

ARM1136™

Revision: r0p1

Technical Reference Manual

ARM®

ARM1136

Technical Reference Manual

Copyright © 2002, 2003 ARM Limited. All rights reserved.

Release Information

Change history

Date	Issue	Change
December 2002	A	First Release for r0p0
February 2003	B	Internal release for r0p1
February 2003	C	First release for r0p1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the 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 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.

Confidentiality Status

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

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

ARM1136 Technical Reference Manual

Preface

About this document	xxii
Feedback	xxvii

Chapter 1

Introduction

1.1	About the ARM1136J-S and ARM1136JF-S processors	1-2
1.2	Components of the processor	1-3
1.3	Power management	1-23
1.4	Configurable options	1-25
1.5	Pipeline stages	1-26
1.6	Typical pipeline operations	1-28
1.7	ARM1136JF-S architecture with Jazelle technology	1-34
1.8	ARM1136JF-S instruction set summary	1-36
1.9	Silicon revision information	1-55

Chapter 2

Programmer's Model

2.1	About the programmer's model	2-2
2.2	Processor operating states	2-3
2.3	Instruction length	2-4
2.4	Data types	2-5
2.5	Memory formats	2-6
2.6	Addresses in an ARM1136JF-S system	2-8

2.7	Operating modes	2-9
2.8	Registers	2-10
2.9	The program status registers	2-16
2.10	Exceptions	2-23
Chapter 3	Control Coprocessor CP15	
3.1	About control coprocessor CP15	3-2
3.2	Accessing CP15 registers	3-3
3.3	Summary of control coprocessor CP15 registers	3-5
3.4	CP15 registers arranged by function	3-9
3.5	CP15 registers mapping	3-12
3.6	Cache configuration and control	3-15
3.7	Debug access to caches and TLB	3-34
3.8	DMA control	3-51
3.9	Memory management unit configuration and control	3-65
3.10	TCM configuration and control	3-83
3.11	System performance monitoring	3-87
3.12	Overall system configuration and control	3-93
Chapter 4	Unaligned and Mixed-Endian Data Access Support	
4.1	About unaligned and mixed-endian support	4-2
4.2	Unaligned access support	4-3
4.3	Unaligned data access specification	4-7
4.4	Operation of unaligned accesses	4-18
4.5	Mixed-endian access support	4-22
4.6	Instructions to reverse bytes in a general-purpose register	4-26
4.7	Instructions to change the CPSR E bit	4-27
Chapter 5	Program Flow Prediction	
5.1	About program flow prediction	5-2
5.2	Branch prediction	5-4
5.3	Return stack	5-8
5.4	Instruction Memory Barrier (IMB) instruction	5-9
5.5	ARM1020T or later IMB implementation	5-10
Chapter 6	Memory Management Unit	
6.1	About the MMU	6-2
6.2	TLB organization	6-4
6.3	Memory access sequence	6-7
6.4	Enabling and disabling the MMU	6-9
6.5	Memory access control	6-11
6.6	Memory region attributes	6-14
6.7	Memory attributes and types	6-17
6.8	MMU aborts	6-27
6.9	MMU fault checking	6-29
6.10	Fault status and address	6-33

6.11	Hardware page table translation	6-35
6.12	MMU descriptors	6-43
6.13	MMU software-accessible registers	6-55
6.14	MMU and Write Buffer	6-59
Chapter 7	Level One Memory System	
7.1	About the level one memory system	7-2
7.2	Cache organization	7-3
7.3	Tightly-coupled memory	7-8
7.4	DMA	7-11
7.5	TCM and cache interactions	7-13
7.6	Cache debug	7-17
7.7	Write Buffer	7-18
Chapter 8	Level Two Interface	
8.1	About the level two interface	8-2
8.2	Synchronization primitives	8-7
8.3	AHB-Lite control signals in the ARM1136JF-S processor	8-9
8.4	Instruction Fetch Interface AHB-Lite transfers	8-20
8.5	Data Read Interface AHB-Lite transfers	8-24
8.6	Data Write Interface AHB-Lite transfers	8-49
8.7	DMA Interface AHB-Lite transfers	8-64
8.8	Peripheral Interface AHB-Lite transfers	8-66
8.9	AHB-Lite	8-69
Chapter 9	Clocking and Resets	
9.1	ARM1136JF-S clocking	9-2
9.2	Reset	9-7
9.3	Reset modes	9-8
Chapter 10	Power Control	
10.1	About power control	10-2
10.2	Power management	10-3
Chapter 11	Coprocessor Interface	
11.1	About the ARM1136JF-S coprocessor interface	11-2
11.2	Coprocessor pipeline	11-3
11.3	Token queue management	11-12
11.4	Token queues	11-16
11.5	Data transfer	11-20
11.6	Operations	11-25
11.7	Multiple coprocessors	11-28
Chapter 12	Vectored Interrupt Controller Port	
12.1	About the PL192 Vectored Interrupt Controller	12-2
12.2	About the ARM1136JF-S VIC port	12-3

12.3	Timing of the VIC port	12-6
12.4	Interrupt entry flowchart	12-9
Chapter 13	Debug	
13.1	Debug systems	13-2
13.2	About the debug unit	13-4
13.3	Debug registers	13-7
13.4	CP14 registers reset	13-24
13.5	CP14 debug instructions	13-25
13.6	Debug events	13-28
13.7	Debug exception	13-32
13.8	Debug state	13-34
13.9	Debug communications channel	13-38
13.10	Debugging in a cached system	13-39
13.11	Debugging in a system with TLBs	13-40
13.12	Monitor mode debugging	13-41
13.13	Halt mode debugging	13-47
13.14	External signals	13-49
Chapter 14	Debug Test Access Port	
14.1	Debug Test Access Port and Halt mode	14-2
14.2	Synchronizing RealView™ ICE	14-3
14.3	Entering debug state	14-4
14.4	Exiting debug state	14-5
14.5	The DBGTap port and debug registers	14-6
14.6	Debug registers	14-8
14.7	Using the Debug Test Access Port	14-24
14.8	Debug sequences	14-34
14.9	Programming debug events	14-48
14.10	Monitor mode debugging	14-50
Chapter 15	Trace Interface Port	
15.1	About the ETM interface	15-2
Chapter 16	Cycle Timings and Interlock Behavior	
16.1	About cycle timings and interlock behavior	16-2
16.2	Register interlock examples	16-7
16.3	Data processing instructions	16-8
16.4	QADD, QDADD, QSUB, and QDSUB instructions	16-11
16.5	ARMv6 media data-processing	16-12
16.6	ARMv6 Sum of Absolute Differences (SAD)	16-14
16.7	Multiplies	16-15
16.8	Branches	16-17
16.9	Processor state updating instructions	16-18
16.10	Single load and store instructions	16-19
16.11	Load and Store Double instructions	16-22

16.12	Load and Store Multiple Instructions	16-24
16.13	RFE and SRS instructions	16-27
16.14	Synchronization instructions	16-28
16.15	Coprocessor instructions	16-29
16.16	SWI, BKPT, Undefined, Prefetch Aborted instructions	16-30
16.17	Thumb instructions	16-31
Chapter 17	AC Characteristics	
17.1	ARM1136JF-S timing diagrams	17-2
17.2	ARM1136JF-S timing parameters	17-3
Appendix A	Signal Descriptions	
A.1	Global signals	A-2
A.2	Static configuration signals	A-3
A.3	Interrupt signals (including VIC interface)	A-4
A.4	AHB interface signals	A-5
A.5	Coprocessor interface signals	A-14
A.6	Debug interface signals (including JTAG)	A-16
A.7	ETM interface signals	A-17
A.8	Test signals	A-18
	Glossary	

List of Tables

ARM1136 Technical Reference Manual

	Change history	ii
Table 1-1	Double-precision VFP operations	1-19
Table 1-2	Flush-to-zero mode	1-20
Table 1-3	Configurable options	1-25
Table 1-4	ARM1136JF-S processor default configurations	1-25
Table 1-5	Key to instruction set tables	1-36
Table 1-6	ARM instruction set summary	1-38
Table 1-7	Addressing mode 2	1-46
Table 1-8	Addressing mode 2P, post-indexed only	1-47
Table 1-9	Addressing mode 3	1-48
Table 1-10	Addressing mode 4	1-48
Table 1-11	Addressing mode 5	1-49
Table 1-12	Operand2	1-49
Table 1-13	Fields	1-50
Table 1-14	Condition codes	1-50
Table 1-15	Thumb instruction set summary	1-51
Table 2-1	Address types in an ARM1136JF-S system	2-8
Table 2-2	Register mode identifiers	2-11
Table 2-3	GE[3:0] settings	2-19
Table 2-4	PSR mode bit values	2-21
Table 2-5	Exception entry and exit	2-25
Table 2-6	Configuration of exception vector address locations	2-39
Table 2-7	Exception vectors	2-40

Table 3-1	CP15 abbreviations	3-4
Table 3-2	Summary of control coprocessor (CP15) register	3-5
Table 3-3	CP15 register functions	3-9
Table 3-4	Cache Operations Register functions	3-19
Table 3-5	Bit fields for Set/Index operations using CP15 c7	3-22
Table 3-6	Block transfer operations	3-25
Table 3-7	Enhanced cache control operations	3-26
Table 3-8	CP15 Register c7 block transfer MCR/MRC operations	3-28
Table 3-9	Cache Type Register field descriptions	3-29
Table 3-10	Ctype encoding	3-29
Table 3-11	Dsize and Lsize field summary	3-30
Table 3-12	Cache size encoding (M=0)	3-30
Table 3-13	Cache associativity encoding (M=0)	3-31
Table 3-14	Line length encoding	3-32
Table 3-15	Example Cache Type Register format	3-32
Table 3-16	Cache debug CP15 operations	3-34
Table 3-17	Cache Debug Control Register bit functions	3-35
Table 3-18	Cache and main TLB Master Valid Registers description	3-37
Table 3-19	Cache, SmartCache, and main TLB Valid bit access functions	3-38
Table 3-20	MicroTLB and main TLB debug operations	3-39
Table 3-21	Main TLB index bit functions	3-41
Table 3-22	TLB Debug Control Register bit functions	3-42
Table 3-23	TLB VA Register bit functions	3-44
Table 3-24	TLB PA Register bit functions	3-46
Table 3-25	SZ field encoding	3-46
Table 3-26	XRGN field encoding, XRGN format	3-47
Table 3-27	AP field encoding	3-47
Table 3-28	TLB Attribute Register bit functions	3-48
Table 3-29	Upper subpage access permission field encoding	3-49
Table 3-30	XRGN field encoding, RGN format	3-49
Table 3-31	DMA registers	3-51
Table 3-32	DMA Channel Status Register bit functions	3-54
Table 3-33	DMA Control Register bit functions	3-57
Table 3-34	DMA Channel Enable Register operations	3-59
Table 3-35	DMA Identification and Status Register functions	3-62
Table 3-36	Data Fault Status Register bits	3-66
Table 3-37	Encoding of domain bits in CP15 c3	3-67
Table 3-38	IFSR bits	3-68
Table 3-39	Memory Region Remap Register fields	3-70
Table 3-40	Inner region remap encoding	3-71
Table 3-41	Outer region remap encoding	3-71
Table 3-42	Default memory regions when MMU is disabled	3-72
Table 3-43	Peripheral Port Memory Remap Register bit functions	3-73
Table 3-44	Size field encoding	3-73
Table 3-45	TLB Type Register field descriptions	3-75
Table 3-46	TLB Operations Register instructions	3-75
Table 3-47	CRm values for TLB Operations Register	3-76

Table 3-48	Values of N for Translation Table Base Register 0	3-80
Table 3-49	Translation Table Base Register 0 bits	3-81
Table 3-50	Translation Table Base Register 1 bits	3-82
Table 3-51	Data TCM Region Register bits	3-84
Table 3-52	Size field encoding	3-85
Table 3-53	Instruction TCM Region Register bits	3-86
Table 3-54	Size field encoding	3-86
Table 3-55	Performance Monitor Control Register bit functions	3-88
Table 3-56	Performance monitoring events	3-89
Table 3-57	Auxiliary Control Register bit functions	3-93
Table 3-58	Coprocessor access rights	3-95
Table 3-59	B bit, U bit, and EE bit settings	3-97
Table 3-60	Control Register bit functions	3-97
Table 3-61	Register 0, ID Code	3-102
Table 4-1	Unaligned access handling	4-4
Table 4-2	Access type descriptions	4-18
Table 4-3	Unalignment fault occurrence when access behavior is architecturally unpredictable	4-19
Table 4-4	Legacy endianness using CP15 c1	4-22
Table 4-5	Mixed-endian configuration	4-24
Table 4-6	B bit, U bit, and EE bit settings	4-25
Table 6-1	Access permission bit encoding	6-12
Table 6-2	TEX field, and C and B bit encodings used in page table formats	6-14
Table 6-3	Cache policy bits	6-15
Table 6-4	Inner and Outer cache policy implementation options	6-16
Table 6-5	Memory attributes	6-17
Table 6-6	Memory ordering restrictions	6-23
Table 6-7	Memory region backwards compatibility	6-26
Table 6-8	Fault Status Register encoding	6-33
Table 6-9	Summary of aborts	6-34
Table 6-10	Access types from first-level descriptor bit values	6-45
Table 6-11	Access types from second-level descriptor bit values	6-48
Table 6-12	CP15 register functions	6-55
Table 7-1	Summary of data accesses to TCM and caches	7-15
Table 7-2	Summary of instruction accesses to TCM and caches	7-16
Table 8-1	HTRANS[1:0] settings	8-9
Table 8-2	HSIZE[2:0] encoding	8-10
Table 8-3	HBURST[2:0] settings	8-10
Table 8-4	HPROT[1:0] encoding	8-11
Table 8-5	HPROT[4:2] encoding	8-11
Table 8-6	HRESP[2:0] mnemonics	8-14
Table 8-7	Mapping of HBSTRB to HWDATA bits for a 64-bit interface	8-16
Table 8-8	Byte lane strobes for example ARMv6 transfers	8-17
Table 8-9	AHB-Lite signals for Cachable fetches	8-20
Table 8-10	AHB-Lite signals for Noncachable fetches	8-21
Table 8-11	HPROTI[4:2] encoding	8-22
Table 8-12	HPROTI[1] encoding	8-23

Table 8-13	HSEBANDI[3:1] encoding	8-23
Table 8-14	Linefills	8-25
Table 8-15	Noncacheable LDRB	8-26
Table 8-16	Noncacheable LDRH	8-26
Table 8-17	Noncacheable LDR or LDM1	8-27
Table 8-18	Noncacheable LDM2 from word 0	8-28
Table 8-19	Noncacheable LDM2 from word 1	8-28
Table 8-20	Noncacheable LDM2 from word 2	8-28
Table 8-21	Noncacheable LDM2 from word 3	8-29
Table 8-22	Noncacheable LDM2 from word 4	8-29
Table 8-23	Noncacheable LDM2 from word 5	8-29
Table 8-24	Noncacheable LDM2 from word 6	8-29
Table 8-25	Noncacheable LDM2 from word 7	8-29
Table 8-26	Noncacheable LDM3 from word 0, Strongly Ordered or Device memory	8-30
Table 8-27	Noncacheable LDM3 from word 0, Noncacheable memory or cache disabled	8-30
Table 8-28	Noncacheable LDM3 from word 1, Strongly Ordered or Device memory	8-30
Table 8-29	Noncacheable LDM3 from word 1, Noncacheable memory or cache disabled	8-31
Table 8-30	Noncacheable LDM3 from word 2, Strongly Ordered or Device memory	8-31
Table 8-31	Noncacheable LDM3 from word 2, Noncacheable memory or cache disabled	8-31
Table 8-32	Noncacheable LDM3 from word 3, Strongly Ordered or Device memory	8-31
Table 8-33	Noncacheable LDM3 from word 3, Noncacheable memory or cache disabled	8-32
Table 8-34	Noncacheable LDM3 from word 4, Strongly Ordered or Device memory	8-32
Table 8-35	Noncacheable LDM3 from word 4, Noncacheable memory or cache disabled	8-32
Table 8-36	Noncacheable LDM3 from word 5, Strongly Ordered or Device memory	8-32
Table 8-37	Noncacheable LDM3 from word 5, Noncacheable memory or cache disabled	8-33
Table 8-38	Noncacheable LDM3 from word 6 or 7, Noncacheable memory or cache disabled	8-33
Table 8-39	Noncacheable LDM4 from word 0	8-33
Table 8-40	Noncacheable LDM4 from word 1, Strongly Ordered or Device memory	8-33
Table 8-41	Noncacheable LDM4 from word 1, Noncacheable memory or cache disabled	8-34
Table 8-42	Noncacheable LDM4 from word 2	8-34
Table 8-43	Noncacheable LDM4 from word 3, Strongly Ordered or Device memory	8-34

Table 8-44	Noncachable LDM4 from word 3, Noncachable memory or cache disabled	8-34
Table 8-45	Noncachable LDM4 from word 4	8-35
Table 8-46	Noncachable LDM4 from word 5, 6, or 7	8-35
Table 8-47	Noncachable LDM5 from word 0, Strongly Ordered or Device memory	8-35
Table 8-48	Noncachable LDM5 from word 0, Noncachable memory or cache disabled	8-35
Table 8-49	Noncachable LDM5 from word 1, Strongly Ordered or Device memory	8-36
Table 8-50	Noncachable LDM5 from word 1, Noncachable memory or cache disabled	8-36
Table 8-51	Noncachable LDM5 from word 2, Strongly Ordered or Device memory	8-36
Table 8-52	Noncachable LDM5 from word 2, Noncachable memory or cache disabled	8-37
Table 8-53	Noncachable LDM5 from word 3, Strongly Ordered or Device memory	8-37
Table 8-54	Noncachable LDM5 from word 3, Noncachable memory or cache disabled	8-37
Table 8-55	Noncachable LDM5 from word 4, 5, 6, or 7	8-38
Table 8-56	Noncachable LDM6 from word 0	8-38
Table 8-57	Noncachable LDM6 from word 1, Strongly Ordered or Device memory	8-38
Table 8-58	Noncachable LDM6 from word 1, Noncachable memory or cache disabled	8-39
Table 8-59	Noncachable LDM6 from word 2	8-39
Table 8-60	Noncachable LDM6 from word 3, 4, 5, 6, or 7	8-39
Table 8-61	Noncachable LDM7 from word 0, Strongly Ordered or Device memory	8-40
Table 8-62	Noncachable LDM7 from word 0, Noncachable memory or cache disabled	8-40
Table 8-64	Noncachable LDM7 from word 1, Noncachable memory or cache disabled	8-41
Table 8-65	Noncachable LDM7 from word 2, 3, 4, 5, 6, or 7	8-41
Table 8-63	Noncachable LDM7 from word 1, Strongly Ordered or Device memory	8-41
Table 8-66	Noncachable LDM8 from word 0	8-42
Table 8-67	Noncachable LDM8 from word 1, 2, 3, 4, 5, 6, or 7	8-42
Table 8-68	Noncachable LDM9	8-43
Table 8-69	Noncachable LDM10	8-43
Table 8-70	Noncachable LDM11	8-44
Table 8-71	Noncachable LDM12	8-44
Table 8-72	Noncachable LDM13	8-45
Table 8-73	Noncachable LDM14	8-45
Table 8-74	Noncachable LDM15	8-46
Table 8-75	Noncachable LDM16	8-46

Table 8-76	Cachable swap	8-47
Table 8-77	Noncachable swap	8-47
Table 8-78	Page table walks	8-47
Table 8-79	HSIDEBAND[3:1] encoding	8-48
Table 8-80	Cachable or Noncachable Write-Through STRB	8-49
Table 8-81	Cachable or Noncachable Write-Through STRH	8-49
Table 8-82	Cachable or Noncachable Write-Through STR or STM1	8-50
Table 8-83	Cachable or Noncachable Write-Through STM2 to words 0, 1, 2, 3, 4, 5, or 6	8-51
Table 8-84	Cachable or Noncachable Write-Through STM2 to word 7	8-52
Table 8-85	Cachable or Noncachable Write-Through STM3 to words 0, 1, 2, 3, 4, or 5	8-52
Table 8-86	Cachable or Noncachable Write-Through STM3 to words 6 or 7	8-53
Table 8-87	Cachable or Noncachable STM4 to word 0, 1, 2, 3, or 4	8-53
Table 8-88	Cachable or Noncachable STM4 to word 5, 6, or 7	8-53
Table 8-89	Cachable or Noncachable STM5 to word 0, 1, 2, or 3	8-54
Table 8-90	Cachable or Noncachable STM5 to word 4, 5, 6, or 7	8-55
Table 8-91	Cachable or Noncachable STM6 to word 0, 1, or 2	8-55
Table 8-92	Cachable or Noncachable STM6 to word 3, 4, 5, 6, or 7	8-55
Table 8-93	Cachable or Noncachable STM7 to word 0 or 1	8-56
Table 8-94	Cachable or Noncachable STM7 to word 2, 3, 4, 5, 6, or 7	8-56
Table 8-95	Cachable or Noncachable STM8 to word 0	8-57
Table 8-96	Cachable or Noncachable STM8 to word 1, 2, 3, 4, 5, 6, or 7	8-57
Table 8-97	Cachable or Noncachable STM9	8-57
Table 8-98	Cachable or Noncachable STM10	8-58
Table 8-99	Cachable or Noncachable STM11	8-58
Table 8-100	Cachable or Noncachable STM12	8-59
Table 8-101	Cachable or Noncachable STM13	8-59
Table 8-102	Cachable or Noncachable STM14	8-60
Table 8-103	Cachable or Noncachable STM15	8-60
Table 8-104	Cachable or Noncachable STM16	8-60
Table 8-105	Half-line Write-Back	8-61
Table 8-106	Full-line Write-Back	8-62
Table 8-107	HSIDEBANDW[3:1] encoding	8-63
Table 8-108	HPROTD[4:2] encoding	8-64
Table 8-109	HPROTD[1] encoding	8-65
Table 8-110	HPROTD[0] encoding	8-65
Table 8-111	HSIDEBANDD[3:1] encoding	8-65
Table 8-112	Example Peripheral Interface reads and writes	8-66
Table 8-113	HPROTP[4:2] encoding	8-67
Table 8-114	HPROTP[1] encoding	8-68
Table 8-115	AHB-Lite interchangeability	8-70
Table 9-1	AHB clock domains	9-2
Table 9-2	Clock domain control signals	9-3
Table 9-3	Synchronous mode clock enable signals	9-5
Table 9-4	Reset modes	9-8
Table 11-1	Coprocessor instructions	11-3

Table 11-2	Coprocessor control signals	11-5
Table 11-3	Pipeline stage update	11-10
Table 11-4	Addressing of queue buffers	11-13
Table 11-5	Retirement conditions	11-27
Table 12-1	VIC port signals	12-3
Table 13-1	Terms used in register descriptions	13-7
Table 13-2	CP14 debug register map	13-8
Table 13-3	Debug ID Register bit field definition	13-9
Table 13-4	Debug Status And Control Register bit field definitions	13-11
Table 13-5	Data Transfer Register bit field definitions	13-14
Table 13-6	Vector Catch Register bit field definitions	13-15
Table 13-7	ARM1136JF-S breakpoint and watchpoint registers	13-17
Table 13-8	Breakpoint Value Registers, bit field definition	13-17
Table 13-9	Breakpoint Control Registers, bit field definitions	13-18
Table 13-10	Meaning of BCR[21:20] bits	13-20
Table 13-11	Watchpoint Value Registers, bit field definitions	13-21
Table 13-12	Watchpoint Control Registers, bit field definitions	13-22
Table 13-13	CP14 debug instructions	13-25
Table 13-14	Debug instruction execution	13-27
Table 13-15	Behavior of the processor on debug events	13-30
Table 13-16	Setting of CP15 registers on debug events	13-31
Table 13-17	Values in the link register after exceptions	13-33
Table 13-18	Read PC value after debug state entry	13-35
Table 14-1	Supported public instructions	14-6
Table 14-2	Scan chain 7 register map	14-21
Table 15-1	Instruction interface signals	15-2
Table 15-2	ETMIACTL[17:0]	15-3
Table 15-3	Data address interface signals	15-4
Table 15-4	ETMDACTL[17:0]	15-5
Table 15-5	Data value interface signals	15-6
Table 15-6	ETMDCTL[3:0]	15-6
Table 15-7	ETMPADV[2:0]	15-7
Table 15-8	Coprocessor interface signals	15-7
Table 15-9	Other connections	15-9
Table 16-1	Pipeline stages	16-3
Table 16-2	Definition of cycle timing terms	16-6
Table 16-3	Register interlock examples	16-7
Table 16-4	Data Processing Instruction cycle timing behavior if destination is not PC	16-8
Table 16-5	Data Processing Instruction cycle timing behavior if destination is the PC	16-8
Table 16-6	QADD, QDADD, QSUB, and QDSUB instruction cycle timing behavior	16-11
Table 16-7	ARMv6 media data-processing instructions cycle timing behavior	16-12
Table 16-8	ARMv6 sum of absolute differences instruction timing behavior	16-14
Table 16-9	Example interlocks	16-14
Table 16-10	Example multiply instruction cycle timing behavior	16-15
Table 16-11	Branch instruction cycle timing behavior	16-17
Table 16-12	Processor state updating instructions cycle timing behavior	16-18
Table 16-13	Cycle timing behavior for stores and loads, other than loads to the PC	16-19

Table 16-14	Cycle timing behavior for loads to the PC	16-20
Table 16-15	<addr_md_1cycle> and <addr_md_2cycle> LDR example instruction explanation	16-21
Table 16-16	Load and Store Double instructions cycle timing behavior	16-22
Table 16-17	<addr_md_1cycle> and <addr_md_2cycle> LDRD example instruction explanation	16-23
Table 16-18	Cycle timing behavior of Load and Store Multiples, other than load multiples including the PC	16-24
Table 16-19	Cycle timing behavior of Load Multiples, where the PC is in the register list	16-26
Table 16-20	RFE and SRS instructions cycle timing behavior	16-27
Table 16-21	Synchronization Instructions cycle timing behavior	16-28
Table 16-22	Coprocessor Instructions cycle timing behavior	16-29
Table 16-23	SWI, BKPT, undefined, prefetch aborted instructions cycle timing behavior	16-30
Table 17-1	AHB-Lite bus interface timing parameters	17-3
Table 17-2	Coprocessor port timing parameters	17-4
Table 17-3	ETM interface port timing parameters	17-5
Table 17-4	Interrupt port timing parameters	17-5
Table 17-5	Debug timing parameters	17-5
Table 17-6	test port timing parameters	17-6
Table 17-7	Static configuration signal port timing parameters	17-6
Table 17-8	Reset port timing parameters	17-7
Table A-1	Global signals	A-2
Table A-2	Static configuration signals	A-3
Table A-3	Interrupt signals	A-4
Table A-4	Port signal name suffixes	A-5
Table A-5	Instruction fetch port signals	A-6
Table A-6	Data read port signals	A-7
Table A-7	Data write port signals	A-9
Table A-8	Peripheral port signals	A-10
Table A-9	DMA port signals	A-12
Table A-10	Core to coprocessor signals	A-14
Table A-11	Coprocessor to core signals	A-15
Table A-12	Debug interface signals	A-16
Table A-13	ETM interface signals	A-17
Table A-14	Test signals	A-18

List of Figures

ARM1136 Technical Reference Manual

	Key to timing diagram conventions	xxv
Figure 1-1	ARM1136JF-S processor block diagram	1-4
Figure 1-2	ARM1136JF-S pipeline stages	1-26
Figure 1-3	Typical operations in pipeline stages	1-28
Figure 1-4	Typical ALU operation	1-29
Figure 1-5	Typical multiply operation	1-30
Figure 1-6	Progression of an LDR/STR operation	1-31
Figure 1-7	Progression of an LDM/STM operation	1-32
Figure 1-8	Progression of an LDR that misses	1-33
Figure 2-1	Big-endian addresses of bytes within words	2-6
Figure 2-2	Little-endian addresses of bytes within words	2-7
Figure 2-3	Register organization in ARM state	2-12
Figure 2-4	ARM1136JF-S register set showing banked registers	2-13
Figure 2-5	Register organization in Thumb state	2-14
Figure 2-6	ARM state and Thumb state registers relationship	2-15
Figure 2-7	Program status register	2-16
Figure 3-1	CP15 MRC and MCR bit pattern	3-3
Figure 3-2	CP15 register map, part one	3-12
Figure 3-3	CP15 register map, part two	3-13
Figure 3-4	CP15 register map, part three	3-14
Figure 3-5	Instruction and Data Cache Lockdown Registers format	3-16
Figure 3-6	Accessing the Cache Operations Register	3-21
Figure 3-7	Register 7 Set/Index format	3-22

Figure 3-8	CP15 Register c7 MVA format	3-23
Figure 3-9	CP15 c7 MVA format for Flush Branch Target Cache Entry function	3-23
Figure 3-10	Cache Dirty Status Register format	3-25
Figure 3-11	Block Address Register format	3-27
Figure 3-12	Block Transfer Status Register format	3-28
Figure 3-13	Cache Type Register format	3-29
Figure 3-14	Dsize and Isize field format	3-30
Figure 3-15	Cache Debug Control Register format	3-35
Figure 3-16	Instruction and Data Debug Cache Register format	3-35
Figure 3-17	Index/Set/Word format	3-36
Figure 3-18	MicroTLB index format	3-40
Figure 3-19	Main TLB index format	3-41
Figure 3-20	TLB Debug Control Register format	3-42
Figure 3-21	TLB VA Registers format	3-44
Figure 3-22	Memory space identifier format	3-45
Figure 3-23	TLB PA Registers format	3-45
Figure 3-24	TLB Attribute Register format	3-48
Figure 3-25	DMA registers	3-52
Figure 3-26	DMA Channel Number Register format	3-53
Figure 3-27	DMA Channel Status Register format	3-54
Figure 3-28	DMA Context ID Register format	3-56
Figure 3-29	DMA Control Register format	3-57
Figure 3-30	DMA Identification and Status Registers format	3-62
Figure 3-31	DMA User Accessibility Register format	3-64
Figure 3-32	Data Fault Status Register format	3-66
Figure 3-33	Domain Access Control Register format	3-67
Figure 3-34	IFSR format	3-68
Figure 3-35	Instruction, Data, and DMA Memory Remap Registers format	3-70
Figure 3-36	Peripheral Port Memory Remap Register format	3-73
Figure 3-37	TLB Type Register format	3-75
Figure 3-38	TLB Operations Register Virtual Address format	3-76
Figure 3-39	TLB Operations Register ASID format	3-76
Figure 3-40	TLB Lockdown Register format	3-78
Figure 3-41	Translation Table Base Control Register format	3-79
Figure 3-42	Translation Table Base Register 0 format	3-80
Figure 3-43	Translation Table Base Register 1 format	3-81
Figure 3-44	TCM Status Register format	3-83
Figure 3-45	Data TCM Region Register format	3-84
Figure 3-46	Instruction TCM Region Register format	3-85
Figure 3-47	Performance Monitor Control Register format	3-87
Figure 3-48	Auxiliary Control Register format	3-93
Figure 3-49	Coprocessor Access Control Register format	3-94
Figure 3-50	Context ID Register format	3-96
Figure 3-51	Control Register format	3-97
Figure 3-52	FCSE PID Register format	3-100
Figure 3-53	Address mapping using CP15 c13	3-101
Figure 4-1	Load unsigned byte	4-7

Figure 4-2	Load signed byte	4-8
Figure 4-3	Store byte	4-8
Figure 4-4	Load unsigned halfword, little-endian	4-9
Figure 4-5	Load unsigned halfword, big-endian	4-9
Figure 4-6	Load signed halfword, little-endian	4-10
Figure 4-7	Load signed halfword, big-endian	4-11
Figure 4-8	Store halfword, little-endian	4-11
Figure 4-9	Store halfword, big-endian	4-12
Figure 4-10	Load word, little-endian	4-13
Figure 4-11	Load word, big-endian	4-14
Figure 4-12	Store word, little-endian	4-15
Figure 4-13	Store word, big-endian	4-16
Figure 6-1	Translation table managed TLB fault checking sequence	6-30
Figure 6-2	Backwards-compatible first-level descriptor format	6-36
Figure 6-3	Backwards-compatible second-level descriptor format	6-37
Figure 6-4	Backwards-compatible section, supersection, and page translation	6-38
Figure 6-5	ARMv6 first-level descriptor formats with subpages enabled	6-39
Figure 6-6	ARMv6 first-level descriptor formats with subpages disabled	6-40
Figure 6-7	ARMv6 second-level descriptor format	6-40
Figure 6-8	ARMv6 section, supersection, and page translation	6-41
Figure 6-9	Creating a first-level descriptor address	6-44
Figure 6-10	Translation for a 1MB section, ARMv6 format	6-46
Figure 6-11	Translation for a 1MB section, backwards-compatible format	6-47
Figure 6-12	Generating a second-level page table address	6-48
Figure 6-13	Large page table walk, ARMv6 format	6-50
Figure 6-14	Large page table walk, backwards-compatible format	6-51
Figure 6-15	4KB small page or 1KB small subpage translations, backwards-compatible format	6-52
Figure 6-16	4KB extended small page translations, ARMv6 format	6-53
Figure 6-17	4KB extended small page or 1KB extended small subpage translations, backwards-compatible format	6-54
Figure 7-1	Level one cache block diagram	7-4
Figure 8-1	Level two interconnect interfaces	8-2
Figure 8-2	Synchronization penalty	8-3
Figure 8-3	Exclusive access read and write with Okay response	8-18
Figure 8-4	Exclusive access read and write with Xfail response	8-18
Figure 8-5	Exclusive access read and write with Xfail response and following transfer	8-19
Figure 8-6	AHB-Lite single-master system	8-69
Figure 8-7	AHB-Lite block diagram	8-72
Figure 9-1	Synchronization between AHB and core clock domains	9-4
Figure 9-2	Synchronization between core clock and AHB domains	9-4
Figure 9-3	Read latency for synchronous 1:1 clocking	9-5
Figure 9-4	Power-on reset	9-8
Figure 11-1	Core and coprocessor pipelines	11-6
Figure 11-2	Coprocessor pipeline and queues	11-7
Figure 11-3	Coprocessor pipeline	11-9
Figure 11-4	Token queue buffers	11-12

Figure 11-5	Queue reading and writing	11-14
Figure 11-6	Queue flushing	11-15
Figure 11-7	Instruction queue	11-16
Figure 11-8	Coprocessor data transfer	11-20
Figure 11-9	Instruction iteration for loads	11-21
Figure 11-10	Load data buffering	11-22
Figure 12-1	Connection of a PL192 VIC to an ARM1136JF-S processor	12-3
Figure 12-2	VIC port timing example	12-6
Figure 12-3	Interrupt entry sequence	12-9
Figure 13-1	Typical debug system	13-2
Figure 13-2	Debug ID Register format	13-9
Figure 13-3	Debug Status And Control Register format	13-11
Figure 13-4	DTR format	13-14
Figure 13-5	Vector Catch Register format	13-15
Figure 13-6	Breakpoint Control Registers, format	13-18
Figure 13-7	Watchpoint Control Registers, format	13-22
Figure 14-1	JTAG DBGTAP state machine diagram	14-2
Figure 14-2	Clock synchronization	14-3
Figure 14-3	Bypass register bit order	14-8
Figure 14-4	Device ID code register bit order	14-9
Figure 14-5	Instruction register bit order	14-10
Figure 14-6	Scan chain select register bit order	14-11
Figure 14-7	Scan chain 0 bit order	14-12
Figure 14-8	Scan chain 1 bit order	14-13
Figure 14-9	Scan chain 4 bit order	14-15
Figure 14-10	Scan chain 5 bit order, EXTEST selected	14-16
Figure 14-11	Scan chain 5 bit order, INTEST selected	14-17
Figure 14-12	Scan chain 6 bit order	14-19
Figure 14-13	Scan chain 7 bit order	14-20
Figure 14-14	Behavior of the ITRsel IR instruction	14-26
Figure 15-1	ETMCPADDRESS format	15-8

Preface

This preface introduces the *ARM1136 r0p1 Technical Reference Manual*. It contains the following sections:

- *About this document* on page xxii
- *Feedback* on page xxvii.

About this document

This document is the technical reference manual for the ARM1136JF-S and ARM1136J-S processors. Because the ARM1136JF-S and ARM1136J-S processors are similar, only the ARM1136JF-S processor is described. Any differences are described where necessary.

Intended audience

This document has been written for hardware and software engineers implementing ARM1136JF-S or ARM1136J-S processor system designs. It provides information to enable designers to integrate the processor into a target system as quickly as possible.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM1136JF-S processor and descriptions of the major functional blocks.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the ARM1136JF-S registers and programming details.

Chapter 3 *Control Coprocessor CP15*

Read this chapter for a description of the ARM1136JF-S control coprocessor CP15 registers and programming details.

Chapter 4 *Unaligned and Mixed-Endian Data Access Support*

Read this chapter for a description of the ARM1136JF-S processor support for unaligned and mixed-endian data accesses.

Chapter 5 *Program Flow Prediction*

Read this chapter for a description of the functions of the ARM1136JF-S Prefetch Unit, including static and dynamic branch prediction and the return stack.

Chapter 6 *Memory Management Unit*

Read this chapter for a description of the ARM1136JF-S *Memory Management Unit* (MMU) and the address translation process.

Chapter 7 *Level One Memory System*

Read this chapter for a description of the ARM1136JF-S level one memory system, including caches, TCM, DMA, SmartCache, TLBs, and write buffer.

Chapter 8 *Level Two Interface*

Read this chapter for a description of the ARM1136JF-S level two memory interface and the peripheral port.

Chapter 9 *Clocking and Resets*

Read this chapter for a description of the ARM1136JF-S clocking modes and the reset signals.

Chapter 10 *Power Control*

Read this chapter for a description of the ARM1136JF-S power control facilities.

Chapter 11 *Coprocessor Interface*

Read this chapter for details of the ARM1136JF-S coprocessor interface.

Chapter 12 *Vectored Interrupt Controller Port*

Read this chapter for a description of the ARM1136JF-S Vectored Interrupt Controller interface.

Chapter 13 *Debug*

Read this chapter for a description of the ARM1136JF-S debug support.

Chapter 14 *Debug Test Access Port*

Read this chapter for a description of the JTAG-based ARM1136JF-S Debug Test Access Port.

Chapter 15 *Trace Interface Port*

Read this chapter for a description of the trace interface port.

Chapter 16 *Cycle Timings and Interlock Behavior*

Read this chapter for a description of the ARM1136JF-S instruction cycle timing and for details of the interlocks.

Chapter 17 *AC Characteristics*

Read this chapter for a description of the timing parameters applicable to the ARM1136JF-S processor.

Appendix A *Signal Descriptions*

Read this appendix for a description of the ARM1136JF-S signals.

Product revision status

The *rn* identifier indicates the revision status of the product described in this document, where:

rn Identifies the major revision of the product.

pn Identifies the minor revision or modification status of the product.

Typographical conventions

The following typographical conventions are used in this document:

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

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

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

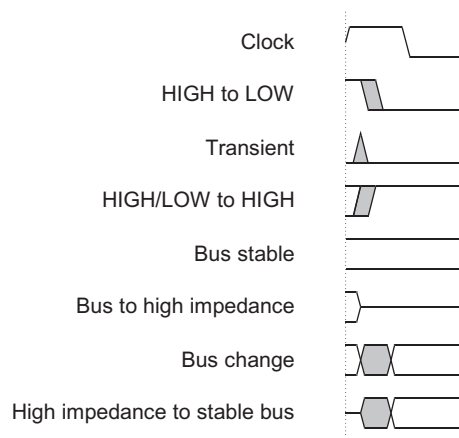
monospace Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

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

monospace bold Denotes language keywords when used outside example code.

Timing diagram conventions

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



Key to timing diagram conventions

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

Further reading

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

ARM periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

This document contains information that is specific to the ARM1136JF-S processor. Refer to the following documents for other relevant information:

- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *ARM1136 Implementation Guide* (ARM DII 0022)
- *AMBA[®] Specification* (ARM IHI 0011)
- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *Jazelle V6 Architecture Reference Manual* (ARM DDI 0225)
- *VFP11[™] Vector Floating-point Coprocessor Technical Reference Manual* (ARM DDI 0274)
- *RealView Compilation Tools Developer Guide* (ARM DUI 0203)

- *ARM PrimeCell® Vectored Interrupt Controller (PL192) Technical Reference Manual* (ARM DDI 0273).

Other publications

This section lists relevant documents published by third parties:

- *IEEE Standard Test Access Port and Boundary-Scan Architecture* specification 1149.1-1990(JTAG).

Figure 14-1 on page 14-2 is printed with permission IEEE Std. 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture Copyright 2001, by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Feedback

ARM Limited welcomes feedback both on the ARM1136JF-S processor, and on the documentation.

Feedback on the ARM1136JF-S processor

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

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

Feedback on this document

If you have any comments on about this document, 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.

Chapter 1

Introduction

This chapter introduces the ARM1136JF-S and ARM1136J-S processors and their features. It contains the following sections:

- *About the ARM1136J-S and ARM1136JF-S processors* on page 1-2
- *Components of the processor* on page 1-3
- *Power management* on page 1-23
- *Configurable options* on page 1-25
- *Pipeline stages* on page 1-26
- *Typical pipeline operations* on page 1-28
- *ARM1136JF-S architecture with Jazelle technology* on page 1-34
- *ARM1136JF-S instruction set summary* on page 1-36
- *Silicon revision information* on page 1-55.

1.1 About the ARM1136J-S and ARM1136JF-S processors

The ARM1136J-S and ARM1136JF-S processors incorporate an integer unit that implements the ARM architecture v6. They support the ARM and Thumb instruction sets, Jazelle technology to enable direct execution of Java bytecodes, and a range of SIMD DSP instructions that operate on 16-bit or 8-bit data values in 32-bit registers.

The ARM1136J-S and ARM1136JF-S processors are high-performance, low-power, ARM cached processor macrocells that provide full virtual memory capabilities.

The ARM1136J-S and ARM1136JF-S processors feature:

- an integer unit with integral EmbeddedICE-RT logic
- an eight-stage pipeline
- branch prediction with return stack
- low interrupt latency
- external coprocessor interface and coprocessors 14 and 15
- Instruction and Data *Memory Management Units* (MMUs), managed using MicroTLB structures backed by a unified main TLB
- Instruction and Data Caches, including a non-blocking Data Cache with *Hit-Under-Miss* (HUM)
- the caches are virtually indexed and physically addressed
- 64-bit interface to both caches
- a bypassable write buffer
- level one *Tightly-Coupled Memory* (TCM) that can be used as a local RAM with DMA, or as SmartCache
- high-speed *Advanced Microprocessor Bus Architecture* (AMBA) level two interfaces supporting prioritized multiprocessor implementations
- *Vector Floating-Point* (VFP) coprocessor support
- external coprocessor support
- trace support
- JTAG-based debug.

Note

- The only difference between the ARM1136JF-S and ARM1136J-S processor is that the ARM1136JF-S processor includes a *Vector Floating-Point* (VFP) coprocessor.
-

1.2 Components of the processor

The main blocks of the ARM1136J-S and ARM1136JF-S processors are:

- *Core* on page 1-5
- *Load Store Unit (LSU)* on page 1-9
- *Prefetch unit* on page 1-9
- *Memory system* on page 1-9
- *Level one memory system* on page 1-13
- *AMBA interface* on page 1-13
- *Coprocessor interface* on page 1-15
- *Debug* on page 1-16
- *Instruction cycle summary and interlocks* on page 1-18
- *Vector Floating-Point (VFP)* on page 1-18
- *System control* on page 1-20
- *Interrupt handling* on page 1-20.

Figure 1-1 on page 1-4 shows the structure of the ARM1136J-S and ARM1136JF-S processors.

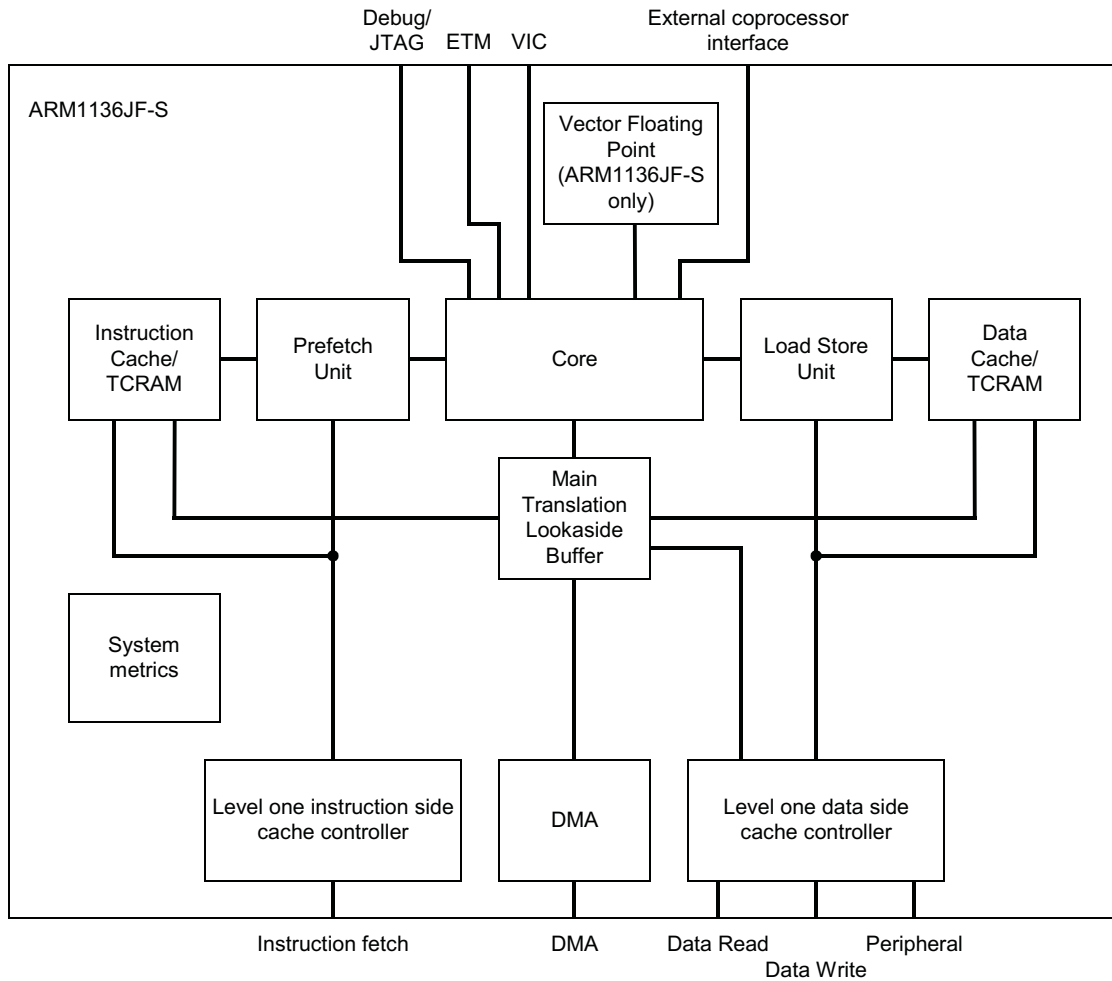


Figure 1-1 ARM1136JF-S processor block diagram

1.2.1 Core

The ARM1136J-S and ARM1136JF-S processors are built around the ARM11 core in an ARMv6 implementation that runs the 32-bit ARM, 16-bit Thumb, and 8-bit Jazelle instruction sets. The processor contains EmbeddedICE-RT logic and a JTAG debug interface to enable hardware debuggers to communicate with the processor. The core is described in more detail in the following sections:

- *Instruction sets*
- *Conditional execution*
- *Registers*
- *Modes and exceptions* on page 1-6
- *Thumb instruction set* on page 1-6
- *DSP instructions* on page 1-6
- *Media extensions* on page 1-6
- *Datapath* on page 1-7
- *Branch prediction* on page 1-8
- *Return stack* on page 1-8.

Instruction sets

The instruction sets are divided into four categories:

- data processing instructions
- load and store instructions
- branch instructions
- coprocessor instructions.

Note

Only load, store, and swap instructions can access data from memory.

Conditional execution

All ARM instructions are conditionally executed and can optionally update the four condition code flags, Negative, Zero, Carry, and Overflow, according to their result.

Registers

The ARM1136JF-S core contains:

- 31 general-purpose 32-bit registers
- seven dedicated 32-bit registers.

Note

At any one time, 16 registers are visible. The remainder are banked registers used to speed up exception processing.

Modes and exceptions

The core provides a set of operating and exception modes, to support systems combining complex operating systems, user applications, and real-time demands. There are seven operating modes, five of which are exception processing modes:

- user mode
- supervisor mode
- fast interrupt
- normal interrupt
- memory aborts
- software interrupts
- undefined instruction.

Thumb instruction set

Thumb is an extension to the ARM architecture. It contains a subset of the most commonly-used 32-bit ARM instructions that has been encoded into 16-bit wide opcodes, to reduce memory requirements.

DSP instructions

The ARM DSP instruction set extensions provide the following:

- 16-bit data operations
- saturating arithmetic
- MAC operations.

Multiply instructions are processed using a single-cycle 32x16 implementation. There are 32x32, 32x16, and 16x16 multiply instructions (MAC).

Media extensions

The ARMv6 instruction set provides media instructions to complement the DSP instructions. The media instructions are divided into the following main groups:

- Additional multiplication instructions for handling 16-bit and 32-bit data, including dual-multiplication instructions that operate on both 16-bit halves of their source registers.

This group includes an instruction that improves the performance and size of code for multi-word unsigned multiplications.

- Instructions to perform *Single Instruction Multiple Data* (SIMD) operations on pairs of 16-bit values held in a single register, or on quadruplets of 8-bit values held in a single register. The main operations supplied are addition and subtraction, selection, pack, and saturation.
- Instructions to extract bytes and halfwords from registers and zero-extend or sign-extend them. These include a parallel extraction of two bytes followed by extension of each byte to a halfword.
- Instructions to perform the unsigned *Sum-of-Absolute-Differences* (SAD) operation. This is used in MPEG motion estimation.

Datapath

The datapath consists of three pipelines:

- ALU/shift pipe
- MAC pipe
- load-store pipe, see *Load Store Unit (LSU)* on page 1-9.

