7-14
7.7 Determining the core and system state 7-15
7.8 The debug communications channel 7-16
7.9 Monitor mode debug 7-21

Chapter 8 Instruction Cycle Times

8.1 Instruction cycle count summary 8-3
8.2 Introduction to detailed instruction cycle timings 8-7
8.3 Branch and ARM branch with link 8-8
8.4 Thumb branch with link 8-9
8.5 Branch and exchange 8-10
8.6 Thumb Branch, Link, and Exchange <immediate> 8-11
8.7 Data operations 8-12
8.8 MRS 8-14
8.9 MSR operations 8-15
8.10 Multiply and multiply accumulate 8-16
8.11 QADD, QDADD, QSUB, and QDSUB 8-20
8.12 Load register 8-21
8.13 Store register 8-26
8.14 Load multiple registers 8-27
8.15 Store multiple registers 8-30
8.16 Load double register 8-31
8.17 Store double register 8-32
8.18 Data swap 8-33
8.19 PLD 8-35
8.20 Software interrupt, undefined instruction, and exception entry 8-36
8.21 Coprocessor data processing operation 8-37
8.22 Load coprocessor register (from memory) 8-38
8.23 Store coprocessor register (to memory) 8-40
8.24 Coprocessor register transfer (to ARM) 8-42
8.25 Coprocessor register transfer (from ARM) 8-43
8.26 Double coprocessor register transfer (to ARM) 8-44
8.27 Double coprocessor register transfer (from ARM) 8-45
8.28 Coprocessor absent 8-46
8.29 Unexecuted instructions 8-47

Chapter 9 AC Parameters

9.1 Timing diagrams 9-2
9.2 AC timing parameter definitions 9-8

Copyright © 2000 ARM Limited. All rights reserved.

Appendix A Signal Descriptions

A.1 Clock interface signals A-2
A.2 Instruction memory interface signals A-3
A.3 Data memory interface signals A-4
A.4 Miscellaneous signals A-6
A.5 Coprocessor interface signals A-7
A.6 Debug signals A-8

Appendix B Differences Between the ARM9E-S and the ARM9TDMI

B.1 Interface signals B-2
B.2 ATPG scan interface B-5
B.3 Timing parameters B-6
B.4 ARM9E-S design considerations B-7
B.5 ARM9E-S debugger considerations B-9

Appendix C Debug in depth

C.1 Scan chains and JTAG interface C-2
C.2 Resetting the TAP controller C-5
C.3 Instruction register C-6
C.4 Public instructions C-7
C.5 Test data registers C-10
C.6 ARM9E-S core clock domains C-17
C.7 Determining the core and system state C-18
C.8 Behavior of the program counter during debug C-24
C.9 Priorities and exceptions C-27
C.10 EmbeddedICE-RT logic C-28
C.11 Vector catching C-39
C.12 Single-stepping C-40
C.13 Coupling breakpoints and watchpoints C-41
C.14 Disabling EmbeddedICE-RT C-44
C.15 EmbeddedICE-RT timing C-45

Copyright © 2000 ARM Limited. All rights reserved.

List of Tables
ARM9E-S Technical Reference Manual

Table 1-1	Key to tables	1-10
Table 1-2	ARM instruction set summary	1-12
Table 1-3	Addressing mode 2	1-16
Table 1-4	Addressing mode 2 (privileged)	1-17
Table 1-5	Addressing mode 3	1-18
Table 1-6	Addressing mode 4 (load)	1-18
Table 1-7	Addressing mode 4 (store)	1-18
Table 1-8	Addressing mode 5 (load)	1-19
Table 1-9	Oprnd2	1-19
Table 1-10	Fields	1-20
Table 1-11	Condition fields	1-20
Table 1-12	Thumb instruction set summary	1-21
Table 2-1	Register mode identifiers	2-10
Table 2-2	PSR mode bit values	2-18
Table 2-3	Exception entry and exit	2-20
Table 2-4	Configuration of exception vector address locations	2-26
Table 2-5	Exception vectors	2-26
Table 3-1	Reset modes	3-3
Table 4-1	Transfer widths	4-4
Table 4-2	InTRANS encoding	4-5
Table 4-3	Significant address bits	4-7
Table 4-4	32-bit instruction fetches	4-7

Copyright © 2000 ARM Limited. All rights reserved.

Table 4-5	Halfword accesses	4-7
Table 4-6	Cycle types	4-8
Table 4-7	Burst types	4-10
Table 4-8	Transfer widths	4-16
Table 4-9	DnTRANS encoding	4-16
Table 4-10	Transfer size encoding	4-21
Table 4-11	Significant address bits	4-21
Table 4-12	Word accesses	4-22
Table 4-13	Halfword accesses	4-22
Table 4-14	Byte accesses	4-22
Table 4-15	Cycle types	4-24
Table 4-16	Burst types	4-28
Table 6-1	Handshake signals	6-7
Table 6-2	Handshake signal connections	6-20
Table 7-1	Coprocessor 14 register map	7-16
Table 8-1	Key to tables	8-3
Table 8-2	ARM instruction cycle counts	8-3
Table 8-3	Key to cycle timing tables	8-7
Table 8-4	Branch and ARM branch with link cycle timings	8-8
Table 8-5	Thumb branch with link cycle timing	8-9
Table 8-6	Branch and exchange cycle timing	8-10
Table 8-7	Thumb branch, link and exchange cycle timing	8-11
Table 8-8	Data operation cycle timing	8-12
Table 8-9	MRS cycle timing	8-14
Table 8-10	MSR cycle timing	8-15
Table 8-11	MUL and MLA cycle timing	8-17
Table 8-12	MULS and MLAS cycle timing	8-17
Table 8-13	SMULL, UMULL, SMLAL, and UMLAL cycle timing	8-18
Table 8-14	SMULLS, UMULLS, SMLALS, and UMLALS cycle timing	8-18
Table 8-15	SMULxy, SMLAxy, SMULWy, and SMLAWy cycle timing	8-19
Table 8-16	SMLALxy cycle timing	8-19
Table 8-17	QADD, QDADD, QSUB, and QDSUB cycle timing	8-20
Table 8-18	Load register operation cycle timing	8-23
Table 8-19	Cycle timing for load operations resulting in interlocks	8-24
Table 8-20	Example sequence LDRB, NOP and ADD cycle timing	8-24
Table 8-21	Example sequence LDRB and STMIA cycle timing	8-25
Table 8-22	Store register operation cycle timing	8-26
Table 8-23	LDM cycle timing	8-28
Table 8-24	STM cycle timing	8-30
Table 8-25	Data swap cycle timing	8-33
Table 8-26	PLD operation cycle timing	8-35
Table 8-27	Exception entry cycle timing	8-36
Table 8-28	Coprocessor data operation cycle timing	8-37
Table 8-29	Load coprocessor register cycle timing	8-38
Table 8-30	Store coprocessor register cycle timing	8-40
Table 8-31	MRC instruction cycle timing	8-42
Table 8-32	MCR instruction cycle timing	8-43

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-33 MRRC instruction cycle timing 8-44
Table 8-34 MCRR instruction cycle timing 8-45
Table 8-35 Coprocessor absent instruction cycle timing 8-46
Table 8-36 Unexecuted instruction cycle timing 8-47
Table 9-1 Target AC timing parameters 9-8
Table A-1 Clock interface signals A-2
Table A-2 Instruction memory interface signals A-3
Table A-3 Data memory interface signals A-4
Table A-4 Miscellaneous signals A-6
Table A-5 Coprocessor interface signals A-7
Table A-6 Debug signals A-8
Table B-1 ARM9E-S signals and ARM9TDMI hard macrocell equivalents ... B-2
Table C-1 Public instructions C-7
Table C-2 Scan chain number allocation C-12
Table C-3 Scan chain 1 bit order C-15
Table C-4 ARM9E-S EmbeddedICE-RT logic register map C-28
Table C-5 Watchpoint control register for data comparison functions C-31
Table C-6 Watchpoint control register for instruction comparison functions C-33
Table C-7 Debug control register bit functions C-34
Table C-8 Interrupt signal control C-35
Table C-9 Debug status register bit functions C-36

 Copyright © 2000 ARM Limited. All rights reserved.

Copyright © 2000 ARM Limited. All rights reserved.

List of Figures
ARM9E-S Technical Reference Manual

Figure 1-1	Five-stage pipeline	1-3
Figure 1-2	The instruction pipeline	1-4
Figure 1-3	ARM9E-S block diagram	1-7
Figure 1-4	ARM9E-S core diagram	1-8
Figure 1-5	ARM9E-S interface diagram	1-9
Figure 2-1	Big-endian addresses of bytes within words	2-4
Figure 2-2	Little-endian addresses of bytes within words	2-5
Figure 2-3	Register organization in ARM state	2-11
Figure 2-4	Register organization in Thumb state	2-13
Figure 2-5	Mapping of Thumb state registers onto ARM state registers	2-14
Figure 2-6	Program status register	2-16
Figure 3-1	Power-on reset	3-3
Figure 3-2	ARM9E-S behavior on exit from reset	3-5
Figure 4-1	Simple memory cycle	4-8
Figure 4-2	Nonsequential instruction fetch cycle	4-9
Figure 4-3	Sequential instruction fetch cycles	4-11
Figure 4-4	Merged I-S cycle	4-12
Figure 4-5	ARM9TDMI effect of DABORT on following memory access	4-19
Figure 4-6	ARM9E-S aborted data memory access	4-20
Figure 4-7	Data replication	4-23
Figure 4-8	Simple memory cycle	4-24
Figure 4-9	Nonsequential data memory cycle	4-26
Figure 4-10	Back to back memory cycles	4-27
Figure 4-11	Sequential access cycles	4-28

 Copyright © 2000 ARM Limited. All rights reserved.

Figure 4-12 Use of CLKEN 4-31
Figure 4-13 Alteration of next memory request during waited bus cycle 4-32
Figure 5-1 Retaking the FIQ exception 5-4
Figure 5-2 Stopping CLK for power saving 5-5
Figure 5-3 Using CLK and CLKEN for best interrupt latency 5-6
Figure 6-1 ARM9E-S LDC/STC cycle timing 6-4
Figure 6-2 ARM9E-S coprocessor clocking 6-5
Figure 6-3 ARM9E-S MCR or MRC transfer timing 6-8
Figure 6-4 ARM9E-S MCRR or MRRC transfer timing 6-10
Figure 6-5 ARM9E-S interlocked MCR 6-12
Figure 6-6 ARM9E-S interlocked MCRR 6-13
Figure 6-7 ARM9E-S late-canceled CDP 6-14
Figure 6-8 ARM9E-S privileged instructions 6-16
Figure 6-9 ARM9E-S busy waiting and interrupts 6-17
Figure 6-10 ARM9E-S coprocessor 15 MCRs 6-18
Figure 6-11 Coprocessor connections 6-19
Figure 7-1 Typical debug system 7-3
Figure 7-2 ARM9E-S block diagram 7-5
Figure 7-3 The ARM9E-S, TAP controller, and EmbeddedICE-RT 7-6
Figure 7-4 Breakpoint timing 7-9
Figure 7-5 Watchpoint entry with data processing instruction 7-11
Figure 7-6 Watchpoint entry with branch 7-12
Figure 7-7 Clock synchronization 7-14
Figure 7-8 Debug comms channel control register 7-17
Figure 7-9 Coprocessor 14 monitor mode debug status register format 7-18
Figure 9-1 Instruction memory interface timing 9-2
Figure 9-2 Data memory interface timing 9-3
Figure 9-3 Clock enable timing 9-3
Figure 9-4 Coprocessor interface timing 9-4
Figure 9-5 Exception and configuration timing 9-4
Figure 9-6 Debug interface timing 9-5
Figure 9-7 Interrupt sensitivity status timing 9-5
Figure 9-8 JTAG interface timing 9-6
Figure 9-9 DBGSDOUT to DBGTDO relationship 9-7
Figure C-1 ARM9E-S scan chain arrangements C-2
Figure C-2 Test access port controller state transitions C-4
Figure C-3 ID code register format C-11
Figure C-4 Typical scan chain cell C-13
Figure C-5 Debug exit sequence C-22
Figure C-6 Debug state entry C-23
Figure C-7 ARM9E-S EmbeddedICE macrocell overview C-30
Figure C-8 Watchpoint control register for data comparison C-31
Figure C-9 Watchpoint control register for instruction comparison C-32
Figure C-10 Debug control register format C-34
Figure C-11 Debug status register C-35
Figure C-12 Debug control and status register structure C-37
Figure C-13 Vector catch register C-38

 Copyright © 2000 ARM Limited. All rights reserved.

Preface

This preface introduces the ARM9E-S and its reference documentation. It contains the following sections:

- *About this document* on page xiv
- *Further reading* on page xvii
- *Feedback* on page xviii.

 Copyright © 2000 ARM Limited. All rights reserved.

About this document

This document is the technical reference manual for the ARM9E-S.

Intended audience

This document has been written for hardware and software engineers who want to design or develop products based upon the ARM9E-S family of processors. It assumes no prior knowledge of ARM products.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM9E-S, and for a summary of the ARM9E-S instruction set.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the programmer's model for the ARM9E-S.

Chapter 3 *Device Reset*

Read this chapter for a description of the reset behavior of the ARM9E-S.

Chapter 4 *Memory Interface*

Read this chapter for a description of the memory interface, including descriptions of the instruction and data interfaces.

Chapter 5 *Interrupts*

Read this chapter for a description of interrupt operation. The chapter includes interrupt latency details.

Chapter 6 *Coprocessor Interface*

Read this chapter for a description of the coprocessor interface. The chapter includes timing diagrams for coprocessor operations.

Chapter 7 *Debug Interface and EmbeddedICE-RT*

Read this chapter for an overview of the debug interface and the EmbeddedICE-RT logic.

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 8 *Instruction Cycle Times*

Read this chapter for a summary of instruction cycle timings and a description of interlocks.

Chapter 9 *AC Parameters*

Read this chapter for a description of the AC timing parameters of the ARM9E-S.

Appendix A Signal Descriptions

Read this chapter for a description of all the ARM9E-S interface signals.

Appendix B *Differences*

Read this chapter for a description of the differences between the ARM9E-S and the ARM9TDMI hard macrocell interface.

Appendix C *Debug in depth*

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

Typographical conventions

The following typographical conventions are used in this book:

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

italic Highlights special terminology, cross-references, and citations.

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

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

`typewriter italic`

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

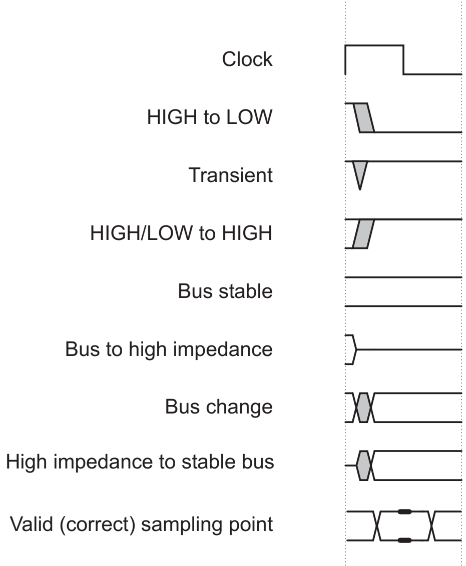
`typewriter bold`

Denotes language keywords when used outside example code.

Copyright © 2000 ARM Limited. All rights reserved.

Timing diagram conventions

This manual contains a number of timing diagrams. The following key explains the components used in these diagrams. Any variations are clearly labeled when they occur. Therefore, you must not attach any additional meaning unless specifically stated.



Key to timing diagram conventions

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

 Copyright © 2000 ARM Limited. All rights reserved.

Further reading

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

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

ARM publications

This document contains information that is specific to the ARM9E-S. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM9TDMI Data Sheet* (ARM DDI 0029)
- *ARM Software Development Kit User Guide* (ARM DUI 0040).

Other publications

This section lists relevant documents published by third parties.

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

Copyright © 2000 ARM Limited. All rights reserved.

Feedback

ARM Limited welcomes feedback both on the ARM9E-S, and on the documentation.

Feedback on the ARM9E-S

If you have any comments or suggestions about this product, please contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on the ARM9E-S Technical Reference Manual

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

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

General suggestions for additions and improvements are also welcome.

 Copyright © 2000 ARM Limited. All rights reserved.

Chapter 1
Introduction

This chapter introduces the ARM9E-S. It contains the following sections:

- *About the ARM9E-S* on page 1-2
- *ARM9E-S architecture* on page 1-5
- *ARM9E-S block, core, and interface diagrams* on page 1-7
- *ARM9E-S instruction set summary* on page 1-10.

Copyright © 2000 ARM Limited. All rights reserved.

1.1 About the ARM9E-S

The ARM9E-S is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family offers high performance for very low power consumption and gate count.

The ARM architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The reduced instruction set and related decode mechanism are much simpler than those of *Complex Instruction Set Computer* (CISC) designs. This simplicity gives:

- a high instruction throughput
- an excellent real-time interrupt response
- a small, cost-effective, processor macrocell.

The ARM9E-S supports the ARMv5TE architecture and features an enhanced multiplier design for improved DSP performance.

The ARM9E-S supports the ARM debug architecture and features support for real-time debug, which allows critical exception handlers to execute while debugging the system.

1.1.1 The instruction pipeline

The ARM9E-S uses a pipeline to increase the speed of the flow of instructions to the processor. This allows several operations to take place simultaneously, and the processing and memory systems to operate continuously.

A five-stage pipeline is used, consisting of Fetch, Decode, Execute, Memory, and Writeback stages. This is shown in Figure 1-1 on page 1-3.

 Copyright © 2000 ARM Limited. All rights reserved.

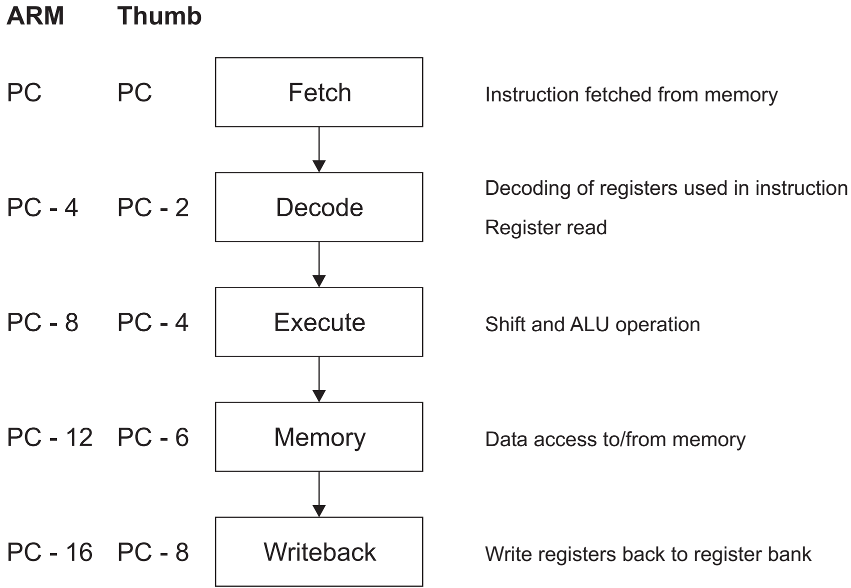


Figure 1-1 Five-stage pipeline

Note

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

During normal operation:

- one instruction is being fetched from memory
- the previous instruction is being decoded
- the instruction before that is being executed
- the instruction before that is performing data accesses (if applicable)
- the instruction before that is writing its data back to the register bank.

Copyright © 2000 ARM Limited. All rights reserved.

Typical pipeline operation is shown in Figure 1-2.

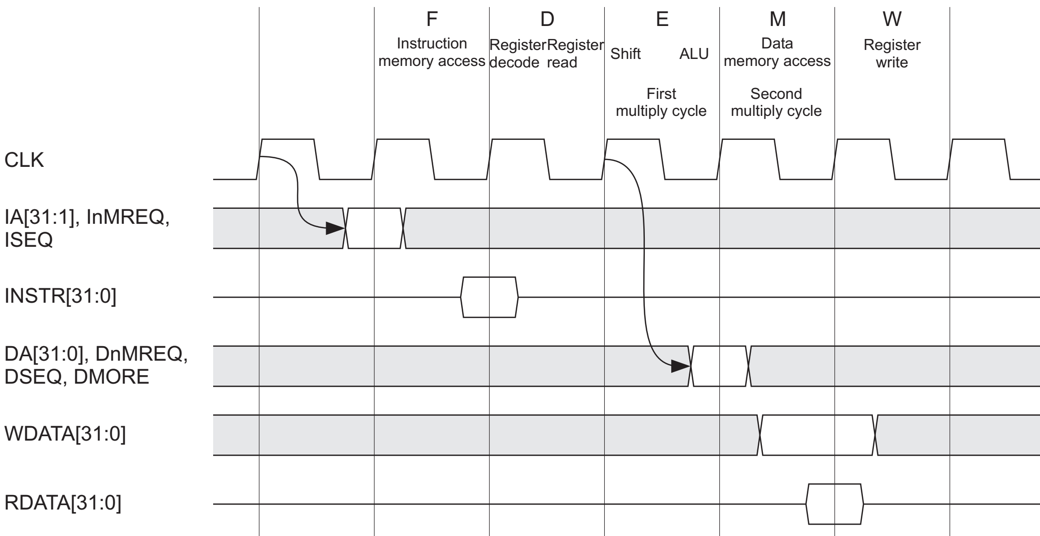


Figure 1-2 The instruction pipeline

1.1.2 Memory access

The ARM9E-S has a Harvard architecture. This features separate address and data buses for both the 32-bit instruction interface and the 32-bit data interface. This achieves a significant decrease in *Cycles Per Instruction* (CPI) by allowing instruction and data accesses to run concurrently.

Only load, store, coprocessor load, coprocessor store, and swap instructions can access data from memory. Data can be 8-bit bytes, 16-bit halfwords or 32-bit words. Words must be aligned to 4-byte boundaries. Halfwords must be aligned to 2-byte boundaries.

1.1.3 Forwarding, interlocking and data dependencies

Due to the nature of the five-stage pipeline, it is possible for a value to be required for use before it has been placed in the register bank by the actions of an earlier instruction. The ARM9E-S control logic automatically detects these cases and stalls the core or forwards data as applicable to overcome these hazards. No intervention is required by software in these cases, although you can improve software performance by re-ordering instructions in certain situations.

Copyright © 2000 ARM Limited. All rights reserved.

1.2 ARM9E-S architecture

The ARM9E-S processor has two instruction sets:

- the 32-bit ARM instruction set used in ARM state
- the 16-bit Thumb instruction set used in Thumb state.

The ARM9E-S is an implementation of the ARMv5TE architecture. For details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*. For full details of the ARM9E-S instruction set, contact ARM at `www.arm.com`.

1.2.1 Instruction compression

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

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

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

The ARM9E-S gives you the choice of running in ARM state, or Thumb state, or a mix of the two. This allows you to optimize both code density and performance to best suit your application requirements.

1.2.2 The Thumb instruction set

The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb instructions operate with the standard ARM register configuration, allowing excellent interoperability between ARM and Thumb states.

Thumb has all the advantages of a 32-bit core:

- 32-bit address space
- 32-bit registers
- 32-bit shifter and *Arithmetic Logic Unit* (ALU)
- 32-bit memory transfer.

Copyright © 2000 ARM Limited. All rights reserved.

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

Thumb code is typically 65% of the size of the ARM code, and provides 160% of the performance of ARM code when running on a processor connected to a 16-bit memory system. Thumb, therefore, makes the ARM9E-S ideally suited to embedded applications with restricted memory bandwidth, where code density is important.

The availability of both 16-bit Thumb and 32-bit ARM instruction sets, gives designers the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications. For example, critical loops for applications such as fast interrupts and DSP algorithms can be coded using the full ARM instruction set, and linked with Thumb code.

 Copyright © 2000 ARM Limited. All rights reserved.

1.3 ARM9E-S block, core, and interface diagrams

The ARM9E-S architecture, core, and interface diagrams are shown in the following figures:

- the *ARM9E-S block diagram* is shown in Figure 1-3
- the *ARM9E-S core diagram* is shown in Figure 1-4 on page 1-8
- the *ARM9E-S interface diagram* is shown in Figure 1-5 on page 1-9.

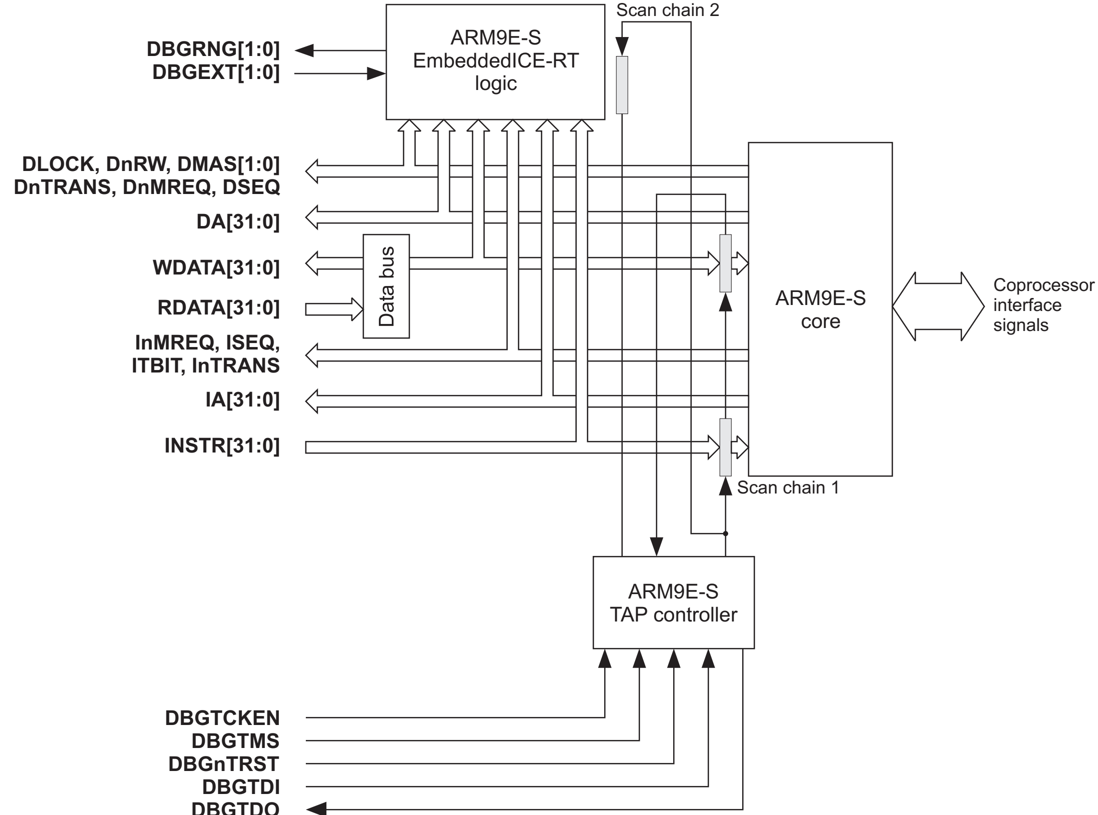


Figure 1-3 ARM9E-S block diagram

Refer to Chapter 7 *Debug Interface and EmbeddedICE-RT* for a description of the EmbeddedICE-RT logic.

 Copyright © 2000 ARM Limited. All rights reserved.

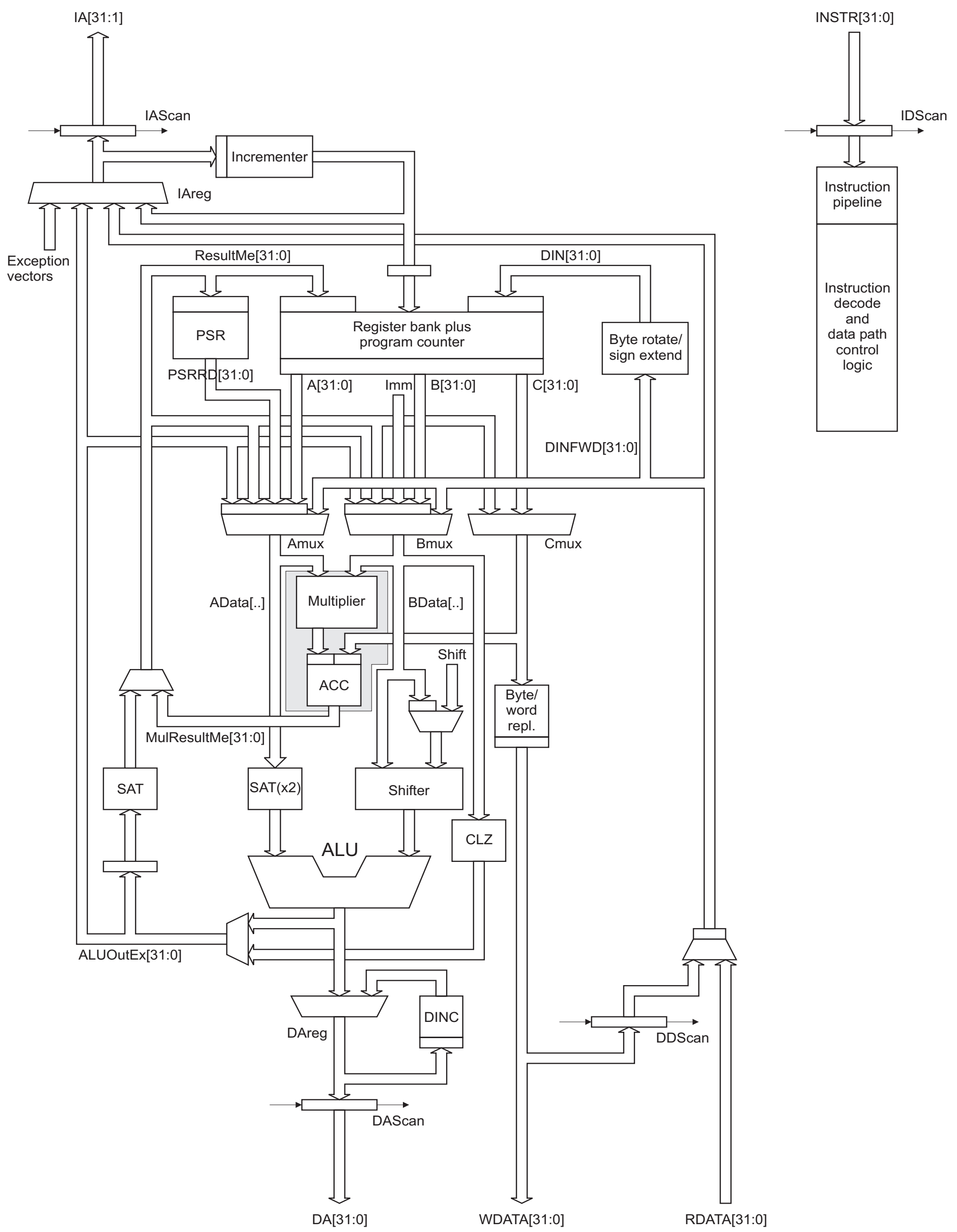


Figure 1-4 ARM9E-S core diagram

Copyright © 2000 ARM Limited. All rights reserved.

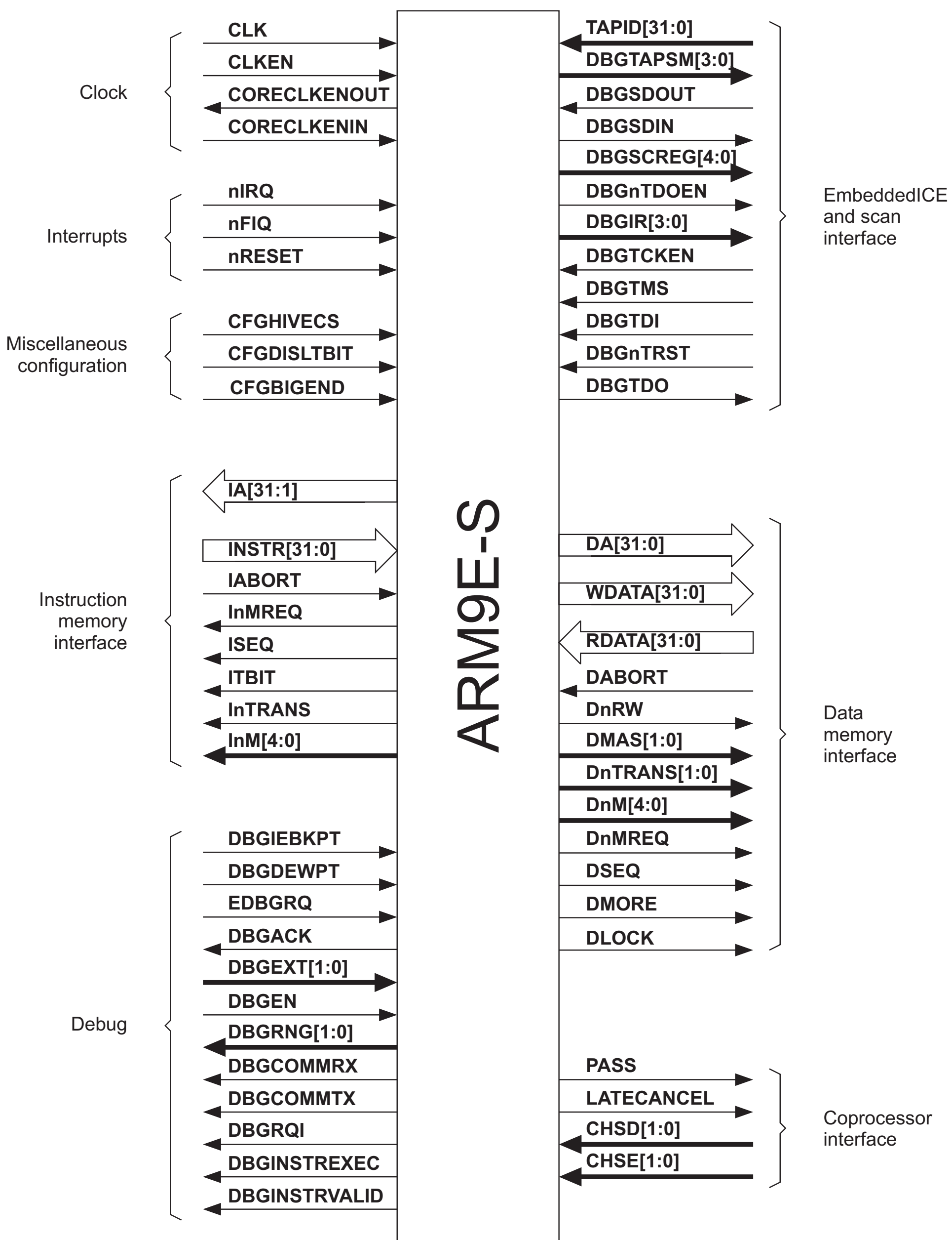


Figure 1-5 ARM9E-S interface diagram

Copyright © 2000 ARM Limited. All rights reserved.

1.4 ARM9E-S instruction set summary

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

- *ARM instruction set summary* on page 1-12
- *Thumb instruction set summary* on page 1-21.

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

The ARM9E-S is an implementation of the ARMv5TE architecture. For a description of both instruction sets, refer to the *ARM Architecture Reference Manual*. Contact ARM for complete descriptions of both instruction sets.

Table 1-1 Key to tables

Symbol	Description
{cond}	See Table 1-11 on page 1-20.
<Oprnd2>	See Table 1-9 on page 1-19.
{field}	See Table 1-10 on page 1-20.
S	Sets condition codes (optional).
B	Byte operation (optional).
H	Halfword operation (optional).
T	Forces **DnTRANS** to be active (0). Cannot be used with pre-indexed addresses.
<a_mode2>	See Table 1-3 on page 1-16.
<a_mode2P>	See Table 1-4 on page 1-17.
<a_mode3>	See Table 1-5 on page 1-18.
<a_mode4L>	See Table 1-6 on page 1-18.
<a_mode4S>	See Table 1-7 on page 1-18.
<a_mode5>	See Table 1-8 on page 1-19.
#32bit_Imm	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.

 Copyright © 2000 ARM Limited. All rights reserved.

Table 1-1 Key to tables (continued)

Symbol	Description
<reglist>	A comma-separated list of registers, enclosed in braces ({ and }).
x	Selects HIGH or LOW 16 bits of register Rm. T selects the HIGH 16 bits. (T = top) B selects the LOW 16 bits. (B = bottom).
y	Selects HIGH or LOW 16 bits of register Rs. T selects the HIGH 16 bits. (T = top) B selects the LOW 16 bits. (B = bottom).

Copyright © 2000 ARM Limited. All rights reserved.

1.4.1 ARM instruction set summary

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

Table 1-2 ARM instruction set summary

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

Copyright © 2000 ARM Limited. All rights reserved.

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
	Saturating add with double	QDADD{cond} Rd, Rn, Rs
	Saturating subtract	QSUB{cond} Rd, Rn, Rs
	Saturating subtract with double	QDSUB{cond} Rd, Rn, Rs
	Multiply 16x16	SMULxy{cond} Rd, Rm, Rs
	Multiply accumulate 16x16+32	SMULAxy{cond} Rd, Rm, Rs, Rn
	Multiply 32x16	SMULWx{cond} Rd, Rm, Rs
	Multiply accumulate 32x16+32	SMLAWx{cond} Rd, Rm, Rs, Rn
	Multiply signed accumulate long 16x16+64	SMLALx{cond} RdLo, RdHi, Rm, Rs
	Count leading zeros	CLZ{cond} Rd, Rm
Logical	Test	TST{cond} Rn, <Oprnd2>
	Test equivalence	TEQ{cond} Rn, <Oprnd2>
	AND	AND{cond}{S} Rd, Rn, <Oprnd2>
	XOR	EOR{cond}{S} Rd, Rn, <Oprnd2>
	OR	ORR{cond}{S} Rd, Rn, <Oprnd2>
	Bit clear	BIC{cond}{S} Rd, Rn, <Oprnd2>
Branch	Branch	B{cond} label
	Branch with link	BL{cond} label
	Branch and exchange	BX{cond} Rn
	Branch, link and exchange	BLX{cond} label
	Branch, link and exchange	BLX{cond} Rn
Load	Word	LDR{cond} Rd, <a_mode2>
	Word with User mode privilege	LDR{cond}T Rd, <a_mode2P>
	Byte	LDR{cond}B Rd, <a_mode2>
	Byte with User mode privilege	LDR{cond}BT Rd, <a_mode2P>

 Copyright © 2000 ARM Limited. All rights reserved.

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
	Byte signed	LDR{cond}SB Rd, <a_mode3>
	Halfword	LDR{cond}H Rd, <a_mode3>
	Halfword signed	LDR{cond}SH Rd, <a_mode3>
Multiple block data operations	Stack operations	LDM{cond}<a_mode4L> Rd{!}, <reglist>
	Increment before	LDM{cond}IB Rd{!}, <reglist>{^}
	Increment after	LDM{cond}IA Rd{!}, <reglist>{^}
	Decrement before	LDM{cond}DB Rd{!}, <reglist>{^}
	Decrement after	LDM{cond}DA Rd{!}, <reglist>{^}
	Stack operations and restore CPSR	LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^
	User registers	LDM{cond}<a_mode4L> Rd{!}, <reglist>^
	Load double	LDR{cond}D Rd, <a_mode3>
Store	Word	STR{cond} Rd, <a_mode2>
	Word with User mode privilege	STR{cond}T Rd, <a_mode2P>
	Byte	STR{cond}B Rd, <a_mode2>
	Byte with User mode privilege	STR{cond}BT Rd, <a_mode2P>
	Halfword	STR{cond}H Rd, <a_mode3>
Multiple block data operations	Stack operations	STM{cond}<a_mode4S> Rd{!}, <reglist>
	Increment before	STM{cond}IB Rd{!}, <reglist>{^}
	Increment after	STM{cond}IA Rd{!}, <reglist>{^}
	Decrement before	STM{cond}DB Rd{!}, <reglist>{^}
	Decrement after	STM{cond}DA Rd{!}, <reglist>{^}
	User registers	STM{cond}<a_mode4S> Rd{!}, <reglist>^
	Store double	STR{cond}D Rd, <a_mode3>
Cache hint	Prefetch DCache line	PLD <a_mode2>

Copyright © 2000 ARM Limited. All rights reserved.

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`
	Byte	`SWP{cond}B Rd, Rm, [Rn]`
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move double to ARM reg from coproc	`MRRC{cond} p<cpnum>, <op1>, Rd, Rn, CRm`
	Move double to coproc from ARM reg	`MCRR{cond} p<cpnum>, <op1>, Rd, Rn, CRm`
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode5>`
	Store	`STC{cond} p<cpnum>, CRd, <a_mode5>`
Software interrupt		`SWI{cond} 24bit_Imm`
Software breakpoint		`BKPT<immediate>`

Copyright © 2000 ARM Limited. All rights reserved.

Addressing mode 2 is summarized in Table 1-3.

Table 1-3 Addressing mode 2

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

Copyright © 2000 ARM Limited. All rights reserved.

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

Table 1-4 Addressing mode 2 (privileged)

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

 Copyright © 2000 ARM Limited. All rights reserved.

Addressing mode 3 is summarized in Table 1-5.

Table 1-5 Addressing mode 3

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

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

Table 1-6 Addressing mode 4 (load)

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

Addressing mode 4 (store) is summarized in Table 1-7.

Table 1-7 Addressing mode 4 (store)

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

Copyright © 2000 ARM Limited. All rights reserved.

Addressing mode 5 (load) is summarized in Table 1-8.

Table 1-8 Addressing mode 5 (load)

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

Oprnd2 is summarized in Table 1-9.

Table 1-9 Oprnd2

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

Copyright © 2000 ARM Limited. All rights reserved.

Fields are summarized in Table 1-10.

Table 1-10 Fields

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

Condition fields are summarized in Table 1-11.

Table 1-11 Condition fields

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

 Copyright © 2000 ARM Limited. All rights reserved.

1.4.2 Thumb instruction set summary

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

Table 1-12 Thumb instruction set summary

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

Copyright © 2000 ARM Limited. All rights reserved.

Table 1-12 Thumb instruction set summary (continued)

Operation		Assembler
	Compare Immediate	`CMP Rd, #8bit_Imm`
Logical	AND	`AND Rd, Rs`
	XOR	`EOR Rd, Rs`
	OR	`ORR Rd, Rs`
	Bit clear	`BIC Rd, Rs`
	Move NOT	`MVN Rd, Rs`
	Test bits	`TST Rd, Rs`
Shift/Rotate	Logical shift left	`LSL Rd, Rs, #5bit_shift_imm` `LSL Rd, Rs`
	Logical shift right	`LSR Rd, Rs, #5bit_shift_imm` `LSR Rd, Rs`
	Arithmetic shift right	`ASR Rd, Rs, #5bit_shift_imm` `ASR Rd, Rs`
	Rotate right	`ROR Rd, Rs`
Branch	Conditional	-
	If Z set	`BEQ label`
	If Z clear	`BNE label`
	If C set	`BCS label`
	If C clear	`BCC label`
	If N set	`BMI label`
	If N clear	`BPL label`
	If V set	`BVS label`
	If V clear	`BVC label`
	If C set and Z clear	`BHI label`
	If C clear or Z set	`BLS label`
	If N set and V set, or If N clear and V clear	`BGE label`

Copyright © 2000 ARM Limited. All rights reserved.

Table 1-12 Thumb instruction set summary (continued)

Operation		Assembler
	If N set and V clear, or If N clear and V set	`BLT label`
	If Z clear, and N and V set, or If Z clear, and N and V clear	`BGT label`
	If Z set, or N set and V clear, or N clear and V set	`BLE label`
	Unconditional	`B label`
	Long branch with link	`BL label`
	Long branch, link and exchange instruction	`BLX label`
Branch and exchange	To address held in Low reg	`BX Rs`
	To address held in High reg	`BX Hs`
Branch, link and exchange	To address held in Low reg	`BLX Rs`
	To address held in High reg	`BLX Hs`
Load	With immediate offset	-
	Word	`LDR Rd, [Rb, #7bit_offset]`
	Halfword	`LDRH Rd, [Rb, #6bit_offset]`
	Byte	`LDRB Rd, [Rb, #5bit_offset]`
	With register offset	-
	Word	`LDR Rd, [Rb, Ro]`
	Halfword	`LDRH Rd, [Rb, Ro]`
	Halfword signed	`LDRSH Rd, [Rb, Ro]`
	Byte	`LDRB Rd, [Rb, Ro]`
	Byte signed	`LDRSB Rd, [Rb, Ro]`
	PC-relative	`LDR Rd, [PC, #10bit_Offset]`
	SP-relative	`LDR Rd, [SP, #10bit_Offset]`

Copyright © 2000 ARM Limited. All rights reserved.

Table 1-12 Thumb instruction set summary (continued)

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

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 2
Programmer's Model

This chapter describes the ARM9E-S programmer's model. It contains the following sections:

- *About the programmer's model* on page 2-2
- *Processor operating states* on page 2-3
- *Memory formats* on page 2-4
- *Instruction length* on page 2-6
- *Data types* on page 2-7
- *Operating modes* on page 2-8
- *Registers* on page 2-9
- *The program status registers* on page 2-16
- *Exceptions* on page 2-20.

 Copyright © 2000 ARM Limited. All rights reserved.

2.1 About the programmer's model

The ARM9E-S processor core implements ARMv5TE architecture. This includes the 32-bit ARM instruction set and the 16-bit Thumb instruction set. For details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*.

The ARM9E-S programmer's model is described in:

- *Processor operating states* on page 2-3
- *Memory formats* on page 2-4
- *Instruction length* on page 2-6
- *Data types* on page 2-7
- *Operating modes* on page 2-8
- *Registers* on page 2-9
- *The program status registers* on page 2-16
- *Exceptions* on page 2-20.

 Copyright © 2000 ARM Limited. All rights reserved.

2.2 Processor operating states

The ARM9E-S has two operating states:

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

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

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

Note

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

2.2.1 Switching state

You can switch the operating state of the ARM9E-S core between ARM state and Thumb state using the `BX` and `BLX` instructions, and loads to the PC. Switching state is described in the *ARM Architecture Reference Manual*. For full details of the ARM9E-S instruction set, contact ARM.

All exceptions are entered, handled, and exited in ARM state. If an exception occurs in Thumb state, the processor reverts to ARM state. The transition back to Thumb state occurs automatically on return from the exception handler.

2.2.2 Interworking ARM and Thumb state

The ARM9E-S allows you to mix ARM and Thumb code as you wish. For details see *Chapter 7 Interworking ARM and Thumb* in the *Software Development Kit User Guide*.

 Copyright © 2000 ARM Limited. All rights reserved.

2.3 Memory formats

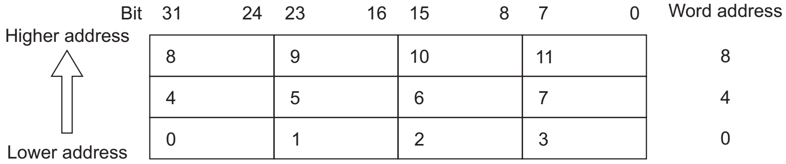
The ARM9E-S views memory as a linear collection of bytes numbered in ascending order from zero. Bytes 0 to 3 hold the first stored word, and bytes 4 to 7 hold the second stored word, for example.

The ARM9E-S can treat words in memory as being stored in either:

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

2.3.1 Big-endian format

In big-endian format, the ARM9E-S stores the most significant byte of a word at the lowest-numbered byte, and the least significant byte at the highest-numbered byte. Therefore, byte 0 of the memory system connects to data lines 31 to 24. This is shown in Figure 2-1.



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

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

2.3.2 Little-endian format

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

Copyright © 2000 ARM Limited. All rights reserved.

Bit	31 24	23 16	15 8	7 0	Word address
Higher address	11	10	9	8	8
↑	7	6	5	4	4
Lower address	3	2	1	0	0

- Least significant byte is at lowest address
- Word is addressed by byte address of least significant byte

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

 Copyright © 2000 ARM Limited. All rights reserved.

2.4 Instruction length

Instructions are either:

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

Copyright © 2000 ARM Limited. All rights reserved.

2.5 Data types

The ARM9E-S supports the following data types:

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

You must align these as follows:

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

Copyright © 2000 ARM Limited. All rights reserved.

2.6 Operating modes

The ARM9E-S has seven modes of operation:

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

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

 Copyright © 2000 ARM Limited. All rights reserved.

2.7 Registers

The ARM9E-S has a total of 37 registers:

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

These registers are not all accessible at the same time. The processor state and operating mode determine which registers are available to the programmer.

2.7.1 The ARM state register set

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

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

Link register Register r14 is used as the subroutine *Link Register* (LR).

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

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

Program counter Register r15 holds the PC.

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

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

 Copyright © 2000 ARM Limited. All rights reserved.

Banked registers have a mode identifier that indicates which User mode register they are mapped to. These mode identifiers are shown in Table 2-1.

Table 2-1 Register mode identifiers

Mode	Mode identifier
User	usr[a]
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usra
Undefined	und

a. The usr identifier is usually omitted from register names. It is only used in descriptions where the User or System mode register is specifically accessed from another operating mode.

FIQ mode has seven banked registers mapped to r8–r14 (r8_fiq–r14_fiq). As a result many FIQ handlers do not need to save any registers.

The Supervisor, Abort, IRQ, and Undefined modes each have alternative mode-specific registers mapped to r13 and r14, allowing a private stack pointer and link register for each mode.

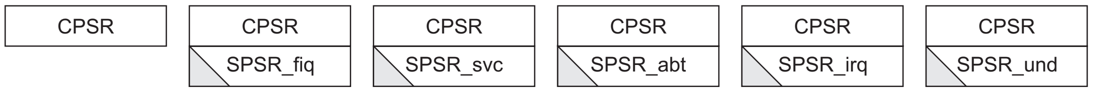
Copyright © 2000 ARM Limited. All rights reserved.

Figure 2-3 shows the ARM state registers.

ARM state general registers and program counter

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

ARM state program status registers



Indicates that the normal register used by the User or System mode has been replaced by an alternative register specific to the exception mode.

Figure 2-3 Register organization in ARM state

Copyright © 2000 ARM Limited. All rights reserved.

2.7.2 The Thumb state register set

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

- eight general registers, r0–r7 (for details of high register access in Thumb state see *Accessing high registers in Thumb state* on page 2-15).
- the PC
- a stack pointer, SP (ARM r13)
- an LR (ARM r14)
- the CPSR.

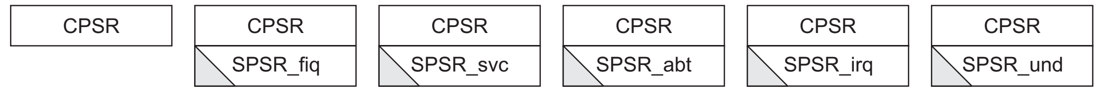
There are banked SPs, LRs, and SPSRs for each privileged mode. This register set is shown in Figure 2-4 on page 2-13.

Copyright © 2000 ARM Limited. All rights reserved.

Thumb state general registers and program counter

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

Thumb state program status registers



Indicates that the normal register used by the User or System mode has been replaced by an alternative register specific to the exception mode.

Figure 2-4 Register organization in Thumb state

Copyright © 2000 ARM Limited. All rights reserved.

2.7.3 The relationship between ARM state and Thumb state registers

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

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

These relationships are shown in Figure 2-5.

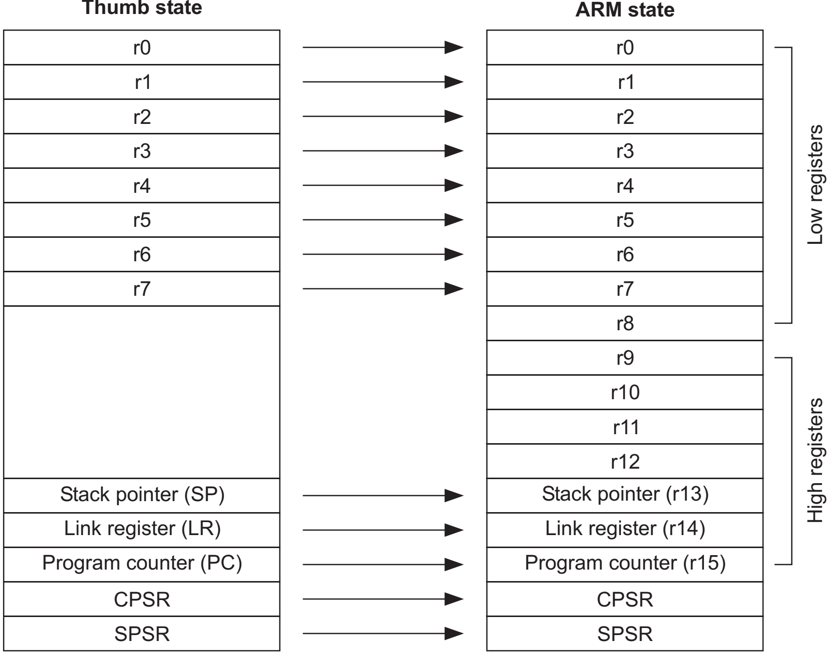


Figure 2-5 Mapping of Thumb state registers onto ARM state registers

Note

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

 Copyright © 2000 ARM Limited. All rights reserved.

2.7.4 Accessing high registers in Thumb state

In Thumb state, the high registers (r8–r15) are not part of the standard register set. With assembly language programming you have limited access to them, but can use them for fast temporary storage.

You can use special variants of the `MOV` instruction to transfer a value from a low register (in the range r0–r7) to a high register, and from a high register to a low register. The `CMP` instruction allows you to compare high register values with low register values. The `ADD` instruction allows you to add high register values to low register values. For more details, refer to the *ARM Architecture Reference Manual*.

Copyright © 2000 ARM Limited. All rights reserved.

2.8 The program status registers

The ARM9E-S contains a CPSR, and five SPSRs for exception handlers to use. The program status registers:

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

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

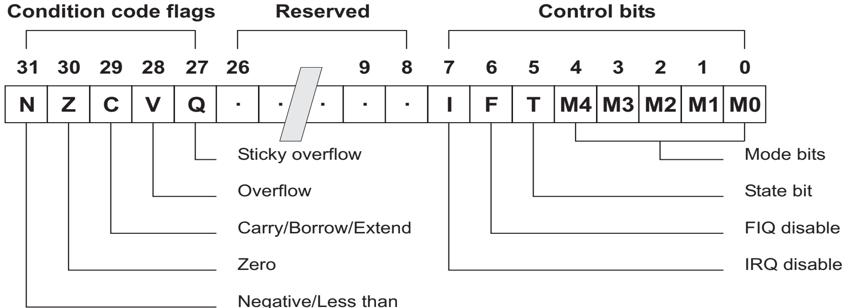


Figure 2-6 Program status register

Note

The unused bits of the status registers might be used in future ARM architectures, and must not be modified by software. The unused bits of the status registers are readable, to allow the processor state to be preserved (for example, during process context switches) and writable, to allow the processor state to be restored. To maintain compatibility with future ARM processors, and as good practice, you are strongly advised to use a read-modify-write strategy when changing the CPSR.

2.8.1 The condition code flags

The N, Z, C, and V bits are the condition code flags. They can be set by arithmetic and logical operations, and also by MSR and LDM instructions. The ARM9E-S tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally on the state of the N, Z, C, and V bits in ARM state. In Thumb state, only the Branch instruction can be executed conditionally. For more information about conditional execution, refer to the *ARM Architecture Reference Manual*.

 Copyright © 2000 ARM Limited. All rights reserved.

The Q flag

The Sticky Overflow (Q) flag can be set by certain multiply and fractional arithmetic instructions:

- QADD
- QDADD
- QSUB
- QDSUB
- SMLAxy
- SMLAWy

The Q flag is *sticky* in that, once set by an instruction, it remains set until explicitly cleared by an `MSR` instruction writing to CPSR. Instructions cannot execute conditionally on the status of the Q flag. To determine the status of the Q flag you must read the PSR into a register and extract the Q flag from this. For details of how the Q flag is set and cleared, see individual instruction definitions in the *ARM Architectural Reference Manual*.

2.8.2 The control bits

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

- *Interrupt disable bits*
- *T bit*
- *Mode bits* on page 2-18.

The control bits change when an exception occurs. When the processor is operating in a privileged mode, software can manipulate these bits.

Interrupt disable bits

The I and F bits are the interrupt disable bits:

- when the I bit is set, IRQ interrupts are disabled
- when the F bit is set, FIQ interrupts are disabled.

T bit

Caution

Never use an `MSR` instruction to force a change to the state of the T bit in the CPSR. If you do this, the processor enters an unpredictable state.

 Copyright © 2000 ARM Limited. All rights reserved.

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor is executing in ARM state.

The operating state is reflected by the **ITBIT** external signal.

Mode bits

The M4, M3, M2, M1, and M0 bits (M[4:0]) are the mode bits. These bits determine the processor operating mode as shown in Table 2-2.

Caution

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

Not all combinations of the mode bits define a valid processor mode, so take care to use only those bit combinations shown.

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10000	User	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR
10001	FIQ	r0–r7, r8_fiq-r12_fiq[a], SP_fiq, LR_fiq PC, CPSR, SPSR_fiq	r0–r7, r8_fiq–r14_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	r0–r7, r8-r12[a], SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0–r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
10011	Supervisor	r0–r7, r8-r12[a], SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0–r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc
10111	Abort	r0–r7, r8-r12[a], SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0–r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
11011	Undefined	r0–r7, r8-r12[a], SP_und, LR_und, PC, CPSR, SPSR_und	r0–r12, r13_und, r14_und, PC, CPSR, SPSR_und
11111	System	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR

a. Access to these registers is limited in Thumb state.

Copyright © 2000 ARM Limited. All rights reserved.

2.8.3 Reserved bits

The remaining bits in the PSRs are unused, but are reserved. When changing a PSR flag or control bits, make sure that these reserved bits are not altered. You must ensure that your program does not rely on reserved bits containing specific values because future processors might use some or all of the reserved bits.

Copyright © 2000 ARM Limited. All rights reserved.

2.9 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the ARM9E-S preserves the current processor state so that the original program can resume when the handler routine has finished.

If two or more exceptions arise simultaneously, the exceptions are dealt with in the fixed order given in *Exception priorities* on page 2-27.

This section provides details of the ARM9E-S exception handling:

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

2.9.1 Exception entry and exit summary

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

Table 2-3 Exception entry and exit

Exception or entry	Return instruction	Previous state		Notes
		ARM r14_x	Thumb r14_x	
SWI	`MOVS PC, R14_svc`	PC + 4	PC+2	Where the PC is the address of the SWI, undefined instruction, or instruction that had the Prefetch Abort.
UNDEF	`MOVS PC, R14_und`	PC + 4	PC+2	
PABT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	
FIQ	`SUBS PC, R14_fiq, #4`	PC + 4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.
IRQ	`SUBS PC, R14_irq, #4`	PC + 4	PC+4	
DABT	`SUBS PC, R14_abt, #8`	PC + 8	PC+8	Where the PC is the address of the Load or Store instruction that generated the Data Abort.
RESET	`NA`	-	-	The value saved in r14_svc upon reset is `UNPREDICTABLE`.
BKPT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	Software breakpoint.

 Copyright © 2000 ARM Limited. All rights reserved.

2.9.2 Entering an exception

When handling an exception the ARM9E-S:

1. Preserves the address of the next instruction in the appropriate LR. When the exception entry is from:
   - ARM state, the ARM9E-S copies the address of the next instruction into the LR (current PC + 4 or PC + 8 depending on the exception).
   - Thumb state, the ARM9E-S writes the value of the PC into the LR, offset by a value (current PC + 4 or PC + 8 depending on the exception) that causes the program to resume from the correct place on return.

   The exception handler does not need to determine the state when entering an exception. For example, in the case of a SWI, `MOVS PC, r14_svc` always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.

2. Copies the CPSR into the appropriate SPSR.
3. Forces the CPSR mode bits to a value which depends on the exception.
4. Forces the PC to fetch the next instruction from the relevant exception vector.

The ARM9E-S can also set the interrupt disable flags to prevent otherwise unmanageable nesting of exceptions.

Note

Exceptions are always entered, handled, and exited in ARM state. When the processor is in Thumb state and an exception occurs, the switch to ARM state takes place automatically when the exception vector address is loaded into the PC.

2.9.3 Leaving an exception

When an exception has completed, the exception handler must move the LR, minus an offset to the PC. The offset varies according to the type of exception, as shown in Table 2-3 on page 2-20.

If the S bit is set and rd = r15, the core copies the SPSR back to the CPSR and clears the interrupt disable flags that were set on entry.

Note

The action of restoring the CPSR from the SPSR automatically resets the T bit to the value it held immediately prior to the exception. The I and F bits are automatically restored to the value they held immediately prior to the exception.

Copyright © 2000 ARM Limited. All rights reserved.

2.9.4 Reset

When the **nRESET** signal is driven LOW a reset occurs, and the ARM9E-S abandons the executing instruction.

When **nRESET** is driven HIGH again the ARM9E-S:

1. Forces CPSR[4:0] to b10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR T bit. Other bits in the CPSR are indeterminate.
2. Forces the PC to fetch the next instruction from the reset vector address.
3. Reverts to ARM state, and resumes execution.

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

Refer to Chapter 3 *Device Reset* for more details of the ARM9E-S reset behavior.

2.9.5 Fast interrupt request

The *Fast Interrupt Request* (FIQ) exception supports fast interrupts. In ARM state, FIQ mode has eight private registers to reduce, or even remove the requirement for register saving (minimizing the overhead of context switching).

An FIQ is externally generated by taking the **nFIQ** signal input LOW. The **nFIQ** input is registered internally to the ARM9E-S. It is the output of this register that is used by the ARM9E-S control logic.

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

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

You can disable FIQ exceptions within a privileged mode by setting the CPSR F flag. When the F flag is clear, the ARM9E-S checks for a LOW level on the output of the nFIQ register at the end of each instruction.

FIQs and IRQs are disabled when an FIQ occurs. Nested interrupts are allowed but it is up to the programmer to save any corruptible registers and to re-enable FIQs and interrupts.

2.9.6 Interrupt request

The *Interrupt Request* (IRQ) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. IRQ has a lower priority than FIQ, and is masked on entry to an FIQ sequence. You can disable IRQ at any time, by setting the I bit in the CPSR from a privileged mode.

Copyright © 2000 ARM Limited. All rights reserved.

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

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

You can disable IRQ exceptions within a privileged mode by setting the CPSR I flag. When the I flag is clear, the ARM9E-S checks for a LOW level on the output of the nIRQ register at the end of each instruction.

FIQs and IRQs are disabled when an IRQ occurs. Nested interrupts are allowed but it is up to you to save any corruptible registers and to re-enable FIQs and interrupts.

2.9.7 Aborts

An abort indicates that the current memory access cannot be completed. An abort is signaled by one of the two external abort input pins, **IABORT** and **DABORT**.

There are two types of abort:

- *Prefetch Abort*
- *Data Abort* on page 2-23.

IRQs are disabled when an abort occurs.

Prefetch Abort

This is signaled by an assertion on the **IABORT** input pin and checked at the end of each instruction fetch.

When a Prefetch Abort occurs, the ARM9E-S marks the prefetched instruction as invalid, but does not take the exception until the instruction reaches the Execute stage of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the abort does not take place.

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

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

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

Data Abort

This is signaled by an assertion on the **DABORT** input pin and checked at the end of each data access, both read and write.

Copyright © 2000 ARM Limited. All rights reserved.

The ARM9E-S implements the *base restored Data Abort model*, which differs from the *base updated Data Abort model* implemented by the ARM7TDMI-S.

The difference in the Data Abort model affects only a very small section of operating system code, in the Data Abort handler. It does not affect user code.

With the *base restored Data Abort model*, when a Data Abort exception occurs during the execution of a memory access instruction, the base register is always restored by the processor hardware to the value it contained *before* the instruction was executed. This removes the need for the Data Abort handler to *unwind* any base register update, which might have been specified by the aborted instruction. This greatly simplifies the software Data Abort handler.

The abort mechanism allows you to implement 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, and its state is not affected by the abort.

After dealing with the cause of the abort, the handler must execute the following return instruction irrespective of the processor operating state at the point of entry:

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

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

2.9.8 Software interrupt instruction

You can use the *Software Interrupt Instruction* (`SWI`) to enter Supervisor mode, usually to request a particular supervisor function. A SWI handler returns by executing the following instruction, irrespective of the processor operating state:

```
MOVS PC, R14_svc
```

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

IRQs are disabled when a software interrupt occurs.

Copyright © 2000 ARM Limited. All rights reserved.

2.9.9 Undefined instruction

When an instruction is encountered that neither the ARM9E-S, nor any coprocessor in the system can handle, the ARM9E-S takes the undefined instruction trap. Software can use this mechanism to extend the ARM instruction set by emulating undefined coprocessor instructions.

After emulating the failed instruction, the trap handler executes the following instruction, irrespective of the processor operating state:

```
MOVS PC,R14_und
```

This action restores the CPSR and returns to the next instruction after the undefined instruction.

IRQs are disabled when an undefined instruction trap occurs. For more information about undefined instructions, refer to the *ARM Architecture Reference Manual*.

2.9.10 Breakpoint instruction (BKPT)

A breakpoint (`BKPT`) instruction operates as though the instruction caused a Prefetch Abort.

A breakpoint instruction does not cause the ARM9E-S to take the Prefetch Abort exception until the instruction reaches the Execute stage of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the breakpoint does not take place.

After dealing with the breakpoint, the handler executes the following instruction irrespective of the processor operating state:

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

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

Note

If the EmbeddedICE-RT logic is configured into stopping mode, a breakpoint instruction causes the ARM9E-S to enter debug state. See *Debug control register* on page C-34.

 Copyright © 2000 ARM Limited. All rights reserved.

2.9.11 Exception vectors

You can configure the location of the exception vector addresses using the input **CFGHIVECS**, as shown in Table 2-4.

Table 2-4 Configuration of exception vector address locations

Value of CFGHIVECS	Exception vector base location
0	`0x0000 0000`
1	`0xFFFF 0000`

Table 2-5 shows the exception vector addresses and entry conditions for the different exception types.

Table 2-5 Exception vectors

Exception	Offset from vector base	Mode on entry	I bit on entry	F bit on entry
Reset	`0x00`	Supervisor	Disabled	Disabled
Undefined instruction	`0x04`	Undefined	Disabled	Unchanged
Software interrupt	`0x08`	Supervisor	Disabled	Unchanged
Abort (prefetch)	`0x0C`	Abort	Disabled	Unchanged
Abort (data)	`0x10`	Abort	Disabled	Unchanged
Reserved	`0x14`	Reserved	-	-
IRQ	`0x18`	IRQ	Disabled	Unchanged
FIQ	`0x1C`	FIQ	Disabled	Disabled

 Copyright © 2000 ARM Limited. All rights reserved.

2.9.12 Exception priorities

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

1. Reset (highest priority).
2. Data Abort.
3. FIQ.
4. IRQ.
5. Prefetch Abort.
6. BKPT, undefined instruction, and SWI (lowest priority).

Some exceptions cannot occur together:

- The BKPT, or undefined instruction, and SWI exceptions are mutually exclusive. Each corresponds to a particular (non-overlapping) decoding of the current instruction.
- When FIQs are enabled, and a Data Abort occurs at the same time as an FIQ, the ARM9E-S enters the Data Abort handler, and proceeds immediately to the FIQ vector.

  A normal return from the FIQ causes the Data Abort handler to resume execution.

  Data Aborts must have higher priority than FIQs to ensure that the transfer error does not escape detection. You must add the time for this exception entry to the worst-case FIQ latency calculations in a system that uses aborts to support virtual memory.

  The FIQ handler must not access any memory that can generate a Data Abort, because the initial Data Abort exception condition is lost.

 Copyright © 2000 ARM Limited. All rights reserved.

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 3
Device Reset

This chapter describes the ARM9E-S reset behavior. It contains the following sections:

- *About device reset* on page 3-2
- *Reset modes* on page 3-3
- *ARM9E-S behavior on exit from reset* on page 3-5.

Copyright © 2000 ARM Limited. All rights reserved.

3.1 About device reset

This section describes the ARM9E-S reset signals and how you must use them for correct operation of the device.

The ARM9E-S has two reset inputs:

nRESET The **nRESET** signal is the main CPU reset that initializes the majority of the ARM9E-S logic.

DBGnTRST The **DBGnTRST** signal is the debug logic reset that you can use to reset the ARM9E-S TAP controller and the EmbeddedICE-RT unit.

Both **nRESET** and **DBGnTRST** are active LOW signals that asynchronously reset logic in the ARM9E-S. You must take care when designing the logic to drive these reset signals.

Copyright © 2000 ARM Limited. All rights reserved.

3.2 Reset modes

Two reset signals are present in the ARM9E-S design to enable you to reset different parts of the design independently. A description of the reset signaling combinations and possible applications is shown in Table 3-1.

Table 3-1 Reset modes

Reset mode	nRESET	DBGnTRST	Application
Power-on reset	0	0	Reset at power up, full system reset.
CPU reset	0	1	Reset of CPU core only, watchdog reset.
EmbeddedICE-RT reset	1	0	Reset of EmbeddedICE-RT circuitry.
Normal	1	1	No reset. Normal run mode.

3.2.1 Power-on reset

You must apply power-on or *cold* reset to the ARM9E-S when power is first applied to the system. In the case of power-on reset, the leading (falling) edge of the reset signals (**nRESET** and **DBGnTRST**) does not have to be synchronous to **CLK**. The trailing (rising) edge of the reset signals must be set up and held about the rising edge of the clock. You must do this to ensure that the entire system leaves reset in a predictable manner. This is particularly important in multi-processor systems. Figure 3-1 shows the application of power-on reset.

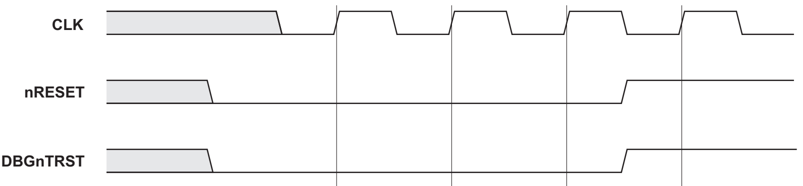


Figure 3-1 Power-on reset

It is recommended that you assert the reset signals for at least three **CLK** cycles to ensure correct reset behavior. Adopting a three-cycle reset eases the integration of other ARM parts into the system, for example, ARM9TDMI based designs.

Copyright © 2000 ARM Limited. All rights reserved.

3.2.2 CPU reset

A CPU or *warm* reset initializes the majority of the ARM9E-S CPU, excluding the ARM9E-S TAP controller and the EmbeddedICE-RT unit. CPU reset is typically used for resetting a system that has been operating for some time, for example, watchdog reset.

Sometimes you might not want to reset the EmbeddedICE-RT unit when resetting the rest of the ARM9E-S, for example, if EmbeddedICE-RT has been configured to breakpoint (or capture) fetches from the reset vector.

For CPU reset, both the leading and trailing edges of **nRESET** must be set up and held about the rising edge of **CLK**. This ensures that there are no metastability issues between the ARM9E-S and the EmbeddedICE-RT unit.

3.2.3 EmbeddedICE-RT reset

EmbeddedICE-RT reset initializes the state of the ARM9E-S TAP controller and the EmbeddedICE-RT unit. EmbeddedICE-RT reset is typically used by the Multi-ICE module for hot connection of a debugger to a system.

EmbeddedICE-RT reset allows initialization of the EmbeddedICE-RT unit without affecting the normal operation of the ARM9E-S.

For EmbeddedICE-RT reset, both the leading and trailing edges of **DBGnTRST** must be set up and held about the rising edge of **CLK**. This ensures that there are no metastability issues between the ARM9E-S and the EmbeddedICE-RT unit.

Refer to *Clocks and synchronization* on page 7-14 for more details of synchronization between the Multi-ICE and ARM9E-S.

3.2.4 Normal operation

During normal operation, neither CPU reset nor EmbeddedICE-RT reset is asserted.

Copyright © 2000 ARM Limited. All rights reserved.

3.3 ARM9E-S behavior on exit from reset

When **nRESET** is driven LOW, the currently executing instruction terminates abnormally. **InMREQ**, **ISEQ**, **DnMREQ**, **DSEQ**, and **DMORE** change asynchronously to indicate an internal cycle. When **nRESET** is driven HIGH, the ARM9E-S starts requesting instructions from memory again once the **nRESET** signal has been registered, and the first memory access starts two cycles later. The **nRESET** signal is sampled on the rising-edge of **CLK**.

The behavior of the memory interface coming out of reset is shown in Figure 3-2.

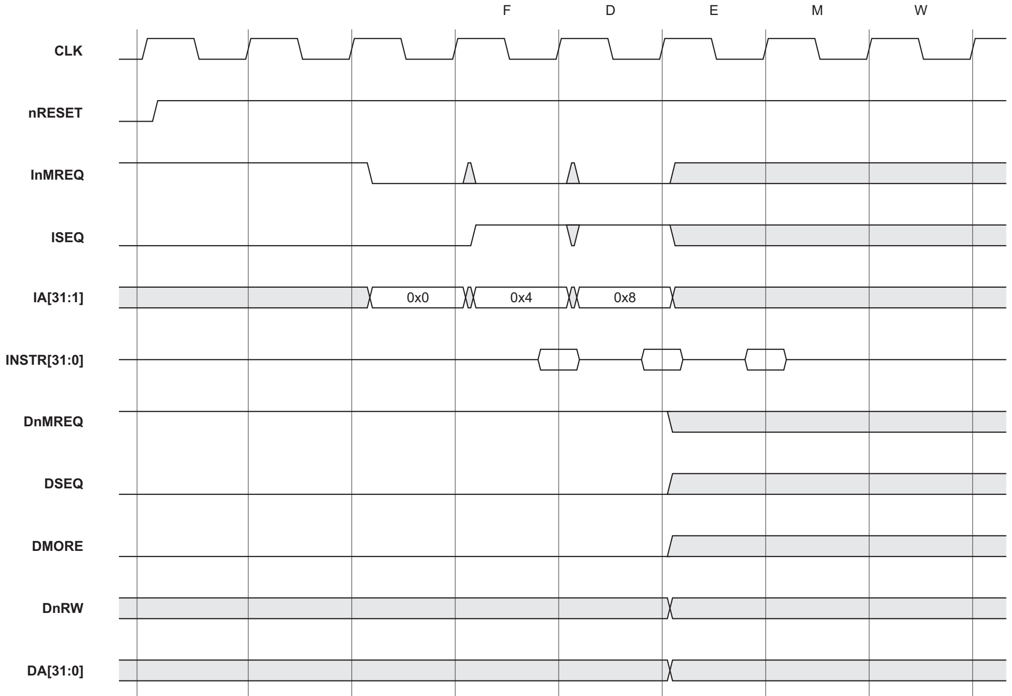


Figure 3-2 ARM9E-S behavior on exit from reset

 Copyright © 2000 ARM Limited. All rights reserved.

 Copyright © 2000 ARM Limited. All rights reserved.

Chapter 4
Memory Interface

This chapter describes the ARM9E-S memory interface. It contains the following sections:

- *About the memory interface* on page 4-2
- *Instruction interface* on page 4-3
- *Instruction interface addressing signals* on page 4-4
- *Instruction interface data timed signals* on page 4-6
- *Endian effects for instruction fetches* on page 4-7
- *Instruction interface cycle types* on page 4-8
- *Data interface* on page 4-13
- *Data interface addressing signals* on page 4-15
- *Data interface data timed signals* on page 4-18
- *Data interface cycle types* on page 4-24
- *Endian effects for data transfers* on page 4-30
- *Use of CLKEN to control bus cycles* on page 4-31.

 Copyright © 2000 ARM Limited. All rights reserved.

4.1 About the memory interface

The ARM9E-S has a Harvard bus architecture with separate instruction and data interfaces. This allows concurrent instruction and data accesses, and greatly reduces the *Cycles Per Instruction* (CPI) of the processor. For optimal performance, single-cycle memory accesses for both interfaces are required, although the core can be wait-stated for nonsequential accesses, or slower memory systems.

For both instruction and data interfaces, the ARM9E-S processor core uses pipelined addressing. This means that the address and control signals are generated the cycle before the data transfer takes place. All memory accesses are timed with the clock **CLK**.

For each interface there are different types of memory access:

- Nonsequential
- Sequential
- Internal
- Coprocessor transfer (for the data interface).

The ARM9E-S can operate in both big-endian and little-endian memory configurations and this is selected by the **CFGBIGEND** input. The endian configuration affects both interfaces, so you must take care when designing the memory interface logic to allow correct operation of the processor core.

For system programming purposes, you must normally provide some mechanism for the data interface to access instruction memory. There are two main reasons for this:

- The use of in-line data for literal pools is very common. This data is fetched using the data interface but is normally contained in the instruction memory space.
- To enable debug using the JTAG interface it must be possible to download code into the instruction memory. This code has to be written to memory through the data interface, because the instruction interface is read-only. In this case it is essential for the data interface to have access to the instruction memory.

A typical implementation of an ARM9E-S based cached processor has Harvard caches and a unified memory structure beyond the caches, therefore giving the data interface access to the instruction memory space. However, for an SRAM-based system, you cannot use this technique, and you must use an alternative method.

It is not necessary for the instruction interface to have access to the data memory area unless the processor needs to execute code from data memory.

 Copyright © 2000 ARM Limited. All rights reserved.

4.2 Instruction interface

The ARM9E-S requests instructions for execution using the instruction memory interface. A new instruction is fetched over the instruction bus whenever an instruction enters the Execute stage of the pipeline.

Instruction fetches take place in the Fetch stage of the pipeline.

4.2.1 Instruction interface signals

The signals in the ARM9E-S instruction interface can be grouped into four categories:

- Clocking and clock control signals:
  - **CLK**
  - **CLKEN**
  - **nRESET**.
- Address class signals:
  - **IA[31:1]**
  - **ITBIT**
  - **InTRANS**
  - **InM[4:0]**.
- Memory request signals:
  - **InMREQ**
  - **ISEQ**.
- Data timed signals:
  - **INSTR[31:0]**
  - **IABORT**.

Each of these signal groups shares a common timing relationship to the bus interface cycle. All signals in the ARM9E-S instruction interface are generated from, or sampled by, the rising edge of **CLK**.

You can extend bus cycles using the **CLKEN** signal (see *Use of CLKEN to control bus cycles* on page 4-31). Unless otherwise stated **CLKEN** is permanently HIGH.

Copyright © 2000 ARM Limited. All rights reserved.

4.3 Instruction interface addressing signals

The address class signals for the instruction memory interface are:

- *IA[31:1]*
- *ITBIT*
- *InTRANS* on page 4-5
- *InM[4:0]* on page 4-5.

4.3.1 IA[31:1]

IA[31:1] is the 31-bit address bus that specifies the address for the transfer. All addresses are byte addresses, so a burst of 32-bit instruction fetches results in the address bus incrementing by four for each cycle.

Note

The ARM9E-S does not produce **IA[0]** as all instruction accesses are halfword-aligned (that is, **IA[0]** = 0).

The address bus provides 4GB of linear addressing space. When a word access is signaled the memory system must ignore **IA[1]**.

4.3.2 ITBIT

The **ITBIT** signal encodes the size of the instruction fetch. The ARM9E-S can request word-sized instructions (when in ARM state) or halfword-sized instructions (when in Thumb state). This is encoded on **ITBIT** as shown in Table 4-1.

Table 4-1 Transfer widths

ITBIT	Transfer width
1	Halfword
0	Word

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

Copyright © 2000 ARM Limited. All rights reserved.

4.3.3 InTRANS

The **InTRANS** signal encodes information about the transfer. A memory management unit uses this signal to determine if an access is from a privileged mode. Therefore, you can use this signal to implement an access permission scheme. The encoding of **InTRANS** is shown in Table 4-2.

Table 4-2 InTRANS encoding

InTRANS	Mode
0	User
1	Privileged

4.3.4 InM[4:0]

InM[4:0] indicates the operating mode of the ARM9E-S. This bus corresponds to the bottom 5 bits of the CPSR, the outputs are inverted with respect to the CPSR.

Copyright © 2000 ARM Limited. All rights reserved.

4.4 Instruction interface data timed signals

The data timed signals for the instruction memory interface are:

- *INSTR[31:0]*
- *IABORT*.

4.4.1 INSTR[31:0]

INSTR[31:0] is the read data bus, and is used by the ARM9E-S to fetch opcodes. The **INSTR[31:0]** signal is sampled on the rising edge of **CLK** at the end of the bus cycle.

4.4.2 IABORT

IABORT indicates that an instruction fetch failed to complete successfully. **IABORT** is sampled at the end of the bus cycle during active memory cycles (S cycles and N cycles).

If **IABORT** is asserted on an opcode fetch, the abort is tracked down the pipeline, and the Prefetch Abort trap is taken if the instruction is executed.

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

For more details about aborts, see *Aborts* on page 2-23.

 Copyright © 2000 ARM Limited. All rights reserved.

4.5 Endian effects for instruction fetches

The ARM9E-S performs 32-bit or 16-bit instruction fetches depending on whether the processor is in ARM or Thumb state. The processor state can be determined externally by the value of the **ITBIT** signal. When this signal is LOW, the processor is in ARM state, and 32-bit instructions are fetched. When **ITBIT** is HIGH, the processor is in Thumb state and 16-bit instructions are fetched.

The address produced by the ARM9E-S is always halfword-aligned. However, the memory system must ignore bit 1of the address, depending on the size of the instruction request. The significant address bits are listed in Table 4-3.

Table 4-3 Significant address bits

ITBIT	Width	Significant address bits
1	Halfword	**IA[31:1]**
0	Word	**IA[31:2]**

When a halfword instruction fetch is performed, a 32-bit memory system can return the complete 32-bit word, and the ARM9E-S extracts the valid halfword field from it. The field extracted depends on the state of the **CFGBIGEND** signal, which determines the endianness of the system (see *Memory formats* on page 2-4).

The fields extracted by the ARM9E-S are shown in Table 4-4.

Table 4-4 32-bit instruction fetches

ITBIT	IA[1]	Little-endian CFGBIGEND = 0	Big-endian CFGBIGEND = 1
0	X	INSTR[31:0]	INSTR[31:0]

When connecting 8-bit or 16-bit memory systems to the ARM9E-S, ensure that the data is presented to the correct byte lanes on the ARM9E-S as shown in Table 4-5.

Table 4-5 Halfword accesses

ITBIT	IA[1]	Little-endian CFGBIGEND = 0	Big-endian CFGBIGEND = 1
1	0	INSTR[15:0]	INSTR[31:16]
1	1	INSTR[31:16]	INSTR[15:0]

Copyright © 2000 ARM Limited. All rights reserved.

4.6 Instruction interface cycle types

The ARM9E-S instruction interface is pipelined. The address class signals and the memory request signals are broadcast in the bus cycle ahead of the bus cycle to which they refer. This gives the maximum time for a memory cycle to decode the address, and respond to the access request.

A single memory cycle is shown in Figure 4-1.

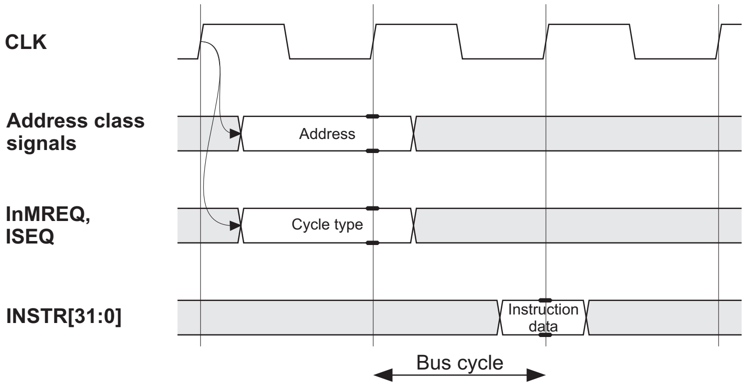


Figure 4-1 Simple memory cycle

The ARM9E-S instruction interface can perform three different types of memory cycle. These are indicated by the state of the **InMREQ and ISEQ** signals. Memory cycle types are encoded on the **InMREQ and ISEQ** signals as shown in Table 4-6.

Table 4-6 Cycle types

InMREQ	ISEQ	Cycle type	Description
0	0	N cycle	Nonsequential cycle
0	1	S cycle	Sequential cycle
1	0	I cycle	Internal cycle
1	1	-	Reserved

A memory controller for the ARM9E-S must commit to an instruction memory access only on an N cycle or an S cycle.

Copyright © 2000 ARM Limited. All rights reserved.

The ARM9E-S instruction interface has three types of memory cycle:

Nonsequential cycle
During this the ARM9E-S core requests a transfer to or from an address that is unrelated to the address used in the preceding cycle.

Sequential cycle
During this the ARM9E-S core requests a transfer to or from an address that is either one word, or one halfword greater than the address used in the preceding cycle.

Internal cycle
During this the ARM9E-S core does not require a transfer because it is performing an internal function, and no useful prefetching can be performed at the same time.

4.6.1 Instruction interface, nonsequential cycles

A nonsequential instruction fetch is the simplest form of an ARM9E-S instruction interface cycle, and occurs when the ARM9E-S requests a transfer to or from an address that is unrelated to the address used in the preceding cycle. The memory controller must initiate a memory access to satisfy this request.

The address class signals and the **InMREQ, ISEQ** = N cycle signals are broadcast on the instruction interface bus. At the end of the next bus cycle the instruction is transferred to the CPU from memory. This is shown in Figure 4-2.

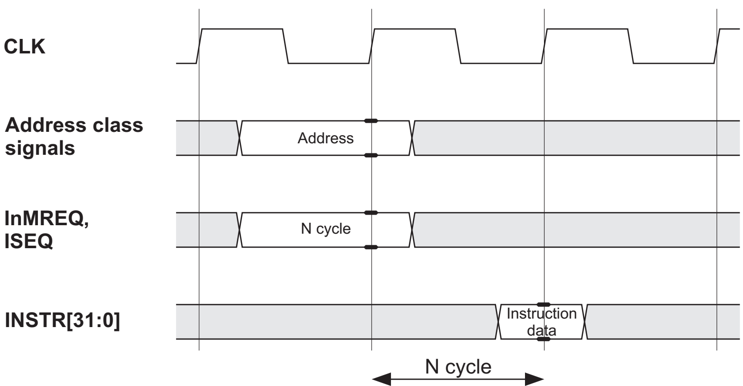


Figure 4-2 Nonsequential instruction fetch cycle

 Copyright © 2000 ARM Limited. All rights reserved.

4.6.2 Instruction interface, sequential cycles

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

During a sequential cycle, the ARM9E-S requests a memory location that is part of a sequential burst. If this is the first cycle in the burst, the address might be the same as the previous internal cycle. Otherwise the address is incremented from the previous instruction fetch that was performed:

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

The types of bursts are shown in Table 4-7.

Table 4-7 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	ARM code fetches
Halfword read	2 bytes	Thumb code fetches

All accesses in a burst are of the same width, direction, and protection type. For more details, see *Instruction interface addressing signals* on page 4-4.

Bursts of byte accesses are not possible with the instruction memory interface.

A burst always starts with an N cycle, or a merged I-S cycle (see *Instruction interface, merged I-S cycles* on page 4-11), and continues with S cycles. A burst comprises transfers of the same type or size. The **IA[31:1]** signal increments during the burst. The other address class signals are unaffected by a burst.

An example of a burst access is shown in Figure 4-3 on page 4-11.

Copyright © 2000 ARM Limited. All rights reserved.

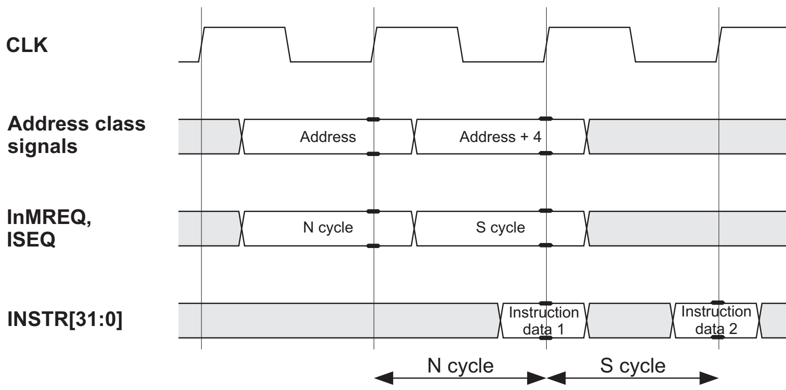


Figure 4-3 Sequential instruction fetch cycles

4.6.3 Instruction interface, internal cycles

During an internal cycle, the ARM9E-S does not require an instruction fetch, because an internal function is being performed, and no useful prefetching can be performed at the same time.

Where possible the ARM9E-S broadcasts the address for the next access, so that decode can start, but the memory controller must not commit to a memory access. This is described further in *Instruction interface, merged I-S cycles*.