ALU/shift pipe

The ALU-shift pipeline executes most of the ALU operations, and includes a 32-bit barrel shifter. It consists of three pipeline stages:

- Shift** The Shift stage contains the full barrel shifter. All shifts, including those required by the LSU, are performed in this stage.
- The saturating left shift, which doubles the value of an operand and saturates it, is implemented in the Shift stage.
- ALU** The ALU stage performs all arithmetic and logic operations, and generates the condition codes for instructions that set these operations.
- The ALU stage consists of a logic unit, an arithmetic unit, and a flag generator. Evaluation of the flags is performed in parallel with the main adder in the ALU. The flag generator is enabled only on flag-setting operations.
- To support the DSP instructions, the carry chains of the main adder are divided to enable 8 and 16-bit SIMD instructions.
- Sat** The Sat stage implements the saturation logic required by the various classes of DSP instructions.

MAC pipe

The MAC pipeline executes all of the enhanced multiply, and multiply-accumulate instructions.

The MAC unit consists of a 32x16 multiplier plus an accumulate unit, which is configured to calculate the sum of two 16x16 multiplies. The accumulate unit has its own dedicated single register read port for the accumulate operand.

To minimize power consumption, each of the MAC and ALU stages is only clocked when required.

Branch prediction

The core uses both static and dynamic branch prediction. All branches are predicted where the target address is an immediate address, or fixed-offset PC-relative address.

The first level of branch prediction is dynamic, through a 128-entry *Branch Target Address Cache* (BTAC). If the PC of a branch matches an entry in the BTAC, the branch history and the target address are used to fetch the new instruction stream.

Dynamically predicted branches might be removed from the instruction stream, and might execute in zero cycles.

If the address mappings are changed, the BTAC must be flushed. A BTAC flush instruction is provided in the CP15 coprocessor.

Static branch prediction is used to handle branches not matched in the BTAC. The static predictor makes a prediction based on the direction of the branches.

Return stack

A three-entry return stack is included to accelerate returns from procedure calls. For each procedure call, the return address is pushed onto a hardware stack. When a procedure return is recognized, the address held in the return stack is popped, and is used by the prefetch unit as the predicted return address.

———— Note —————

See *Pipeline stages* on page 1-26 for details of the pipeline stages and instruction progression.

See Chapter 3 *Control Coprocessor CP15* for system coprocessor programming information.

1.2.2 Load Store Unit (LSU)

The *Load Store Unit* (LSU) manages all load and store operations. The load-store pipeline decouples loads and stores from the MAC and ALU pipelines.

When LDM and STM instructions are issued to the LSU pipeline, other instructions run concurrently, subject to the requirements of supporting precise exceptions.

1.2.3 Prefetch unit

The prefetch unit fetches instructions from the Instruction Cache, Instruction TCM, or from external memory and predicts the outcome of branches in the instruction stream. Refer to Chapter 5 *Program Flow Prediction* for more details.

1.2.4 Memory system

The core provides a level-one memory system with the following features:

- separate instruction and data caches
- separate instruction and data RAMs
- 64-bit datapaths throughout the memory system
- virtually indexed, physically tagged caches
- complete memory management
- support for four sizes of memory page
- two-channel DMA into TCMs
- separate I-fetch, D-read, D-write interfaces, compatible with multi-layer AHB-Lite
- 32-bit dedicated peripheral interface
- export of memory attributes for second-level memory system.

The memory system is described in more detail in the following sections:

- *Instruction and data caches* on page 1-10
- *Cache power management* on page 1-10
- *Instruction and data TCM* on page 1-10
- *TCM DMA engine* on page 1-11
- *DMA features* on page 1-11
- *Memory Management Unit* on page 1-11.

Instruction and data caches

The core provides separate instruction and data caches. The cache has the following features:

- The instruction and data cache can be independently configured during synthesis to sizes between 4KB and 64KB.
- Both caches are 4-way set-associative. Each way can be locked independently.
- Cache replacement policies are pseudo-random or round-robin.
- The cache line length is eight words.
- Cache lines can be either Write-Back or Write-Through, determined by the MicroTLB entry.
- Each cache can be disabled independently, using the system control coprocessor.
- Data cache misses are non-blocking with up to three outstanding data cache misses being supported.
- Support is provided for streaming of sequential data from LDM and LDRD operations, and for sequential instruction fetches.
- On a cache-miss, critical word first filling of the cache is performed.
- For optimum area and performance, all of the cache RAMs, and the associated tag and valid RAMs, are designed to be implemented using standard ASIC RAM compilers.

Cache power management

To reduce power consumption, the number of full cache reads is reduced by taking advantage of the sequential nature of many cache operations. If a cache read is sequential to the previous cache read, and the read is within the same cache line, only the data RAM set that was previously read is accessed. In addition, the tag RAM is not accessed during this sequential operation.

To further reduce unnecessary power consumption, only the addressed words within a cache line are read at any time.

Instruction and data TCM

Because some applications might not respond well to caching, configurable memory blocks are provided for Instruction and Data *Tightly Coupled Memories* (TCMs). These ensure high-speed access to code or data.

An Instruction TCM is typically used to hold interrupt or exception code that must be accessed at high speed, without any potential delay resulting from a cache miss.

A Data TCM is typically used to hold a block of data for intensive processing, such as audio or video processing.

TCM DMA engine

To support use of the TCMs by data-intensive applications, the core provides two DMA channels to transfer data to or from the Instruction or Data TCM blocks. DMA can proceed in parallel with CPU accesses to the TCM blocks. Arbitration is on a cycle-by-cycle basis. The DMA channels connect with the *System-on-Chip* (SoC) backplane through a dedicated 64-bit AMBA AHB-Lite port.

The DMA controller is programmed using the CP15 system-control coprocessor. DMA accesses can only be to or from the TCM, and an external memory. No coherency support with the caches is provided.

Note

Only one of the two DMA channels can be active at any time.

DMA features

The DMA has the following features:

- runs in background of CPU operations
- CPU has priority access to TCM during DMA
- DMA programmed with virtual addresses
- DMA to either the instruction or data TCM
- allocated by a privileged process (OS)
- DMA progress accessible from software
- interrupt on DMA event.

Memory Management Unit

The *Memory Management Unit* (MMU) has a single *Translation Lookaside Buffer* (TLB) for both instructions and data. The MMU includes a 4KB page mapping size to enable a smaller RAM and ROM footprint for embedded systems and operating systems such as WindowsCE that have many small mapped objects. The ARM1136J-S and ARM1136JF-S processors implement the *Fast Context Switch Extension* (FCSE) and high vectors extension that are required to run Microsoft WindowsCE. See Chapter 6 *Memory Management Unit* for more details.

The MMU is responsible for protection checking, address translation, and memory attributes, some of which can be passed to an external level two memory system. The memory translations are cached in MicroTLBs for each of the instruction and data caches, with a single main TLB backing the MicroTLBs.

The MMU has the following features:

- matching of virtual address and ASID
- checking of domain access permissions
- checking of memory attributes
- virtual-to-physical address translation
- support for four page (region) sizes
- mapping of accesses to cache, TCM, peripheral port, or external memory
- TLB loading for hardware and software.

Paging

Four page sizes are supported:

- 16MB super sections
- 1MB sections
- 64KB large pages
- 4KB small pages.

Domains

Sixteen access domains are supported.

TLB

A two-level TLB structure is implemented. Entries in the main eight-way TLB are lockable. Hardware TLB loading is supported, and is backwards compatible with previous versions of the ARM architecture.

ASIDs

TLB entries can be global, or can be associated with particular processes or applications using *Application Space Identifiers* (ASIDs). ASIDs enable TLB entries to remain resident during context switches, avoiding the requirement of reloading them subsequently, and also enable task-aware debugging.

System control coprocessor

Cache, TCM, and DMA operations are controlled through a dedicated coprocessor, CP15, integrated within the core. This coprocessor provides a standard mechanism for configuring the level one memory system, and also provides functions such as memory barrier instructions. See *System control* on page 1-20 for more information.

1.2.5 Level one memory system

You can individually configure the *Instruction TCM* (ITCM) and *Data TCM* (DTCM) sizes with sizes of 0KB, 4KB, 8KB, 16KB, 32KB, or 64KB anywhere in the memory map. For flexibility in optimizing the TCM subsystem for performance, power, and RAM type, the TCMs are external to the processor. The **INITRAM** pin enables booting from the ITCM. Both the ITCM and DTCM support wait states and DMA activity. See Chapter 7 *Level One Memory System* for more details.

1.2.6 AMBA interface

The bus interface provides high bandwidth between the processor, second level caches, on-chip RAM, peripherals, and interfaces to external memory.

Separate bus interfaces are provided for:

- instruction fetch, 64-bit data
- data read, 64-bit data
- data write, 64-bit data
- peripheral access, 32-bit data
- DMA, 64-bit data.

All buses are multi-layer AHB-Lite compatible, enabling them to be merged in smaller systems. Additional signals are provided on each port to support:

- shared-memory synchronization primitives
- second-level cache
- bus transactions.

The ports support the following bus transactions:

Instruction fetch

Servicing instruction cache misses and uncachable instruction fetches.

Data read Servicing data cache misses, hardware handled TLB misses, and uncachable data reads.

Data write Servicing cache Write-Backs (including cache cleans), write-through, and uncachable data.

DMA Servicing the DMA engine for writing and reading the TCMs. This behaves as a single bidirectional port.

These ports enable several simultaneous outstanding transactions, providing high performance from second-level memory systems that support parallelism, and for high use of pipelined and multi-page memories such as SDRAM.

The AMBA interface is described in more detail in the following sections:

- *Bus clock speeds*
- *Unaligned accesses*
- *Mixed-endian support*
- *Write buffer*
- *Peripheral port* on page 1-15.

Bus clock speeds

The bus interface ports can operate either synchronously or asynchronously to the CPU clock, enabling the choice of CPU and bus clock frequencies.

Unaligned accesses

The core supports unaligned data access. Words and halfwords can be aligned to any byte boundary, enabling access to compacted data structures with no software overhead. This is useful for multi-processor applications, legacy code support, and reducing memory space requirements.

The BIU automatically generates multiple bus cycles for unaligned accesses.

Mixed-endian support

The core provides the option of switching between big and little-endian data access modes. This supports the sharing of data with big-endian systems, and improves handling of certain types of data.

Write buffer

All memory writes take place through the write buffer. The write buffer decouples the CPU pipeline from the system bus for external memory writes. Memory reads are checked for dependency against the write buffer contents.

Peripheral port

The peripheral port is a 32-bit AHB-Lite interface that provides direct access to local, non-shared peripherals without using bandwidth on the main AHB bus system. Accesses to regions of memory that are marked as device and non-shared are routed to the peripheral port instead of to the data read or data write ports.

See Chapter 8 *Level Two Interface* for more details.

1.2.7 Coprocessor interface

The ARM1136J-S and ARM1136JF-S processors support the connection of external coprocessors through the coprocessor interface. This interface supports all ARM coprocessor instructions:

- LDC
- LDCL
- STC
- STCL
- MRC
- MRRC
- MCR
- MCRR
- CDP.

Data for all loads to coprocessors is returned by the memory system in the order of the accesses in the program. HUMB operation of the cache is suppressed for coprocessor instructions.

The external coprocessor interface assumes that all coprocessor instructions are executed in order.

Externally-connected coprocessors follow the early stages of the core pipeline to enable instructions and data to be passed between the two pipelines. The coprocessor runs one pipeline stage behind the core pipeline.

To prevent the coprocessor interface introducing critical paths, wait states can be inserted in external coprocessor operations. These wait states enable critical signals to be retimed.

The VFP unit connects to the internal coprocessor interface, which has different timings and behavior, using controlled internal interconnection delays.

Chapter 11 *Coprocessor Interface* describes the interface for on-chip coprocessors such as floating-point or other application-specific hardware acceleration units.

1.2.8 Debug

The debug coprocessor, CP14, implements a full range of debug features described in Chapter 13 *Debug* and Chapter 14 *Debug Test Access Port*.

The core provides extensive support for real-time debug and performance profiling.

Debug is described in more detail in the following sections:

- *System performance monitoring*
- *ETM interface*
- *ETM trace buffer*
- *Software access to trace buffer* on page 1-17
- *Real-time debug facilities* on page 1-17
- *Debug and trace Environment* on page 1-18
- *ETM interface logic* on page 1-18.

System performance monitoring

This is a group of counters that can be configured to gather statistics on the operation of the processor and memory system. See *System performance monitoring* on page 3-87 for more details.

ETM interface

The core supports the connection of an external *Embedded Trace Macrocell* (ETM) unit to provide real-time code tracing of the core in an embedded system.

Various processor signals are collected and driven out from the core as the ETM interface. The interface is unidirectional and runs at the full speed of the core. The ETM interface is designed for direct connection to the external ETM unit without any additional glue logic, and can be disabled for power saving. See Chapter 15 *Trace Interface Port* for more details.

ETM trace buffer

You can extend the functionality of the ETM by adding an on-chip trace buffer. The trace buffer is an on-chip memory area where trace information is stored during capture instead of being exported immediately through the trace port at the operating frequency of the core.

This information can then be read out at a reduced clock rate from the trace buffer when capture is complete. This is done through the JTAG port of the SoC instead of through a dedicated trace port.

This two-step process avoids the requirement for a wide trace port that uses many high-speed device pins to implement. In effect, a zero-pin trace port is created where the device already has a JTAG port and associated pins.

Software access to trace buffer

The buffered trace information can be accessed through an AHB slave-based memory-mapped peripheral included as part of the trace buffer. This information can be used to carry out internal diagnostics on a closed system where a JTAG port is not normally brought out.

Real-time debug facilities

The ARM1136J-S and ARM1136JF-S processors contain an EmbeddedICE-RT logic unit to provide real-time debug facilities. It has the following capabilities:

- up to six breakpoints
- thread-aware breakpoints
- up to two watchpoints
- *Debug Communications Channel (DCC)*.

The EmbeddedICE-RT logic is connected directly to the core and monitors the internal address and data buses. You can access the EmbeddedICE-RT logic in one of two ways:

- executing CP14 instructions
- through a JTAG-style interface and associated TAP controller.

The EmbeddedICE-RT logic supports two modes of debug operation:

Halt mode On a debug event, such as a breakpoint or watchpoint, the core is stopped and forced into debug state. This enables the internal state of the core, and the external state of the system, to be examined independently from other system activity. When the debugging process has been completed, the core and system state is restored, and normal program execution resumed.

Monitor mode

On a debug event, a debug exception is generated instead of entering debug state, as in halt mode. A debug monitor program is activated by the exception entry and it is then possible to debug the processor while enabling the execution of critical interrupt service routines. The debug monitor program communicates with the debug host over the DCC.

Debug and trace Environment

Several external hardware and software tools are available to enable real-time debugging using the EmbeddedICE-RT logic and execution trace using the ET.

ETM interface logic

You can connect an optional external ETM to the core to provide real-time tracing of instructions and data in an embedded system. The core includes the logic and interface to enable you to trace program execution and data transfers using ETM11RV. Further details are in the *Embedded Trace Macrocell Specification*. See Appendix A *Signal Descriptions* for details of ETM-related signals.

1.2.9 Instruction cycle summary and interlocks

Chapter 16 *Cycle Timings and Interlock Behavior* describes instruction cycles and gives examples of interlock timing.

1.2.10 Vector Floating-Point (VFP)

The ARM1136J-S processor does not include a *Vector Floating-Point (VFP)* coprocessor.

The VFP coprocessor within the ARM1136JF-S processor supports floating point arithmetic. The VFP is implemented as a dedicated functional block, and is mapped as coprocessor numbers 10 and 11. Using the coprocessor access register, software can determine whether the VFP is present.

The VFP implements the ARM VFPv2 floating point coprocessor instruction set. It supports single and double-precision arithmetic on vector-vector, vector-scalar, and scalar-scalar data sets. Vectors can consist of up to eight single-precision, or four double-precision elements.

The VFP has its own bank of 32 registers for single-precision operands, which can be used in pairs for double-precision operands. Loads and stores of VFP registers can operate in parallel with arithmetic operations.

The VFP supports a wide range of single and double precision operations, including ABS, NEG, COPY, MUL, MAC, DIV, and SQRT. Most of these are effectively executed in a single cycle. Table 1-1 lists the exceptions. These issue latencies also apply to individual elements in a vector operation.

Table 1-1 Double-precision VFP operations

Instruction types	Issue latency
DP MUL and MAC	2 cycle
SP DIV, SQRT	14 cycles
DP DIV, SQRT	28 cycles
All other instructions	1 cycle

See *VFP11 Vector Floating-point Coprocessor Technical Reference Manual* for more details.

IEEE754 compliance

The VFP supports all five floating point exceptions defined by IEEE754:

- invalid operation
- divide by zero
- overflow
- underflow
- inexact.

Trapping of these exceptions can be individually enabled or disabled. If disabled, the IEEE754-defined default results are returned. All rounding modes are supported, and basic single and basic double formats are used.

For full compliance, support code is required to handle arithmetic where operands or results are de-norms. This support code is normally installed on the Undefined instruction exception handler.

Flush-to-zero mode

A flush-to-zero mode is provided where a default treatment of de-norms is applied. Table 1-2 shows the default behavior in flush-to-zero mode.

Table 1-2 Flush-to-zero mode

Operation	Flush-to-zero
De-norm operand(s)	Treated as 0+ Inexact flag set
De-norm result	Returned as 0+ Inexact Flag set

Operations not supported

The following operations are not directly supported by the VFP:

- remainder
- binary (decimal) conversions
- direct comparisons between single and double-precision values.

These are normally implemented as C library functions.

1.2.11 System control

The control of the memory system and its associated functionality, and other system-wide control attributes are managed through a dedicated system control coprocessor, CP15. See *Overall system configuration and control* on page 3-93 for more details.

1.2.12 Interrupt handling

Interrupt handling in the ARM1136J-S and ARM1136JF-S processors is compatible with previous ARM architectures, but has several additional features to improve interrupt performance for real-time applications.

Interrupt handling is described in more detail in the following sections:

- *VIC port* on page 1-21
- *Low interrupt latency configuration* on page 1-21
- *Configuration* on page 1-22
- *Exception processing enhancements* on page 1-22.

VIC port

The core has a dedicated port that enables an external interrupt controller, such as the *ARM Vectored Interrupt Controller (VIC)*, to supply a vector address along with an *interrupt request (IRQ)* signal. This provides faster interrupt entry but can be disabled for compatibility with earlier interrupt controllers.

Low interrupt latency configuration

This mode minimizes the worst-case interrupt latency of the processor, with a small reduction in peak performance, or instructions-per-cycle. You can tune the behavior of the core to suit the application requirements.

The low-latency configuration disables HUM operation of the cache. In low-latency mode, on receipt of an interrupt, the ARM113JF-S processor:

- abandons any pending restartable memory operations
- on return from the interrupt, the memory operations are then restarted.

In low interrupt latency configuration, software must only use multi-word load/store instructions that are fully restartable. They must not be used on memory locations that produce side-effects for the type of access concerned.

The instructions that this currently applies to are:

ARM LDC, all forms of LDM, LDRD, and STC, and all forms of STM and STRD.

Thumb LDMIA, STMIA, PUSH, and POP.

To achieve optimum interrupt latency, memory locations accessed with these instructions must not have large numbers of wait-states associated with them. To minimize the interrupt latency, the following is recommended:

- multiple accesses to areas of memory marked as Device or Strongly Ordered must not be performed
- areas of memory marked as Device or Strongly Ordered must not be performed to slow areas of memory, that is, those that take many cycles in generating a response
- SWP operations must not be performed to slow areas of memory.

Configuration

Configuration is through the system control coprocessor. To ensure that a change between normal and low interrupt latency configurations is synchronized correctly, you must use software systems that only change the configuration while interrupts are disabled.

Exception processing enhancements

The ARMv6 architecture contains several enhancements to exception processing, to reduce interrupt handler entry and exit time:

- SRS** Save return state to a specified stack frame.
- RFE** Return from exception.
- CPS** Directly modify the CPSR.

1.3 Power management

The ARM1136J-S and ARM1136JF-S processors include several microarchitectural features to reduce energy consumption:

- Accurate branch and return prediction, reducing the number of incorrect instruction fetch and decode operations.
- Use of physically tagged caches, which reduce the number of cache flushes and refills, to save energy in the system.
- The use of MicroTLBs reduces the power consumed in translation and protection look-ups for each memory access.
- The caches use sequential access information to reduce the number of accesses to the TagRAMs and to unmatched data RAMs.
- Extensive use of gated clocks and gates to disable inputs to unused functional blocks. Because of this, only the logic actively in use to perform a calculation consumes any dynamic power.

The ARM1136J-S and ARM1136JF-S processors support four levels of power management:

Run mode This mode is the normal mode of operation in which all of the functionality of the ARM1136JF-S processor is available.

Standby mode

This mode disables most of the clocks of the device, while keeping the device powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the standby state. The transition from the standby mode to the run mode is caused by one of the following:

- an interrupt, either masked or unmasked
- a debug request, regardless of whether debug is enabled
- reset.

Shutdown mode

This mode has the entire device powered down. All state, including cache and TCM state, must be saved externally. The part is returned to the run state by the assertion of reset. This state saving is performed with interrupts disabled, and finishes with a DrainWriteBuffer operation. The ARM1136JF-S processor then communicates with the power controller that it is ready to be powered down.

Dormant mode

This mode enables the ARM113JF-S processor to be powered down, while leaving the state of the caches and the TCM powered up and maintaining their state. Although software visibility of the valid bits is provided to enable implementation of dormant mode, the following are required for full implementation of dormant mode:

- modification of the RAMs to include an input clamp
- implementation of separate power domains.

Power management features are described in more detail in Chapter 10 *Power Control*.

1.4 Configurable options

The configurable features in ARM1136JF-S processors are shown in Table 1-3.

Table 1-3 Configurable options

Feature	Range of options
Cache way size	1KB, 2KB, 4KB, 8KB, or 16KB
TCM block size	0KB, 4KB, 8KB, 16KB, 32KB, or 64KB

The number of TCM blocks and the number of TCM blocks supporting SmartCache are restricted to a minimum to reduce the impact on performance.

In addition, the form of the BIST solution for the RAM blocks in the ARM1136JF-S design is determined when the processor is implemented. For details, see the *ARM1136 Implementation Guide*.

The default configuration of ARM1136J-S and ARM1136JF-S processors is shown in Table 1-4.

Table 1-4 ARM1136JF-S processor default configurations

Feature	Default value
Cache way size	4KB
TCM block size	16KB
Inclusion of VFP	There are two variants of ARM1136JF-S processors: <ul style="list-style-type: none"> • ARM1136JF-S includes a VFP • ARM1136J-S does not include a VFP

1.5 Pipeline stages

Figure 1-2 shows:

- the two Fetch stages
- a Decode stage
- an Issue stage
- the four stages of the ARM1136JF-S integer execution pipeline.

These eight stages make up the ARM1136JF-S pipeline.

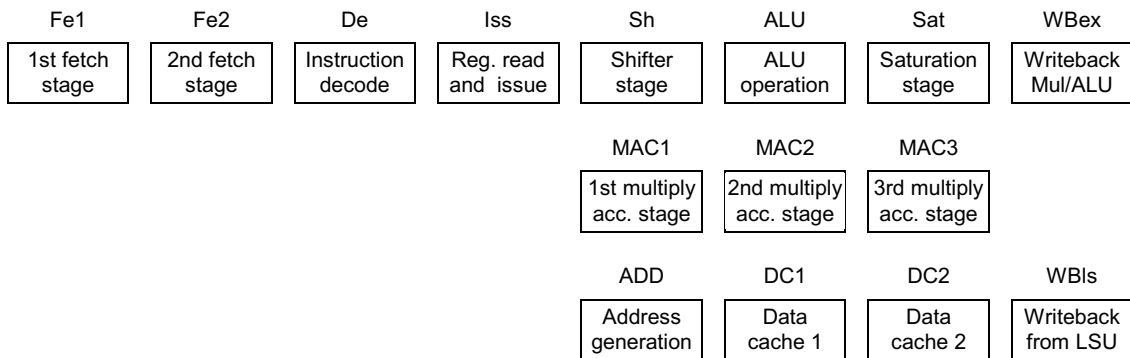


Figure 1-2 ARM1136JF-S pipeline stages

The pipeline stages are:

- Fe1** First stage of instruction fetch and branch prediction.
- Fe2** Second stage of instruction fetch and branch prediction.
- De** Instruction decode.
- Iss** Register read and instruction issue.
- Sh** Shifter stage.
- ALU** Main integer operation calculation.
- Sat** Pipeline stage to enable saturation of integer results.
- WBex** Write back of data from the multiply or main execution pipelines.
- MAC1** First stage of the multiply-accumulate pipeline.
- MAC2** Second stage of the multiply-accumulate pipeline.

MAC3	Third stage of the multiply-accumulate pipeline.
ADD	Address generation stage.
DC1	First stage of Data Cache access.
DC2	Second stage of Data Cache access.
WBls	Write back of data from the Load Store Unit.

By overlapping the various stages of operation, the ARM1136JF-S processor maximizes the clock rate achievable to execute each instruction. It delivers a throughput approaching one instruction for each cycle.

The Fetch stages can hold up to four instructions, where branch prediction is performed on instructions ahead of execution of earlier instructions.

The Issue and Decode stages can contain any instruction in parallel with a predicted branch.

The Execute, Memory, and Write stages can contain a predicted branch, an ALU or multiply instruction, a load/store multiple instruction, and a coprocessor instruction in parallel execution.

1.6 Typical pipeline operations

Figure 1-3 shows all the operations in each of the pipeline stages in the ALU pipeline, the load/store pipeline, and the HUM buffers.

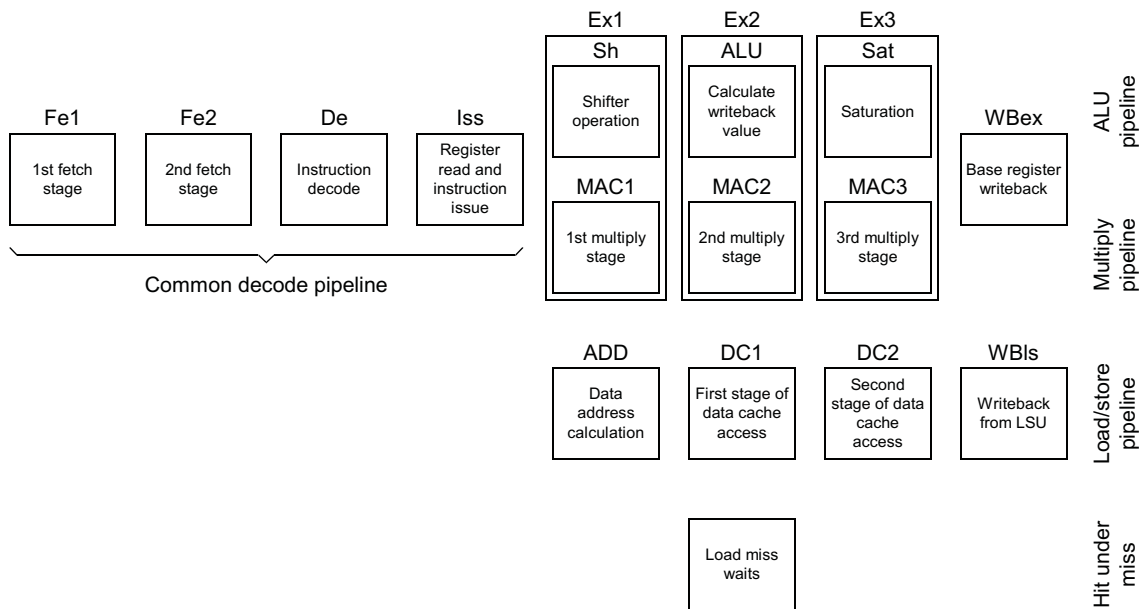


Figure 1-3 Typical operations in pipeline stages

Figure 1-4 shows a typical ALU data processing instruction. The load/store pipeline and the HUM buffer are not used.

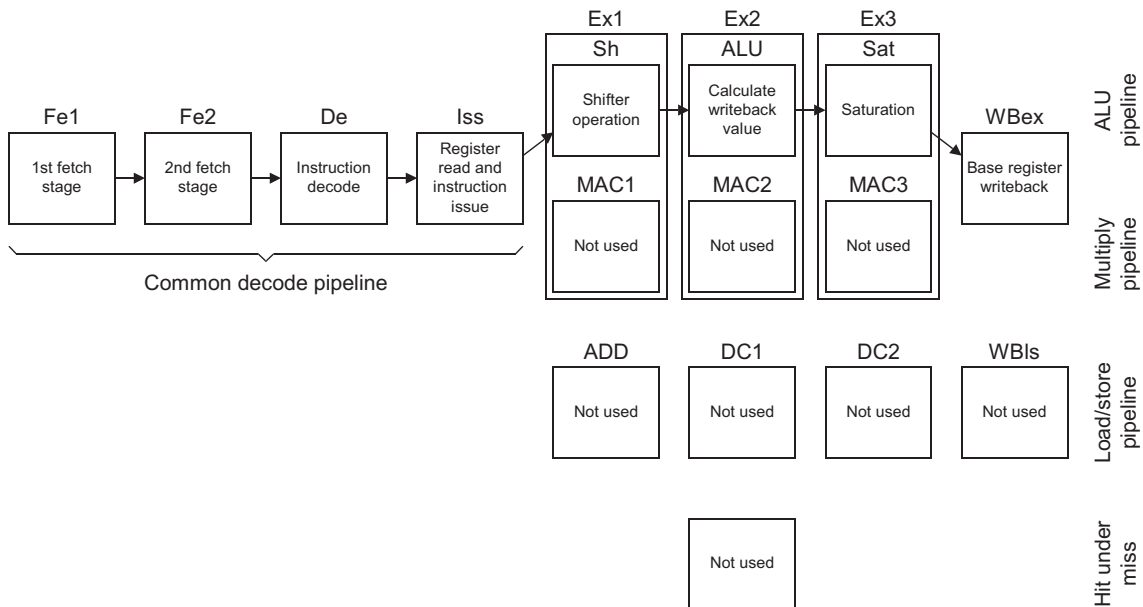


Figure 1-4 Typical ALU operation

Figure 1-5 shows a typical multiply operation. The MUL instruction can loop in the MAC1 stage until it has passed through the first part of the multiplier array enough times. Then it progresses to MAC2 and MAC3 where it passes once through the second half of the array to produce the final result.

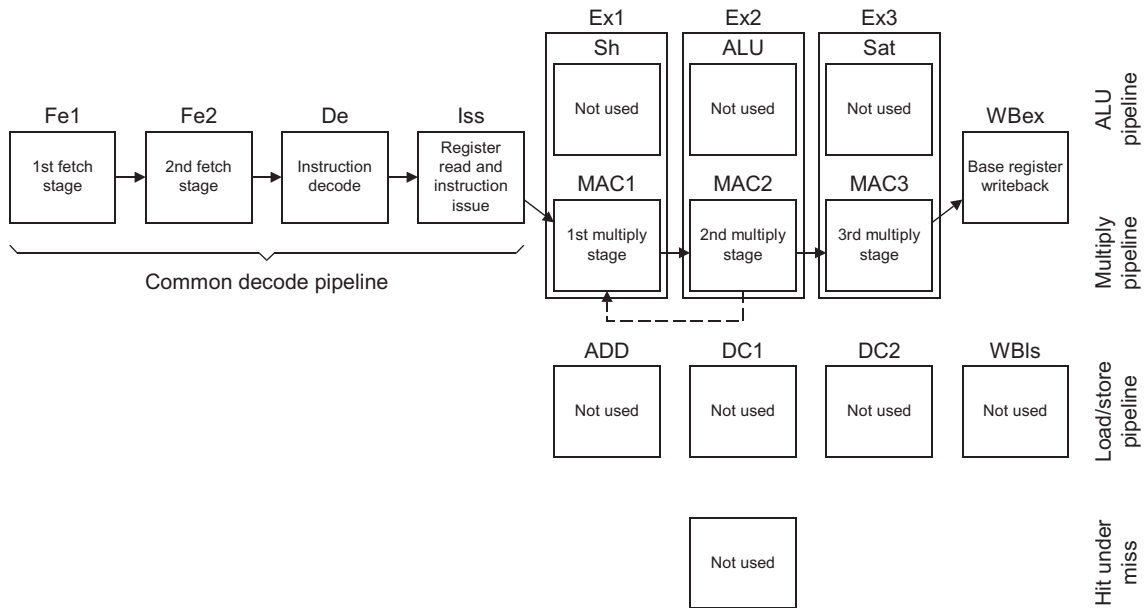


Figure 1-5 Typical multiply operation

1.6.1 Instruction progression

Figure 1-6 shows an LDR/STR operation that hits in the Data Cache.

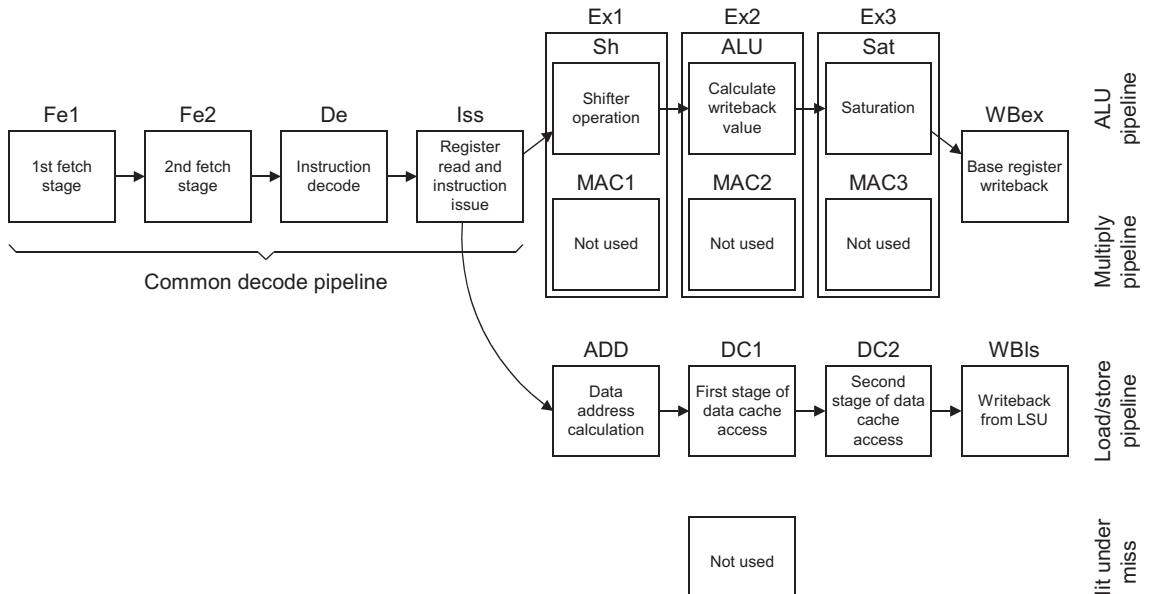


Figure 1-6 Progression of an LDR/STR operation

Figure 1-7 shows the progression of an LDM/STM operation using the load/store pipeline to complete. Other instructions can use the ALU pipeline at the same time as the LDM/STM completes in the load/store pipeline.

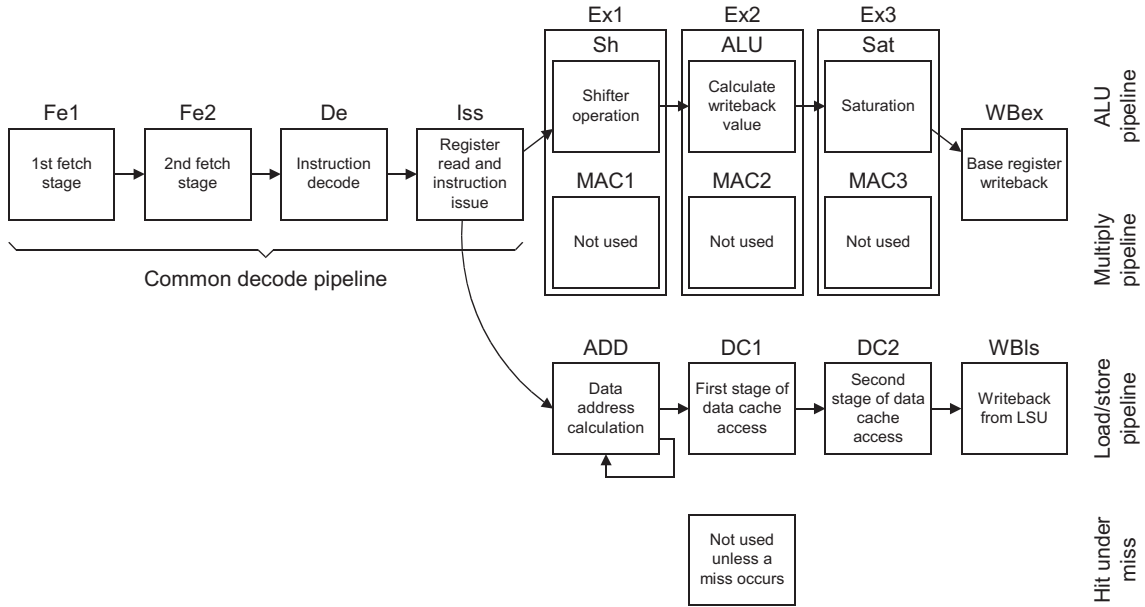


Figure 1-7 Progression of an LDM/STM operation

Figure 1-8 shows the progression of an LDR that misses. When the LDR is in the HUM buffers, other instructions, including independent loads that hit in the cache, can run under it.

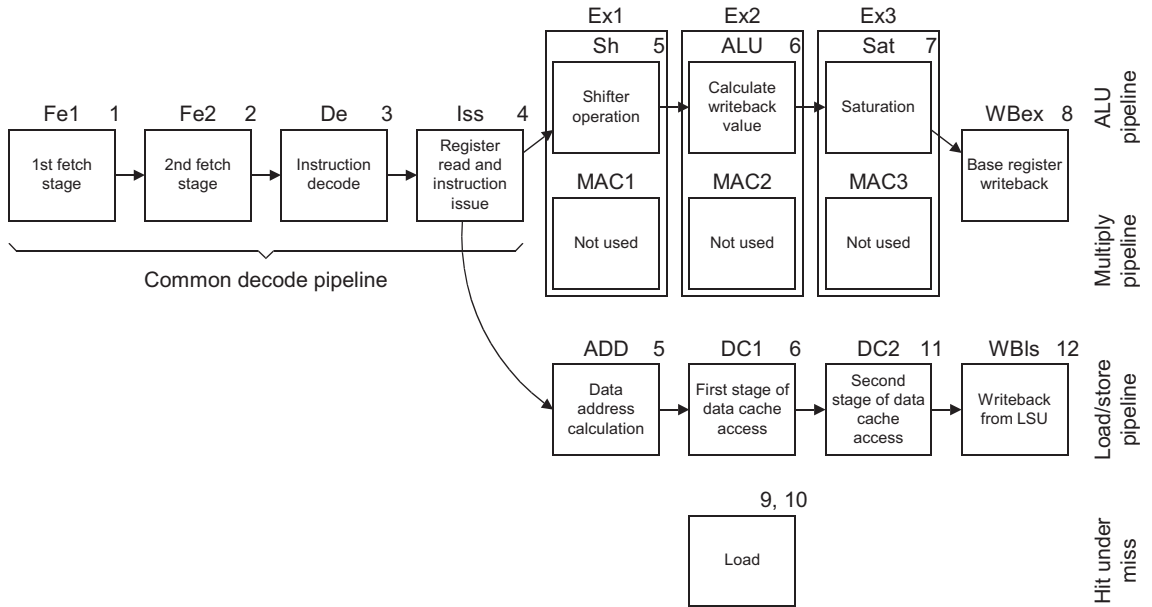


Figure 1-8 Progression of an LDR that misses

See Chapter 16 *Cycle Timings and Interlock Behavior* for details of instruction cycle timings.

1.7 ARM1136JF-S architecture with Jazelle technology

The ARM1136JF-S processor has three instruction sets:

- the 32-bit ARM instruction set used in ARM state, with media instructions
- the 16-bit Thumb instruction set used in Thumb state
- the 8-bit Java bytecode used in Java state.

For details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*. For full details of the ARM1136JF-S Java instruction set, see the *Jazelle V1 Architecture Reference Manual*.

1.7.1 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 ARM1136JF-S processor gives you the choice of running in ARM state, or Thumb state, or a mix of the two. This enables you to optimize both code density and performance to best suit your application requirements.

1.7.2 The Thumb instruction set

The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are 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, enabling 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.

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

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

1.7.3 Java bytecodes

ARM architecture v6 with Jazelle technology executes variable length Java bytecodes. Java bytecodes fall into two classes:

Hardware execution

Bytecodes that perform stack-based operations.

Software execution

Bytecodes that are too complex to execute directly in hardware are executed in software. An ARM register is used to access a table of exception handlers to handle these particular bytecodes.

A complete list of the ARM1136JF-S processor-supported Java bytecodes and their corresponding hardware or software instructions is in the *Jazelle V1 Architecture Reference Manual*.

1.8 ARM1136JF-S instruction set summary

This section provides:

- an *Extended ARM instruction set summary* on page 1-38
- a *Thumb instruction set summary* on page 1-51.

A key to the ARM and Thumb instruction set tables is given in Table 1-5.

The ARM1136JF-S processor is an implementation of the ARM architecture v6 with ARM Jazelle technology. For a description of the ARM and Thumb instruction sets refer to the *ARM Architecture Reference Manual*. Contact ARM Limited for complete descriptions of all instruction sets.

Table 1-5 Key to instruction set tables

Symbol	Description
{!}	Update base register after operation if ! present.
B	Byte operation.
H	Halfword operation.
T	Forces execution to be handled as having User mode privilege. Cannot be used with pre-indexed addresses.
x	Selects HIGH or LOW 16 bits of register Rm. T selects the HIGH 16 bits. (T = top) and 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) and B selects the LOW 16 bits (B = bottom).
{cond}	Updates condition flags if cond present. See Table 1-14 on page 1-50.
{field}	See Table 1-13 on page 1-50.
{S}	Sets condition codes (optional).
<a_mode2>	See Table 1-7 on page 1-46.
<a_mode2P>	See Table 1-8 on page 1-47.
<a_mode3>	See Table 1-9 on page 1-48.
<a_mode4>	See Table 1-10 on page 1-48.
<a_mode5>	See Table 1-11 on page 1-49.
<cp_num>	One of the coprocessors p0 to p15.

Table 1-5 Key to instruction set tables (continued)

Symbol	Description
<effect>	Specifies what effect is wanted on the interrupt disable bits, A, I, and F in the CPSR: IE = Interrupt enable ID = Interrupt disable. If <effect> is specified, the bits affected are specified in <iflags>.
<endian_specifier>	BE = Set E bit in instruction, set CPSR E bit. LE = Reset E bit in instruction, clear CPSR E bit.
<HighReg>	One of the registers r8 to r15.
<iflags>	A sequence of one or more of the following: a = Set A bit. i = Set I bit. f = Set F bit. If <effect> is specified, the sequence determines which interrupt flags are affected.
<immed_8*4>	A 10-bit constant, formed by left-shifting an 8-bit value by two bits.
<immed_8>	An 8-bit constant.
<immed_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.
<label>	The target address to branch to.
<LowReg>	One of the registers R0 to r7.
<mode>	The new mode number for a mode change. See <i>Mode bits</i> on page 2-21.
<op1>, <op2>	Specify, in a coprocessor-specific manner, which coprocessor operation to perform.
<operand2>	See Table 1-12 on page 1-49.
<option>	Specifies additional instruction options to the coprocessor. An integer in the range 0 to 255 surrounded by { and }.
<reglist>	A comma-separated list of registers, enclosed in braces {and}.
<rotation>	One of ROR #8, ROR #16, or ROR #24.
<shift>	0 = LSL #N for N= 0 to 31 1 = ASR #N for N = 1 to 32.

1.8.1 Extended ARM instruction set summary

The extended ARM instruction set summary is given in Table 1-6.