4.6.4 Instruction interface, merged I-S cycles

Where possible, the ARM9E-S performs an optimization on the bus to allow extra time for memory decode. When this happens, the address of the next memory cycle is broadcast during an internal cycle on this bus. This allows the memory controller to decode the address, but it must not initiate a memory access during this cycle. In a merged I-S cycle, the next cycle is a sequential cycle to the same memory location. This commits to the access, and the memory controller must initiate the memory access. This is shown in Figure 4-4 on page 4-12.

Copyright © 2000 ARM Limited. All rights reserved.

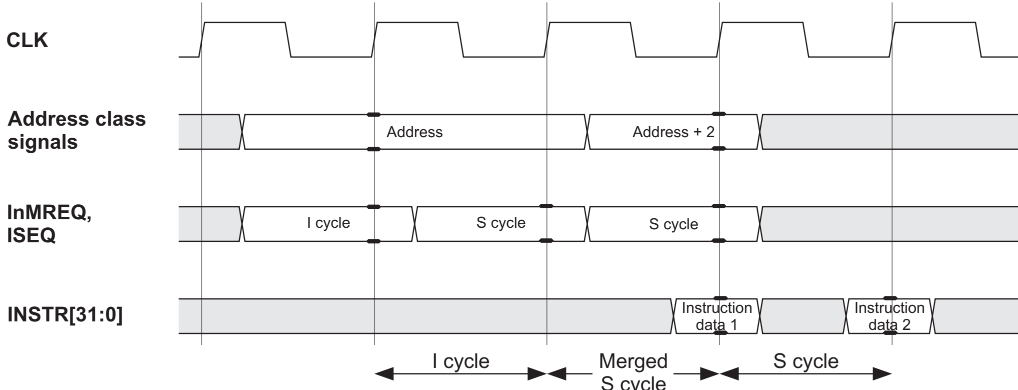


Figure 4-4 Merged I-S cycle

There is an exception to the merged I-S behavior in the case of a coprocessor 15 `MCR`. In this case the **IA** bus is used to transmit data to CP15 (see *Coprocessor 15 MCRs* on page 6-18).

Note

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

Copyright © 2000 ARM Limited. All rights reserved.

4.7 Data interface

The ARM9E-S requests data using the data memory interface.

Data transfers take place in the Memory stage of the pipeline. The operation of the data interface is very similar to the instruction interface.

4.7.1 Data interface signals

The signals in the ARM9E-S data bus interface can be grouped into four categories:

- Clocking and clock control signals:
  - **CLK**
  - **CLKEN**
  - **nRESET**.
- Address class signals:
  - **DA[31:0]**
  - **DnTRANS**
  - **DnRW**
  - **DnM[4:0]**
  - **DMAS[1:0]**
  - **DLOCK**.
- Memory request signals:
  - **DnMREQ**
  - **DSEQ**
  - **DMORE**.
- Data timed signals:
  - **WDATA[31:0]**
  - **RDATA[31:0]**
  - **DABORT**.

Note

All memory accesses are conditioned by the state of the memory request signals. You must not initiate a memory access unless the memory request signals indicate that one is required. See *Data interface cycle types* on page 4-24 for more details.

Copyright © 2000 ARM Limited. All rights reserved.

Each of these signal groups shares a common timing relationship to the bus interface cycle. All signals in the ARM9E-S data interface are generated from, or sampled by the rising edge of **CLK**.

You can extend bus cycles using the **CLKEN** signal (see *Use of CLKEN to control bus cycles* on page 4-31). Unless otherwise stated **CLKEN** is permanently HIGH.

Copyright © 2000 ARM Limited. All rights reserved.

4.8 Data interface addressing signals

The address class signals are:

- *DA[31:0]*
- *DnRW*
- *DMAS[1:0]* on page 4-16
- *DnTRANS* on page 4-16
- *DLOCK* on page 4-17
- *DnM[4:0]* on page 4-17.

4.8.1 DA[31:0]

DA[31:0] is the 32-bit address bus that specifies the address for the transfer. All addresses are byte addresses, so a burst of word accesses results in the address bus incrementing by 4 for each cycle.

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

4.8.2 DnRW

DnRW specifies the direction of the transfer. **DnRW** indicates an ARM9E-S write cycle when HIGH, and an ARM9E-S read cycle when LOW. A burst of S cycles is always either a read burst, or a write burst, because the direction cannot be changed in the middle of a burst.

——— **Note** ———

You must not initiate writes to memory purely on the basis of **DnRW**. You must use the status of the data interface request signals to condition writes to memory. See *Data interface cycle types* on page 4-24 for more details.

———————————

Copyright © 2000 ARM Limited. All rights reserved.

4.8.3 DMAS[1:0]

The **DMAS[1:0]** bus encodes the size of the transfer. The ARM9E-S can transfer word, halfword, and byte quantities. This is encoded on **DMAS[1:0]** as shown in Table 4-8.

Table 4-8 Transfer widths

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

The size of transfer does not change during a burst of S cycles. Bursts of halfword or byte accesses are not possible on the ARM9E-S data interface.

Note

A writable memory system for the ARM9E-S *must* have individual byte write enables. Both the C compiler and the ARM debug tool chain (for example, Multi-ICE) assume that arbitrary bytes in the memory can be written. If individual byte write capability is not provided, you might not be able to use these tools.

4.8.4 DnTRANS

The **DnTRANS** bus encodes information about the transfer. A memory management unit uses this signal to determine if an access is from a privileged mode. Therefore, you can use this signal to implement an access permission scheme. The encoding of **DnTRANS** is shown in Table 4-9.

Table 4-9 DnTRANS encoding

DnTRANS	Mode
0	User
1	Privileged

Copyright © 2000 ARM Limited. All rights reserved.

4.8.5 DLOCK

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

If **DLOCK** is asserted in a cycle, then this indicates that there is another access in the next cycle that must be locked to the first. In the case of a multi-master system, the ARM processor must not be degranted the bus when a locked transaction is being performed.

4.8.6 DnM[4:0]

DnM[4:0] indicates the operating mode of the ARM9E-S. This bus corresponds to the bottom five bits of the CPSR, unless a forced User mode access is being performed, in which case **DnM[4:0]** indicates User mode. These bits are inverted with respect to the CPSR.

 Copyright © 2000 ARM Limited. All rights reserved.

4.9 Data interface data timed signals

The data timed signals are:

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

4.9.1 WDATA[31:0]

WDATA[31:0] is the write data bus. All data written out from the ARM9E-S is broadcast on this bus. Data transfers from the ARM9E-S to a coprocessor also use this bus during C cycles. In normal circumstances, a memory system must sample the **WDATA[31:0]** bus on the rising edge of **CLK** at the end of a write bus cycle. The value on **WDATA[31:0]** is valid only during write cycles.

4.9.2 RDATA[31:0]

RDATA[31:0] is the read data bus, and is used by the ARM9E-S to fetch data. It is sampled on the rising edge of **CLK** at the end of the bus cycle, and is also used during C cycles to transfer data from a coprocessor to the ARM9E-S.

4.9.3 DABORT

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

If **DABORT** is asserted on a data access, it causes the ARM9E-S to take the Data Abort trap.

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

The ARM9E-S design differs from ARM9TDMI in that ARM9TDMI features a combinational path from **DABORT** to **DnMREQ**, **DSEQ**, and **DMORE**. This path is present so that an aborted memory access can cancel memory accesses requested by following instructions.

An example of this is shown in Figure 4-5 on page 4-19, where a load instruction follows an aborted store.

Copyright © 2000 ARM Limited. All rights reserved.

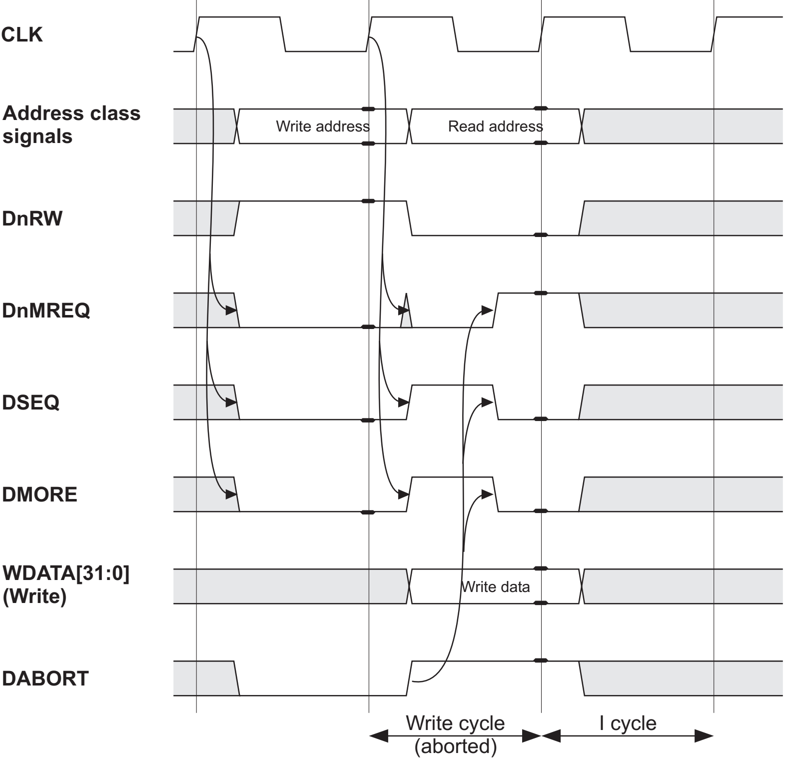


Figure 4-5 ARM9TDMI effect of DABORT on following memory access

This **DABORT** to **DnMREQ**, **DSEQ**, and **DMORE** path has been removed from the ARM9E-S design because:

- a combinational input to output path is undesirable in an ASIC design flow
- the path is critical in ARM9TDMI.

Due to this modification, the memory system connected to ARM9E-S is responsible for ignoring a data memory request made during the cycle of an aborted data transfer. This is necessary to prevent a following memory access from corrupting memory after an aborted access. The memory system must ignore **DnMREQ**, **DSEQ**, and **DMORE** in this case.

Copyright © 2000 ARM Limited. All rights reserved.

Figure 4-6 shows the ARM9E-S behavior for an aborted STR instruction followed by an LDM instruction. While the STR instruction is canceled, a memory request is made in the first cycle of the LDM before the Data Abort exception is taken.

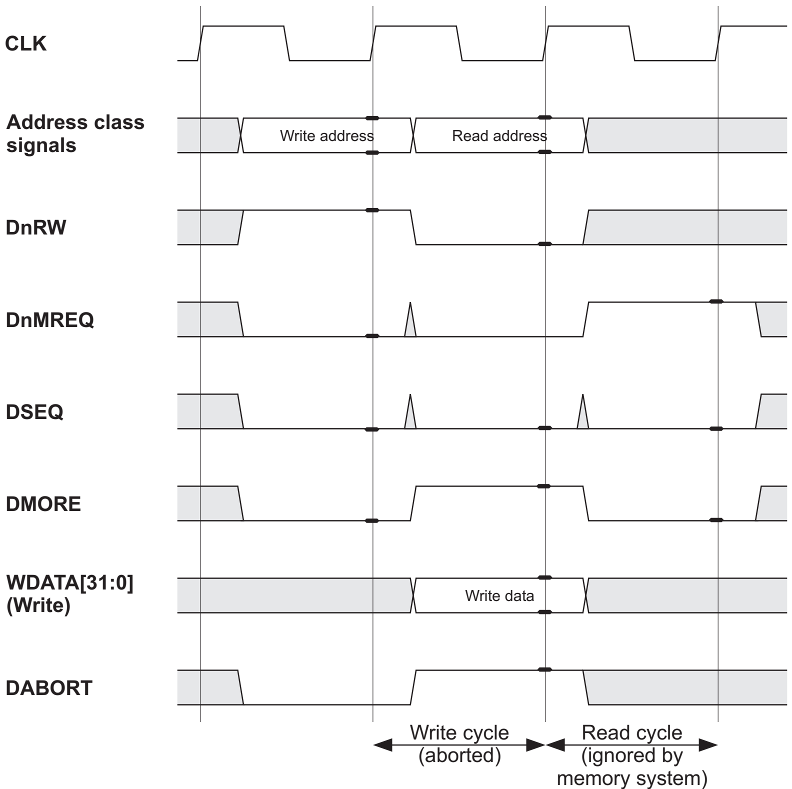


Figure 4-6 ARM9E-S aborted data memory access

For more details about aborts, see *Aborts* on page 2-23.

Copyright © 2000 ARM Limited. All rights reserved.

4.9.4 Byte and halfword accesses

The ARM9E-S indicates the size of a transfer using the **DMAS[1:0]** signals. These are encoded as shown in Table 4-10.

Table 4-10 Transfer size encoding

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

All writable memory in an ARM9E-S based system must support the writing of individual bytes to allow the use of the C compiler and the ARM debug tool chain (for example, Multi-ICE).

The address produced by the ARM9E-S is always byte-aligned. However, the memory system must ignore the insignificant bits of the address. The significant address bits are listed in Table 4-11.

Table 4-11 Significant address bits

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

Reads

When a halfword or byte read is performed, a 32-bit memory system can return the complete 32-bit word, and the ARM9E-S extracts the valid halfword or byte field from it. The fields extracted depend on the state of the **CFGBIGEND** signal, which determines the endianness of the system (see *Memory formats* on page 2-4).

Copyright © 2000 ARM Limited. All rights reserved.

The fields extracted by the ARM9E-S are shown in Table 4-12.

Table 4-12 Word accesses

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

When performing a word load, the ARM9E-S can rotate the data returned internally if the address used is unaligned. Refer to the *ARM Architectural Reference Manual* for more details.

When connecting 8-bit to 16-bit memory systems to the ARM9E-S, you must make sure that the data is presented to the correct byte lanes on the ARM9E-S as shown in Table 4-13 and Table 4-14.

Table 4-13 Halfword accesses

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

Table 4-14 Byte accesses

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

Copyright © 2000 ARM Limited. All rights reserved.

Writes

When the ARM9E-S performs a byte or halfword write, the data being written is replicated across the bus, as illustrated in Figure 4-7. The memory system can use the most convenient copy of the data. A writable memory system must be capable of performing a write to any single byte in the memory system. This capability is required by the ARM C compiler and the Debug tool chain.

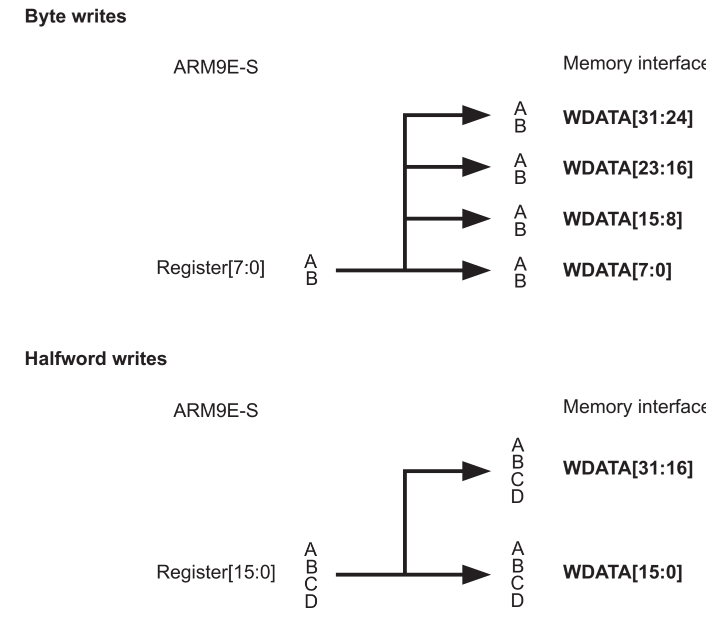


Figure 4-7 Data replication

Copyright © 2000 ARM Limited. All rights reserved.

4.10 Data interface cycle types

The ARM9E-S data interface is pipelined, and so the address class signals and the memory request signals are broadcast in the bus cycle ahead of the bus cycle to which they refer. This gives the maximum time for a memory controller to decode the address, and respond to the access request.

A single memory cycle is shown in Figure 4-8.

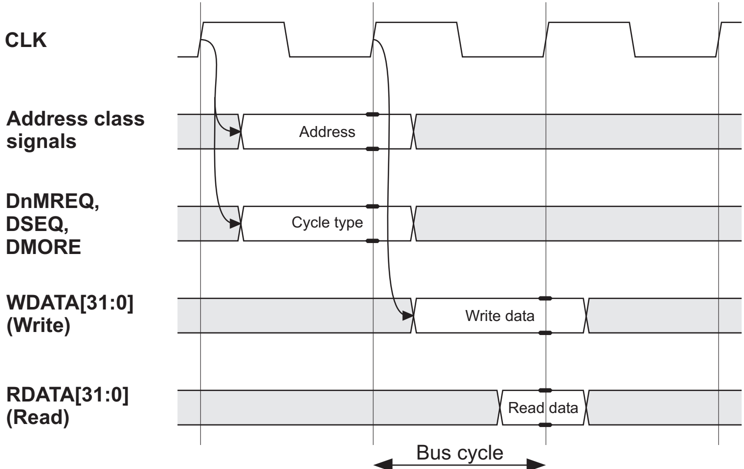


Figure 4-8 Simple memory cycle

The ARM9E-S data interface can perform four different types of memory cycle. These are indicated by the state of the **DnMREQ and DSEQ** signals. Memory cycle types are encoded on the **DnMREQ and DSEQ** signals as shown in Table 4-15.

Table 4-15 Cycle types

DnMREQ	DSEQ	Cycle type	Description
0	0	N cycle	Nonsequential cycle
0	1	S cycle	Sequential cycle
1	0	I cycle	Internal cycle
1	1	C cycle	Coprocessor register transfer cycle

Copyright © 2000 ARM Limited. All rights reserved.

A memory controller for the ARM9E-S must commit to a data memory access only on an N cycle or an S cycle.

The ARM9E-S data interface has four types of memory cycle:

Nonsequential cycle
: During this cycle the ARM9E-S core requests a transfer to or from an address that is unrelated to the address used in the preceding cycle.

Sequential cycle
: During this cycle the ARM9E-S core requests a transfer to or from an address that is one word greater than the address used in the preceding cycle.

Internal cycle
: During this cycle the ARM9E-S core does not require a transfer because it is performing an internal function.

Coprocessor register transfer cycle
: During this cycle the ARM9E-S core uses the data bus to communicate with a coprocessor, but does not require any action by the memory system.

4.10.1 Data interface, nonsequential cycles

A nonsequential cycle is the simplest form of an ARM9E-S data interface cycle, and occurs when the ARM9E-S requests a transfer to or from an address that is unrelated to the address used in the preceding cycle. The memory controller must initiate a memory access to satisfy this request.

The address class signals and the **DnMREQ** and **DSEQ** `= N cycle` are broadcast on the data bus. At the end of the next bus cycle the data is transferred between the CPU and the memory. This is shown in Figure 4-9 on page 4-26.

Copyright © 2000 ARM Limited. All rights reserved.

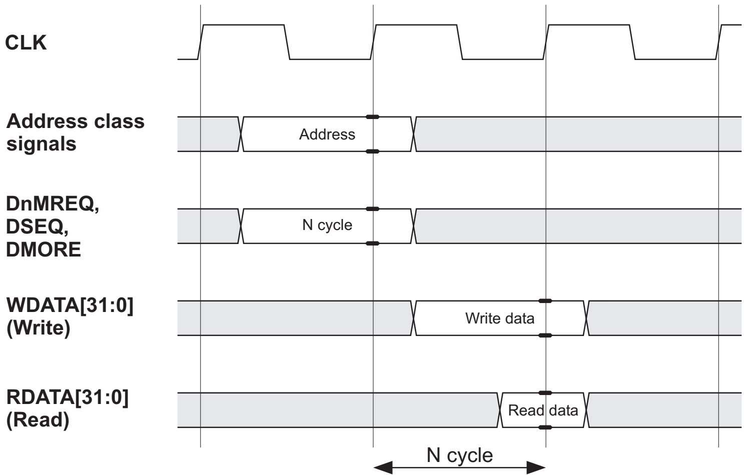


Figure 4-9 Nonsequential data memory cycle

The ARM9E-S can perform back to back, nonsequential memory cycles. This happens, for example, when an `STR` instruction and an `LDR` instruction are executed in succession, as shown in Figure 4-10 on page 4-27.

 Copyright © 2000 ARM Limited. All rights reserved.

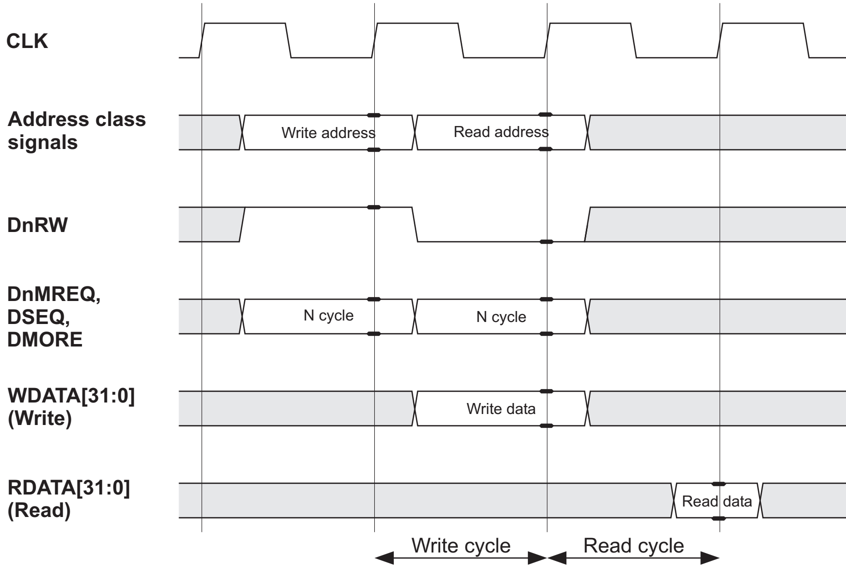


Figure 4-10 Back to back memory cycles

If you are designing a memory controller for the ARM9E-S, and your memory system is unable to cope with this case, use the **CLKEN** signal to extend the bus cycle to allow sufficient cycles for the memory system (see *Use of CLKEN to control bus cycles* on page 4-31).

4.10.2 Data interface, sequential cycles

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

During a sequential cycle, the ARM9E-S requests a memory location that is part of a sequential burst. If this is the first cycle in the burst, the address can be the same as the previous internal cycle. Otherwise the address is incremented from the previous cycle. For a burst of word accesses, the address is incremented by 4 bytes.

Bursts of halfword or byte accesses are not possible on the ARM9E-S data interface.

A burst always starts with an N cycle and continues with S cycles. A burst comprises transfers of the same type. The **DA[31:0]** signal increments during the burst. The other address class signals are unaffected by a burst.

 Copyright © 2000 ARM Limited. All rights reserved.

The types of bursts are shown in Table 4-16.

Table 4-16 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	`LDM` instruction
Word write	4 bytes	`STM` instruction

All accesses in a burst are of the same width, direction, and protection type. For more details, see *Instruction interface addressing signals* on page 4-4.

An example of a burst access is shown in Figure 4-11.

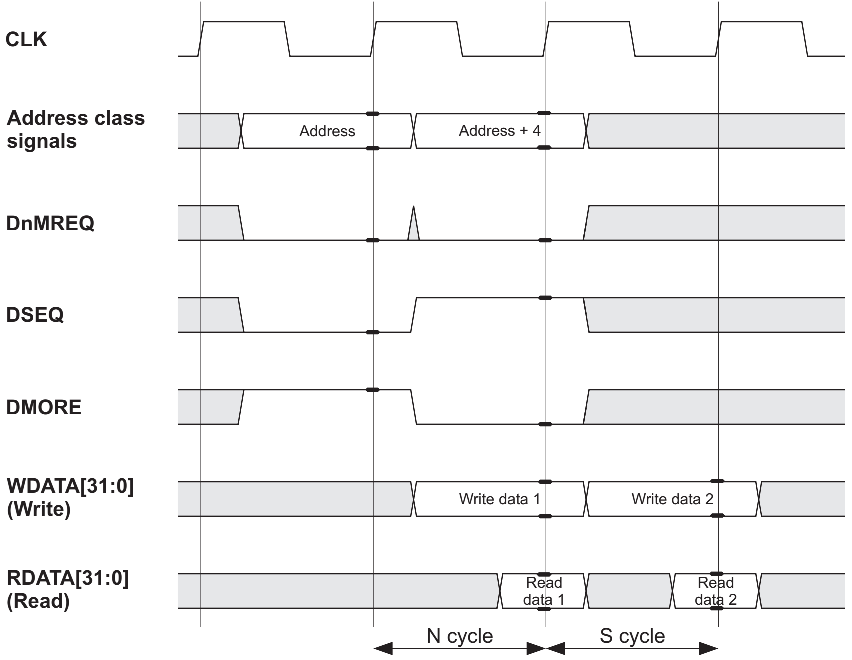


Figure 4-11 Sequential access cycles

The **DMORE** signal is active during load and store multiple instructions and only ever goes HIGH when **DnMREQ** is LOW. This signal effectively gives the same information as **DSEQ**, but a cycle ahead. This information is provided to allow external logic more time to decode sequential cycles.

 Copyright © 2000 ARM Limited. All rights reserved.

4.10.3 Data interface, internal cycles

During an internal cycle, the ARM9E-S does not require a memory access, as an internal function is being performed.

4.10.4 Data interface, merged I-S cycles

The ARM9E-S does not perform merged I-S cycles on the data memory interface.

4.10.5 Data interface, coprocessor register transfer cycles

During a coprocessor register transfer cycle, the ARM9E-S uses the data interface to transfer data to or from a coprocessor. A memory cycle is not required and the memory controller does not initiate a transaction.

The coprocessor interface is described in Chapter 6 *Coprocessor Interface*.

Copyright © 2000 ARM Limited. All rights reserved.

4.11 Endian effects for data transfers

The ARM9E-S supports 32-bit, 16-bit, and 8-bit data memory access sizes. The endian configuration of the processor, set by **CFGBIGEND**, affects only nonword transfers (16-bit and 8-bit transfers).

4.11.1 Writes

For data writes by the processor, the write data is duplicated on the data bus. So for a 16-bit data store, one copy of the data appears on the upper half of the write data bus, **WDATA[31:16]**, and the same data appears on the lower half, **WDATA[15:0]**. For 8-bit writes four copies are output, one on each byte lane:

- **WDATA[31:24]**
- **WDATA[23:16]**
- **WDATA[15:8]**
- **WDATA[7:0]**.

This considerably eases the memory control logic design and helps overcome any endian effects.

4.11.2 Reads

For data reads, the processor reads a specific part of the read data bus. This is determined by:

- the endian configuration
- the size of the transfer
- bits 1 and 0 of the data address bus.

Table 4-13 on page 4-22 shows which bits of the data bus are read for 16-bit reads, and Table 4-14 on page 4-22 shows which bits are read for 8-bit transfers.

For simplicity of design, 32-bits of data can be read from memory and the processor ignores any unwanted bits.

Copyright © 2000 ARM Limited. All rights reserved.

4.12 Use of CLKEN to control bus cycles

The pipelined nature of the ARM9E-S bus interface means that there is a distinction between *clock* cycles and *bus* cycles. You can use **CLKEN** to stretch a *bus* cycle, so that it lasts for many *clock* cycles. The **CLKEN** input extends the timing of bus cycles in increments of complete **CLK** cycles:

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

The **CLKEN** input extends bus cycles on both the instruction and data interfaces when asserted.

In the pipeline, the address class signals and the memory request signals are ahead of the data transfer by one *bus* cycle. In a system using **CLKEN** this can be more than one **CLK** cycle. This is illustrated in Figure 4-12, which shows **CLKEN** being used to extend a nonsequential cycle. In the example, the first N cycle is followed by another N cycle to an unrelated address, and the address for the second access is broadcast before the first access completes.

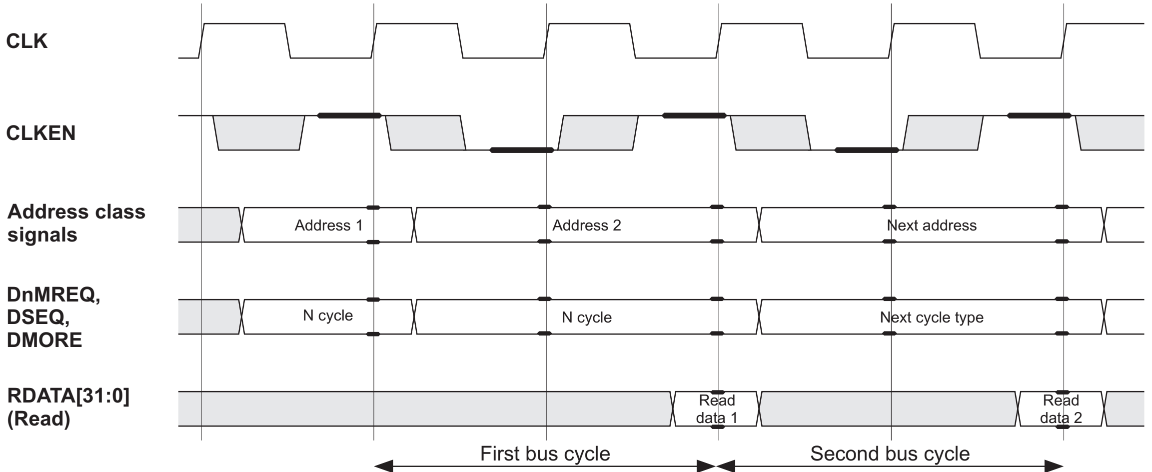


Figure 4-12 Use of CLKEN

Note

When designing a memory controller, you must sample the values of **InMREQ**, **ISEQ**, **DnMREQ**, **DSEQ**, **DMORE**, and the address class signals only when **CLKEN** is HIGH. This ensures that the state of the memory controller is not accidentally updated

 Copyright © 2000 ARM Limited. All rights reserved.

during a waited cycle. In addition, the ARM9E-S can alter the request for a subsequent memory cycle during a waited (**CLKEN** LOW) cycle. See *Withdrawal of memory requests in waited cycles*.

4.12.1 Withdrawal of memory requests in waited cycles

The ARM9E-S can alter the value of the memory request and address signals during cycles in which **CLKEN** is LOW. This is done to improve the worst case interrupt latency of ARM9E-S systems. For example, a pending memory request can be withdrawn if the core is about to take an interrupt and the access is unnecessary.

The ARM9E-S does not alter or withdraw any access to which it is *committed*. An access is said to be *committed* when the address and request signals are sampled on the rising edge of **CLK** when **CLKEN** is HIGH.

The ARM9E-S only attempts to alter or withdraw an uncommitted access during the extended (or waited) bus cycle of a previous access. Alteration of the next memory request during a waited bus cycle is shown in Figure 4-13.

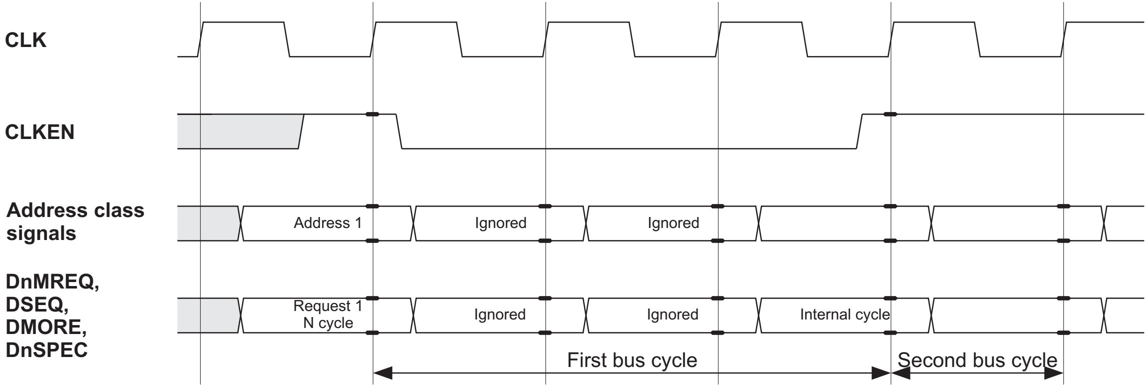


Figure 4-13 Alteration of next memory request during waited bus cycle

Note

This behavior affects the **IA**, **InMREQ**, **ISEQ**, **DA**, **DnMREQ**, **DSEQ**, **DMORE**, and **DnSPEC** outputs of the ARM9E-S.

 Copyright © 2000 ARM Limited. All rights reserved.

Chapter 5
Interrupts

This chapter describes the ARM9E-S interrupt behavior. It contains the following sections:

- *About interrupts* on page 5-2
- *Hardware interface* on page 5-3
- *Maximum interrupt latency* on page 5-7
- *Minimum interrupt latency* on page 5-8.

Copyright © 2000 ARM Limited. All rights reserved.

5.1 About interrupts

The ARM9E-S provides a two-level, fixed-priority asynchronous hardware interrupt scheme. *Asynchronous* is used here to mean asynchronous to the instruction flow, not to the processor clock (**CLK**). Refer to Chapter 9 *AC Parameters* for details on interrupt signal timing.

The *Fast Interrupt Request* (FIQ) exception provides support for fast interrupts. The *Interrupt Request* (IRQ) exception provides support for normal priority interrupts. Refer to *Exceptions* on page 2-20 for more details about the programmer's model for interrupts.

This chapter discusses:

- issues concerning the hardware interface to the ARM9E-S interrupt mechanism that a system designer must be aware of when integrating an ARM9E-S system
- issues that a programmer must be aware of when writing interrupt handler routines
- the worst case and best case interrupt latency.

Copyright © 2000 ARM Limited. All rights reserved.

5.2 Hardware interface

The hardware interrupt is described under the following headings:

- *Generating an interrupt*
- *Synchronization*
- *Re-enabling interrupts after an interrupt exception*
- *Interrupt registers* on page 5-5.

5.2.1 Generating an interrupt

You can make the ARM9E-S take the FIQ or IRQ exceptions (if interrupts are enabled within the core) by asserting (LOW) the **nFIQ** or **nIRQ** inputs, respectively.

It is essential that once asserted, the interrupt input remains asserted until the ARM9E-S has completed its interrupt exception entry sequence. When an interrupt input is asserted, it must remain asserted until the ARM9E-S acknowledges to the source of the interrupt that the interrupt has been taken. This acknowledgement normally occurs when the interrupt service routine accesses the peripheral causing the interrupt, for example:

- by reading an interrupt status register in the systems interrupt controller
- by writing to a clear interrupt control bit
- by writing data to, or reading data from the interrupting peripheral.

5.2.2 Synchronization

The **nFIQ** and **nIRQ** inputs are synchronous inputs to the ARM9E-S, and must be setup and held about the rising edge of the ARM9E-S clock, **CLK**. If interrupt events that are asynchronous to **CLK** are present in a system, synchronization register(s) that are external to the ARM9E-S are required.

5.2.3 Re-enabling interrupts after an interrupt exception

You must take care when re-enabling interrupts (for example at the end of an interrupt routine or with a reentrant interrupt handler). You must ensure that the original source of the interrupt has been removed before interrupts are enabled again on the ARM9E-S. If you cannot guarantee this, the ARM9E-S might retake the interrupt exception prematurely.

When considering the timing relation of removing the source of interrupt and re-enabling interrupts on the ARM9E-S, you must take into account the pipelined nature of the ARM9E-S and the memory system to which it is connected. For example, the instruction that causes the removal of the interrupt request (that is, deassertion of

Copyright © 2000 ARM Limited. All rights reserved.

nFIQ or **nIRQ**) typically does not take effect until after the Memory stage of that instruction. The instruction that re-enables interrupts on the ARM9E-S can cause the ARM9E-S to be sensitive to interrupts as early as the Execute stage of that instruction.

For example, consider the following instruction sequence:

```
STR r0, [r1] ;Write to interrupt controller, clearing interrupt
SUBS pc, r14, #4 ;Return from interrupt routine
```

The execution of this code sequence is illustrated in Figure 5-1.

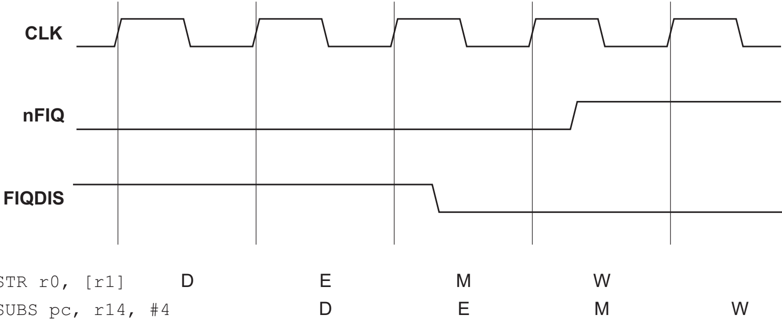


Figure 5-1 Retaking the FIQ exception

In Figure 5-1, the `STR` to the interrupt controller does not cause the deassertion of the **nFIQ** input until cycle 4. The `SUBS` instruction causes the ARM9E-S to be sensitive to interrupts during cycle 3**.**

Because of this timing relationship, the ARM9E-S retakes the FIQ exception in this example.

The **FIQDIS** (and similarly **IRQDIS**) output from the ARM9E-S indicates when the ARM9E-S is sensitive to the state of the **nFIQ** (**nIRQ**) input (0 for sensitive, 1 for insensitive). If **nFIQ** is asserted in the same cycle that **FIQDIS** is LOW, the ARM9E-S takes the FIQ exception in a later cycle, even if the **nFIQ** input is subsequently deasserted.

There are several approaches that you can adopt to ensure that interrupts are not enabled too early on the ARM9E-S. The best approach is highly dependent on the overall system, and can be a combination of hardware and software.

 Copyright © 2000 ARM Limited. All rights reserved.

Example approaches are:

- Analyze the system and ensure enough instructions separate the instruction that removes the interrupt and the instruction that re-enables interrupts on the ARM9E-S.
- Have a software polling mechanism that reads back a status bit from the system interrupt controller until it indicates that the interrupt has been removed before re-enabling interrupts.
- Have a hardware system that stalls the ARM9E-S until the interrupt has been removed.

5.2.4 Interrupt registers

Before use, the **nFIQ** and **nIRQ** inputs are registered internally to the ARM9E-S. To improve interrupt latency, the registers are not conditioned by **CLKEN**, and run freely, off the system clock, **CLK**. Internally, the ARM9E-S can use the registered **nFIQ** or **nIRQ** status to prepare for interrupt entry, even if the rest of the core is being waited by **CLKEN**. The registered interrupt signals can only update if **CLK** is running. Because of this, the best interrupt latency can only be achieved if **CLK** is not stopped. This requirement is counteracted by power saving features of a system (for instance, stopping **CLK** while waiting for a slow memory device, or a power-down mode where **CLK** is stopped). In systems like this, you can still achieve the best interrupt latency if you replace the final disabled **CLK** cycle with one waited (**CLKEN** = 0) cycle.

Figure 5-2 shows a system where **CLK** is stopped by external clock-gating for a number of cycles.

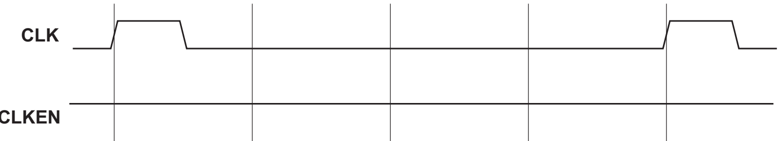


Figure 5-2 Stopping CLK for power saving

Figure 5-3 on page 5-6 shows a system which achieves most of the power saving benefits of the system shown in Figure 5-2, while at the same time achieving best interrupt latency.

Copyright © 2000 ARM Limited. All rights reserved.

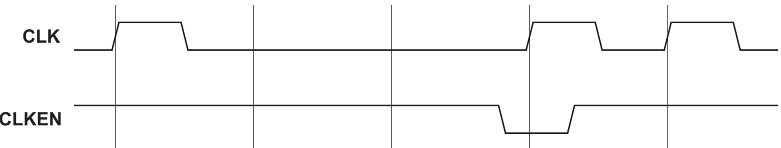


Figure 5-3 Using CLK and CLKEN for best interrupt latency

The system shown in Figure 5-3 combines **CLK** stopping and **CLKEN** waiting for best power and interrupt latency performance.

Copyright © 2000 ARM Limited. All rights reserved.

5.3 Maximum interrupt latency

The processor samples the interrupt input pins on the rising-edge of the system clock, **CLK**. The sampled signal is examined and can cause an interrupt in the following cases:

- Whenever a new instruction is scheduled to enter the Execute stage of the pipeline.
- Whenever a new instruction is in the Execute stage for the first *cycle* of its execution. Here *cycle* refers to **CLK** cycles with **CLKEN** HIGH.
- Whenever a coprocessor instruction is being busy waited in the Execute stage.
- Whenever a new instruction which interlocked in the Execute stage has just progressed to its first *active* Execute cycle.

If the sampled signal is asserted at the same time as a multicycle instruction has started its second or later cycle of execution, the interrupt exception entry does not start until the instruction has completed.

The worst-case interrupt latency occurs when the longest possible `LDM` instruction incurs a Data Abort. The processor must enter the Data Abort mode before taking the interrupt so that the interrupt exception exit can occur correctly. This causes a worst-case latency of 24 cycles:

- The longest `LDM` instruction is one that loads all of the registers, including the PC. Counting the first Execute cycle as 1, the `LDM` takes 16 cycles.
- The last word to be transferred by the `LDM` is transferred in cycle 17, and the abort status for the transfer is returned in this cycle.
- If a Data Abort happens, the processor detects this in cycle 18 and prepares for the Data Abort exception entry in cycle 19.
- Cycles 20 and 21 are the Fetch and Decode stages of the Data Abort entry respectively.
- During cycle 22, the processor prepares for FIQ entry, issuing Fetch and Decode cycles in cycles 23 and 24.
- Therefore, the first instruction in the FIQ routine enters the Execute stage of the pipeline in stage 25, giving a worst-case latency of 24 cycles.

 Copyright © 2000 ARM Limited. All rights reserved.

5.4 Minimum interrupt latency

The minimum latency for **FIQ** or **IRQ** is the shortest time the request can be sampled by the input register (one cycle), plus the exception entry time (three cycles). The first interrupt instruction enters the Execute pipeline stage four cycles after the interrupt is asserted.

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 6
ARM9E-S Coprocessor Interface

This chapter describes the ARM9E-S coprocessor interface. It contains the following sections:

- *About the coprocessor interface* on page 6-2
- *LDC/STC* on page 6-4
- *MCR/MRC* on page 6-8
- *MCRR/MRRC* on page 6-10
- *Interlocked MCR* on page 6-12
- *Interlocked MCRR* on page 6-13
- *CDP* on page 6-14
- *Privileged instructions* on page 6-16
- *Busy-waiting and interrupts* on page 6-17
- *Coprocessor 15 MCRs* on page 6-18
- *Connecting coprocessors* on page 6-19.

 Copyright © 2000 ARM Limited. All rights reserved.

6.1 About the coprocessor interface

The ARM9E-S supports the connection of coprocessors. All types of ARM coprocessors are supported. Coprocessors determine the instructions they need to execute using a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM pipeline and the coprocessor pipeline. The coprocessor determines when an instruction is being fetched by the ARM9E-S, so that the instruction can be loaded into the coprocessor, and the pipeline follower advanced.

The coprocessor can be run either in step with the ARM9E-S pipeline, or one cycle behind, depending on the timing priorities. The implications of the two approaches are discussed in:

- *Coprocessor pipeline operates in step with the ARM9E-S*
- *Coprocessor pipeline one cycle behind the ARM9E-S*.

6.1.1 Coprocessor pipeline operates in step with the ARM9E-S

In this case, the pipeline follower inside the coprocessor matches that of the ARM9E-S exactly. This complicates the timing of key signals such as the **INSTR** and **CLKEN** inputs, because these now become more heavily loaded and therefore incur more delay. For this reason, this method is only recommended for tightly integrated coprocessors such as CP15, the system coprocessor.

6.1.2 Coprocessor pipeline one cycle behind the ARM9E-S

This method is recommended for external coprocessors. A coprocessor interface block pipelines the instruction and control signals so that the loading is reduced on these critical signals. This means that the pipeline in the coprocessor operates one cycle behind the ARM9E-S pipeline. The disadvantage of this is that outputs of the coprocessor are still expected in the correct pipeline stage, as seen from the ARM9E-S. The most critical signal in this situation is likely to be **CHSD[1:0]**, the coprocessor decode handshake signal. This must return the availability of the coprocessor by the end of the decode cycle, as seen by the ARM9E-S. This is equivalent to the fetch cycle of the coprocessor pipeline, and therefore there is not much time to generate this signal. This means that the design might have to insert wait states for external coprocessor accesses.

There are three classes of coprocessor instructions:

- `LDC/STC`
- `MCR/MRC`
- `CDP`.

Copyright © 2000 ARM Limited. All rights reserved.

Examples of how a coprocessor must execute these instruction classes are given in:

- *LDC/STC* on page 6-4
- *MCR/MRC* on page 6-8
- *Interlocked MCR* on page 6-12
- *CDP* on page 6-14.

Note

For the sake of clarity, all timing diagrams assume a system where the coprocessor pipeline operates in step with the ARM9E-S.

Copyright © 2000 ARM Limited. All rights reserved.

6.2 LDC/STC

The number of words transferred is determined by how the coprocessor drives the **CHSD[1:0]** and **CHSE[1:0]** buses. In the example ARM9E-S `LDC/STC` cycle timing shown in Figure 6-1, four words of data are transferred.

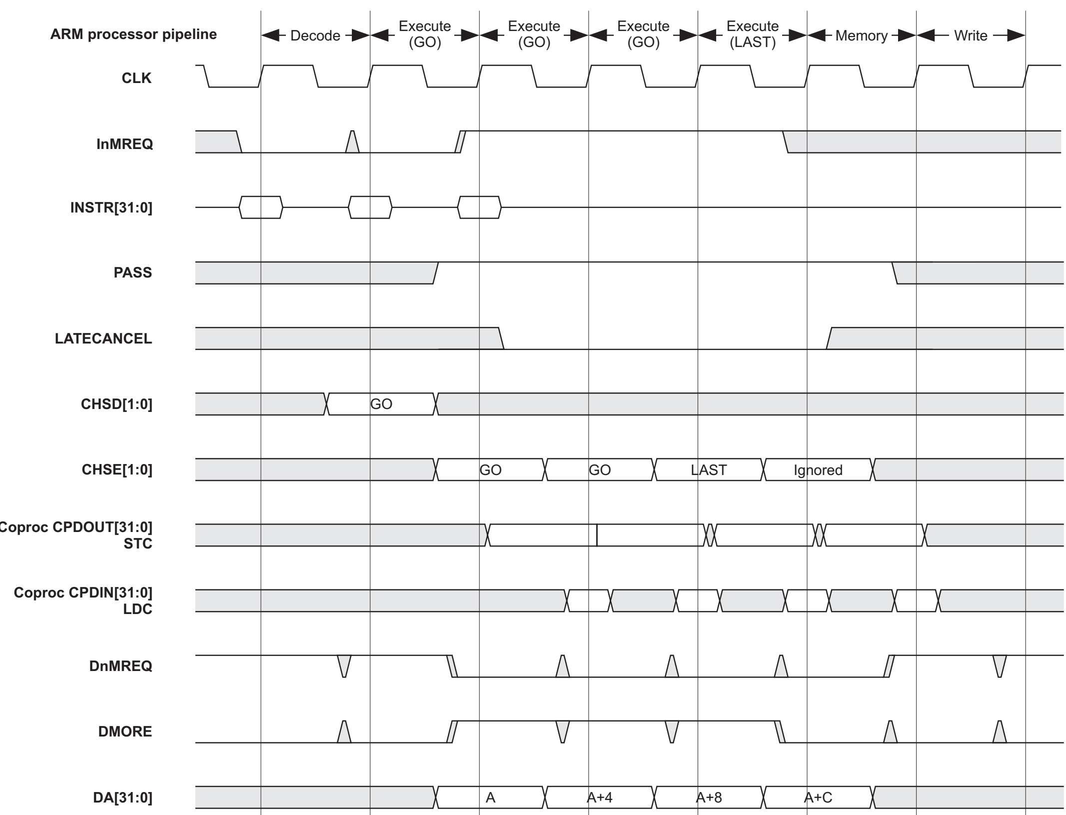


Figure 6-1 ARM9E-S LDC/STC cycle timing

As with all other instructions, the ARM9E-S processor core performs the main decode using the rising edge of the clock during the Decode stage. From this, the core commits to executing the instruction, and so performs an instruction fetch. The coprocessor instruction pipeline keeps in step with the ARM9E-S by monitoring **InMREQ**.

Copyright © 2000 ARM Limited. All rights reserved.

At the rising edge of **CLK**, if **CLKEN** is HIGH, and **InMREQ** is LOW, an instruction fetch is taking place, and **INSTR[31:0]** contains the fetched instruction on the next rising edge of the clock, when **CLKEN** is HIGH. This means that:

- the last instruction fetched must enter the Decode stage of the coprocessor pipeline
- the instruction in the Decode stage of the coprocessor pipeline must enter its Execute stage
- the fetched instruction must be sampled.

In all other cases, the ARM9E-S pipeline is stalled, and the coprocessor pipeline must not advance.

Figure 6-2 shows the timing for these signals, and indicates when the coprocessor pipeline must advance its state. In this timing diagram, Coproc clock shows the effective clock applied to the pipeline follower in the coprocessor. It is derived such that the coprocessor state must only advance on rising **CLK** edges when **CLKEN** is HIGH. The method of implementing this is dependent on the design style used, such as clock gating or register recirculating.

For efficient coprocessor design, an unmodified version of **CLK** must be applied to the Execution stage of the coprocessor. This allows the coprocessor to continue executing an instruction even when the ARM9E-S pipeline is stalled.

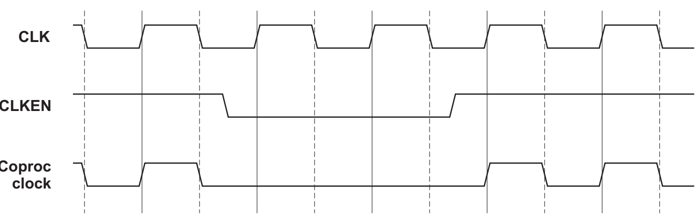


Figure 6-2 ARM9E-S coprocessor clocking

During the Execute stage, the condition codes are compared with the flags to determine whether the instruction really executes or not. The output **PASS** is asserted (HIGH) if the instruction in the Execute stage of the coprocessor pipeline:

- is a coprocessor instruction
- has passed its condition codes.

Copyright © 2000 ARM Limited. All rights reserved.

If a coprocessor instruction busy-waits, **PASS** is asserted on every cycle until the coprocessor instruction is executed. If an interrupt occurs during busy-waiting, **PASS** is driven LOW, and the coprocessor stops execution of the coprocessor instruction.

A further output, **LATECANCEL**, cancels a coprocessor instruction when the instruction preceding it caused a Data Abort, or a previous instruction caused a watchpoint. **LATECANCEL** can be asserted even if there is no coprocessor instruction being executed. For coprocessor instructions, **LATECANCEL** is valid on the rising edge of **CLK** on the cycle that follows the first Execute cycle of the coprocessor instruction. See *CDP* on page 6-14 for an example of **LATECANCEL** behavior.

On the rising edge of the clock, the ARM9E-S processor core examines the coprocessor handshake signals **CHSD[1:0]** or **CHSE[1:0]**:

- If a new instruction is entering the Execute stage in the next cycle, the core examines **CHSD[1:0]**.
- If the currently executing coprocessor instruction requires another Execute cycle, the core examines **CHSE[1:0]**.

The handshake signals encode one of four states:

ABSENT If there is no coprocessor attached that can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM9E-S processor core takes the undefined instruction trap.

WAIT If there is a coprocessor attached that can handle the instruction, but not immediately, the coprocessor handshake signals are driven to indicate that the ARM9E-S processor core must stall until the coprocessor can catch up. This is known as the *busy-wait* condition. In this case, the ARM9E-S processor core loops in an idle state waiting for **CHSE[1:0]** to be driven to another state, or for an interrupt to occur.

If **CHSE[1:0]** changes to ABSENT, the undefined instruction trap is taken. If **CHSE[1:0]** changes to GO or LAST, the instruction proceeds as follows.

If an interrupt occurs, the ARM9E-S processor core is forced out of the busy-wait state. This is indicated to the coprocessor by the **PASS** signal going LOW. The instruction is restarted later and so the coprocessor must not commit to the instruction (it must not change any of the coprocessor state) until it has seen **PASS** HIGH, when the handshake signals indicate the GO or LAST condition.

GO The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM9E-S processor core and the coprocessor must also consider the

Copyright © 2000 ARM Limited. All rights reserved.

state of the **PASS** signal before actually committing to the instruction. For an `LDC` or `STC` instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still have to be transferred. When only one further word is to be transferred, the coprocessor drives the handshake signals with LAST. During the Execute stage, the ARM9E-S processor core outputs the address for the `LDC` or `STC`. Also in this cycle, **DnMREQ** is driven LOW, indicating to the memory system that a memory access is required at the data end of the device. The timing for the data on **RDATA[31:0]** for an `LDC` and **WDATA[31:0]** for an `STC` is shown in Figure 4-1 on page 4-4.

LAST An `LDC` or `STC` can be used for more than one item of data. If this is the case, possibly after busy waiting, the coprocessor drives the coprocessor handshake signals with a number of GO states, and in the penultimate cycle drives LAST (LAST indicating that the next transfer is the final one). If there is only one transfer, the sequence is [WAIT,[WAIT,...]],LAST.

6.2.1 Coprocessor handshake encoding

Table 6-1 shows how the handshake signals **CHSD[1:0]** and **CHSE[1:0]** are encoded.

Table 6-1 Handshake signals

Handshake signal	CHSD[1:0], CHSE[1:0]
ABSENT	10
WAIT	00
GO	01
LAST	11

Copyright © 2000 ARM Limited. All rights reserved.

6.3 MCR/MRC

`MCR` and `MRC` cycles look very similar to `STC` or `LDC`. An example is shown in Figure 6-3.

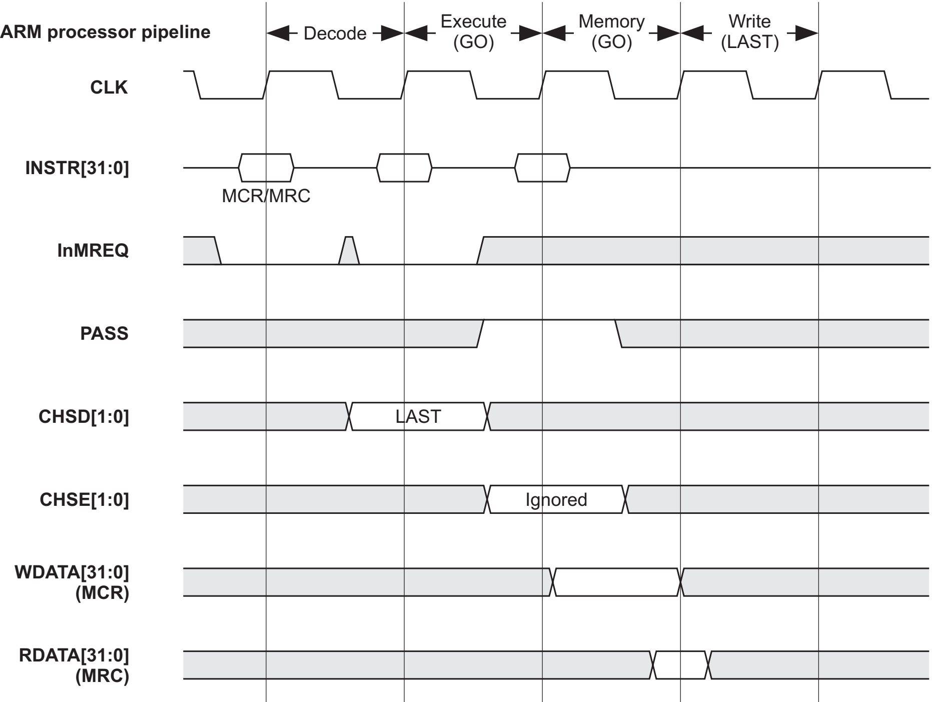


Figure 6-3 ARM9E-S MCR or MRC transfer timing

First **InMREQ** is driven LOW to denote that the instruction on **INSTR[31:0]** is entering the decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSD[1:0]** as required.

In the next cycle **InMREQ** is driven LOW to denote that the instruction has now been issued to the Execute stage. If the condition codes pass, and the instruction is to be executed, the **PASS** signal is driven HIGH and the **CHSD[1:0]** handshake bus is examined by the core (it is ignored in all other cases).

 Copyright © 2000 ARM Limited. All rights reserved.

For any successive Execute cycles the **CHSE[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction is committed. In the case of an `MCR`, the **WDATA[31:0]** bus is driven with the register data. In the case of an `MRC`, **RDATA[31:0]** is sampled at the end of the ARM9E-S Memory stage and written to the destination register during the next cycle.

Copyright © 2000 ARM Limited. All rights reserved.

6.4 MCRR/MRRC

`MCRR` and `MRRC` cycles look very similar to `STC` or `LDC`. An example is shown in Figure 6-4.

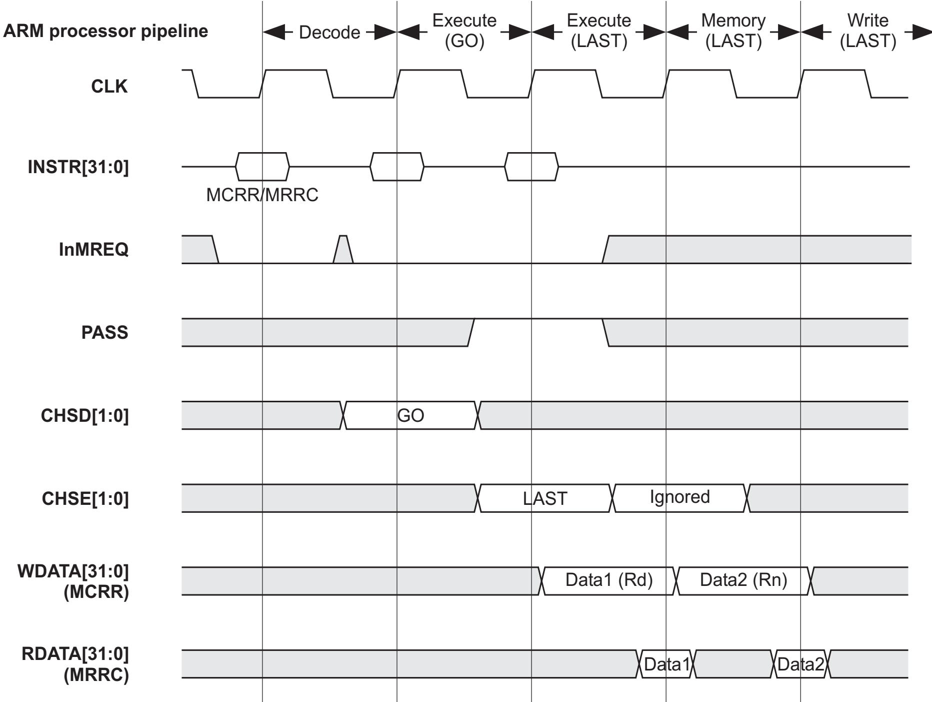


Figure 6-4 ARM9E-S MCRR or MRRC transfer timing

First **InMREQ** is driven LOW to denote that the instruction on **INSTR[31:0]** is entering the Decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSD[1:0]** as required.

In the next cycle **InMREQ** is driven LOW to denote that the instruction has now been issued to the Execute stage. If the condition codes pass, and the instruction is to be executed, the **PASS** signal is driven HIGH and the **CHSD[1:0]** handshake bus is examined by the core (it is ignored in all other cases).

For any successive Execute cycles the **CHSE[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction proceeds to its final Execute cycle. In the case of an `MCRR`, the **WDATA[31:0]** bus is driven with the first register data during

Copyright © 2000 ARM Limited. All rights reserved.

the second Execute cycle, and the second register data in the Memory cycle. In the case of an `MRRC`, **RDATA[31:0]** is sampled at the end of the second Execute and first Memory cycles and written to the destination registers during the next cycle.

 Copyright © 2000 ARM Limited. All rights reserved.

6.5 Interlocked MCR

If the data for an `MCR` operation is not available inside the ARM9E-S pipeline during its first Decode cycle, the ARM9E-S pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination from a preceding `LDR` instruction. In this situation the `MCR` instruction enters the Decode stage of the coprocessor pipeline, and remains there until it can enter the Execute stage.

Figure 6-5 gives an example of an interlocked MCR.

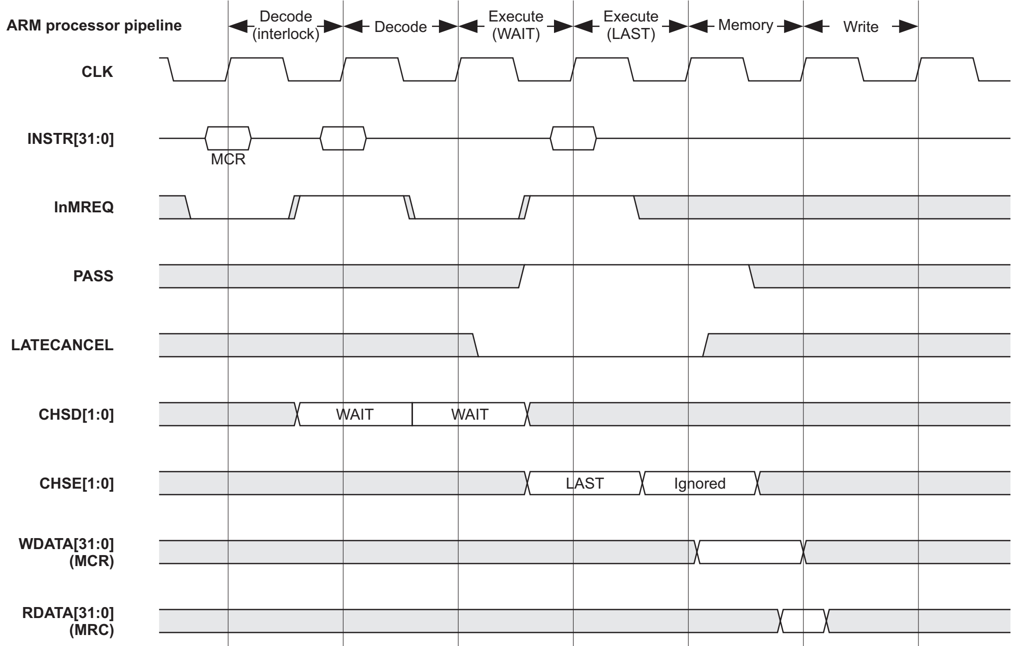


Figure 6-5 ARM9E-S interlocked MCR

Copyright © 2000 ARM Limited. All rights reserved.

6.6 Interlocked MCRR

If the data for an `MCRR` operation is not available inside the ARM9E-S pipeline during its first Decode cycle, the ARM9E-S pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination from a preceding `LDR` instruction. In this situation the `MCRR` instruction enters the Decode stage of the coprocessor pipeline, and remains there until it can enter the Execute stage.

Figure 6-6 gives an example of an interlocked MCRR.

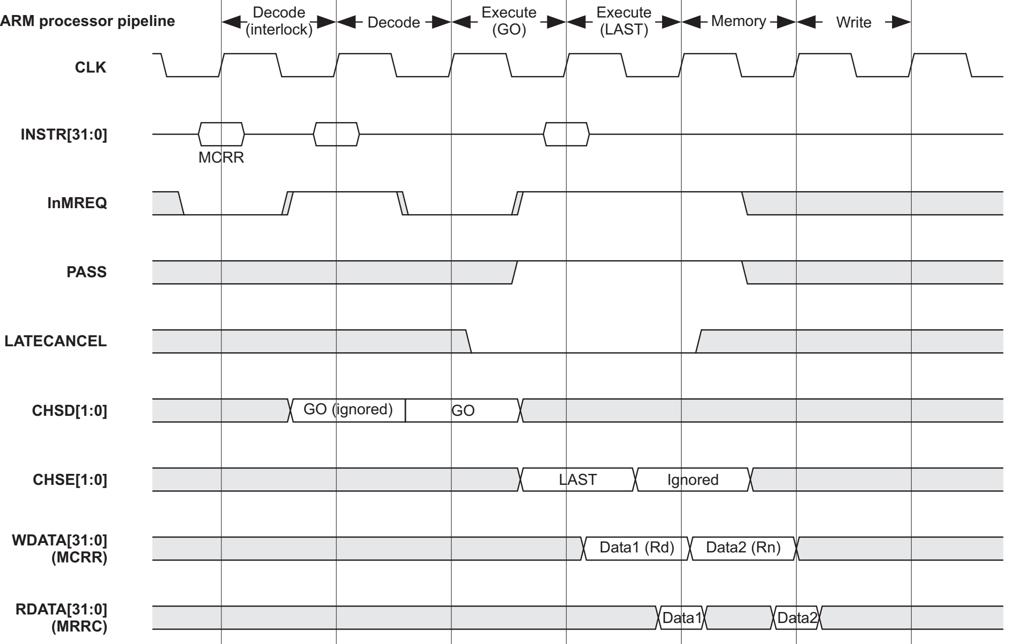


Figure 6-6 ARM9E-S interlocked MCRR

Copyright © 2000 ARM Limited. All rights reserved.

6.7 CDP

`CDP` instructions normally execute in a single cycle. Like all the previous cycles, **InMREQ** is driven LOW to signal when an instruction is entering the Decode stage and again when it reaches the Execute stage of the pipeline:

- if the coprocessor can accept the instruction for execution, the **PASS** signal is driven HIGH during the Execute cycle
- if the coprocessor can execute the instruction immediately it drives **CHSD[1:0]** with LAST
- if the instruction requires a busy-wait cycle, the coprocessor drives **CHSD[1:0]** with WAIT and then **CHSE[1:0]** with LAST.

Figure 6-7 shows a `CDP` which is canceled due to the previous instruction causing a Data Abort.

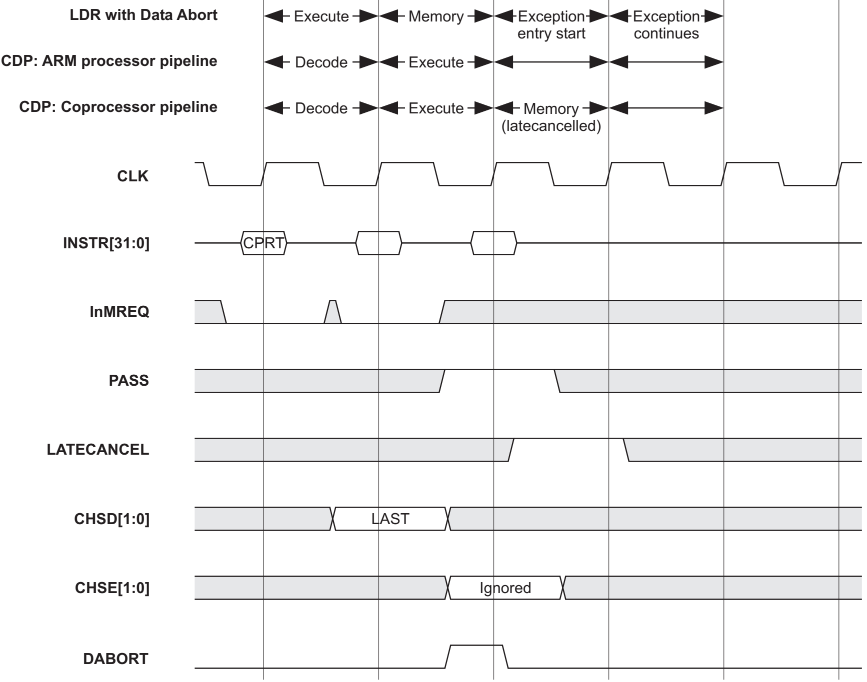


Figure 6-7 ARM9E-S late-canceled CDP

 Copyright © 2000 ARM Limited. All rights reserved.

The `CDP` instruction enters the Execute stage of the pipeline and is signaled to execute by **PASS**. In the following cycle **LATECANCEL** is asserted. This causes the coprocessor to terminate execution of the `CDP` instruction and prevents the `CDP` instruction from causing state changes to the coprocessor.

Copyright © 2000 ARM Limited. All rights reserved.

6.8 Privileged instructions

The coprocessor might restrict certain instructions for use in privileged modes only. To do this, the coprocessor has to track the **InTRANS** output. Figure 6-8 shows how **InTRANS** changes after a mode change.

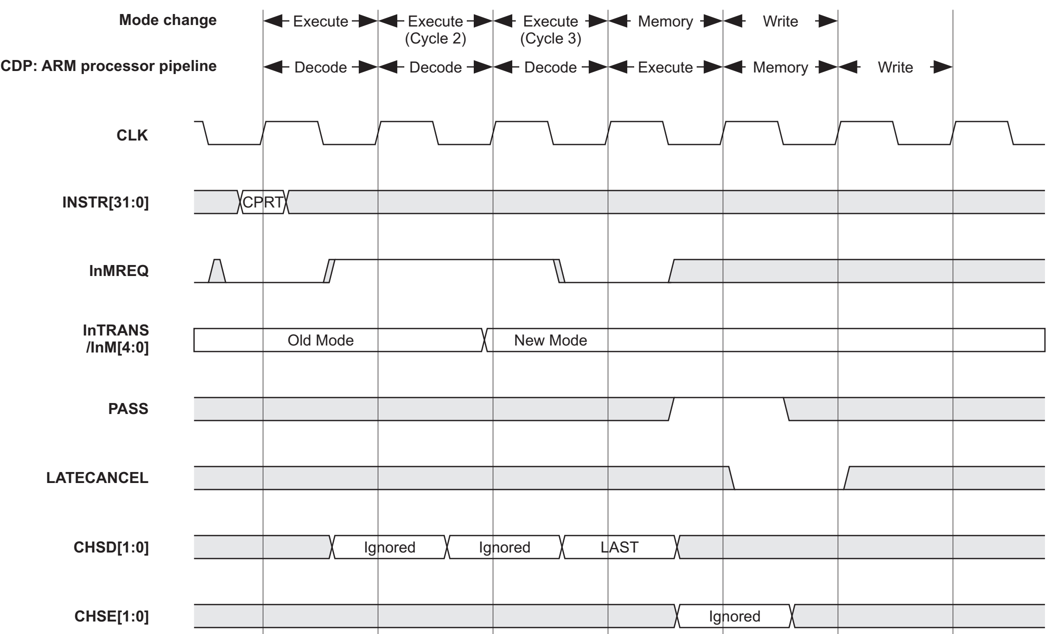


Figure 6-8 ARM9E-S privileged instructions

The first two **CHSD** responses are ignored by the ARM9E-S because it is only the final **CHSD** response, as the instruction moves from Decode into Execute, that counts. This allows the coprocessor to change its response as **InTRANS**/**InM** changes.

Copyright © 2000 ARM Limited. All rights reserved.

6.9 Busy-waiting and interrupts

The coprocessor is permitted to stall, or busy-wait, the processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the Decode stage instruction drives WAIT onto **CHSD[1:0]**. When the instruction concerned enters the Execute stage of the pipeline the coprocessor can drive WAIT onto **CHSE[1:0]** for as many cycles as necessary to keep the instruction in the busy-wait loop.

For interrupt latency reasons, the coprocessor can be interrupted while busy-waiting, causing the instruction to be abandoned. Abandoning execution is done through **PASS**. The coprocessor must monitor the state of **PASS** during every busy-wait cycle. If it is HIGH, the instruction must still be executed. If it is LOW, the instruction must be abandoned. Figure 6-9 shows a busy-waited coprocessor instruction being abandoned due to an interrupt.

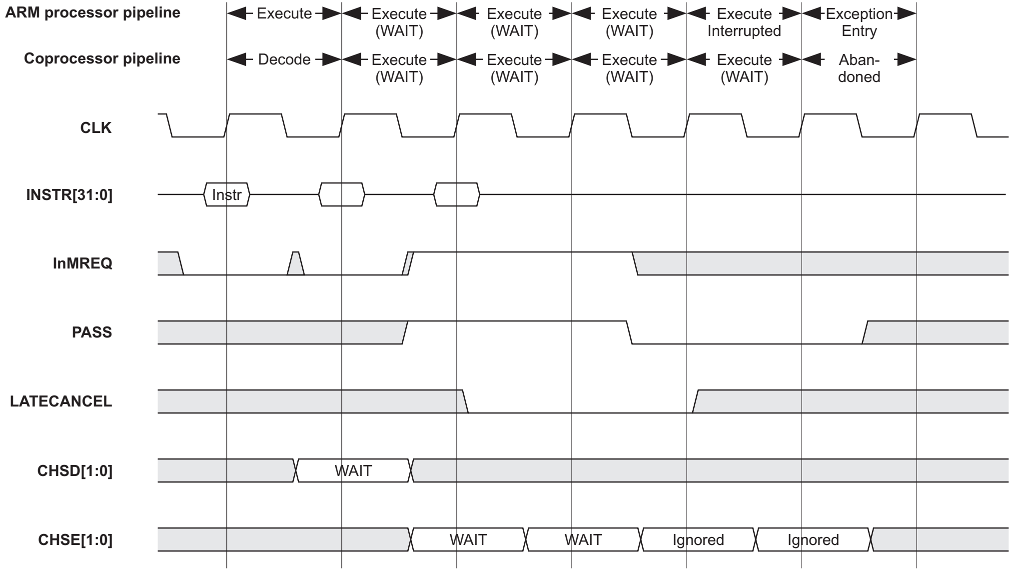


Figure 6-9 ARM9E-S busy waiting and interrupts

Copyright © 2000 ARM Limited. All rights reserved.

6.10 Coprocessor 15 MCRs

Coprocessor 15 is typically reserved for use as a system control coprocessor. For an `MCR` to coprocessor 15, it is possible to transfer the coprocessor data to the coprocessor on the **IA** and **DA** buses. To do this the coprocessor must drive GO on the coprocessor handshake signals for a number of cycles. For each cycle that the coprocessor responds with GO on the handshake signals, the coprocessor data is driven onto **IA** and **DA** as shown in Figure 6-10.

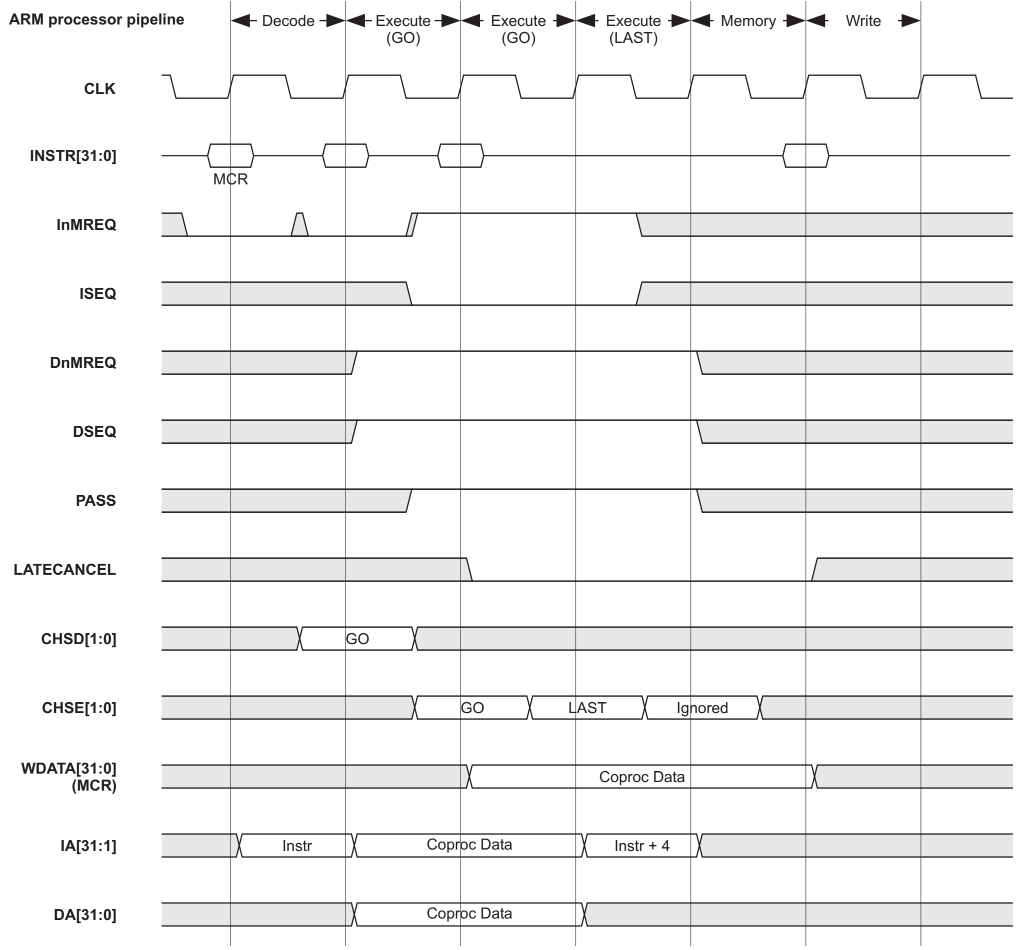


Figure 6-10 ARM9E-S coprocessor 15 MCRs

Copyright © 2000 ARM Limited. All rights reserved.

6.11 Connecting coprocessors

A coprocessor in an ARM9E-S system needs to have 32-bit connections to:

- data from memory (instruction stream and `LDC`)
- write data from the ARM9E-S (`MCR`)
- read data to the ARM9E-S (`MRC`).

6.11.1 Connecting a single coprocessor

An example of how to connect a coprocessor into an ARM9E-S system is shown in Figure 6-11.

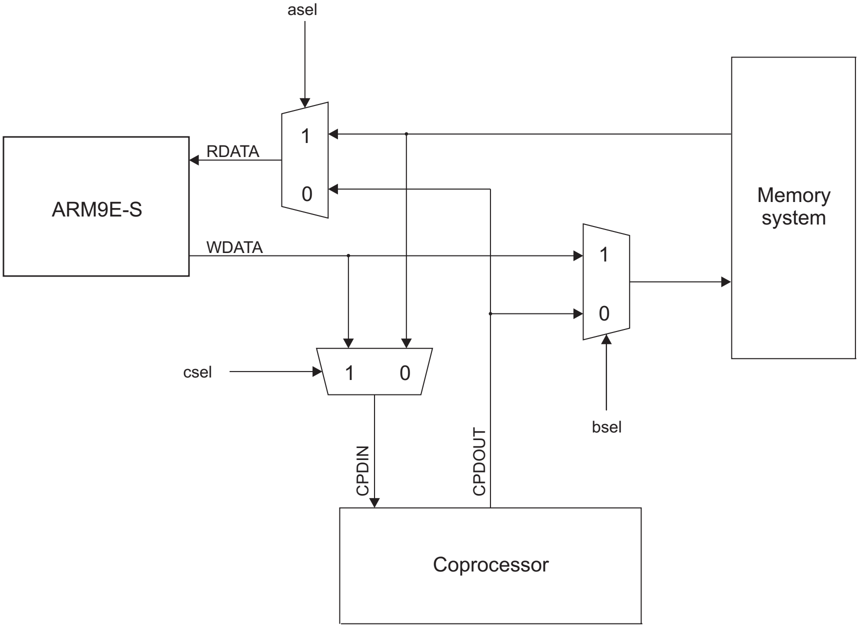
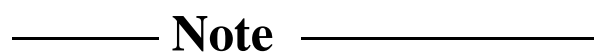


Figure 6-11 Coprocessor connections

The logic for Figure 6-11 is as follows:

```
on RISING CLK
    asel = not (DnMREQ and DSEQ) and (not DnRW)
    bsel = (not DnMREQ) and (not PASS)
    csel = DnMREQ and DSEQ
```

 Copyright © 2000 ARM Limited. All rights reserved.



Note

The **RDATA** enable term (asel) is specially constructed to select the coprocessor output data during `MRC` and `STC` operations. This is to allow the connection of the ETM module to the ARM9E-S **RDATA** and **WDATA** buses while still allowing tracing of `MRC` and `STC` data.

6.11.2 Connecting multiple coprocessors

If you have multiple coprocessors in your system, connect the handshake signals as shown in Table 6-2.

Table 6-2 Handshake signal connections

Signal	Connection
PASS, **LATECANCEL**	Connect these signals to all coprocessors present in the system.
CHSD, **CHSE**	Combine the individual bit 1 of **CHSD**, and **CHSE** by ANDing. Combine the individual bit 0 of **CHSD**, and **CHSE** by ORing. Connect the **CHSD**, and **CHSE** inputs to the ARM9E-S.

You must also multiplex the output data from the coprocessors.

The handshaking arrangement for a two-coprocessor system is shown in Example 6-1.

Example 6-1

In the case of two coprocessors that have handshaking signals **CHSD1**, and **CHSE1**, and **CHSD2**, and **CHSE2**, respectively, the following connections are made:

```
ARM9E-S      CP1          CP2
CHSD[1]<= CHSD1[1] ANDCHSD2[1]
CHSD[0]<= CHSD1[0] OR CHSD2[0]
CHSE[1]<= CHSE1[1] ANDCHSE2[1]
CHSE[0]<= CHSE1[0] OR CHSE2[0]
```


Copyright © 2000 ARM Limited. All rights reserved.

6.11.3 No external coprocessor

If you are implementing a system that does not include any external coprocessors, you must tie both **CHSD** and **CHSE** to 10 (ABSENT). This indicates that no external coprocessors are present in the system. If any coprocessor instructions are received, they cause the processor to take the undefined instruction trap, allowing the coprocessor instructions to be emulated in software if required.

The coprocessor-specific outputs from the ARM9E-S must be left unconnected:

- **PASS**
- **LATECANCEL**

6.11.4 Undefined instructions

The ARM9E-S implements full ARMv5TE architecture undefined instruction handling. This means that any instruction defined in the *ARM Architecture Reference Manual* as `UNDEFINED`, automatically causes the ARM9E-S to take the undefined instruction trap. Any coprocessor instruction that is not accepted by a coprocessor also results in the ARM9E-S taking the undefined instruction trap.

Copyright © 2000 ARM Limited. All rights reserved.

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 7
Debug Interface and EmbeddedICE-RT

This chapter describes the ARM9E-S debug interface in the following sections:

- *About the debug interface* on page 7-2
- *Debug systems* on page 7-3
- *Debug interface signals* on page 7-9
- *ARM9E-S core clock domains* on page 7-14
- *Determining the core and system state* on page 7-15.

This chapter also describes the ARM9E-S EmbeddedICE-RT logic in the following sections:

- *About EmbeddedICE-RT* on page 7-6
- *Disabling EmbeddedICE-RT* on page 7-8
- *The debug communications channel* on page 7-16
- *Monitor mode debug* on page 7-21.

Copyright © 2000 ARM Limited. All rights reserved.

7.1 About the debug interface

The ARM9E-S debug interface is based on IEEE Std. 1149.1-1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM9E-S contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself. ARM9E-S supports two modes of debug operation:

- *Halt mode*
- *Monitor mode*.

7.1.1 Halt mode

In halt mode debug, the debug extensions allow the core to be forced into *debug state*. In debug state, the core is stopped and isolated from the rest of the system. This allows the internal state of the core, and the external state of the system, to be examined while all other system activity continues as normal. When debug has been completed, the core and system state can be restored, and program execution resumed.

7.1.2 Monitor mode

On a breakpoint or watchpoint, an Instruction Abort or Data Abort is generated instead of entering halt mode debug. When used in conjunction with a debug monitor program activated by the abort exception entry, it is possible to debug the ARM9E-S while allowing the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the ARM9E-S debug communication channel. Monitor mode debug is described in *Monitor mode debug* on page 7-21.

Copyright © 2000 ARM Limited. All rights reserved.

7.2 Debug systems

The ARM9E-S forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM9E-S. Figure 7-1 shows a typical debug system.

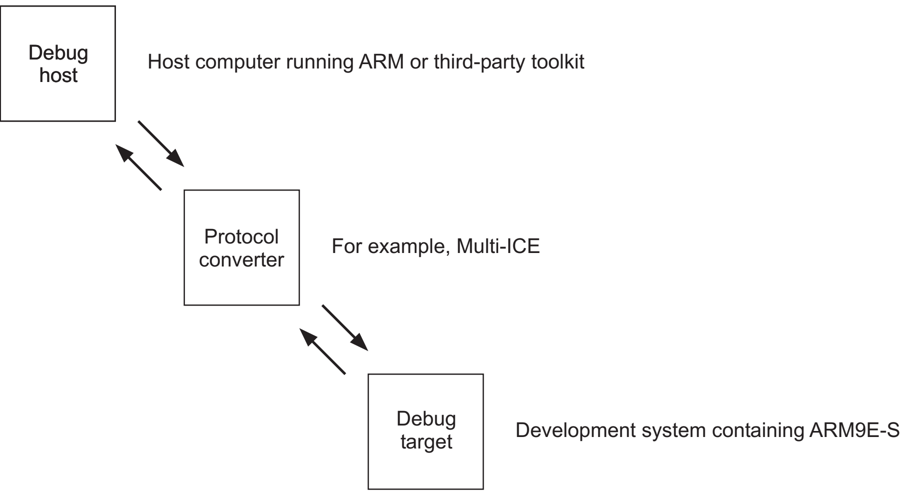


Figure 7-1 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter* on page 7-4
- *The ARM9E-S* on page 7-4 (the debug target).

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

7.2.1 The debug host

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

 Copyright © 2000 ARM Limited. All rights reserved.

7.2.2 The protocol converter

An interface, such as an RS232 or parallel connection, connects the debug host to the ARM9E-S development system. The messages broadcast over this connection must be converted to the interface signals of the ARM9E-S. The protocol converter performs this conversion.

7.2.3 The ARM9E-S

The ARM9E-S has hardware extensions that ease debugging at the lowest level. The debug extensions:

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

The major blocks of the ARM9E-S are:

ARM9E-S core This is the CPU core, with hardware support for debug.

EmbeddedICE-RT logic

This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in *About EmbeddedICE-RT* on page 7-6.

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

These blocks are shown in Figure 7-2 on page 7-5.

Copyright © 2000 ARM Limited. All rights reserved.

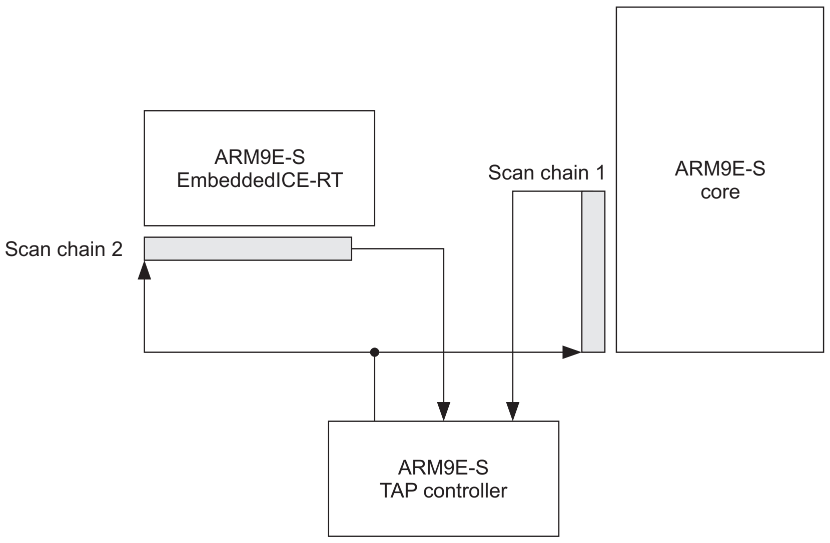


Figure 7-2 ARM9E-S block diagram

In halt mode debug a request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE-RT logic*, forces the ARM9E-S into debug state. The events that activate debug are:

- a breakpoint (a given instruction fetch)
- a watchpoint (a data access)
- an external debug request
- scanned debug request (a debug request scanned into the EmbeddedICE-RT delay control register).

The internal state of the ARM9E-S is examined using the JTAG serial interface, that allows instructions to be serially inserted into the core pipeline without using the external data bus. So, for example, when in debug state, a *store multiple* (`STM`) can be inserted into the instruction pipeline, and this exports the contents of the ARM9E-S registers. This data can be serially shifted out without affecting the rest of the system.

Copyright © 2000 ARM Limited. All rights reserved.

7.3 About EmbeddedICE-RT

The ARM9E-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9E-S core.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 7-3 shows the relationship between the core, EmbeddedICE-RT, and the TAP controller. It only shows the signals that are pertinent to EmbeddedICE-RT.