Table 1-6 ARM instruction set summary

Operation	Assembler
Arithmetic	
Add	ADD{cond}{S} <Rd>, <Rn>, <operand2>
Add with carry	ADC{cond}{S} <Rd>, <Rn>, <operand2>
Subtract	SUB{cond}{S} <Rd>, <Rn>, <operand2>
Subtract with carry	SBC{cond}{S} <Rd>, <Rn>, <operand2>
Reverse subtract	RSB{cond}{S} <Rd>, <Rn>, <operand2>
Reverse subtract with carry	RSC{cond}{S} <Rd>, <Rn>, <operand2>
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>
Saturating add	QADD{cond} <Rd>, <Rm>, <Rn>
Saturating add with double	QDADD{cond} <Rd>, <Rm>, <Rn>
Saturating subtract	QSUB{cond} <Rd>, <Rm>, <Rn>
Saturating subtract with double	QDSUB{cond} <Rd>, <Rm>, <Rn>
Multiply 16x16	SMULxy{cond} <Rd>, <Rm>, <Rs>
Multiply-accumulate 16x16+32	SMLAxy{cond} <Rd>, <Rm>, <Rs>, <Rn>
Multiply 32x16	SMULwxy{cond} <Rd>, <Rm>, <Rs>
Multiply-accumulate 32x16+32	SMLAwxy{cond} <Rd>, <Rm>, <Rs>, <Rn>
Multiply signed accumulate long 16x16+64	SMLALxy{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
Count leading zeros	CLZ{cond} <Rd>, <Rm>

Table 1-6 ARM instruction set summary (continued)

Operation		Assembler
Compare	Compare	CMP{cond} <Rn>, <operand2>
	Compare negative	CMN{cond} <Rn>, <operand2>
Logical	Move	MOV{cond}{S} <Rd>, <operand2>
	Move NOT	MVN{cond}{S} <Rd>, <operand2>
	Test	TST{cond} <Rn>, <operand2>
	Test equivalence	TEQ{cond} <Rn>, <operand2>
	AND	AND{cond}{S} <Rd>, <Rn>, <operand2>
	XOR	EOR{cond}{S} <Rd>, <Rn>, <operand2>
	OR	ORR{cond}{S} <Rd>, <Rn>, <operand2>
	Bit clear	BIC{cond}{S} <Rd>, <Rn>, <operand2>
Branch	Branch	B{cond} <label>
	Branch with link	BL{cond} <label>
	Branch and exchange	BX{cond} <Rm>
	Branch, link and exchange	BLX <label>
	Branch, link and exchange	BLX{cond} <Rm>
	Branch and exchange to Java state	BXJ{cond} <Rm>
Status register handling	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_{field}, #<immed_8r>
	Move immediate to CPSR flags	MSR{cond} CPSR_{field}, #<immed_8r>
Load	Word	LDR{cond} <Rd>, <a_mode2>
	Word with User mode privilege	LDR{cond}T <Rd>, <a_mode2P>

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler
PC as destination, branch and exchange	LDR{cond} R15, <a_mode2P>
Byte	LDR{cond}B <Rd>, <a_mode2>
Byte with User mode privilege	LDR{cond}BT <Rd>, <a_mode2P>
Byte signed	LDR{cond}SB <Rd>, <a_mode3>
Halfword	LDR{cond}H <Rd>, <a_mode3>
Halfword signed	LDR{cond}SH <Rd>, <a_mode3>
Doubleword	LDR{cond}D <Rd>, <a_mode3>
Return from exception	RFE<a_mode4> <Rn>{!}
Load multiple	
Stack operations	LDM{cond}<a_mode4L> <Rn>{!}, <reglist>
Increment before	LDM{cond}IB <Rn>{!}, <reglist>{^}
Increment after	LDM{cond}IA <Rn>{!}, <reglist>{^}
Decrement before	LDM{cond}DB <Rn>{!}, <reglist>{^}
Decrement after	LDM{cond}DA <Rn>{!}, <reglist>{^}
Stack operations and restore CPSR	LDM{cond}<a_mode4> <Rn>{!}, <reglist+pc>^
User registers	LDM{cond}<a_mode4> <Rn>{!}, <reglist>^
Soft preload	
Memory system hint	PLD <a_mode2>
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>
Doubleword	STR{cond}D <Rd>, <a_mode3>
Store return state	SRS<a_mode4> <mode>{!}
Store multiple	
Stack operations	STM{cond}<a_mode4S> <Rn>{!}, <reglist>

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler	
Increment before	STM{cond}IB <Rn>{!}, <reglist>{^}	
Increment after	STM{cond}IA <Rn>{!}, <reglist>{^}	
Decrement before	STM{cond}DB <Rn>{!}, <reglist>{^}	
Decrement after	STM{cond}DA <Rn>{!}, <reglist>{^}	
User registers	STM{cond}<a_mode4S> <Rn>{!}, <reglist>^	
Swap	Word	SWP{cond} <Rd>, <Rm>, [<Rn>]
	Byte	SWP{cond}B <Rd>, <Rm>, [<Rn>]
Change state	Change processor state	CPS<effect> <iflags>{, <mode>}
	Change processor mode	CPS <mode>
	Change endianness	SETEND <endian_specifier>
Byte-reverse	Byte-reverse word	REV{cond} <Rd>, <Rm>
	Byte-reverse halfword	REV16{cond} <Rd>, <Rm>
	Byte-reverse signed halfword	REVSH{cond} <Rd>, <Rm>
Synchronization primitives	Load exclusive	LDREX{cond} <Rd>, [<Rn>]
	Store exclusive	STREX{cond} <Rd>, <Rm>, [<Rn>]
Coprocessor	Data operations	CDP{cond} <cp_num>, <op1>, <CRd>, <CRn>, <CRm>{, <op2>}
	Move to ARM reg from coproc	MRC{cond} <cp_num>, <op1>, <Rd>, <CRn>, <CRm>{, <op2>}
	Move to coproc from ARM reg	MCR{cond} <cp_num>, <op1>, <Rd>, <CRn>, <CRm>{, <op2>}
	Move double to ARM reg from coproc	MRRC{cond} <cp_num>, <op1>, <Rd>, <Rn>, <CRm>
	Move double to coproc from ARM reg	MCRR{cond} <cp_num>, <op1>, <Rd>, <Rn>, <CRm>
	Load	LDC{cond} <cp_num>, <CRd>, <a_mode5>
	Store	STC{cond} <cp_num>, <CRd>, <a_mode5>

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler	
Alternative coprocessor	Data operations	CDP2 <cp_num>, <op1>, <CRd>, <CRn>, <CRm>{, <op2>}
	Move to ARM reg from coproc	MRC2 <cp_num>, <op1>, <Rd>, <CRn>, <CRm>{, <op2>}
	Move to coproc from ARM reg	MCR2 <cp_num>, <op1>, <Rd>, <CRn>, <CRm>{, <op2>}
	Move double to ARM reg from coproc	MRRC2 <cp_num>, <op1>, <Rd>, <Rn>, <CRm>
	Move double to coproc from ARM reg	MCCR2 <cp_num>, <op1>, <Rd>, <Rn>, <CRm>
	Load	LDC2 <cp_num>, <CRd>, <a_mode5>
	Store	STC2 <cp_num>, <CRd>, <a_mode5>
Software interrupt	SWI{cond} <immed_24>	
Software breakpoint	BKPT <immed_16>	
Parallel add /subtract	Signed add high 16 + 16, low 16 + 16, set GE flags	SADD16{cond} <Rd>, <Rn>, <Rm>
	Saturated add high 16 + 16, low 16 + 16	QADD16{cond} <Rd>, <Rn>, <Rm>
	Signed high 16 + 16, low 16 + 16, halved	SHADD16{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 + 16, low 16 + 16, set GE flags	UADD16{cond} <Rd>, <Rn>, <Rm>
	Saturated unsigned high 16 + 16, low 16 + 16	UQADD16{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 + 16, low 16 + 16, halved	UHADD16{cond} <Rd>, <Rn>, <Rm>
	Signed high 16 + low 16, low 16 - high 16, set GE flags	SADDSUBX{cond} <Rd>, <Rn>, <Rm>
	Saturated high 16 + low 16, low 16 - high 16	QADDSUBX{cond} <Rd>, <Rn>, <Rm>
	Signed high 16 + low 16, low 16 - high 16, halved	SHADDSUBX{cond} <Rd>, <Rn>, <Rm>

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler
Unsigned high 16 + low 16, low 16 - high 16, set GE flags	UADDSUBX{cond} <Rd>, <Rn>, <Rm>
Saturated unsigned high 16 + low 16, low 16 - high 16	UQADDSUBX{cond} <Rd>, <Rn>, <Rm>
Unsigned high 16 + low 16, low 16 - high 16, halved	UHADDSUBX{cond} <Rd>, <Rn>, <Rm>
Signed high 16 - low 16, low 16 + high 16, set GE flags	SSUBADDX{cond} <Rd>, <Rn>, <Rm>
Saturated high 16 - low 16, low 16 + high 16	QSUBADDX{cond} <Rd>, <Rn>, <Rm>
Signed high 16 - low 16, low 16 + high 16, halved	SHSUBADDX{cond} <Rd>, <Rn>, <Rm>
Unsigned high 16 - low 16, low 16 + high 16, set GE flags	USUBADDX{cond} <Rd>, <Rn>, <Rm>
Saturated unsigned high 16 - low 16, low 16 + high 16	UQSUBADDX{cond} <Rd>, <Rn>, <Rm>
Unsigned high 16 - low 16, low 16 + high 16, halved	UHSUBADDX{cond} <Rd>, <Rn>, <Rm>
Signed high 16-16, low 16-16, set GE flags	SSUB16{cond} <Rd>, <Rn>, <Rm>
Saturated high 16 - 16, low 16 - 16	QSUB16{cond} <Rd>, <Rn>, <Rm>
Signed high 16 - 16, low 16 - 16, halved	SHSUB16{cond} <Rd>, <Rn>, <Rm>
Unsigned high 16 - 16, low 16 - 16, set GE flags	USUB16{cond} <Rd>, <Rn>, <Rm>
Saturated unsigned high 16 - 16, low 16 - 16	UQSUB16{cond} <Rd>, <Rn>, <Rm>
Unsigned high 16 - 16, low 16 - 16, halved	UHSUB16{cond} <Rd>, <Rn>, <Rm>
Four signed 8 + 8, set GE flags	SADD8{cond} <Rd>, <Rn>, <Rm>
Four saturated 8 + 8	QADD8{cond} <Rd>, <Rn>, <Rm>

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler
Four signed 8 + 8, halved	SHADD8{cond} <Rd>, <Rn>, <Rm>
Four unsigned 8 + 8, set GE flags	UADD8{cond} <Rd>, <Rn>, <Rm>
Four saturated unsigned 8 + 8	UQADD8{cond} <Rd>, <Rn>, <Rm>
Four unsigned 8 + 8, halved	UHADD8{cond} <Rd>, <Rn>, <Rm>
Four signed 8 - 8, set GE flags	SSUB8{cond} <Rd>, <Rn>, <Rm>
Four saturated 8 - 8	QSUB8{cond} <Rd>, <Rn>, <Rm>
Four signed 8 - 8, halved	SHSUB8{cond} <Rd>, <Rn>, <Rm>
Four unsigned 8 - 8	USUB8{cond} <Rd>, <Rn>, <Rm>
Four saturated unsigned 8 - 8	UQSUB8{cond} <Rd>, <Rn>, <Rm>
Four unsigned 8 - 8, halved	UHSUB8{cond} <Rd>, <Rn>, <Rm>
Sum of absolute differences	USAD8{cond} <Rd>, <Rm>, <Rs>
Sum of absolute differences and accumulate	USADA8{cond} <Rd>, <Rm>, <Rs>, <Rn>
Sign/zero extend and add	
Two low 8/16, sign extend to 16 + 16	SADD8T016{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Low 8/32, sign extend to 32, + 32	SADD8T032{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Low 16/32, sign extend to 32, + 32	SADD16T032{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Two low 8/16, zero extend to 16, + 16	UADD8T016{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Low 8/32, zero extend to 32, + 32	UADD8T032{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Low 16/32, zero extend to 32, + 32	UADD16T032{cond} <Rd>, <Rn>, <Rm>{, <rotation>}
Two low 8, sign extend to 16, packed 32	SUNPK8T016{cond} <Rd>, <Rm>{, <rotation>}
Low 8, sign extend to 32	SUNPK8T032{cond} <Rd>, <Rm>{, <rotation>}
Low 16, sign extend to 32	SUNPK16T032{cond} <Rd>, <Rm>{, <rotation>}
Two low 8, zero extend to 16, packed 32	UUNPK8T016{cond} <Rd>, <Rm>{, <rotation>}

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler	
Low 8, zero extend to 32	UUNPK8T032{cond} <Rd>, <Rm>{, <rotation>}	
Low 16, zero extend to 32	UUNPK16T032{cond} <Rd>, <Rm>{, <rotation>}	
Signed multiply and multiply, accumulate	Signed (high 16 x 16) + (low 16 x 16) + 32, and set Q flag.	SMLAD{cond} <Rd>, <Rm>, <Rs>, <Rn>
	As SMLAD, but high x low, low x high, and set Q flag	SMLADX{cond} <Rd>, <Rm>, <Rs>, <Rn>
	Signed (high 16 x 16) - (low 16 x 16) + 32	SMLSD{cond} <Rd>, <Rm>, <Rs>, <Rn>
	As SMLSD, but high x low, low x high	SMLSXD{cond} <Rd>, <Rm>, <Rs>, <Rn>
	Signed (high 16 x 16) + (low 16 x 16) + 64	SMLALD{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	As SMLALD, but high x low, low x high	SMLALDX{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	Signed (high 16 x 16) - (low 16 x 16) + 64	SMLSLD{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	As SMLSLD, but high x low, low x high	SMLSLDX{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	32 + truncated high 16 (32 x 32)	SMMLA{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 + rounded high 16 (32 x 32)	SMMLAR{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 - truncated high 16 (32 x 32)	SMMLS{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 -rounded high 16 (32 x 32)	SMMLSR{cond} <Rd>, <Rm>, <Rs>, <Rn>
	Signed (high 16 x 16) + (low 16 x 16), and set Q flag	SMUAD{cond} <Rd>, <Rm>, <Rs>
	As SMUAD, but high x low, low x high, and set Q flag	SMUADX{cond} <Rd>, <Rm>, <Rs>
	Signed (high 16 x 16) - (low 16 x 16)	SMUSD{cond} <Rd>, <Rm>, <Rs>
	As SMUSD, but high x low, low x high	SMUSDX{cond} <Rd>, <Rm>, <Rs>
Truncated high 16 (32 x 32)	SMMUL{cond} <Rd>, <Rm>, <Rs>	
Rounded high 16 (32 x 32)	SMMULR{cond} <Rd>, <Rm>, <Rs>	

Table 1-6 ARM instruction set summary (continued)

Operation	Assembler	
Unsigned 32 x 32, + two 32, to 64	UMAAL{cond} <RdLo>, <RdHi>, <Rm>, <Rs>	
Saturate, select, and pack	Signed saturation at bit position n	SSAT{cond} <Rd>, #<immed_5>, <Rm>{, <shift>}
	Unsigned saturation at bit position n	USAT{cond} <Rd>, #<immed_5>, <Rm>{, <shift>}
	Two 16 signed saturation at bit position n	SSAT16{cond} <Rd>, #<immed_4>, <Rm>
	Two 16 unsigned saturation at bit position n	USAT16{cond} <Rd>, #<immed_4>, <Rm>
	Select bytes from Rn/Rm based on GE flags	SEL{cond} <Rd>, <Rn>, <Rm>
	Pack low 16/32, high 16/32	PKHBT{cond} <Rd>, <Rn>, <Rm>{, LSL #<immed_5>}
	Pack high 16/32, low 16/32	PKHTB{cond} <Rd>, <Rn>, <Rm>{, ASR #<immed_5>}

Addressing mode 2 is summarized in Table 1-7.

Table 1-7 Addressing mode 2

Addressing mode	Assembler
Offset	-
Immediate offset	[<Rn>, #+<immed_12>]
Zero offset	[<Rn>]
Register offset	[<Rn>, +/-<Rm>]
Scaled register offset	[<Rn>, +/-<Rm>, LSL #<immed_5>]
	[<Rn>, +/-<Rm>, LSR #<immed_5>]
	[<Rn>, +/-<Rm>, ASR #<immed_5>]
	[<Rn>, +/-<Rm>, ROR #<immed_5>]
	[<Rn>, +/-<Rm>, RRX]
Pre-indexed offset	-
Immediate offset	[<Rn>], #+<immed_12>

Table 1-7 Addressing mode 2 (continued)

Addressing mode	Assembler
Zero offset	[<Rn>]
Register offset	[<Rn>, +/-<Rm>]!
Scaled register offset	[<Rn>, +/-<Rm>, LSL #<immed_5>]!
	[<Rn>, +/-<Rm>, LSR #<immed_5>]!
	[<Rn>, +/-<Rm>, ASR #<immed_5>]!
	[<Rn>, +/-<Rm>, ROR #<immed_5>]!
	[<Rn>, +/-<Rm>, RRX]!
Post-indexed offset	-
Immediate	[<Rn>], #+/-<immed_12>
Zero offset	[<Rn>]
Register offset	[<Rn>], +/-<Rm>
Scaled register offset	[<Rn>], +/-<Rm>, LSL #<immed_5>
	[<Rn>], +/-<Rm>, LSR #<immed_5>
	[<Rn>], +/-<Rm>, ASR #<immed_5>
	[<Rn>], +/-<Rm>, ROR #<immed_5>
	[<Rn>], +/-<Rm>, RRX

Addressing mode 2P, post-indexed only, is summarized in Table 1-8.

Table 1-8 Addressing mode 2P, post-indexed only

Addressing mode	Assembler
Post-indexed offset	-
Immediate offset	[<Rn>], #+/-<immed_12>
Zero offset	[<Rn>]
Register offset	[<Rn>], +/-<Rm>
Scaled register offset	[<Rn>], +/-<Rm>, LSL #<immed_5>

Table 1-8 Addressing mode 2P, post-indexed only (continued)

Addressing mode	Assembler
	[<Rn>], +/-<Rm>, LSR #<immed_5>
	[<Rn>], +/-<Rm>, ASR #<immed_5>
	[<Rn>], +/-<Rm>, ROR #<immed_5>
	[<Rn>], +/-<Rm>, RRX

Addressing mode 3 is summarized in Table 1-9.

Table 1-9 Addressing mode 3

Addressing mode	Assembler
Immediate offset	[<Rn>, #+/-<immed_8>]
Pre-indexed	[<Rn>, #+/-<immed_8>]!
Post-indexed	[<Rn>], #+/-<immed_8>
Register offset	[<Rn>, +/- <Rm>]
Pre-indexed	[<Rn>, +/- <Rm>]!
Post-indexed	[<Rn>], +/- <Rm>

Addressing mode 4 is summarized in Table 1-10.

Table 1-10 Addressing mode 4

Addressing mode		Stack type	
Block load		Stack pop (LDM, RFE)	
IA	Increment after	FD	Full descending
IB	Increment before	ED	Empty descending
DA	Decrement after	FA	Full ascending
DB	Decrement before	EA	Empty ascending
Block store		Stack push (STM, SRS)	
IA	IA Increment after	EA	Empty ascending

Table 1-10 Addressing mode 4

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

Addressing mode 5 is summarized in Table 1-11.

Table 1-11 Addressing mode 5

Addressing mode	Assembler
Immediate offset	[<Rn>, #+/-<immed_8*4>]
Immediate pre-indexed	[<Rn>, #+/-<immed_8*4>]!
Immediate pre-indexed	[<Rn>], #+/-<immed_8*4>
Unindexed	[<Rn>], <option>

Operand2 is summarized in Table 1-12.

Table 1-12 Operand2

Operation	Assembler
Immediate value	#<immed_8r>
Logical shift left	<Rm> LSL #<immed_5>
Logical shift right	<Rm> LSR #<immed_5>
Arithmetic shift right	<Rm> ASR #<immed_5>
Rotate right	<Rm> ROR #<immed_5>
Register	<Rm>
Logical shift left	<Rm> LSL <Rs>
Logical shift right	<Rm> LSR <Rs>

Table 1-12 Operand2 (continued)

Operation	Assembler
Arithmetic shift right	<Rm> ASR <Rs>
Rotate right	<Rm> ROR <Rs>
Rotate right extended	<Rm> RRX

Fields are summarized in Table 1-13.

Table 1-13 Fields

Suffix	Sets this bit in the MSR field_mask	MSR instruction bit number
c	Control field mask bit (bit 0)	16
x	Extension field mask bit (bit 1)	17
s	Status field mask bit (bit 2)	18
f	Flags field mask bit (bit 3)	19

Condition codes are summarized in Table 1-14.

Table 1-14 Condition codes

Suffix	Description
EQ	Equal
NE	Not equal
HS/CS	Unsigned higher or same
L0/CC	Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same

Table 1-14 Condition codes (continued)

Suffix	Description
GE	Greater or equal
LT	Less than
GT	Greater than
LE	Less than or equal
AL	Always

1.8.2 Thumb instruction set summary

The Thumb instruction set summary is given in Table 1-15.

Table 1-15 Thumb instruction set summary

Operation	Assembler	
Move	Immediate, update flags	MOV <Rd>, #<immed_8>
	LowReg to LowReg, update flags	MOV <Rd>, <Rm>
	HighReg to LowReg	MOV <Rd>, <Rm>
	LowReg to HighReg	MOV <Rd>, <Rm>
	HighReg to HighReg	MOV <Rd>, <Rm>
Arithmetic	Add	ADD <Rd>, <Rn>, #<immed_3>
	Add immediate	ADD <Rd>, #<immed_8>
	Add LowReg and LowReg, update flags	ADD <Rd>, <Rn>, <Rm>
	Add HighReg to LowReg	ADD <Rd>, <Rm>
	Add LowReg to HighReg	ADD <Rd>, <Rm>
	Add HighReg to HighReg	ADD <Rd>, <Rm>
	Add immediate to PC	ADD <Rd>, PC, #<immed_8*4>
	Add immediate to SP	ADD <Rd>, SP, #<immed_8*4>
	Add immediate to SP	ADD SP, #<immed_7*4> ADD SP, SP, #<immed_7*4>

Table 1-15 Thumb instruction set summary (continued)

Operation	Assembler
Add with carry	ADC <Rd>, <Rs>
Subtract immediate	SUB <Rd>, <Rn>, #<immed_3>
Subtract immediate	SUB <Rd>, #<immed_8>
Subtract	SUB <Rd>, <Rn>, <Rm>
Subtract immediate	SUB SP, #<immed_8>
Subtract immediate from SP	SUB <Rd>, #<immed_7*4>
Subtract with carry	SBC <Rd>, <Rm>
Negate	NEG <Rd>, <Rm>
Multiply	MUL <Rd>, <Rm>
Compare	
Compare immediate	CMP <Rn>, #<immed_8>
Compare LowReg and LowReg, update flags	CMP <Rn>, <Rm>
Compare LowReg and HighReg	CMP <Rn>, <Rm>
Compare HighReg and LowReg	CMP <Rn>, <Rm>
Compare HighReg and HighReg	CMP <Rn>, <Rm>
Compare negative	CMN <Rn>, <Rm>
Logical	
AND	AND <Rd>, <Rm>
XOR	EOR <Rd>, <Rm>
OR	ORR <Rd>, <Rm>
Bit clear	BIC <Rd>, <Rm>
Move NOT	MVN <Rd>, <Rm>
Test bits	TST <Rd>, <Rm>
Shift/Rotate	
Logical shift left	LSL <Rd>, <Rm>, #<immed_5> LSL <Rd>, <Rs>
Logical shift right	LSR <Rd>, <Rm>, #<immed_5> LSR <Rd>, <Rs>

Table 1-15 Thumb instruction set summary (continued)

Operation	Assembler
Arithmetic shift right	ASR <Rd>, <Rm>, #<immed_5> ASR <Rd>, <Rs>
Rotate right	ROR <Rd>, <Rs>
Branch	
Conditional	B{cond} <label>
Unconditional	B <label>
Branch with link	BL <label>
Branch, link and exchange	BLX <label>
Branch, link and exchange	BLX <Rm>
Branch and exchange	BX <Rm>
Load	
With immediate offset	-
Word	LDR <Rd>, [<Rn>, #<immed_5>]
Halfword	LDRH <Rd>, [<Rn>, #<immed_5*2>]
Byte	LDRB <Rd>, [<Rn>, #<immed_5*4>]
With register offset	-
Word	LDR <Rd>, [<Rn>, <Rm>]
Halfword	LDRH <Rd>, [<Rn>, <Rm>]
Signed halfword	LDRSH <Rd>, [<Rn>, <Rm>]
Byte	LDRB <Rd>, [<Rn>, <Rm>]
Signed byte	LDRSB <Rd>, [<Rn>, <Rm>]
PC-relative	LDR <Rd>, [PC, #<immed_8*4>]
SP-relative	LDR <Rd>, [SP, #<immed_8*4>]
Multiple	LDMIA <Rn>!, <reglist>
Store	
With immediate offset	-
Word	STR <Rd>, [<Rn>, #<immed_5*4>]
Halfword	STRH <Rd>, [<Rn>, #<immed_5*2>]

Table 1-15 Thumb instruction set summary (continued)

Operation		Assembler
	Byte	STRB <Rd>, [<Rn>, #<immed_5>]
	With register offset	-
	Word	STR <Rd>, [<Rn>, <Rm>]
	Halfword	STRH <Rd>, [<Rn>, <Rm>]
	Byte	STRB <Rd>, [<Rn>, <Rm>]
	SP-relative	STR <Rd>, [SP, #<immed_8*4>]
	Multiple	STMIA <Rn>!, <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>
Change state	Change processor state	CPS<effect> <iflags>
	Change endianness	SETEND <endian_specifier>
Byte-reverse	Byte-reverse word	REV <Rd>, <Rm>
	Byte-reverse halfword	REV16 <Rd>, <Rm>
	Byte-reverse signed halfword	REVSH <Rd>, <Rm>
Software interrupt		SWI <immed_8>
Software breakpoint		BKPT <immed_8>
Sign or zero extend	Sign extend 16 to 32	SEXT16 <Rd>, <Rm>
	Sign extend 8 to 32	SEXT8 <Rd>, <Rm>
	Zero extend 16 to 32	UEXT16 <Rd>, <Rm>
	Zero extend 8 to 32	UEXT8 <Rd>, <Rm>

1.9 Silicon revision information

There are no functional differences between ARM1136 r0p0 and ARM1136 r0p1.

Chapter 2

Programmer's Model

This chapter describes the ARM1136JF-S registers and provides information for programming the microprocessor. It contains the following sections:

- *About the programmer's model* on page 2-2
- *Processor operating states* on page 2-3
- *Instruction length* on page 2-4
- *Data types* on page 2-5
- *Memory formats* on page 2-6
- *Addresses in an ARM1136JF-S system* on page 2-8
- *Operating modes* on page 2-9
- *Registers* on page 2-10
- *The program status registers* on page 2-16
- *Exceptions* on page 2-23.

2.1 About the programmer's model

The ARM1136JF-S processor implements ARM architecture v6 with Java extensions. This includes the 32-bit ARM instruction set, 16-bit Thumb instruction set, and the 8-bit Java instruction set. For details of both the ARM and Thumb instruction sets, see the *ARM Architecture Reference Manual*. For the Java instruction set see the *Jazelle V1 Architecture Reference Manual*.

2.2 Processor operating states

The ARM1136JF-S processor has three operating states:

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

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

Java state Variable length, byte-aligned Java instructions.

In Thumb state, the *Program Counter* (PC) uses bit 1 to select between alternate halfwords. In Java state, all instruction fetches are in words.

Note

Transition between ARM and Thumb states does not affect the processor mode or the register contents. For details on entering and exiting Java state see *Jazelle VI Architecture Reference Manual*.

2.2.1 Switching state

You can switch the operating state of the ARM1136JF-S processor 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*.
- ARM state and Java state using the BXJ instruction.

All exceptions are entered, handled, and exited in ARM state. If an exception occurs in Thumb state or Java state, the processor reverts to ARM state. Exception return instructions restore the SPSR to the CPSR, which can also cause a transition back to Thumb state or Java state.

2.2.2 Interworking ARM and Thumb state

The ARM1136JF-S processor enables you to mix ARM and Thumb code. For details see the chapter about interworking ARM and Thumb in the *RealView Compilation Tools Developer Guide*.

2.3 Instruction length

Instructions are one of:

- 32 bits long (in ARM state)
- 16 bits long (in Thumb state)
- variable length, multiples of 8 bits (in Java state).

2.4 Data types

The ARM1136JF-S processor supports the following data types:

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

Note

- When any of these types are described as unsigned, the N-bit data value represents a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.
- When any of these types are described as signed, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

For best performance 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.

ARM1136JF-S processor introduces mixed-endian and unaligned access support. For details see Chapter 4 *Unaligned and Mixed-Endian Data Access Support*.

Note

You cannot use LDRD, LDM, LDC, STRD, STM, or STC instructions to access 32-bit quantities if they are unaligned.

2.5 Memory formats

The ARM1136JF-S processor views memory as a linear collection of bytes numbered in ascending order from zero. Bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word, for example.

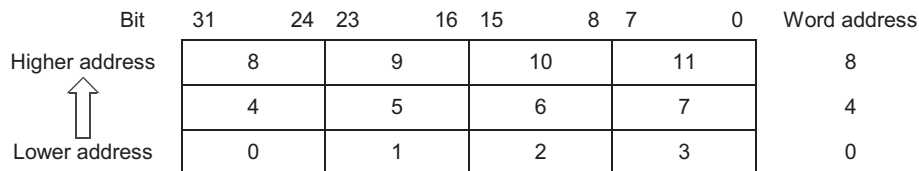
The ARM1136JF-S processor can treat words in memory as being stored in either:

- *Legacy big-endian format*
- *Little-endian format.*

Additionally, the ARM1136JF-S processor supports mixed-endian and unaligned data accesses. For details see Chapter 4 *Unaligned and Mixed-Endian Data Access Support*.

2.5.1 Legacy big-endian format

In legacy big-endian format, the ARM1136JF-S processor stores the most significant byte of a word at the lowest-numbered byte, and the least significant byte at the highest-numbered byte. Therefore, byte 0 of the memory system connects to data lines 31-24. This is shown in Figure 2-1.

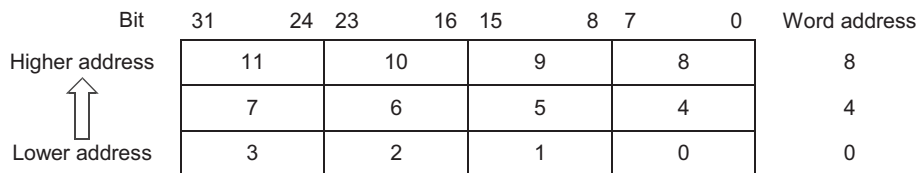


- Most significant byte is at lowest address
- Word is addressed by byte address of most significant byte

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

2.5.2 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 7-0. This is shown in Figure 2-2 on page 2-7.



- 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

2.6 Addresses in an ARM1136JF-S system

Three distinct types of address exist in an ARM1136JF-S system:

- *Virtual Address (VA)*
- *Modified Virtual Address (MVA)*
- *Physical Address (PA)*.

Table 2-1 shows the address types in an ARM1136JF-S system.

Table 2-1 Address types in an ARM1136JF-S system

ARM1136JF-S processor	Caches	TLBs	AMBA bus
Virtual Address	Virtual index Physical Address	Translates Virtual Address to Physical Address	Physical Address

This is an example of the address manipulation that occurs when the ARM1136JF-S processor requests an instruction (see Figure 1-1 on page 1-4):

1. The VA of the instruction is issued by the ARM1136JF-S processor.
2. The Instruction Cache is indexed by the lower bits of the VA. The VA is translated using the ProcID to the MVA, and then to PA in the *Translation Lookaside Buffer (TLB)*. The TLB performs the translation in parallel with the Cache lookup.
3. If the protection check carried out by the TLB on the MVA does not abort and the PA tag is in the Instruction Cache, the instruction data is returned to the ARM1136JF-S processor.
4. The PA is passed to the AMBA bus interface to perform an external access, in the event of a cache miss.

2.7 Operating modes

In all states there are 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.

2.8 Registers

The ARM1136JF-S processor has a total of 37 registers:

- 31 general-purpose 32-bit registers
- six 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.

2.8.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are accessible at any time. In privileged modes, mode-specific banked registers become available. Figure 2-3 on page 2-12 shows which registers are available in each mode.

The ARM state register set contains 16 directly-accessible registers, r0-r15. Another register, the *Current Program Status Register* (CPSR), contains condition code flags, status bits, and current mode bits. Registers r0-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 the return address 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 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 this is word-aligned
 - in Thumb state this is halfword-aligned
 - in Java state this is byte-aligned.

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

Banked registers have a mode identifier that indicates which mode they relate to. These mode identifiers are listed in Table 2-2.

Table 2-2 Register mode identifiers

Mode	Mode identifier
User	usr ^a
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usr ^a
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 have to save any registers.

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

Figure 2-3 on page 2-12 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	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_und
------	------------------	------------------	------------------	------------------	------------------


 = banked register

Figure 2-3 Register organization in ARM state

Figure 2-4 on page 2-13 shows an alternative view of the ARM registers.

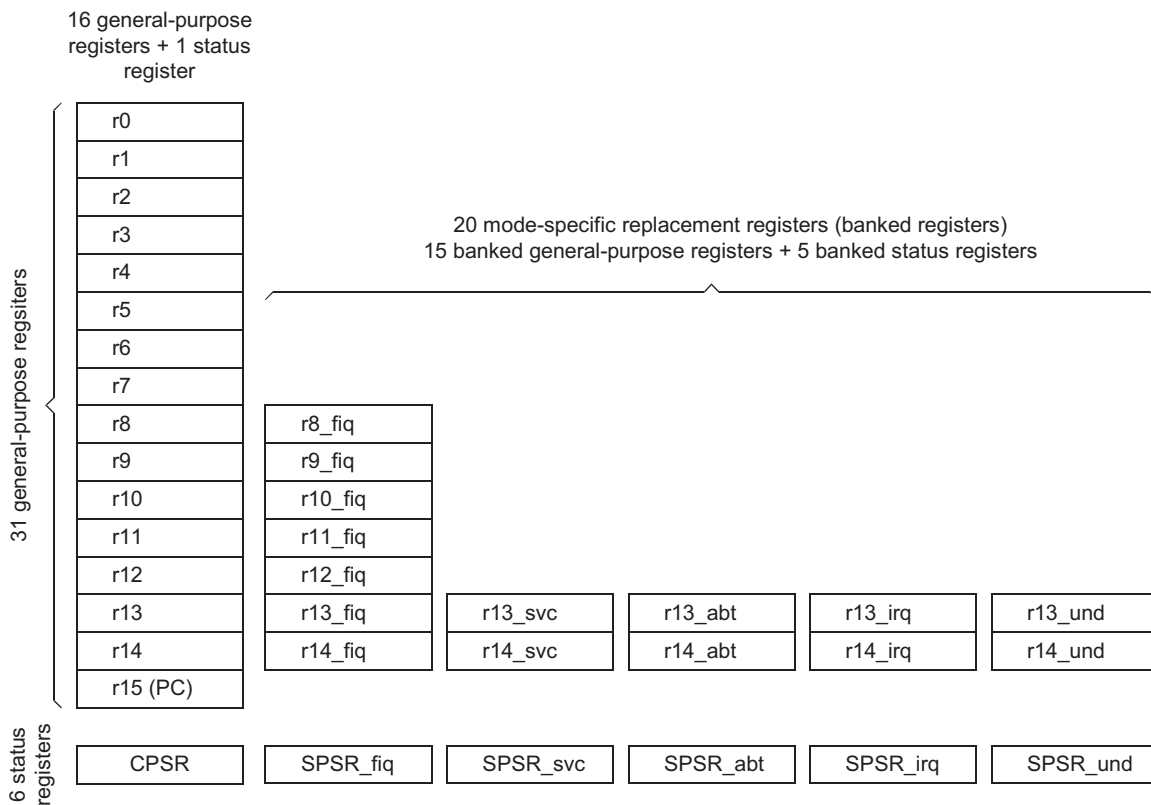


Figure 2-4 ARM1136JF-S register set showing banked registers

2.8.2 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 2-14)
- 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. The Thumb state register set is shown in Figure 2-5.

Thumb state general registers and program counter

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

Thumb state program status registers

CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_und
------	------------------	------------------	------------------	------------------	------------------

 = banked register

Figure 2-5 Register organization in Thumb state

2.8.3 Accessing high registers in Thumb state

In Thumb state, the high registers, r8–r15, are not part of the standard register set. You can use special variants of the MOV instruction to transfer a value from a low register, in the range r0–r7, to a high register, and from a high register to a low register. The CMP instruction enables you to compare high register values with low register values. The ADD instruction enables you to add high register values to low register values. For more details, see the *ARM Architecture Reference Manual*.

2.8.4 ARM state and Thumb state registers relationship

The relationships between the Thumb state and ARM state registers are shown in Figure 2-6. See the *Jazelle V1 Architecture Reference Manual* for details of Java state registers.

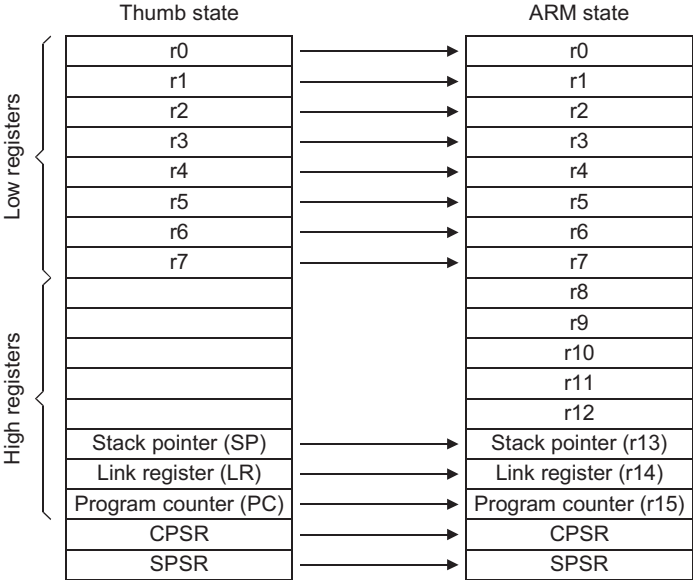


Figure 2-6 ARM state and Thumb state registers relationship

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

2.9 The program status registers

The ARM1136JF-S processor contains one 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 2-7, and described in the sections from *The condition code flags* to *Reserved bits* on page 2-22 inclusive.

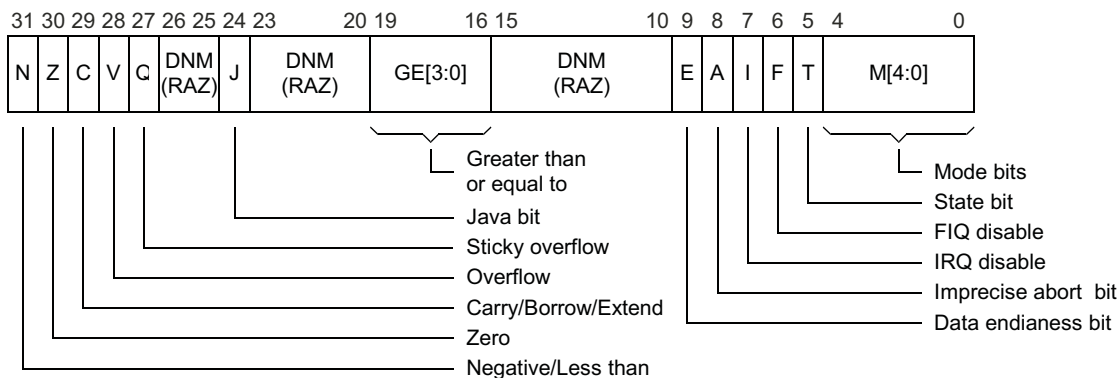


Figure 2-7 Program status register

Note

The bits identified in Figure 2-7 as *Do Not Modify* (DNM) (*Read As Zero* (RAZ)) must not be modified by software. These bits are:

- Readable, to enable the processor state to be preserved (for example, during process context switches)
- Writable, to enable 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.

2.9.1 The condition code flags

The N, Z, C, and V bits are the condition code flags. You can set them by arithmetic and logical operations, and also by MSR and LDM instructions. The ARM1136JF-S processor tests these flags to determine whether to execute an instruction.

In ARM state, most instructions can execute conditionally on the state of the N, Z, C, and V bits. The exceptions are:

- BKPT
- CDP2
- CPS
- LDC2
- MCR2
- MCRR2
- MRC2
- MRRC2
- PLD
- SETEND
- RFE
- SRS
- STC2.

In Thumb state, only the Branch instruction can be executed conditionally. For more information about conditional execution, see the *ARM Architecture Reference Manual*.

2.9.2 The Q flag

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

- QADD
- QDADD
- QSUB
- QDSUB
- SMLAD
- SMLAxy
- SMLAWy
- SMLSD
- SMUAD
- SSAT
- SSAT16
- USAT
- USAT16.

The Q flag is sticky in that, when set by an instruction, it remains set until explicitly cleared by an MSR instruction writing to the 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 Architecture Reference Manual*.

2.9.3 The J bit

The J bit in the CPSR indicates when the ARM1136JF-S processor is in Java state.

When:

J = 0 The processor is in ARM or Thumb state, depending on the T bit.

J = 1 The processor is in Java state.

———— **Note** —————

- The combination of J = 1 and T = 1 causes similar effects to setting T=1 on a non Thumb-aware processor. That is, the next instruction executed causes entry to the Undefined Instruction exception. Entry to the exception handler causes the processor to re-enter ARM state, and the handler can detect that this was the cause of the exception because J and T are both set in SPSR_und.
- MSR cannot be used to change the J bit in the CPSR.
- The placement of the J bit avoids the status or extension bytes in code running on ARMv5TE or earlier processors. This ensures that OS code written using the deprecated CPSR, SPSR, CPSR_all, or SPSR_all syntax for the destination of an MSR instruction continues to work.

2.9.4 The GE[3:0] bits

Some of the SIMD instructions set GE[3:0] as greater-than-or-equal bits for individual halfwords or bytes of the result, as shown in Table 2-3 on page 2-19.

Table 2-3 GE[3:0] settings

	GE[3]	GE[2]	GE[1]	GE[0]
Instruction	A op B ≥ C	A op B ≥ C	A op B ≥ C	A op B ≥ C
Signed				
SADD16	$[31:16] + [31:16] \geq 0$	$[31:16] + [31:16] \geq 0$	$[15:0] + [15:0] \geq 0$	$[15:0] + [15:0] \geq 0$
SSUB16	$[31:16] - [31:16] \geq 0$	$[31:16] - [31:16] \geq 0$	$[15:0] - [15:0] \geq 0$	$[15:0] - [15:0] \geq 0$
SADDSUBX	$[31:16] + [15:0] \geq 0$	$[31:16] + [15:0] \geq 0$	$[15:0] - [31:16] \geq 0$	$[15:0] - [31:16] \geq 0$
SSUBADDX	$[31:16] - [15:0] \geq 0$	$[31:16] - [15:0] \geq 0$	$[15:0] + [31:16] \geq 0$	$[15:0] + [31:16] \geq 0$
SADD8	$[31:24] + [31:24] \geq 0$	$[23:16] + [23:16] \geq 0$	$[15:8] + [15:8] \geq 0$	$[7:0] + [7:0] \geq 0$
SSUB8	$[31:24] - [31:24] \geq 0$	$[23:16] - [23:16] \geq 0$	$[15:8] - [15:8] \geq 0$	$[7:0] - [7:0] \geq 0$
Unsigned				
UADD16	$[31:16] + [31:16] \geq 2^{16}$	$[31:16] + [31:16] \geq 2^{16}$	$[15:0] + [15:0] \geq 2^{16}$	$[15:0] + [15:0] \geq 2^{16}$
USUB16	$[31:16] - [31:16] \geq 0$	$[31:16] - [31:16] \geq 0$	$[15:0] - [15:0] \geq 0$	$[15:0] - [15:0] \geq 0$
UADDSUBX	$[31:16] + [15:0] \geq 2^{16}$	$[31:16] + [15:0] \geq 2^{16}$	$[15:0] - [31:16] \geq 0$	$[15:0] - [31:16] \geq 0$
USUBADDX	$[31:16] - [15:0] \geq 0$	$[31:16] - [15:0] \geq 0$	$[15:0] + [31:16] \geq 2^{16}$	$[15:0] + [31:16] \geq 2^{16}$
UADD8	$[31:24] + [31:24] \geq 2^8$	$[23:16] + [23:16] \geq 2^8$	$[15:8] + [15:8] \geq 2^8$	$[7:0] + [7:0] \geq 2^8$
USUB8	$[31:24] - [31:24] \geq 0$	$[23:16] - [23:16] \geq 0$	$[15:8] - [15:8] \geq 0$	$[7:0] - [7:0] \geq 0$

Note

GE bit is 1 if A op B ≥ C, otherwise 0.

The SEL instruction uses GE[3:0] to select which source register supplies each byte of its result.

Note

- For unsigned operations, the GE bits are determined by the usual ARM rules for carries out of unsigned additions and subtractions, and so are carry-out bits.
- For signed operations, the rules for setting the GE bits are chosen so that they have the same sort of greater than or equal functionality as for unsigned operations.

2.9.5 The E bit

ARM and Thumb instructions are provided to set and clear the E-bit. The E bit controls load/store endianness. For details of where the E bit is used see Chapter 4 *Unaligned and Mixed-Endian Data Access Support*.

Architecture versions prior to ARMv6 specify this bit as SBZ. This ensures no endianness reversal on loads or stores.

2.9.6 The A bit

The A bit is set automatically. It is used to disable imprecise Data Aborts. For details of how to use the A bit see *Imprecise Data Abort mask in the CPSR/SPSR* on page 2-37.

2.9.7 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 2-21.

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

Interrupt disable bits

The I and F bits are the interrupt disable bits:

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

T bit

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor is executing in ARM state, or Java state depending on the J bit.

———— Note —————

Never use an MSR instruction to force a change to the state of the T bit in the CPSR. If an MSR instruction does try to modify this bit the result is architecturally Unpredictable. In the ARM1136JF-S processor this bit is not affected.

Mode bits

Caution

An illegal value programmed into M[4:0] causes the processor to enter an unrecoverable state. If this occurs, you must apply reset. Not all combinations of the mode bits define a valid processor mode, so take care to use only those bit combinations shown.

M[4:0] are the mode bits. These bits determine the processor operating mode as shown in Table 2-4.

Table 2-4 PSR mode bit values

M[4:0]	Mode	Visible state registers	
		Thumb	ARM
b10000	User	r0-r7, r8-r12 ^a , SP, LR, PC, CPSR	r0-r14, PC, CPSR
b10001	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
b10010	IRQ	r0-r7, r8-r12 ^a , SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0-r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
b10011	Supervisor	r0-r7, r8-r12 ^a , SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0-r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc
b10111	Abort	r0-r7, r8-r12 ^a , SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0-r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
b11011	Undefined	r0-r7, r8-r12 ^a , SP_und, LR_und, PC, CPSR, SPSR_und	r0-r12, r13_und, r14_und, PC, CPSR, SPSR_und
b11111	System	r0-r7, r8-r12 ^a , SP, LR, PC, CPSR	r0-r14, PC, CPSR

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

2.9.8 Modification of PSR bits by MSR instructions

In previous architecture versions, MSR instructions can modify the flags byte, bits [31:24], of the CPSR in any mode, but the other three bytes are only modifiable in privileged modes.

After the introduction of ARM architecture v6, however, each CPSR bit falls into one of the following categories:

- Bits that are freely modifiable from any mode, either directly by MSR instructions or by other instructions whose side-effects include writing the specific bit or writing the entire CPSR.
Bits in Figure 2-7 on page 2-16 that are in this category are N, Z, C, V, Q, GE[3:0], and E.
- Bits that must never be modified by an MSR instruction, and so must only be written as a side-effect of another instruction. If an MSR instruction does try to modify these bits the results are architecturally Unpredictable. In the ARM1136JF-S processor these bits are not affected.
Bits in Figure 2-7 on page 2-16 that are in this category are J and T.
- Bits that can only be modified from privileged modes, and that are completely protected from modification by instructions while the processor is in User mode. The only way that these bits can be modified while the processor is in User mode is by entering a processor exception, as described in *Exceptions* on page 2-23.
Bits in Figure 2-7 on page 2-16 that are in this category are A, I, F, and M[4:0].

2.9.9 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.

2.10 Exceptions

Exceptions occur 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 ARM1136JF-S processor preserves the current processor state so that the original program can resume when the handler routine has finished.

If two or more exceptions occur simultaneously, the exceptions are dealt with in the fixed order given in *Exception priorities* on page 2-40.

This section provides details of the ARM1136JF-S exception handling:

- *Exception entry and exit summary* on page 2-25
- *Entering an ARM exception* on page 2-26
- *Leaving an ARM exception* on page 2-26.

Several enhancements are made in ARM architecture v6 to the exception model, mostly to improve interrupt latency, as follows:

- New instructions are added to give a choice of stack to use for storing the exception return state after exception entry, and to simplify changes of processor mode and the disabling and enabling of interrupts.
- The interrupt vector definitions on ARMv6 are changed to support the addition of hardware to prioritize the interrupt sources and to look up the start vector for the related interrupt handling routine.
- A low interrupt latency configuration is added in ARMv6. In terms of the instruction set architecture, it specifies that multi-access load/store instructions (ARM LDC, LDM, LDRD, STC, STM, and STRD, and Thumb LDMIA, POP, PUSH, and STMIA) can be interrupted and then restarted after the interrupt has been processed.
- Support for an imprecise Data Abort that behaves as an interrupt rather than as an abort, in that it occurs asynchronously relative to the instruction execution. Support involves the masking of a pending imprecise Data Abort at times when entry into Abort mode is deemed unrecoverable.

2.10.1 Changes to existing interrupt vectors

In ARMv5, the IRQ and FIQ exception vectors are fixed unless high vectors are enabled. Interrupt handlers typically have to start with an instruction sequence to determine the cause of the interrupt and branch to a routine to handle it.

On ARM1136JF-S processors the IRQ exception can be determined directly from the value presented on the *Vectored Interrupt Controller (VIC)* port. The vector interrupt behavior is explicitly enabled when the VE bit in CP15 c1 is set. See Chapter 12 *Vectored Interrupt Controller Port*.

An example of a hardware block that can interface to the VIC port is the PrimeCell VIC (PL192), which is available from ARM. This takes a set of inputs from various interrupt sources, prioritizes them, and presents the interrupt type of the highest-priority interrupt being requested and the address of its handler to the processor core. The VIC also masks any lower priority interrupts. Such hardware reduces the time taken to enter the handling routine for the required interrupt.

2.10.2 New instructions for exception handling

This section describes the instructions added to accelerate the handling of exceptions. Full details of these instructions are given in the *ARM Architecture Reference Manual*.

Store Return State (SRS)

This instruction stores r14_<current_mode> and spsr_<current_mode> to sequential addresses, using the banked version of r13 for a specified mode to supply the base address (and to be written back to if base register Write-Back is specified). This enables an exception handler to store its return state on a stack other than the one automatically selected by its exception entry sequence.

The addressing mode used is a version of ARM addressing mode 4 (see the *ARM Architecture Reference Manual, Part A*), modified to assume a {r14,SPSR} register list, rather than using a list specified by a bit mask in the instruction. This enables the SRS instruction to access stacks in a manner compatible with the normal use of STM instructions for stack accesses.

Return From Exception (RFE)

This instruction loads the PC and CPSR from sequential addresses. This is used to return from an exception that has had its return state saved using the SRS instruction (see *Store Return State (SRS)*), and again uses a version of ARM addressing mode 4, modified this time to assume a {PC,CPSR} register list.

Change Processor State (CPS)

This instruction provides new values for the CPSR interrupt masks, mode bits, or both, and is designed to shorten and speed up the read/modify/write instruction sequence used in ARMv5 to perform such tasks. Together with the SRS instruction, it enables an

exception handler to save its return information on the stack of another mode and then switch to that other mode, without modifying the stack belonging to the original mode or any registers other than the new mode stack pointer.

This instruction also streamlines interrupt mask handling and mode switches in other code. In particular it enables short code sequences to be made atomic efficiently in a uniprocessor system by disabling interrupts at their start and re-enabling interrupts at their end. A similar Thumb instruction is also provided. However, the Thumb instruction can only change the interrupt masks, not the processor mode as well, to avoid using too much instruction set space.

2.10.3 Exception entry and exit summary

Table 2-5 summarizes the PC value preserved in the relevant r14 on exception entry, and the recommended instruction for exiting the exception handler. Full details of Java state exceptions are provided in the *Jazelle VI Architecture Reference Manual*.

Table 2-5 Exception entry and exit

Exception or entry	Return instruction	Previous state			Notes
		ARM r14_x	Thumb r14_x	Java r14_x	
SWI	MOVS PC, R14_svc	PC + 4	PC+2	-	Where the PC is the address of the SWI or undefined instruction. Not used in Java state.
UNDEF	MOVS PC, R14_und	PC + 4	PC+2	-	
PABT	SUBS PC, R14_abt, #4	PC + 4	PC+4	PC+4	Where the PC is the address of instruction that had the Prefetch Abort.
FIQ	SUBS PC, R14_fiq, #4	PC + 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	PC+4	
DABT	SUBS PC, R14_abt, #8	PC + 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 on reset is Unpredictable.
BKPT	SUBS PC, R14_abt, #4	PC + 4	PC+4	PC+4	Software breakpoint.

2.10.4 Entering an ARM exception

When handling an ARM exception the ARM1136JF-S processor:

1. Preserves the address of the next instruction in the appropriate LR. When the exception entry is from:

ARM and Java states:

The ARM1136JF-S processor 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

Thumb state:

The ARM1136JF-S processor writes the value of the PC into the LR, offset by a value (current PC + 2, 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 have to determine the state when entering an exception. For example, in the case of a SWI, `MOVS PC, r14_svc` always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.

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

The ARM1136JF-S processor 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 or Java state and an exception occurs, the switch to ARM state takes place automatically when the exception vector address is loaded into the PC.

2.10.5 Leaving an ARM 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 2-5 on page 2-25.

Typically the return instruction is an arithmetic or logical operation with the S bit set and `rd = r15`, so the core copies the SPSR back to the CPSR.

Note

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

2.10.6 Reset

When the **nRESETIN** signal is driven LOW a reset occurs, and the ARM1136JF-S processor abandons the executing instruction.

When **nRESETIN** is driven HIGH again the ARM1136JF-S processor:

1. Forces CPSR M[4:0] to b10011 (Supervisor mode), sets the A, I, and F bits in the CPSR, and clears the CPSR T bit and J bit. The E bit is set based on the state of the **BIGENDINIT** and **UBITINIT** pins. 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.

See Chapter 9 *Clocking and Resets* for more details of the reset behavior for the ARM1136JF-S processor.

2.10.7 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 ARM1136JF-S processor. It is the output of this register that is used by the ARM1136JF-S processor control logic.

Irrespective of whether exception entry is from ARM state, Thumb state, or Java 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 ARM1136JF-S processor 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. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable FIQs and interrupts.

2.10.8 Interrupt request

The 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.

Irrespective of whether exception entry is from ARM state, Thumb state, or Java 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 ARM1136JF-S processor checks for a LOW level on the output of the nIRQ register at the end of each instruction.

IRQs are disabled when an IRQ occurs. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable IRQs.

2.10.9 Low interrupt latency configuration

The FI bit, bit 21, in CP15 register 1 enables a low interrupt latency configuration. This mode reduces the interrupt latency of the ARM1136JF-S processor. This is achieved by:

- disabling *Hit-Under-Miss* (HUM) functionality
- abandoning restartable external accesses so that the core can react to a pending interrupt faster than is normally the case
- recognizing low-latency interrupts as early as possible in the main pipeline.

To ensure that a change between normal and low interrupt latency configurations is synchronized correctly, the FI bit must only be changed in using the sequence:

1. Drain Write Buffer.
2. Change FI Bit.
3. Drain Write Buffer with interrupt disabled.

You must ensure that software systems only change the FI bit shortly after Reset, while interrupts are disabled.

In low interrupt latency configuration, software must only use multi-word load/store instructions in ways that are fully restartable. In particular, they must not be used on memory locations that produce non-idempotent side-effects for the type of memory access concerned.

This enables, but does not require, implementations to make these instructions interruptible when in low interrupt latency configuration. If the instruction is interrupted before it is complete, the result might be that one or more of the words are accessed twice, but the idempotency of the side-effects, if any, of the memory accesses ensures that this does not matter.