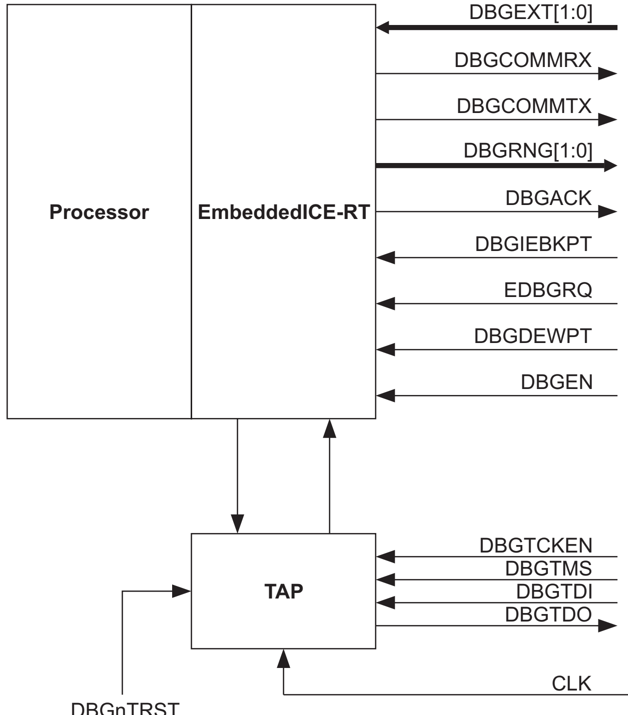


Figure 7-3 The ARM9E-S, TAP controller, and EmbeddedICE-RT

The EmbeddedICE-RT logic comprises:

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

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

 Copyright © 2000 ARM Limited. All rights reserved.

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

Note

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

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

The EmbeddedICE-RT logic can be configured into a mode of operation where watchpoints or breakpoints generate Data or Prefetch Aborts respectively. This allows a *Real-Time* (RT) debug monitor system to debug the ARM while still allowing critical fast interrupt requests to be serviced.

Copyright © 2000 ARM Limited. All rights reserved.

7.4 Disabling EmbeddedICE-RT

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

——— **Caution** ———

Hard wiring the **DBGEN** input LOW *permanently* disables all debug functionality.

———————————

When **DBGEN** is LOW, it inhibits **DBGDEWPT**, **DBGIEBKPT**, and **EDBGRQ** to the core, and **DBGACK** from the ARM9E-S is always LOW.

 Copyright © 2000 ARM Limited. All rights reserved.

7.5 Debug interface signals

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

- **DBGIEBKPT**, **DBGDEWPT**, and **EDBGRQ** are system requests for the ARM9E-S to enter debug state
- **DBGACK** is used by the ARM9E-S to flag back to the system that it is in debug state.

7.5.1 Entry into debug state on breakpoint

An instruction being fetched from memory is sampled at the end of a cycle. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the same cycle. This is shown in Figure 7-4.

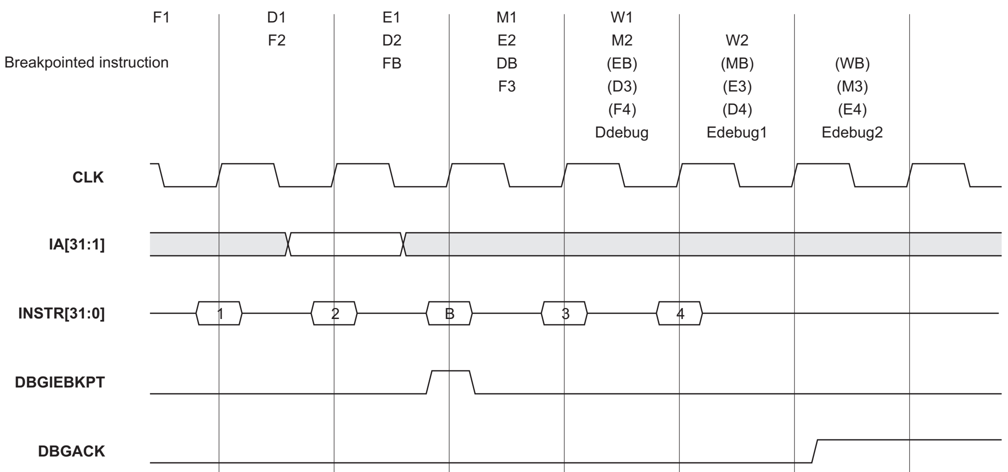


Figure 7-4 Breakpoint timing

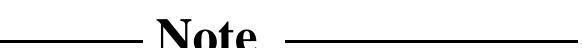
You can build external logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. You must apply their output to the **DBGIEBKPT** input. This signal is ORed with the internally-generated **Breakpoint** signal before being applied to the ARM9E-S core control logic.

Note

The timing of the **DBGIEBKPT** input makes it unlikely that data-dependent external breakpoints are possible.

 Copyright © 2000 ARM Limited. All rights reserved.

A breakpointed instruction is allowed to enter the Execute stage of the pipeline, but any state change as a result of the instruction is prevented. All instructions prior to the breakpointed instruction complete as normal.



Note

If a breakpointed instruction does not reach the Execute stage, for instance, if an earlier instruction is a branch, then both the breakpointed instruction and breakpoint status are discarded and the ARM does not enter debug state.

The Decode cycle of the debug entry sequence occurs during the execute cycle of the breakpointed instruction. The latched **Breakpoint** signal forces the processor to start the debug sequence.

In Figure 7-4 on page 7-9 instruction B is breakpointed. The debug entry sequence is initiated when instruction B enters the Execute stage. The ARM completes the debug entry sequence and asserts **DBGACK** two cycles later.

7.5.2 Breakpoints and exceptions

A breakpointed instruction can have a Prefetch Abort associated with it. If so, the Prefetch Abort takes priority and the breakpoint is ignored. (If there is a Prefetch Abort, instruction data might be invalid, the breakpoint might have been data-dependent, and as the data might be incorrect, the breakpoint might have been triggered incorrectly.)

`SWI` and undefined instructions are treated in the same way as any other instruction that can have a breakpoint set on it. Therefore, the breakpoint takes priority over the `SWI` or undefined instruction.

On an instruction boundary, if there is a breakpointed instruction and an interrupt (**nIRQ** or **nFIQ**), the interrupt is taken and the breakpointed instruction is discarded. When the interrupt has been serviced, the execution flow is returned to the original program. This means that the instruction which was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters debug state when it reaches the execute stage of the pipeline.

When the processor has entered debug state, it is important that further interrupts do not affect the instructions executed. For this reason, as soon as the processor enters debug state, interrupts are disabled, although the state of the I and F bits in the *Program Status Register* (PSR) are not affected.

Copyright © 2000 ARM Limited. All rights reserved.

7.5.3 Watchpoints

Entry into debug state following a watchpointed memory access is imprecise. This is necessary because of the nature of the pipeline.

You can build external logic, such as external watchpoint comparators, to extend the functionality of the EmbeddedICE-RT logic. You must apply their output to the **DBGDEWPT** input. This signal is ORed with the internally-generated **Watchpoint** signal before being applied to the ARM9E-S core control logic.

Note

The timing of the **DBGDEWPT** input makes it unlikely that data-dependent external watchpoints are possible.

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Where this instruction is a single-cycle data-processing instruction, entry into debug state is delayed for one cycle while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 7-5.

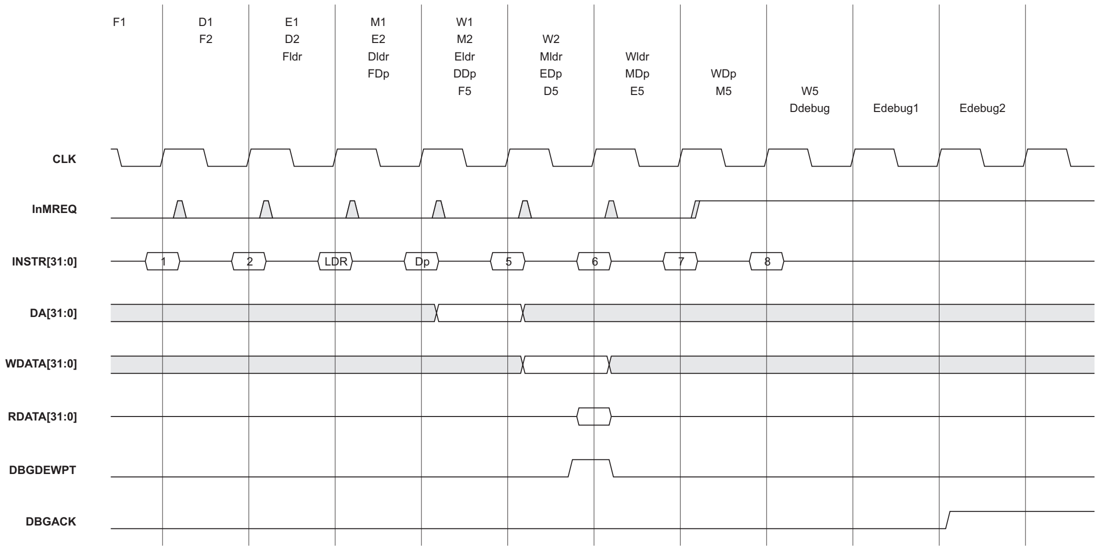


Figure 7-5 Watchpoint entry with data processing instruction

Copyright © 2000 ARM Limited. All rights reserved.

Although instruction 5 enters the Execute stage, it is not executed, and there is no state update as a result of this instruction.

Once the debugging session is complete, normal continuation involves a return to instruction 5, the next instruction in the code sequence which has not yet been executed.

The instruction following the instruction that generated the watchpoint might have modified the *Program Counter* (PC). If this happens, it is not possible to determine the instruction that caused the watchpoint. A timing diagram showing debug entry after a watchpoint where the next instruction is a branch is shown in Figure 7-6.

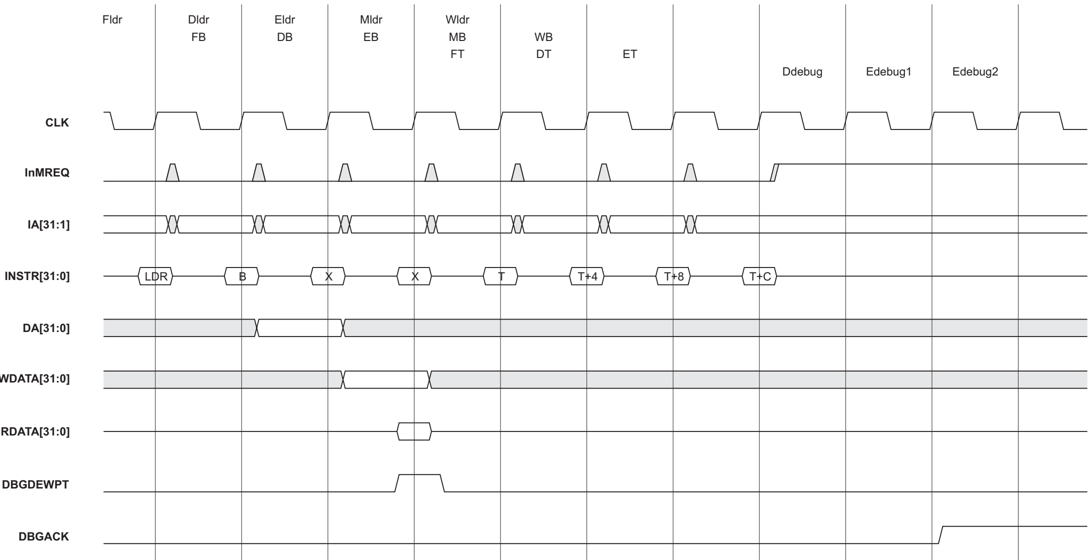


Figure 7-6 Watchpoint entry with branch

You can always restart the processor. When the processor has entered debug state, the ARM9E-S core can be interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is five instructions on from the address of the next instruction to be executed. Therefore, if on entry to debug state, in ARM state, the instruction `SUB PC, PC, #20` is scanned in and the processor restarted, execution flow returns to the next instruction in the code sequence.

 Copyright © 2000 ARM Limited. All rights reserved.

7.5.4 Watchpoints and exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence is performed, and then the processor enters debug state. If there is an interrupt pending, the ARM9E-S allows the exception entry sequence to occur and then enters debug state.

7.5.5 Debug request

A debug request can take place through the EmbeddedICE-RT logic or by asserting the **EDBGRQ** signal. The request is registered and passed to the processor. Debug request takes priority over any pending interrupt. Following registering, the core enters debug state when the instruction at the Execute stage of the pipeline has completely finished executing (once Memory and Write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions are issued to the Execute stage of the pipeline.

When a debug request occurs, the ARM9E-S enters debug state even if the EmbeddedICE-RT is configured for monitor mode debug.

7.5.6 Actions of the ARM9E-S in debug state

Once the ARM9E-S is in debug state, both memory interfaces indicate internal cycles. This allows the rest of the memory system to ignore the ARM9E-S and function as normal. Because the rest of the system continues operation, the ARM9E-S ignores aborts and interrupts.

The **CFGBIGEND** signal must not be changed by the system while in debug state. If it changes, not only is there a synchronization problem, but the view of the ARM9E-S seen by the programmer changes without the knowledge of the debugger. The **nRESET** signal must also be held stable during debug. If the system applies reset to the ARM9E-S (**nRESET** is driven LOW), the state of the ARM9E-S changes without the knowledge of the debugger.

 Copyright © 2000 ARM Limited. All rights reserved.

7.6 ARM9E-S core clock domains

The ARM9E-S has a single clock, **CLK**, that is qualified by two clock enables:

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

During normal operation, **CLKEN** conditions **CLK** to clock the core. When the ARM9E-S is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

7.6.1 Clocks and synchronization

If the system and test clocks are asynchronous, they must be synchronized externally to the ARM9E-S macrocell. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM9E-S macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal, and waits for the **RTCK** (Returned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received. Figure 7-7 shows this synchronization.

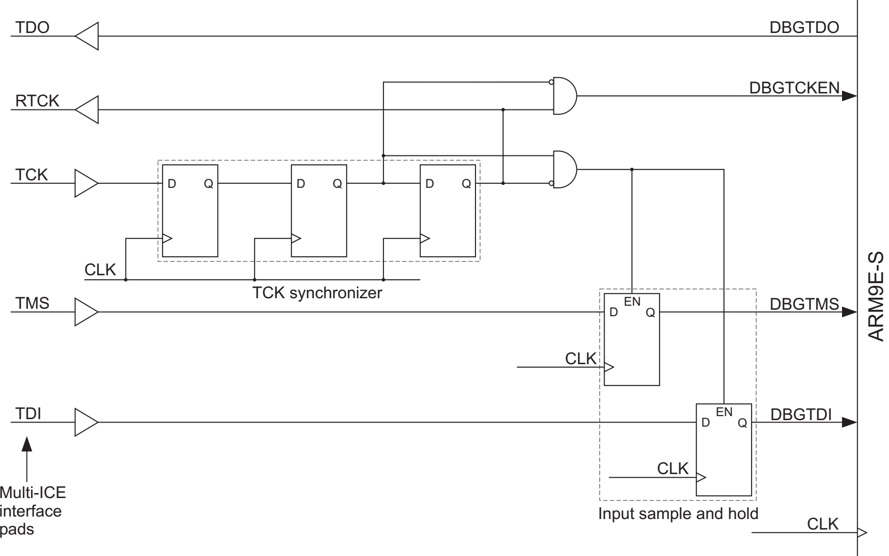


Figure 7-7 Clock synchronization

Copyright © 2000 ARM Limited. All rights reserved.

7.7 Determining the core and system state

When the ARM9E-S is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

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

For more details about determining the core state, see *Determining the core and system state* on page C-18.

Copyright © 2000 ARM Limited. All rights reserved.

7.8 The debug communications channel

The ARM9E-S EmbeddedICE-RT logic contains a communications channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel comprises:

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

These registers are located in fixed locations in the EmbeddedICE-RT logic register map (as shown in *EmbeddedICE-RT logic* on page C-28) and are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

In addition to the comms channel registers, the processor can access a 1-bit debug status register for use in the monitor mode debug configuration.

7.8.1 Debug comms channel registers

Coprocessor 14 contains 4 registers, allocated as shown in Table 7-1.

Table 7-1 Coprocessor 14 register map

Register name	Register number	Notes
Comms channel control	C0	Read only[a]
Comms channel data read	C1	For reads
Comms channel data write	C1	For writes
Comms channel monitor mode debug status	C2	Read/write

a. You can clear bit 0 of the comms channel control register by writing to it from the debugger (JTAG) side.

Seen from the debugger, the registers are accessed using the scan chain in the usual way. Seen from the processor, these registers are accessed using coprocessor register transfer instructions.

 Copyright © 2000 ARM Limited. All rights reserved.

7.8.2 Debug comms channel control register

The debug comms channel control register is read-only.[1] The register controls synchronized handshaking between the processor and the debugger. The debug comms channel control register is shown in Figure 7-8.

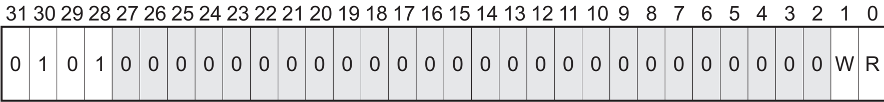


Figure 7-8 Debug comms channel control register

The function of each register bit is described below:

Bits 31:28 Contain a fixed pattern that denotes the EmbeddedICE version number (in this case 0011).

Bits 27:2 Are reserved.

Bit 1 Denotes if the comms data write register is available (from the viewpoint of the processor). Seen from the processor, if the comms data write register is free (W=0), new data can be written.

If the register is not free (W=1), the processor must poll until W=0.

Seen from the debugger, when W=1, some new data has been written that can then be scanned out.

Bit 0 Denotes if there is new data in the comms data read register. Seen from the processor, if R=1, there is some new data that can be read using an `MRC` instruction.

Seen from the debugger, if R=0, the comms data read register is free, and new data may be placed there through the scan chain. If R=1, this denotes that data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

1. The control register should be viewed as read-only. However, the debugger can clear the R bit by performing a write to the debug comms channel control register. This feature must not be used under normal circumstances.

 Copyright © 2000 ARM Limited. All rights reserved.

You can use the following instructions to access these registers:

```
MRC p14, 0, Rd, c0, c0
```

This returns the debug comms control register into Rd.

```
MCR p14, 0, Rn, c1, c0
```

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

```
MRC p14, 0, Rd, c1, c0
```

This returns the debug data read register into Rd.

Note

The Thumb instruction set does not support coprocessor instructions. Therefore, the processor must be in ARM state before you can access the debug comms channel.

7.8.3 Comms channel monitor mode debug status register

The coprocessor 14 monitor mode debug status register is provided for use by a debug monitor when the ARM9E-S is configured into the monitor mode debug mode.

The coprocessor 14 monitor mode debug status register is a 1-bit wide read/write register having the format shown in Figure 7-9.

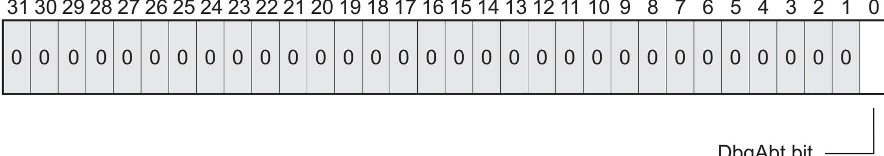


Figure 7-9 Coprocessor 14 monitor mode debug status register format

Bit 0 of the register, the DbgAbt bit, indicates whether the processor took a Prefetch or Data Abort in the past because of a breakpoint or watchpoint. If the ARM9E-S core takes a Prefetch Abort as a result of a breakpoint or watchpoint, then the bit is set. If on a particular instruction or data fetch, both the debug abort and external abort signals are asserted, the external abort takes priority and the DbgAbt bit is not set. You can read or write the DbgAbt bit using `MRC` or `MCR` instructions.

 Copyright © 2000 ARM Limited. All rights reserved.

A typical use of this bit is by a monitor mode debug aware abort handler. This examines the DbgAbt bit to determine whether the abort was externally or internally generated. If the DbgAbt bit is set, the abort handler initiates communication with the debugger over the comms channel.

7.8.4 Communications using the comms channel

You can send and receive messages using the comms channel. These are described in:

- *Sending a message to the debugger*
- *Receiving a message from the debugger* on page 7-20.

Sending a message to the debugger

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

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

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

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

The debugger has two options available for reading data from the comms data write register:

- Poll the debug comms channel control register before reading the comms data written. If the W bit is set, there is valid data present in the debug comms data write register. The debugger can then read this data and scan the data out. The action of reading the data clears the debug comms channel control register W bit. Then the communications process can begin again.
- Poll the comms data write register, obtaining data and valid status. The data scanned out consists of the contents of the comms data write register (which might or might not be valid), and a flag that indicates whether the data read is valid or not. The status flag is present in the Addr[0] bit position of scan chain 2 when the data is scanned out. See *Test data registers* on page C-10 for details of scan chain 2.

 Copyright © 2000 ARM Limited. All rights reserved.

Receiving a message from the debugger

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

- If the R bit is LOW, the comms data read register is free, and data can be placed there for the processor to read.
- If the R bit is set, previously deposited data has not yet been collected, so the debugger must wait.

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

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

Copyright © 2000 ARM Limited. All rights reserved.

7.9 Monitor mode debug

ARM9E-S contains logic that allows the debugging of a system without stopping the core entirely. This allows the continued servicing of critical interrupt routines while the core is being interrogated by the debugger. Setting bit 4 of the debug control register enables the monitor mode debug features of ARM9E-S. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM to enter abort mode, taking the Prefetch or Data Abort vectors respectively. There are a number of restrictions you must be aware of when the ARM is configured for monitor mode debugging:

- Breakpoints and watchpoints cannot be data-dependent. No support is provided for use of the range functionality. Breakpoints and watchpoints can only be based on:
  - instruction or data addresses
  - external watchpoint conditioner (**DBGEXTERN**)
  - User or Privileged mode access (**DnTRANS/InTRANS**)
  - read/write access (watchpoints)
  - access size (breakpoints **ITBIT**, watchpoints **DMAS[1:0]**)
  - chained comparisons.
- The single-step hardware must not be enabled.
- External breakpoints or watchpoints are not supported.
- The vector catching hardware can be used but must not be configured to catch the Prefetch or Data Abort exceptions.
- No support is provided to mix halt mode debug and monitor mode debug functionality.

The fact that an abort has been generated by the monitor mode is recorded in the monitor mode debug status register in coprocessor 14 (see *Comms channel monitor mode debug status register* on page 7-18).

Because the monitor mode debug bit does not put the ARM9E-S into debug state, it now becomes necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than being changed when in debug state. In the event that the watchpoint registers are written to during an access, all matches from the affected watchpoint unit using the register being updated are disabled for the cycle of the update.

Copyright © 2000 ARM Limited. All rights reserved.

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

1. Disable the watchpoint unit using the control register for that watchpoint unit.
2. Change the other registers.
3. Re-enable the watchpoint unit by rewriting the control register.

Copyright © 2000 ARM Limited. All rights reserved.

Chapter 8
Instruction Cycle Times

This chapter gives the instruction cycle timings and illustrates interlock conditions present in the ARM9E-S design. It contains the following sections:

- *Instruction cycle count summary* on page 8-3
- *Introduction to detailed instruction cycle timings* on page 8-7
- *Branch and ARM branch with link* on page 8-8
- *Thumb branch with link* on page 8-9
- *Branch and exchange* on page 8-10
- *Thumb Branch, Link, and Exchange <immediate>* on page 8-11
- *Data operations* on page 8-12
- *MRS* on page 8-14
- *MSR operations* on page 8-15
- *Multiply and multiply accumulate* on page 8-16
- *QADD, QDADD, QSUB, and QDSUB* on page 8-20
- *Load register* on page 8-21
- *Store register* on page 8-26
- *Load multiple registers* on page 8-27
- *Store multiple registers* on page 8-30
- *Load double register* on page 8-31

 Copyright © 2000 ARM Limited. All rights reserved.

- *Store double register* on page 8-32
- *Data swap* on page 8-33
- *PLD* on page 8-35
- *Software interrupt, undefined instruction, and exception entry* on page 8-36
- *Coprocessor data processing operation* on page 8-37
- *Load coprocessor register (from memory)* on page 8-38
- *Store coprocessor register (to memory)* on page 8-40
- *Coprocessor register transfer (to ARM)* on page 8-42
- *Coprocessor register transfer (from ARM)* on page 8-43
- *Double coprocessor register transfer (to ARM)* on page 8-44
- *Double coprocessor register transfer (from ARM)* on page 8-45
- *Coprocessor absent* on page 8-46
- *Unexecuted instructions* on page 8-47.

 Copyright © 2000 ARM Limited. All rights reserved.

8.1 Instruction cycle count summary

Table 8-1 shows the key to the other tables in this chapter.

Table 8-1 Key to tables

Symbol	Meaning
b	The number of busy-wait states during coprocessor accesses.
n	The number of words transferred in an LDM/STM/LDC/STC.
C	Coprocessor register transfer cycle (C-cycle).
I	Internal cycle (I-cycle).
N	Nonsequential cycle (N-cycle).
S	Sequential cycle (S-cycle).

Table 8-2 summarizes the ARM9E-S instruction cycle counts and bus activity when executing the ARM instruction set.

Table 8-2 ARM instruction cycle counts

Instruction	Cycles	Instruction bus	Data bus	Comment
CLZ	1	1S	1I	All cases.
Data Op	1	1S	1I	Normal case, PC not destination.
Data Op	2	1S+1I	2I	With register controlled shift, PC not destination.
Data Op	3	2S+1N	3I	PC destination register.
Data Op	4	2S+1N+1I	4I	With register controlled shift, PC destination register.
LDR	1	1S	1N	Normal case, not loading PC.
LDR	2	1S+1I	1N+1I	Not loading PC and following instruction uses loaded word (1 cycle load-use interlock).
LDR	3	1S+2I	1N+2I	Loaded byte, halfword, or unaligned word used by following instruction (2-cycle load-use interlock).
LDR	5	2S+2I+1N	1N+4I	PC is destination register.

 Copyright © 2000 ARM Limited. All rights reserved.

Table 8-2 ARM instruction cycle counts (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
`LDRD`	2	1S+1I	1N+1S	Normal case.
`LDRD`	3	1S+2I	1N+1S+1I	Last loaded word used by following instruction.
`STR`	1	1S	1N	All cases.
`STRD`	2	1S+1I	1N+1S	All cases.
`LDM`	2	1S+1I	1S+1I	Loading 1 register, not the PC.
`LDM`	n	1S+(n-1)I	1N+(n-1)S	Loading n registers, n > 1, not loading the PC.
`LDM`	n+1	1S+nI	1N+(n-1)S+1I	Loading n registers, n > 1, not loading the PC, last word loaded used by following instruction.
`LDM`	n+4	2S+1N+(n+1)I	1N+(n-1)S+4I	Loading n registers including the PC, n > 0.
`LDM`	5	2S+2I+1N	1N+4I	Load PC.
`STM`	2	1S+1I	1N+1I	Storing 1 register.
`STM`	n	1S+(n-1)I	1N+(n-1)S	Storing n registers, n > 1.
`SWP`	2	1S+1I	2N	Normal case.
`SWP`	3	1S+2I	2N+1I	Loaded word used by following instruction.
`PLD`	1	1S	1I	All cases, **DnSPEC** asserted.
`B, BL, BX, BLX`	3	2S+1N	3I	All cases.
`SWI`, Undefined	3	2S+1N	3I	All cases.
Coprocessor absent	b+4	2S+1N+1I+bI	4I+bI	All cases.
`CDP`	b+1	1S+bI	(1+b)I	All cases.
`LDC`, `STC`	b+n	1S+(b+n-1)I	bI+1N+(n-1)S	All cases.
`MCR`	b+1	1S+bI	bI+1C	All cases.
`MCRR`	b+2	1S+(b+1)I	bI+2C	All cases.
`MRC`	b+1	1S+bI	bI+1C	Normal case.
`MRC`	b+2	1S+(b+1)I	(b+1)I+1C	Following instruction uses transferred data.

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-2 ARM instruction cycle counts (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
MRC (dest = PC)	b+4	1S+(b+3)I	(b+3)I+1C	Destination is PC.
MRRC	b+2	1S+(b+1)I	bI+2C	Normal case.
MRRC	b+3	1S+(b+2)I	(b+1)I+2C	Following instruction uses last transferred data.
MRS	2	1S+1I	2I	All cases.
MSR	1	1S	1I	If only flags are updated (mask_f).
MSR	3	1S+2I	3I	If any bits other than just the flags are updated (all masks other than mask_f).
MUL, MLA	2	1S+1I	2I	Normal case.
MUL, MLA	3	1S+2I	3I	Following instruction uses the result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
MULS, MLAS	4	1S+3I	4I	All cases, sets flags.
QADD, QDADD, QSUB, QDSUB	1	1S	1I	Normal case.
QADD, QDADD, QSUB, QDSUB	2	1S+1I	2I	Following instruction uses the result in its first Execute cycle.
SMULL, UMULL, SMLAL, UMLAL	3	1S+2I	3I	Normal case.
SMULL, UMULL, SMLAL, UMLAL	4	1S+3I	4I	Following instruction uses RdHi result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
SMULLS, UMULLS, SMLALS, UMLALS	5	1S+4I	5I	All cases, sets flags.
SMULxy, SMLAxy	1	1S	1I	Normal case.
SMULxy, SMLAxy	2	1S+1I	2I	Following instruction uses the result in its first Execute or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-2 ARM instruction cycle counts (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
`SMULWx`, `SMLAWx`	1	1S	1I	Normal case.
`SMULWx`, `SMLAWx`	2	1S+1I	2I	Following instruction uses the result in its first Execute or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
`SMLALxy`	2	1S+1I	2I	Normal case.
`SMLALxy`	3	1S+2I	3I	Following instruction uses RdHi result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.

Copyright © 2000 ARM Limited. All rights reserved.

8.2 Introduction to detailed instruction cycle timings

The pipelined architecture of ARM9E-S overlaps the execution of several instructions in different pipeline stages. The tables in this section show the number of cycles required by an instruction, once that instruction has reached the Execute stage of the pipeline. The instruction cycle count is the number of cycles that an instruction occupies the execute stage of the pipeline. The other pipeline stages (Fetch, Decode, Memory, Writeback) are only occupied for one cycle by any instruction (in this model, interlock cycles are grouped in with the instruction generating the data that creates the interlock condition, not the instruction dependent on the data).

The request, address, and control signals on both the instruction and data interfaces are pipelined so that they are generated in the cycle before the one to which they apply, and are shown as such in the following tables.

The instruction address, **IA[31:1]**, is incremented for prefetching instructions in most cases. The increment varies with the instruction length:

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

The letter i is used to indicate the instruction length.

Note

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

Table 8-3 shows the key to the cycle timing tables, Table 8-4 to Table 8-36.

Table 8-3 Key to cycle timing tables

Symbol	Meaning
pc	The address of the branch instruction.
pc’	The branch target address.
(pc’)	The memory contents of that address.
i	4 when in ARM state, or 2 when in Thumb state.
-	Indicates that the signal is not active, and therefore not valid in this cycle.
	A blank entry in the table indicates that the status of the signal is not determined by the instruction in that cycle. The status of the signal is determined either by the preceding or succeeding instruction.

Copyright © 2000 ARM Limited. All rights reserved.

8.3 Branch and ARM branch with link

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

1. During the first cycle, a branch instruction calculates the branch destination while performing a prefetch from the current PC. This prefetch is performed in all case, because by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch. If the previous instruction requested a data memory access, the data is transferred in this cycle.
2. During the second cycle, the ARM9E-S performs a fetch from the branch destination. If the link bit is set, the return address to be stored in r14 is calculated.
3. During the third cycle, the ARM9E-S performs a fetch from the destination + i, refilling the instruction pipeline.

Table 8-4 Branch and ARM branch with link cycle timings

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc’	N cycle	(pc + 2i)	-	I cycle	
2	pc’ + i	S cycle	(pc’)	-	I cycle	-
3	pc’ + 2i	S cycle	(pc’ + i)	-	I cycle	-
			(pc’ + 2i)			-

Copyright © 2000 ARM Limited. All rights reserved.

8.4 Thumb branch with link

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

1. The first instruction acts as a simple data operation. It takes a single cycle to add the PC to the upper part of the offset, and stores the result in r14. If the previous instruction requested a data memory access, the data is transferred in this cycle.
2. The second instruction acts similarly to the ARM `BL` instruction over three cycles:
   a. During the first cycle, the ARM9E-S calculates the final branch target address while performing a prefetch from the current PC.
   b. During the second cycle, the ARM9E-S performs a fetch from the branch destination, while calculating the return address to be stored in r14.
   c. During the third cycle, the ARM9E-S performs a fetch from the destination + 2, refilling the instruction pipeline.

Table 8-5 shows the cycle timings of the complete operation.

Table 8-5 Thumb branch with link cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc+3i	S cycle	(pc+i)	-	I cycle	
2	pc’	N cycle	(pc+3i)	-	I cycle	-
3	pc’+i	S cycle	(pc’)	-	I cycle	-
4	pc’+i	S cycle	(pc’+i)	-	I cycle	-
			(pc’+i)			-

Copyright © 2000 ARM Limited. All rights reserved.

8.5 Branch and exchange

A Branch and Exchange (`BX`), Branch, Link and Exchange register (`BLX <register>`) or ARM `BLX <immediate>` operation takes three cycles, and is similar to a Branch:

1. During the first cycle, the ARM9E-S extracts the branch destination and the new core state while performing a prefetch from the current PC. This prefetch is performed in all cases, because by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch. In the case of `BX` and `BLX<register>`, the branch destination new state comes from the register. For `BLX<immediate>` the destination is calculated as a PC offset. The state is always changed. If the previous instruction requested a memory access (and there is no interlock in the case of `BX`, `BLX <register>`), the data is transferred in this cycle.
2. During the second cycle, the ARM9E-S performs a fetch from the branch destination, using the new instruction width, dependent on the state that has been selected. If the link bit is set, the return address to be stored in r14 is calculated.
3. During the third cycle, the ARM9E-S performs a fetch from the destination +2 or +4 dependent on the new specified state, refilling the instruction pipeline.

Table 8-6 shows the cycle timings, where:

i Is the instruction width before the `BX/BLX` instruction.

i' Is the instruction width after the `BX/BLX` instruction.

t' Is the state of the **ITBIT** signal after the `BX/BLX` instruction.

Table 8-6 Branch and exchange cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	ITBIT	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc'	N cycle	(pc + 2i)	t'	-	I cycle	
2	pc' + i'	S cycle	(pc')	t'	-	I cycle	-
3	pc' + 2i'	S cycle	(pc' + i')	t'	-	I cycle	-
			(pc' + 2i')				-

Copyright © 2000 ARM Limited. All rights reserved.

8.6 Thumb Branch, Link, and Exchange <immediate>

A Thumb Branch, Link, and Exchange immediate (`BLX <immediate>`) operation is similar to a Thumb `BL` operation. It comprises two consecutive Thumb instructions, and takes four cycles:

1. The first instruction acts as a simple data operation. It takes a single cycle to add the PC to the upper part of the offset, and stores the result in r14. If the previous instruction requested a data memory access, the data is transferred in this cycle.
2. The second instruction acts similarly to the ARM `BLX` instruction:
   a. During the first cycle, the ARM9E-S calculates the final branch target address while performing a prefetch from the current PC.
   b. During the second cycle, the ARM9E-S performs a fetch from the branch destination, using the new instruction width, dependent on the state that has been selected. The return address to be stored in r14 is calculated.
   c. During the third cycle, the ARM9E-S performs a fetch from the destination + 4, refilling the instruction pipeline.

Table 8-7 shows the cycle timings of the complete operation.

Table 8-7 Thumb branch, link and exchange cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	ITBIT	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc+3i	S cycle	(pc+2i)	t	-	I cycle	
2	pc’	N cycle	(pc+3i)	t’	-	I cycle	-
3	pc’+i	S cycle	(pc’)	t’	-	I cycle	-
4	pc’+2i	S cycle	(pc’+i)	t’	-	I cycle	-
			(pc’+2i)				-

 Copyright © 2000 ARM Limited. All rights reserved.

8.7 Data operations

A normal data operation executes in a single execute cycle except where the shift is determined by the contents of a register. A normal data operation requires up to two operands, that are read from the register file onto the A and B buses.

The ALU combines the A bus operand with the (shifted) B bus operand according to the operation specified in the instruction. The ARM9E-S pipelines this result and writes it into the destination register, when required. Compare and test operations do not write a result as they only affect the status flags.

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

When a register specified shift is used, an additional execute cycle is needed to read the shifting register operand. The instruction prefetch occurs during this first cycle.

The PC can be one or more of the register operands. When the PC is the destination, the external bus activity is affected. When the ARM9E-S writes the result to the PC, the contents of the instruction pipeline are invalidated, and the ARM9E-S takes the address for the next instruction prefetch from the ALU rather than the incremented address. The ARM9E-S refills the instruction pipeline before any further instruction execution takes place. Exceptions are locked out while the pipeline is refilling.

Note

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

The data operation cycle timings are shown in Table 8-8.

Table 8-8 Data operation cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	S cycle	(pc+2i)	-	I cycle	
				(pc+3i)			-
dest=pc	1	pc’	N cycle	(pc+2i)	-	I cycle	
	2	pc’+ i	S cycle	(pc’)	-	I cycle	-
	3	pc’+2i	S cycle	(pc’+i)	-	I cycle	-
				(pc’+ 2i)			-

 Copyright © 2000 ARM Limited. All rights reserved.

Table 8-8 Data operation cycle timing (continued)

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
shift(Rs)	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-
shift (Rs) dest=pc	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc’	N cycle	-	-	I cycle	-
	3	pc’+i	S cycle	(pc’)	-	I cycle	-
	4	pc’+2i	S cycle	(pc’+i)	-	I cycle	-
				(pc’+2i)			-

Copyright © 2000 ARM Limited. All rights reserved.

8.8 MRS

An MRS operation always takes two cycles to execute. The first cycle allows any pending state changes to the PSR to be made. The second cycle passes the PSR register through the ALU so that it can be written to the destination register.

Note

The MRS instruction can only be executed when in ARM state.

Table 8-9 shows the MRS cycle timing.

Table 8-9 MRS cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc+3i	I cycle	(pc+2i)	-	I cycle	
2	pc+3i	S cycle	-	-	I cycle	-
			(pc+3i)			-

 Copyright © 2000 ARM Limited. All rights reserved.

8.9 MSR operations

An `MSR` operation takes one cycle to execute if it only updates the status flags of the CPSR, and three cycles if it updates other parts of the PSR.

Note

`MSR` instructions can only be executed in ARM state.

Table 8-10 shows the cycle timings for MSR operations.

Table 8-10 MSR cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
MSR flags	1	pc+3i	S cycle	(pc+2i)	-	I cycle	
				(pc+3i)			-
MSR other	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	I cycle	-	-	I cycle	-
	3	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-

Copyright © 2000 ARM Limited. All rights reserved.

8.10 Multiply and multiply accumulate

The multiply instructions make use of special hardware that implements integer multiplication. All cycles except the last are internal.

During the first (Execute) stage of a multiply instruction, the multiplier and multiplicand operands are read onto the A and B buses, on which the multiplier unit is connected. The first stage of the multiplier performs Booth recoding and partial product summation, using 16 bits of the multiplier operand each cycle.

During the second (Memory) stage of a multiply instruction, the partial product result from the Execute stage is added with an optional accumulate term (read onto the C bus) and a possible feedback term from a previous multiply step for multiplications which require additional cycles.

Note

In Thumb state, only the `MULS` and `MLAS` operations are possible.

8.10.1 Interlocks

The multiply unit in ARM9E-S operates in both the Execute and Memory stage of the pipeline. Because of this, the multiplier result is not available until the end of the Memory stage of the pipeline. If the following instruction requires the use of the multiplier result, then it must be interlocked so that the correct value is available. This applies to all instructions that require the multiply result for the first Execute cycle or first Memory cycle of the instruction except for multiply accumulate instructions using the previous multiply result as the accumulator operand.

As an example, the following sequence incurs a single-cycle interlock:

```
MUL     r0, r1, r2
SUB     r4, r0, r3
```

The following cycle also incurs a single-cycle interlock:

```
MLA     r0, r1, r2, r3
STR     r0, [r8]
```

The following example does not incur an interlock:

```
MLA     r0, r1, r2, r0
MLA     r0, r3, r4, r0
```


Copyright © 2000 ARM Limited. All rights reserved.

Table 8-11 shows the cycle timing for `MUL` and `MLA` instructions with and without interlocks.

Table 8-11 MUL and MLA cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-
Interlock	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	I cycle	-	-	I cycle	-
	3	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-

The `MULS` and `MLAS` instructions always take four cycles to execute, and cannot generate interlocks in following instructions.

Table 8-12 shows the cycle timing for `MULS` and `MLAS` instructions.

Table 8-12 MULS and MLAS cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc+3i	I cycle	(pc+2i)	-	I cycle	
2	pc+3i	I cycle	-	-	I cycle	-
3	pc+3i	I cycle	-	-	I cycle	-
4	pc+3i	S cycle	-	-	I cycle	-
			(pc+3i)			-

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-13 shows the cycle timing for `SMULL`, `UMULL`, `SMLAL`, and `UMLAL` instructions with and without interlocks.

Table 8-13 SMULL, UMULL, SMLAL, and UMLAL cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	I cycle	-	-	I cycle	-
	3	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-
Interlock	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	I cycle	-	-	I cycle	-
	3	pc+3i	I cycle	-	-	I cycle	-
	4	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-

The `SMULLS`, `UMULLS`, `SMLALS`, and `UMLALS` instructions always take five cycles to execute, and cannot generate interlocks in following instructions.

Table 8-14 shows the cycle timing for the `SMULLS`, `UMULLS`, `SMLALS`, and `UMLALS` instructions.

Table 8-14 SMULLS, UMULLS, SMLALS, and UMLALS cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc+3i	I cycle	(pc+2i)	-	I cycle	
2	pc+3i	I cycle	-	-	I cycle	-
3	pc+3i	I cycle	-	-	I cycle	-
4	pc+3i	I cycle	-	-	I cycle	-
5	pc+3i	S cycle	-	-	I cycle	-
			(pc+3i)			-

.

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-15 shows the cycle timing for SMULxy, SMLAxy, SMULWy, and SMLAWy instructions with and without interlocks.

Table 8-15 SMULxy, SMLAxy, SMULWy, and SMLAWy cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	S cycle	(pc+2i)	-	I cycle	
		b	b	(pc+3i)		b	-
Interlock	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-

Table 8-16 shows the cycle timing for SMLALxy instructions with and without interlocks.

Table 8-16 SMLALxy cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-
Interlock	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	I cycle	-	-	I cycle	
	3	pc+3i	S cycle	-	-	I cycle	
				(pc+3i)			-

 Copyright © 2000 ARM Limited. All rights reserved.

8.11 QADD, QDADD, QSUB, and QDSUB

This class of instructions normally takes one cycle to execute and is only available in ARM state.

8.11.1 Interlocks

The instructions in this class use both the Execute and Memory stages of the pipeline. Because of this, the result of an instruction in this class is not available until the end of the Memory stage of the pipeline. If a following instruction requires the use of the result, then it must be interlocked so that the correct value is available. This applies to all instructions that require the result for the first Execute cycle. Instructions that require the result of a QADD or similar instruction for the first Memory cycle do not incur an interlock.

As an example, the following sequence incurs a single-cycle interlock:

```
QADD    r0, r1, r2
SUB     r4, r0, r3
```

The following cycle does not incur a single-cycle interlock:

```
QDSUB   r0, r1, r2
STR     r0, [r8]
```

The following example does not incur an interlock:

```
QADD    r0, r4, r5
MLA     r0, r3, r4, r0
```

Table 8-17 shows the cycle timing for QADD, QDADD, QSUB, and QDSUB instructions with and without interlocks.

Table 8-17 QADD, QDADD, QSUB, and QDSUB cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
Normal	1	pc+3i	S cycle	(pc+2i)	-	I cycle	
				(pc+3i)		b	-
Interlock	1	pc+3i	I cycle	(pc+2i)	-	I cycle	
	2	pc+3i	S cycle	-	-	I cycle	
				(pc+3i)			-

 Copyright © 2000 ARM Limited. All rights reserved.

8.12 Load register

A load register operation typically occupies the Execute stage for one cycle. There might be a number of cycles before the loaded value is available for later instructions. A load to the PC occupies the Execute stage for five cycles.

Note

Destination equals PC is not possible in Thumb state.

8.12.1 Interlocks

The result of an aligned word load instruction is not available until the end of the Memory stage of the pipeline. If the following instruction requires the use of this result then it must be interlocked so that the correct value is available. This interlock is referred to as a single-cycle load-use interlock.

The following example incurs a single-cycle interlock:

```
LDR r0, [r1]
ADD r2, r0, r3
ORR r4, r4, r5
```

The following example does not incur an interlock:

```
LDR r0, [r1]
ORR r4, r4, r5
ADD r2, r0, r3
```

Unaligned word loads, load byte (`LDRB`), and load halfword (`LDRH`) instructions use the byte rotate unit in the Write stage of the pipeline. This introduces a two-cycle load-use interlock, that can affect the two instructions immediately following the load instruction.

The following example incurs a two-cycle interlock:

```
LDRB r0, [r1, #1]
ADD r2, r0, r3
ORR r4, r4, r5
```

The following example incurs a single-cycle interlock:

```
LDRB r0, [r1, #1]
ORR r4, r4, r5
ADD r2, r0, r3
```

Once an interlock has been incurred for one instruction it does not have to be incurred for a later instruction.

Copyright © 2000 ARM Limited. All rights reserved.

For example, the following sequence incurs a two-cycle interlock on the first ADD instruction, but the second ADD does not incur any interlocks:

```
LDRB    r0, [r1, #1]
ADD     r2, r0, r3
ADD     r4, r0, r5
```

A two-cycle interlock refers to the number of unwaited ARM9E-S clock cycles to which the interlock applies. If a multi-cycle instruction separates a load instruction and the instruction using the result of the load, then no interlock can apply. The following example does not incur an interlock:

```
LDRB    r0, [r1]
MUL     r6, r7, r8
ADD     r4, r0, r5
```

There is no forwarding path from loaded data to the C read port of the register bank, which is used for the store data of STR and STM instructions and for the accumulate operand of multiply accumulate instructions. The result of a load must reach the Write stage of the pipeline before it can be made available at the C read port, resulting in a single-cycle load-use interlock from loaded data to the C read port.

The following example incurs a single-cycle interlock:

```
LDR     r0, [r1]
STR     r0, [r2]
```

The following example also incurs a single-cycle interlock:

```
LDR     r0, [r1]
MLA     r2, r3, r4, r0
```

The following example does not incur an interlock:

```
LDR     r0, [r1]
NOP ** Code to be changed to remove NOP **
STR     r0, [r2]
```

Most interlock conditions are determined when the instruction being interlocked is still in the Decode stage of the pipeline. Load multiple and Store multiple instructions can incur a Decode stage interlock when the base register is not available due to a previous instruction. Store multiple instructions can also incur an Execute stage interlock when the first register to be stored is not available due to a previous instruction. This is referred to as a second-cycle interlock.

The following example incurs a single-cycle interlock:

```
LDR     r0, [r1]
STMIA   r0, {r1-r2}
```


Copyright © 2000 ARM Limited. All rights reserved.

The following example incurs a second-cycle interlock:

```
LDR      r0, [r1]
STMIA    r2, {r0-r1}
```

A second-cycle interlock can be incurred on the first word of data stored by an `STM` instruction or during the first cycle of a register controlled shift. The following example does not incur an interlock:

```
LDR      r3, [r1]
STMIA    r0, {r2-r3}
```

Table 8-18 shows the cycle timing for basic load register operations, where:

s Represents the current mode-dependent value.

t Is either 0, when the T bit is specified in the instruction (for example `LDRT`) or s at all other times.

Table 8-18 Load register operation cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	DnTRANS	RDATA
Normal	1	pc+3i	S cycle	(pc+2i)	da	N cycle	t	
				(pc+3i)				(da)
dest=pc	1	pc+3i	I cycle	(pc+2i)	da	N cycle	t	
	2	pc+3i	I cycle	-	-	I cycle	s	(da)
	3	pc’	N cycle	(pc+3i)	-	I cycle	s	-
	4	pc’+i	S cycle	(pc’)	-	I cycle	s	-
	5	pc’+2i	S cycle	(pc’+i)	-	I cycle	s	-
				(pc’+2i)				-

Note

Destination equals PC is not possible in Thumb state.

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-19 shows the cycle timing for load operations resulting in simple interlocks.

Table 8-19 Cycle timing for load operations resulting in interlocks

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA
Single-cycle interlock	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	S cycle	-	-	I cycle	(da)
				(pc+3i)			-
Two-cycle interlock	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	-	I cycle	(da)
	3	pc+3i	S cycle	-	-	I cycle	-
				(pc+3i)			-

With more complicated interlock cases you cannot consider the load instruction in isolation. This is because in these cases the load instruction has vacated the Execute stage of the pipeline and a later instruction has occupied it.

Table 8-20 shows the one-cycle interlock incurred for the following sequence of instructions:

```
LDRB    r0, [r1]
NOP
ADD     r2, r0, r1
```

Table 8-20 Example sequence LDRB, NOP and ADD cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA
LDRB r0, [r1]	1	pc+3i	S cycle	(pc+2i)	da	N cycle	
NOP	2	pc+4i	I cycle	(pc+3i)	-	I cycle	(da)
	3	pc+4i	S cycle	-	-	I cycle	-
ADD r2, r0, r1	4	pc+5i	S cycle	(pc+4i)	-	I cycle	-
				(pc+5i)			-

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-21 shows the cycle timing for the following code sequence:

```
LDRB    r0, [r2]
STMIA   r3, {r0-r1}
```

Table 8-21 Example sequence LDRB and STMIA cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA
LDRB r0, [r2]	1	pc+3i	S cycle	(pc+2i)	da	N cycle	
STMIA r3, {r0-r1}	2	pc+4i	I cycle	(pc+3i)	-	I cycle	(da)
	3	pc+4i	I cycle	-	r3	N cycle	-
	4	pc+4i	S cycle	-	r3+4	S cycle	r0
				(pc+4i)			r1

Copyright © 2000 ARM Limited. All rights reserved.

8.13 Store register

A store register operation executes in a single cycle. During the Execute cycle, the store address is calculated, and the data to be stored is read onto the C bus.

Table 8-22 shows the cycle timing for a store register operation, where:

s Represents the current mode-dependent value.

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

Table 8-22 Store register operation cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	DnTRANS	WDATA
1	pc+3i	S cycle	(pc+2i)	da	N cycle	t	
			(pc+3i)				Rd

 Copyright © 2000 ARM Limited. All rights reserved.

8.14 Load multiple registers

A load multiple (`LDM`) takes several cycles to execute, depending on the number of registers transferred and whether the PC is in the list of registers transferred.

1. During the first cycle, the ARM9E-S calculates the address of the first word to be transferred, while performing an instruction prefetch.
2. During the second and subsequent cycles, ARM9E-S reads the data requested in the previous cycle and calculates the address of the next word to be transferred. The new value for the base register is calculated.

When a Data Abort occurs, the instruction continues to completion. The ARM9E-S prevents all register writing after the abort. The ARM9E-S restores the modified base pointer (which the load activity before the abort occurred might have overwritten).

When the PC is in the list of registers to be loaded, the ARM9E-S invalidates the current contents of the instruction pipeline. The PC is always the last register to be loaded, so an abort at any point prevents the PC from being overwritten.

Note

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

8.14.1 Interlocks

An `LDM` instruction can cause an interlock if a following instruction is dependent on the last data value transferred. This is similar to the interlock cases present with a single word register load. There is an exception to this case for a single-word `LDM` where, due to the presence of an idle cycle at the end of a single-word `LDM`, no interlock condition exists.

For example, the following sequence incurs a single-cycle interlock:

```
LDMIA r0, {r1-r2}
ADD  r3, r2, r4
```

The following sequence incurs a single-cycle interlock:

```
LDMIA r0, {r1-r2}
STR  r2, [r3]
```

The following sequence does not incur an interlock:

```
LDMIA r0, {r1}
STR  r1, [r2]
```


Copyright © 2000 ARM Limited. All rights reserved.

The LDM cycle timings are shown in Table 8-23.

Table 8-23 LDM cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA
1 register (not PC)	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	S cycle	-	-	I cycle	(da)
				(pc+3i)			-
n registers (n > 1) (not PC)	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	da++	S cycle	(da)
	.	pc+3i	I cycle	-	da++	S cycle	(da++)
	n	pc+3i	S cycle	-	da++	S cycle	(da++)
				(pc+3i)			(da++)
1 register dest=pc	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	-	I cycle	(da)
	3	pc’	N cycle	-	-	I cycle	-
	4	pc’+i	S cycle	(pc’)	-	I cycle	-
	5	pc’+2i	S cycle	(pc’+i)	-	I cycle	-
				(pc’+2i)			-
n registers (n > 1) (incl pc)	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	da++	S cycle	(da)
	.	pc+3i	I cycle	-	da++	S cycle	(da++)
	n	pc+3i	I cycle	-	da++	S cycle	(da++)
	n + 1	pc+3i	I cycle	-	-	I cycle	(da++)
	n + 2	pc’	N cycle	-	-	I cycle	-
	n + 3	pc’+i	S cycle	(pc’)	-	I cycle	-
	n + 4	pc’+2i	S cycle	(pc’+i)	-	I cycle	-
				(pc’+2i)			-

 Copyright © 2000 ARM Limited. All rights reserved.

Table 8-23 LDM cycle timing (continued)

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA
n registers (n > 1) (1 cycle interlock)	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	da++	S cycle	(da)
	.	pc+3i	I cycle	-	da++	S cycle	(da++)
	n	pc+3i	I cycle	-	da++	S cycle	(da++)
	n + 1	pc+3i	S cycle	-	-	I cycle	(da++)
				(pc+3i)			-

Copyright © 2000 ARM Limited. All rights reserved.

8.15 Store multiple registers

Store multiple (STM) instructions proceed in a similar fashion as load multiple instructions.

1. During the first cycle, the ARM9E-S calculates the address of the first word to be transferred, while performing an instruction prefetch and also calculating the new value for the base register.
2. During the second and subsequent cycles, ARM9E-S stores the data requested in the previous cycle and calculates the address of the next word to be transferred.

When a Data Abort occurs, the instruction continues to completion. The ARM9E-S restores the modified base pointer (which the load activity before the abort occurred might have overwritten).

The STM cycle timings are shown in Table 8-24.

Table 8-24 STM cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	WDATA
1 register	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	S cycle	-	-	I cycle	R
				(pc+3i)			-
n registers (n > 1)	1	pc+3i	I cycle	(pc+2i)	da	N cycle	
	2	pc+3i	I cycle	-	da++	S cycle	R
	.	pc+3i	I cycle	-	da++	S cycle	R’
	n	pc+3i	S cycle	-	da++	S cycle	R’’
				(pc+3i)			R’’’

Copyright © 2000 ARM Limited. All rights reserved.

8.16 Load double register

The `LDRD` instruction behaves in the same way as an `LDM` of two registers. Refer to *Load multiple registers* on page 8-27 and the appropriate entries in Table 8-23 on page 8-28.

Copyright © 2000 ARM Limited. All rights reserved.

8.17 Store double register

The STRD instruction behaves in the same way as an STM of two registers. Refer to *Store multiple registers* on page 8-30 and the appropriate entries in Table 8-24 on page 8-30.

Copyright © 2000 ARM Limited. All rights reserved.

8.18 Data swap

A data swap is similar to a back-to-back load and store instruction. The data is read from external memory in the second cycle and the contents of the register are written to the external memory in the third cycle (which is merged with the first Execute cycle of the next instruction).

The data swapped can be a byte or word quantity.

The swap operation might be aborted in either the read or the write cycle. An aborted swap operation does not affect the destination register.

Note

Data swap instructions are not available in Thumb state.

The **DLOCK** output of ARM9E-S is driven HIGH for both read and write cycles to indicate to the memory system that it is an atomic operation.

8.18.1 Interlocks

A swap operation can cause one and two-cycle interlocks in a similar fashion to a load register instruction.

Table 8-25 shows the cycle timing for the basic data swap operation.

Table 8-25 Data swap cycle timing

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA	WDATA
Normal	1	pc+3i	I cycle	(pc+2i)	da	N cycle		
	2	pc+3i	S cycle	-	da	N cycle	(da)	-
				(pc+3i)			-	Rd
1 cycle interlock	1	pc+3i	I cycle	(pc+2i)	da	N cycle		
	2	pc+3i	I cycle	-	da	N cycle	(da)	-
	3	pc+3i	S cycle	-	-	I cycle	-	Rd
				(pc+3i)			-	-

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-25 Data swap cycle timing (continued)

Cycle		IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA	WDATA
2 cycle interlock	1	pc+3i	I cycle	(pc+2i)	da	N cycle		
	2	pc+3i	I cycle	-	da	N cycle	(da)	-
	3	pc+3i	I cycle	-	-	I cycle	-	Rd
	4	pc+3i	S cycle	-	-	I cycle	-	-
				(pc+3i)			-	-

Copyright © 2000 ARM Limited. All rights reserved.

8.19 PLD

A `PLD` operation executes in a single cycle. During the Execute cycle, the prefetch address is calculated and broadcast on **DA[31:0]**. **DnMREQ** and **DSEQ** indicate an internal cycle, and **DnSPEC** is asserted.

Table 8-26 shows the cycle timings for PLD instructions.

Table 8-26 PLD operation cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA	WDATA
1	pc+3i	S cycle	(pc+2i)	da	I cycle		
			(pc+3i)			-	-

Copyright © 2000 ARM Limited. All rights reserved.

8.20 Software interrupt, undefined instruction, and exception entry

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

1. During the first cycle, the ARM9E-S constructs the forced address, and a mode change might take place.
2. During the second cycle, the ARM9E-S performs a fetch from the exception address. The return address to be stored in r14 is calculated. The state of the CPSR is saved in the relevant SPSR.
3. During the third cycle, the ARM9E-S performs a fetch from the exception address + 4, refilling the instruction pipeline.

The exception entry cycle timings are show in Table 8-27, where:

pc Is one of:

- the address of the `SWI` instruction for SWIs
- the address of the instruction following the last one to be executed before entering the exception for interrupts
- the address of the aborted instruction for Prefetch Aborts
- the address of the instruction following the one that attempted the aborted data transfer for Data Aborts.

Xn Is the appropriate exception address.

Table 8-27 Exception entry cycle timing

Cycle	IA	InMREQ, ISEQ	InTRANS	ITBIT	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	Xn	N cycle	1	0		-	I cycle	
2	Xn+4	S cycle	1	0	(Xn)	-	I cycle	-
3	Xn+8	S cycle	1	0	(Xn+4)	-	I cycle	-
					(Xn+8)			-

Note

The value on the **INSTR** bus can be unpredictable in the case of Prefetch Abort or Data Abort entry.

Copyright © 2000 ARM Limited. All rights reserved.

8.21 Coprocessor data processing operation

A coprocessor data (CDP) operation is a request from the ARM9E-S for the coprocessor to initiate some action. There is no need for the coprocessor to complete the action immediately, but the coprocessor must commit to completion before driving **CHSD** or **CHSE** to LAST.

If the coprocessor cannot perform the requested task, it leaves **CHSD** at ABSENT. When the coprocessor is able to perform the task, but cannot commit immediately, the coprocessor drives **CHSD** to WAIT, and in subsequent cycles drives **CHSE** to WAIT until able to commit, where it drives **CHSE** to LAST.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The coprocessor data operation cycle timings are shown in Table 8-28.

Table 8-28 Coprocessor data operation cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA/ WDATA	P[c]	LC[d]	CHSD	CHSE
ready										LAST	
	1	pc+3i	S cycle	(pc+2i)	-	I cycle		1	0		-
				(pc+3i)			-				
not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		LAST
	n + 1	pc+3i	S cycle	-	-	I cycle	-	1	0		-
				(pc+3i)			-				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

 Copyright © 2000 ARM Limited. All rights reserved.

8.22 Load coprocessor register (from memory)

The load coprocessor (LDC) operation transfers one or more words of data from memory to a coprocessor.

The coprocessor commits to the transfer only when it is ready to accept the data. The coprocessor indicates that it is ready for the transfer to commence by driving **CHSD** or **CHSE** to GO. The ARM9E-S produces addresses and requests data memory reads on behalf of the coprocessor, which is expected to accept the data at sequential rates. The coprocessor is responsible for determining the number of words to be transferred. It indicates this using the **CHSD** and **CHSE** signals, setting the appropriate signal to LAST in the cycle before it is ready to initiate the transfer of the last data word.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The load coprocessor register cycle timings are shown in Table 8-29.

Table 8-29 Load coprocessor register cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
1 register ready										LAST	
	1	pc+3i	S cycle	(pc+2i)	da	N cycle		1	0		-
				(pc+3i)			(da)				
1 register not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		LAST
	n+1	pc+3i	S cycle	-	da	N cycle	-	1	0		-
				(pc+3i)			(da)				

Copyright © 2000 ARM Limited. All rights reserved.

Table 8-29 Load coprocessor register cycle timing (continued)

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
m registers (m > 1) ready										GO	
	1	pc+3i	I cycle	(pc+2i)	da	N cycle		1	0		GO
	2	pc+3i	I cycle	-	da++	S cycle	(da)	1	0		GO
	.	pc+3i	I cycle	-	da++	S cycle	(da++)	1	0		GO
	m-1	pc+3i	I cycle	-	da++	S cycle	(da++)	1	0		LAST
	m	pc+3i	S cycle	-	da++	S cycle	(da++)	1	0		-
				(pc+3i)			(da++)				
m registers (m > 1) not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		GO
	n+1	pc+3i	I cycle	-	da	N cycle	-	1	0		GO
	n+2	pc+3i	I cycle	-	da++	S cycle	(da)	1	0		GO
	.	pc+3i	I cycle	-	da++	S cycle	(da++)	1	0		GO
	n+ m-1	pc+3i	I cycle	-	da++	S cycle	(da++)	1	0		LAST
	n+ m	pc+3i	S cycle	-	da++	S cycle	(da++)	1	0		-
				(pc+3i)			(da++)				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

 Copyright © 2000 ARM Limited. All rights reserved.

8.23 Store coprocessor register (to memory)

The store coprocessor (`STC`) operation transfers one or more words of data from a coprocessor to memory.

The coprocessor commits to the transfer only when it is ready to write the data. The coprocessor indicates that it is ready for the transfer to commence by driving **CHSD** or **CHSE** to GO. The ARM9E-S produces addresses and requests data memory writes on behalf of the coprocessor, which is expected to produce the data at sequential rates. The coprocessor is responsible for determining the number of words to be transferred. It indicates this using the **CHSD** and **CHSE** signals, setting the appropriate signal to LAST in the cycle before it is ready to initiate the transfer of the last data word.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The store coprocessor register cycle timings are shown in Table 8-30.

Table 8-30 Store coprocessor register cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DRQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
1 register ready										LAST	
	1	pc+3i	S cycle	(pc+2i)	da	N cycle		1	0		-
				(pc+3i)			CPData1				
1 register not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		LAST
	n+1	pc+3i	S cycle	-	da	N cycle	-	1	0		-
				(pc+3i)			CPData1				

 Copyright © 2000 ARM Limited. All rights reserved.

Table 8-30 Store coprocessor register cycle timing (continued)

Cycle		IA	IREQ[a]	INSTR	DA	DRQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
m registers (m > 1) ready										GO	
	1	pc+3i	I cycle	(pc+2i)	da	N cycle		1	0		GO
	2	pc+3i	I cycle	-	da++	S cycle	CPData1	1	0		GO
	.	pc+3i	I cycle	-	da++	S cycle	CPData	1	0		GO
	m-1	pc+3i	I cycle	-	da++	S cycle	CPDatam-2	1	0		LAST
	m	pc+3i	S cycle	-	da++	S cycle	CPDatam-1	1	0		-
				(pc+3i)			CPDatam				
m registers (m > 1) not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		GO
	n+1	pc+3i	I cycle	-	da	N cycle	-	1	0		GO
	n+2	pc+3i	I cycle	-	da++	S cycle	CPData1	1	0		GO
	.	pc+3i	I cycle	-	da++	S cycle	CPData	1	0		GO
	n+m-1	pc+3i	I cycle	-	da++	S cycle	CPDatam-2	1	0		LAST
	n+m	pc+3i	S cycle	-	da++	S cycle	CPDatam-1	1	0		-
				(pc+3i)			CPDatam				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DRQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

Copyright © 2000 ARM Limited. All rights reserved.

8.24 Coprocessor register transfer (to ARM)

The move from coprocessor (`MRC`) operation transfers a single coprocessor register into the specified ARM register.

Data is transferred over the data bus interface, in a similar fashion to a load register operation.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The `MRC` instruction cycle timings are shown in Table 8-31.

Table 8-31 MRC instruction cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
ready										LAST	
	1	pc+3i	S cycle	(pc+2i)	-	C cycle		1	0		-
				(pc+3i)			CPData				
not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		LAST
	n+1	pc+3i	S cycle	-	-	C cycle	-	1	0		-
				(pc+3i)			CPData				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

Copyright © 2000 ARM Limited. All rights reserved.

8.25 Coprocessor register transfer (from ARM)

The move to coprocessor (`MCR`) operation transfers a specified ARM register to a coprocessor register.

Data is transferred over the data bus interface, in a similar fashion to a store register operation.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The `MCR` instruction cycle timings are shown in Table 8-32.

Table 8-32 MCR instruction cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	WDATA	P[c]	LC[d]	CHSD	CHSE
ready										LAST	
	1	pc+3i	S cycle	(pc+2i)	-	C cycle		1	0		-
				(pc+3i)			Rd				
not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		LAST
	n+1	pc+3i	S cycle	-	-	C cycle	-	1	0		-
				(pc+3i)			Rd				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

Copyright © 2000 ARM Limited. All rights reserved.

8.26 Double coprocessor register transfer (to ARM)

The move double from coprocessor (`MRRC`) operation transfers two coprocessor registers into the specified ARM registers.

Data is transferred over the data bus interface, in a similar fashion to a load register operation.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The `MRRC` instruction cycle timings are shown in Table 8-33.

Table 8-33 MRRC instruction cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA	P[c]	LC[d]	CHSD	CHSE
ready										GO	
	1	pc+3i	I cycle	(pc+2i)	-	C cycle		1	0		LAST
	2	pc+3i	S cycle	-	-	C cycle	CPData1	1	0		-
				(pc+3i)			CPData2				
not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		GO
	n+1	pc+3i	I cycle	-	-	C cycle	-	1	0		LAST
	n+2	pc+3i	S cycle	-	-	C cycle	CPData1	1	0		-
				(pc+3i)			CPData2				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

Copyright © 2000 ARM Limited. All rights reserved.

8.27 Double coprocessor register transfer (from ARM)

The move double to coprocessor (`MCRR`) operation transfers two specified ARM registers to a coprocessor.

Data is transferred over the data bus interface, in a similar fashion to a store register operation.

An interrupt can cause the ARM9E-S to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 6-17).

Note

Coprocessor operations are only available in ARM state.

The `MCRR` instruction cycle timings are shown in Table 8-34.

Table 8-34 MCRR instruction cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	WDATA	P[c]	LC[d]	CHSD	CHSE
ready										GO	
	1	pc+3i	I cycle	(pc+2i)	-	C cycle		1	0		LAST
		pc+3i	S cycle	-	-	C cycle	Rd	1	0		-
				(pc+3i)			Rn				
not ready										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	1	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	1	0		GO
	n+1	pc+3i	I cycle	-	-	C cycle	-	1	0		LAST
	n+2	pc+3i	S cycle	(pc+3i)	-	C cycle	Rd	1	0		-
							Rn				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

 Copyright © 2000 ARM Limited. All rights reserved.

8.28 Coprocessor absent

If no coprocessor is able to process a coprocessor instruction, the instruction is treated as an UNDEFINED instruction. This allows software to emulate coprocessor instructions when no hardware coprocessor is present.

Note

By default, **CHSD** and **CHSE** must be driven to ABSENT unless the coprocessor instruction is being handled by a coprocessor. Coprocessor operations are only available in ARM state.

The cycle timings for coprocessor absent instructions are shown in Table 8-35.

Table 8-35 Coprocessor absent instruction cycle timing

Cycle		IA	IREQ[a]	INSTR	DA	DREQ[b]	RDATA/ WDATA	P[c]	LC[d]	CHSD	CHSE
coprocessor absent in decode										ABSENT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0	-	-
	2	`0x4`	N cycle	-	-	I cycle	-	0	0	-	-
	3	`0x8`	S cycle	(`0x4`)	-	I cycle	-	0	0		-
	4	`0xC`	S cycle	(`0x8`)	-	I cycle	-	0	0		-
				(`0xC`)			-				
coprocessor absent in execute										WAIT	
	1	pc+3i	I cycle	(pc+2i)	-	I cycle		1	0		WAIT
	.	pc+3i	I cycle	-	-	I cycle	-	0	0		WAIT
	n	pc+3i	I cycle	-	-	I cycle	-	0	0		ABSENT
	n+1	`0x4`	N cycle	-	-	I cycle	-	0	0		-
	n+2	`0x8`	S cycle	(`0x4`)	-	I cycle	-	0	0		
	n+3	`0xC`	S cycle	(`0x8`)	-	I cycle	-	0	0		
				(`0xC`)			-				

a. **IREQ** = **InMREQ**, **ISEQ**.
b. **DREQ** = **DnMREQ**, **DSEQ**.
c. **P** = **PASS**.
d. **LC** = **LATECANCEL**.

 Copyright © 2000 ARM Limited. All rights reserved.

8.29 Unexecuted instructions

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

Table 8-36 shows the instruction cycle timing for an unexecuted instruction.

Table 8-36 Unexecuted instruction cycle timing

Cycle	IA	InMREQ, ISEQ	INSTR	DA	DnMREQ, DSEQ	RDATA/ WDATA
1	pc + 3i	S cycle	(pc + 2i)	-	I cycle	
			(pc + 3i)			-

Copyright © 2000 ARM Limited. All rights reserved.

 Copyright © 2000 ARM Limited. All rights reserved.

Chapter 9
AC Parameters

This chapter gives the AC timing parameters of the ARM9E-S. It contains the following sections:

- *Timing diagrams* on page 9-2
- *AC timing parameter definitions* on page 9-8.

 Copyright © 2000 ARM Limited. All rights reserved.

9.1 Timing diagrams

The timing diagrams in this section are:

- Figure 9-1 *Instruction memory interface timing*
- Figure 9-2 *Data memory interface timing* on page 9-3
- Figure 9-3 *Clock enable timing* on page 9-3
- Figure 9-4 *Coprocessor interface timing* on page 9-4
- Figure 9-5 *Exception and configuration timing* on page 9-4
- Figure 9-6 *Debug interface timing* on page 9-5
- Figure 9-7 *Interrupt sensitivity status timing* on page 9-5
- Figure 9-8 *JTAG interface timing* on page 9-6
- Figure 9-9 *DBGSDOUT to DBGTDO relationship* on page 9-7.

Instruction memory interface timing parameters are shown in Figure 9-1.

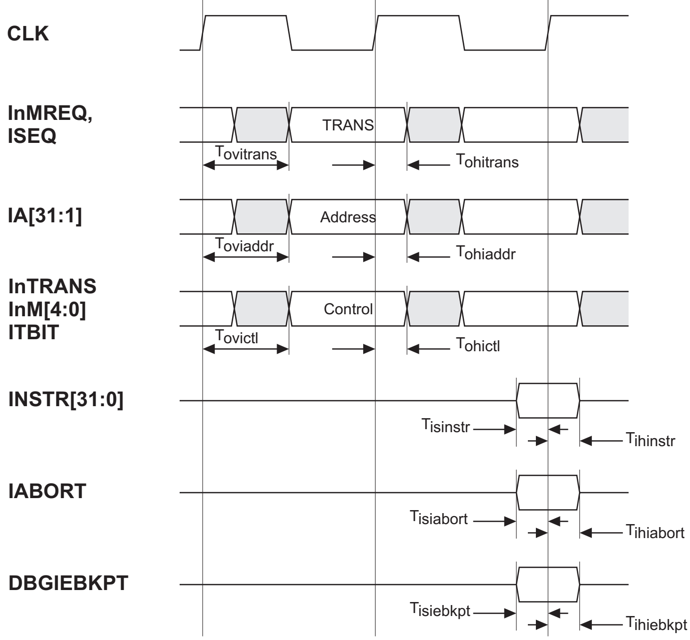


Figure 9-1 Instruction memory interface timing

Copyright © 2000 ARM Limited. All rights reserved.

Data memory interface timing parameters are shown in Figure 9-2.

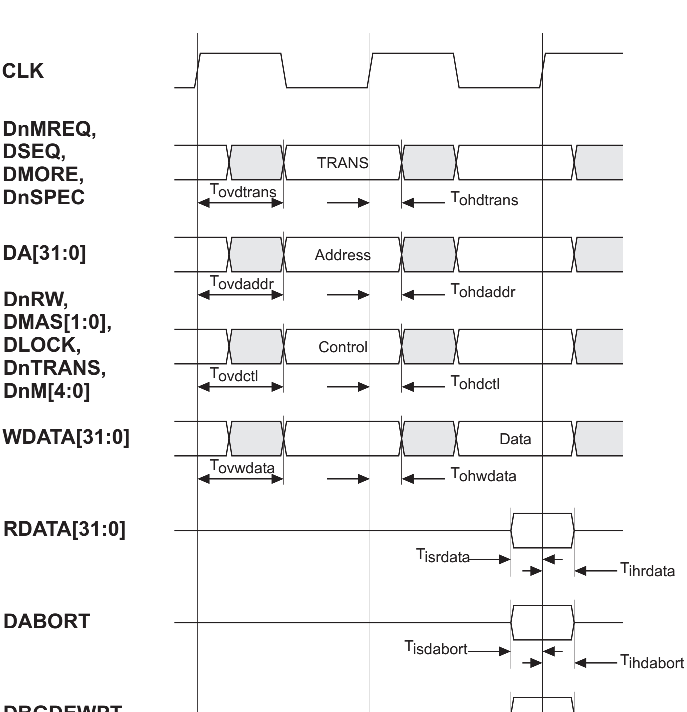


Figure 9-2 Data memory interface timing

Clock enable timing parameters are shown in Figure 9-3.

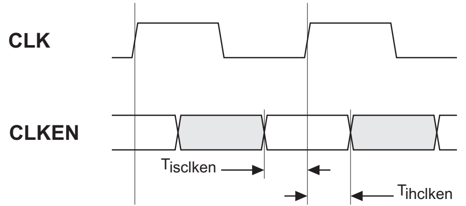


Figure 9-3 Clock enable timing

Copyright © 2000 ARM Limited. All rights reserved.

Coprocessor interface timing parameters are shown in Figure 9-4.

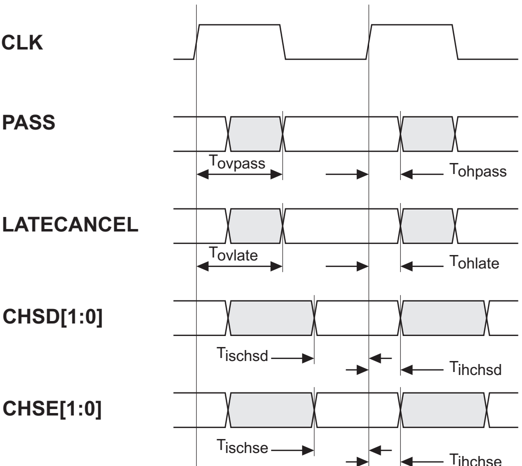


Figure 9-4 Coprocessor interface timing

Exception and configuration timing parameters are shown in Figure 9-5.

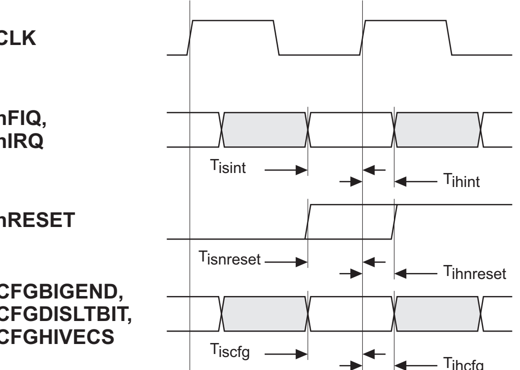


Figure 9-5 Exception and configuration timing

Copyright © 2000 ARM Limited. All rights reserved.

Debug interface timing parameters are shown in Figure 9-6.

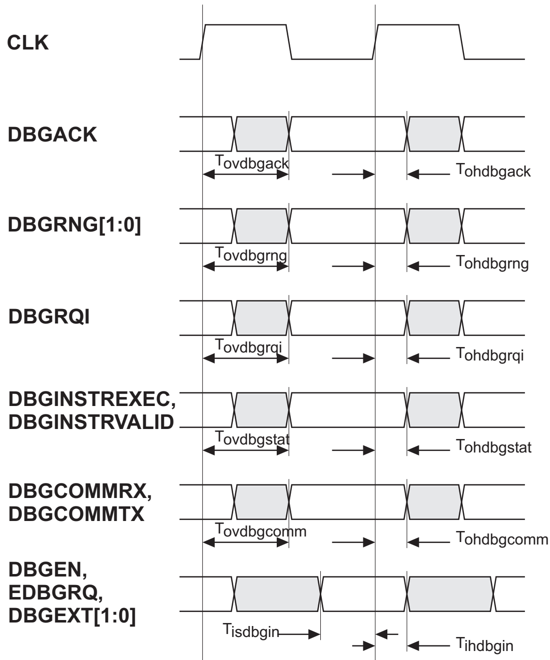


Figure 9-6 Debug interface timing

Sensitive to interrupt timing parameters are shown in Figure 9-7.

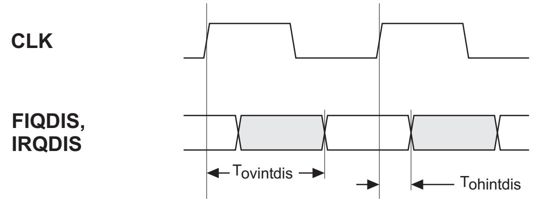


Figure 9-7 Interrupt sensitivity status timing

Copyright © 2000 ARM Limited. All rights reserved.

JTAG interface timing parameters are shown in Figure 9-8.

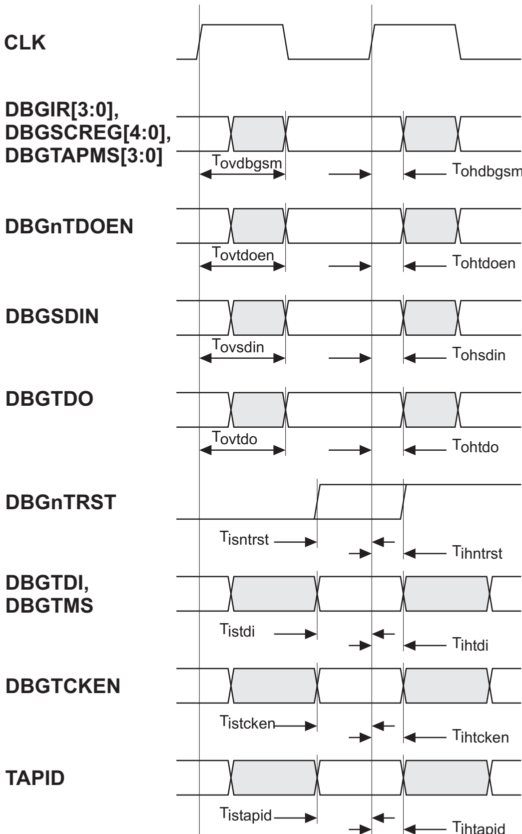


Figure 9-8 JTAG interface timing

Copyright © 2000 ARM Limited. All rights reserved.

The relationship between **DBGSDOUT** and **DBGTDO** is shown in Figure 9-9.

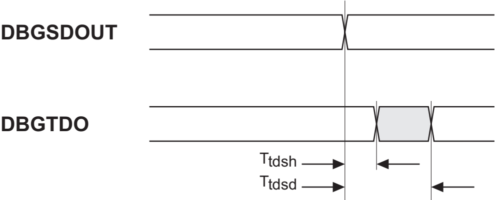


Figure 9-9 DBGSDOUT to DBGTDO relationship

 Copyright © 2000 ARM Limited. All rights reserved.

9.2 AC timing parameter definitions

Table 9-1 shows target AC parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

Note

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

Table 9-1 Target AC timing parameters

Symbol	Parameter	Min	Max
Tcyc	**CLK** cycle time	100%	-
Tisclken	**CLKEN** input setup to rising **CLK**	40%	-
Tihclken	**CLKEN** input hold from rising **CLK**	-	0%
Tovitrans	Rising **CLK** to instruction transaction valid	-	80%
Tohitrans	Instruction transaction hold time from rising **CLK**	>0%	-
Toviaddr	Rising **CLK** to **IA** valid	-	80%
Tohiaddr	**IA** hold time from rising **CLK**	>0%	-
Tovictl	Rising **CLK** to instruction control valid	-	80%
Tohictl	Instruction control hold time from rising **CLK**	>0%	-
Tisinstr	**INSTR** input setup to rising **CLK**	20%	-
Tihinstr	**INSTR** input hold from rising **CLK**	-	0%
Tisiabort	**IABORT** input setup to rising **CLK**	15%	-
Tihiabort	**IABORT** input hold from rising **CLK**	-	0%
Tisiebkpt	**DBGIEBKPT** input setup to rising **CLK**	15%	-
Tihiebkpt	**DBGIEBKPT** input hold from rising **CLK**	-	0%
Tovdtrans	Rising **CLK** to data transaction valid	-	70%
Tohdtrans	Data transaction hold time from **CLK** rising	>0%	-
Tovdaddr	Rising **CLK** to **DA** valid	-	80%

Copyright © 2000 ARM Limited. All rights reserved.

Table 9-1 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max
Tohdaddr	DA hold time from CLK rising	>0%	-
Tovdctl	Rising **CLK** to data control valid	-	70%
Tohdctl	Data control hold time from **CLK** rising	>0%	-
Tovwdata	Rising **CLK** to **WDATA** valid	-	20%
Tohwdata	**WDATA** hold time from **CLK** rising	>0%	-
Tisrdata	**RDATA** input setup to rising **CLK**	20%	-
Tihrdata	**RDATA** input hold from rising **CLK**	-	0%
Tisdabort	**DABORT** input setup to rising **CLK**	15%	-
Tihdabort	**DABORT** input hold from rising **CLK**	-	0%
Tisdewpt	**DBGDEWPT** input setup to rising **CLK**	15%	-
Tihdewpt	**DBGDEWPT** input hold from rising **CLK**	-	0%
Tovintdis	Rising **CLK** to Sensitive to interrupt status valid	-	70%
Tohintdis	Sensitive to interrupt status hold from **CLK** rising	>0%	-
Tovpass	Rising **CLK** to **PASS** valid	-	40%
Tohpass	**PASS** hold time from **CLK** rising	>0%	-
Tovlate	Rising **CLK** to **CPLATECANCEL** valid	-	25%
Tohlate	**CPLATECANCEL** hold from **CLK** rising	>0%	-
Tischsd	**CHSD** input setup to rising **CLK**	30%	-
Tihchsd	**CHSD** input hold from rising **CLK**	-	0%
Tischse	**CHSE** input setup to rising **CLK**	30%	-
Tihchse	**CHSE** input hold from rising **CLK**	-	0%
Tisint	Interrupt input setup to rising **CLK**	15%	-
Tihint	Interrupt input hold from rising **CLK**	-	0%
Tisnreset	**nRESET** input setup to rising **CLK**	25%	-
Tihnreset	**nRESET** input hold from rising **CLK**	-	0%

 Copyright © 2000 ARM Limited. All rights reserved.

Table 9-1 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max
Tiscfg	Configuration input setup to rising **CLK**	20%	-
Tihcfg	Configuration input hold from rising **CLK**	-	0%
Tovdbgack	**CLK** rising to **DBGACK** valid	-	60%
Tohdbgack	**DBGACK** hold time from **CLK** rising	>0%	-
Tovdbgrng	**CLK** rising to **DBGRNG** valid	-	80%
Tohdbgrng	**DBGRNG** hold time from **CLK** rising	>0%	-
Tovdbgrqi	**CLK** rising to **DBGRQI** valid	-	45%
Tohdbgrqi	**DBGRQI** hold time from **CLK** rising	>0%	-
Tovdbgstat	Rising **CLK** to debug status valid	-	30%
Tohdbgstat	Debug status hold from **CLK** rising	>0%	-
Tovdbgcomm	Rising **CLK** to comms channel outputs valid	-	60%
Tohdbgcomm	Comms channel output hold time from rising **CLK**	>0%	-
Tisdbgin	Debug inputs input setup to rising **CLK**	35%	-
Tihdbgin	Debug inputs input hold from rising **CLK**	-	0%
Tovdbgsm	**CLK** rising to debug state valid	-	30%
Tohdbgsm	Debug state hold from **CLK** rising	>0%	-
Tovtdoen	**CLK** rising to **DBGnTDOEN** valid	-	40%
Tohtdoen	**DBGnTDOEN** hold from **CLK** rising	>0%	-
Tovsdin	**CLK** rising to **DBGSDIN** valid	-	20%
Tohsdin	**DBGSDIN** hold from **CLK** rising	>0%	-
Tovtdo	**CLK** rising to **DBGTDO** valid	-	35%
Tohtdo	**DBGTDO** hold from **CLK** rising	>0%	-
Tisntrst	**DBGnTRST** input setup to **CLK** rising	25%	-
Tihntrst	**DBGnTRST** input hold from **CLK** rising	-	0%
Tistdi	**DBGTDI** input setup to **CLK** rising	25%	-

 Copyright © 2000 ARM Limited. All rights reserved.

Table 9-1 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max
Tihtdi	**DBGTDI** input hold from **CLK** rising	-	0%
Tistcken	**DBGTCKEN** input setup to **CLK** rising	35%	-
Tihtcken	**DBGTCKEN** input hold from **CLK** rising	-	0%
Tistapid	**TAPID** input setup to **CLK** rising	20%	-
Tihtapid	**TAPID** input hold time from **CLK** rising	-	0%
Ttdsd	**DBGTDO** delay from **DBGSDOUT** changing	-	-
Ttdsh	**DBGTDO** hold time from **DBGSDOUT** changing	-	-

Copyright © 2000 ARM Limited. All rights reserved.

Copyright © 2000 ARM Limited. All rights reserved.

Appendix A
Signal Descriptions

This appendix lists and describes all the ARM9E-S interface signals. It contains the following sections:

- *Clock interface signals* on page A-2
- *Instruction memory interface signals* on page A-3
- *Data memory interface signals* on page A-4
- *Miscellaneous signals* on page A-6
- *Coprocessor interface signals* on page A-7
- *Debug signals* on page A-8.

 Copyright © 2000 ARM Limited. All rights reserved.

A.1 Clock interface signals

The clock interface signals are given in Table A-1.

Table A-1 Clock interface signals

Name	Direction	Description
CLK System clock	Input	This clock times all operations in the ARM9E-S processor. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock can be stretched in either phase. Synchronous wait states can be added using the **CLKEN** signal. Through the use of the **DBGTCKEN** signal, this clock also times debug operations.
CLKEN Wait-state control	Input	ARM9E-S can be stalled for integer clock cycles by driving **CLKEN** LOW. This signal must be held HIGH at all other times.
CORECLKENOUT	Output	The principal state advance signal for the ARM9E-S core. This output must be connected directly to the **CORECLKENIN** input for correct operation. This signal has been exported from the core to ease buffer tree insertion from the **CORECLKENIN** input. You must take care when loading and routing the **CORECLKENOUT** to **CORECLKENIN** connection.
CORECLKENIN	Input	This input must be connected to the **CORECLKENOUT** output.

Copyright © 2000 ARM Limited. All rights reserved.

A.2 Instruction memory interface signals

The instruction memory interface signals are shown in Table A-2.

Table A-2 Instruction memory interface signals

Name	Direction	Description
IA[31:1] Instruction address	Output	The processor instruction address bus.
IABORT Instruction abort	Input	This is an input that allows the memory system to tell the processor that the requested instruction memory access is not allowed.
INSTR[31:0] Instruction data	Input	This bus is used to transfer instructions between the memory system and the processor.
DBGIEBKPT Instruction breakpoint	Input	This is an input that allows external hardware to halt the execution of the processor for debug purposes. If HIGH at the end of an instruction Fetch it causes the ARM9E-S to enter debug state if that instruction reaches the Execute stage of the processor pipeline.
InMREQ Not instruction memory request	Output	If LOW at the end the cycle, then the processor requires a memory access during the following cycle.
InM[4:0] Instruction mode	Output	These contain the current mode of the processor and are valid with the address.
InTRANS Not memory translate	Output	When LOW the processor is in User mode, when HIGH the processor is in a privileged mode. This signal is valid with the address.
ISEQ Instruction Sequential	Output	If HIGH at the end of the cycle then any instruction memory access during the following cycle is sequential from the last instruction memory access.
ITBIT Instruction Thumb bit	Output	When HIGH the processor is in Thumb state, when LOW the processor is in ARM state. This signal is valid with the address.