Note

There is a similar existing requirement with unaligned and multi-word load/store instructions that access memory locations that can abort in a recoverable way. An abort on one of the words accessed can cause a previously-accessed word to be accessed twice, once before the abort and again after the abort handler has returned. The requirement in this case is either:

- all side-effects are idempotent
- the abort must either occur on the first word accessed or not at all.

The instructions that this rule currently applies to are:

- ARM instructions LDC, all forms of LDM, LDRD, STC, all forms of STM, STRD, and unaligned LDR, STR, LDRH, and STRH
- Thumb instructions LDMIA, PUSH, POP, and STMIA, and unaligned LDR, STR, LDRH, and STRH.

System designers are also advised that memory locations accessed with these instructions must not have large numbers of wait-states associated with them if the best possible interrupt latency is to be achieved.

2.10.10 Interrupt latency example

This section gives an extended example to show how the combination of new facilities improves interrupt latency. The example is not necessarily entirely realistic, but illustrates the main points.

The assumptions made are:

1. *Vector Interrupt Controller* (VIC) hardware exists to prioritize interrupts and to supply the address of the highest priority interrupt to the processor core on demand.

In the ARMv5 system, the address is supplied in a memory-mapped I/O location, and loading it acts as an entering interrupt handler acknowledgement to the VIC. In the ARMv6 system, the address is loaded and the acknowledgement given automatically, as part of the interrupt entry sequence. In both systems, a store to a memory-mapped I/O location is used to send a finishing interrupt handler acknowledgement to the VIC.

2. The system has the following layers:

Real-time layer

Contains handlers for a number of high-priority interrupts. These interrupts can be prioritized, and are assumed to be signaled to the processor core by means of the FIQ interrupt. Their handlers do not use the facilities supplied by the other two layers. This means that all memory they use must be locked down in the TLBs and caches. (It is possible to use additional code to make access to nonlocked memory possible, but this is not discussed in this example.)

Architectural completion layer

Contains Prefetch Abort, Data Abort and Undefined instruction handlers whose purpose is to give the illusion that the hardware is handling all memory requests and instructions on its own, without requiring software to handle TLB misses, virtual memory misses, and near-exceptional floating-point operations, for example. This illusion is not available to the real-time layer, because the software handlers concerned take a significant number of cycles, and it is not reasonable to have every memory access to take large numbers of cycles. Instead, the memory concerned has to be locked down.

Non real-time layer

Provides interrupt handlers for low-priority interrupts. These interrupts can also be prioritized, and are assumed to be signaled to the processor core using the IRQ interrupt.

3. The corresponding exception priority structure is as follows, from highest to lowest priority:

- a. FIQ1 (highest priority FIQ)
- b. FIQ2
- c. ...
- d. FIQm (lowest priority FIQ)
- e. Data Abort
- f. Prefetch Abort
- g. Undefined instruction
- h. SWI
- i. IRQ1 (highest priority IRQ)
- j. IRQ2
- k. ...

1. IRQn (lowest priority IRQ)

The processor core prioritization handles most of the priority structure, but the VIC handles the priorities within each group of interrupts.

————— Note —————

This list reflects the priorities that the handlers are subject to, and differs from the priorities that the exception entry sequences are subject to. The latter priorities are presented in the *ARM Architecture Reference Manual Part A*, and the difference occurs because simultaneous Data Abort and FIQ exceptions result in the sequence:

- a. Data Abort entry sequence executed, updating r14_abt, SPSR_abt, PC, and CPSR.
- b. FIQ entry sequence executed, updating r14_fiq, SPSR_fiq, PC, and CPSR.
- c. FIQ handler executes to completion and returns.
- d. Data Abort handler executes to completion and returns.

4. Stack and register usage is:

- The FIQ1 interrupt handler has exclusive use of r8_fiq to r12_fiq. In ARMv5, r13_fiq points to a memory area, that is mainly for use by the FIQ1 handler. However, a few words are used during entry for other FIQ handlers. In ARMv6, the FIQ1 interrupt handler has exclusive use of r13_fiq.
- The Undefined instruction, Prefetch Abort, Data Abort, and non-FIQ1 FIQ handlers use the stack pointed to by r13_abt. This stack is locked down in memory, and therefore of known, limited depth.
- All IRQ and SWI handlers use the stack pointed to by r13_svc. This stack does not have to be locked down in memory.
- The stack pointed to by r13_usr is used by the current process. This process can be privileged or unprivileged, and uses System or User mode accordingly.

5. Timings are roughly consistent with ARM10 timings, with the pipeline reload penalty being three cycles. It is assumed that pipeline reloads are combined to execute as quickly as reasonably possible, and in particular that:

- If an interrupt is detected during an instruction that has set a new value for the PC, after that value has been determined and written to the PC but before the resulting pipeline refill is completed, the pipeline refill is abandoned and the interrupt entry sequence started as soon as possible.

- Similarly, if an FIQ is detected during an exception entry sequence that does not disable FIQs, after the updates to r14, the SPSR, the CPSR, and the PC but before the pipeline refill has completed, the pipeline refill is abandoned and the FIQ entry sequence started as soon as possible.

FIQs in the example system in ARMv5

In ARMv5, all FIQ interrupts come through the same vector, at address `0x0000001C` or `0xFFFF001C`. To implement the above system, the code at this vector must get the address of the correct handler from the VIC, branch to it, and transfer to using `r13_abt` and the Abort mode stack if it is not the FIQ1 handler. The following code does, assuming that `r8_fiq` holds the address of the VIC:

```

FIQhandler
    LDR    PC, [R8,#HandlerAddress]
    ...
FIQ1handler
... Include code to process the interrupt ...
    STR    R0, [R8,#AckFinished]
    SUBS   PC, R14, #4
    ...
FIQ2handler
    STMIA  R13, {R0-R3}
    MOV    R0, LR
    MRS    R1, SPSR
    ADD    R2, R13, #8
    MRS    R3, CPSR
    BIC    R3, R3, #0x1F
    ORR    R3, R3, #0x1B ; = Abort mode number
    MSR    CPSR_c, R3
    STMFD  R13!, {R0,R1}
    LDMIA  R2, {R0,R1}
    STMFD  R13!, {R0,R1}
    LDMDB  R2, {R0,R1}
    BIC    R3, R3, #0x40 ; = F bit
    MSR    CPSR_c, R3
... FIQs are now re-enabled, with original R2, R3, R14, SPSR on stack
... Include code to stack any more registers required, process the interrupt
... and unstack extra registers
    ADR    R2, #VICaddress
    MRS    R3, CPSR
    ORR    R3, R3, #0x40 ; = F bit
    MSR    CPSR_c, R3
    STR    R0, [R2,#AckFinished]
    LDR    R14, [R13,#12] ; Original SPSR value
    MSR    SPSR_fsxc, R14
    LDMFD  R13!, {R2,R3,R14}

```

```

        ADD    R13, R13, #4
        SUBS   PC, R14, #4
...

```

The major problem with this is the length of time that FIQs are disabled at the start of the lower priority FIQs. The worst-case interrupt latency for the FIQ1 interrupt occurs if a lower priority FIQ2 has fetched its handler address, and is approximately:

- 3 cycles for the pipeline refill after the LDR PC instruction fetches the handler address
- + 24 cycles to get to and execute the MSR instruction that re-enables FIQs
- + 3 cycles to re-enter the FIQ exception
- + 5 cycles for the LDR PC instruction at FIQhandler
- = 35 cycles.

Note

FIQs must be disabled for the final store to acknowledge the end of the handler to the VIC. Otherwise, more badly timed FIQs, each occurring close to the end of the previous handler, can cause unlimited growth of the locked-down stack.

FIQs in the example system in ARMv6

Using the VIC and the new instructions, there is no longer any requirement for everything to go through the single FIQ vector, and the changeover to a different stack occurs much more smoothly. The code is:

```

FIQ1handler
... Include code to process the interrupt ...
    STR    R0, [R8,#AckFinished]
    SUBS   PC, R14, #4
...
FIQ2handler
    SUB    R14, R14, #4
    SRSFD  R13_abt!
    CPSIE  f, #0x1B ; = Abort mode
    STMFD  R13!, {R2,R3}
... FIQs are now re-enabled, with original R2, R3, R14, SPSR on stack
... Include code to stack any more registers required, process the interrupt
... and unstack extra registers
    LDMFD  R13!, {R2,R3}
    ADR    R14, #VICaddress
    CPSID  f
    STR    R0, [R14,#AckFinished]

```

RFEFD R13!

...

The worst-case interrupt latency for a FIQ1 now occurs if the FIQ1 occurs during an FIQ2 interrupt entry sequence, after it disables FIQs, and is approximately:

- 3 cycles for the pipeline refill for the FIQ2 exception entry sequence
- + 5 cycles to get to and execute the CPSIE instruction that re-enables FIQs
- + 3 cycles to re-enter the FIQ exception
- = 11 cycles.

Note

In the ARMv5 system, the potential additional interrupt latency caused by a long LDM or STM being in progress when the FIQ is detected was only significant because the memory system could stretch its cycles considerably. Otherwise, it was dwarfed by the number of cycles lost because of FIQs being disabled at the start of a lower-priority interrupt handler. In ARMv6, this is still the case, but it is a lot closer.

Alternatives to the example system

Two alternatives to the design in *FIQs in the example system in ARMv6* on page 2-33 are:

- The first alternative is not to reserve the FIQ registers for the FIQ1 interrupt, but instead either to:
 - share them out among the various FIQ handlers

The first restricts the registers available to the FIQ1 handler and adds the software complication of managing a global allocation of FIQ registers to FIQ handlers. Also, because of the shortage of FIQ registers, it is not likely to be very effective if there are many FIQ handlers.
 - require the FIQ handlers to treat them as normal callee-save registers.

The second adds a number of cycles of loading important addresses and variable values into the registers to each FIQ handler before it can do any useful work. That is, it increases the effective FIQ latency by a similar number of cycles.

- The second alternative is to use IRQs for all but the highest priority interrupt, so that there is only one level of FIQ interrupt. This achieves very fast FIQ latency, 5-8 cycles, but at a cost to all the lower-priority interrupts that every exception entry sequence now disables them. You then have the following possibilities:
 - None of the exception handlers in the architectural completion layer re-enable IRQs. In this case, all IRQs suffer from additional possible interrupt latency caused by those handlers, and so effectively are in the non real-time layer. In other words, this results in there only being one priority for interrupts in the real-time layer.
 - All of the exception handlers in the architectural completion layer re-enable IRQs to permit IRQs to have real-time behavior. The problem in this case is that all IRQs can then occur during the processing of an exception in the architectural completion layer, and so they are all effectively in the real-time layer. In other words, this effectively means that there are no interrupts in the non real-time layer.
 - All of the exception handlers in the architectural completion layer re-enable IRQs, but they also use additional VIC facilities to place a lower limit on the priority of IRQs that is taken. This permits IRQs at that priority or higher to be treated as being in the real-time layer, and IRQs at lower priorities to be treated as being in the non real-time layer. The price paid is some additional complexity in the software and in the VIC hardware.

———— **Note** —————

For either of the last two options, the new instructions speed up the IRQ re-enabling and the stack changes that are likely to be required.

2.10.11 Aborts

An abort can be caused by either:

- the MMU signalling an internal abort
- an external abort being raised from the AHB interfaces, by an AHB error response.

There are two types of abort:

- *Prefetch Abort* on page 2-36
- *Data Abort* on page 2-36.

IRQs are disabled when an abort occurs.

Prefetch Abort

This is signaled with the Instruction Data as it enters the pipeline Decode stage.

When a Prefetch Abort occurs, the ARM1136JF-S processor marks the prefetched instruction as invalid, but does not take the exception until the instruction is to be executed. 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

Data Abort on the ARM1136JF-S processor can be precise or imprecise. Precise Data Aborts are those generated after performing an instruction side CP15 operation, and all those generated by the MMU:

- alignment faults
- translation faults
- domain faults
- permission faults.

Data Aborts that occur because of watchpoints are imprecise in that the processor and system state presented to the abort handler is the processor and system state at the boundary of an instruction shortly after the instruction that caused the watchpoint (but before any following load/store instruction). Because the state that is presented is consistent with an instruction boundary, these aborts are restartable, even though they are imprecise.

Errors that cause externally generated Data Aborts, signaled by **HRESPR[0]**, **HRESPW[0]** or **HRESPP[0]**, might be precise or imprecise. Two separate FSR encodings indicate if the external abort is precise or imprecise.

External Data Aborts are precise if:

- all external aborts to loads when the CP15 Register 1 FI bit, bit 21, is set are precise
- all aborts to loads or stores to Strongly Ordered memory are precise
- all aborts to loads to the Program Counter or the CSPR are precise
- all aborts on the load part of a SWP are precise
- all other external aborts are imprecise.

External aborts are supported on cachable locations. The abort is transmitted to the processor only if a word requested by the processor had an external abort.

Precise Data Aborts

A precise Data Abort is signaled when the abort exception enables the processor and system state presented to the abort handler to be consistent with the processor and system state when the aborting instruction was executed. With precise Data Aborts, the restarting of the processor after the cause of the abort has been rectified is straightforward.

The ARM1136JF-S processor implements the *base restored Data Abort model*, which differs from the *base updated Data Abort model* implemented by the ARM7TDMI-S.

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 requirement for the Data Abort handler to unwind any base register update, which might have been specified by the aborted instruction. This simplifies the software Data Abort handler. See *ARM Architecture Reference Manual* for more details.

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

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

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

Imprecise Data Aborts

An imprecise Data Abort is signaled when the processor and system state presented to the abort handler cannot be guaranteed to be consistent with the processor and system state when the aborting instruction was issued.

2.10.12 Imprecise Data Abort mask in the CPSR/SPSR

An imprecise Data Abort caused, for example, by an External Error on a write that has been held in a Write Buffer, is asynchronous to the execution of the causing instruction and can occur many cycles after the instruction that caused the memory access has retired. For this reason, the imprecise Data Abort can occur at a time that the processor is in Abort mode because of a precise Data Abort, or can have live state in Abort mode, but be handling an interrupt.

To avoid the loss of the Abort mode state (r14 and SPSR_abt) in these cases, which leads to the processor entering an unrecoverable state, the existence of a pending imprecise Data Abort must be held by the system until a time when the Abort mode can safely be entered.

A mask is added into the CPSR to indicate that an imprecise Data Abort can be accepted. This bit is referred to as the A bit. The imprecise Data Abort causes a Data Abort to be taken when imprecise Data Aborts are not masked. When imprecise Data Aborts are masked, then the implementation is responsible for holding the presence of a pending imprecise Data Abort until the mask is cleared and the abort is taken.

The A bit is set automatically on entry into Abort Mode, IRQ, and FIQ Modes, and on Reset.

2.10.13 Software interrupt instruction

You can use the *software interrupt instruction* (SWI) to enter Supervisor mode, usually to request a particular supervisor function. The SWI handler reads the opcode to extract the SWI function number. 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.

IRQs are disabled when a software interrupt occurs.

2.10.14 Undefined instruction

When an instruction is encountered that neither the ARM1136JF-S processor, nor any coprocessor in the system, can handle the ARM1136JF-S processor 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, see the *ARM Architecture Reference Manual*.

2.10.15 Breakpoint instruction (BKPT)

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

A breakpoint instruction does not cause the ARM1136JF-S processor 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 halt mode, a breakpoint instruction causes the ARM1136JF-S processor to enter debug state. See *Halt mode debugging* on page 13-47.

2.10.16 Exception vectors

You can configure the location of the exception vector addresses by setting the V bit in CP15 c1 Control Register as shown in Table 2-6.

Table 2-6 Configuration of exception vector address locations

Value of V bit	Exception vector base location
0	0x00000000
1	0xFFFF0000

Table 2-7 shows the exception vector addresses and entry conditions for the different exception types.

Table 2-7 Exception vectors

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

2.10.17 Exception priorities

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

1. Reset (highest priority).
2. Precise Data Abort.
3. FIQ.
4. IRQ.
5. Prefetch Abort.
6. Imprecise Data Aborts.
7. 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 precise Data Abort occurs at the same time as an FIQ, the ARM1136JF-S processor 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.

Precise 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 if this happens.

Chapter 3

Control Coprocessor CP15

This chapter describes the ARM1136JF-S control coprocessor CP15 registers and how they are accessed. It also provides information for programming the microprocessor. It contains the following sections:

- *About control coprocessor CP15* on page 3-2
- *Accessing CP15 registers* on page 3-3
- *Summary of control coprocessor CP15 registers* on page 3-5.
- *CP15 registers arranged by function* on page 3-9
- *CP15 registers mapping* on page 3-12
- *Cache configuration and control* on page 3-15
- *Debug access to caches and TLB* on page 3-34
- *DMA control* on page 3-51
- *Memory management unit configuration and control* on page 3-65
- *TCM configuration and control* on page 3-83
- *System performance monitoring* on page 3-87
- *Overall system configuration and control* on page 3-93.

3.1 About control coprocessor CP15

The Control Coprocessor, CP15, implements a range of control functions and status information for the ARM1136JF-S processor. The main functions controlled by CP15 are:

- overall system control and configuration of the ARM1136JF-S processor
- cache configuration and management
- *Tightly-Coupled Memory* (TCM) configuration and management
- *Memory Management Unit* (MMU) configuration and management
- DMA control
- debug accesses to the caches and *Translation Lookaside Buffer* (TLB)
- system performance monitoring.

3.2 Accessing CP15 registers

You can access CP15 registers with MRC and MCR instructions. The instruction bit pattern of the MCR and MRC instructions is shown in Figure 3-1.

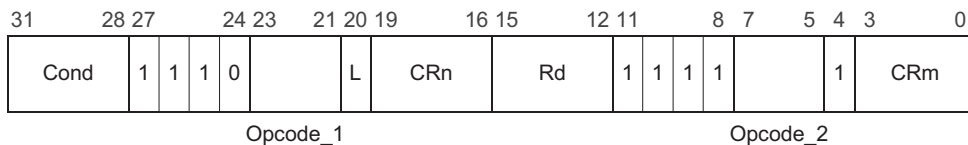


Figure 3-1 CP15 MRC and MCR bit pattern

The assembler for these instructions is:

```
MCR{cond} P15, <Opcode_1>, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

```
MRC{cond} P15, <Opcode_1>, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

Instructions CDP, LDC, and STC, together with unprivileged MRC and MCR instructions to privileged-only CP15 locations, cause the Undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and Opcode_2 fields specify a particular action when addressing registers. The L bit distinguishes between an MRC (L=1) and an MCR (L=0).

———— **Note** ————

Attempting to read from a nonreadable register, or to write to a nonwritable register causes Unpredictable results.

The Opcode_1, Opcode_2, and CRm fields Should Be Zero in all instructions that access CP15, except when the values specified are used to select desired operations. Using other values results in Unpredictable behavior.

In all cases, reading from or writing any data values to any CP15 registers, including those fields specified as Unpredictable, Should Be One, or Should Be Zero, does not cause any physical damage to the chip.

Table 3-1 shows the terms and abbreviations used throughout this chapter.

Table 3-1 CP15 abbreviations

Term	Abbreviation	Description
Unpredictable	UNP	For reads: The data returned when reading from this location is unpredictable. It can have any value. For writes: Writing to this location causes unpredictable behavior, or an unpredictable change in device configuration.
Undefined	UND	An instruction that accesses CP15 in the manner indicated takes the Undefined instruction trap.
Should Be Zero	SBZ	When writing to this location, all bits of this field should be 0.
Should Be One	SBO	When writing to this location, all bits in this field should be 1.

3.3 Summary of control coprocessor CP15 registers

Table 3-2 lists the registers described in this section.

Table 3-2 Summary of control coprocessor (CP15) register

Names	Type	Reset value	Description
Auxiliary Control	Read/write	0x00000007	See <i>Auxiliary Control Register</i> on page 3-93
Cache Debug Control	Read/write	0x00000000	See <i>Cache Debug Control Register</i> on page 3-34
Cache Operations	Read/write	-	See <i>Cache Operations Register</i> on page 3-17
Cache Type	Read-only	Implementation defined ^a	See <i>Cache Type Register</i> on page 3-28
Context ID	Read/write	0x00000000	See <i>Context ID Register</i> on page 3-95
Control	Read/write	0x000500F8 ^b	See <i>Control Register</i> on page 3-96
Coprocessor Access Control	Read/write	0x00000000	See <i>Coprocessor Access Control Register</i> on page 3-94
Count 0 (PMN0)	Read/write	0x00000000	See <i>Count Register 0, PMN0</i> on page 3-91
Count 1 (PMN1)	Read/write	0x00000000	See <i>Count Register 1, PMN1</i> on page 3-91
Cycle Counter (CCNT)	Read/write	Unpredictable	See <i>Cycle Counter Register, CCNT</i> on page 3-92
Data Cache Lockdown	Read/write	0xFFFFFFF0	See <i>Cache Lockdown Registers</i> on page 3-15
Data Cache Master Valid	Read/write	0x00000000	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37
Data Debug Cache	Read-only	0x00000000	See <i>Cache debug operations</i> on page 3-34
Data Fault Status	Read-only	0x00000000	See <i>Data Fault Status Register</i> on page 3-66
Data Memory Remap	Read/write	0x01C97CC8	See <i>Memory Region Remap Registers</i> on page 3-69
Data MicroTLB Attribute	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Data MicroTLB Entry	Write-only	-	See <i>MMU debug operations</i> on page 3-38
Data MicroTLB PA	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Data MicroTLB VA	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Data SmartCache Master Valid	Read/write	0x00000000	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37

Table 3-2 Summary of control coprocessor (CP15) register (continued)

Names	Type	Reset value	Description
Data Tag RAM Read Operation	Write-only	-	See <i>Cache debug operations</i> on page 3-34
Data TCM Region	Read/write	Implementation defined ^c	See <i>Data TCM Region Register</i> on page 3-83
DMA Channel Number	Read/write	0x00000000	See <i>DMA Channel Number Register</i> on page 3-53
DMA Channel Status	Read-only	0x00000000	See <i>DMA Channel Status Registers</i> on page 3-53
DMA Context ID	Read/write	0x00000000	See <i>DMA Context ID Registers</i> on page 3-55
DMA Control	Read/write	0x00000000	See <i>DMA Control Register</i> on page 3-56
DMA Enable	Write-only	-	See <i>DMA Enable Register</i> on page 3-59
DMA External Start Address	Read/write	0x00000000	See <i>DMA External Start Address Registers</i> on page 3-61
DMA Identification and Status	Read-only	0x00000000 0x00000001	See <i>DMA Identification and Status Registers</i> on page 3-61
DMA Internal End Address	Read/write	0x00000000	See <i>DMA Internal End Address Register</i> on page 3-63
DMA Internal Start Address	Read/write	0x00000000	See <i>DMA Internal Start Address Register</i> on page 3-63
DMA Memory Remap	Read/write	0x01C97CC8	See <i>Memory Region Remap Registers</i> on page 3-69
DMA User Accessibility	Read/write	0x00000000	See <i>DMA User Accessibility Register</i> on page 3-64
Domain Access Control	Read/write	0x00000000	See <i>Domain Access Control Register</i> on page 3-67
Data Fault Address	Read-only	0x00000000	See <i>Fault Address Register</i> on page 3-65
FCSE PID	Read/write	0x00000000	See <i>FCSE PID Register</i> on page 3-100
ID Code	Read-only	0x4107B360	See <i>ID Code Register</i> on page 3-102
Instruction Cache Data RAM Read Operation	Write-only	-	See <i>Cache debug operations</i> on page 3-34
Instruction Cache Lockdown	Read/write	0xFFFFFFFF	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37
Instruction Cache Master Valid	Read/write	0x00000000	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37

Table 3-2 Summary of control coprocessor (CP15) register (continued)

Names	Type	Reset value	Description
Instruction Debug Cache	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Instruction Fault Address	Read/write	0x00000000	See <i>Instruction Fault Address Register</i> on page 3-67
Instruction Fault Status	Read-only	0x00000000	See <i>Instruction Fault Status Register</i> on page 3-68
Instruction Memory Remap	Read/write	0X01C97CC8	See <i>Memory Region Remap Registers</i> on page 3-69
Instruction MicroTLB Attribute	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Instruction MicroTLB Entry	Write-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Instruction MicroTLB PA	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Instruction MicroTLB VA	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Instruction SmartCache Master Valid	Read/write	0x00000000	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37
Instruction Tag RAM Read Operation	Write-only	-	See <i>Cache debug operations</i> on page 3-34
Instruction TCM Region	Read/write	Implementation defined ^c	See <i>Instruction TCM Region Register</i> on page 3-85
Main TLB Attribute	Read/write	0x00000000	See <i>MMU debug operations</i> on page 3-38
Main TLB Entry	Read-only	0x00000000	See <i>MMU debug operations</i> on page 3-38
Main TLB Master Valid	Read/write	0x00000000	See <i>Cache and main TLB Master Valid Registers</i> on page 3-37
Main TLB PA	Read/write	0x00000000	See <i>MMU debug operations</i> on page 3-38
Main TLB VA	Read/write	0x00000000	See <i>MMU debug operations</i> on page 3-38
Performance Monitor Control	Read/write	0x00000000	See <i>Performance Monitor Control Register (PMNC)</i> on page 3-87
Peripheral Port Memory Remap	Read/write	0x01c97cc8	See <i>Memory Region Remap Registers</i> on page 3-69
TCM Status	Read-only	0x00010001	See <i>TCM Status Register</i> on page 3-83
TLB Debug Control	Read/write	0x00000000	See <i>MMU debug operations</i> on page 3-38
TLB Lockdown	Read/write	0x00000000	See <i>TLB Lockdown Register</i> on page 3-77

Table 3-2 Summary of control coprocessor (CP15) register (continued)

Names	Type	Reset value	Description
TLB Operations	Read-only	-	See <i>TLB Operations Register</i> on page 3-75
TLB Type	Read-only	0x00080800	See <i>TLB Type Register</i> on page 3-74
Translation Table Base Control	-	0x00000000	See <i>Translation Table Base Control Register</i> on page 3-79
Translation Table Base 0	-	0x00000000	See <i>Translation Table Base Register 0</i> on page 3-80
Translation Table Base 1	-	0x00000000	See <i>Translation Table Base Register 1</i> on page 3-81

- a. The cache type reset value is determined by the size of the caches implemented.
- b. Bits 25, 22, and 7 depend on the value of macrocell input signals **BIGENDINIT** and **UBITINIT**. See Table 3-59 on page 3-97.
- c. The cache type reset value is determined by the size of the caches implemented.

3.4 CP15 registers arranged by function

The CP15 system control registers control the system functions shown in Table 3-3.

Table 3-3 CP15 register functions

Register/s	Function
Data Cache Lockdown Register	See <i>Cache configuration and control</i> on page 3-15
Instruction Cache Lockdown Register	
Cache Operations Register	
Cache Type Register	
Cache Dirty Status Register	
Cache Debug Control Register	See <i>Debug access to caches and TLB</i> on page 3-34
Data Tag RAM Read Operation	
Data Cache Master Valid Register	
Instruction Cache Data RAM Read Operation	
Instruction and Data Debug Cache Registers	
Instruction Cache Master Valid Register	
Data SmartCache Master Valid Register	
Instruction SmartCache Master Valid Register	
Main TLB Master Valid Register	
Data MicroTLB Attribute Register	
Data MicroTLB Entry Operation	
Data MicroTLB PA Register	
Data MicroTLB VA Register	
Instruction Cache Data RAM Register	
Instruction MicroTLB Attribute Register	
Instruction MicroTLB Entry Operation	
Instruction MicroTLB PA Register	
Instruction MicroTLB VA Register	
Instruction Tag RAM Read Operation	
Main TLB Attribute Register	
Main TLB Entry Register	
Main TLB PA Register	
Main TLB VA Register	
TLB Debug Control Register	

Table 3-3 CP15 register functions (continued)

Register/s	Function
DMA registers	See <i>DMA control</i> on page 3-51
DMA Channel Number Register	
DMA Channel Status Registers	
DMA Context ID Registers	
DMA Control Registers	
DMA Enable Register	
DMA External Start Address Registers	
DMA Identification and Status Registers	
DMA Internal End Address Register	
DMA Internal Start Address Register	
DMA User Accessibility Register	
Fault Address Register	See <i>Memory management unit configuration and control</i> on page 3-65
Data Fault Status Register	
Instruction Fault Address Register	
Instruction Fault Status Register	
Instruction TCM Region Register	
Memory Region Remap Registers	
Main TLB Master Valid Register	
Peripheral Port Memory Remap Register	
Translation Table Base Control Register	
Translation Table Base Register 0	
Translation Table Base Register 1	
TCM Status Register	See <i>TCM configuration and control</i> on page 3-83
Data TCM Region Register	
Instruction TCM Region Register	
Domain Access Control Register	
Performance Monitor Control Register (PMNC)	See <i>System performance monitoring</i> on page 3-87
Count Register 0 (PMN0)	
Count Register 1 (PMN1)	
Cycle Counter Register (CCNT)	
Coprocessor Access Control Register	

Table 3-3 CP15 register functions (continued)

Register/s	Function
Control Register	See <i>Overall system configuration and control</i> on page 3-93
Auxiliary Control Register	
Coprocessor Access Control Register	
Context ID Register	
FCSE PID Register	
ID Code Register	

3.5 CP15 registers mapping

CP15 defines 16 registers, c0-c15, that are used to perform system control functions. Several of these register numbers provide access to more than one register.

Figure 3-2 to Figure 3-4 on page 3-14 show how the CP15 system control registers are mapped into c0-c15.

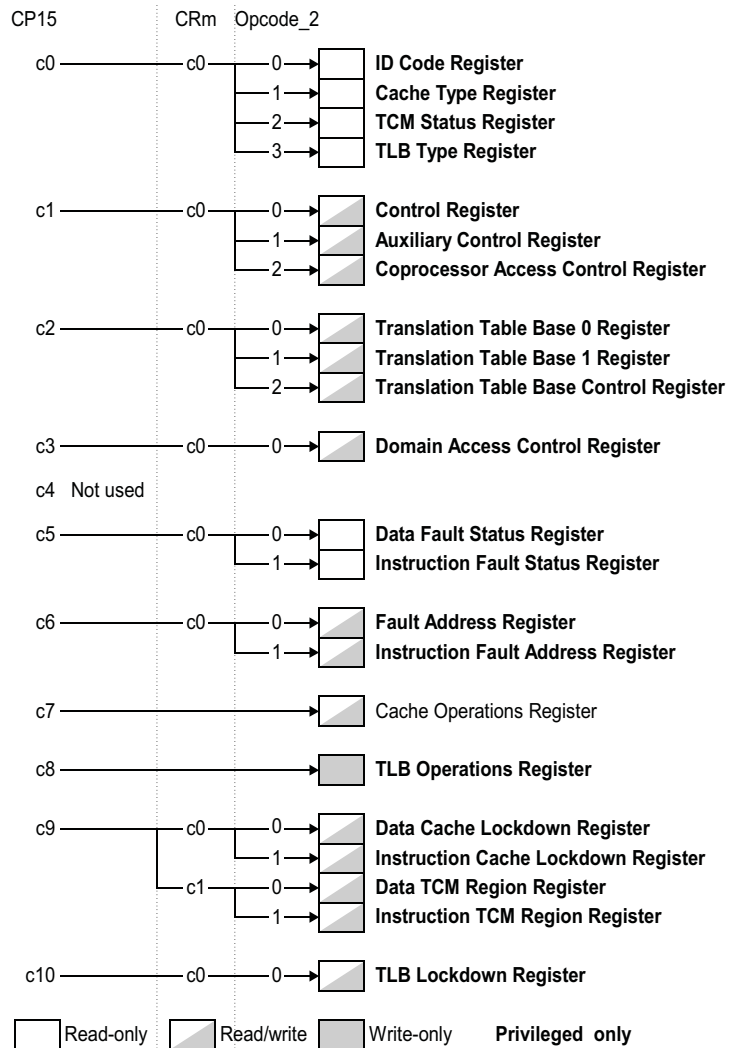


Figure 3-2 CP15 register map, part one

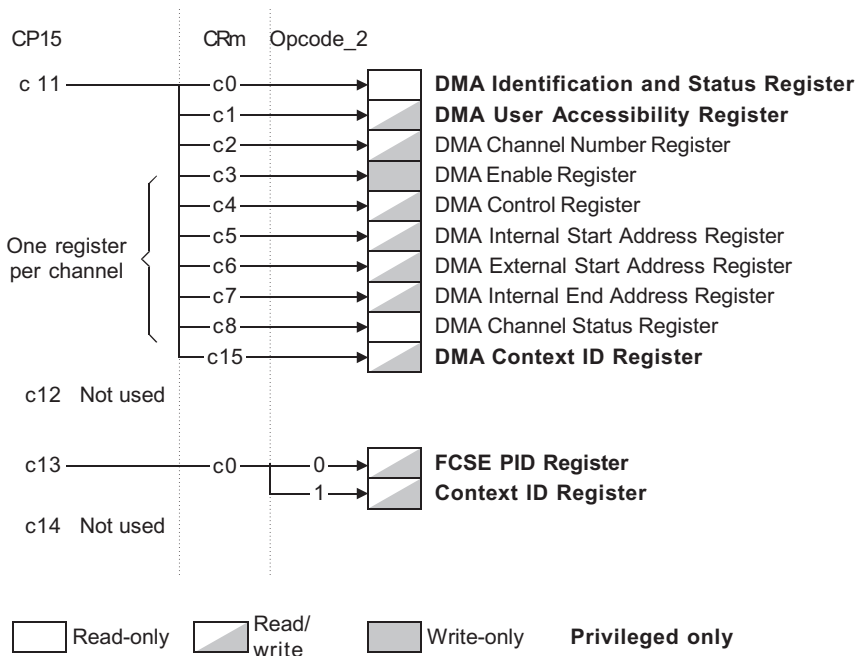


Figure 3-3 CP15 register map, part two

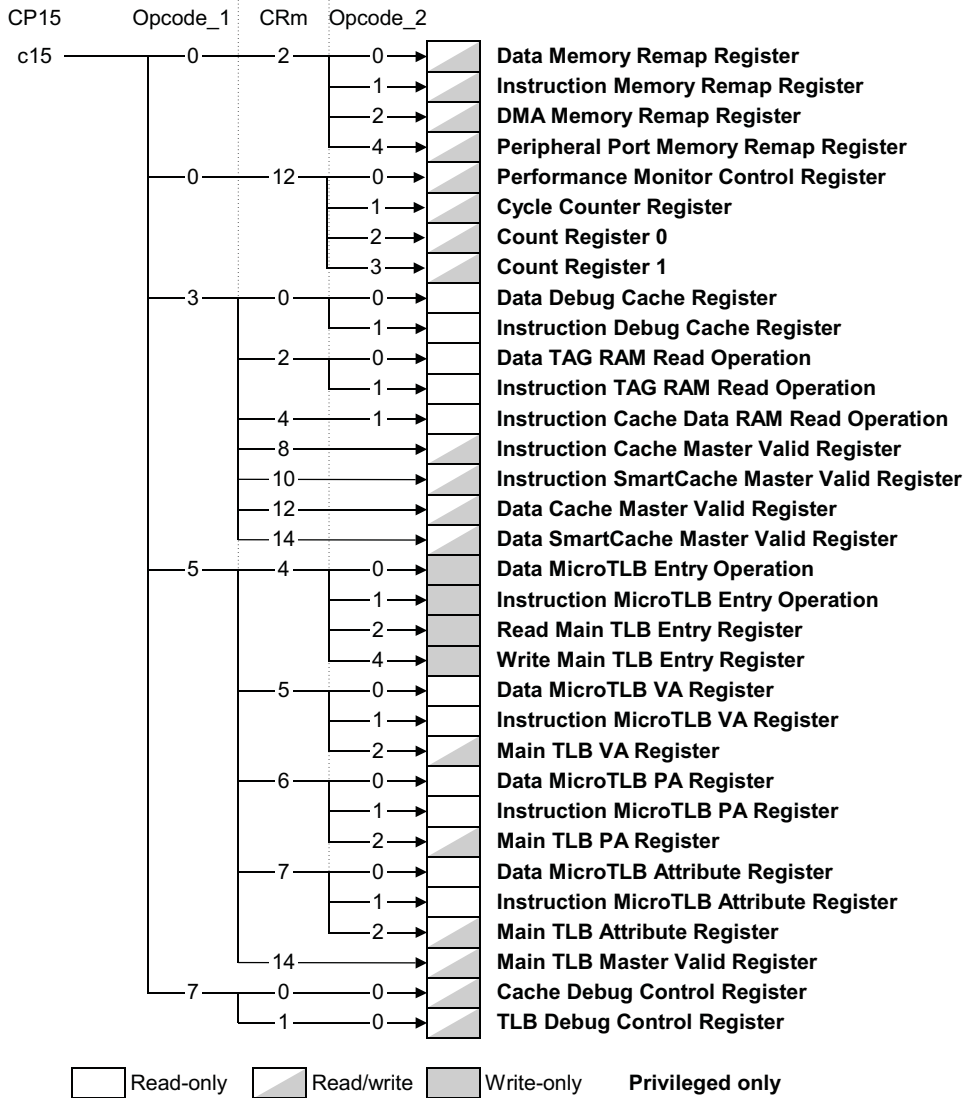


Figure 3-4 CP15 register map, part three

3.6 Cache configuration and control

The ARM1136JF-S Cache configuration and control is implemented using the following registers:

- *Cache Lockdown Registers*
- *Cache Operations Register* on page 3-17
- *Cache Type Register* on page 3-28
- *Cache Dirty Status Register* on page 3-33.

3.6.1 Cache Lockdown Registers

There are two Cache Lockdown Registers:

- Data Cache Lockdown Register
- Instruction Cache Lockdown Register.

You can access the Data Cache Lockdown Registers by reading or writing CP15 c9 with the CRm field set to c0 and the Opcode_2 field set to 0. For example:

```
MRC p15, 0, <Rd>, c9, c0, 0 ; Read Data Cache Lockdown Register
MCR p15, 0, <Rd>, c9, c0, 0 ; Write Data Cache Lockdown Register
```

You can access the Instruction Cache Lockdown Register by reading or writing CP15 c9 with the CRm field set to c0 and the Opcode_2 field set to 1. For example:

```
MRC p15, 0, Rn, c9, c0, 1 ; Read Instruction Cache Lockdown Register
MCR p15, 0, Rn, c9, c0, 1 ; Write Instruction Cache Lockdown Register
```

ARM1136JF-S processors only supports one method of using cache lockdown registers, called Format C. This method is a cache way based locking scheme. It enables you to lockdown each cache way independently. This gives you some control over cache pollution caused by particular applications, in addition to providing a traditional lockdown function for locking critical regions into the cache.

A locking bit for each cache way determines if the normal cache allocation mechanisms (Random or Round-Robin) are able to access that cache way.

ARM1136JF-S processors have an associativity of 4. If all ways are locked, the ARM1136JF-S processor behaves as if only ways 3 to 1 are locked and way 0 is unlocked.

The format of the ARM1136JF-S processor Instruction and Data Cache Lockdown Registers is shown in Figure 3-5 on page 3-16.



Figure 3-5 Instruction and Data Cache Lockdown Registers format

The L bits for cache ways 3 to 0 are bits [3:0] respectively. If a cache way is not implemented, then the L bit for that way is hardwired to 1, and writes to that bit are ignored.

L = 0 Allocation to the cache way is determined by the standard replacement algorithm (reset state).

L = 1 No allocation is performed to this cache way.

A Cache Lockdown Register must only be changed when it is certain that all outstanding accesses that might cause a cache line fill have completed. For this reason, a Drain Write Buffer instruction must be executed before the Cache Lockdown Register is changed.

The following procedure for lock down into a data or instruction cache way *i*, with *N* cache ways, using Format C, ensures that only the target cache way *i* is locked down.

This is the architecturally defined method for locking data into caches:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts. If this is not possible, all code and data used by any exception handlers that can be called must be treated as code and data prior to step 2.
2. Ensure that all data used by the following code, apart from the data that is to be locked down, is either:
 - in an uncachable area of memory, including the TCM
 - in an already locked cache way.
3. Ensure that the data to be locked down is in a Cachable area of memory.
4. Ensure that the data to be locked down is not already in the cache, using cache Clean and/or Invalidate instructions as appropriate.
5. Enable allocation to the target cache way by writing to CP15 c9, with the CRm field set to 0, setting L equal to 0 for bit *i* and L equal to 1 for all other ways.

6. Ensure that the memory cache line is loaded into the cache by using an LDR instruction to load a word from the memory cache line, for each of the cache lines to be locked down in cache way *i*.
7. Write to CP15 c9, CRm = c0, setting L to 1 for bit *i* and restore all the other bits to the values they had before this routine was started.

3.6.2 Cache Operations Register

You can use the Cache Operations Register to control the Instruction and Data Caches, and the Write Buffer. You can also use it to implement similar functions on prefetch buffers and branch target caches, if they exist, and to implement the Wait For Interrupt clock control function.

You can also use CP15 c7 to perform block transfer operations, see *Block transfer operations using CP15 c7* on page 3-25.

You can use the following instruction to write to the Cache Operations Register, CP15 c7:

```
MCR p15,0, <Rd>, c7, <CRm>, <Opcode_2>
```

The function of each cache operation is selected by the Opcode_2 and CRm fields in the MCR instruction used to write CP15 c7.

The functions that you can perform using CP15 c7 are shown in Table 3-4 on page 3-19.

Writing the Cache Operations Register with a combination of CRm and Opcode_2 not listed in Table 3-4 on page 3-19 gives Unpredictable results.

In the ARM1136JF-S processor, reading from the Cache Operations Register, except for reads from the Cache Dirty Status Register or the Block Transfer Status Register, causes an Undefined instruction trap.

If Opcode_1 = 0, these instructions are applied to a level one cache system. All other Opcode_1 values are reserved.

All CP15 c7 operations can only be executed in a privileged mode of operation, except Drain Write Buffer, Flush Prefetch Buffer, and Data Memory Barrier. These can be operated in User mode. Attempting to execute a privileged instruction in User mode results in the Undefined instruction trap being taken.

The following definitions apply to Table 3-4 on page 3-19:

Clean Applies to Write-Back Data Caches. This means that if the cache line contains stored data that has not yet been written out to main memory, it is written to main memory now, and the line is marked as clean.

Invalidate This means that the cache line (or all the lines in the cache) is marked as invalid, so that no cache hits occur for that line until it is re-allocated to an address. For Write-Back Data Caches, this does not include cleaning the cache line unless that is also stated.

Prefetch This means the memory cache line at the specified virtual address is loaded into the cache. There is no alignment requirement for the virtual address.

Drain Write Buffer

This instruction acts as an explicit memory barrier. This instruction completes when all explicit memory transactions occurring in program order before this instruction are completed. No instructions occurring in program order after this instruction are executed until this instruction completes. Therefore, no explicit memory transactions occurring in program order after this instruction are started until this instruction completes. See *Explicit Memory Barriers* on page 6-24.

It can be used instead of Strongly Ordered memory when the timing of specific stores to the memory system needs to be controlled. For example, when a store to an interrupt acknowledge location must be completed before interrupts are enabled.

Drain Write Buffer can be executed in both privileged and User modes of operation.

Wait For Interrupt

This puts the processor into a low-power state and stops it executing more instructions until an interrupt (or debug) request occurs, regardless of whether the interrupts are disabled by the masks in the CPSR. When an interrupt does occur, the MCR instruction completes and the IRQ or FIQ handler is entered as normal. The return link in r14_irq or r14_fiq contains the address of the MCR instruction plus 8, so that the normal instruction used for interrupt return (SUBS PC, R14, #4) returns to the instruction following the MCR.

Flush Prefetch Buffer

Flushing the instruction prefetch buffer has the effect that all instructions occurring in program order after this instruction are fetched from the memory system after the execution of this instruction, including the level one cache or TCM. This operation is useful for ensuring the correct execution of self-modifying code. See *Explicit Memory Barriers* on page 6-24.

Data This is the value that is written to CP15 c7. This is the value in the register <Rd> specified in the MCR instruction.

If the data is stated to be a virtual address, it does not have to be cache line aligned. This address is looked up in the cache for the particular operation. Invalidation and cleaning operations have no effect if they miss in the cache. If the corresponding entry is not in the TLB, these instructions can cause a TLB miss exception or hardware page table walk, depending on the miss handling mechanism.

For the cache control operations, the virtual addresses that are passed to the cache are not translated by the FCSE extension.

If the data is stated to be set/Index format (see Figure 3-7 on page 3-22), it identifies the cache line that the operation is to be applied to by specifying which cache set it belongs to and what its Index is within the set. The Index corresponds to the number of the cache way, and the set number corresponds to the line number within a cache way.

Table 3-4 lists the cache operation functions and the associated data and instruction formats for CP15 c7.

Table 3-4 Cache Operations Register functions

Function	Data	Instruction
Wait For Interrupt.	SBZ	MCR p15, 0, <Rd>, c7, c0, 4
Invalidate Entire Instruction Cache. Also flushes the branch target cache.	SBZ	MCR p15, 0, <Rd>, c7, c5, 0
Invalidate Instruction Cache Line (using MVA).	MVA	MCR p15, 0, <Rd>, c7, c5, 1
Invalidate Instruction Cache Line (using Index).	Set/Index	MCR p15, 0, <Rd>, c7, c5, 2
Flush Prefetch Buffer ^a .	SBZ	MCR p15, 0, <Rd>, c7, c5, 4
Flush Entire Branch Target Cache.	SBZ	MCR p15, 0, <Rd>, c7, c5, 6
Flush Branch Target Cache Entry.	MVA ^b	MCR p15, 0, <Rd>, c7, c5, 7
Invalidate Entire Data Cache.	SBZ	MCR p15, 0, <Rd>, c7, c6, 0
Invalidate Data Cache Line (using MVA).	MVA	MCR p15, 0, <Rd>, c7, c6, 1
Invalidate Data Cache Line (using Index).	Set/Index	MCR p15, 0, <Rd>, c7, c6, 2
Invalidate Both Caches. Also flushes the branch target cache.	SBZ	MCR p15, 0, <Rd>, c7, c7, 0

Table 3-4 Cache Operations Register functions (continued)

Function	Data	Instruction
Clean Entire Data Cache.	SBZ	MCR p15, 0, <Rd>, c7, c10, 0
Clean Data Cache Line (using MVA).	MVA	MCR p15, 0, <Rd>, c7, c10, 1
Clean Data Cache Line (using Index).	Set/Index	MCR p15, 0, <Rd>, c7, c10, 2
Drain Write Buffer ^a .	SBZ	MCR p15, 0, <Rd>, c7, c10, 4
Data Memory Barrier ^a .	SBZ	MCR p15, 0, <Rd>, c7, c10, 5
Read Cache Dirty Status Register.	Data	MRC p15, 0, <Rd>, c7, c10, 6
Prefetch Instruction Cache Line.	MVA	MCR p15, 0, <Rd>, c7, c13, 1
Clean and Invalidate Entire Data Cache.	SBZ	MCR p15, 0, <Rd>, c7, c14, 0
Clean and Invalidate Data Cache Line (using MVA).	MVA	MCR p15, 0, <Rd>, c7, c14, 1
Clean and Invalidate Data Cache Line (using Index).	Set/Index	MCR p15, 0, <Rd>, c7, c14, 2

- a. These operations are accessible in both User and privileged modes of operation. All other operations are only accessible in privileged modes of operation.
- b. The range of MVA bits used in this function is different to the range of bits used in other functions that have MVA data.

The cache invalidation operations apply to all cache locations, including those locked in the cache. An explicit flush of the relevant lines in the branch target cache must be performed after invalidation of Instruction Cache lines or the results are Unpredictable. This is not required after an entire Instruction Cache invalidation.

Figure 3-6 on page 3-21 shows the functions and registers that you can access using MCR and MRC instructions with CP15 c7, Cache Operations Control Register. For details of the functions that you can access using MCRR and MCRR2 instructions, see *Enhanced cache control operations using MCRR and MCRR2 instructions* on page 3-26.

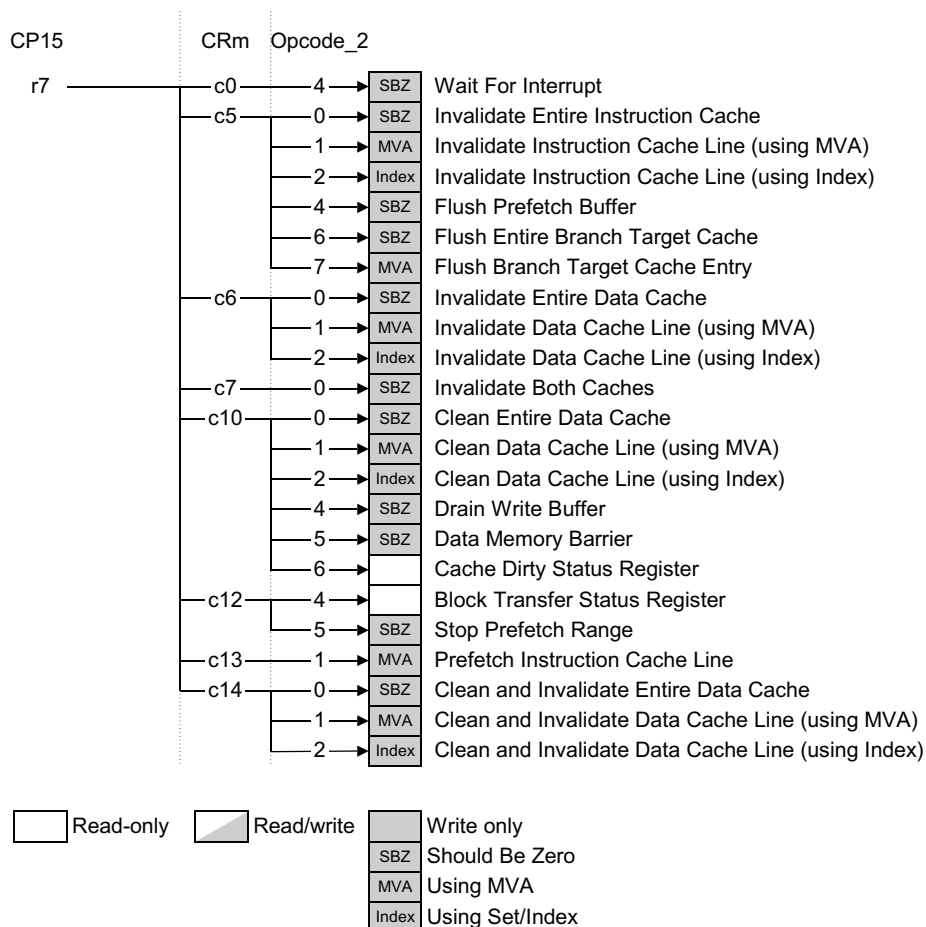


Figure 3-6 Accessing the Cache Operations Register

The operations that act on a single cache line identify the line using the contents of <Rd> as the address, passed in the MCR instruction. The data is interpreted using:

- *Set/Index format*
- *Modified Virtual Address (MVA) format* on page 3-23.