Copyright © 2000 ARM Limited. All rights reserved.

A.3 Data memory interface signals

The data memory interface signals are shown in Table A-3.

Table A-3 Data memory interface signals

Name	Direction	Description
DA[31:0] Data address	Output	The processor data address bus.
DABORT Data abort	Input	This is an input that allows the memory system to tell the processor that the requested data memory access is not allowed.
RDATA [31:0] Read data	Input	This bus is used to transfer data between the memory system and the processor during read cycles (when **DnRW** is LOW).
WDATA [31: 0] Write data	Output	This bus is used to transfer data between the memory system and the processor during write cycles (when **DnRW** is HIGH).
DBGDEWPT Data watchpoint	Input	This is an input that allows external hardware to halt the execution of the processor for debug purposes. If HIGH at the end of a data memory request cycle, it causes the ARM9E-S to enter debug state.
DLOCK Data lock	Output	If HIGH, then any data memory access in the following cycle is locked, and the memory controller must wait until **DLOCK** goes LOW before allowing another device to access the memory.
DMAS[1:0] Data memory access size	Output	These encode the size of a data memory access in the following cycle. A word access is encoded as 10 (binary), a halfword access as 0l, and a byte access as 00. The encoding 11 is reserved.
DMORE Data more	Output	If HIGH at the end of the cycle, then the data memory access in the following cycle is directly followed by a sequential data memory access.
DnMREQ Not data memory request	Output	If LOW at the end the cycle, then the processor requires a data memory access in the following cycle.

Copyright © 2000 ARM Limited. All rights reserved.

Table A-3 Data memory interface signals (continued)

Name	Direction	Description
DnM[4:0] Data mode	Output	The processor mode that any data memory accesses must be performed in. Valid with the data address.
DnRW Data not read, write	Output	If LOW at the end of the cycle, then any data memory access in the following cycle is a read. If HIGH then it is a write.
DnSPEC Not data speculative request	Output	If LOW at the end of the cycle, then the processor is indicating to the memory system that the data stored at the memory location specified by **DA** might be required in subsequent cycles. **DnSPEC** is a speculative signal, so the memory system does not have to perform any action based on **DnSPEC** unless it sees fit. The memory system must return an abort for a speculative access. **DnSPEC** is *not* asserted in the same cycle as **DnMREQ**.
DnTRANS Data not memory translate	Output	If LOW at the end of a cycle, then any data memory access must be performed with User mode privileges. If HIGH it must have Supervisor mode privileges.
DSEQ Data sequential address	Output	If HIGH at the end of the cycle, then any data memory access in the following cycle is sequential from the last data memory access.

 Copyright © 2000 ARM Limited. All rights reserved.

A.4 Miscellaneous signals

The miscellaneous signals are shown in Table A-4.

Table A-4 Miscellaneous signals

Name	Direction	Description
nFIQ Not fast interrupt	Input	This is the Fast Interrupt Request signal. This input is a synchronous input to the core. It is not synchronized internally to the core.
nIRQ Not interrupt request	Input	This is the Interrupt Request signal. This input is a synchronous input to the core. It is not synchronized internally to the core.
CFGBIGEND Big-endian configuration	Input	When HIGH, the ARM9E-S processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian. This is a static configuration signal.
CFGDISLTBIT	Input	When HIGH, the ARM9E-S disables certain ARMv5T defined behavior involving loading data to the PC. This input must be tied LOW for normal operation and full ARMv5T compatibility. This is a static configuration signal.
CFGHIVECS High vectors configuration	Input	When LOW, the ARM9E-S exception vectors start at address `0x0000 0000`. When HIGH the ARM9E-S exception vectors start at address `0xFFFF 0000`. This is a static configuration signal.
nRESET Not reset	Input	This active LOW reset signal is used to start the processor from a known address. This is a level-sensitive asynchronous reset.
FIQDIS FIQ disabled	Output	When HIGH, indicates that the ARM9E-S is insensitive to the state of the **nFIQ** input signal.
IRQDIS IRQ disabled	Output	When HIGH, indicates that the ARM9E-S is insensitive to the state of the **nIRQ** input signal.

Copyright © 2000 ARM Limited. All rights reserved.

A.5 Coprocessor interface signals

The coprocessor interface signals are shown in Table A-5.

Table A-5 Coprocessor interface signals

Name	Direction	Description
PASS	Output	This signal indicates that there is a coprocessor instruction in the Execute stage of the pipeline, and it must be executed.
CHSD[1:0] Coprocessor handshake decode	Input	The handshake signals from the Decode stage of the pipeline follower of the coprocessor.
CHSE[1:0] Coprocessor handshake execute	Input	The handshake signals from the Execute stage of the pipeline follower of the coprocessor.
LATECANCEL Coprocessor late cancel	Output	If HIGH during the first memory cycle of a coprocessor instruction, then the coprocessor must cancel the instruction without changing any internal state. This signal is only asserted in cycles where the previous instruction accessed memory and a Data Abort occurred.

 Copyright © 2000 ARM Limited. All rights reserved.

A.6 Debug signals

The debug signals are shown in Table A-6.

Table A-6 Debug signals

Name	Direction	Description
DBGIR[3:0] TAP controller instruction register	Output	These four bits reflect the current instruction loaded into the TAP controller instruction register. These bits change when the TAP state machine is in the UPDATE-IR state.
DBGnTRST Not test reset	Input	This is the active LOW reset signal for the EmbeddedICE internal state. This signal is a level-sensitive asynchronous reset input.
DBGnTDOEN Not **DBGTDO** enable	Output	When LOW, this signal denotes that serial data is being driven out on the **DBGTDO** output. **DBGnTDOEN** is usually used as an output enable for a **DBGTDO** pin in a packaged part.
DBGSCREG[4:0]	Output	These five bits reflect the ID number of the scan chain currently selected by the TAP Scan Chain Register controller. These bits change when the TAP state machine is in the UPDATE-DR state.
DBGSDIN Output boundary scan serial input data	Output	This signal contains the serial data to be applied to an external scan chain.
DBGSDOUT Input boundary scan serial output data	Input	This is the serial data out of an external scan chain. When an external boundary scan chain is not connected, this input must be tied LOW.
DBGTAPSM[3:0] TAP controller state machine	Output	This bus reflects the current state of the TAP controller state machine.
DBGTCKEN	Input	Synchronous enable for debug logic accessed using the JTAG interface.
DBGTDI	Input	Test data input to the debug logic.
DBGTDO	Output	Output from the debug logic.
DBGTMS	Input	Test mode select for the TAP controller.

 Copyright © 2000 ARM Limited. All rights reserved.

Table A-6 Debug signals (continued)

Name	Direction	Description
DBGCOMMRX Communications channel receive	Output	When HIGH, this signal denotes that the comms channel receive buffer contains valid data waiting to be read by the ARM9E-S.
DBGCOMMTX Communications channel transmit	Output	When HIGH, this signal denotes that the comms channel transmit buffer is empty.
DBGACK Debug acknowledge	Output	When HIGH, indicates that the processor is in debug state.
DBGEN Debug enable	Input	This input signal allows the debug features of the processor to be disabled. This signal must be LOW when debugging is not required.
DBGRQI Internal debug request	Output	This signal represents the state of bit 1 of the debug control register that is combined with **EDBGRQ** and presented to the core debug logic.
EDBGRQ	Input	External debug request. An external debugger may force the processor to enter debug state by asserting this signal.
DBGEXT[1:0] EmbeddedICE external input	Input	This input to the EmbeddedICE logic allows breakpoints and watchpoints to be dependent on external conditions.
DBGINSTREXEC	Output	Instruction executed. Indicates that the instruction in the Execute stage of the processors pipeline has been executed.
DBGINSTRVALID	Output	Instruction valid. Indicates that the instruction in the Execute stage of the processors pipeline was valid and has been executed (unless it failed its conditions codes).
DBGRNG[1:0] EmbeddedICE Rangeout	Output	This output indicates that the corresponding EmbeddedICE watchpoint unit has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the enable control bit of the watchpoint unit.
TAPID[31:0] Boundary scan ID code	Input	This input specifies the ID code value shifted out on **DBGTDO** when the IDCODE instruction is entered into the TAP controller.

Copyright © 2000 ARM Limited. All rights reserved.

Copyright © 2000 ARM Limited. All rights reserved.

Appendix B
Differences Between the ARM9E-S and the ARM9TDMI

This appendix describes the differences between the ARM9E-S and ARM9TDMI macrocell interfaces. It contains the following sections:

- *Interface signals* on page B-2
- *ATPG scan interface* on page B-5
- *Timing parameters* on page B-6
- *ARM9E-S design considerations* on page B-7
- *ARM9E-S debugger considerations* on page B-9.

 Copyright © 2000 ARM Limited. All rights reserved.

B.1 Interface signals

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

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

CP Shows coprocessor expansion interface signals.

DBG Shows scan-based EmbeddedICE debug support input or output.

Other signals provide the interface for the system designer, which is primarily memory-mapped. Table B-1 shows the ARM9E-S signals with their ARM9TDMI hard macrocell equivalent signals.

Table B-1 ARM9E-S signals and ARM9TDMI hard macrocell equivalents

ARM9E-S signal	Function	ARM9TDMI hard macrocell equivalent	Note
CFGBIGEND	1 = big-endian configuration. 0 = little-endian configuration.	**BIGEND**	-
CFGDISLTBIT	1 = disable specific ARMv5T behavior. 0 = enable (default).	-	-
CFGHIVECS	1 = exception vectors start at `0xFFFF 0000`. 0 = exception vectors start at `0x0000 0000`.	**HIVECS**	-
CLK	Rising edge master clock. All inputs are sampled on the rising edge of **CLK**. All timing dependencies are from the rising edge of **CLK**.	**GCLK**	a
CLKEN	System memory interface clock enable: 1 = advance the core on rising **CLK**. 0 = prevent the core advancing on rising **CLK**.	**nWAIT**	b
DA[31:0]	32-bit data address output bus, available in the cycle preceding the memory cycle.	**DA[31:0]**	c
DABORT	Data Abort.	**DABORT**	d
DBGCOMMRX	EmbeddedICE communication channel receive buffer full output.	**COMMRX**	-
DBGCOMMTX	EmbeddedICE communication channel transmit buffer empty output.	**COMMTX**	-
DBGDEWPT	External data watchpoint (tie LOW when not used).	**DEWPT**	e

 Copyright © 2000 ARM Limited. All rights reserved.

Table B-1 ARM9E-S signals and ARM9TDMI hard macrocell equivalents (continued)

ARM9E-S signal	Function	ARM9TDMI hard macrocell equivalent	Note
DBGEXT[1:0]	EmbeddedICE **EXTERN** debug qualifiers (tie LOW when not required).	**EXTERN0**, **EXTERN1**	-
DBGIEBKPT	External breakpoint (tie LOW when not used).	**IEBKPT**	e
DBGINSTREXEC	Instruction executed.	**INSTREXEC**	-
DBGINSTRVALID	Instruction valid.	-	-
DBGIR[3:0]	TAP controller instruction register.	**IR[3:0]**	-
DBGnTDOEN	**TDO** enable.	**nTDOEN**	f
DBGnTRST	TAP controller reset (asynchronous assertion).	**nTRST**	f
DBGRNG[1:0]	EmbeddedICE rangeout qualifier outputs.	**RANGEOUT1**, **RANGEOUT0**	-
DBGRQI	Internal status of debug request.	**DBGRQI**	g
DBGSCREG[4:0]	Scan chain register select.	**SCREG[4:0]**	-
DBGSDIN	Boundary scan serial data in.	**SDIN**	-
DBGSDOUT	Boundary scan serial data out.	**SDOUT**	-
DBGTAPSM[3:0]	TAP controller state machine state.	**TAPSM[3:0]**	-
DBGTCKEN	Multi-ICE clock input qualifier sampled on the rising edge of **CLK**. Used to qualify **CLK** to enable the debug subsystem.	**-**	-
DBGTDI	Test data input.	**TDI**	f
DBGTDO	Test data output.	**TDO**	f
DBGTMS	Test mode select.	**TMS**	f
EDBGRQ	External debug request.	**EDBGRQ**	h
IA[31:1]	31-bit instruction address output bus, available in the cycle preceding the Memory cycle.	**IA[31:1]**	c
INSTR[31:0]	Instruction data bus used to transfer instructions between the memory system and the ARM9E-S.	**ID[31:0]**	-
InMREQ	Instruction memory request.	**InMREQ**	c

 Copyright © 2000 ARM Limited. All rights reserved.

Table B-1 ARM9E-S signals and ARM9TDMI hard macrocell equivalents (continued)

ARM9E-S signal	Function	ARM9TDMI hard macrocell equivalent	Note
nFIQ	Fast interrupt request.	**nFIQ**	i
nIRQ	Interrupt request.	**nIRQ**	i
RDATA[31:0]	Data input bus.	**DDIN[31:0]**	j
WDATA[31:0]	Data output bus. This bus is always driven.	**DD[31:0]**	j

a. **CLK** is a rising edge clock. It is inverted with respect to the **GCLK** signal used on the ARM9TDMI hard macrocell.

b. **CLKEN** is sampled on the rising edge of **CLK**. The **nWAIT** signal on the ARM9TDMI hard macrocell must be held throughout the high phase of **GCLK**. This means that the address class outputs (**IA[31:1]**, **DA[31:0]**, **DnRW**, **DMAS**, **InTRANS**, **DnTRANS**, and **ITBIT**) can still change in a cycle in which **CLKEN** is taken LOW. You must take this possibility into account when designing a memory system.

c. All the address class signals (**IA[31:1]**, **DA[31:0]**, **DnRW**, **DMAS**, **InTRANS**, **DnTRANS**, and **ITBIT**) change on the rising edge of **CLK**. In a system with a low-frequency clock this means that the signals can change in the first phase of the clock cycle. This is unlike the ARM9TDMI hard macrocell where they always change in the last phase of the cycle.

d. The ARM9TDMI featured a combinational path from **DABORT** to **DnMREQ**. This path does not exist in ARM9E-S.

e. With ARM9TDMI, the breakpoint and watchpoint inputs had to be asserted in the phase 1 of the cycle following the cycle in which the data was returned from the memory system. With ARM9E-S, external breakpoints and watchpoints must be returned in the same cycle as the data.

f. All JTAG signals are synchronous to **CLK** on the ARM9E-S. There is no asynchronous **TCK** as on the ARM9TDMI hard macrocell. An external synchronizing circuit can be used to generate **TCLKEN** when an asynchronous **TCK** is required. However, **CLK** must be running.

g. The **DBGRQI** signal in ARM9TDMI features a combinational input to output path from **EDBGRQ**. This has been removed in ARM9E-S.

h. **EDBGRQ** must be synchronized externally to the macrocell. It is *not* an asynchronous input as on the ARM9TDMI hard macrocell.

i. **nFIQ** and **nIRQ** are synchronous inputs to the ARM9E-S, and are sampled on the rising edge of **CLK**. Asynchronous interrupts are not supported.

j. The ARM9E-S supports only unidirectional data buses, **RDATA[31:0]**, and **WDATA[31:0]**. When a bidirectional bus is required, you must implement external bus combining logic.

Copyright © 2000 ARM Limited. All rights reserved.

B.2 ATPG scan interface

Where automatic scan path is inserted for automatic test pattern generation, three signals are instantiated on the macrocell interface:

- **SCANENABLE** is LOW for normal usage, HIGH for scan test
- **SCANIN** is the serial scan path input
- **SCANOUT** is the serial scan path output.

Copyright © 2000 ARM Limited. All rights reserved.

B.3 Timing parameters

The timing constraints have been adjusted to balance the external timing parameters with the area of the synthesized core. All inputs are sampled on the rising edge of **CLK**. The timing diagrams associated with these timing parameters are shown in *Timing diagrams* on page 9-2.

The clock enables are sampled on every rising clock edge:

- **CLKEN** setup time is $T_{isclken}$, hold time is $T_{ihclken}$.
- **DBGTCKEN** setup time is $T_{istcken}$, hold time is $T_{ihtcken}$.

All other inputs are sampled on rising edge of **CLK** when the clock enable is active HIGH, for example:

- **IABORT** setup time is $T_{isiabort}$, hold time is $T_{ihiabort}$, when **CLKEN** is active.
- **RDATA** setup time is $T_{isrdata}$, hold time is $T_{ihrdata}$, when **CLKEN** is active.
- **DBGTMS**, **DBGTDI** setup time is T_{istdi}, hold time is T_{ihtdi}, when **DBGTCKEN** is active.

Outputs are all sampled on the rising edge of **CLK** with the appropriate clock enable active, for example:

- **IA** output hold time is $T_{ohiaddr}$, valid time is $T_{oviaddr}$ when **CLKEN** is active.
- **InMREQ, ISEQ** output hold time is $T_{ohitrans}$, valid time is $T_{ovitrans}$ when **CLKEN** is active.

Similarly, all memory, coprocessor, and debug signal expansion signals are defined with input setup parameters of T_{is}... , hold parameters of T_{ih}... , output hold parameters of T_{oh}...and output valid parameters of T_{ov}... .

Copyright © 2000 ARM Limited. All rights reserved.

B.4 ARM9E-S design considerations

When an ARM9TDMI hard macrocell design is being converted to ARM9E-S, the following areas require special consideration:

- *Master clock*
- *JTAG interface timing*
- *Interrupt timing*
- *Address class signal timing* on page B-8
- *Data Aborts* on page B-8.

B.4.1 Master clock

The master clock to the ARM9E-S, **CLK**, is inverted with respect to **GCLK** used on the ARM9TDMI hard macrocell. The rising edge of the clock is the active edge of the clock, on which all inputs are sampled.

All outputs are generated safely from the rising edge of **CLK**, with the following exceptions:

CORECLKENOUT
: This signal can change from the rising edge of **CLK** and has a causal relationship with **CLKEN**.

DBGTDO
: This signal can change from the rising edge of **CLK** and has a causal relationship with **DBGSDOUT**.

B.4.2 JTAG interface timing

All JTAG signals on the ARM9E-S are synchronous to the master clock input, **CLK**. When an external **TCK** is used, use an external synchronizer to the ARM9E-S.

B.4.3 Interrupt timing

As with all ARM9E-S signals, the interrupt signals, **nIRQ** and **nFIQ**, are sampled on the rising edge of **CLK**.

When you are converting an ARM9TDMI hard macrocell design where the **ISYNC** signal is asserted LOW, add a synchronizer to the design to synchronize the interrupt signals before they are applied to the ARM9E-S.

 Copyright © 2000 ARM Limited. All rights reserved.

B.4.4 Address class signal timing

The address class outputs (**IA[31:1], DA[31:0], DnRW, DMAS, InTRANS, DnTRANS, and ITBIT**) on the ARM9E-S all change in response to the rising edge of **CLK**. This means that they can change in the first phase of the clock in some systems. When exact compatibility is required, add latches to the outside of the ARM9E-S to make sure that they can change only in the second phase of the clock.

Because the **CLKEN** signal is sampled only on the rising edge of the clock, the address class outputs still change in a cycle in which **CLKEN** is LOW. (This is similar to the behavior of **I/DnMREQ** and **I/DSEQ** in an ARM9TDMI hard macrocell system, when a wait state is inserted using **nWAIT**.) Make sure that the memory system design takes this into account.

Also make sure that the correct address is used for the memory cycle, even though **IA/DA[31:0]** might have moved on to the address for the next memory cycle.

For further details, refer to Chapter 4 *Memory Interface*.

B.4.5 Data Aborts

The ARM9TDMI featured a combinational path from **DABORT** to **DnMREQ, DSEQ, and DMORE**. This path does not exist in ARM9E-S. A consequence of this change is that, in the case of two back-to-back memory accesses (for example a load followed by a store), the second access is not canceled by the ARM processor if the first is aborted. In these situations, the system must ignore the second memory request. For more details, see *DABORT* on page 4-18.

Copyright © 2000 ARM Limited. All rights reserved.

B.5 ARM9E-S debugger considerations

There are a number of differences between the ARM9TDMI and ARM9E-S that a JTAG debugger must be aware of:

- The EmbeddedICE version number in the debug channel status register is different. See *Debug comms channel control register* on page 7-17.
- From (test) reset, the ARM9E-S is configured into monitor mode debug. A debugger requiring the ARM processor halt mode debug features must clear the monitor mode enable bit in the debug control register. See *Debug control register* on page C-34.
- There are a number of instructions that have different cycle counts on ARM9E-S to ARM9TDMI. In particular, the MRS instruction always requires two cycles to execute on ARM9E-S. See Chapter 8 *Instruction Cycle Times* for more details on instruction cycle timing.
- The NV condition code cannot be used to provide a convenient single-cycle non-interlocking `NOP` operation. This is due to ARM9E-S implementing the ARMv5TE architecture. A special opcode, `0xE320 F000` provides a guaranteed single-cycle, non-interlocking `NOP` for ARM9E-S. This opcode is using an UNPREDICTABLE part of the instruction space, so that its behavior cannot be guaranteed over all ARM variants.

 Copyright © 2000 ARM Limited. All rights reserved.

 Copyright © 2000 ARM Limited. All rights reserved.

Appendix C
Debug in depth

This appendix describes in further detail the debug features of the ARM9E-S, and includes additional information about the EmbeddedICE-RT logic. It contains the following sections:

- *Scan chains and JTAG interface* on page C-2
- *Resetting the TAP controller* on page C-5
- *Instruction register* on page C-6
- *Public instructions* on page C-7
- *Test data registers* on page C-10
- *ARM9E-S core clock domains* on page C-17
- *Determining the core and system state* on page C-18
- *Behavior of the program counter during debug* on page C-24
- *Priorities and exceptions* on page C-27
- *EmbeddedICE-RT logic* on page C-28
- *Vector catching* on page C-39
- *Single-stepping* on page C-40
- *Coupling breakpoints and watchpoints* on page C-41
- *Disabling EmbeddedICE-RT* on page C-44
- *EmbeddedICE-RT timing* on page C-45.

Copyright © 2000 ARM Limited. All rights reserved.

C.1 Scan chains and JTAG interface

There are two JTAG-style scan chains within the ARM9E-S. These allow debugging and EmbeddedICE-RT programming.

The scan chains allow commands to be serially shifted into the ARM core, allowing the state of the core and the system to be interrogated. The JTAG interface requires only five pins on the package.

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

C.1.1 Debug scan chains

The two scan paths used for debug purposes are referred to as scan chain 1 and scan chain 2, and are shown in Figure C-1.

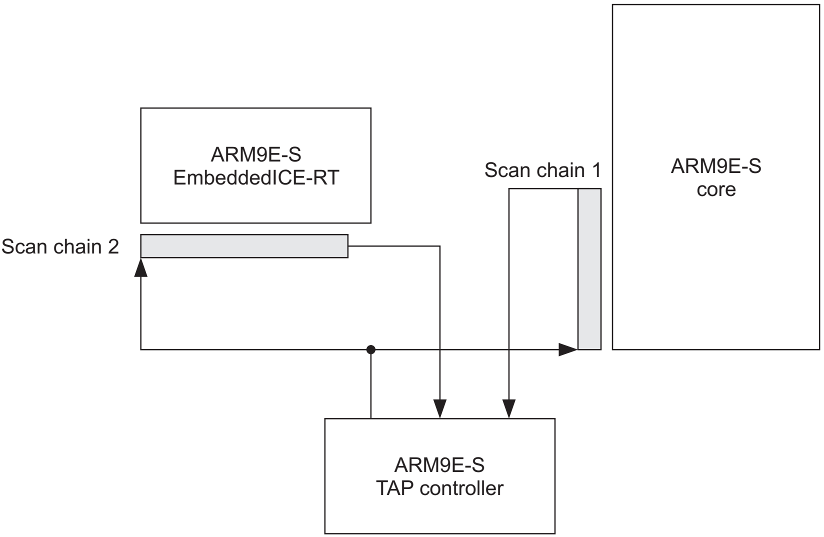


Figure C-1 ARM9E-S scan chain arrangements

Copyright © 2000 ARM Limited. All rights reserved.

Scan chain 1

Scan chain 1 is used for debugging the ARM9E-S core when it has entered debug state. You can use it to:

- inject instructions into the ARM pipeline
- read and write its registers
- perform memory accesses.

Scan chain 2

Scan chain 2 allows access to the EmbeddedICE-RT registers. Refer to *Test data registers* on page C-10 for details.

C.1.2 TAP state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure C-2 on page C-4 shows the state transitions that occur in the TAP controller. The state numbers shown in the diagram are output from the ARM9E-S on the **DBGTAPSM[3:0]** bits.

 Copyright © 2000 ARM Limited. All rights reserved.

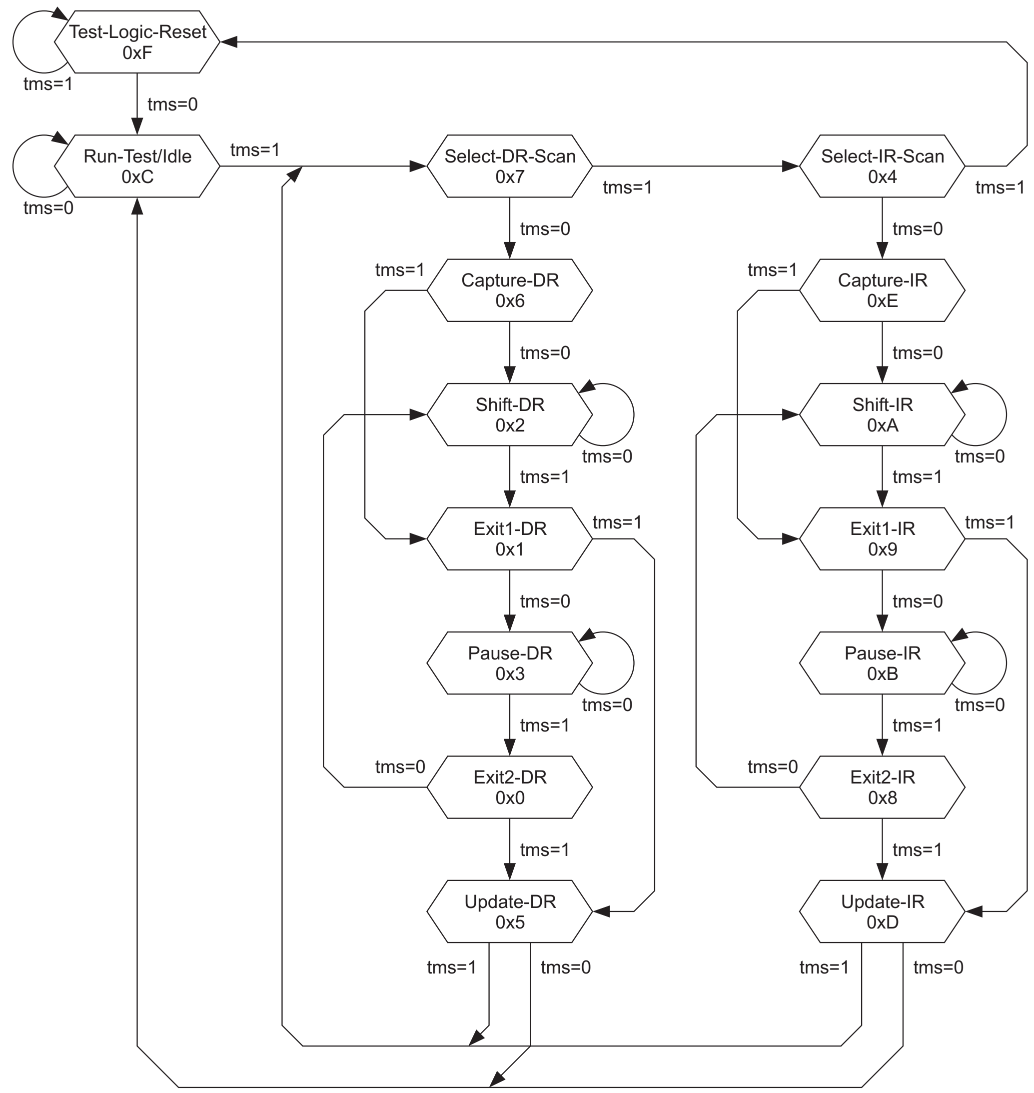


Figure C-2 Test access port controller state transitions[1]

1. From IEEE Std 1149.1-1990. Copyright 1999 IEEE. All rights reserved.

 Copyright © 2000 ARM Limited. All rights reserved.

C.2 Resetting the TAP controller

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

- to ready the boundary-scan interface for use, drive **DBGnTRST** LOW, and then HIGH again
- to prevent the boundary-scan interface from being used, the **DBGnTRST** input can be tied permanently LOW.

Note

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

The action of reset is as follows:

1. System mode is selected. This means that the boundary-scan cells do not intercept any of the signals passing between the external system and the core.
2. The IDCODE instruction is selected. When the TAP controller is put into the SHIFT-DR state, and **CLK** is pulsed while enabled by **DBGTCKEN**, the contents of the ID register are clocked out of **DBGTDO**.

 Copyright © 2000 ARM Limited. All rights reserved.

C.3 Instruction register

The instruction register is four bits in length.

There is no parity bit.

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

Copyright © 2000 ARM Limited. All rights reserved.

C.4 Public instructions

Instructions are loaded into the TAP state machine by scanning the appropriate bit pattern for the instruction when the TAP controller is in the SHIFT-IR state, and then advancing the TAP controller through the UPDATE-IR state.

Table C-1 shows the public instructions.

Table C-1 Public instructions

Instruction	Binary code
EXTEST	0000
SAMPLE/PRELOAD	0011
SCAN_N	0010
INTEST	1100
IDCODE	1110
BYPASS	1111
RESTART	0100

In the following descriptions, the ARM9E-S samples **DBGTDI** and **DBGTMS** on the rising edge of **CLK** with **DBGTCKEN** HIGH. All output transitions on **DBGTDO** occur as a result of the rising edge of **CLK** with **DBGTCKEN** HIGH.

C.4.1 EXTEST (0000)

The EXTEST instruction allows a boundary scan chain to be connected between the **DBGSDIN** and **DBGSDOUT** pins. External logic, based on the **DBGTAPSM**, **DBGSCREG**, and **DBGIR** signals is required to use the EXTEST function for such a boundary scan chain. Using EXTEST with scan chain 1 or scan chain 2 selected is UNPREDICTABLE.

C.4.2 SAMPLE/PRELOAD (0011)

You must use this instruction to preload the boundary scan register with known data prior to selecting INTEST or EXTEST instructions.

Copyright © 2000 ARM Limited. All rights reserved.

C.4.3 SCAN_N (0010)

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

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

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

C.4.4 INTEST (1100)

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

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

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

 Copyright © 2000 ARM Limited. All rights reserved.

C.4.5 IDCODE (1110)

The IDCODE instruction connects the device identification code register (or ID register) between **DBGTDI and DBGTDO**. The ID register is a 32-bit register that allows the manufacturer, part number, and version of a component to be read through the TAP. See *ARM9E-S device identification (ID) code register* on page C-10 for details of the ID register format.

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

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

C.4.6 BYPASS (1111)

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

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

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

All unused instruction codes default to the BYPASS instruction.

C.4.7 RESTART (0100)

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

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

Copyright © 2000 ARM Limited. All rights reserved.

C.5 Test data registers

There are six test data registers that can be selected to connect between **DBGTDI** and **DBGTDO**:

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

In addition, other scan chains can be added between **DBGSDOUT** and **DBGSDIN**, and selected when in INTEST mode.

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

C.5.1 Bypass register

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

Length 1 bit.

Operating mode When the BYPASS instruction, or any undefined instruction, is the current instruction in the instruction register, serial data is transferred from **DBGTDI** to **DBGTDO** in the SHIFT-DR state with a delay of one **CLK** cycle enabled by **DBGTCKEN**.

A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state. There is no parallel output from the bypass register.

C.5.2 ARM9E-S device identification (ID) code register

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

Length 32 bits. The format of the ID register is shown in Figure C-3 on page C-11.

The 32-bit device identification code is loaded into the register from the parallel inputs of the **TAPID[31:0]** input pins during the CAPTURE-DR state.

Copyright © 2000 ARM Limited. All rights reserved.

The recommended generic value for **TAPID[31:0]** in a base ARM9E-S implementation is `0x15900F0F`.

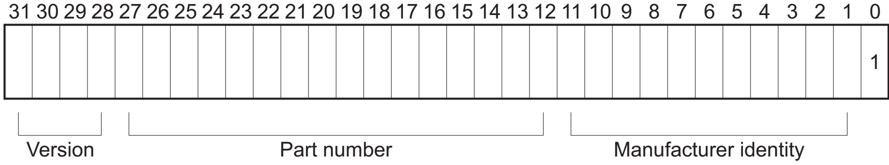


Figure C-3 ID code register format

Note

IEEE Standard 1149.1 requires that bit 0 of the ID register be set to 1.

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

There is no parallel output from the ID register.

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

C.5.3 Instruction register

Purpose Specifies a TAP instruction.

Length 4 bits.

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

During the CAPTURE-IR state, the binary value b0001 is loaded into this register. This value is shifted out during SHIFT-IR (least significant bit first), while a new instruction is shifted in (least significant bit first).

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

On reset, IDCODE specifies the current instruction.

 Copyright © 2000 ARM Limited. All rights reserved.

C.5.4 Scan path select register

Purpose Changes the current active scan chain.

Length 5 bits.

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

During the CAPTURE-DR state, the value b10000 is loaded into this register. This value is shifted out during SHIFT-DR (least significant bit first), while a new value is shifted in (least significant bit first). During the UPDATE-DR state, the value in the scan path select register selects a scan chain to become the currently active scan chain. All further instructions such as INTEST then apply to that scan chain.

The currently selected scan chain changes only when a SCAN_N instruction is executed, or when a reset occurs. On reset, scan chain 3 is selected as the active scan chain.

The number of the currently-selected scan chain is reflected on the **DBGSCREG[4:0]** output bus. You can use the TAP controller to drive external chains in addition to those within the ARM9E-S macrocell. The external scan chain is connected between **DBGSDIN** and **DBGSDOUT**, and must be assigned a number. The control signals are derived from **DBGSCREG[4:0]**, **DBGIR[4:0]**, **DBGTAPSM[3:0]** and the clock, **CLK**, and clock enable, **DBGTCKEN**.

Table C-2 lists the scan chain numbers allocated by ARM.

Table C-2 Scan chain number allocation

Scan chain number	Function
0	Reserved
1	Debug
2	EmbeddedICE-RT programming

Copyright © 2000 ARM Limited. All rights reserved.

Table C-2 Scan chain number allocation (continued)

Scan chain number	Function
3	External boundary scan
4–15	Reserved
16–31	Unassigned

The scan chain present between **DBGSDIN** and **DBGSDOUT** is connected between **DBGTDI** and **DBGTDO** whenever scan chain 3 is selected, or when any unassigned scan chain number 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 **DBGSDOUT**. The multiplexor can be controlled by decoding **DBGSCREG[4:0]**.

C.5.5 Scan chains 1 and 2

The scan chains allow serial access to the core logic and to the EmbeddedICE hardware for programming purposes. Each scan chain cell is simple, and comprises a serial register and a multiplexor. A typical cell is shown in Figure C-4.

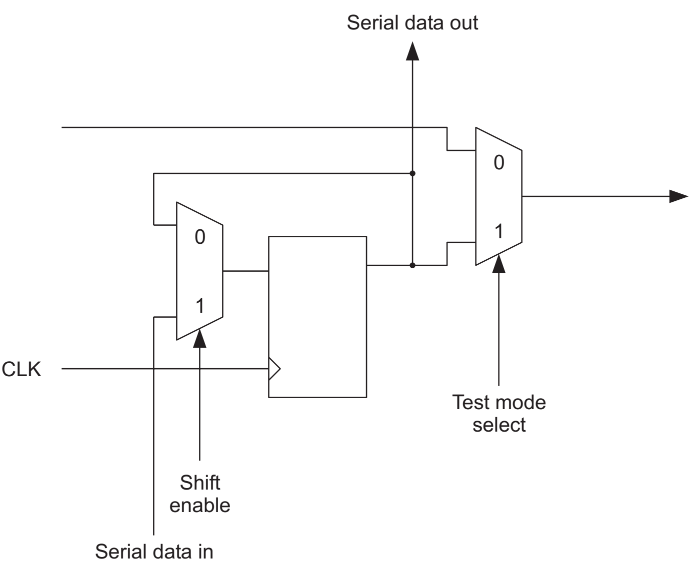


Figure C-4 Typical scan chain cell

Copyright © 2000 ARM Limited. All rights reserved.

The scan cells perform three basic functions:

- capture
- shift
- update.

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

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

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

Scan chain 1

Purpose Scan chain 1 is used for communication between the debugger and the ARM9E-S core. It is used to read and write data, and to scan instructions into the instruction pipeline. The SCAN_N instruction is used to select scan chain 1.

Length 67 bits.

Scan chain 1 provides serial access to **RDATA[31:0]** when the core is doing a read, and to the **WDATA[31:0]** bus when the core is doing a write. It also provides serial access to the **INSTR[31:0]** bus, and to the control bits, SYSPEED and WPTANDBKPT. For compatibility with the ARM9TDMI, there is one additional unused bit that must be zero when writing, and is UNPREDICTABLE when reading.

There are 67 bits in this scan chain, the order being (from serial data in to out):

1. **INSTR[31:0]**
2. SYSPEED
3. WPTANDBKPT
4. unused bit
5. **RDATA[31:0]** or **WDATA[31:0]**.

Bit 0 of **RDATA** or **WDATA** is therefore the first bit to be shifted out.

 Copyright © 2000 ARM Limited. All rights reserved.

Table C-3 shows the bit allocations for scan chain 1.

Table C-3 Scan chain 1 bit order

Bit number	Function	Type
66	**RDATA[0] /WDATA[0]**	Bidir
...	...	Bidir
35	**RDATA[31] /WDATA[31]**	Bidir
34	Unused	-
33	WPTANDBKPT	Input
32	SYSSPEED	Input
31	**INSTR[31]**	Input
...	...	Input
0	**INSTR[0]**	Input

The scan chain order is the same as for the ARM9TDMI. The unused bit is to retain compatibility with ARM9TDMI.

The two control bits serve the following purposes:

- While debugging, the value placed in the SYSSPEED control bit determines whether the ARM9E-S synchronizes back to system speed before executing the instruction. See *System speed access* on page C-26 for further details.
- After the ARM9E-S has entered debug state, the first time SYSSPEED is captured and scanned out, its value tells the debugger whether the core has entered debug state from a breakpoint (SYSSPEED LOW), or a watchpoint (SYSSPEED HIGH). If the instruction directly following one which causes a watchpoint has a breakpoint set on it, then the WPTANDBKPT bit is set. This situation does not affect how to restart the code.
- For a read the data value taken from the 32 bits in the scan chain allocated for data is used to deliver the **RDATA[31:0]** value to the core.
- When a write is being performed by the processor the **WDATA[31:0]** value is returned in the data part of the scanned out value.

Copyright © 2000 ARM Limited. All rights reserved.

Scan chain 2

Purpose Scan chain 2 allows access to the EmbeddedICE registers. To do this, scan chain 2 must be selected using the SCAN_N instruction, and then the TAP controller instruction must be changed to INTEST.

Length 38 bits.

Scan chain order From **DBGTDI** to **DBGTDO**. Read/write, register address bits 4 to 0, data values bits 31 to 0.

No action occurs during CAPTURE-DR.

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

During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read, 1 = write).

Copyright © 2000 ARM Limited. All rights reserved.

C.6 ARM9E-S core clock domains

The ARM9E-S has a single clock, **CLK**, that is qualified by two clock enables:

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

During normal operation, **CLKEN** conditions **CLK** to clock the core. When the ARM9E-S is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

 Copyright © 2000 ARM Limited. All rights reserved.

C.7 Determining the core and system state

When the ARM9E-S is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

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

C.7.1 Determining the core state

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

To force the processor into ARM state, execute the following sequence of Thumb instructions on the core (with the SYSSPEED bit set LOW):

```
STR R0, [R1]; Save R0 before use
MOV R0, PC  ; Copy PC into R0
STR R0, [R1]; Now save the PC in R0
BX  PC      ; Jump into ARM state
MOV R8, R8  ; NOP
MOV R8, R8  ; NOP
```

Note

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

You can use the sequences of ARM instructions shown in Example C-1 on page C-19 to determine the processor state.

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

```
STMIA R0, {R0-R15}
```

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

Copyright © 2000 ARM Limited. All rights reserved.

Note

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

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

The debugger must restore the original mode before exiting debug state. For example, if the debugger has been requested to return the state of the User mode registers and FIQ mode registers, and debug state is entered in Supervisor mode, the instruction sequence can be as shown in Example C-1.

Example C-1 Determining the core state

```
STMIA 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
STMIA R0, {R13,R14}; Save registers not previously visible
ORR  R0, 0x01; Select FIQ mode
MSR  CPSR, R0; Enter FIQ mode
STMIA R0, {R8-R14}; Save banked FIQ registers
```

All these instructions execute at debug speed. Debug speed is much slower than system speed. This is because between each core clock, 67 clocks occur in order to shift in an instruction, or shift out data. Executing instructions this slowly is acceptable for accessing the core state because the ARM9E-S is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, you can only scan the following ARM or Thumb instructions into the instruction pipeline for execution:

- all data processing operations
- all load, store, load multiple, and store multiple instructions
- MSR and MRS
- B, BL, and BX.

 Copyright © 2000 ARM Limited. All rights reserved.

C.7.2 Determining the system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously. Therefore, the ARM9E-S must be forced to synchronize back to system speed. Bit 32 of scan chain 1, SYSSPEED, controls this.

You can place a legal debug instruction onto the instruction data bus of scan chain 1 with bit 32 (the SYSSPEED bit) LOW. This instruction is then executed at debug speed. To execute an instruction at system speed, a NOP (such as `MOV R0, R0`) must be scanned in as the next instruction with bit 32 set HIGH.

After the system speed instructions are scanned into the instruction data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. This causes the ARM9E-S automatically to resynchronize back to **CLK** conditioned with **CLKEN** when the TAP controller enters RUN-TEST/IDLE state, and executes the instruction at system speed. Debug state is reentered once the instruction completes execution, when the processor switches itself back to **CLK** conditioned with **DBGTCKEN**. When the instruction completes, **DBGACK** is HIGH. At this point INTEST can be selected in the TAP controller, and debugging can resume.

To determine if a system speed instruction has completed, the debugger must look at SYSCOMP (bit 3 of the debug status register). The ARM9E-S must access memory through the data data bus interface, as this access can be stalled indefinitely by **CLKEN**. Therefore, the only way to determine if the memory access has completed is to examine the SYSCOMP bit. When this bit is HIGH, the instruction has completed.

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

Instructions that can have the SYSSPEED bit set

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

- loads
- stores
- load multiple
- store multiple.

When the ARM9E-S returns to debug state after a system speed access, the SYSSPEED bit is set LOW. The state of this bit gives the debugger information about why the core entered debug state the first time this scan chain is read.

 Copyright © 2000 ARM Limited. All rights reserved.

C.7.3 Exit from debug state

Leaving debug state involves:

- restoring the internal state of the ARM9E-S
- causing a branch to the next instruction to be executed
- synchronizing back to **CLK** conditioned with **CLKEN**.

After restoring the internal state, a branch instruction must be loaded into the pipeline. See *Behavior of the program counter during debug* on page C-24 for details on calculating the branch.

The SYSSPEED bit of scan chain 1 forces the ARM9E-S to resynchronize back to **CLK** conditioned with **CLKEN**. The penultimate instruction in the debug sequence is a branch to the instruction at which execution is to resume. This is scanned in with bit 32 (SYSSPEED) set LOW. The final instruction to be scanned in is a NOP (such as `MOV R0, R0`), with bit 32 set HIGH. The core is then clocked to load this instruction into the pipeline.

Next, the RESTART instruction is selected in the TAP controller. When the state machine enters the RUN-TEST/IDLE state, the scan chain reverts back to System mode, and clock resynchronization to **CLK** conditioned with **CLKEN** occurs within the ARM9E-S. Normal operation then resumes, with instructions being fetched from memory.