Set/Index format

The Index tag format shown in Figure 3-7 on page 3-22 is used when a specific line in the cache has to be accessed.

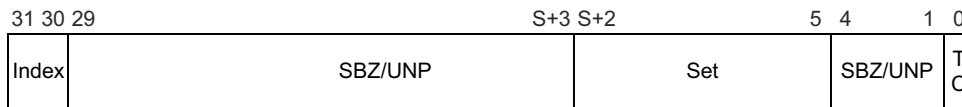
**Figure 3-7 Register 7 Set/Index format**

Table 3-5 shows the bit fields for Index operations using CP15 c7, and their meanings.

Table 3-5 Bit fields for Set/Index operations using CP15 c7

Bits	Name	Description
[31:30]	Index	Index in set being accessed
[29:S+3]	-	SBZ/UNP
[S+2:5]	Set	Set being accessed
[4:1]	-	SBZ/UNP
[1]	TC	0 = Cache operation 1 = TCM operation

In Table 3-5 and Figure 3-7, S is the logarithm to the base 2 of the Size parameter. This parameter is in the Cache Type Register, CP15 c0, see *Cache Type Register* on page 3-28. Example 3-1 is an example using the command Clean Data Cache Line (using Index).

Example 3-1 Clean Data Cache Line (using Index)

```

;code is specific to ARM1136JF-S with 32KB caches
MOV R0, #0:SHL:5
seg_loop
MOV R1, #0:SHL:26
line_loop
ORR R2,R1,R0
MCR p15,0,R2,c7,c10,2
ADD R1,R1,#1:SHL:26
CMP R1,#0
BNE line_loop
ADD R0,R0,#1:SHL:5
CMP R0,#1:SHL:9
BNE seg_loop

```

Modified Virtual Address (MVA) format

The MVA format is useful for flushing a particular address or range of addresses in the caches. Figure 3-8 shows the MVA format for the Cache Operations Register functions:

- Invalidate Instruction Cache Line
- Invalidate Data Cache Line
- Clean Data Cache Line
- Prefetch Instruction Cache Line
- Clean and Invalidate Data Cache Line.

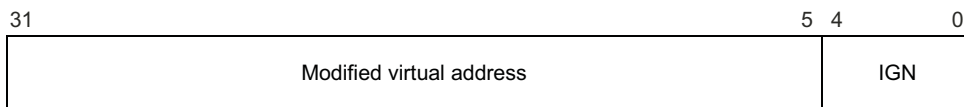


Figure 3-8 CP15 Register c7 MVA format

Bits 0 - 4 are ignored.

Figure 3-9 shows the MVA format for the Cache Operations Register Flush Branch Target Cache Entry function.

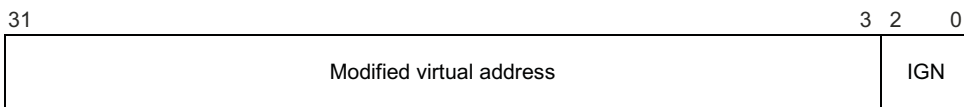


Figure 3-9 CP15 c7 MVA format for Flush Branch Target Cache Entry function

Bits 0 - 2 are ignored.

Cache cleaning and invalidating operations for TCM configured as SmartCache

All cache line and block cleaning and invalidation operations based on virtual address, as defined in CP15 c7, include TCM regions that are configured as SmartCache.

The Set/Index operations are supported for the TCMs operating as SmartCache. In this case, the Index number is taken to be the TCM number, and the meaning of the set number is unchanged. To distinguish between these operations as applied to the Cache and as applied to TCM, the bottom bit of the Set/Index is used, as shown in Figure 3-7 on page 3-22.

The line length of the TCM operating as SmartCache must be the same as the cache line length, defined in the Cache Type Register.

The TC bit, bit 0, indicates if this register is referring to the TCMs rather than the Cache:

TC = 0 Register refers to the cache.

TC = 1 Register refers to the TCM.

Invalidate and Clean Entire Cache operations do not affect the TCMs.

Clean, and Clean and Invalidate, Entire Data Cache operations

CP15 c7 specifies operations for cleaning the entire Data Cache, and also for performing a clean and invalidate of the entire Data Cache. These are blocking operations that can be interrupted. If they are interrupted, the r14 value that is captured on the interrupt is the address of the instruction that launched the cache clean operation + 4. This enables the standard return mechanism for interrupts to restart the operation.

If it is essential that the cache is clean (or clean and invalid) for a particular operation, the sequence of instructions for cleaning (or cleaning and invalidating) the cache for that operation must handle the arrival of an interrupt at any time in which interrupts are not disabled. This is because interrupts can write to a previously clean cache. For this reason, the Cache Dirty Status Register indicates if the cache has been written to since the last clean of the cache was started. This register can be interrogated to determine if the cache is clean, and if this is done while interrupts are disabled, the following operation(s) can rely on having a clean cache. The following sequence shows this approach:

```

; interrupts are assumed to be enabled at this point
Loop1  MOV R1, #0
       MCR CP15, 0, R1, C7, C10, 0      ; Clean (or Clean & Invalidate) Cache
       MRS R2, CPSR
       CPSID iaf                        ; Disable interrupts
       MRC CP15, 0, R1, C7, C10, 6     ; Read Cache Dirty Status Register
       ANDS R1, R1, #01                 ; Check if it is clean
       BEQ UseClean
       MSR CPSR, R2                     ; Re-enable interrupts
       B Loop1                          ; - clean the cache again
UseCleanDo_Clean_Operations            ; Perform whatever operation relies on
; the cache being clean/invalid.
; To reduce impact on interrupt
; latency, this sequence should be
; short
       MSR CPSR, R2                     ; Re-enable interrupts

```

The long Cache clean operation is performed with interrupts enabled throughout this routine.

The Clean Entire Data Cache operation and Clean and Invalidate Entire Data Cache operation have no effect on TCMs operating as SmartCache.

The format of Cache Dirty Status register is shown in Figure 3-10.



Figure 3-10 Cache Dirty Status Register format

If the C bit is 0, no write has hit the cache since the last cache clean or reset successfully left the cache clean.

If the C bit is 1, the cache might contain dirty data.

The instructions that you can use to access the Cache Dirty Status Register are shown in Table 3-4 on page 3-19.

Block transfer operations using CP15 c7

The block operations shown in Table 3-6 are supported using CP15 c7.

Table 3-6 Block transfer operations

Operation	Blocking?	Instruction or data	User or privileged	Exception behavior
Prefetch Range	Nonblocking	Instruction or data	User or privileged	None
Clean Range	Blocking	Data	User or privileged	Data Abort
Clean and Invalidate Range	Blocking	Data only	Privileged	Data Abort
Invalidate Range	Blocking	Instruction or data	Privileged	Data Abort

Each of the range operations is started using an MCRR operation, with the data of the two registers being used to specify the Block Start Address and the Block End Address. All block operations are performed on the cache, or SmartCache, lines that include the range of addresses between the Block Start Address and Block End Address inclusive. If the Block Start Address is greater than the Block End Address the effect is architecturally Unpredictable. The ARM1136JF-S processor does not perform cache operations.

Only one block transfer at a time is supported. Attempting to start a second block transfer while a first nonblocking block transfer is in progress causes the first block transfer to be abandoned and the second block transfer to be started. The Block Transfer Status Register indicates if a block transfer is in progress. Block transfers must be stopped on a context switch.

All block transfers are interruptible. When blocking transfers are interrupted, the r14 value that is captured is the address of the instruction that launched the block operation + 4. This enables the standard return mechanism for interrupts to restart the operation.

ARM1136JF-S processors enable following instructions to be executed while a nonblocking Prefetch Range instruction is being executed. The r14 value captured on an interrupt is determined by the execution state presented to the interrupt in following instruction stream.

If the FCSE PID is changed while a Prefetch Range operation is running, it is Unpredictable at which point this change is seen by the Prefetch Range.

Exception behavior

The blocking block transfers cause a Data Abort on a translation fault if a valid page table entry cannot be fetched. The FAR indicates the address that caused the fault, and the DFSR indicates the reason for the fault.

Any fault on a Prefetch Range operation results in the operation failing without signaling an error.

Enhanced cache control operations using MCRR and MCRR2 instructions

The list of CP15 c7 instructions shown in Table 3-4 on page 3-19 is augmented with additional operations shown in Table 3-7. These operations can only be performed using an MCRR or MCRR2 instruction, and all other operations to these registers are ignored.

Table 3-7 Enhanced cache control operations

Function	Instruction
Invalidate Instruction Cache Range	MCRR p15,0,<End Address>,<Start Address>,5
Invalidate Data Cache Range	MCRR p15,0,<End Address>,<Start Address>,6
Clean Data Cache Range ^a	MCRR p15,0,<End Address>,<Start Address>,12

Table 3-7 Enhanced cache control operations (continued)

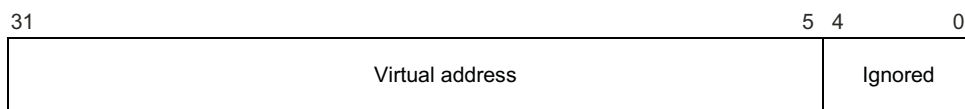
Function	Instruction
Prefetch Instruction Cache Range ^a	MCRR p15,1,<End Address>,<Start Address>,12
Prefetch Data Cache Range ^a	MCRR p15,2,<End Address>,<Start Address>,12
Clean and Invalidate Data Cache Range	MCRR p15,0,<End Address>,<Start Address>,14

a. These operations are accessible in both User and privileged modes of operation (see *User access to CP15 c7 operations* on page 3-28). All other operations listed here are only accessible in privileged modes of operation.

The <End Address> and <Start Address> in Table 3-7 on page 3-26 is the true Virtual Address before any modification by the *Fast Context Switch Extension* (FCSE). This address is translated by the FCSE logic.

Each of the Range operations operates between cache, or SmartCache, lines containing the <Start Address> and the <End Address>, inclusive of <Start Address> and <End Address>.

The <Start Address> and <End Address> data values passed by the MCRR instructions described in Table 3-7 on page 3-26 have the format shown in Figure 3-11.

**Figure 3-11 Block Address Register format**

Because the least significant address bits are ignored, the transfer automatically adjusts to a line length multiple spanning the programmed addresses.

The <Start Address> is the first virtual address of the block transfer. It uses the Virtual Address bits [31:5].

The <End Address> is the virtual address where the block transfer stops. This address is at the start of the line containing the last address to be handled by the block transfer. It uses the Virtual Address bits [31:5].

You can stop a Prefetch Range operation by performing either:

- A stop Prefetch Range operation. This is a CP15 c7 MCR or MCR2 operation as shown in Table 3-8. This operation is accessible in both User and privileged modes of operation (see *User access to CP15 c7 operations*).

- Another block operation. See *Block transfer operations using CP15 c7* on page 3-25.

Also, you can determine the status of an Instruction or Data Prefetch an MRC as shown in Table 3-8.

Table 3-8 CP15 Register c7 block transfer MCR/MRC operations

Function	Data	Instruction
Stop Prefetch Range ^a	SBZ	MCR p15, 0, <Rd>, c7, c12, 5
Read Block Transfer Status Register (read-only) ^a	Data	MRC p15, 0, <Rd>, c7, c12, 4

a. These operations are accessible in both User and privileged modes of operation (see *User access to CP15 c7 operations*).

The Block Transfer Status Register has the format shown in Figure 3-12.

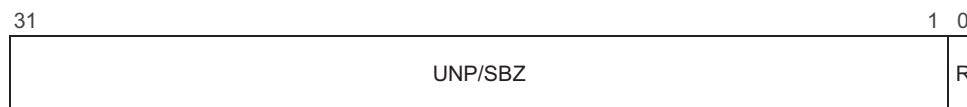


Figure 3-12 Block Transfer Status Register format

If the R bit is 0, there is no block prefetch in operation. If the R bit is 1 there is a prefetch in operation.

User access to CP15 c7 operations

A small number of CP15 c7 operations can be executed by code while in User mode. Attempting to execute a privileged operation in User mode using CP15 c7 results in an Undefined instruction trap being taken.

3.6.3 Cache Type Register

This is a read-only register that contains information about the size and architecture of the caches, enabling operating systems to establish how to perform such operations as cache cleaning and lockdown. All ARMv4T and later cached processors contain this register, enabling RTOS vendors to produce future-proof versions of their operating systems.

You can access the Cache Type Register by reading CP15 c0 with the Opcode_2 field set to 1. For example:

```
MRC p15,0,<Rd>,c0,c0,1; returns cache details
```

The format of the Cache Type Register is shown in Figure 3-13.



Figure 3-13 Cache Type Register format

Cache Type Register field descriptions are shown in Table 3-9.

Table 3-9 Cache Type Register field descriptions

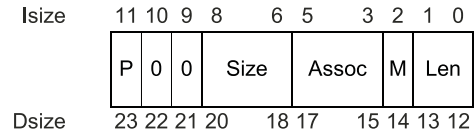
Bits	Field name	Description
[28:25]	Ctype	Specifies if the cache supports lockdown or not, and how it is cleaned. See Table 3-10 on page 3-29. For ARM1136JF-S processor Ctype = b1110.
[24]	S bit	Specifies whether the cache is a Unified Cache (S=0), or separate Instruction and Data Caches (S=1). For ARM1136JF-S processors S = 1.
[23:12]	Dsize	Specifies the size, line length, and associativity of the Data Cache.
[11:0]	Isize	Specifies the size, line length, and associativity of the Instruction Cache.

The Ctype field specifies if the cache supports lockdown or not, and how it is cleaned. The encoding for ARM1136JF-S processors is shown in Table 3-10.

Table 3-10 Ctype encoding

Value	Method	Cache cleaning	Cache lockdown
b1110	Write-Back	Register 7 operations	Format C

The Dsize and Isize fields in the Cache Type Register have the same format. This is shown in Figure 3-14.

**Figure 3-14 Dsize and Isize field format**

A summary of Dsize and Isize fields shown in Figure 3-14 is shown in Table 3-11.

Table 3-11 Dsize and Isize field summary

Field	Description
P bit	The P bit indicates if there is a restriction on page allocation for bits [13:12] of the virtual address: 0 = no restriction 1 = restriction applies to bits [13:12] of the virtual address. For ARM1136JF-S processors, the P bit is set if the Cache size is greater than 16KB. For more details see <i>Restrictions on page table mappings</i> on page 6-41.
Size	The Size field determines the cache size in conjunction with the M bit.
Assoc	The Assoc field determines the cache associativity in conjunction with the M bit. For ARM1136JF-S processor Ctype = b010.
M bit	The multiplier bit. Determines the cache size and cache associativity values in conjunction with the Size and Assoc fields. In the ARM1136JF-S processor the M bit is set to 0 for the Data and Instruction Caches.
Len	The Len field determines the line length of the cache. For ARM1136JF-S processor Len = b10.

The size of the cache is determined by the Size field and the M bit. The M bit is 0 for the Data and Instruction Caches. Bits [20:18] for the Data Cache and bits [8:6] for the Instruction Cache are the Size field. Table 3-12 shows the cache size encoding.

Table 3-12 Cache size encoding (M=0)

Size field	Cache size
b000	0.5KB
b001	1KB
b010	2KB

Table 3-12 Cache size encoding (M=0) (continued)

Size field	Cache size
b011	4KB
b100	8KB
b101	16KB
b110	32KB
b111	64KB

The associativity of the cache is determined by the Assoc field and the M bit. The M bit is 0 for the Data and Instruction Caches. Bits [17:15] for the Data Cache and bits [5:3] for the Instruction Cache are the Assoc field. Table 3-13 shows the cache associativity encoding.

Table 3-13 Cache associativity encoding (M=0)

Assoc field	Associativity
b000	Reserved
b001	
b010	4-way
b011	Reserved
b100	
b101	
b110	
b111	

The line length of the cache is determined by the Len field. Bits [13:12] for the Data Cache and bits [1:0] for the Instruction Cache are the Len field. Table 3-14 shows the line length encoding.

Table 3-14 Line length encoding

Len field	Cache line length
b00	Reserved
b01	Reserved
b10	8 words (32 bytes)
b11	Reserved

The Cache Type Register values for an ARM1136JF-S processor with the following configuration is shown in Table 3-15:

- separate Instruction and Data Caches
- cache size = 16KB
- associativity = 4-way
- line length = eight words
- caches use Write-Back, CP15 c7 for cache cleaning, and Format C for cache lockdown.

Table 3-15 Example Cache Type Register format

Function	Register bits	Value	
Reserved	[31:29]	b000	
Ctype	[28:25]	b1110	
S	[24]	b1 = Harvard cache	
Dsize	P	b0	
	Reserved	[22, 21]	b00
	Size	[20:18]	b0101 = 16KB
	Assoc	[17:15]	b010 = 4-way
	M	[14]	b0
Len	[13:12]	b10 = 8 words per line (32 bytes)	

Table 3-15 Example Cache Type Register format (continued)

Function		Register bits	Value
Isize	P	[11]	b0
	Reserved	[10:9]	b00
	Size	[8:6]	b0101 = 16KB
	Assoc	[5:3]	b010 = 4-way
	M	[2]	b0
	Len	[1:0]	b10 = 8 words per line (32 bytes)

3.6.4 Cache Dirty Status Register

See *Clean, and Clean and Invalidate, Entire Data Cache operations* on page 3-24.

3.7 Debug access to caches and TLB

The debug access to ARM1136JF-S processor caches and TLBs is achieved using:

- *Cache debug operations*
- *Cache and main TLB Master Valid Registers* on page 3-37
- *MMU debug operations* on page 3-38.

3.7.1 Cache debug operations

The CP15 instructions and registers available to debug the cache are shown in Table 3-16.

Table 3-16 Cache debug CP15 operations

Function	Data	Instruction
Read to Data Debug Cache Register	Data	MRC p15, 3, <Rd>, c15, c0, 0
Read to Instruction Debug Cache Register	Data	MRC p15, 3, <Rd>, c15, c0, 1
Data Tag RAM Read Operation	Set/Index	MCR p15, 3, <Rd>, c15, c2, 0
Instruction Tag RAM Read Operation	Set/Index	MCR p15, 3, <Rd>, c15, c2, 1
Instruction Cache Data RAM Read Operation	Set/Index/Word	MCR p15, 3, <Rd>, c15, c4, 1
Write to Cache Debug Control Register	Data	MCR p15, 7, <Rd>, c15, c0, 0
Read to Cache Debug Control Register	Data	MRC p15, 7, <Rd>, c15, c0, 0

The CP15 cache debug operations registers are also shown in Figure 3-4 on page 3-14.

For debug operations, the cache refill operations can be disabled, while keeping the caches themselves enabled. This enables the debugger to access the system without unsettling the state of the processor.

The cache refill operations are disabled using the Cache Debug Control Register.

Cache Debug Control Register

You can access the Cache Debug Control Register to activate the cache debug features by reading or writing CP15 c15 with the CRm field set to c0:

```
MRC p15, 7, <Rd>, c15, c0, 0    ; Read cache debug control register
MCR p15, 7, <Rd>, c15, c0, 0    ; Write cache debug control register
```

The format of the Cache Debug Control Register is shown in Figure 3-15 on page 3-35.

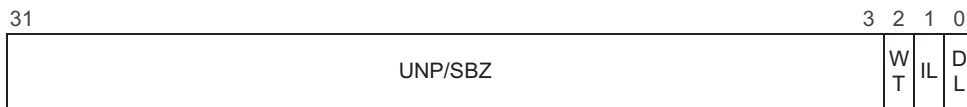
**Figure 3-15 Cache Debug Control Register format**

Table 3-17 describes the functions of the Cache Debug Control Register bits.

Table 3-17 Cache Debug Control Register bit functions

Bits	Reset value	Name	Description
[31:3]	UNP/SBZ	-	Reserved
[2]	0	WT	1 = force Write-Through behavior for regions marked as Write-Back 0 = do not force Write-Through for regions marked as Write-Back (normal operation)
[1]	0	IL	1 = Instruction Cache linefill disabled 0 = cache linefill enabled (normal operation)
[0]	0	DL	1 = Data Cache linefill disabled 0 = linefill enabled (normal operation)

Instruction and Data Debug Cache Registers

The Instruction and Data Debug Cache Registers are CP15 registers that can be read into an ARM register. You can access the Instruction and Data Debug Cache Registers by using the following instructions:

```
MRC p15, 3, <Rd>, c15, c0, 1    ; Read Instruction Debug Cache Register
MRC p15, 3, <Rd>, c15, c0, 0    ; Read Data Debug Cache Register
```

The format of the data returned is shown in Figure 3-16.

**Figure 3-16 Instruction and Data Debug Cache Register format**

For the Instruction Cache, the dirty bits are returned as 0.

The Tag address is formed from the Tag RAM contents and the Tag Index. For a cache way size of greater than 1KB, bits [31:10] are formed from the Tag contents. This ensures that the data format returned is consistent regardless of cache size.

For SmartCache debug, the base address register can be read to determine the addresses that are covered by the SmartCache.

The debugger can then use the addresses generated from the Tag to access memory, including the cache. For SmartCache debug, the refill disable (using the Cache Debug Control Register) must be implemented to avoid this reading of data for debug purposes bringing data into the SmartCache.

Instruction Cache Data RAM entries can be read in a similar manner to the reading of the Tag RAM. An MCR operation transfers the Set, Index, and Word to the Instruction Cache, and this causes a read of this word in the Instruction Cache Data RAM into the Instruction Debug Cache Register. This register is then read by an MRC operation. Providing access to the Instruction Cache Data RAM in this way ensures that problems caused by an incoherent instruction cache can be debugged.

The format of Set/Index/Word data is shown in Figure 3-17, where A and S are the logarithms base 2 of the cache size parameters Associativity and N sets, rounded up to an integer in the case of A. These parameters can be found in the Cache Type Register. For CP15 instructions that require Set/Index, the same format is used, but the Word field is ignored.

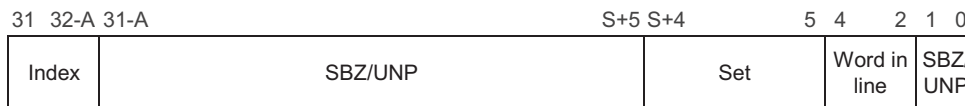


Figure 3-17 Index/Set/Word format

3.7.2 Cache and main TLB Master Valid Registers

The cache and main TLB Master Valid Registers are described in Table 3-18.

Table 3-18 Cache and main TLB Master Valid Registers description

Register	Description
Data Cache Master Valid Register	These registers enable the Valid bits held in the Instruction and Data Valid RAM for the cache and SmartCache to be masked, so that a single cycle invalidation of the cache can be performed without requiring special resettable RAM cells.
Instruction Cache Master Valid Register	
Data SmartCache Master Valid Register	
Instruction SmartCache Master Valid Register	
	The number of Master Valid bits is a function of the cache and SmartCache size. There are 64 cache Master Valid bits for a 16KB cache, and 64 SmartCache Valid bits for a 16KB SmartCache. The number of bits scales linearly with cache size. The maximum number of 32 bit registers required for the largest cache size (64K) is 8, as is the maximum number for the SmartCache. The registers fill from the LSB of the lowest numbered register upwards with these Valid bits.
	Unimplemented Valid bits are Unpredictable for reads and <i>Should Be Zero or Preserved (SBZP)</i> for writes.
Main TLB Master Valid Register	<p>The Main TLB Valid bits are implemented as a pair of registers.</p> <p>If you modify the values of the Valid bits using this mechanism the effects can be Unpredictable. These registers can only be written to when the cache and main TLB are disabled, and the values to be written are the values that were previously read out.</p> <p>The instructions to access the Valid bits are shown in Table 3-19 on page 3-38. The Register Number fields for these instructions refer to the multiple registers required to capture all the Valid bits. For the main TLB accesses the register number field must be 0 or 1. For the cache and SmartCache Master Valid bits, the highest Register Number is one less than the number of times 8KB divides into the cache size.</p>

Table 3-19 Cache, SmartCache, and main TLB Valid bit access functions

Function	Instruction
Read Instruction Cache Master Valid Register	MRC p15, 3, <Rd>, c15, c8, <Register Number>
Write Instruction Cache Master Valid Register	MCR p15, 3, <Rd>, c15, c8, <Register Number>
Read Instruction SmartCache Master Valid Register	MRC p15, 3, <Rd>, c15, c10, <Register Number>
Write Instruction SmartCache Master Valid Register	MCR p15, 3, <Rd>, c15, c10, <Register Number>
Read Data Cache Master Valid Register	MRC p15, 3, <Rd>, c15, c12, <Register Number>
Write Data Cache Master Valid Register	MCR p15, 3, <Rd>, c15, c12, <Register Number>
Read Data SmartCache Master Valid Register	MRC p15, 3, <Rd>, c15, c14, <Register Number>
Write Data SmartCache Master Valid Register	MCR p15, 3, <Rd>, c15, c14, <Register Number>
Read Main TLB Master Valid Register	MRC p15, 5, <Rd>, c15, c14, <Register Number>
Write Main TLB Master Valid Register	MCR p15, 5, <Rd>, c15, c14, <Register Number>

3.7.3 MMU debug operations

The debug architecture for the ARM1136JF-S processor is described in Chapter 13 *Debug*. The External Debug Interface is based on JTAG, and is as described in Chapter 14 *Debug Test Access Port*.

The MMU debug functions are described in:

- *Operations for TLB debug control* on page 3-39
- *MicroTLB debug* on page 3-39
- *Main TLB debug* on page 3-40
- *Control of main TLB and MicroTLB loading and matching* on page 3-41
- *TLB VA Registers* on page 3-44
- *TLB PA Registers* on page 3-45
- *TLB Attribute Registers* on page 3-47.

Operations for TLB debug control

The CP15 c15 operations used for the debug of the main TLB and MicroTLBs are shown in Table 3-20.

Table 3-20 MicroTLB and main TLB debug operations

Function	Data	Instruction
Read TLB Debug Control Register	Data	MRC p15, 7, <Rd>, c15, c1, 0
Write to TLB Debug Control Register	Data	MCR p15, 7, <Rd>, c15, c1, 0
Read Data MicroTLB Entry Operation	MicroTLB index	MCR p15, 5, <Rd>, c15, c4, 0
Read Instruction MicroTLB Entry Operation	MicroTLB index	MCR p15, 5, <Rd>, c15, c4, 1
Read Main TLB Entry Register	Main TLB index	MCR p15, 5, <Rd>, c15, c4, 2
Write Main TLB Entry Register	Main TLB index	MCR p15, 5, <Rd>, c15, c4, 4
Read Data MicroTLB VA Register	Data	MRC p15, 5, <Rd>, c15, c5, 0
Read Data MicroTLB PA Register	Data	MRC p15, 5, <Rd>, c15, c6, 0
Read Data MicroTLB Attribute Register	Data	MRC p15, 5, <Rd>, c15, c7, 0
Read Instruction MicroTLB VA Register	Data	MRC p15, 5, <Rd>, c15, c5, 1
Read Instruction MicroTLB PA Register	Data	MRC p15, 5, <Rd>, c15, c6, 1
Read Instruction MicroTLB Attribute Register	Data	MRC p15, 5, <Rd>, c15, c7, 1
Read Main TLB VA Register	Data	MRC p15, 5, <Rd>, c15, c5, 2
Write Main TLB VA Register	Data	MCR p15, 5, <Rd>, c15, c5, 2
Read Main TLB PA Register	Data	MRC p15, 5, <Rd>, c15, c6, 2
Write Main TLB PA Register	Data	MCR p15, 5, <Rd>, c15, c6, 2
Read Main TLB Attribute Register	Data	MRC p15, 5, <Rd>, c15, c7, 2
Write Main TLB Attribute Register	Data	MCR p15, 5, <Rd>, c15, c7, 2

MicroTLB debug

The debugger can read MicroTLB entries using CP15 c15 operations that specify the index in the MicroTLB to determine which entry is being read. The read operation reads the requested MicroTLB entry into the following registers:

- MicroTLB VA Register

- MicroTLB PA Register
- MicroTLB Attributes Register.

It is not possible to write the MicroTLB entries using this mechanism.

The format of the VA, PA, and Attributes registers for the main TLB and MicroTLB entries are described in:

- *TLB VA Registers* on page 3-44
- *TLB PA Registers* on page 3-45
- *TLB Attribute Registers* on page 3-47.

The format of the Index register used to access the MicroTLB entries is shown in Figure 3-18. Values of the MicroTLB index greater than 10 do not access any MicroTLB entry.



Figure 3-18 MicroTLB index format

Main TLB debug

The debugger can read or write the individual entries of the main TLB using CP15 c15 operations that specify the index of the main TLB entry to be written or read. This enables a debugger to determine the individual entries within the main TLB. The read operation reads the requested main TLB entry into the following registers:

- Main TLB VA Register
- Main TLB PA Register
- Main TLB Attributes Register.

In a similar manner, the write operation copies these registers into the main TLB.

The format of the VA, PA, and Attributes registers for the main TLB and MicroTLB entries are described in:

- *TLB VA Registers* on page 3-44
- *TLB PA Registers* on page 3-45
- *TLB Attribute Registers* on page 3-47.

The format of the Index register used to access the main TLB entries is shown in Figure 3-19.

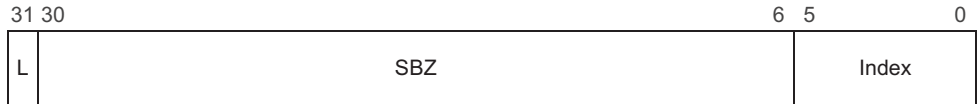


Figure 3-19 Main TLB index format

The bit functions of the main TLB index are shown in Table 3-21.

Table 3-21 Main TLB index bit functions

Bits	Name	Meaning
31	L	Lockable region. Indicates whether the index refers to the lockable region or the set-associative region: 0 = Index refers to the set-associative region 1 = Index refers to the lockable region.
[30:6]	-	SBZ
[5:0]	Index	Indicates which entry in the main TLB is being referred to. The meaning of this field depends on the setting of the L bit: L = 0 Index[5] indicates which way of the main TLB set-associative region is being accessed. Index[4:0] indexes the set of the RAM. L = 1 Index[5:3] SBZ. Index[2:0] indicates which entry in the lockable region is being accessed.

Control of main TLB and MicroTLB loading and matching

You can disable the MicroTLB automatic loading from the main TLB, the loading of the main TLB after a hardware page table walk, and the matching of entries in either the main TLB or the MicroTLB using the TLB Debug Control Register in CP15 c15.

When the automatic loading from the MicroTLB is disabled, all MicroTLB misses are serviced from the main TLB, and do not update the MicroTLB. When the loading of the main TLB is disabled, then misses do not result in the main TLB being updated. This has a significant impact on performance, but enables debug operations to be performed in as unobtrusive a manner as possible.

Disabling the matches in the MicroTLB or main TLB causes all accesses to be serviced by reading from the main TLB or by doing a page table walk respectively. This enables alternative page mappings to be created without having to change the TLB contents. This enables debugging to be performed in as unobtrusive a manner as possible. Disabling matches without also disabling the loading of the corresponding TLB can have Unpredictable effects.

The format of the TLB Debug Control Register is shown in Figure 3-20.

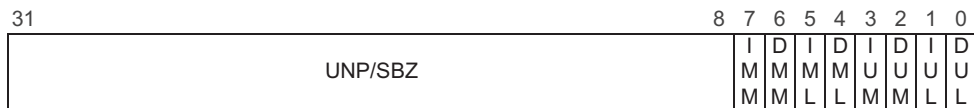


Figure 3-20 TLB Debug Control Register format

Table 3-22 describes the functions of the TLB Debug Control Register bits.

Table 3-22 TLB Debug Control Register bit functions

Bits	Reset value	Name	Description
[31:8]	UNP/SBZ	-	Reserved
[7]	0	IMM	1 = Instruction main TLB match disabled 0 = Instruction main TLB match enabled
[6]	0	DMM	1 = Data main TLB match disabled 0 = Data main TLB match enabled
[5]	0	IML	1 = Instruction main TLB load disabled 0 = Instruction main TLB load enabled
[4]	0	DML	1 = Data main TLB load disabled 0 = Data main TLB load enabled
[3]	0	IUM	1 = Instruction MicroTLB match disabled 0 = Instruction MicroTLB match enabled
[2]	0	DUM	1 = Data MicroTLB match disabled 0 = Data MicroTLB match enabled
[1]	0	IUL	1 = Instruction MicroTLB load and flush disabled 0 = Instruction MicroTLB load and flush enabled
[0]	0	DUL	1 = Data MicroTLB load and flush disabled 0 = Data MicroTLB load and flush enabled

Because the ARM1136JF-S processor has a unified main TLB, the IMM bit must be set to the same as the DMM bit, and the IML bit must be set to the same as the DML bit, or else the effect is Unpredictable.

TLB VA Registers

The TLB VA Registers are:

- Data MicroTLB VA Register (read-only)
- Instruction MicroTLB VA Register (read-only)
- Main TLB VA Register.

You can access the TLB VA Registers through CP15 c15 using the following instructions:

```
MRC p15, 5, <Rd>, c15, c5, 0    ; Read Data MicroTLB VA Register
MRC p15, 5, <Rd>, c15, c5, 1    ; Read Instruction MicroTLB VA Register
MRC p15, 5, <Rd>, c15, c5, 2    ; Read Main TLB VA Register
MCR p15, 5, <Rd>, c15, c5, 2    ; Write Main TLB VA Register
```

The TLB VA Registers have the format shown in Figure 3-21.

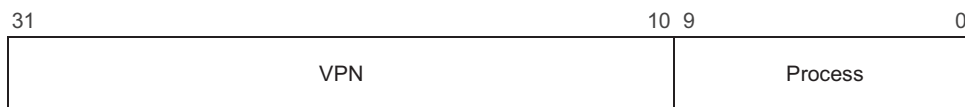


Figure 3-21 TLB VA Registers format

Table 3-23 describes the functions of the TLB VA Register bits.

Table 3-23 TLB VA Register bit functions

Bits	Name	Function
[31:10]	VPN	Virtual Page Number. Bits of the virtual page number that are not translated as part of the page table translation because the size of the tables are Unpredictable when read, and Should Be Zero when written.
[9:0]	PROCESS	Memory space identifier that determines if the entry is a global mapping, or an ASID dependent entry. The format of the memory space identifier is shown in Figure 3-22 on page 3-45.

The format of the memory space identifier is shown in Figure 3-22 on page 3-45.

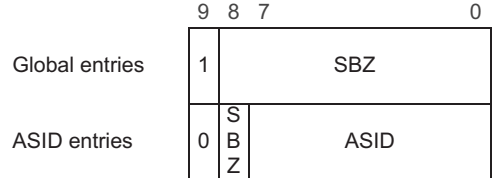


Figure 3-22 Memory space identifier format

TLB PA Registers

The TLB PA Registers are:

- Data MicroTLB PA Register (read-only)
- Instruction MicroTLB PA Register (read-only)
- Main TLB PA Register.

You can access the TLB PA Registers through CP15 c15 using the following instructions:

```
MRC p15, 5, <Rd>, c15, c6, 0 ; Read Data MicroTLB PA Register
MRC p15, 5, <Rd>, c15, c6, 1 ; Read Instruction MicroTLB PA Register
MRC p15, 5, <Rd>, c15, c6, 2 ; Read Main TLB PA Register
MCR p15, 5, <Rd>, c15, c6, 2 ; Write Main TLB PA Register
```

The TLB PA Registers have the format shown in Figure 3-23.

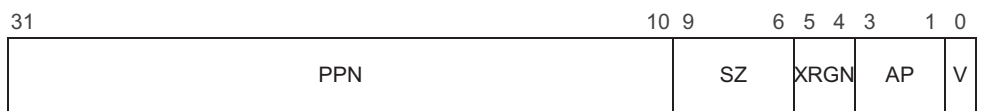


Figure 3-23 TLB PA Registers format

Table 3-24 describes the functions of the TLB PA Register bits.

Table 3-24 TLB PA Register bit functions

Bits	Name	Function
[31:10]	PPN	Physical Page Number. Bits of the physical page number that are not translated as part of the page table translation are Unpredictable when read and Should Be Zero when written.
[9:6]	SZ	Region size. The region size that is contained in the MicroTLB might be smaller than specified in the page tables. The MicroTLB can split main TLB entries that cover regions which cover areas of memory contained in the TCM into smaller sizes. In addition, subpages are reported as separate pages in the MicroTLBs. The format of the SZ field is shown in Table 3-25.
[5:4]	XRGN	Extended Region Type. The region type bits determine the attributes for the memory region, as shown in Table 3-30 on page 3-49 and Table 3-26 on page 3-47.
[3:1]	AP	Access Permission. For MicroTLB entries the access permissions refer to the subpage that is contained in that MicroTLB entry, according to the format in Table 3-27 on page 3-47. For main TLB entries, this register contains the access permission fields for the first subpage or the entire page/section if the page does not support subpages.
0	V	Valid bit. Indicates that this TLB entry is valid.

Table 3-25 shows the encoding of the SZ field.

Table 3-25 SZ field encoding

SZ	Description
b1111	1KB subpage (used by MicroTLB only)
b1110	4KB page
b1100	16KB subpage (used by MicroTLB only)
b1000	64KB page
b0000	1MB section (or part of 16MB supersection for MicroTLB)
b0001	16M Supersection (used by main TLB only)
All other values	Reserved

Table 3-26 shows the encoding of the XRGN field, XRGN format.

Table 3-26 XRGN field encoding, XRGN format

XRGN	Description
b00	Noncachable
b01	Outer WB, Allocate On Write
b10	Outer WT, No Allocate on Write
b11	Outer WB, No Allocate on Write

Table 3-27 shows the encoding of the AP field.

Table 3-27 AP field encoding

AP field	Supervisor permissions	User permissions	Description
b000	No access	No access	All accesses generate a permission fault
b001	Read/write	No access	Supervisor access only
b010	Read/write	Read-only	Writes in User mode generate permission faults
b011	Read/write	Read/write	Full access
b100	No access	No access	Domain fault encoded field
b101	Read-only	No access	Supervisor read-only
b110	Read-only	Read-only	Supervisor/User read-only
b111	-	-	Reserved

TLB Attribute Registers

The TLB Attribute Registers are:

- Data MicroTLB Attribute Register (read-only)
- Instruction MicroTLB Attribute Register (read-only)
- Main TLB Attribute Register.

You can access the TLB Attribute Registers through CP15 c15 using the following instructions:

MRC p15, 5, <Rd>, c15, c7, 0 ; Read Data MicroTLB Attribute Register
 MRC p15, 5, <Rd>, c15, c7, 1 ; Read Instruction MicroTLB Attribute Register
 MRC p15, 5, <Rd>, c15, c7, 2 ; Read Main TLB Attribute Register
 MCR p15, 5, <Rd>, c15, c7, 2 ; Write Main TLB Attribute Register

The TLB Attribute Registers have the format shown in Figure 3-24.



Figure 3-24 TLB Attribute Register format

Table 3-28 describes the functions of the TLB Attribute Register bits.

Table 3-28 TLB Attribute Register bit functions

Bits	Name	Function
[31:30]	AP3	Subpage access permissions for the fourth subpage if the page or section supports subpages. Unpredictable on read and Should Be Zero on a write if the entry does not support subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-29 on page 3-49. This field is Unpredictable for reads from the MicroTLB.
[29:28]	AP2	Subpage access permissions for third subpage if the page or section supports subpages. Unpredictable on read and Should Be Zero on a write if the entry does not support subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-29 on page 3-49. This field is Unpredictable for reads from the MicroTLB.
[27:26]	AP1	Subpage access permissions for second subpage if the page or section supports subpages. Unpredictable on read and Should Be Zero on a write if the entry does not support subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-29 on page 3-49. This field is Unpredictable for reads from the MicroTLB.
25	SPV	Subpage Valid. Indicates that the page or section supports subpages. Pages that support subpages must be marked as Global. Attempting to use subpages with nonglobal pages has Unpredictable results. This field is 0 for reads from the MicroTLB: 0 = Subpages are not supported 1 = Subpages are supported.
[24:9]	-	Should Be Zero.

Table 3-28 TLB Attribute Register bit functions (continued)

Bits	Name	Function
[8:5]	Domain	Domain number of the TLB entry.
4	XN	Execute Never attribute. This field is Unpredictable for a read from the Data MicroTLB Attribute Register.
[3:1]	RGN	Region type. The format of the extended region field is shown in Table 3-30.
0	S	Shared attribute.

The Upper subpage access permission field encodings are shown in Table 3-29.

Table 3-29 Upper subpage access permission field encoding

Upper subpage permissions AP[1:0]	CP15		Description
	S	R	
b00	0	0	All accesses generate a permission fault.
b00	1	0	Supervisor read-only. User no access.
b00	0	1	Supervisor or User read-only.
b00	1	1	Unpredictable.
b01	X	X	Supervisor access only.
b10	X	X	Supervisor full access. User read-only.
b11	X	X	Full access.

Table 3-30 shows the encoding of the RGN field, RGN format.

Table 3-30 XRGN field encoding, RGN format

RGN	Description
b000	Noncachable
b001	Strongly Ordered
b010	Reserved

Table 3-30 XRGN field encoding, RGN format (continued)

RGN	Description
b011	Device
b100	Reserved
b101	Reserved
b110	Inner WT, No Allocate on Write
b111	Inner WB, No Allocate on Write

3.8 DMA control

ARM1136JF-S DMA control is provided by:

- *DMA registers*
- *DMA Channel Number Register* on page 3-53
- *DMA Channel Status Registers* on page 3-53
- *DMA Context ID Registers* on page 3-55
- *DMA Context ID Registers* on page 3-55
- *DMA Control Register* on page 3-56
- *DMA Enable Register* on page 3-59
- *DMA External Start Address Registers* on page 3-61
- *DMA Identification and Status Registers* on page 3-61
- *DMA Internal End Address Register* on page 3-63
- *DMA Internal Start Address Register* on page 3-63
- *DMA User Accessibility Register* on page 3-64.

3.8.1 DMA registers

CP15 Register c11 accesses the DMA registers. The value of the CRm field determines which register is accessed. The possible values of CRm are shown in Table 3-31.

Table 3-31 DMA registers

Register	CRm	Opcode_2	Read/write	Notes
DMA Identification and Status Register	0	Present, Queued, Running, or Interrupting	Privileged only, Read-only	-
DMA User Accessibility Register	1	0	Privileged only, Read/write	-
DMA Channel Number Register	2	0	Read/write	-
DMA Enable Register	3	Stop, Start, or Clear	Write-only	One register per channel
DMA Control Register	4	0	Read/write	One register per channel
DMA Internal Start Address Register	5	0	Read/write	One register per channel
DMA External Start Address Register	6	0	Read/write	One register per channel
DMA Internal End Address Register	7	0	Read/write	One register per channel

Table 3-31 DMA registers (continued)

Register	CRm	Opcode_2	Read/write	Notes
DMA Channel Status Register	8	0	Read-only	One register per channel
Reserved (SBZ/UNP)	9-14	-	Read/write	-
DMA Context ID Register	15	0	Privileged only, Read/write	One register per channel

The Enable, Control, Internal Start Address, External Start Address, Internal End Address, Channel Status, and Context ID registers are multiple registers, with one register of each for each channel that is implemented. The register accessed is determined by the DMA Channel Number Register, as described in *DMA Channel Number Register* on page 3-53.

Figure 3-25 shows the functions and registers that you can access using MCR and MRC instructions with CP15 c11, the DMA registers.

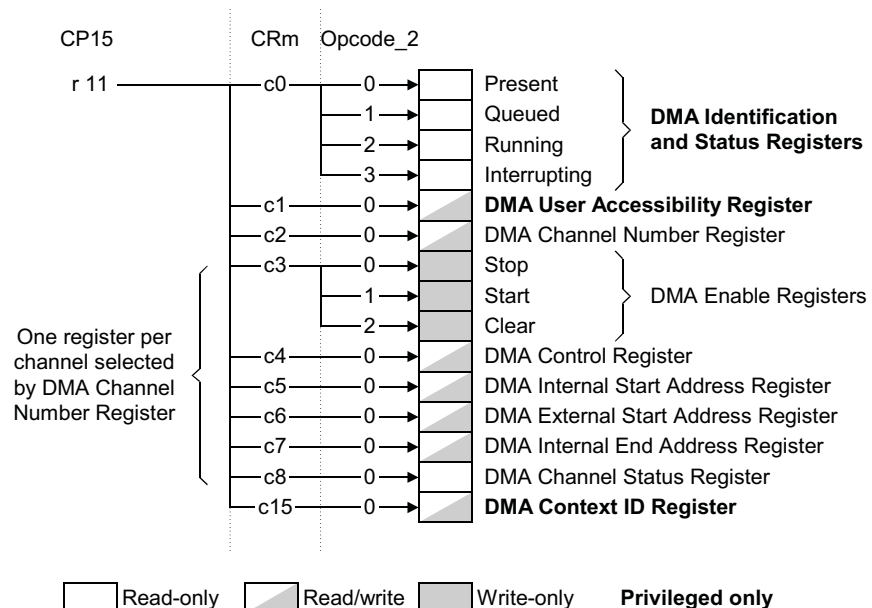


Figure 3-25 DMA registers

User Access to CP15 c11 operations

Several CP15 c11 operations can be executed by code while in User mode.

Attempting to execute a privileged operation in User mode using CP15 c11 results in the Undefined instruction trap being taken.

3.8.2 DMA Channel Number Register

The Enable, Control, Internal Start Address, External Start Address, Internal End Address, Channel Status, and Context ID registers are multiple registers with one register of each for each channel that is implemented. The value contained in the channel number register is used to determine which of the multiple registers is accessed when one of these registers is specified.

You can access this register by User processes if the U Bit of the DMA User Accessibility Register for any channel is set to 1. If no channel has the U bit set to 1 then attempting to access them by a User process results in an Undefined instruction trap.

The DMA Channel Number Register format is shown in Figure 3-26.

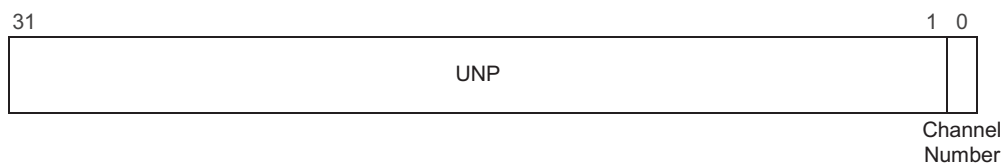


Figure 3-26 DMA Channel Number Register format

3.8.3 DMA Channel Status Registers

The DMA Channel Status Register for each channel defines the status of the most recently started DMA operation on that channel. It is a read-only register.

You can access the DMA Channel Status Register by setting the DMA Channel Number Register to the appropriate DMA channel and reading CP15 c11 with the CRm field set to c8:

```
MRC p15, 0, <Rd>, c11, c8, 0; Read DMA Channel Status Register
```

The format of the DMA Channel Status Registers is shown in Figure 3-27 on page 3-54.

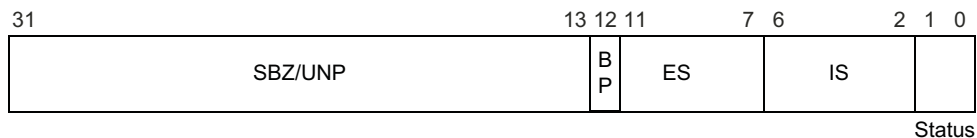


Figure 3-27 DMA Channel Status Register format

The functions of the bits in the DMA Channel Status Register are shown in Table 3-32.

Table 3-32 DMA Channel Status Register bit functions

Bits	Function
[31:13]	Reserved. Should Be Zero or Unpredictable.
[12] BP	The DMA parameters bit: 0 = DMA parameters are acceptable 1 = DMA parameters are conditioned inappropriately. The external start and end addresses, and the stride must all be multiples of the transaction size. If this is not the case, the BP bit is set to 1, and the DMA channel does not start.
[11:7] ES	The External Address Error Status bits: b00xxx = No error (reset value) b01001 = Unshared data error b11010 = External abort (can be imprecise) b11100 = External abort on translation of first-level page table b11110 = External abort on translation of second-level page table b10101 = Translation fault (section) b10111 = Translation fault (page) b11001 = Domain fault (section) b11011 = Domain fault (page) b11101 = Permission fault (section) b11111 = Permission fault (page). All other encodings are Reserved.

Table 3-32 DMA Channel Status Register bit functions (continued)

Bits	Function
[6:2] IS	The Internal Address Error Status bits: b00xxx = No error (reset value) b01000 = TCM out of range b11100 = External abort on translation of first-level page table b11110 = External abort on translation of second-level page table b10101 = Translation fault (section) b10111 = Translation fault (page) b11001 = Domain fault (section) b11011 = Domain fault (page) b11101 = Permission fault (section) b11111 = Permission fault (page). All other encodings are Reserved.
[1:0] Status	The Status bits: b00 = Idle b01 = Queued b10 = Running b11 = Complete or Error.

In the event of an error, the faulting address is contained in the appropriate Start Address Register, unless the error is an External Error (ES) that is set to b11010.

A channel with the state of Queued changes to Running automatically if the other channel (if implemented) changes to Idle, or Complete or Error, with no error.

When a channel has completed all of the transfers of the DMA, so that all changes to memory locations caused by those transfers are visible to other observers, its status is changed from Running to Complete or Error. This change does not happen before the external accesses from the transfer have completed.

If the U bit for the channel is set to 0, then attempting to read the register by a User process results in an Undefined instruction trap.

An Unshared data error is signaled on the External Address Error Status bits if a DMA transfer in User mode, or that has the UM bit set in the DMA Control Register, attempts to access external memory locations if those memory locations are not marked as Shared. A DMA transfer where the external address is within the range of the TCM also results in an Unshared data error.

3.8.4 DMA Context ID Registers

The DMA Context ID Register for each implemented DMA channel contains the processor Context ID of the process that is using the channel.

You can access the DMA Context ID register in a privileged mode by setting the DMA Channel Number Register to the appropriate DMA channel and reading or writing CP15 c11 with the CRm field set to c15:

```
MRC p15, 0, <Rd>, c11, c15, 0 ; Read DMA Context ID Register
MCR p15, 0, <Rd>, c11, c15, 0 ; Write DMA Context ID Register
```

The DMA Context ID Register must be written with the processor Context ID of the process to use the channel as part of the initialization of that channel. Where the channel is designated as a User-accessible channel, the Context ID must be written by the privileged process that initializes the channel for User use at the same time that the U bit for the channel is written to.

The format of the DMA Context ID Registers is shown in Figure 3-28.



Figure 3-28 DMA Context ID Register format

The bottom eight bits of the Context ID register are used in the address translation from virtual to physical addresses to enable different virtual address maps to co-exist. Attempting to write this register while the DMA channel is Running or Queued has no effect.

The bottom eight bits of the Context ID register are accessible to the AHB memory on **DMAASID[7:0]**.

This register can only be read by a privileged process to provide anonymity of the DMA channel usage from User processes. It can only be written by a privileged process for security reasons. On a context switch, where the state of the DMA is being stacked and restored, this register must be included in the saved state.

Attempting to access this privileged register by a User process results in an Undefined instruction trap being taken.

3.8.5 DMA Control Register

Each implemented DMA channel has its own DMA Control Register for controlling DMA operation.

You can access the DMA Control Register by setting the DMA Channel Number Register to the appropriate DMA channel and reading or writing CP15 c11 with the CRm field set to c4:

```
MRC p15, 0, <Rd>, c11, c4, 0 ; Read DMA Control Register
MCR p15, 0, <Rd>, c11, c4, 0 ; Write DMA Control Register
```

The register format for the DMA Control Registers is shown in Figure 3-29.

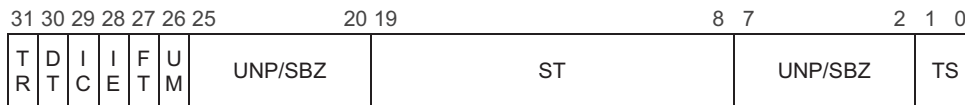


Figure 3-29 DMA Control Register format

Table 3-33 shows the functions controlled by the DMA Control Register bits.

If the U bit for the channel is set to 0, then attempting to access the register by a User process results in an Undefined instruction trap. Attempting to write to the DMA Control Register while the channel has the status of Running or Queued results in Unpredictable effects.

Table 3-33 DMA Control Register bit functions

Bits	Function
[31] TR	Target TCM: 0 = Data TCM 1 = Instruction TCM.
[30] DT	Direction of transfer: 0 = from level two memory to the TCM 1 = from the TCM to the level two memory.
[29] IC	Interrupt on Completion: 0 = No Interrupt on Completion 1 = Interrupt on Completion. The Interrupt on Completion bit indicates that the DMA channel must assert an interrupt on completing the DMA transfer. The interrupt is deasserted (from this source) if the Clear operation is performed on the channel causing the interrupt (see <i>DMA Enable Register</i> on page 3-59). The U bit has no effect on whether an interrupt is generated on completion.

Table 3-33 DMA Control Register bit functions (continued)

Bits	Function
[28] IE	<p>Interrupt on Error:</p> <p>0 = No Interrupt on Error (if the U bit is 0)</p> <p>1 = Interrupt on Error (regardless of the U bit).</p> <p>The Interrupt on Error bit indicates that the DMA channel must assert an interrupt on an error. The interrupt is deasserted (from this source) when the channel is set to Idle with a Clear operation, see <i>DMA Enable Register</i> on page 3-59. All DMA transactions on channels that have the U bit set to 1 Interrupt on Error regardless of the value written to this bit.</p>
[27] FT	<p>Full Transfer. Indicates that the DMA transfers all words of data as part of the DMA that is transferring data from the TCM to the external memory:</p> <p>0 = Transfer at least those locations in the address range of the DMA in the TCM that:</p> <ul style="list-style-type: none"> • have been changed by a store operation since the location was written to or read from by an earlier DMA • had the FT bit equal to 0 (or since Reset, whichever is the more recent operation). <p>1 = Transfer all locations in the address range of the DMA, regardless of whether or not the locations have been changed by a store. An access by the DMA to the TCM with the FT bit equal to 1 does not cause the record of what locations have been written to be changed.</p>
[26] UM	<p>User Mode. Indicates that the permission checks are based on the DMA being in User or privileged mode:</p> <p>0 = Transfer is a privileged transfer</p> <p>1 = Transfer is a User mode transfer.</p> <p>The UM bit is provided so that the User mode can be emulated by a privileged mode process. For a User mode process the setting of the UM bit is irrelevant and behaves as if set to 1.</p>
[25:20]	Reserved.
[19:8] ST	<p>Stride (in bytes). The Stride indicates the increment on the external address between each consecutive access of the DMA. A Stride of zero indicates that the external address is not to be incremented. This is designed to facilitate the accessing of volatile locations such as a FIFO.</p> <p>The value of the stride must be aligned to the Transaction Size, otherwise this results in Unpredictable behavior.</p> <p>The Stride is interpreted as a positive number (or zero).</p> <p>The internal address increment is not affected by the stride, but is fixed at the transaction size.</p>
[7:2]	Reserved.
[1:0] TS	<p>Transaction Size. The transaction size denotes the size of the transactions performed by the DMA channel. This is particularly important for Device or Strongly Ordered memory locations because it ensures that accesses to such memory occur at their programmed size:</p> <p>b00 = Byte</p> <p>b01 = Halfword</p> <p>b10 = Word</p> <p>b11 = Doubleword (8 bytes).</p>

Note

On ARM1136JF-S processors, setting the FT bit to 0 causes the DMA to look for dirty information at a granularity of four words, for the data TCM. That is, if any word/byte within a four-word range (aligned to a four-word boundary) has been written to, then these four words are written back. The FT bit has no effect for transfers from the Instruction TCM.

3.8.6 DMA Enable Register

Each implemented DMA channel has its own register location that can be written to Start, Stop, or Clear a channel. This is done by performing an MCR to the DMA Enable Register for that channel.

You can access the DMA Enable Register by setting the DMA Channel Number Register to the appropriate DMA channel and writing to CP15 c11 with the CRm value set to 3:

```
MCR p15, 0, <Rd>, c11, c3, <Opcode_2> ; Write DMA Enable Register
```

The value of `Opcode_2` in the MCR instruction determines the operation to be performed, as shown in Table 3-34.

Table 3-34 DMA Channel Enable Register operations

Opcode_2	Operation
0	Stop
1	Start
2	Clear
3-7	Reserved

Start

The Start command causes the channel to start DMA transfers. The channel status is changed to Running on the execution of a Start command if the other DMA channel is not in operation at that time, otherwise it is set to Queued.

A channel is in operation if:

- its channel status is Queued
- its channel status is Running

- its channel status is Complete or Error, with either the Internal or External Address Error Status not indicating No Error.

The channel status is described in *DMA Channel Status Registers* on page 3-53.

Stop

The Stop command can be issued when the channel status is Running. The DMA channel ceases to do memory accesses as soon as possible after the issuing of the instruction. This acceleration approach cannot be used for DMA transactions to or from memory regions marked as Device.

The DMA channel can take several cycles to stop after issuing a Stop instruction. The channel status remains at Running until the channel has stopped. The channel status is set to Idle at the point that all outstanding memory accesses have completed. The Start Address Registers contain the addresses required to restart the operation when the channel has stopped.

If the Stop command is issued when the channel status is Queued, the channel status is changed to Idle.

The Stop has no effect if the channel status is not Running or Queued.

The channel status is described in *DMA Channel Status Registers* on page 3-53.

Clear

The Clear command causes the channel status to change from Complete or Error to Idle. It also clears the interrupt that is set by the channel as a result of an error or completion (as defined in the control register in *Control Register* on page 3-96). The contents of the Internal and External Start Address Registers are unchanged by this command.

Issuing a Clear command when the channel has the status of Running or Queued has no effect.

If the U bit for a channel is set to 1 then the above operations for that channel can be performed by a User process. If the U bit for the channel is set to 0, then attempting to perform an operation by a User process results in an Undefined instruction trap.

The channel status is described in *DMA Channel Status Registers* on page 3-53.

Debug implications for the DMA

The level one DMA behaves as a separate engine from the processor core, and when started works autonomously. As a result, if the level one DMA has channels with the status of Running or Queued, then these channels continue to run, or start running, even

if the processor is stopped by debug mechanisms. This results in the contents of the TCM changing while the processor is stopped in Debug. The DMA channels must be stopped by a Stop operation to avoid this situation.

3.8.7 DMA External Start Address Registers

The DMA External Start Address Register for each channel defines the first address in external memory for that DMA channel. That is, the first address to or from where the data is to be transferred.

You can access the DMA External Start Address Register by setting the DMA Channel Number Register to the appropriate DMA channel and reading or writing CP15 c11 with the CRm field set to c6:

```
MRC p15, 0, <Rd>, c11, c6, 0    ; Read DMA External Start Address Register
MCR p15, 0, <Rd>, c11, c6, 0    ; Write DMA External Start Address Register
```

The External Start Address is a virtual address, whose physical mapping must be described in the page tables at the time that the channel is started. The memory attributes for that Virtual Address are used in the transfer, so memory permission faults might be generated.

The External Start Address must lie in the external memory beyond the level one memory system otherwise the results are Unpredictable.

The contents of this register are Unpredictable while the DMA channel is Running. When the channel is stopped because of a Stop command, or an error, it contains the address required to restart the transaction. On completion, it contains the address equal to the final address that was accessed plus the Stride.

The External Start Address must be aligned to the transaction size set in the control register, otherwise the effects are Unpredictable.

If the U bit for the channel is set to 0, then attempting to access the register by a User process results in an Undefined instruction trap. Attempting to write this register while the DMA channel is Running or Queued has no effect.

3.8.8 DMA Identification and Status Registers

The DMA Identification and Status Registers define the DMA channels that are physically implemented on the particular device and their current status. They can be used by processes handling DMA to determine the physical resources implemented and their availability.

You can access the DMA Identification and Status Registers by reading CP15 c11 in a privileged mode with the CRm field set to c0:

MRC p15, 0, <Rd>, c11, c0, <Opcode_2> ; Read DMA ID and Status Register

The Opcode_2 value identifies the registers implemented and their status as shown in Table 3-35.

Table 3-35 DMA Identification and Status Register functions

Opcode_2	Function
0	Present: 1 = the channel is Present 0 = the channel is not Present.
1	Queued: 1 = the channel is Queued 0 = the channel is not Queued.
2	Running: 1 = the channel is Running 0 = the channel is not Running.
3	Interrupting: 1 = the channel is Interrupting (through completion or an error) 0 = the channel is not Interrupting.
4-7	Reserved. Unpredictable.

The DMA Identification and Status Registers 0-3 have the format shown in Figure 3-30.

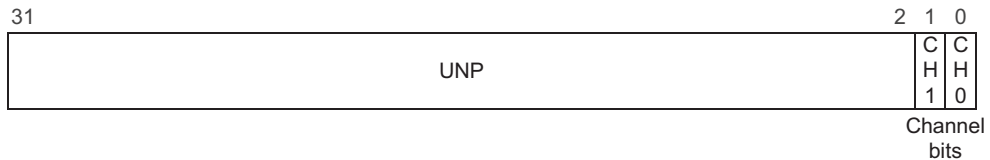


Figure 3-30 DMA Identification and Status Registers format

The bottom two bits, the Channel bits, of each register correspond to the two channels that are defined architecturally:

bit 0 corresponds to channel 0

bit 1 corresponds to channel 1.

These registers can only be read by a privileged process. Attempting to access them by a User process results in an Undefined instruction trap.

3.8.9 DMA Internal End Address Register

This register defines the Internal End Address. The Internal End Address is the final internal address (modulo the transaction size) that the DMA is to access plus the transaction size. Therefore the Internal End Address is the first (incremented) address that the DMA does not access.

If the Internal End Address is the sum of the Internal Start Address, the DMA transfer completes immediately without performing transactions.

When the transaction associated with the final internal address has completed, the whole DMA transfer is complete.

You can access the DMA Internal End Address Register by setting the DMA Channel Number Register to the appropriate DMA channel and reading or writing CP15 c11 with the CRm field set to c7:

```
MRC p15, 0, <Rd>, c11, c7, 0      ; Read DMA Internal End Address Register
MCR p15, 0, <Rd>, c11, c7, 0      ; Write DMA Internal End Address Register
```

The Internal End Address must be aligned to the transaction size set in the DMA Control Register or the effects are Unpredictable.

If the U bit of the DMA User Accessibility Register for the channel is set to 0, then attempting to access the DMA Internal End Address Register by a User process results in an Undefined instruction trap. Attempting to write to this register while the DMA channel is Running or Queued has no effect.

3.8.10 DMA Internal Start Address Register

This register defines the first address in the TCM for each DMA channel, that is the first address to or from which the data is to be transferred. The Internal Start Address is a Virtual Address, whose physical mapping must be described in the page tables at the time that the channel is started.

You can access the DMA Internal Start Address Register by setting the DMA Channel Number Register to the appropriate DMA channel and reading or writing CP15 c11 with the CRm field set to c5:

```
MRC p15, 0, <Rd>, c11, c5, 0      ; Read DMA Internal Start Address Register
MCR p15, 0, <Rd>, c11, c5, 0      ; Write DMA Internal Start Address Register
```

The memory attributes for that Virtual Address are used in the transfer, so memory permission faults might be generated. The Internal Start Address must lie within a TCM, otherwise an error is reported in the DMA Channel Status Register. The marking of memory locations in the TCM as being Device results in Unpredictable effects.

The contents of this register are Unpredictable while the DMA channel is Running. When the channel is stopped because of a Stop command, or an error, it contains the address required to restart the transaction. On completion, it contains the address equal to the Internal End Address.

The Internal Start Address must be aligned to the transaction size set in the DMA Control Register or the effects are Unpredictable.

If the U bit of the DMA User Accessibility Register for the channel is set to 0, then attempting to access the DMA Internal Start Address Register by a User process results in an Undefined instruction trap. Attempting to write this register while the DMA channel is Running or Queued has no effect. That is, it fails without issuing an error.

3.8.11 DMA User Accessibility Register

This register contains a bit for each channel, referred to as the U bit for that channel, that indicates if the registers for that channel can be accessed by a User mode process.

You can access the DMA User Accessibility Register in a privileged mode by reading or writing CP15 c11 with the CRm field set to c1:

MRC p15, 0, <Rd>, c11, c1, 0; Read DMA User Accessibility Register
MCR p15, 0, <Rd>, c11, c1, 0; Write DMA User Accessibility Register

The registers that can be accessed if the U bit for that channel is 1 are:

- *DMA Channel Status Registers* on page 3-53
- *DMA Control Register* on page 3-56
- *DMA Enable Register* on page 3-59
- *DMA External Start Address Registers* on page 3-61
- *DMA Internal Start Address Register* on page 3-63
- *DMA Internal End Address Register* on page 3-63

The contents of these registers must be preserved on a task switch if the registers are User-accessible.

If the U bit for that channel is set to 0, then attempting to access the registers by a User process results in an Undefined instruction trap.

The DMA User Accessibility Register has the format shown in Figure 3-31.

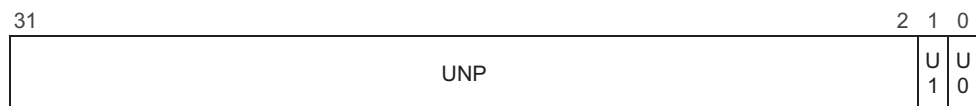


Figure 3-31 DMA User Accessibility Register format

3.9 Memory management unit configuration and control

ARM1136JF-S *Memory Management Unit* (MMU) configuration and control is provided by:

- *Fault Address Register*
- *Data Fault Status Register* on page 3-66
- *Domain Access Control Register* on page 3-67
- *Instruction Fault Address Register* on page 3-67
- *Instruction Fault Status Register* on page 3-68
- *Memory Region Remap Registers* on page 3-69
- *TLB Type Register* on page 3-74
- *TLB Operations Register* on page 3-75
- *TLB Lockdown Register* on page 3-77
- *Translation Table Base Control Register* on page 3-79
- *Translation Table Base Register 0* on page 3-80
- *Translation Table Base Register 1* on page 3-81
- *DMA Control Register* on page 3-56.

3.9.1 Fault Address Register

Reading CP15 c6 with Opcode_2 is set 0 returns Fault Address Register (FAR) as specified by the Opcode_2 value.

You can access the Fault Address Register by reading or writing CP15 c6 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c6, c0, 0      ; Read Fault Address Register
MCR p15, 0, <Rd>, c6, c0, 0      ; Write Fault Address Register
```

The Fault Address Register holds the modified virtual address of the access being attempted when a fault occurred. The Fault Address Register is only updated for precise data faults, not for imprecise data faults or prefetch faults.

Writing CP15 c6 with Opcode_2 set to 0 sets a FAR to the value of the data written. This is useful for a debugger to restore the value of a FAR.

The ARM1136JF-S processor also updates the FAR on debug exception entry because of watchpoints. This is architecturally Unpredictable. See *Effect of a debug event on CP15 registers* on page 13-30 for more details.

3.9.2 Data Fault Status Register

The Data Fault Status Register contains the source of the last data fault. The Data Fault Status Register indicates the domain and type of access being attempted when an abort occurred.

You can access the Data Fault Status Register by reading or writing CP15 c5 with the CRm and Opcode_2 fields set to 0:

```
MRC p15, 0, <Rd>, c5, c0, 0 ; Read Data Fault Status Register
MCR p15, 0, <Rd>, c5, c0, 0 ; Write Data Fault Status Register
```

The format of the Data Fault Status Register is shown in Figure 3-32.

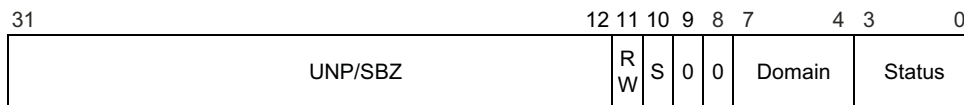


Figure 3-32 Data Fault Status Register format

Table 3-36 shows the bit fields for the Data Fault Status Register.

Table 3-36 Data Fault Status Register bits

Bits	Meaning
[31:12]	UNP/SBZ.
[11]	Not Read/Write. Indicates what type of access caused the abort: 0 = Read 1 = Write Aborts on CP15 operations. This bit is set to 1.
[10]	Part of the Status field. See Bits [3:0] in this table.
[9:8]	Always read as 0.
[7:4]	Specifies which of the 16 domains (D15-D0) was being accessed when a data fault occurred.
[3:0]	Type of fault generated (see <i>Fault status and address</i> on page 6-33).

Reading CP15 c5 with Opcode_2 set to 0 returns the value of the Data Fault Status Register.

Writing CP15 c5 with Opcode_2 set to 0 sets the Data Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the Data Fault Status Register. The register must be written using a read-modify-write sequence.

3.9.3 Domain Access Control Register

The Domain Access Control Register consists of sixteen discrete two-bit fields, each of which defines the access permissions for one of the sixteen domains (D15-D0).

You can access the Domain Access Control Register by reading or writing CP15 c3 with the CRm and Opcode_2 fields set to 0:

```
MRC p15, 0, <Rd>, c3, c0, 0      ; Read Domain Access Control Register
MCR p15, 0, <Rd>, c3, c0, 0      ; Write Domain Access Control Register
```

Figure 3-33 shows the two-bit domain access permission fields of the Domain Access Control Register.

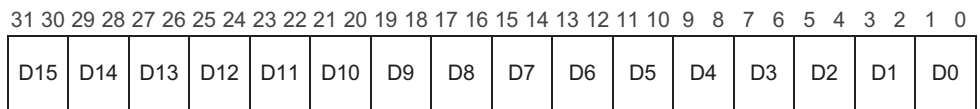


Figure 3-33 Domain Access Control Register format

Table 3-37 shows the encoding of the bits in the Domain Access Control Register.

Table 3-37 Encoding of domain bits in CP15 c3

Value	Access type	Description
b00	No access	Any access generates a domain fault
b01	Client	Accesses are checked against the access permission bits in the TLB entry
b10	Reserved	Any access generates a domain fault
b11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a permission fault cannot be generated

3.9.4 Instruction Fault Address Register

Reading CP15 c6 returns the *Instruction Fault Address Register* (IFAR) as specified by the Opcode_2 value.

You can access the Instruction Fault Address Register by reading or writing CP15 c6 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c6, c0, 1 ; Read Instruction Fault Address Register
MCR p15, 0, <Rd>, c6, c0, 1 ; Write Instruction Fault Address Register
```

The IFAR holds the virtual address of the instruction that triggered the watchpoint. The contents are Unpredictable after a precise Data Abort or Instruction Abort occurs.

If the watchpoint is taken when in ARM state, the IFAR contains the address of the instruction that triggered it plus $0x8$. If the watchpoint is taken while in Thumb state, the IFAR contains the address of the instruction that triggered it plus $0x4$. If the watchpoint is taken while in Java state, the IFAR contains the address of the instruction causing it.

Writing CP15 c6 with Opcode_2 set to 1 sets the IFAR to the value of the data written. This is useful for a debugger to restore the value of the IFAR.

3.9.5 Instruction Fault Status Register

The *Instruction Fault Status Register* (IFSR) contains the source of the last instruction fault. The IFSR indicates the type of access being attempted when an abort occurred.

You can access the IFSR by reading or writing CP15 c6 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c5, c0, 1 ; Read Fault Address Register
MCR p15, 0, <Rd>, c5, c0, 1 ; Write Fault Address Register
```

The format of the IFSR is shown in Figure 3-34.

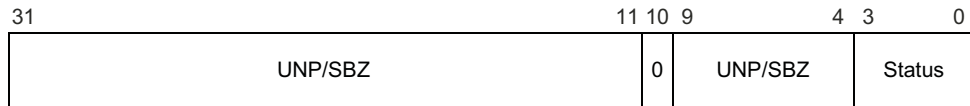


Figure 3-34 IFSR format

Table 3-38 shows the bit fields for the IFSR.

Table 3-38 IFSR bits

Bits	Meaning
[31:11]	UNP/SBZ

Table 3-38 IFSR bits (continued)

Bits	Meaning
[10]	Always 0
[9:4]	UNP/SBZ
[3:0]	Type of fault generated (see <i>Fault status and address</i> on page 6-33)

The encoding of these bits is shown in *Fault status and address* on page 6-33.

Reading CP15 c5 with the Opcode_2 field set to 1 returns the value of the IFSR.

Writing CP15 c5 with the Opcode_2 field set to 1 sets the IFSR to the value of the data written. This is useful for a debugger to restore the value of the IFSR. The register must be written using a read-modify-write sequence. Bits [31:4] Should Be Zero.

3.9.6 Memory Region Remap Registers

The Memory Region Remap registers are:

- Data Memory Remap Register
- Instruction Memory Remap Register
- DMA Memory Remap Register
- Peripheral Port Memory Remap Register, see *Remapping the peripheral port when the MMU is disabled* on page 3-72.

You can use the Memory Region Remap Registers to remap memory region types. The remapping takes place on the outputs of the MMU, and overrides the settings specified in the MMU page tables, or the default behavior when the MMU is disabled.

You can remap both Inner and Outer attributes.

You can access the Memory Region Remap Registers using the following instructions:

```
MRC p15, 0, <Rd>, c15, c2, 0 ; Read the Data Memory Remap Register
MCR p15, 0, <Rd>, c15, c2, 0 ; Write the Data Memory Remap Register
MRC p15, 0, <Rd>, c15, c2, 1 ; Read the Instruction Memory Remap Register
MCR p15, 0, <Rd>, c15, c2, 1 ; Write the Instruction Memory Remap Register
MRC p15, 0, <Rd>, c15, c2, 2 ; Read the DMA Memory Remap Register
MCR p15, 0, <Rd>, c15, c2, 2 ; Write the DMA Memory Remap Register
MRC p15, 0, <Rd>, c15, c2, 4 ; Read Peripheral Port Memory Remap Register
MCR p15, 0, <Rd>, c15, c2, 4 ; Write Peripheral Port Memory Remap Register
```

Each memory region register is split into two parts covering the Inner and Outer attributes respectively. The Inner attributes are covered by five three bit fields, and the Outer attributes are covered by four two bit fields.

The Shared bit can also be remapped. If the Shared bit as read from the TLB or page tables is 0, then it is remapped to bit 15 of this register. If the Shared bit as read from the TLB or page tables is 1, then it is remapped to bit 16 of this register.

The format of the Instruction, Data, and DMA Memory Remap Registers is shown in Figure 3-35.

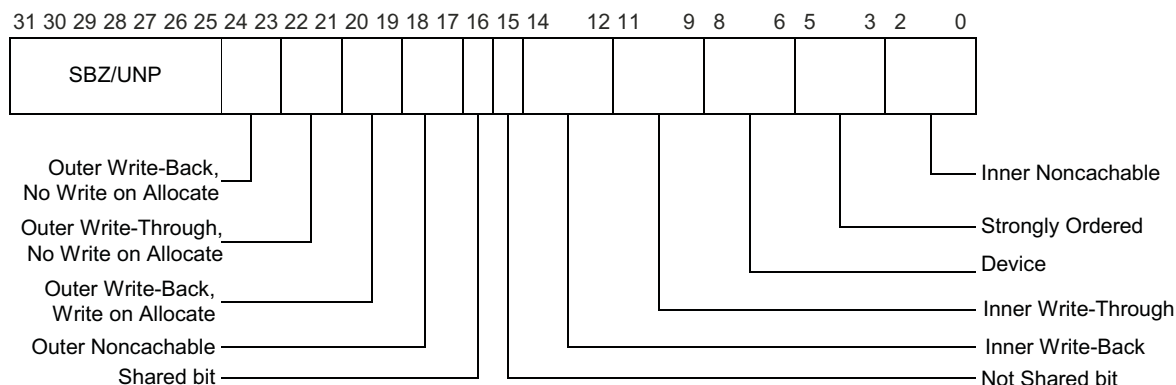


Figure 3-35 Instruction, Data, and DMA Memory Remap Registers format

Table 3-39 shows the functions of the bits in the Instruction, Data, and DMA Memory Remap Registers.

Table 3-39 Memory Region Remap Register fields

Register bits	Remapped region	Reset value
[31:25]	SBZ/UNP	-
[24:23]	Outer Write-Back, No Write on Allocate	b11
[22:21]	Outer Write-Through, No Write on Allocate	b10
[20:19]	Outer Write-Back, Write on Allocate	b01
[18:17]	Outer Noncachable	b00
16	Shared bit	b1
15	Not Shared bit	b0
[14:12]	Inner Write-Back	b111

Table 3-39 Memory Region Remap Register fields (continued)

Register bits	Remapped region	Reset value
[11:9]	Inner Write-Through	b110
[8:6]	Device	b011
[5:3]	Strongly Ordered	b001
[2:0]	Inner Noncachable	b000

The reset value for each field ensures that by default no remapping occurs.

The encoding used for Inner regions is shown in Table 3-40.

Table 3-40 Inner region remap encoding

Inner region	Encoding
Noncachable	b000
Strongly Ordered	b001
Reserved	b010
Device	b011
Reserved	b100
Reserved	b101
Write-Through	b110
Write-Back	b111

The encoding used for Outer regions is shown in Table 3-41.

Table 3-41 Outer region remap encoding

Outer region	Encoding
Noncachable	b00
Write-Back, Write on Allocate	b01
Write-Through, No Write on Allocate	b10
Write-Back, No Write on Allocate	b11

When the MMU is disabled the region type prior to remapping is as shown in Table 3-42.

Table 3-42 Default memory regions when MMU is disabled

Condition	Region type
Data Cache enabled	Data, Strongly Ordered
Data Cache disabled	Data, Strongly Ordered
Instruction Cache enabled	Instruction, Write-Through
Instruction Cache disabled	Instruction, Strongly Ordered

This enables different mappings to be selected with the MMU disabled, that cannot be done using only the I, C, and M bits in CP15 c1.

Remapping the peripheral port when the MMU is disabled

The peripheral port is accessed by memory locations whose page table attributes are Non-Shared Device. You can program a region of memory to be remapped to being Non-Shared Device while the MMU is disabled to provide access to the peripheral port when the MMU is disabled. This mapping only occurs while the MMU is disabled.

If the region of memory-mapped by this mechanism overlaps with the regions of memory that are contained within the TCMs, then the memory locations that are mapped as both TCM and Non-Shared Device are treated as TCM. Therefore, the overlapping region does not access the peripheral port. When the MMU is enabled, the contents of the Peripheral Port Memory Remap Register are ignored.

The peripheral port is only used by data accesses. Unaligned accesses and exclusive accesses are not supported by the peripheral port (because they are not supported in Device memory), and attempting to perform such accesses has Unpredictable results when using the peripheral port with the MMU disabled. Any remapping on Non-Shared Device memory by the Data Memory Remap Register has an effect on regions mapped to Non-Shared Device by the Peripheral Port Memory Remap Register. This enables the peripheral port to be entirely disabled using the Data Memory Remap register.

The format of the Peripheral Port Memory Remap Register is shown in Figure 3-36 on page 3-73.

**Figure 3-36 Peripheral Port Memory Remap Register format**

Table 3-43 shows the functions of the bits in the Peripheral Port Memory Remap Register.

Table 3-43 Peripheral Port Memory Remap Register bit functions

Bits	Field name	Function
[31:12]	Base Address	Gives the virtual base address of the region of memory to be remapped to the peripheral port. Because the Peripheral Port Memory Remap Register is only used while the MMU is disabled, the virtual base address is equal to the physical base address that is used. The Base Address is assumed to be aligned to the size of the remapped region. Any bits in the range $[(\log_2(\text{Region size})-1):12]$ are ignored. The Base Address is 0 at Reset.
[11:5]	UNP/SBZ	-
[4:0]	Size	Indicates the size of the memory region that is to be remapped to be used by the peripheral port. The Size is 0 at Reset, indicating that no remapping is to take place. The encoding of the Size field is shown in Table 3-44.

The encoding of the Size field for different remap region sizes is shown in Table 3-44.

Table 3-44 Size field encoding

Size field	Region Size
b00000	0KB
b00011	4KB
b00100	8KB
b00101	16KB
b00110	32KB

Table 3-44 Size field encoding (continued)

Size field	Region Size
b00111	64KB
b01000	128KB
b01001	256KB
b01010	512KB
b01011	1MB
b01100	2MB
b01101	4MB
b01110	8MB
b01111	16MB
b10000	32MB
b10001	64MB
b10010	128MB
b10011	256MB
b10100	512MB
b10101	1GB
b10110	2GB

3.9.7 TLB Type Register

The TLB has 64 entries organized as a unified two-way set associative TLB. In addition, it has eight lockable entries, as specified by the read-only TLB Type Register.

You can access the TLB Type Register by reading CP15 c0 with the Opcode_2 field set to 3. For example:

```
MRC p15,0,<Rd>,c0,c0,3; returns TLB details
```

The format of the TLB Type Register is shown in Figure 3-37 on page 3-75.

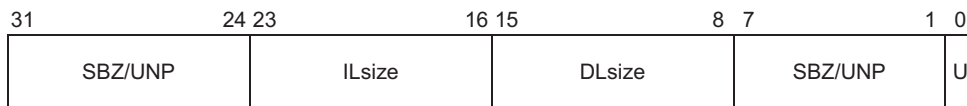


Figure 3-37 TLB Type Register format

TLB Type Register field descriptions are shown in Table 3-45.

Table 3-45 TLB Type Register field descriptions

Bits	Field	Description
[31:24]	SBZ/UNP	-
[23:16]	ILsize	Specifies the number of instruction TLB lockable entries. For ARM1136JF-S processors this is 0.
[15:8]	DLsize	Specifies the number of unified or data TLB lockable entries. For ARM1136JF-S processors this is 8.
[7:1]	SBZ/UNP	-
[0]	U	Specifies if the TLB is unified (0), or if there are separate instruction and data TLBs (1). For ARM1136JF-S processors this is 0.

3.9.8 TLB Operations Register

The TLB Operations Register, CP15 c8, is a write-only register used to manage the *Translation Lookaside Buffer (TLB)*.

The defined TLB operations are listed in Table 3-46. The function to be performed is selected by the Opcode_2 and CRm fields in the MCR instruction used to write CP15 c8. Writing other Opcode_2 or CRm values is Unpredictable.

Reading from CP15 c8 is Unpredictable.

Table 3-46 shows the TLB Operations Register instructions.

Table 3-46 TLB Operations Register instructions

Function	Data	Instruction
Invalidate TLB	SBZ	MCR p15, 0, <Rd>, c8, <CRm>, 0
Invalidate TLB Single Entry	MVA	MCR p15, 0, <Rd>, c8, <CRm>, 1
Invalidate TLB Single Entry on ASID Match	ASID	MCR p15, 0, <Rd>, c8, <CRm>, 2

The CRm value indicates to the hardware what type of access caused the TLB function to be invoked.

Table 3-47 shows the CRm values for the TLB Operations Register, and their meanings. All other CRm values are reserved

Table 3-47 CRm values for TLB Operations Register

CRm	Meaning
c5	Instruction TLB operation
c6	Data TLB operation
c7	Unified TLB operation

Note

The ARM1136JF-S processor has a unified TLB. Any TLB operations specified for the Instruction or Data TLB perform the equivalent operation on the unified TLB.

The Invalidate TLB Single Entry operation uses Virtual Address as an argument. The format is shown in Figure 3-38.

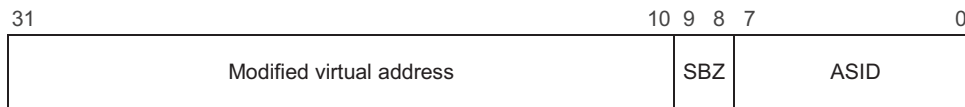


Figure 3-38 TLB Operations Register Virtual Address format

The Invalidate TLB Single Entry on ASID Match function requires an ASID as an argument. The format is shown in Figure 3-39.

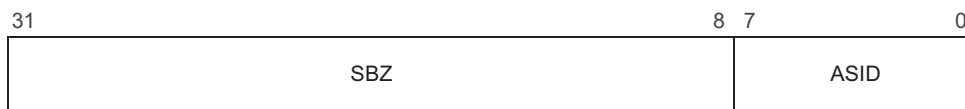


Figure 3-39 TLB Operations Register ASID format

Functions that update the contents of the TLB occur in program order. Therefore, an explicit data access prior to the TLB function uses the old TLB contents, and an explicit data access after the TLB function uses the new TLB contents. For instruction accesses, TLB updates are guaranteed to have taken effect before the next pipeline flush. This includes flush prefetch buffer operations and exception return sequences.

Invalidate TLB

Invalidate TLB invalidates all the unlocked entries in the TLB. This function causes the prefetch buffer to be flushed. Therefore, all following instructions are fetched after the TLB invalidation.

Invalidate TLB Single Entry

You can use Invalidate TLB Single Entry to invalidate an area of memory prior to remapping. You must perform an Invalidate TLB Single Entry of a virtual address in each area to be remapped (section, small page, or large page).

This function invalidates a TLB entry that matches the provided virtual address and ASID, or a global TLB entry that matches the provided VA. This function invalidates a matching locked entry. If the page table VA register Process field selects global entries, then this function has no effect.

Invalidate TLB Single Entry on ASID Match

This is a single interruptible operation that invalidates all TLB entries that match the provided ASID value. This function invalidates locked entries.

In ARM1136JF-S processors, this operation takes several cycles to complete and the instruction is interruptible. When interrupted the r14 state is set to indicate that the MCR instruction has not executed. Therefore, r14 points to the address of the MCR + 4. The interrupt routine then automatically restarts at the MCR instruction.

If this operation is interrupted and later restarted, any entries fetched into the TLB by the interrupt that uses the provided ASID are invalidated by the restarted invalidation.

3.9.9 TLB Lockdown Register

The TLB lockdown register controls where hardware page table walks place the TLB entry, in the set associative region or the lockdown region of the TLB, and if in the lockdown region, which entry is written. The lockdown region of the TLB contains eight entries. See *TLB organization* on page 6-4 for a description of the structure of the TLB.

You can access the TLB lockdown register by reading or writing CP15 c10 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c10, c0, 0 ; Read TLB Lockdown victim
MCR p15, 0, <Rd>, c10, c0, 0 ; Write TLB Lockdown victim
```

Figure 3-40 shows the TLB Lockdown Register format.



Figure 3-40 TLB Lockdown Register format

Writing the TLB Lockdown Register with the preserve bit (P bit) set to:

- 1** Means subsequent hardware page table walks place the TLB entry in the lockdown region at the entry specified by the victim, in the range 0 to 7.
- 0** Means subsequent hardware page table walks place the TLB entry in the set associative region of the TLB.

TLB entries in the lockdown region are preserved so that invalidate TLB operations only invalidate the unreserved entries in the TLB. That is, those in the set-associative region. Invalidate TLB Single Entry operations invalidate any TLB entry corresponding to the modified virtual address given in <Rd>, regardless of their preserved state. That is, if they are in the lockdown or set-associative regions of the TLB. See *TLB Operations Register* on page 3-75 for a description of the TLB invalidate operations.

The victim automatically increments after any table walk that results in an entry being written into the lockdown part of the TLB.

Example 3-2 is a code sequence that locks down an entry to the current victim.

Example 3-2 Lock down an entry to the current victim

```
ADR r1,LockAddr ; set r1 to the value of the address to be locked down
MCR p15,0,r1,c8,c7,1 ; invalidate TLB single entry to ensure that
; LockAddr is not already in the TLB
MRC p15,0,r0,c10,c0,0 ; read the lockdown register
ORR r0,r0,#1 ; set the preserve bit
MCR p15,0,r0,c10,c0,0 ; write to the lockdown register
LDR r1,[r1] ; TLB will miss, and entry will be loaded
MRC p15,0,r0,c10,c0,0 ; read the lockdown register (victim will have
```

```

; incremented)
BIC r0,r0,#1 ; clear preserve bit
MCR p15,0,r0,c10,c0,0 ; write to the lockdown register

```

3.9.10 Translation Table Base Control Register

These bits determine if a page table miss for a specific virtual address must use Translation Table Base Register 0 or Translation Table Base Register 1.

You can access the Translation Table Base Control Register by reading or writing CP15 c2 with the Opcode_2 field set to 2:

```

MRC p15, 0, <Rd>, c2, c0, 2 ; Read Translation Table Base Control Register
MCR p15, 0, <Rd>, c2, c0, 2 ; Write Translation Table Base Control Register

```

Figure 3-41 shows the format of the bits in the Translation Table Base Control Register.

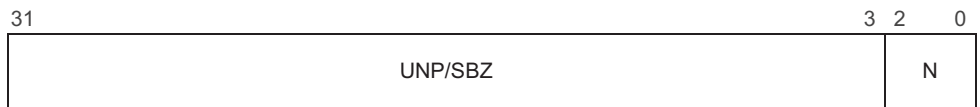


Figure 3-41 Translation Table Base Control Register format

The page table base register is selected as follows:

1. If $N = 0$, always use Translation Table Base Register 0. This is the default case at reset. It is backwards compatible with ARMv5 or earlier processors.
2. If N is greater than 0, then if bits [31:32- N] of the virtual address are all 0, use Translation Table Base Register 0, otherwise use Translation Table Base Register 1. N must be in the range 0-7.

Reading from CP15 c2 returns the size of the page table boundary for Translation Table Base Register 0. Bits [31:3] Should Be Zero.

Writing to CP15 c2 updates the size of the first-level translation table base boundary for Translation Table Base Register 0. Bits [31:3] Should Be Zero.

Table 3-48 shows the values of N for Translation Table Base Register 0.

Table 3-48 Values of N for Translation Table Base Register 0

N	Translation Table Base Register 0 page table boundary size
0	16KB
1	8KB
2	4KB
3	2KB
4	1KB
5	512-byte
6	256-byte
7	128-byte

3.9.11 Translation Table Base Register 0

Use Translation Table Base Register 0 for process-specific addresses, where each process maintains a separate first-level page table. On a context switch you must modify both Translation Table Base Register 0 and the Translation Table Base Control Register, if appropriate.

You can access the Translation Table Base Register 0 by reading or writing CP15 c2 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c2, c0, 0 ; Read Translation Table Base Register 0
MCR p15, 0, <Rd>, c2, c0, 0 ; Write Translation Table Base Register 0
```

Figure 3-42 shows the format of the bits in Translation Table Base Register 0.

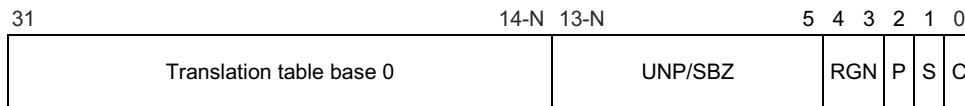


Figure 3-42 Translation Table Base Register 0 format

The functions of the bits in the Translation Table Base Register 0 are shown in Table 3-49.

Table 3-49 Translation Table Base Register 0 bits

Bits	Name	Function
[31:14-N]	Translation table base 0	Pointer to the level one translation table
[13-N:5]	-	UNP/SBZ
[4:3]	RGN	Outer cachable attributes for page table walking: b00 = Outer Noncachable b01 = Outer Cachable Write-Back cached, Write Allocate b10 = Outer Cachable Write-Through, No Allocate on Write b11 = Outer Cachable Write-Back, No Allocate on Write
[2]	P	Indicates to the memory controller that, if supported ECC is (1) enabled or (0) disabled. For ARM1136JF-S processors this bit Should Be Zero.
[1]	S	The page table walk is to Sharable (1) or Non-Shared (0) memory.
[0]	C	The page table walk is Inner Cachable (1) or Inner Noncachable (0).

3.9.12 Translation Table Base Register 1

Use Translation Table Base Register 1 for operating system and I/O addresses.

You can access Translation Table Base Register 1 by reading or writing CP15 c2 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c2, c0, 1      ; Read Translation Table Base Register 1
MCR p15, 0, <Rd>, c2, c0, 1      ; Write Translation Table Base Register 1
```

Figure 3-43 shows the format of the bits in Translation Table Base Register 1.

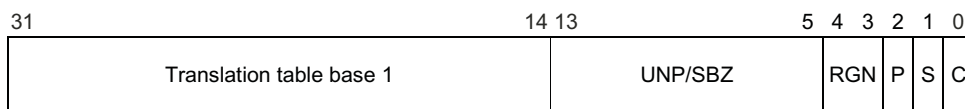


Figure 3-43 Translation Table Base Register 1 format

Writing to CP15 c2 updates the pointer to the first-level translation table from the value in bits [31:14] of the written value. Bits [13:0] Should Be Zero. Translation Table Base Register 1 must reside on a 16KB page boundary.

The functions of the bits in the Translation Table Base Register 1 are shown in Table 3-50.

Table 3-50 Translation Table Base Register 1 bits

Bits	Name	Function
[31:14]	Translation table base 1	Pointer to the level one translation table
[13:5]	-	UNP/SBZ
[4:3]	RGN	Outer cachable attributes for page table walking: b00 = Outer Noncachable b01 = Outer Cachable Write-Back cached, Write Allocate b10 = Outer Cachable Write-Through, No Allocate on Write b11 = Outer Cachable Write-Back, No Allocate on Write
[2]	P	Indicates to the memory controller that, if supported ECC is (1) enabled or (0) disabled. For ARM1136JF-S processors this bit Should Be Zero.
[1]	S	The page table walk is to Sharable (1) or Non-Shared (0) memory.
[0]	C	The page table walk is Inner Cachable (1) or Inner Noncachable (0). All page table accesses are Outer Cachable.

Note

The ARM1136JF-S processor cannot page table walk from level one cache. Therefore, if C = 1, to ensure coherency, you must either store page tables in Inner Write-Through memory or, if in Inner Write-Back, you must clean the appropriate cache entries after modification to ensure that they are seen by the hardware page table walking mechanism.

3.10 TCM configuration and control

ARM1136JF-S TCM configuration and control is provided by:

- *TCM Status Register*
- *Data TCM Region Register*
- *Instruction TCM Region Register* on page 3-85
- *Domain Access Control Register* on page 3-67.

3.10.1 TCM Status Register

Only one Instruction TCM and one Data TCM is implemented in the ARM1136JF-S processor.

You can access the TCM Status Register by reading CP15 c0 with the Opcode_2 field set to 2. For example:

```
MRC p15,0,<Rd>,c0,c0,2; returns TCM status register
```

The format of the TCM Status Register is shown in Figure 3-44.

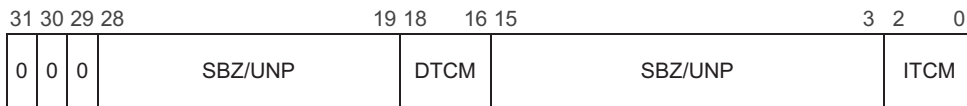


Figure 3-44 TCM Status Register format

ITCM Specifies the number of Instruction TCMs implemented. For ARM1136JF-S processors this value is 1.

DTCM Specifies the number of Data TCMs implemented. For ARM1136JF-S processors this value is 1.

3.10.2 Data TCM Region Register

ARM1136JF-S processors have a single TCM on each side. The Data TCM has its own region register that describes the physical base address and size of it, and controls its enabling and mode of operation.

The Data TCM Region Register is accessible only in a privileged mode of operation. You can access the Data TCM Region Register by reading or writing CP15 c9 with the CRm field set to c1 and the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c9, c1, 0 ; Read Data TCM Region Register
MCR p15, 0, <Rd>, c9, c1, 0 ; Write Data TCM Region Register
```

Changing the Data TCM Region Register while a Prefetch Range or DMA operation is running has Unpredictable effects.

The format of the Data TCM Region Register is shown in Figure 3-46 on page 3-85.

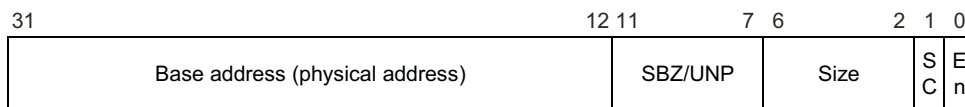


Figure 3-45 Data TCM Region Register format

The meanings of the bit fields in the Data TCM Region Register are shown in Table 3-51.

Table 3-51 Data TCM Region Register bits

Bits	Meaning
[31:12]	This is the physical base address of the TCM. The base address must be aligned to the size of the TCM. Any bits in the range $[(\log_2(\text{RAMSize})-1):12]$ are ignored. The base address is 0 at Reset.
[11:7]	Should Be Zero or Unpredictable.
[6:2]	On reads, the Size field indicates the size of the TCM. On writes this field is ignored. See <i>Tightly-coupled memory</i> on page 7-8.
[1]	The SC bit indicates if the TCM is enabled as SmartCache: 0 = Local RAM (state on Reset) 1 = SmartCache.
[0]	The En bit indicates if the TCM is enabled: 0 = TCM disabled (state on Reset) 1 = TCM enabled.

The encoding of the values of the Size field are shown in Table 3-52 on page 3-86. All unused values are reserved.

Table 3-52 Size field encoding

Size field	Memory size
b00000	0KB
b00011	4KB
b00100	8KB
b00101	16KB
b00110	32KB
b00111	64KB

3.10.3 Instruction TCM Region Register

ARM1136JF-S processors have a single TCM on each side. The Instruction TCM has its own region register that describes the physical base address and size of it, and controls its enabling and mode of operation.

The Instruction TCM Region Register is accessible only in a privileged mode of operation. You can access the Instruction TCM Region Register by reading or writing CP15 c9 with the CRm field set to c1 and the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c9, c1, 1 ; Read Instruction TCM Region Register
MCR p15, 0, <Rd>, c9, c1, 1 ; Write Instruction TCM Region Register
```

Changing the Instruction TCM Region Register while a Prefetch Range or DMA operation is running has Unpredictable effects.

The format of the Instruction TCM Region Register is shown in Figure 3-46.

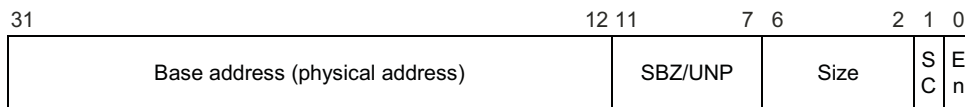


Figure 3-46 Instruction TCM Region Register format

The meanings of the bit fields in the Instruction TCM Region Register are shown in Table 3-53.

Table 3-53 Instruction TCM Region Register bits

Bits	Meaning
[31:12]	This is the physical base address of the TCM. The base address must be aligned to the size of the TCM. Any bits in the range $[(\log_2(\text{RAMSize})-1):12]$ are ignored. The base address is 0 at Reset.
[11:7]	Should Be Zero or Unpredictable.
[6:2]	On reads, the Size field indicates the size of the TCM. On writes this field is ignored. See <i>Tightly-coupled memory</i> on page 7-8.
[1]	The SC bit indicates if the TCM is enabled as SmartCache: 0 = Local RAM (state on Reset) 1 = SmartCache.
[0]	The En bit indicates if the TCM is enabled: 0 = TCM disabled 1 = TCM enabled. The value of the TCM enable bit is determined on Reset by the pin INITRAM .

The encoding of the values of the Size field are shown in Table 3-54. All unused values are reserved.

Table 3-54 Size field encoding

Size field	Memory size
b00000	0KB
b00011	4KB
b00100	8KB
b00101	16KB
b00110	32KB
b00111	64KB

3.11 System performance monitoring

System performance monitoring uses a series of system events, such as cache misses, TLB misses, pipeline stalls, and other related features to enable system developers to profile the performance of their systems. It is implemented as part of CP15.

ARM1136JF-S system performance monitoring is provided by four registers, mapped into CP15 c12:

- *Performance Monitor Control Register (PMNC)*
- *Count Register 0, PMN0* on page 3-91
- *Count Register 1, PMN1* on page 3-91
- *Cycle Counter Register, CCNT* on page 3-92.

3.11.1 Performance Monitor Control Register (PMNC)

The Performance Monitor Control Register controls the operation of the Count Register 0 (PMN0), Count Register 1 (PMN1), and Cycle Counter Register (CCNT). This register:

- controls which events PMN0 and PMN1 monitor
- detects which counter overflowed
- enables and disables interrupt reporting
- extends CCNT counting by six more bits (cycles between counter rollover = 2^{38})
- resets all counters to zero
- enables the entire performance monitoring mechanism.