The delay, waiting until the state machine is in RUN-TEST/IDLE state, allows conditions to be set up in other devices in a multiprocessor system without taking immediate effect. Then, when 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 ARM9E-S is in debug state. You can use this signal to inhibit peripherals such as watchdog timers that have real-time characteristics. Also, you can use **DBGACK** to mask out memory accesses that are caused by the debugging process. For example, when the ARM9E-S enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction plus two other instructions that have been prefetched. On entry to debug state, the pipeline is flushed. So, on exit from debug state, the pipeline must be refilled to its previous state. Therefore, because of the debugging process, more memory accesses occur than are normally expected. It is possible, using the **DBGACK** signal and a small amount of external logic, for a peripheral which is sensitive to the number of memory accesses to return the same result with and without debugging.

Note

You can only use **DBGACK** in such a way using breakpoints. It does not mask the correct number of memory accesses after a watchpoint.

 Copyright © 2000 ARM Limited. All rights reserved.

For example, consider a peripheral that simply counts the number of instruction fetches. This device must return the same answer after a program has run both with and without debugging.

Figure C-5 shows the behavior of the ARM9E-S on exit from debug state.

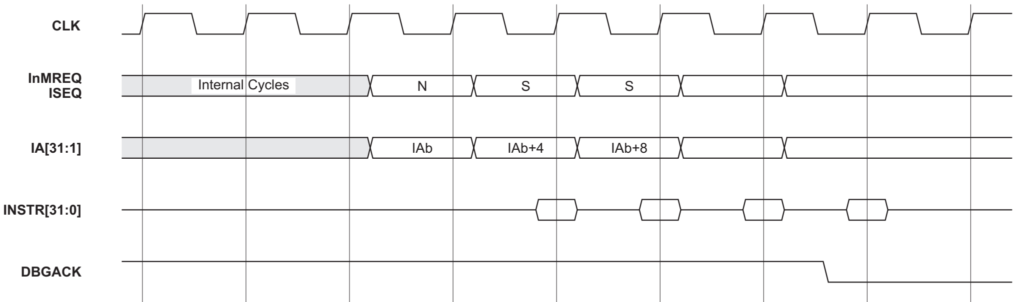


Figure C-5 Debug exit sequence

In Figure C-6 on page C-23, you can see that two instructions are fetched after the instruction which breakpoints. Figure C-5 shows that **DBGACK** masks the first three instruction fetches out of the debug state, corresponding to the breakpoint instruction, and the two instructions prefetched after it.

Under some circumstances **DBGACK** can remain HIGH for more than three instruction fetches. Therefore, if you require precise instruction access counting, you must provide some external logic to generate a modified **DBGACK** that always falls after three instruction fetches.

Note

When system speed accesses occur, **DBGACK** remains HIGH throughout. It then falls after the system speed memory accesses are completed, and finally rises again as the processor reenters debug state. Therefore, **DBGACK** masks all system speed memory accesses.

Copyright © 2000 ARM Limited. All rights reserved.

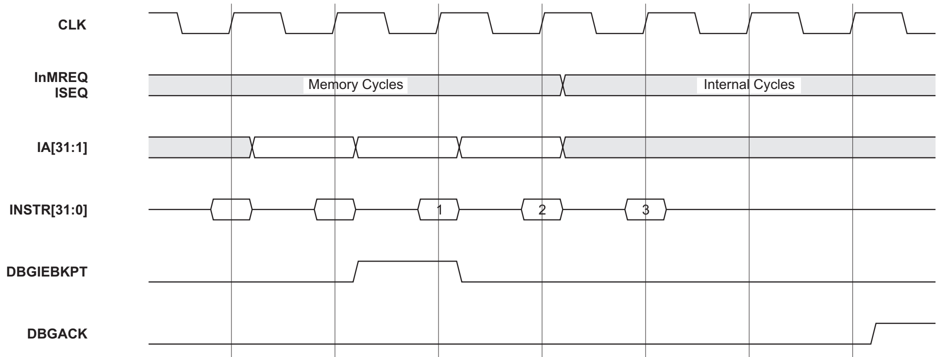


Figure C-6 Debug state entry

Copyright © 2000 ARM Limited. All rights reserved.

C.8 Behavior of the program counter during debug

The debugger must keep track of what happens to the PC, so that you can force the ARM9E-S to branch back to the place at which program flow was interrupted by debug. Program flow can be interrupted by any of the following:

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

C.8.1 Breakpoints

Entry to debug state from a breakpointed instruction advances the PC by 16 bytes in ARM state, or 8 bytes in Thumb state. Each instruction executed in debug state advances the PC by one address (4 bytes). The normal way to exit from debug state after a breakpoint is to remove the breakpoint and branch back to the previously breakpointed address.

For example, if the ARM9E-S entered debug state from a breakpoint set on a given address and two debug speed instructions were executed, a branch of seven addresses must occur (four for debug entry, plus two for the instructions, plus one for the final branch). The following sequence shows ARM instructions scanned into scan chain 1. This is the *Most Significant Bit* (MSB) first, so the first digit represents the value to be scanned into the SYSSPEED bit, followed by the instruction.

```
0 EAFFFFF9 ; B -7 addresses (two's complement)
1 E1A00000 ; NOP (MOV R0, R0), SYSSPEED bit is set
```

After the ARM9E-S enters debug state, it must execute a minimum of two instructions before the branch, although these can both be NOPs (`MOV R0, R0`). For small branches, you can replace the final branch with a subtract, with the PC as the destination (`SUB PC, PC, #28` in the above example).

C.8.2 Watchpoints

To return to program execution after entry to debug state from a watchpoint, use the same procedure described in *Breakpoints*.

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

 Copyright © 2000 ARM Limited. All rights reserved.

C.8.3 Watchpoint with another exception

If a watchpointed access also has a Data Abort returned, the ARM9E-S enters debug state in Abort mode. Entry into debug is held off until the core changes into Abort mode, and has fetched the instruction from the abort vector.

A similar sequence follows when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM9E-S enters debug state in the mode of the exception. The debugger must check to see if an exception has occurred by examining the current and previous mode (in the CPSR and SPSR), and the value of the PC. When an exception has taken place, you must be given the choice of servicing the exception before debugging.

For example, suppose that an abort has occurred on a watchpointed access and ten instructions have been executed in debug state. You can use the following sequence to return to program execution:

```
0 EAFFFFF1; B -15 addresses (two's complement)
1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set
```

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

Note

After the abort service routine, the instruction that caused the abort and watchpoint is refetched and executed. This triggers the watchpoint again, and the ARM9E-S reenters debug state.

C.8.4 Watchpoint and breakpoint

It is possible to have a watchpoint and breakpoint condition occurring simultaneously. This can happen when an instruction causes a watchpoint, and the following instruction has been breakpointed. You must perform the same calculation as for *Breakpoints* on page C-24 to determine where to resume. In this case, it is at the breakpoint instruction, because this has not been executed.

 Copyright © 2000 ARM Limited. All rights reserved.

C.8.5 Debug request

Entry into debug state through a debug request is similar to a breakpoint. Entry to debug state adds four addresses to the PC, and every instruction executed in debug state adds one address.

For example, the following sequence handles a situation in which the user has invoked a debug request, and then decides to return to program execution immediately:

```
0 EAFFFFFB; B -5 addresses (2's complement)
1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set
```

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

C.8.6 System speed access

When a system speed access is performed during debug state, the value of the PC increases by five addresses. System speed instructions access the memory system, and so it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM9E-S enters Abort mode before returning to debug state.

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

C.8.7 Summary of return address calculations

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

```
-(4+N+5S)
```

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

Copyright © 2000 ARM Limited. All rights reserved.

C.9 Priorities and exceptions

When a breakpoint or a debug request occurs, the normal flow of the program is interrupted. Therefore you can treat debug as another type of exception. The interaction of the debugger with other exceptions is described in *Behavior of the program counter during debug* on page C-24. This section covers the priorities.

C.9.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a Prefetch Abort, the abort is taken and the breakpoint is disregarded. Normally, Prefetch Aborts occur when, for example, an access is made to a virtual address that does not physically exist, and the returned data is therefore invalid. In such a case, the normal action of the operating system is to swap in the page of memory, and to return to the previously invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it might be data-dependent), the ARM9E-S enters debug state.

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

C.9.2 Interrupts

When the ARM9E-S enters debug state, interrupts are automatically disabled.

If an interrupt is pending during the instruction prior to entering debug state, the ARM9E-S enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM9E-S is in the mode expected by your program. The ARM9E-S must check the PC, the CPSR, and the SPSR to determine accurately the reason for the exception.

Debug, therefore, takes higher priority than the interrupt, but the ARM9E-S does recognize that an interrupt has occurred.

C.9.3 Data Aborts

When a Data Abort occurs on a watchpointed access, the ARM9E-S enters debug state in Abort mode. The watchpoint, therefore, has higher priority than the abort, but the ARM9E-S remembers that the abort happened.

Copyright © 2000 ARM Limited. All rights reserved.

C.10 EmbeddedICE-RT logic

The EmbeddedICE-RT logic is integral to the ARM9E-S processor core. It has two hardware breakpoint or watchpoint units, each of which can be configured to monitor either the instruction memory interface or the data memory interface. Each watchpoint unit has registers that set the address, data, and control fields for both values and masks. The registers used are shown in Table C-4.

Because the ARM9E-S processor core has a Harvard Architecture, you must specify whether the watchpoint unit examines the instruction or the data interface. This is specified by bit 3 of the control value register:

- when bit 3 is set, the data interface is examined
- when bit 3 is clear, the instruction interface is examined.

There cannot be a *don't care* case for this bit because the comparators cannot compare the values on both buses simultaneously. Therefore, bit 3 of the control mask register is always clear and cannot be programmed HIGH. Bit 3 also determines whether the internal **IBREAKPT** or **DWPT** signal must be driven by the result of the comparison. Figure C-7 on page C-30 gives an overview of the operation of the EmbeddedICE-RT logic.

The ARM9E-S EmbeddedICE-RT logic has dedicated hardware that allows single-stepping through code. This reduces the work required by an external debugger, and removes the need to flush the instruction cache. There is also hardware to allow efficient trapping of accesses to the exception vectors. These blocks of logic free the two general-purpose hardware breakpoint or watchpoint units for use by the programmer at all times.

The general arrangement of the EmbeddedICE-RT logic is shown in Figure C-7 on page C-30.

C.10.1 Register map

The EmbeddedICE-RT logic register map is shown in Table C-4.

Table C-4 ARM9E-S EmbeddedICE-RT logic register map

Address	Width	Function	Type
00000	6	Debug control	Read/write
00001	5	Debug status	Read-only
00010	8	Vector catch control	Read/write
00100	6	Debug comms control	Read-only[a]

 Copyright © 2000 ARM Limited. All rights reserved.

Table C-4 ARM9E-S EmbeddedICE-RT logic register map (continued)

Address	Width	Function	Type
00101	32	Debug comms data	Read/write
01000	32	Watchpoint 0 address value	Read/write
01001	32	Watchpoint 0 address mask	Read/write
01010	32	Watchpoint 0 data value	Read/write
01011	32	Watchpoint 0 data mask	Read/write
01100	9	Watchpoint 0 control value	Read/write
01101	8	Watchpoint 0 control mask	Read/write
10000	32	Watchpoint 1 address value	Read/write
10001	32	Watchpoint 1 address mask	Read/write
10010	32	Watchpoint 1 data value	Read/write
10011	32	Watchpoint 1 data mask	Read/write
10100	9	Watchpoint 1 control value	Read/write
10101	8	Watchpoint 1 control mask	Read/write

a. An attempted write to the comms channel control register can be used to reset bit 0 of that register.

C.10.2 Programming and reading EmbeddedICE-RT logic registers

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

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

This is shown in Figure C-7 on page C-30.

Copyright © 2000 ARM Limited. All rights reserved.

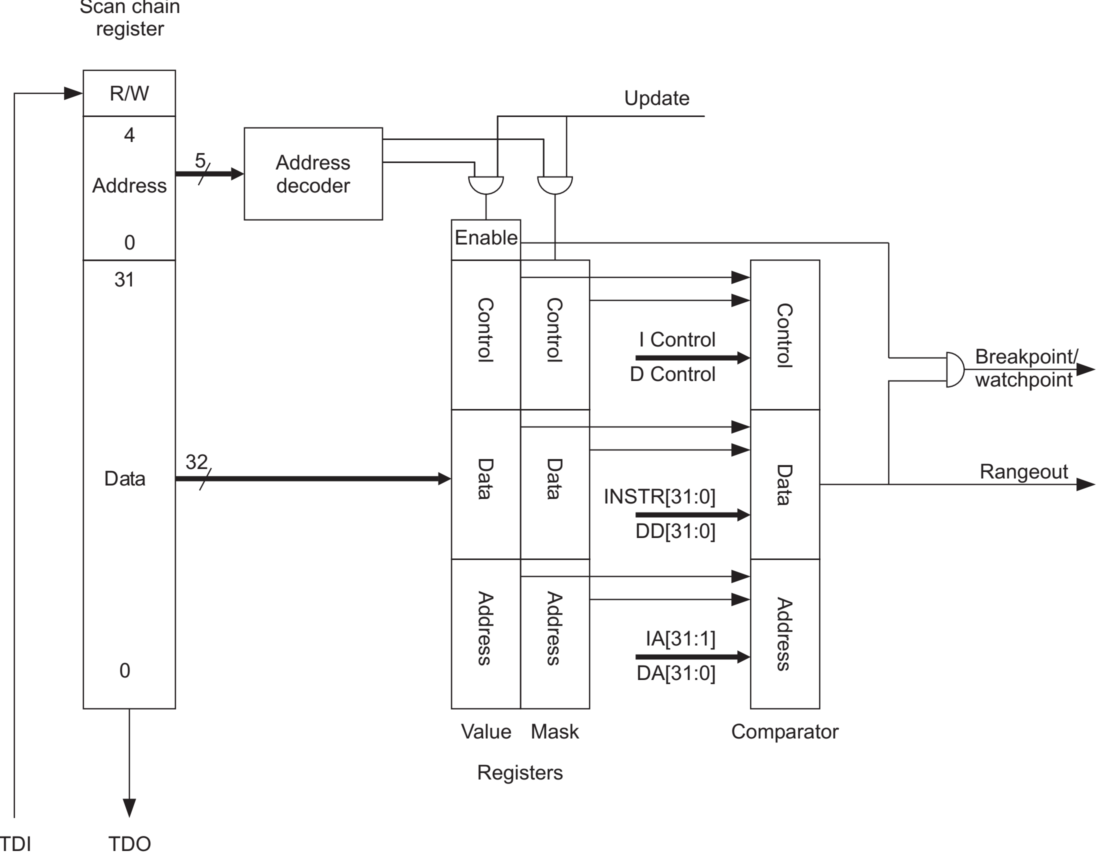


Figure C-7 ARM9E-S EmbeddedICE macrocell overview

If a watchpoint is requested on a particular memory location but the data value is irrelevant, you can program the data mask register to `0xFFFF FFFF` (all bits set to 1), so that the entire data bus value is masked.

C.10.3 Using the mask registers

For each value register there is an associated mask register in the same format. Setting a bit to 1 in the mask register causes the corresponding bit in the value register to be ignored in any comparison.

 Copyright © 2000 ARM Limited. All rights reserved.

C.10.4 Watchpoint control registers

The format of the control registers depends on how bit 3 is programmed.

If bit 3 of the control register is programmed to a 1, the breakpoint comparators examine the data address, data, and control signals.

In this case, the format of the control register is as shown in Figure C-8.

Note

You cannot mask bit 8 and bit 3.

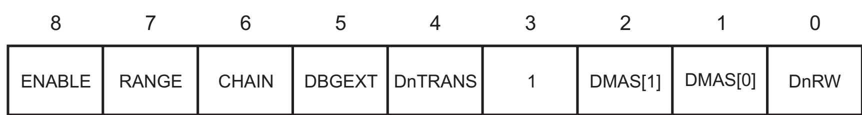


Figure C-8 Watchpoint control register for data comparison

Data comparison bit functions are described in Table C-5.

Table C-5 Watchpoint control register for data comparison functions

Bit number	Name	Function
0	**DnRW**	Compares against the data not read/write signal from the core in order to detect the direction of the data data bus activity. **DnRW** is 0 for a read, and 1 for a write.
2:1	**DMAS[1:0]**	Compares against the **DMAS[1:0]** signal from the core in order to detect the size of the data data bus activity.
4	**DnTRANS**	Compares against the data not translate signal from the core in order to determine between a User mode (**DnTRANS** = 0) data transfer, and a privileged mode (**DnTRANS** = 1) transfer.
5	**DBGEXT**	Is an external input into the EmbeddedICE-RT logic that allows the watchpoint to be dependent upon some external condition. The **DBGEXT** input for watchpoint 0 is labeled **DBGEXT[0]**, and the **DBGEXT** input for watchpoint 1 is labeled **DBGEXT[1]**.

 Copyright © 2000 ARM Limited. All rights reserved.

Table C-5 Watchpoint control register for data comparison functions (continued)

Bit number	Name	Function
6	**CHAIN**	Selects the chain output of another watchpoint unit in order to implement some debugger requests. For example, *breakpoint on address YYY only when in process XXX*. In the ARM9E-S EmbeddedICE-RT logic, 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 or 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 **DBGnTRST** is LOW.
7	**RANGE**	Can be connected to the range output of another watchpoint register. In the ARM9E-S EmbeddedICE-RT logic, the address comparator output of watchpoint 1 is connected to the **RANGE** input of watchpoint 0. This allows you to couple two watchpoints for detecting conditions that occur simultaneously, for example, for range-checking.
8	**ENABLE**	If a watchpoint match occurs, the internal **DWPT** signal is only asserted when the **ENABLE** bit is set. This bit only exists in the value register. It cannot be masked.

If bit 3 of the control register is programmed to 0, the comparators examine the instruction address, instruction data, and instruction control buses. In this case bits [2] and [0] of the mask register must be set to *don't care* (programmed to 1_1). The format of the register in this case is as shown in Figure C-9.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	DBGEXT	InTRANS	0	X	ITBIT	X

Figure C-9 Watchpoint control register for instruction comparison

Copyright © 2000 ARM Limited. All rights reserved.

Instruction comparison bit functions are described in Table C-6.

Table C-6 Watchpoint control register for instruction comparison functions

Bit number	Name	Function
1	**ITBIT**	Compares against the Thumb state signal from the core to determine between a Thumb (**ITBIT** = 1) instruction fetch or an ARM (**ITBIT** = 0) instruction fetch.
4	**InTRANS**	Compares against the not translate signal from the core in order to determine between a user mode (**InTRANS** = 0) instruction fetch, and a privileged mode (**InTRANS** = 1) fetch.
5	**DBGEXT**	Is an external input into the EmbeddedICE-RT logic that allows the watchpoint to be dependent upon some external condition. The **DBGEXT** input for watchpoint 0 is labelled **DBGEXT[0]**, and the **DBGEXT** input for watchpoint 1 is labeled **DBGEXT[1]**.
6	**CHAIN**	Selects the chain output of another watchpoint unit in order to implement some debugger requests. For example, *breakpoint on address YYY only when in process XXX*. In the ARM9E-S EmbeddedICE-RT logic, 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 or 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.
7	**RANGE**	Can be connected to the range output of another watchpoint register. In the ARM9E-S EmbeddedICE-RT logic, the address comparator output of watchpoint 1 is connected to the **RANGE** input of watchpoint 0. This allows you to couple two watchpoints for detecting conditions that occur simultaneously, for example, for range-checking.
8	**ENABLE**	If a watchpoint match occurs, the internal **IBREAKPT**signal is only asserted when the **ENABLE** bit is set. This bit only exists in the value register, it cannot be masked.

.

Copyright © 2000 ARM Limited. All rights reserved.

C.10.5 Debug control register

The debug control register is 6 bits wide. Writing control bits occurs during a register write access (with the read/write bit HIGH). Reading control bits occurs during a register read access (with the read/write bit LOW).

Figure C-10 shows the function of each bit in this register.

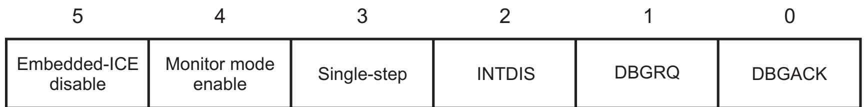


Figure C-10 Debug control register format

These functions are described in Table C-7 and Table C-8 on page C-35.

Table C-7 Debug control register bit functions

Bit number	Name	Function
5	Embedded-ICE disable	Controls the address and data comparison logic contained within the Embedded-ICE logic. When set to 1, the address and data comparators are disabled. When set to 0, the address and data comparators are enabled. You can use this bit to save power in a system where the Embedded-ICE functionality is not required. The reset state of this bit is 0 (comparators enabled). An extra piece of logic initialized by debug reset ensures that the Embedded-ICE logic is automatically disabled out of reset. This extra logic is set by debug reset and is automatically reset on the first access to scan chain 2.
4	Monitor mode enable	Controls the selection between monitor mode debug (monitor mode enable = 1) and halt mode debug. In monitor mode, breakpoints and watchpoints cause Prefetch Aborts and Data Aborts to be taken (respectively). At reset, the monitor mode enable bit is set to 1.
3	Single-step	Controls the single-step hardware. This is explained in more detail in *Single-stepping* on page C-40.
2	INTDIS	If bit 2 (**INTDIS**) is asserted, the interrupt signals to the processor are inhibited. Table C-8 shows interrupt signal control.
1:0	DBGRQ, DBGACK	These bits allow the values on **DBGRQ** and **DBGACK** to be forced.

Copyright © 2000 ARM Limited. All rights reserved.

Table C-8 Interrupt signal control

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

Both **IRQ** and **FIQ** are disabled when the processor is in debug state (DBGACK =1), or when **INTDIS** is forced.

As shown in Figure C-12 on page C-37, the value stored in bit 1 of the control register is synchronized and then ORed with the external **EDBGRQ** before being applied to the processor.

In the case of **DBGACK**, the value of **DBGACK** from the core is ORed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of the ARM9E-S. This allows the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed (in which case the internal **DBGACK** signal from the core is LOW).

The structure of the debug control and status registers is shown in Figure C-12 on page C-37.

C.10.6 Debug status register

The debug status register is five bits wide. If it is accessed for a read (with the read/write bit LOW), the status bits are read. The format of the debug status register is shown in Figure C-11.

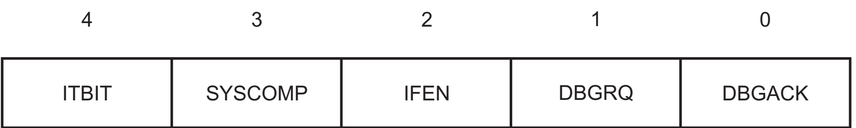


Figure C-11 Debug status register

Copyright © 2000 ARM Limited. All rights reserved.

The function of each bit in this register is shown in Table C-9.

Table C-9 Debug status register bit functions

Bit number	Name	Function
1:0	DBGRQ, DBGACK	Allow the values on the synchronized versions of **EDBGRQ** and **DBGACK** to be read.
2	IFEN	Allows the state of the core interrupt enable signal to be read.
3	SYSCOMP	Allows the state of the **SYSCOMP** bit from the core to be read. This allows the debugger to determine that a memory access from the debug state has completed.
4	ITBIT	Allows the status of the output **ITBIT** to be read. This enables the debugger to determine what state the processor is in, and therefore which instructions to execute.

The structure of the debug control and status registers is shown in Figure C-12 on page C-37.

Copyright © 2000 ARM Limited. All rights reserved.

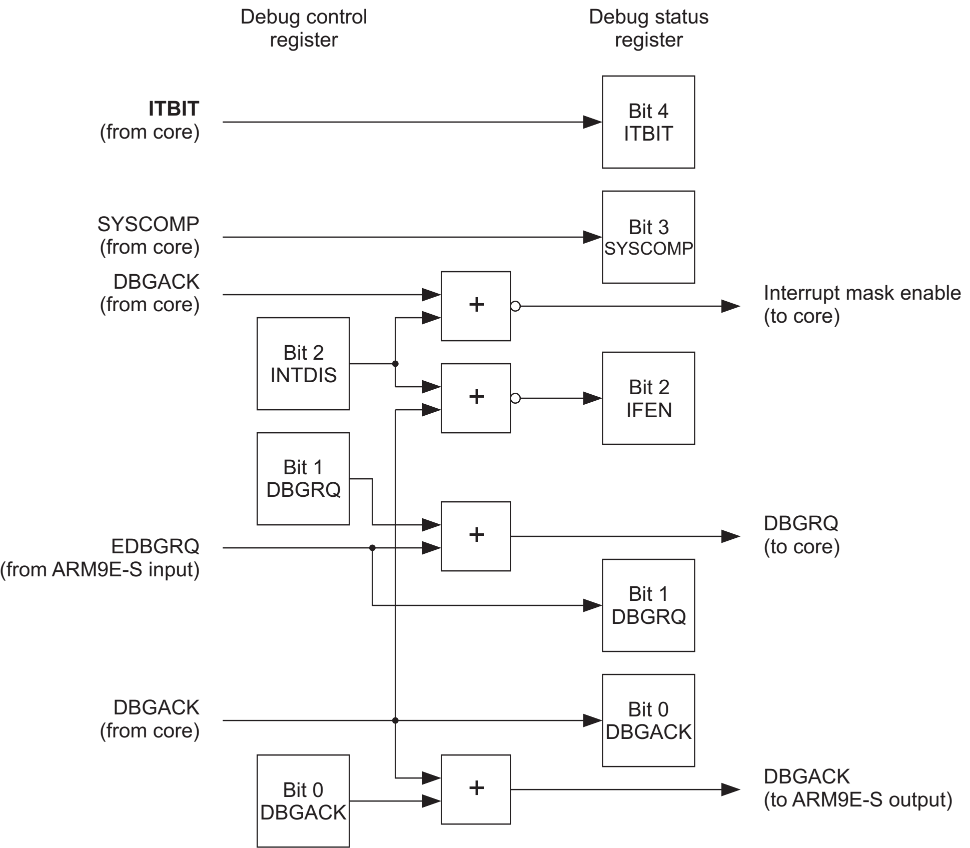


Figure C-12 Debug control and status register structure

 Copyright © 2000 ARM Limited. All rights reserved.

C.10.7 Vector catch register

The ARM9E-S EmbeddedICE-RT logic controls hardware to enable accesses to the exception vectors to be trapped in an efficient manner. This is controlled by the vector catch register, as shown in Figure C-13. The functionality is described in *Vector catching* on page C-39.

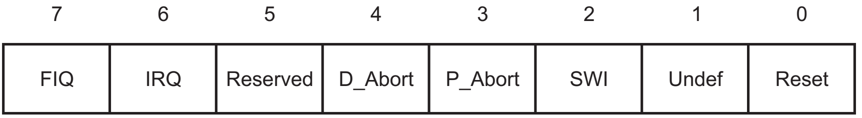


Figure C-13 Vector catch register

Copyright © 2000 ARM Limited. All rights reserved.

C.11 Vector catching

The ARM9E-S EmbeddedICE-RT logic contains hardware that allows efficient trapping of fetches from the vectors during exceptions. This is controlled by the vector catch register. If one of the bits in this register is set HIGH and the corresponding exception occurs, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector.

For example, if the processor executes a SWI instruction while bit 2 of the vector catch register is set, the ARM9E-S fetches an instruction from location `0x8`. The vector catch hardware detects this access and forces the internal **IBREAKPT** signal HIGH into the ARM9E-S control logic. This, in turn, forces the ARM9E-S to enter debug state.

The behavior of the hardware is independent of the watchpoint comparators, leaving them free for general use. The vector catch register is sensitive only to fetches from the vectors during exception entry. Therefore, if code branches to an address within the vectors during normal operation, and the corresponding bit in the vector catch register is set, the processor is not forced to enter debug state.

In monitor mode debug, vector catching is disabled on Data Aborts and Prefetch Aborts to avoid the processor being forced into an unrecoverable state as a result of the aborts that are generated for the monitor mode debug.

 Copyright © 2000 ARM Limited. All rights reserved.

C.12 Single-stepping

The ARM9E-S EmbeddedICE-RT logic contains logic that allows efficient single-stepping through code. This leaves the watchpoint comparators free for general use.

Enable this function by setting bit 3 of the debug control register. The state of this bit must only be altered while the processor is in debug state. If the processor exits debug state and this bit is HIGH, the processor fetches an instruction, executes it, and then immediately reenters debug state. This happens independently of the watchpoint comparators. If a system speed data access is performed while in debug state, the debugger must ensure that the control bit is clear first.

Note

This bit must not be set when using monitor mode debug.

Copyright © 2000 ARM Limited. All rights reserved.

C.13 Coupling breakpoints and watchpoints

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

C.13.1 Breakpoint and watchpoint coupling example

Let:

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

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

`A[31:0]` be the **IA** bus from the ARM9E-S if control register bit 3 is clear, or the **DA** bus from the ARM9E-S if control register bit 3 is set

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

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

`D[31:0]` be the **INSTR** bus from the ARM9E-S if control register bit 3 is clear, or the **RDATA** bus from the ARM9E-S if control register bit 3 is set and the processor is doing a read, or the **WDATA** bus from the ARM9E-S if control register bit 3 is set and the processor is doing a write

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

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

`C[9:0]` be the combined control bus from the ARM9E-S, other watchpoint registers, and the **DBGEXT** signal.

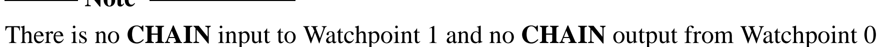
CHAINOUT signal

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

```
WHEN (({Av[31:0],Cv[4,2:0]} XNOR {A[31:0],C[4,2:0]}) OR
{Am[31:0],Cm[4:0]} == 0xFFFFFFFFF)
CHAINOUT = ((({Dv[31:0],Cv[6:4]} XNOR {D[31:0],C[7:5]}) OR
{Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFF)
```

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

Note



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

Copyright © 2000 ARM Limited. All rights reserved.

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

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

C.13.2 DBGRNG signal

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

```
DBGRNG = ((({Av[31:0],Cv[4,2:0]} XNOR {A[31:0],C[4,2:0]}) OR
{Am[31:0],Cm[4:0]}) == 0xFFFFFFFFF) AND
((({Dv[31:0],Cv[7:5]} XNOR {D[31:0],C[7:5]}) OR
Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFF)
```

The **RANGE** input to Watchpoint unit 0 is derived as the address comparison of Watchpoint unit 1, that is:

```
RANGEIN = ((Av[31:0] XNOR A[31:0]) OR Am[31:0] == 0xFFFF FFFF)
```

This **RANGE** input allows you to couple two breakpoints together to form range breakpoints.

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

For Watchpoint 1:

1. Program Watchpoint 1 with an address value of `0x00000000` and an address mask of `0X0000001F`.
2. Clear the **ENABLE** bit.
3. Program all other Watchpoint 1 registers as normal for a breakpoint.

   An address within the first 32 bytes causes the **RANGE** output to go HIGH because the address matches, but does not trigger the breakpoint because the ENABLE is LOW.

 Copyright © 2000 ARM Limited. All rights reserved.

For Watchpoint 0:

1. Program Watchpoint 0 with an address value of `0x00000000` and an address mask of `0X000000FF`.
2. Set the ENABLE bit.
3. Program the RANGE bit to match a 0.
4. Program all other Watchpoint 0 registers as normal for a breakpoint.

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

Copyright © 2000 ARM Limited. All rights reserved.

C.14 Disabling EmbeddedICE-RT

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

When **DBGEN** is LOW:

- **DBGIEBKPT**, **DBGDEWPT**, and **DBGRQ** are forced LOW to the core. (DBGRQ is the internal DBGRQ, which is a combination of the external input **EDBGRQ** and the debug control register bit 1 DBGRQ.)
- **DBGACK** is forced LOW from the ARM9E-S.
- Interrupts pass through to the processor uninhibited.

Copyright © 2000 ARM Limited. All rights reserved.

C.15 EmbeddedICE-RT timing

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

Refer to Chapter 9 *AC Parameters* for details of the required setup and hold times for these signals.

Copyright © 2000 ARM Limited. All rights reserved.

 Copyright © 2000 ARM Limited. All rights reserved.

Index

The items in this index are listed in alphabetic order. The references given are to page numbers.

A

Abort 2-23
  Data 2-23, C-27
  handler 2-24
  mode 2-8
  Prefetch 2-23, C-27
  vector C-25
Aborted watchpoint C-26
Access
  system speed C-24
  watchpointed C-25, C-27
Address bits, significant 4-7
Addressing mode 2 1-16
Addressing mode 2 (privileged) 1-17
Addressing mode 3 1-18
Addressing mode 4 (load) 1-18
Addressing mode 4 (store) 1-18
Alignment 2-7
ARM
  instruction set 1-5
  instruction set summary 1-12
  state 1-5, 2-3
ARM state to Thumb state 2-3
ARM9E-S
  architecture 1-5
  block diagram 1-7
  core diagram 1-7
  functional diagram 1-7
  instruction set 1-10
  signals compared to ARM9TDMI B-2

B

Banked registers 2-9, C-19
Big-endian 2-4
BKPT 2-25
Block diagram, ARM9E-S 1-7
Boundary-scan
  chain cells C-5
  interface C-5
Breakpoint instruction 2-25
Breakpoints 7-7, 7-9, C-24
  instruction boundary 7-10
  Prefetch Abort 7-10
Burst types 4-10
Bus cycles, CLKEN 4-31
Busy-wait 6-6, 6-17
  abandoned 6-17
  interrupted 6-17
Bypass register C-9, C-10
Byte 2-7
  access 4-21

C

C flag 2-16
CFGBIGEND A-6
CFGDISLTBIT A-6

Copyright © 2000 ARM Limited. All rights reserved.

CFGHIVECS A-6
CHAIN C-42
CHSD A-7
CHSE A-7
CLK A-2
CLKEN A-2
Clock
  domains 7-14
  maximum skew 9-8
  system 7-14
  test 7-14
Code density 1-5
Cold reset 3-3
Compression, instruction 1-5
Condition code flags 2-16
Configuration input timing 9-4
Control bits 2-17
Coprocessor
  expansion interface signals B-2
  handshake signals 6-6
  interface 6-2
  MCR 6-18
  register transfer cycle 4-29
  register transfer instructions 7-16
Coprocessor instructions
  Busy-wait 6-6
  during busy-wait 6-17
  during interrupts 6-17
  privileged instructions 6-16
  privileged modes 6-16
Core diagram, ARM9E-S 1-7
CORECLKENIN A-2
CORECLKENOUT A-2
CPSR 2-9, 2-12, 2-14, 2-16
  mode C-25
CPU reset 3-4
Current program status register 2-9, 2-12, 2-14, 2-16
Cycle
  internal 4-9, 4-11
  merged I-S 4-11
  nonsequential 4-9
  sequential 4-9

D

DA A-4
DABORT A-4
Data
  Abort 2-23, C-27
  dependencies 1-4
  interface 4-13
  memory interface timing 9-3
  types 2-7
DBGACK A-9
DBGCOMMRX A-9
DBGCOMMTX A-9
DBGDEWPT A-4
DBGEN A-9
DBGEXT A-9
DBGIEBKPT A-3
DBGINSTREXEC A-9
DBGINSTRVALID A-9
DBGIR A-8
DBGnTDOEN A-8
DBGnTRST 3-2, A-8
DBGRNG A-9
DBGRQI A-9
DBGSCREG A-8
DBGSDIN A-8
DBGTAPSM A-8
DBGTCKEN A-8
DBGTDI A-8
DBGTDO A-8
DBGTMS A-8
Debug
  comms control register 7-16
  comms data read register 7-16
  comms data write register 7-16
  control register 7-6
  entry from ARM state C-18
  entry from Thumb state C-18
  expansion signals B-6
  extensions 7-2
  hardware extensions 7-4
  interface 7-2
  interface signals 7-5
  Multi-ICE 7-14
  request C-24
  state 7-5
  state, processor restart on exit C-9
  status register 7-6
  support 7-6
Decode 1-2
Determining
  core state 7-15
  system state 7-15
Device identification code C-9, C-10
Device reset 3-2
Disabling EmbeddedICE-RT 7-8
DLOCK A-4
DMAS A-4
DMORE A-4
DnM A-5
DnMREQ A-4
DnRW A-5
DnTRANS A-5
DSEQ A-5

E

EDBGRQ A-9
EmbeddedICE-RT C-28
  debug status register 7-15
  disabling 7-8
  functionality C-28
  hardware C-28
  logic 7-4, 7-6
  operation 7-6
  overview 7-6
  programming C-2
  register map C-28
  registers, accessing C-3
  reset 3-4
  single stepping C-40
Endian effects 4-7, 4-30
Endianness 2-4
Exception
  entry and exit 2-20
  entry, ARM state 2-21
  entry, Thumb state 2-21
  priority 2-27
  vectors 2-26
Exceptions 2-20
  FIQ 2-22
  IRQ 2-22
Execute 1-2

F

F bit, FIQ disable 2-17
Fetch 1-2
FIQ
  disable, F bit 2-17
  exception 2-22
  mode 2-8
Flags 2-16
Forwarding 1-4
Functional diagram, ARM9E-S 1-7

H

Halfword 2-7
Halfword access 4-21
High registers 2-15

I

I bit, IRQ disable 2-17
IA A-3
IABORT A-3
ID register C-5, C-9, C-10
IDCODE instruction C-5, C-11
Identification register See ID register
InM A-3

Copyright © 2000 ARM Limited. All rights reserved.

InMREQ A-3
INSTR A-3
Instruction
  compression 1-5
  coprocessor register transfer 7-16
  fetch, nonsequential 4-9
  fetch, sequential 4-10
  interface 4-3
  interface cycle types 4-8
  length 2-6
  pipeline 1-2
  pipeline operation 1-4
  register C-9, C-10
  SCAN_N C-8, C-12
  system speed C-26
Instruction set
  ARM 1-5, 1-12
  summary 1-10
  Thumb 1-5
Interface
  boundary-scan C-5
  debug 7-2
Interlocking 1-4
Internal cycle 4-9, 4-11
Interrupts
  disable flags 2-21
Interworking 2-3
INTEST
  instruction C-12
  mode C-16
InTRANS A-3
IRQ
  disable, I bit 2-17
  exception 2-22
  mode 2-8
ISEQ A-3
ITBIT A-3

J

JTAG instructions
  IDCODE C-5, C-11
  INTEST C-12
  RESTART C-9
  SCAN_N C-12, C-16
  SCAN_N TAP C-14
  TAP C-11
JTAG interface 7-4, 7-5

L

LATECANCEL A-7
Link register 2-9, 2-12, 2-14
Little-endian 2-4
Low registers 2-15
LR 2-12, 2-14

M

MCR 6-18
Memory 1-2
  access 1-4
  cycle 4-8
  formats 2-4
  interface 4-2
  requests, withdrawal of 4-32
Merged I-S cycle 4-11
Mode
  abort 2-8, C-25
  bits 2-18
  FIQ 2-8
  identifier 2-10
  IRQ 2-8
  operating 2-8
  privileged 2-8
  PSR bit values 2-18
  supervisor 2-8
  System 2-8
  Undefined 2-8
  User 2-8
Multi-ICE 7-14

N

N flag 2-16
nFIQ A-6
nIRQ A-6
Nonsequential cycle 4-9
nRESET 3-2, A-6

O

Operating modes 2-8
Operating state
  ARM 2-3
  T bit 2-18
  Thumb 2-3

P

PASS A-7
PC 2-12, 2-14
Pipeline
  ARM 6-2
  coprocessor 6-2
  Pipeline follower 6-2
Power-on reset 3-3
Prefetch Abort 2-23, C-27
Priority of exceptions 2-27
Privileged modes 2-8
Processor state, determining C-18
Program counter 2-9, 2-12, 2-14
Program status registers 2-16
PSR
  control bits 2-17
  mode bit values 2-18
  reserved bits 2-19

Q

Q flag 2-17

R

RDATA A-4
Register
  banked 2-9
  current program status 2-9
  general-purpose 2-9
  high 2-15
  link 2-9
  program status 2-16
  saved program status 2-9
  status 2-9
Register, debug
  bypass C-10
  comms control 7-16
  comms data read 7-16
  comms data write 7-16
  control 7-6
  EmbeddedICE-RT debug status 7-15
  EmbeddedICE-RT, accessing C-3
  ID C-5, C-10
  instruction C-9, C-10
  scan path select C-8, C-10
  status 7-6
  test data C-10
Reserved bits, PSR 2-19
Reset 2-22, 3-2
  behavior 3-5
  CPU 3-4
  EmbeddedICE-RT 3-4
  modes 3-3
  power-on 3-3
  warm 3-4
RESTART instruction C-9
Restart on exit from debug C-9

S

Saved program status register 2-9
Scan
  cells C-14
  path C-2
  path select register C-8, C-10
Scan chains
  number allocation C-12
  scan chain 1 C-2, C-10, C-14
  scan chain 2 C-2, C-10, C-16
SCANENABLE B-5
SCANIN B-5

Copyright © 2000 ARM Limited. All rights reserved.

SCANOUT B-5
SCAN_N C-8, C-12, C-16
Sequential cycle 4-9
Serial interface, JTAG 7-4, 7-5
Signal types
  address class 4-4, 4-15
  data timed 4-6, 4-18
  debug interface 7-5
Signals
  CFGBIGEND A-6
  CFGDISLTBIT A-6
  CFGHIVECS A-6
  CHAIN C-42
  CHSD A-7
  CHSE A-7
  CLK A-2
  CLKEN A-2
  CORECLKENIN A-2
  CORECLKENOUT A-2
  DA A-4
  DABORT A-4
  DBGACK A-9
  DBGCOMMRX A-9
  DBGCOMMTX A-9
  DBGDEWPT A-4
  DBGEN A-9
  DBGEXT A-9
  DBGIEBKPT A-3
  DBGINSTREXEC A-9
  DBGINSTRVALID A-9
  DBGIR A-8
  DBGnTDOEN A-8
  DBGnTRST 3-2, A-8
  DBGRNG A-9
  DBGRQI A-9
  DBGSCREG A-8
  DBGSDIN A-8
  DBGSDOUT A-8
  DBGTAPSM A-8
  DBGTCKEN A-8
  DBGTDI A-8
  DBGTDO A-8
  DBGTMS A-8
  DLOCK A-4
  DMAS A-4
  DMORE A-4
  DnM A-5
  DnMREQ A-4
  DnRW A-5
  DnTRANS A-5
  DSEQ A-5
  EDBGRQ A-9
  IA A-3
  IABORT A-3
  InM A-3
  InMREQ A-3
  INSTR A-3
  InTRANS A-3
  ISEQ A-3
  ITBIT A-3
  LATECANCEL A-7
  nFIQ A-6
Signals (continued)
  nIRQ A-6
  nRESET 3-2, A-6
  PASS A-7
  RDATA A-4
  SCANENABLE B-5
  SCANIN B-5
  SCANOUT B-5
  TAPID A-9
  WDATA A-4
Significant address bits 4-7
Single-step core operation C-8
Single-stepping C-40
Software interrupt 2-24
SP 2-12, 2-14
SPSR 2-9, 2-12, 2-14, 2-16, C-25
Stack pointer 2-12, 2-14
State
  ARM 1-5
  debug 7-5
  Thumb 1-5
States
  core C-18
  system C-18
  TAP C-14
  TAP controller 7-2
State, switching 2-3
Status registers 2-9
Sticky overflow flag 2-17
Stored program status register 2-12, 2-14, 2-16
Supervisor mode 2-8
SWI 2-24
Switching states 2-3
System mode 2-8
System speed instruction C-26
System state, determining 7-15

T

T bit 2-18
TAP 7-2
  controller 7-4, C-2, C-3, C-14
  controller, states 7-2
  instruction C-11
  state C-14
TAPID A-9
Test Access Port 7-2
Test clock 7-14
Test data registers C-10
Thumb
  instruction set 1-5
  state 1-5, 2-3, 2-12
Thumb state to ARM state 2-3
Timing
  configuration input 9-4
  data memory interface 9-3
  exception input 9-4

U

Undefined instruction 2-25
Undefined mode 2-8
Unused instruction codes C-9
User mode 2-8

V

V flag 2-16
Vector, exception 2-26

W

Warm reset 3-4
Watchpointed
  access C-25, C-27
  memory access C-25
Watchpoints 7-6, 7-7, C-24
  aborted C-26
  timing 7-11
  with exception C-24
WDATA A-4
Word 2-7
Writeback 1-2

Z

Z flag 2-16

Copyright © 2000 ARM Limited. All rights reserved.

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Microchip:

AT91SAM9260B-QU AT91SAM9260B-CU AT91SAM9R64-CU-999