You can access the Performance Monitor Control Register by reading or writing CP15 c12 with the Opcode_2 field set to 0:

MRC p15, 0, <Rd>, c15, c12, 0 ; Read Performance Monitor Control Register
MCR p15, 0, <Rd>, c15, c12, 0 ; Write Performance Monitor Control Register

The format of the Performance Monitor Control Register is shown in Figure 3-47.

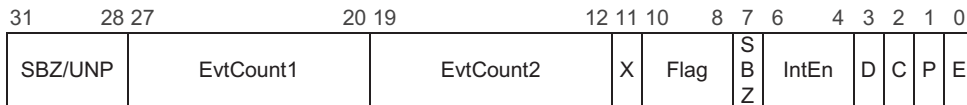


Figure 3-47 Performance Monitor Control Register format

Table 3-55 shows the functions of the bit fields in the Performance Monitor Control Register.

Table 3-55 Performance Monitor Control Register bit functions

Bits	Name	Function
[27:20]	EvtCount1	Identifies the source of events for Count Register 0, as defined in Table 3-56 on page 3-89.
[19:12]	EvtCount2	Identifies the source of events for Count Register 1, as defined in Table 3-56 on page 3-89.
[11]	X	Enable Export of the events to the event bus. This enables an external monitoring block, such as the ETM to trace events: 0 = Export disabled, EVNTBUS held at 0x0 1 = Export enabled, EVNTBUS driven by the events.
[10:8]	Flag	Overflow/Interrupt Flag. Identifies which counter overflowed: Bit 10 = Cycle Counter Register overflow flag Bit 9 = Count Register 1 overflow flag Bit 8 = Count Register 0 overflow flag. For reads: 0 = no overflow (reset) 1 = overflow has occurred. For writes: 0 = no effect 1 = clear this bit.
[6:4]	IntEn	Interrupt Enable. Used to enable and disable interrupt reporting for each counter: Bit 6 = Cycle Counter interrupt enable Bit 5 = Count Register 1 interrupt enable Bit 4 = Count Register 0 interrupt enable. For these registers: 0 = disable interrupt 1 = enable interrupt.
[3]	D	Cycle count divider: 0 = Cycle Counter Register counts every processor clock cycle 1 = Cycle Counter Register counts every 64th processor clock cycle.

Table 3-55 Performance Monitor Control Register bit functions (continued)

Bits	Name	Function
[2]	C	Cycle Counter Register Reset on Write, UNP on Read: 0 = no action 1 = reset the Cycle Counter Register to 0x0.
[1]	P	Count Register Reset on Write, UNP on Read: 0 = no action 1 = reset both Count Registers to 0x0.
[0]	E	Enable: 0 = all three counters disabled 1 = all three counters enabled.

If an interrupt is generated by this unit, the ARM1136JF-S processor pin **PMUIRQ** is asserted. This output pin can then be routed to an external interrupt controller for prioritization and masking. This is the only mechanism by which the interrupt is signaled to the core.

There is a delay of three cycles between enabling the counter and the counter starting to count events. In addition, the information used to count events is taken from various pipeline stages. This means that the absolute counts recorded might vary because of pipeline effects. This has a negligible effect except in case where the counters are enabled for a very short time.

The events that can be monitored are shown in Table 3-56.

Table 3-56 Performance monitoring events

Event number	EVNTBUS bit position	Event definition
0x0	[0]	Instruction cache miss to a cachable location requires fetch from external memory.
0x1	[1]	Stall because instruction buffer cannot deliver an instruction. This could indicate an Instruction Cache miss or an Instruction MicroTLB miss. This event occurs every cycle in which the condition is present.
0x2	[2]	Stall because of a data dependency. This event occurs every cycle in which the condition is present.
0x3	[3]	Instruction MicroTLB miss.
0x4	[4]	Data MicroTLB miss.

Table 3-56 Performance monitoring events (continued)

Event number	EVNTBUS bit position	Event definition
0x5	[6:5]	Branch instruction executed, branch might or might not have changed program flow.
0x6	[7]	Branch mispredicted.
0x7	[9:8]	Instruction executed.
0x9	[10]	Data cache access, not including Cache operations. This event occurs for each nonsequential access to a cache line, for cachable locations.
0xA	[11]	Data cache access, not including Cache Operations. This event occurs for each nonsequential access to a cache line, regardless of whether or not the location is cachable.
0xB	[12]	Data cache miss, not including Cache Operations.
0xC	[13]	Data cache Write-Back. This event occurs once for each half line of four words that are written back from the cache.
0xD	[15:14]	Software changed the PC. This event occurs any time the PC is changed by software and there is not a mode change. For example, a MOV instruction with PC as the destination triggers this event. Executing a SWI from User mode does not trigger this event, because it incurs a mode change.
0xF	[16]	Main TLB miss.
0x10	[17]	External memory request (Cache Refill, Noncachable, Write-Through, Write-Back).
0x11	[18]	Stall because of Load Store Unit request queue being full. This event takes place each clock cycle in which the condition is met. A high incidence of this event indicates the BCU is often waiting for transactions to complete on the external bus.
0x12	[19]	The number of times the Write Buffer was drained because of a Drain Write Buffer command or Strongly Ordered operation.
0x20	-	ETMEXTOUT[0] signal was asserted for a cycle.
0x21	-	ETMEXTOUT[1] signal was asserted for a cycle.

Table 3-56 Performance monitoring events (continued)

Event number	EVNTBUS bit position	Event definition
0x22	-	If both ETMEXTOUT[0] and ETMEXTOUT[1] signals are asserted then the count is incremented by two.
0xFF	-	An increment each cycle.
All other values	-	Reserved. Unpredictable behavior.

In addition to the two counters within ARM1136JF-S processors, each of the events shown in Table 3-56 on page 3-89 is available on an external bus, **EVNTBUS**. You can connect this bus to the ETM unit or other external trace hardware to enable the events to be monitored. If this functionality is not required, you must set X bit in the Performance Monitor Control Register to the 0.

3.11.2 Count Register 0, PMN0

You can use the two counter registers, Count Register 0 and Count Register 1, to count the instances of two different events selected from a list of events by the Performance Monitor Control Register. Each counter is a 32-bit counter that can trigger an interrupt on overflow. By combining different statistics you can obtain a variety of useful metrics that enable you to optimize system performance.

You can access Count Register 0 by reading or writing CP15 c15 with the Opcode_2 field set to 2:

```
MRC p15, 0, <Rd>, c15, c12, 2 ; Read Count Register 0
MCR p15, 0, <Rd>, c15, c12, 2 ; Write Count Register 0
```

The value in Count Register 0 is 0 at Reset.

3.11.3 Count Register 1, PMN1

You can use the two counter registers, Count Register 0 and Count Register 1, to count the instances of two different events selected from a list of events by the Performance Monitor Control Register. Each counter is a 32-bit counter that can trigger an interrupt on overflow. By combining different statistics you can obtain a variety of useful metrics that enable you to optimize system performance.

You can access Count Register 1 by reading or writing CP15 c12 with the Opcode_2 field set to 3:

```
MRC p15, 0, <Rd>, c15, c12, 3 ; Read Count Register 1  
MCR p15, 0, <Rd>, c15, c12, 3 ; Write Count Register 1
```

The value in Count Register 1 is 0 at Reset.

3.11.4 Cycle Counter Register, CCNT

You can use the Cycle Counter Register to count the core clock cycles. It is a 32-bit counter that can trigger an interrupt on overflow. You can use it in conjunction with the Performance Monitor Control Register and the two Counter Registers to provide a variety of useful metrics that enable you to optimize system performance.

You can access the Cycle Counter Register by reading or writing CP15 c12 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c15, c12, 1 ; Read Cycle Counter Register  
MCR p15, 0, <Rd>, c15, c12, 1 ; Write Cycle Counter Register
```

The value in the Cycle Counter Register is Unpredictable at Reset. The counter can be set to zero by the Performance Monitor Control Register.

The Cycle Counter Register can be configured to count every 64th clock cycle by the Performance Monitor Control Register.

3.12 Overall system configuration and control

The overall system configuration and control of the ARM1136JF-S processor is provided by:

- *Auxiliary Control Register*
- *Coprocessor Access Control Register* on page 3-94
- *Context ID Register* on page 3-95
- *Control Register* on page 3-96
- *FCSE PID Register* on page 3-100
- *ID Code Register* on page 3-102.

3.12.1 Auxiliary Control Register

You can use the Auxiliary Control Register to enable and disable program flow prediction operations. It is selected by reading or writing CP15 c1 with the Opcode_2 field set to 1:

```
MRC p15,0,<Rd>,c1,c0,1      ; Read Auxiliary Control Register
MCR p15,0,<Rd>,c1,c0,1      ; Write Auxiliary Control Register
```

The format of the Auxiliary Control Register is shown in Figure 3-48.

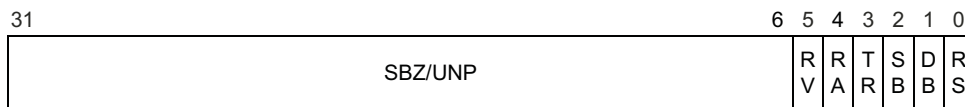


Figure 3-48 Auxiliary Control Register format

The functions of the Auxiliary Control Register bits are shown in Table 3-57.

Table 3-57 Auxiliary Control Register bit functions

Bits	Name	Function
[31:6]	-	Reserved. These bits must be updated using a read-modify-write technique to ensure that currently unallocated bits are not unnecessarily modified.
[5]	RV	Disable block transfer cache operations.
[4]	RA	Disable clean entire data cache.

Table 3-57 Auxiliary Control Register bit functions (continued)

Bits	Name	Function
[3]	TR	MicroTLB random replacement. This bit selects Random replacement for the MicroTLBs if the caches are configured to have Random replacement (using CP15 c1 RR bit). 0 = MicroTLB replacement is Round Robin 1 = MicroTLB replacement is Random if cache replacement is also Random. This bit is cleared on reset.
[2]	SB	Static branch prediction enable. This bit enables the use of static branch prediction if program flow prediction is enabled. See CP15, Control Register. 0 = Static branch prediction is disabled 1 = Static branch prediction is enabled. This bit is set on reset.
[1]	DB	Dynamic branch prediction enable. This bit enables the use of dynamic branch prediction if program flow prediction is enabled. See CP15, Control Register. 0 = Dynamic branch prediction is disabled 1 = Dynamic branch prediction is enabled. This bit is set on reset.
[0]	RS	Return stack enable. This bit enables the use of the return stack if program flow prediction is enabled. See CP15, Control Register. 0 = Return stack is disabled 1 = Return stack is enabled. This bit is set on reset.

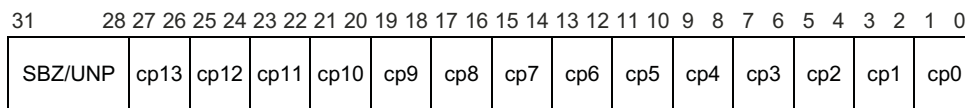
3.12.2 Coprocessor Access Control Register

The Coprocessor Access Control Register controls accesses to all coprocessors other than CP14 and CP15. You can access the Coprocessor Access Control Register by reading or writing CP15 c1 with the Opcode_2 field set to 2:

MRC p15,0,<Rd>,c1,c0,2; Read Coprocessor Access Control Register

MCR p15,0,<Rd>,c1,c0,2; Write Coprocessor Access Control Register

Figure 3-49 shows the format of the Coprocessor Access Control Register.

**Figure 3-49 Coprocessor Access Control Register format**

Each pair of bits corresponds to the access rights for each coprocessor. These are as shown in Table 3-58.

Table 3-58 Coprocessor access rights

Bits	Meaning
b00	Access denied. Attempts to access the corresponding coprocessor generate an Undefined exception.
b01	Supervisor access only.
b10	Reserved.
b11	Full access.

After updating this register you must execute an *Instruction Memory Barrier (IMB)* sequence. None of the instructions executed after changing this register and before the IMB must be coprocessor instructions affected by the change in coprocessor access rights.

After a system reset, all coprocessor access rights are set to Access denied.

If a coprocessor is not implemented then attempting to write the coprocessor access right bits for that entry to values other than b00 has no effect. This mechanism can be used by software to determine which coprocessors are present.

3.12.3 Context ID Register

CP15 c13 accesses the Process Identifier Registers:

- Context ID Register
- *FCSE PID Register* on page 3-100.

You can access the Context ID Register by reading or writing CP15 c13 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c13, c0, 1 ; Read Context ID Register
MCR p15, 0, <Rd>, c13, c0, 1 ; Write Context ID Register
```

The format of the Context ID Register is shown in Figure 3-50 on page 3-96.



Figure 3-50 Context ID Register format

The bottom eight bits of the Context ID Register are used for the current ASID that is running. The top bits extend the ASID. The current ASID value in use is exported to the MMU the core bus, **COREASID [7:0]**. To ensure that all accesses are related to the correct context ID, you must ensure that software executes a drain write buffer operation before changing this register.

The whole of this register is used by both the *Embedded Trace Macrocell* (ETM) and by the debug logic. Its value can be broadcast by the ETM to indicate the currently running process. You must program each process with a unique number. Therefore, if an ASID is reused, the ETM can distinguish between processes. It is used by ETM to determine how virtual to physically memory is mapped.

Its value can also be used to enable process-dependent breakpoints and watchpoints. After changing this register, an IMB sequence must be executed before any instructions are executed that are from an ASID-dependent memory region. Code that updates the ASID must be executed from a global memory region.

3.12.4 Control Register

You can use the Control Register to enable and disable system configuration options. You can access the Control Register by reading or writing CP15 c1 with the CRm and Opcode_2 fields set to 0:

```
MRC p15,0,<Rd>,c1,c0,0 ; Read Control Register configuration data
MCR p15,0,<Rd>,c1,c0,0 ; Write Control Register configuration data
```

It is recommended that you access this register using a read-modify-write sequence.

All defined control bits are set to zero on Reset except:

- the V bit that is set to zero at Reset if the **VINITHI** signal is LOW, or one if the **VINITHI** signal is HIGH

- the B bit, U bit, and EE bit are set according to the state of the **BIGENDINIT** and **UBITINIT** inputs shown in Table 3-59.

Table 3-59 B bit, U bit, and EE bit settings

BIGENDINIT	UBITINIT	B	U	EE
0	0	0	0	0
0	1	0	1	0
1	0	1	0	0
1	1	0	1	1

Figure 3-51 shows the format of the Control Register.

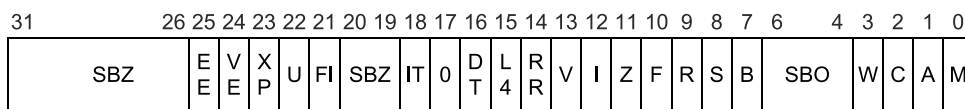
**Figure 3-51 Control Register format**

Table 3-60 describes the functions of the Control Register bits.

Table 3-60 Control Register bit functions

Bit	Name	Function
[31:26]	-	Reserved. When read returns an Unpredictable value. When written Should Be Zero, or a value read from bits [31:26] on the same processor. Using a read-modify-write sequence when modifying this register provides the greatest future compatibility.
[25]	EE bit	This bit determines the setting of the CPSR E bit on taking an exception: 0 = CPSR E bit is set to 0 on taking an exception 1 = CPSR E bit is set to 1 on taking an exception.
[24]	VE bit	Configure vectored interrupt. Enables the VIC interface to determine the interrupt vectors: 0 = Interrupt vectors are fixed. See the description of the V bit (bit 13) 1 = Interrupt vectors are defined by the VIC interface.

Table 3-60 Control Register bit functions (continued)

Bit	Name	Function
[23]	XP bit	Configure extended page table configuration. This bit configures the hardware page translation mechanism: 0 = Subpage AP bits enabled 1 = Subpage AP bits disabled.
[22]	U bit	This bit enables unaligned data access operation, including support for mixed little-endian and big-endian data.
[21]	FI bit	Configure fast interrupt configuration. This bit enables low interrupt latency features: 0 = All performance features enabled 1 = Low interrupt latency configuration enabled. See <i>Low interrupt latency configuration</i> on page 2-28.
[20:19]	-	SBZ. When read returns an Unpredictable value. When written Should Be Zero.
[18]	IT bit	Global Instruction TCM enable/disable bit. This bit is used in ARM946 and ARM966 processors to enable the Instruction TCM. In ARMv6, the TCM blocks have individual enables that apply to each block. As a result, this bit is now redundant and Should Be One. See <i>Instruction TCM Region Register</i> on page 3-85 for a description of the ARM1136JF-S TCM enables.
[17]	-	SBZ. When read returns an Unpredictable value. When written Should Be Zero.
[16]	DT bit	Global Data TCM enable/disable bit. This bit is used in ARM946 and ARM966 processors to enable the Data TCM. In ARMv6, the TCM blocks have individual enables that apply to each block. As a result, this bit is now redundant and Should Be One. See <i>Instruction TCM Region Register</i> on page 3-85 for a description of the ARM1136JF-S TCM enables.
[15]	L4 bit	Configure if load instructions to PC set T bit: 0 = Loads to PC set the T bit 1 = Loads to PC do not set the T bit (ARMv4 behavior). For more details see the <i>ARM Architecture Reference Manual</i> .
[14]	RR bit	Replacement strategy for the Instruction and Data Caches: 0 = Normal replacement strategy (Random replacement) 1 = Predictable replacement strategy (Round-Robin replacement).
[13]	V bit	Location of exception vectors: 0 = Normal exception vectors selected, address range = 0x00000000-0x0000001C 1 = High exception vectors selected, address range = 0xFFFF0000-0xFFFF001C.
[12]	I bit	Level one Instruction Cache enable/disable: 0 = Instruction Cache disabled 1 = Instruction Cache enabled.

Table 3-60 Control Register bit functions (continued)

Bit	Name	Function
[11]	Z bit	<p>Program flow prediction: 0 = Program flow prediction disabled 1 = Program flow prediction enabled.</p> <p>Program flow prediction includes static and dynamic branch prediction and the return stack. This bit enables all three forms of program flow prediction. You can enable or disable each form individually.</p> <p>See <i>Auxiliary Control Register</i> on page 3-93.</p>
[10]	F bit	The meaning of this bit is implementation-defined. This bit Should Be Zero for ARM1136JF-S processors.
[9]	R bit	<p>ROM protection. This bit modifies the ROM protection system: 0 = ROM protection disabled 1 = ROM protection enabled.</p> <p>Modifying the R bit does not affect the access permissions of entries already in the TLB.</p> <p>See <i>MMU software-accessible registers</i> on page 6-55.</p>
[8]	S bit	<p>System protection. This bit modifies the MMU protection system: 0 = MMU protection disabled 1 = MMU protection enabled.</p> <p>Modifying the S bit does not affect the access permissions of entries already in the TLB.</p>
[7]	B bit	<p>This bit configures the ARM1136JF-S processor to rename the low four-byte addresses within a 32-bit word: 0 = Little-endian memory system 1 = Big-endian word-invariant memory system.</p>
[6:4]	-	When read returns one and when written Should Be One.
[3]	W bit	Write buffer enable/disable. Not implemented in the ARM1136JF-S processor because all memory writes take place through the write buffer. This bit reads as 1 and ignores writes.
[2]	C bit	<p>Level one Data Cache enable/disable: 0 = Data cache disabled 1 = Data cache enabled.</p>
[1]	A bit	<p>Strict data address alignment fault enable/disable: 0 = Strict alignment fault checking disabled 1 = Strict alignment fault checking enabled.</p> <p>The A bit setting takes priority over the U bit. The Data Abort trap is taken if strict alignment is enabled and the data access is not aligned to the width of the accessed data item.</p>

Table 3-60 Control Register bit functions (continued)

Bit	Name	Function
[0]	M bit	MMU enable/disable: 0 = MMU disabled 1 = MMU enabled.

Take care with the address mapping of the code sequence used to enable the MMU (see *Enabling the MMU* on page 6-9). See *Disabling the MMU* on page 6-9 for restrictions and effects of having caches enabled with the MMU disabled.

3.12.5 FCSE PID Register

The use of the FCSE PID Register is deprecated.

You can access the FCSE PID Register by reading or writing CP15 c13 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c13, c0, 0 ; Read FCSE PID Register
MCR p15, 0, <Rd>, c13, c0, 0 ; Write FCSE PID Register
```

Reading from the FCSE PID Register returns the value of the process identifier.

Writing the FCSE PID Register updates the process identifier to the value in bits [31:25]. Bits [24:0] Should Be Zero as shown in Figure 3-52.

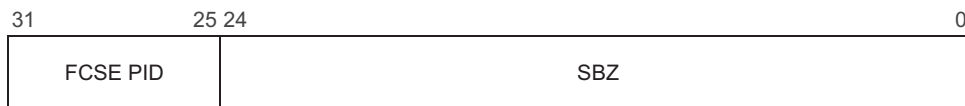


Figure 3-52 FCSE PID Register format

Addresses issued by the ARM1136JF-S processor in the range 0-32MB are translated by the ProcID. Address A becomes A + (ProcID x 32MB). This translated address is used by the MMU. Addresses above 32MB are not translated. This is shown in Figure 3-53 on page 3-101. The ProcID is a seven-bit field, enabling 64 x 32MB processes to be mapped.

———— **Note** —————

If ProcID is 0, as it is on Reset, then there is a flat mapping between the ARM1136JF-S processor and the MMU.

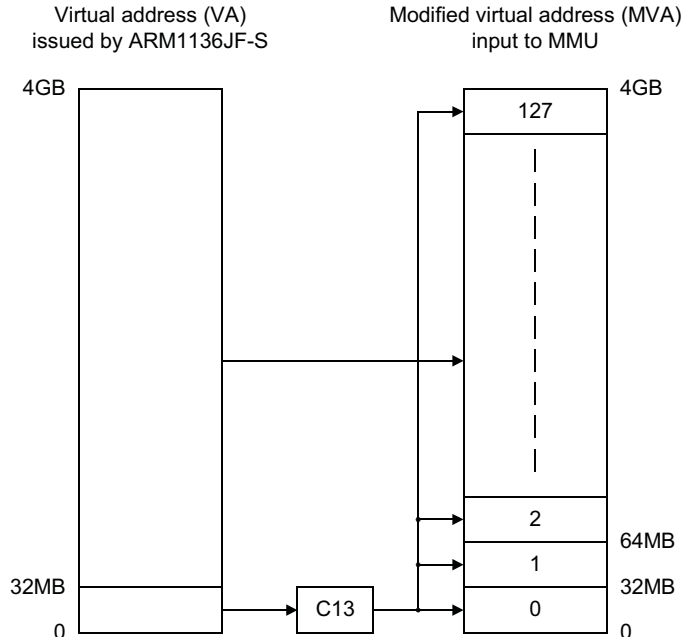


Figure 3-53 Address mapping using CP15 c13

Changing the ProcID, performing a fast context switch

A fast context switch is performed by writing to CP15 c13 FCSE PID Register. The contents of the TLBs do not have to be flushed after a fast context switch because they still hold valid address tags.

From zero to six instructions after the MCR used to write the ProcID might have been fetched with the old ProcID:

```
{ProcID = 0}
MOV r0, #1           ; Fetched with ProcID = 0
MCR p15,0,r0,c13,c0,0 ; Fetched with ProcID = 0
A0 (any instruction) ; Fetched with ProcID = 0/1
A1 (any instruction) ; Fetched with ProcID = 0/1
A2 (any instruction) ; Fetched with ProcID = 0/1
A3 (any instruction) ; Fetched with ProcID = 0/1
A4 (any instruction) ; Fetched with ProcID = 0/1
A5 (any instruction) ; Fetched with ProcID = 0/1
A6 (any instruction) ; Fetched with ProcID = 1
```

You must not rely on this behavior for future compatibility. An IMB must be executed between changing the ProcID and fetching from locations that are transmitted by the ProcID.

3.12.6 ID Code Register

This is a read-only register that returns a 32-bit device ID code.

You can access the ID Code Register by reading CP15 c0 with the Opcode_2 field set to any value other than 1, 2, or 3 (the CRm field Should Be Zero when reading). For example:

```
MRC p15,0,<Rd>,c0,c0,0; returns ID code register
```

The contents of the ID Code Register are shown in Table 3-61.

Table 3-61 Register 0, ID Code

Register bits	Function	Value
[31:24]	Implementor	0x41
[23:20]	Specification revision	0x0
[19:16]	Architecture (ARMv6)	0x7
[15:4]	Part number	0xB36
[3:0]	Layout revision	Revision

Chapter 4

Unaligned and Mixed-Endian Data Access Support

This chapter describes the unaligned and mixed-endianness data access support for the ARM1136JF-S processor. It contains the following sections:

- *About unaligned and mixed-endian support* on page 4-2
- *Unaligned access support* on page 4-3
- *Unaligned data access specification* on page 4-7
- *Operation of unaligned accesses* on page 4-18
- *Mixed-endian access support* on page 4-22
- *Instructions to reverse bytes in a general-purpose register* on page 4-26
- *Instructions to change the CPSR E bit* on page 4-27.

4.1 About unaligned and mixed-endian support

The ARM1136JF-S processor executes the ARM architecture v6 instructions that support mixed-endian access in hardware, and assist unaligned data accesses. The extensions to ARMv6 that support unaligned and mixed-endian accesses include the following:

- CP15 Register c1 has a U bit that enables unaligned support. This bit was specified as zero in previous architectures, and resets to zero for legacy-mode compatibility.
- Architecturally defined unaligned word and halfword access specification for hardware implementation.
- Byte reverse instructions that operate on general-purpose register contents to support signed/unsigned halfword data values.
- Separate instruction and data endianness, with instructions fixed as little-endian format, naturally aligned, but with legacy support for 32-bit word-invariant binary images and ROM.
- A PSR endian control flag, the E-bit, cleared on reset and exception entry, that adds a byte-reverse operation to the entire load and store instruction space as data is loaded into and stored back out of the register file. In previous architectures this Program Status Register bit was specified as zero. It is not set in legacy code written to conform to architectures prior to ARMv6.
- ARM and Thumb instructions to set and clear the E-bit explicitly.
- A byte-invariant addressing scheme to support fine-grain big-endian and little-endian shared data structures, to conform to a shared memory standard.

The original ARM architecture was designed as little-endian. This provides a consistent address ordering of bits, bytes, words, cache lines, and pages, and is assumed by the documentation of instruction set encoding and memory and register bit significance. Subsequently, big-endian support was added to enable big-endian byte addressing of memory. A little-endian nomenclature is used for bit-ordering and byte addressing throughout this manual.

4.2 Unaligned access support

Instructions must always be aligned as follows:

- ARM 32-bit instructions must be word boundary aligned (Address [1:0] = b00)
- Thumb 16-bit instructions must be halfword boundary aligned (Address [0] = 0).

Unaligned data access support is described in:

- *Legacy support*
- *ARMv6 extensions*
- *Legacy and ARMv6 configurations* on page 4-4
- *Legacy data access in ARMv6 (U=0)* on page 4-4
- *Support for unaligned data access in ARMv6 (U=1)* on page 4-5
- *ARMv6 unaligned data access restrictions* on page 4-5.

4.2.1 Legacy support

For ARM architectures prior to ARM architecture v6, data access to non-aligned word and halfword data was treated as aligned from the memory interface perspective. That is, the address is treated as truncated with Address[1:0], treated as zero for word accesses, and Address[0] treated as zero for halfword accesses.

Load single word ARM instructions are also architecturally defined to rotate right the word aligned data transferred by a non word-aligned access, see the *ARM Architecture Reference Manual*.

Alignment fault checking is specified for processors with architecturally compliant *Memory Management Units* (MMUs), under control of CP15 Register c1 A control bit, bit 1. When a transfer is not naturally aligned to the size of data transferred a Data Abort is signaled with an Alignment fault status code, see *ARM Architecture Reference Manual* for more details.

4.2.2 ARMv6 extensions

ARMv6 adds unaligned word and halfword load and store data access support. When enabled, one or more memory accesses are used to generate the required transfer of adjacent bytes transparently, apart from a potentially greater access time where the transaction crosses a word-boundary.

The memory management specification defines a programmable mechanism to enable unaligned access support. This is controlled and programmed using the CP15 Register c1 U control bit, bit 22.

Non word-aligned for load and store multiple/double, semaphore, synchronization, and coprocessor accesses always signal Data Abort with Alignment Faults Status Code when the U bit is set.

Strict alignment checking is also supported in ARMv6, under control of the CP15 Register c1 A control bit (bit 1) and signals a Data Abort with Alignment Fault Status Code if a 16-bit access is not halfword aligned or a single 32-bit load/store transfer is not word aligned.

ARMv6 alignment fault detection is a mandatory function associated with address generation rather than optionally supported in external memory management hardware.

4.2.3 Legacy and ARMv6 configurations

The unaligned access handling is summarized in Table 4-1.

Table 4-1 Unaligned access handling

CP15 register c1 U bit	CP15 register c1 A bit	Unaligned access model
0	0	Legacy ARMv5. See <i>Legacy data access in ARMv6 (U=0)</i> .
0	1	Legacy natural alignment check.
1	0	ARMv6 unaligned half/word access, else strict word alignment check.
1	1	ARMv6 strict half/word alignment check.

For a fuller description of the options available, see *Control Register* on page 3-96.

4.2.4 Legacy data access in ARMv6 (U=0)

The ARM1136JF-S processor emulates earlier architecture unaligned accesses to memory as follows:

- If A bit is asserted alignment faults occur for:
 - Halfword access** Address[0] is 1.
 - Word access** Address[1:0] is not b00.
 - LDRD or STRD** Address [2:0] is not b000.
 - Multiple access** Address [1:0] is not b00.

- If alignment faults are enabled and the access is not aligned then the Data Abort vector is entered with an Alignment Fault status code.
- If no alignment fault is enabled, that is, if bit 1 of CP15 Register c1, the A bit, is not set:

Byte access	Memory interface uses full Address [31:0].
Halfword access	Memory interface uses Address [31:1]. Address [0] asserted as 0.
Word access	Memory interface uses Address [31:2]. Address [1:0] asserted as 0.

 - ARM load data rotates the aligned read data and rotates this right by the byte-offset denoted by Address [1:0], see the *ARM Architecture Reference Manual*.
 - ARM and Thumb load-multiple accesses always treated as aligned. No rotation of read data.
 - ARM and Thumb store word and store multiple treated as aligned. No rotation of write data.
 - ARM load and store doubleword operations treated as 64-bit aligned.
 - Thumb load word data operations are Unpredictable if not word aligned.
 - ARM and Thumb halfword data accesses are Unpredictable if not halfword aligned.

4.2.5 Support for unaligned data access in ARMv6 (U=1)

The ARM1136JF-S processor memory interfaces can generate unaligned low order byte address offsets only for halfword and single word load and store operations, and byte accesses unless the A bit is set. These accesses produce an alignment fault if the A bit is set, and for some of the cases described in *ARMv6 unaligned data access restrictions*.

If alignment faults are enabled and the access is not aligned then the Data Abort vector is entered with an Alignment Fault status code.

4.2.6 ARMv6 unaligned data access restrictions

The following restrictions apply for ARMv6 unaligned data access:

- Accesses are not guaranteed atomic. They might be synthesized out of a series of aligned operations in a shared memory system without guaranteeing locked transaction cycles.
- Unaligned accesses loading the PC produce an alignment trap.

- Accesses typically take a number of cycles to complete compared to a naturally aligned transfer. The real-time implications must be carefully analyzed and key data structures might require to have their alignment adjusted for optimum performance.
- Accesses can abort on either or both halves of an access where this occurs over a page boundary. The Data Abort handler must handle restartable aborts carefully after an Alignment Fault status code is signaled.

As a result, shared memory schemes must not rely on seeing monotonic updates of non-aligned data of loads, stores, and swaps for data items greater than byte width.

Unaligned access operations must not be used for accessing Device memory-mapped registers, and must be used with care in Shared memory structures that are protected by aligned semaphores or synchronization variables.

An Unalignment trap occurs if unaligned accesses to Strongly Ordered or Device when both:

- the MMU is enabled, that is CP15 c1 bit 0, M bit, is 1
- the Subpage AP bits are disabled, that is CP15 c1 bit 23, XP bit, is 1.

Unaligned accesses to Non-shared Device memory when Subpage AP bits are enabled, that is CP15 c1 bit 23, XP bit, is 0, have Unpredictable results.

Swap and synchronization primitives, multiple-word or coprocessor access produce an alignment fault regardless of the setting of the A bit.

4.3 Unaligned data access specification

The architectural specification of unaligned data representations is defined in terms of bytes transferred between memory and register, regardless of bus width and bus endianness.

Little-endian data items are described using lower-case byte labeling bX..b0 (byteX to byte 0) and a pointer is always treated as pointing to the least significant byte of the addressed data.

Big-endian data items are described using upper-case byte labeling B0..BX (BYTE0 to BYTEX) and a pointer is always treated as pointing to the most significant byte of the addressed data.

4.3.1 Load unsigned byte, endian independent

The addressed byte is loaded from memory into the low eight bits of the general-purpose register and the upper 24 bits are zeroed, as shown in Figure 4-1.

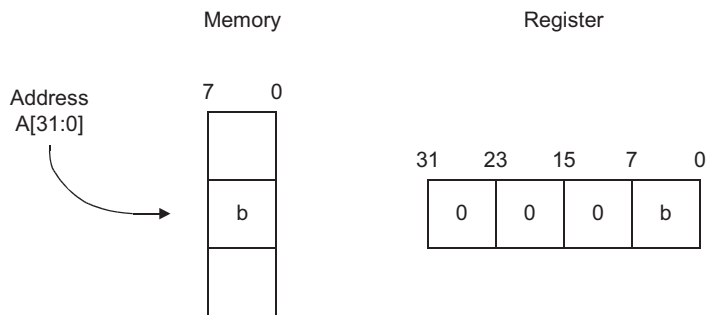


Figure 4-1 Load unsigned byte

4.3.2 Load signed byte, endian independent

The addressed byte is loaded from the memory into the low eight bits of the general-purpose register and the sign bit is extended into the upper 24 bits of the register as shown in Figure 4-2 on page 4-8.

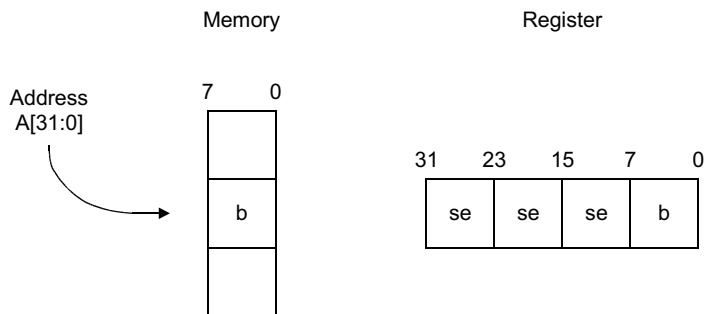


Figure 4-2 Load signed byte

In Figure 4-2, se means b (bit 7) sign extension.

4.3.3 Store byte, endian independent

The low eight bits of the general-purpose register are stored into the addressed byte in memory, as shown in Figure 4-3.

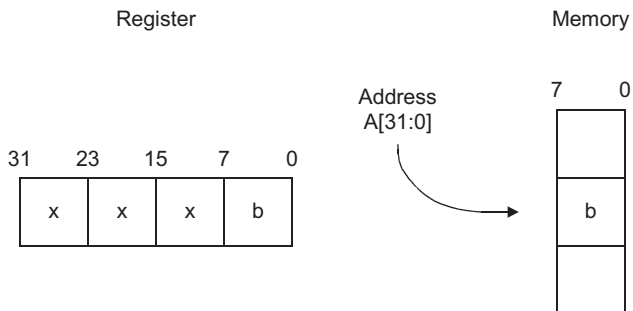


Figure 4-3 Store byte

4.3.4 Load unsigned halfword, little-endian

The addressed byte-pair is loaded from memory into the low 16 bits of the general-purpose register, and the upper 16 bits are zeroed so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register, as shown in Figure 4-4 on page 4-9.

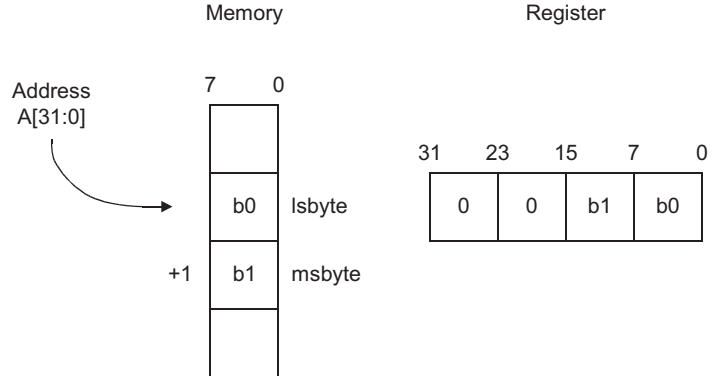


Figure 4-4 Load unsigned halfword, little-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.5 Load unsigned halfword, big-endian

The addressed byte-pair is loaded from memory into the low 16 bits of the general-purpose register, and the upper 16 bits are zeroed so that the most-significant addressed byte in memory appears in bits [15:8] of the ARM register, as shown in Figure 4-5.

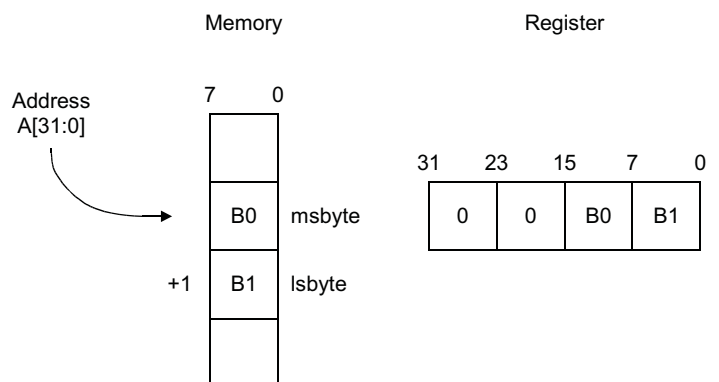


Figure 4-5 Load unsigned halfword, big-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.6 Load signed halfword, little-endian

The addressed byte-pair is loaded from memory into the low 16-bits of the general-purpose register, so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register and the upper 16 bits are sign-extended from bit 15, as shown in Figure 4-6.

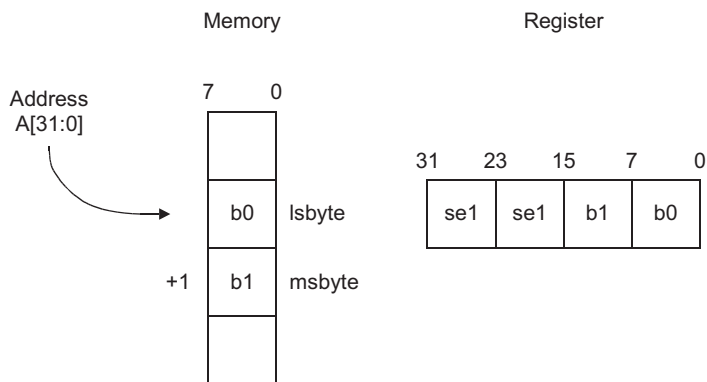


Figure 4-6 Load signed halfword, little-endian

In Figure 4-6, se1 means bit 15 (b1 bit 7) sign extended.

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.7 Load signed halfword, big-endian

The addressed byte-pair is loaded from memory into the low 16-bits of the general-purpose register, so that the most significant addressed byte in memory appears in bits [15:8] of the ARM register and bits [31:16] replicate the sign bit in bit 15, as shown in Figure 4-7 on page 4-11.

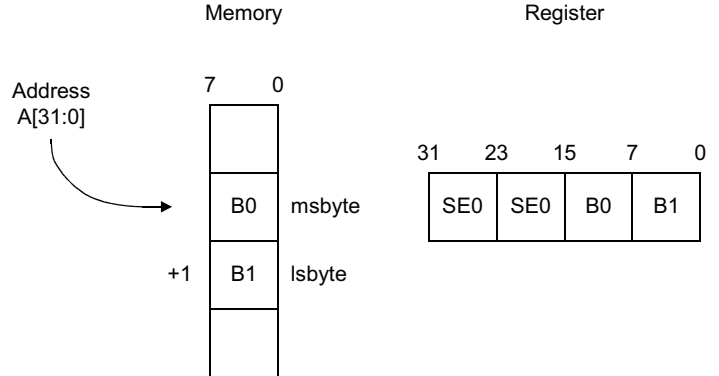


Figure 4-7 Load signed halfword, big-endian

In Figure 4-7, SE0 means bit 15 (B0 bit 7) sign extended.

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.8 Store halfword, little-endian

The low 16 bits of the general-purpose register are stored into the memory with bits [7:0] written to the addressed byte in memory, bits [15:8] to the incremental byte address in memory, as shown in Figure 4-8.

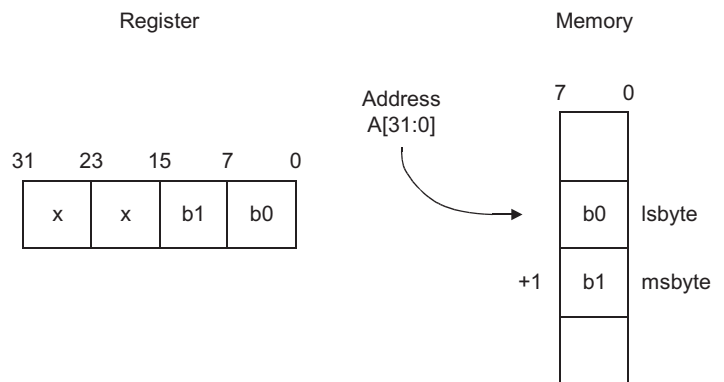


Figure 4-8 Store halfword, little-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.9 Store halfword, big-endian

The low 16 bits of the general-purpose register are stored into the memory with bits [15:8] written to the addressed byte in memory, bits [7:0] to the incremental byte address in memory, as shown in Figure 4-9.

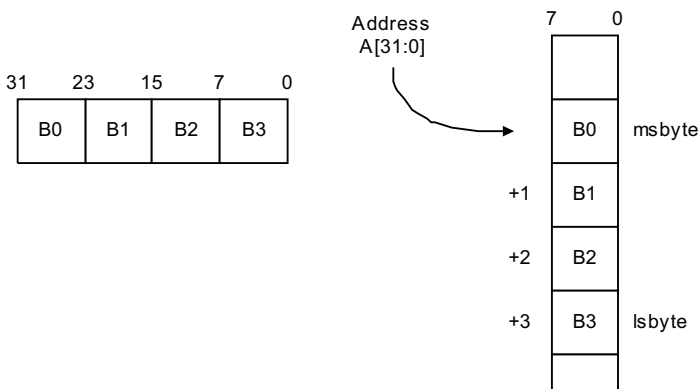


Figure 4-9 Store halfword, big-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.10 Load word, little-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register, as shown in Figure 4-10 on page 4-13.

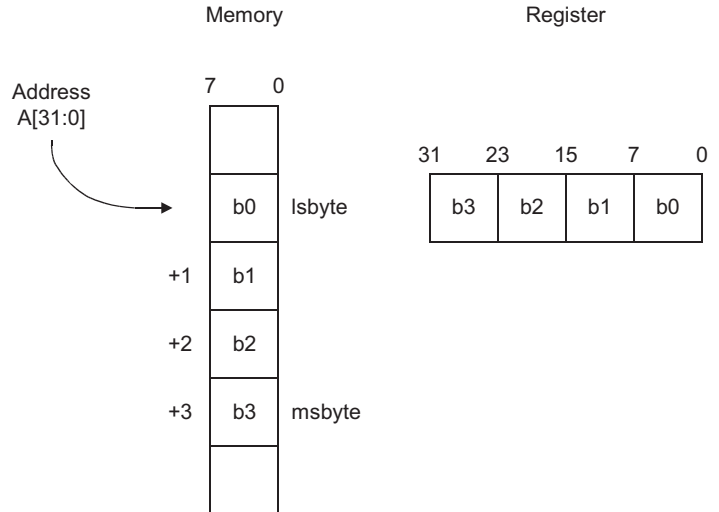


Figure 4-10 Load word, little-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.11 Load word, big-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register so that the most significant addressed byte in memory appears in bits [31:24] of the ARM register, as shown in Figure 4-11 on page 4-14.

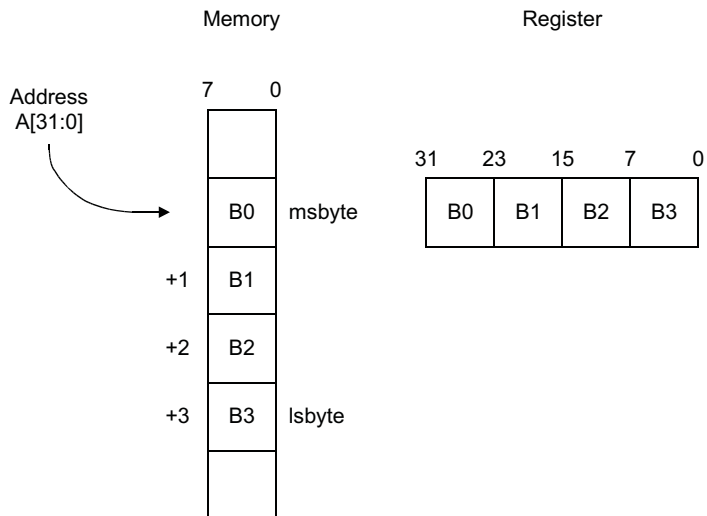


Figure 4-11 Load word, big-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.12 Store word, little-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits [7:0] of the ARM register are transferred to the least-significant addressed byte in memory, as shown in Figure 4-12 on page 4-15.

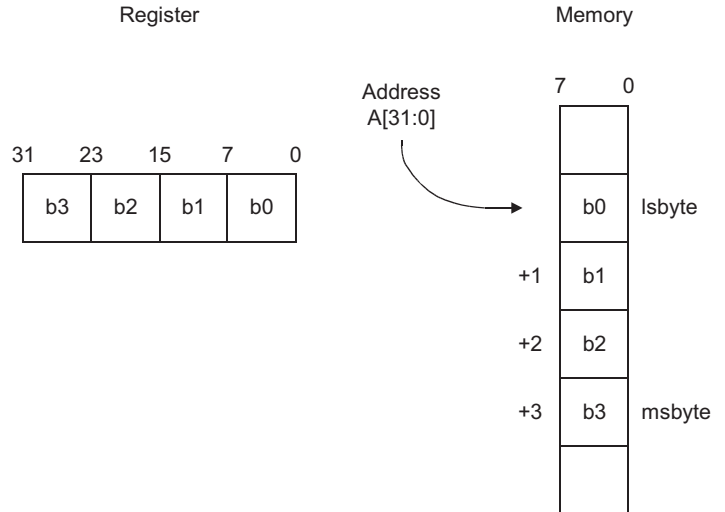


Figure 4-12 Store word, little-endian

If strict alignment fault checking is enabled and Address bits [1:0] are not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.13 Store word, big-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits [31:24] of the ARM register are transferred to the most-significant addressed byte in memory, as shown in Figure 4-13 on page 4-16.

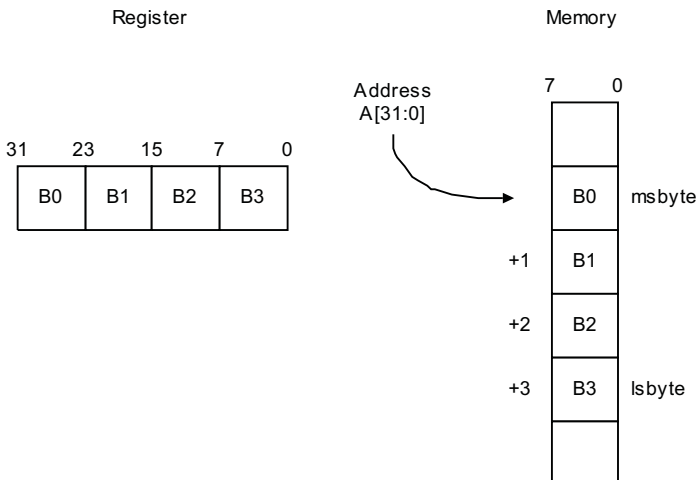


Figure 4-13 Store word, big-endian

If strict alignment fault checking is enabled and Address bits [1:0] are not zero, then a Data Abort is generated and the MMU returns a Misaligned fault in the Fault Status Register.

4.3.14 Load double, load multiple, load coprocessor (little-endian, E = 0)

The access is treated as a series of incrementing aligned word loads from memory. The data is treated as load word data (see *Load word, little-endian* on page 4-13) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.15 Load double, load multiple, load coprocessor (big-endian, E=1)

The access is treated as a series of incrementing aligned word loads from memory. The data is treated as load word data (see *Load word, big-endian* on page 4-14) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.16 Store double, store multiple, store coprocessor (little-endian, E=0)

The access is treated as a series of incrementing aligned word stores to memory. The data is treated as store word data (see *Store word, little-endian* on page 4-15) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.17 Store double, store multiple, store coprocessor (big-endian, E=1)

The access is treated as a series of incrementing aligned word stores to memory. The data is treated as store word data (see *Store word, big-endian* on page 4-16) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.4 Operation of unaligned accesses

Alignment faults and operation of non-faulting accesses of the ARM1136JF-S processor are described in this section.

Table 4-3 on page 4-19 gives details of when an alignment fault must occur for an access and of when the behavior of an access is architecturally Unpredictable. When an access neither generates an alignment fault and is not Unpredictable, details of precisely which memory locations are accessed are also given in the table.

The access type descriptions used in the Table 4-3 on page 4-19 are determined from the load/store instruction given in Table 4-2.

Table 4-2 Access type descriptions

Access type	ARM instructions	Thumb instructions
Byte	LDRB, LDRBT, LDRSB, STRB, STRBT, SWPB (either access)	LDRB, LDRSB, STRB
Halfword	LDRH, LDRSH, STRH	LDRH, LDRSH, STRH
WLoad	LDR, LDRT, SWP (load access, if U is set to 0)	LDR
WStore	STR, STRT, SWP (store access, if U is set to 0)	STR
WSync	LDREX, STREX, SWP (either access, if U is set to 1)	---
Two-word	LDRD, STRD	---
Multi-word	LDC, LDM, RFE, SRS, STC, STM	LDMIA, POP, PUSH, STMIA

The following terminology is used to describe the memory locations accessed:

Byte[X] This means the byte whose address is X in the current endianness model. The correspondence between the endianness models is that Byte[A] in the LE endianness model, Byte[A] in the BE-8 endianness model, and Byte[A EOR 3] in the BE-32 endianness model are the same actual byte of memory.

Halfword[X] This means the halfword consisting of the bytes whose addresses are X and X+1 in the current endianness model, combined to form a halfword in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

Word[X] This means the word consisting of the bytes whose addresses are X, X+1, X+2, and X+3 in the current endianness model, combined to form a word in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

Note

It is a consequence of these definitions that if X is word-aligned, Word[X] consists of the same four bytes of actual memory in the same order in the LE and BE-32 endianness models.

Align(X) This means $X \text{ AND } 0\text{xFFFFFFFC}$. That is, X with its least significant two bits forced to zero to make it word-aligned.

There is no difference between Addr and Align(Addr) on lines where Addr[1:0] is set to 0b00. You can use this to simplify the control of when the least significant bits are forced to zero.

For the Two-word and Multi-word access types, the Memory accessed column only specifies the lowest word accessed. Subsequent words have addresses constructed by successively incrementing the address of the lowest word by 4, and are constructed using the same endianness model as the lowest word.

Table 4-3 Unalignment fault occurrence when access behavior is architecturally unpredictable

A	U	Addr [2:0]	Access type(s)	Behavior	Memory accessed	Notes
0	0	-	-	-	-	Legacy, no alignment faulting
0	0	bxxx	Byte	Normal	Byte[Addr]	-
0	0	bxx0	Halfword	Normal	Halfword[Addr]	-
0	0	bxx1	Halfword	Unpredictable	-	-
0	0	bxxx	WLoad	Normal	Word[Align(Addr)]	Loaded data rotated right by $8 * \text{Addr}[1:0]$ bits
0	0	bxxx	WStore	Normal	Word[Align(Addr)]	Operation unaffected by Addr[1:0]
0	0	bx00	WSync	Normal	Word[Addr]	-
0	0	bxx1, b x1x	WSync	Unpredictable	-	-
0	0	bxxx	Multi-word	Normal	Word[Align(Addr)]	Operation unaffected by Addr[1:0]
0	0	b000	Two-word	Normal	Word[Addr]	-
0	0	bxx1, bx1x, b1xx	Two-word	Unpredictable	-	-
0	1	-	-	-	-	ARMv6 unaligned support

**Table 4-3 Unalignment fault occurrence
when access behavior is architecturally unpredictable (continued)**

A	U	Addr [2:0]	Access type(s)	Behavior	Memory accessed	Notes
0	1	bxxx	Byte	Normal	Byte[Addr]	-
0	1	bxxx	Halfword	Normal	Halfword[Addr]	-
0	1	bxxx	WLoad, WStore	Normal ^a	Word[Addr]	-
0	1	bx00	WSync, Multi-word, Two-word	Normal	Word[Addr]	-
0	1	bxx1, bx1x	WSync, Multi-word, Two-word	Alignment Fault	-	-
1	x	-	-	-	-	Full alignment faulting
1	x	bxxx	Byte	Normal	Byte[Addr]	-
1	x	bxx0	Halfword	Normal	Halfword[Addr]	-
1	x	bxx1	Halfword	Alignment Fault	-	-
1	x	bx00	WLoad, WStore, WSync, Multi-word	Normal	Word[Addr]	-
1	x	bxx1, bx1x	WLoad, WStore, WSync, Multi-word	Alignment Fault	-	-
1	x	b000	Two-word	Normal	Word[Addr]	-
1	0	b100	Two-word	Alignment Fault	-	U set to 0: 64-bit alignment of LDRD/STRD
1	1	b100	Two-word	Normal	Word[Addr]	U set to 1: 32-bit alignment of LDRD/STRD
1	x	bxx1, bx1x	Two-word	Alignment Fault	-	-

- a. Alignment faults occur when accesses using Addr[1:0] of b1x or bx1 are made to Strongly Ordered or Device memory when CP15 c1 XP and M bits are set that load the PC. Accesses to Non-shared Device memory when XP bit of CP15 c1 is 0.

The following causes override the behavior specified in the Table 4-3 on page 4-19:

- An LDR instruction that loads the PC, has Addr[1:0] != 0b00, and is specified in the table as having Normal behavior instead has Unpredictable behavior.

The reason why this applies only to LDR is that most other load instructions are Unpredictable regardless of alignment if the PC is specified as their destination register.

The exceptions are ARM LDM and RFE instructions, and Thumbs POP instruction. If the instruction for them is Addr[1:0] != 0b00, the effective address of the transfer has its two least significant bits forced to 0 if A is set 0 and U is set to 0. Otherwise the behavior specified in *Unalignment fault occurrence when access behavior is architecturally unpredictable* on page 4-19 is either Unpredictable or Alignment Fault regardless of the destination register.

- Any WLoad, WStore, WSync, Two-word, or Multi-word instruction that accesses device memory, has Addr[1:0] != 0b00, and is specified in *Unalignment fault occurrence when access behavior is architecturally unpredictable* on page 4-19 as having Normal behavior instead has Unpredictable behavior.
- Any Halfword instruction that accesses device memory, has Addr[0] != 0, and is specified in the table as having Normal behavior instead has Unpredictable behavior.

4.5 Mixed-endian access support

Mixed-endian data access is described in:

- *Legacy fixed instruction and data endianness*
- *ARMv6 support for mixed-endian data*
- *Instructions to change the CPSR E bit on page 4-27.*

4.5.1 Legacy fixed instruction and data endianness

Prior to ARMv6 the endianness of both instructions and data are locked together, and the configuration of the processor and the external memory system must either be hard-wired or programmed in the first few instructions of the bootstrap code.

Where the endianness is configurable under program control, the MMU provides a mechanism in CP15 c1 to set the B bit, which enables byte addressing renaming with 32-bit words. This model of big-endian access, called BE-32 in this document, relies on a word-invariant view of memory where an aligned 32-bit word reads and writes the same word of data in memory when configured as either big-endian or little-endian. This enables an ARM 32-bit instruction sequence to be executed to program the B bit, but no byte or halfword data accesses or 16-bit Thumb instructions can be used until the processor configuration matches the system endianness.

This behavior is still provided for legacy software when the U bit in CP15 Register c1 is zero, as shown in Table 4-4.

Table 4-4 Legacy endianness using CP15 c1

U	B	Instruction endianness	Data endianness	Description
0	0	LE	LE	LE (reset condition)
0	1	BE-32	BE-32	Legacy BE (32-bit word-invariant)

4.5.2 ARMv6 support for mixed-endian data

In ARMv6 the instruction and data endianness are separated:

- instructions are fixed little-endian
- data accesses can be either little-endian or big-endian as controlled by bit 9, the E bit, of the Program Status Register.

The values of the U, B, and E bits on any exception entry, including reset, are determined by the CPSR Register 15 EE bit.

Fixed little-endian Instructions

Instructions must be naturally aligned and are always treated as being stored in memory in little-endian format. That is, the PC points to the least-significant-byte of the instruction.

Instructions have to be treated as data by exception handlers (decoding SWI calls and Undefined instructions, for example).

Instructions can also be written as data by debuggers, Just-In-Time compilers, or in operating systems that update exception vectors.

Mixed-endian data access

The operating-system typically has a required endian representation of internal data structures, but applications and device drivers have to work with data shared with other processors (DSP or DMA interfaces) that might have fixed big-endian or little-endian data formatting.

A byte-invariant addressing mechanism is provided that enables the load/store architecture to be qualified by the CPSR E bit that provides byte reversing of big-endian data in to, and out of, the processor register bank transparently. This byte-invariant big-endian representation is referred to as BE-8 in this document.

The effect on byte, halfword, word, and multi-word accesses of setting the CPSR E bit when the U bit enables unaligned support is described in *Mixed-endian configuration supported* on page 4-24.

Byte data access

The same physical byte in memory is accessed whether big-endian or little-endian:

- Unsigned byte load as described in *Load unsigned byte, endian independent* on page 4-7.
- Signed byte load as described in *Load signed byte, endian independent* on page 4-7.
- Byte store as described in *Store byte, endian independent* on page 4-8.

Halfword data access

The same two physical bytes in memory are accessed whether big-endian or little-endian. Big-endian halfword load data is byte-reversed as read into the processor register to ensure little-endian internal representation, and similarly is byte-reversed on store to memory:

- Unsigned halfword load as described in *Load unsigned halfword, little-endian* on page 4-8 (LE), and *Load unsigned halfword, big-endian* on page 4-9 (BE-8).
- Signed halfword load as described in *Load signed halfword, little-endian* on page 4-10 (LE), and *Load signed halfword, big-endian* on page 4-10 (BE-8).
- Halfword store as described in *Store halfword, little-endian* on page 4-11 (LE), and *Store halfword, big-endian* on page 4-12 (BE-8).

Load Word

The same four physical bytes in memory are accessed whether big-endian or little-endian. Big-endian word load data is byte reversed as read into the processor register to ensure little-endian internal representation, and similarly is byte-reversed on store to memory:

- Word load as described in *Load word, little-endian* on page 4-12 (LE), and *Load word, big-endian* on page 4-13 (BE-8).
- Word store as described in *Store word, little-endian* on page 4-14 (LE), and *Store word, big-endian* on page 4-15 (BE-8).

Mixed-endian configuration supported

This behavior is enabled when the U bit in CP15 Register c1 is set. This is only supported when the B bit in CP15 Register c1 is reset, as shown in Table 4-5.

Table 4-5 Mixed-endian configuration

U	B	E	Instruction endianness	Data endianness	Description
1	0	0	LE	LE	LE instructions, little-endian data load/store. Unaligned data access allowed.
1	0	1	LE	BE-8	LE instructions, big-endian data load/store. Unaligned data access allowed.
1	1	0	BE-32	BE-32	Legacy BE instructions/data.
1	1	1	-	-	Reserved.

4.5.3 Reset values of the U, B, and EE bits

The Reset values of the U, B, and EE bits are determined by the pins **BIGENDINIT** and **UBITINIT** as shown in Table 4-6.

Table 4-6 B bit, U bit, and EE bit settings

BIGENDINIT	UBITINIT	B	U	EE
0	0	0	0	0
0	1	0	1	0
1	0	1	0	0
1	1	0	1	1

4.6 Instructions to reverse bytes in a general-purpose register

When an application or device driver has to interface to memory-mapped peripheral registers or shared-memory DMA structures that are not the same endianness as that of the internal data structures, or the endianness of the Operating System, an efficient way of being able to explicitly transform the endianness of the data is required.

The following new instructions are added to the ARM and Thumb instruction sets to provide this functionality:

- reverse word (4 bytes) register, for transforming big and little-endian 32-bit representations
- reverse halfword and sign-extend, for transforming signed 16-bit representations
- Reverse packed halfwords in a register for transforming big- and little-endian 16-bit representations.

These instructions are described in *ARM1136JF-S instruction set summary* on page 1-36.

4.6.1 All load and store operations

All load and store instructions take account of the CPSR E bit. Data is transferred directly to registers when E = 0, and byte reversed if E = 1 for halfword, word, or multiple word transfers.

Operation:

When CPSR[<E-bit>] = 1 then byte reverse load/store data

4.7 Instructions to change the CPSR E bit

ARM and Thumb instructions are provided to set and clear the E-bit efficiently:

SETEND BE Sets the CPSR E bit
SETEND LE Resets the CPSR E bit.

These are specified as unconditional operations to minimize pipelined implementation complexity.

These instructions are described in *ARM1136JF-S instruction set summary* on page 1-36.

Chapter 5

Program Flow Prediction

This chapter outlines how program flow prediction locates branches in the instruction stream and the strategies used for determining if a branch is likely to be taken or not. It also describes the two architecturally-defined SWI functions required for backwards-compatibility with earlier architectures for flushing the *Prefetch Unit* (PU) buffers. It contains the following sections:

- *About program flow prediction* on page 5-2
- *Branch prediction* on page 5-4
- *Return stack* on page 5-8
- *Instruction Memory Barrier (IMB) instruction* on page 5-9
- *ARM1020T or later IMB implementation* on page 5-10.

5.1 About program flow prediction

Program flow prediction in ARM1136JF-S processors is carried out by:

The core Implements static branch prediction and the Return Stack.

The Prefetch Unit Implements dynamic branch prediction.

The ARM1136JF-S processor is responsible for handling branches the first time they are executed, that is, when no historical information is available for dynamic prediction by the PU.

The core makes static predictions about the likely outcome of a branch early in its pipeline and then resolves those predictions when the outcome of conditional execution is known. Condition codes are evaluated at three points in the core pipeline, and branches are resolved as soon as the flags are guaranteed not to be modified by a preceding instruction.

When a branch is resolved, the core passes information to the PU so that it can make a *Branch Target Address Cache* (BTAC) allocation or update an existing entry as appropriate. The core is also responsible for identifying likely procedure calls and returns to predict the returns. It can handle nested procedures up to three deep.

The core includes:

- a *Static Branch Predictor* (SBP)
- a *Return Stack* (RS)
- branch resolution logic
- a BTAC update interface to the PU.

The ARM1136JF-S PU is responsible for fetching instructions from the memory system as required by the integer unit, and coprocessors. The bus from the memory system to the PU is 64 bits wide. It supplies two words every clock cycle if the access hits in the Instruction Cache except in cases where the fetch is to the last word in a line. In this case only one word is provided by the cache. The PU buffers up to three instructions in its FIFO to:

- detect branch instructions ahead of the integer unit requirement
- dynamically predict those that it considers are to be taken
- provide branch folding of predicted branches if possible.

This reduces the cycle time of the branch instructions, so increasing processor performance.

The PU includes:

- a BTAC
- branch update and allocate logic

- a *Dynamic Branch Predictor* (DBP), and associated update mechanism
- branch folding logic.

It is responsible for providing the core with instructions, and for requesting cache accesses. The pattern of cache accesses is based on the predicted instruction stream as determined by the dynamic branch prediction mechanism or the core flush mechanism.

The BTAC can:

- be globally flushed by a CP15 instruction
- have individual entries flushed by a CP15 instruction
- be enabled or disabled by a CP15 instruction.

For details of CP15 instructions see Chapter 3 *Control Coprocessor CP15*.

The PU also handles the cache access multiplexing for:

- CP15 instruction handling
- data accesses to the Instruction TCM
- DMA accesses to the TCM.

The PU holds the pending Instruction Cache miss information prior to acceptance by the level two instruction side controller (this handles the case of Prefetch stalls). The PU prefetches all instruction types regardless of the state of the core. That is, for ARM state, Thumb state, or Java state. However the rate of draining of the PU is a function of these states, and the functioning of the branch prediction hardware is a function of the state.

The PU is responsible for fetching the instruction stream as dictated by:

- the Program Counter
- the dynamic branch predictor
- static prediction results in the core
- procedure calls and returns signaled by the Return Stack residing in the core
- exceptions, instruction aborts, and interrupts signaled by the core.

5.2 Branch prediction

In ARM processors that have no PU, the target of a branch is not known until the end of the Execute stage. At the Execute stage it is known whether or not the branch is taken. The best performance is obtained by predicting all branches as not taken and filling the pipeline with the instructions that follow the branch in the current sequential path. In ARM processors without a PU, an untaken branch requires one cycle and a taken branch requires three or more cycles.

Branch prediction enables the detection of branch instructions before they enter the integer unit. This permits the use of a branch prediction scheme that closely models actual conditional branch behavior.

The increased pipeline length of the ARM1136JF-S processor makes the performance penalty of any changes in program flow, such as branches or other updates to the PC, more significant than was the case on the ARM9TDMI or ARM1020T cores. Therefore, a significant amount of hardware is dedicated to prediction of these changes. Two major classes of program flow are addressed in the ARM1136JF-S prediction scheme:

1. Branches (including BL, and BLX immediate), where the target address is a fixed offset from the program counter. The prediction amounts to an examination of the probability that a branch passes its condition codes. These branches are handled in the Branch Predictors.
2. Loads, Moves, and ALU operations writing to the PC, which can be identified as being likely to be a return from a procedure call. Two identifiable cases are Loads to the PC from an address derived from r13 (the stack pointer), and Moves or ALU operations to the PC derived from r14 (the Link Register). In these cases, if the calling operation can also be identified, the likely return address can be stored in a hardware implemented stack, termed a *Return Stack* (RS). Typical calling operations are BL and BLX instructions. In addition Moves or ALU operations to the Link Register from the PC are often preludes to a branch that serves as a calling operation. The Link Register value derived is the value required for the RS. This was most commonly done on ARMv4T, before the BLX <register> instruction was introduced in ARMv5T.

A third class of program flow change that has been considered is all other Loads, Moves, and ALU operations that are not recognized as being associated with the return from a procedure call, as described in step 2. above. These could be predicted with a dynamic branch predictor, with the requirement to check the derived address. System simulations suggest that a simple implementation of such approaches is unlikely to be efficient.

Branch prediction is required in the design to reduce the core CPI loss that arises from the longer pipeline. To improve the branch prediction accuracy, a combination of static and dynamic techniques is employed. It is possible to disable the predictors.

5.2.1 Enabling program flow prediction

The enabling of program flow prediction is controlled by the CP15 Register c1 Z bit (bit 11), which is set to 0 on Reset. See *Control Register* on page 3-96. The return stack, dynamic predictor, and static predictor can also be individually controlled using the Auxiliary Control Register. See *Auxiliary Control Register* on page 3-93.

5.2.2 Dynamic branch predictor

The first line of branch prediction in the ARM1136JF-S processor is dynamic, through a simple BTAC. It is virtually addressed and holds virtual target addresses. In addition, a two bit value holds the predicted direction of the branch. If the address mappings change, this cache must be flushed. A dynamic branch predictor flush is included in the CP15 coprocessor control instructions.

A BTAC works by storing the existence of branches at particular locations in memory. The branch target address and a prediction of whether or not it might be taken is also stored.

The BTAC provides dynamic prediction of branches, including BL and BLX instructions in both ARM, Thumb, and Java states. The BTAC is a 128-entry direct-mapped cache structure used for allocation of Branch Target Addresses for resolved branches. The BTAC uses a 2-bit saturating prediction history scheme to provide the dynamic branch prediction. When a branch has been allocated into the BTAC, it is only evicted in the case of a capacity clash. That is, by another branch at the same index.

The prediction is based on the previous behavior of this branch. The four possible states of the prediction bits are:

- strongly predict branch taken
- weakly predict branch taken
- weakly predict branch not taken
- strongly predict branch not taken.

The history is updated for each occurrence of the branch. This updating is scheduled by the core when the branch has been resolved.

Branch entries are allocated into the BTAC after having been resolved at Execute. BTAC hits enable branch prediction with zero cycle delay. When a BTAC hit occurs, the Branch Target Address stored in the BTAC is used as the Program Counter for the next

Fetch. Both branches resolved taken and not taken are allocated into the BTAC. This enables the BTAC to do the most useful amount of work and improves performance for tight backward branching loops.

The BTAC has a latency of two cycles and a throughput of one cycle. It is pipelined to enable it to do the one lookup per cycle during the instruction buffer refill time.

5.2.3 Static branch predictor

The second level of branch prediction in the ARM1136JF-S processor uses static branch prediction that is based solely on the characteristics of a branch instruction. It does not make use of any history information. The scheme used in the ARM1136JF-S processor predicts that all forward conditional branches are not taken and all backward branches are taken. Around 65% of all branches are preceded by enough non-branch cycles to be completely predicted.

Branch prediction is performed only when the Z bit in CP15 Register c1 is set to 1. See *Control Register* on page 3-96 for details of this register. Dynamic prediction works on the basis of caching the previously seen branches in the BTAC, and like all caches suffers from the compulsory miss that exists on the first encountering of the branch by the predictor. A second, static predictor is added to the design to counter these misses, and to mop-up any capacity and conflict misses in the BTAC. The static predictor amounts to an early evaluation of branches in the pipeline, combined with a predictor based on the direction of the branches to handle the evaluation of condition codes that are not known at the time of the handling of these branches. Only items that have not been predicted in the dynamic predictor are handled by the static predictor.

The static branch predictor is hard-wired with backward branches being predicted as taken, and forward branches as not taken. The SBP looks at the MSB of the branch offset to determine the branch direction. Statically predicted taken branches incur a one-cycle delay before the target instructions start refilling the pipeline. The SBP works in both ARM and Thumb states. The SBP does not function in Java state. It can be disabled using CP15 Register c1. See *Control Register* on page 3-96.

5.2.4 Branch folding

Branch folding is a technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches significantly below 1.

Branch folding is done for all dynamically predicted branches. Branch folding is not done for:

- BL and BLX instructions (to avoid losing the link)

- predicted branches onto predicted branches
- branches that are breakpointed or have generated an abort when fetched.

5.2.5 Incorrect predictions and correction

Branches are resolved at or before the Ex3 stage of the core pipeline. A misprediction causes the pipeline to be flushed, and the correct instruction stream to be fetched. If branch folding is implemented, the failure of the condition codes of a folded branch causes the instruction that follows the folded branch to fail. Whenever a potentially incorrect prediction is made, the following information, necessary for recovering from the error, is stored:

- a fall-through address in the case of a predicted taken branch instruction
- the branch target address in the case of a predicted not taken branch instruction.

The PU passes the conditional part of any optimized branch into the integer unit. This enables the integer unit to compare these bits with the processor flags and determine if the prediction was correct or not. If the prediction was incorrect, the integer unit flushes the PU and requests that prefetching begins from the stored recovery address.

5.3 Return stack

A return stack is used for predicting the class of program flow changes that includes loads, moves, and ALU operations, writing to the PC that can be identified as being likely to be a procedure call or return.

The return stack is a three-entry circular buffer used for the prediction of procedure calls and procedure returns. Only unconditional procedure returns are predicted.

When a procedure call instruction is predicted, the return address is taken from the Execute stage of the pipeline and pushed onto the return stack. The instructions recognized as procedure calls are:

- BL <dest>
- BLX <dest>
- BLX <reg>.

The first two instructions are predicted by the BTAC, unless they result in a BTAC miss. The third instruction is not predicted. The SBP predicts unconditional procedure calls as taken, and conditional procedure calls as not taken.

When a procedure return instruction is predicted, an instruction fetch from the location at the top of the return stack occurs, and the return stack is popped. The instructions recognized as procedure returns are:

- BX r14
- LDM sp!, {...,pc}
- LDR pc, [sp...].

The SBP only predicts procedure returns that are always predicted as taken.

Two classes of return stack mispredictions can exist:

- condition code failures of the return operation
- incorrect return location.

In addition, an empty return stack gives no prediction.

5.4 Instruction Memory Barrier (IMB) instruction

The SBP in the core might statically predict a branch as taken. In this case the request to fetch from the branch target path is marked as speculative. In some circumstances it is likely that the prefetch buffer and pipeline contains out-of-date instructions. In these circumstances the prefetch buffer must be flushed. The *Instruction Memory Barrier* (IMB) instruction provides a way to do this for the ARM1020T processor. The ARM1136JF-S processor maintains this capability for backwards compatibility with the ARM1020T.

To implement the two IMB instructions, you must include processor-specific code in the SWI handler:

IMB	The IMB instruction flushes all information about all instructions.
IMBRange	When only a small area of code is altered before being executed the IMBRange instruction can be used to efficiently and quickly flush any stored instruction information from addresses within a small range. By flushing only the required address range information, the rest of the information remains to provide improved system performance.

These instructions are implemented as calls to specific SWI numbers:

IMB	SWI 0xF00000
IMBRange	SWI 0xF00001.

5.4.1 Generic IMB use

Use SWI functions to provide a well-defined interface between code that is:

- independent of the ARM processor implementation it is running on
- specific to the ARM processor implementation it is running on.

The implementation-independent code is provided with a function that is available on all processor implementations using the SWI interface, and that can be accessed by privileged and, where appropriate, non-privileged (User mode) code.

Using SWIs to implement the IMB instructions means that any code that is written now is compatible with any future processors, even if those processors implement IMB in different ways. This is achieved by changing the operating system SWI service routines for each of the IMB SWI numbers that differ from processor to processor.

5.5 ARM1020T or later IMB implementation

For ARM1020T or later processors, executing the SWI instruction is sufficient in itself to cause IMB operation. Also, for ARM1020T or later, both the IMB and the IMBRange instructions flush all stored information about the instruction stream.

This means that all IMB instructions can be implemented in the operating system by returning from the IMB or IMBRange service routine and that the service routines can be exactly the same. The following service routine code can be used:

```
IMB_SWI_handler
IMBRange_SWI_handler

MOVS PC, R14_svc      ; Return to the code after the SWI call
```

Note

- In new code, you are strongly encouraged to use the IMBRange instruction whenever the changed area of code is small, even if there is no distinction between it and the IMB instruction on ARM1020T or ARM1136JF-S processors. Future processors might implement the IMBRange instruction in a much more efficient and faster manner, and code migrated from the ARM920T core is likely to benefit when executed on these processors.
 - ARM1136JF-S processors implement a Flush Prefetch Buffer operation that is user-accessible and acts as an IMB. For more details see *Cache Operations Register* on page 3-17.
-

5.5.1 Execution of IMB instructions

This section comprises three examples that show what can happen during the execution of IMB instructions. The pseudo code in the square brackets shows what happens to execute the IMB instruction (or IMBRange) in the SWI handler.

Example 5-1 shows how code that loads a program from a disk, and then branches to the entry point of that program, must execute an IMB instruction between loading the program and trying to execute it.

Example 5-1 Loading code from disk

```
IMB EQU 0xF00000
.
.
; code that loads program from disk
.
```

```

.
SWI IMB
    [branch to IMB service routine]
    [perform processor-specific operations to execute IMB]
    [return to code]
.
MOV PC, entry_point_of_loaded_program
:
.

```

Compiled BitBlit routines optimize large copy operations by constructing and executing a copying loop that has been optimized for the exact operation wanted. When writing such a routine an IMB is required between the code that constructs the loop and the actual execution of the constructed loop. This is shown in Example 5-2.

Example 5-2 Running BitBlit code

```

IMBRange EQU 0xF00001.
.
; code that constructs loop code
; load R0 with the start address of the constructed loop
; load R1 with the end address of the constructed loop
SWI IMBRange
    [branch to IMBRange service routine]
    [read registers R0 and R1 to set up address range parameters]
    [perform processor-specific operations to execute IMBRange]
    [within address range]
    [return to code]
; start of loop code
.
.

```

When writing a self-decompressing program, an IMB must be issued after the routine that decompresses the bulk of the code and before the decompressed code starts to be executed. This is shown in Example 5-3.

Example 5-3 Self-decompressing code

```

IMB EQU 0xF00000
.
.
; copy and decompress bulk of code
SWI IMB

```

```
; start of decompressed code
```

```
.  
.  
.
```

Chapter 6

Memory Management Unit

This chapter describes the *Memory Management Unit (MMU)* and how it is used. It contains the following sections:

- *About the MMU* on page 6-2
- *TLB organization* on page 6-4
- *Memory access sequence* on page 6-7
- *Enabling and disabling the MMU* on page 6-9
- *Memory access control* on page 6-11
- *Memory region attributes* on page 6-14
- *Memory attributes and types* on page 6-17
- *MMU aborts* on page 6-27
- *MMU fault checking* on page 6-29
- *Fault status and address* on page 6-33
- *Hardware page table translation* on page 6-35
- *MMU descriptors* on page 6-43
- *MMU software-accessible registers* on page 6-55
- *MMU and Write Buffer* on page 6-59.

6.1 About the MMU

The ARM1136JF-S MMU works with the cache memory system to control accesses to and from external memory. The MMU also controls the translation of virtual addresses to physical addresses.

The ARM1136JF-S processor implements an ARMv6 MMU to provide address translation and access permission checks for the instruction and data ports of the ARM1136JF-S processor. The MMU controls table-walking hardware that accesses translation tables in main memory. A single set of two-level page tables stored in main memory controls the contents of the instruction and data side *Translation Lookaside Buffers* (TLBs). The finished virtual address to physical address translation is put into the TLB. The TLBs are enabled from a single bit in CP15 Control Register c1, providing a single address translation and protection scheme from software.

The MMU features are:

- standard ARMv6 MMU mapping sizes, domains, and access protection scheme
- mapping sizes are 4KB, 64KB, 1MB, and 16MB
- the access permissions for 1MB sections and 16MB supersections are specified for the entire section
- you can specify access permissions for 64KB large pages and 4KB small pages separately for each quarter of the page (these quarters are called subpages)
- 16 domains
- one 64-entry unified TLB and a lockdown region of eight entries
- you can mark entries as a global mapping, or associated with a specific application space identifier to eliminate the requirement for TLB flushes on most context switches
- access permissions extended to enable supervisor read-only and supervisor/user read-only modes to be simultaneously supported
- memory region attributes to mark pages shared by multiple processors
- hardware page table walks
- Round-Robin replacement algorithm.

The MMU memory system architecture enables fine-grained control of a memory system. This is controlled by a set of virtual to physical address mappings and associated memory properties held within one or more structures known as TLBs within the MMU. The contents of the TLBs are managed through hardware translation lookups from a set of translation tables in memory.

To prevent requiring a TLB invalidation on a context switch, you can mark each virtual to physical address mapping as being associated with a particular application space, or as global for all application spaces. Only global mappings and those for the current application space are enabled at any time. By changing the *Application Space Identifier* (ASID) you can alter the enabled set of virtual to physical address mappings. The set of memory properties associated with each TLB entry include:

Memory access permission control

This controls if a program has no-access, read-only access, or read/write access to the memory area. When an access is attempted without the required permission, a memory abort is signaled to the processor. The level of access possible can also be affected by whether the program is running in User mode, or a privileged mode, and by the use of domains. See *Memory access control* on page 6-11 for more details.

Memory region attributes

These describe properties of a memory region. Examples include Device, Noncachable, Write-Through, and Write-Back. If an entry for a virtual address is not found in a TLB then a set of translation tables in memory are automatically searched by hardware to create a TLB entry. This process is known as a translation table walk. If the ARM1136JF-S processor is in ARMv5 backwards-compatible mode some new features, such as ASIDs, are not available. The MMU architecture also enables specific TLB entries to be locked down in a TLB. This ensures that accesses to the associated memory areas never require looking up by a translation table walk. This minimizes the worst-case access time to code and data for real-time routines.

6.2 TLB organization

The TLB organization is described in:

- *MicroTLB*
- *Main TLB* on page 6-5
- *TLB control operations* on page 6-5
- *Page-based attributes* on page 6-6
- *Supersections* on page 6-6.

6.2.1 MicroTLB

The first level of caching for the page table information is a small MicroTLB of ten entries that is implemented on each of the instruction and data sides. These entities are implemented in logic, providing a fully associative lookup of the virtual addresses in a cycle. This means that a MicroTLB miss signal is returned at the end of the DC1 cycle. In addition to the virtual address, an *Address Space Identifier* (ASID) is used to distinguish different address mappings that might be in use.

The current ASID is a small identifier, eight bits in size, that is programmed using CP15 when different address mappings are required. A memory mapping for a page or section can be marked as being global or referring to a specific ASID. The MicroTLB uses the current ASID in the comparisons of the lookup for all pages for which the global bit is not set.

The MicroTLB returns the physical address to the cache for the address comparison, and also checks the protection attributes in sufficient time to signal a Data Abort in the DC2 cycle. A additional set of attributes, to be used by the cache line miss handler, are provided by the MicroTLB. The timing requirements for these are less critical than for the physical address and the abort checking.

You can configure MicroTLB replacement to be round-robin or random replacement. By default the round-robin replacement algorithm is used. The random replacement algorithm is designed to be selected for rare pathological code that causes extreme use of the MicroTLB. With such code, you can often improve the situation by using a random replacement algorithm for the MicroTLB. You can only select random replacement of the MicroTLB if random cache selection is in force, as set by the Control Register RR bit. If the RR bit is 0, then you can select random replacement of the MicroTLB by setting the Auxiliary Control Register bit 3.

All main TLB maintenance operations affect both the instruction and data MicroTLBs, causing them to be flushed.

The virtual addresses held in the MicroTLB include the FCSE translation from *Virtual Address (VA)* to *Modified Virtual Address (MVA)*, see the *ARM Architecture Reference Manual Part B*. The process of loading the MicroTLB from the main TLB includes the FCSE translation if appropriate. The MicroTLB has 10 entries.

6.2.2 Main TLB

The main TLB is the second layer in the TLB structure that catches the cache misses from the MicroTLBs. It provides a centralized source for lockable translation entries.

Misses from the instruction and data MicroTLBs are handled by a unified main TLB, that is accessed only on MicroTLB misses. Accesses to the main TLB take a variable number of cycles, according to competing requests between each of the MicroTLBs and other implementation-dependent factors. Entries in the lockable region of the main TLB are lockable at the granularity of a single entry, as described in *TLB Lockdown Register* on page 3-77.

Main TLB implementation

The main TLB is implemented as a combination of two elements:

- a fully-associative array of eight elements, which is lockable
- a low-associativity Tag RAM and DataRAM structure similar to that used in the Cache.

The implementation of the low-associativity region is a 64-entry 2-way associative structure. Depending on the RAMs available, you can implement this as either:

- four 32-bit wide RAMs
- two 64-bit wide RAMs
- a single 128-bit wide RAM.

Main TLB misses

Main TLB misses are handled in hardware by the level two page table walk mechanism, as used on previous ARM processors. See *TLB Operations Register* on page 3-75.

6.2.3 TLB control operations

The TLB control operations are described in *TLB Operations Register* on page 3-75 and *TLB Lockdown Register* on page 3-77.

6.2.4 Page-based attributes

The page-based attributes for access protection are described in *Memory access control* on page 6-11. The memory types and page-based cache control attributes are described in *Memory region attributes* on page 6-14 and *Memory attributes and types* on page 6-17. The ARM1136JF-S processor interprets the Shared bit in the MMU for regions that are Cachable as making the accesses Noncachable. This ensures memory coherency without incurring the cost of dedicated cache coherency hardware. The behavior of memory system when the MMU is disabled is described in *Enabling and disabling the MMU* on page 6-9.

6.2.5 Supersections

In addition to the ARMv6 page types, ARM1136JF-S processors support 16MB pages, which are known as *supersections*. These are designed for mapping large expanses of the memory map in a single TLB entry.

Supersections are defined using a first level descriptor in the page tables, similar to the way a Section is defined. Because each first level page table entry covers a 1MB region of virtual memory, the 16MB supersections require that 16 identical copies of the first level descriptor of the supersection exist in the first level page table.

Every supersection is defined to have its Domain as 0.

Supersections can be specified regardless of whether subpages are enabled or not, as controlled by the CP15 Control Register XP bit (bit 23). The page table formats of supersections are shown in Figure 6-4 on page 6-38 and Figure 6-8 on page 6-41.

6.3 Memory access sequence

When the ARM1136JF-S processor generates a memory access, the MMU:

1. Performs a lookup for a mapping for the requested virtual address and current ASID in the relevant Instruction or Data MicroTLB.
2. If step 1 misses then a lookup for a mapping for the requested virtual address and current ASID in the main TLB is performed.

If no global mapping, or mapping for the currently selected ASID, for the virtual address can be found in the TLBs then a translation table walk is automatically performed by hardware. See *Hardware page table translation* on page 6-35.

If a matching TLB entry is found then the information it contains is used as follows:

1. The access permission bits and the domain are used to determine if the access is allowed. If the access is not allowed the MMU signals a memory abort, otherwise the access is enabled to proceed. *Memory access control* on page 6-11 describes how this is done.
2. The memory region attributes are used to control the cache and write buffer, and to determine if the access is cached, uncached, or device, and if it is shared, as described in *Memory region attributes* on page 6-14.
3. The physical address is used for any access to external or tightly coupled memory to perform Tag matching for cache entries.

6.3.1 TLB match process

Each TLB entry contains a virtual address, a page size, a physical address, and a set of memory properties. Each is marked as being associated with a particular application space, or as global for all application spaces. Register c13 in CP15 determines the currently selected application space. A TLB entry matches if bits [31:N] of the virtual address match, where N is \log_2 of the page size for the TLB entry. It is either marked as global, or the *Application Space Identifier* (ASID) matches the current ASID. The behavior of a TLB if two or more entries match at any time, including global and ASID-specific entries, is Unpredictable. The operating system must ensure that, at most, one TLB entry matches at any time. A TLB can store entries based on the following four block sizes:

Supersections	Consist of 16MB blocks of memory.
Sections	Consist of 1MB blocks of memory.
Large pages	Consist of 64KB blocks of memory.

Small pages Consist of 4KB blocks of memory.

Supersections, sections, and large pages are supported to permit mapping of a large region of memory while using only a single entry in a TLB. If no mapping for an address is found within the TLB, then the translation table is automatically read by hardware and a mapping is placed in the TLB. See *Hardware page table translation* on page 6-35 for more details.

6.3.2 Virtual to physical translation mapping restrictions

You can use the ARM1136JF-S MMU architecture in conjunction with virtually indexed physically tagged caches. For details of any mapping page table restrictions for virtual to physical addresses see *Restrictions on page table mappings* on page 6-41.

6.3.3 Tightly-Coupled Memory

There are no page table restrictions for mappings to the *Tightly-Coupled Memory* (TCM). For details of the TCM see *Tightly-coupled memory* on page 7-8.

6.4 Enabling and disabling the MMU

You can enable and disable the MMU by writing the M bit, bit 0, of the CP15 Control Register c1. On reset, this bit is cleared to 0, disabling the MMU.

6.4.1 Enabling the MMU

Before you enable the MMU you must:

1. Program all relevant CP15 registers. This includes setting up suitable translation tables in memory.
2. Disable and invalidate the Instruction Cache. You can then re-enable the Instruction Cache when you enable the MMU.

To enable the MMU proceed as follows:

1. Program the Translation Table Base and Domain Access Control Registers.
2. Program first-level and second-level descriptor page tables as required.
3. Enable the MMU by setting bit 0 in the CP15 Control Register.

6.4.2 Disabling the MMU

To disable the MMU proceed as follows:

1. Clear bit 2 in the CP15 Control Register c1. The Data Cache must be disabled prior to, or at the same time as the MMU being disabled, by clearing bit 2 of the Control Register.

———— **Note** —————

If the MMU is enabled, then disabled, and subsequently re-enabled, the contents of the TLBs are preserved. If these are now invalid, you must invalidate the TLBs before the MMU is re-enabled (see TLB Operations Register c8 on page 2-23).

2. Clear bit 0 in the CP15 Control Register c1.

When the MMU is disabled, memory accesses are treated as follows:

- All data accesses are treated as Noncacheable. The value of the C bit, bit 2, of the CP15 Control Register c1 Should Be Zero.
- All instruction accesses are treated as Cacheable if the I bit, bit 12, of the CP15 Control Register c1 is set to 1, and Noncacheable if the I bit is set to 0.

- All explicit accesses are Strongly Ordered. The value of the W bit, bit 3, of the CP15 Control Register c1 is ignored.
- No memory access permission checks are performed, and no aborts are generated by the MMU.
- The physical address for every access is equal to its virtual address. This is known as a flat address mapping.
- The FCSE PID Should Be Zero when the MMU is disabled. This is the reset value of the FCSE PID. If the MMU is to be disabled the FCSE PID must be cleared.
- All change of program flow prediction is disabled. The state of the Z bit, bit 11, of the CP15 Control Register c1 is ignored. This prevents speculative fetches before the memory region types are defined, protecting read-sensitive I/O locations.
- All CP15 MMU and cache operations work as normal when the MMU is disabled.
- Instruction and data prefetch operations work as normal. However, the Data Cache cannot be enabled when the MMU is disabled. Therefore a data prefetch operation has no effect. Instruction prefetch operations have no effect if the Instruction Cache is disabled. No memory access permissions are performed and the address is flat mapped.
- Accesses to the TCMs work as normal if the TCMs are enabled.

6.5 Memory access control

Access to a memory region is controlled by

- *Domains*
- *Access permissions* on page 6-12
- *Execute never bits in the TLB entry* on page 6-13.

6.5.1 Domains

A domain is a collection of memory regions. The ARM architecture supports 16 domains. Domains provide support for multi-user operating systems. All regions of memory have an associated domain.

A domain is the primary access control mechanism for a region of memory and defines the conditions in which an access can proceed. The domain determines whether:

- access permissions are used to qualify the access
- access is unconditionally allowed to proceed
- access is unconditionally aborted.

In the latter two cases, the access permission attributes are ignored.

Each page table entry and TLB entry contains a field that specifies which domain the entry is in. Access to each domain is controlled by a 2-bit field in the Domain Access Control Register, CP15 c3. Each field enables very quick access to be achieved to an entire domain, so that whole memory areas can be efficiently swapped in and out of virtual memory. Two kinds of domain access are supported:

Clients Clients are users of domains in that they execute programs and access data. They are guarded by the access permissions of the TLB entries for that domain.

A client is a domain user, and each access has to be checked against the access permission settings for each memory block and the system protection bit, the S bit, and the ROM protection bit, the R bit, in CP15 Control Register c1. Table 6-1 on page 6-12 shows the access permissions.

Managers Managers control the behavior of the domain, the current sections and pages in the domain, and the domain access. They are not guarded by the access permissions for TLB entries in that domain.

Because a manager controls the domain behavior, each access has only to be checked to be a manager of the domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This enables flexible memory protection for programs that access different memory resources.

6.5.2 Access permissions

The access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then a permission fault is raised.

The access permissions are determined by a combination of the AP and APX bits in the page table, and the S and R bits in CP15 Control Register c1. For page tables not supporting the APX bit, the value 0 is used.

Changes to the S and R bits do not affect the access permissions of entries already in the TLB. You must flush the TLB to enable the updated S and R bits to take effect.

———— **Note** ————

The use of the S and R bits is deprecated.

The encoding of the access permission bits is shown in Table 6-1.

Table 6-1 Access permission bit encoding

S	R	APX	AP[1:0]	Privileged permissions	User permissions	Description
0	0	0	b00	No access	No access	All accesses generate a permission fault
x	x	0	b01	Read/write	No access	Privileged access only
x	x	0	b10	Read/write	Read-only	Writes in User mode generate permission faults
x	x	0	b11	Read/write	Read/write	Full access
0	0	1	b00	-	-	Reserved
0	0	1	b01	Read-only	No access	Privileged read-only
0	0	1	b10	Read-only	Read-only	Privileged/User read-only
0	0	1	b11	-	-	Reserved
0	1	0	b00	Read-only	Read-only	Privileged/User read-only
1	0	0	b00	Read-only	No access	Privileged read-only
1	1	0	b00	-	-	Reserved

Table 6-1 Access permission bit encoding (continued)

S	R	APX	AP[1:0]	Privileged permissions	User permissions	Description
0	1	1	xx	-	-	Reserved
1	0	1	xx	-	-	Reserved
1	1	1	xx	-	-	Reserved

6.5.3 Execute never bits in the TLB entry

Each memory region can be tagged as not containing executable code. If the Execute Never, XN, bit of the TLB Attributes Entry Register, CP15 c10, is set to 1, then any attempt to execute an instruction in that region results in a permission fault. If the XN bit is cleared to 0, then code can execute from that memory region. see *TLB Attribute Registers* on page 3-47 for more details.

6.6 Memory region attributes

Each TLB entry has an associated set of memory region attributes. These control:

- accesses to the caches
- how the write buffer is used
- if the memory region is shareable and must be kept coherent.

6.6.1 C and B bit, and type extension field encodings

Page table formats use five bits to encode the memory region type. These are **TEX[2:0]**, and the C and B bits. Table 6-2 shows the mapping of the *Type Extension Field* (TEX) and the Cachable and Bufferable bits (C and B) to memory region type. For page tables formats with no TEX field you must use the value b000.

Additionally certain page tables contain the shared bit, S. This bit only applies to Normal, not Device or Strongly Ordered memory, and determines if the memory region is Shared (1), or Non-Shared (0). If not present the S bit is assumed to be 0 (Non-Shared).

Table 6-2 TEX field, and C and B bit encodings used in page table formats

Page table encodings			Description	Memory type	Page shareable?
TEX	C	B			
b000	0	0	Strongly Ordered	Strongly Ordered	Shared ^a
b000	0	1	Shared Device	Device	Shared ^a
b000	1	0	Outer and Inner Write-Through, No Allocate on Write	Normal	s ^b
b000	1	1	Outer and Inner Write-Back, No Allocate on Write	Normal	s ^b
b001	0	0	Outer and Inner Noncachable	Normal	s ^b
b001	0	1	Reserved	-	-
b001	1	0	Reserved	-	-
b001	1	1	Reserved	-	-
b010	0	0	Non-Shared Device	Device	Non-shared
b010	0	1	Reserved	-	-

Table 6-2 TEX field, and C and B bit encodings used in page table formats (continued)

Page table encodings			Description	Memory type	Page shareable?
TEX	C	B			
010	1	X	Reserved	-	-
011	X	X	Reserved	-	-
1BB	A	A	Cached memory. BB = Outer policy, AA = Inner policy. See Table 6-3.	Normal	s ^b

- a. Shared, regardless of the value of the S bit in the page table.
b. s is Shared if the value of the S bit in the page table is 1, or Non-shared if the value of the S bit is 0 or not present.

The Inner and Outer cache policy bits AA (C and B bits) and BB (TEX[1:0]) control the operation of memory accesses to the external memory. Table 6-3 indicates how the MMU and cache interpret the cache policy bits.

Table 6-3 Cache policy bits

TEX[1:0] (BB) or CB (AA) bits	Cache policy
b00	Noncachable, Unbuffered
b01	Write-Back cached, Write Allocate, Buffered
b10	Write-Through cached, No Allocate on Write, Buffered
b11	Write-Back cached, No Allocate on Write, Buffered

The terms Inner and Outer refer to levels of caches that can be built in a system. Inner refers to the innermost caches, including level one. Outer refers to the outermost caches. The boundary between Inner and Outer caches is defined in the implementation of a cached system. Inner must always include level one. In a system with three levels of caches, an example is for the Inner attributes to apply to level one and level two, while the Outer attributes apply to level three. In a two-level system, it is envisaged that Inner always applies to level one and Outer to level two.

In ARM1136JF-S processors, Inner refers to level one and **HPROT** shows the Outer Cachable properties. The **HSIDEBAND** signals show the Inner Cachable values.

For an explanation of Strongly Ordered and Device see *Memory attributes and types* on page 6-17.

You can choose which write allocation policy an implementation supports. The Allocate On Write and No Allocate On Write cache policies indicate which allocation policy is preferred for a memory region, but you must not rely on the memory system implementing that policy. ARM1136JF-S processors do not support Inner Allocate on Write.

Not all Inner and Outer cache policies are mandatory. Table 6-4 gives possible implementation options.

Table 6-4 Inner and Outer cache policy implementation options

Cache policy	Implementation options	Supported by ARM1136JF-S processors?
Inner Noncachable	Mandatory.	Yes
Inner Write-Through	Mandatory.	Yes
Inner Write-Back	Optional. If not supported, the memory system must implement this as Inner Write-Through.	Yes
Outer Noncachable	Mandatory.	System-dependent
Outer Write-Through	Optional. If not supported, the memory system must implement this as Outer Noncachable.	System-dependent
Outer Write-Back	Optional. If not supported, the memory system must implement this as Outer Write-Through.	System-dependent

6.6.2 Shared

This bit indicates that the memory region can be shared by multiple processors. For a full explanation of the Shared attribute see *Memory attributes and types* on page 6-17.

6.7 Memory attributes and types

The ARM1136JF-S processor provides a set of memory attributes that have characteristics that are suited to particular devices, including memory devices, that can be contained in the memory map. The ordering of accesses for regions of memory is also defined by the memory attributes. There are three mutually exclusive main memory type attributes:

- Strongly Ordered
- Device
- Normal.

These are used to describe the memory regions. The marking of the same memory locations as having two different attributes in the MMU, for example using synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

A summary of the memory attributes is shown in Table 6-5.

Table 6-5 Memory attributes

Memory type attribute	Shared/ Non-shared	Other attributes	Description
Strongly Ordered	-	-	All memory accesses to Strongly Ordered memory occur in program order. Some backwards compatibility constraints exist with ARMv5 instructions that change the CPSR interrupt masks (see <i>Strongly Ordered memory attribute</i> on page 6-21). All Strongly Ordered accesses are assumed to be shared.
Device	Shared	-	Designed to handle memory-mapped peripherals that are shared by several processors.
	Non-shared	-	Designed to handle memory-mapped peripherals that are used only by a single processor.
Normal	Shared	Noncachable/ Write-Through Cachable/ Write-Back Cachable	Designed to handle normal memory that is shared between several processors.
	Non-shared	Noncachable/ Write-Through Cachable/ Write-Back Cachable	Designed to handle normal memory that is used only by a single processor.

6.7.1 Normal memory attribute

The Normal memory attribute is defined on a per-page basis in the MMU and provides memory access orderings that are suitable for normal memory. This type of memory stores information without side effects. Normal memory can be writable or read-only.

For writable normal memory, unless there is a change to the physical address mapping:

- a load from a specific location returns the most recently stored data at that location for the same processor
- two loads from a specific location, without a store in between, return the same data for each load.

For read-only normal memory:

- two loads from a specific location return the same data for each load.

This behavior describes most memory used in a system, and the term memory-like is used to describe this sort of memory. In this section, writable normal memory and read-only normal memory are not distinguished.

Regions of memory with the Normal attribute can be Shared or Non-Shared, on a per-page basis in the MMU. The marking of the same memory locations as being Shared Normal and Non-Shared Normal in the MMU, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

All explicit accesses to memory marked as Normal must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page 6-22. Accesses to Normal memory conform to the Weakly Ordered model of memory ordering. A description of this model is in standard texts describing memory ordering issues.

Shared Normal memory

The Shared Normal memory attribute is designed to describe normal memory that can be accessed by multiple processors or other system masters.

A region of memory marked as Shared Normal is one in which the effect of interposing a cache, or caches, on the memory system is entirely transparent. Implementations can use a variety of mechanisms to support this, from not caching accesses in shared regions to more complex hardware schemes for cache coherency for those regions.

ARM1136JF-S processors do not cache shareable locations at level one.

In systems that implement a TCM, the regions of memory covered by the TCM must not be marked as Shared. Marking an area of memory covered by the TCM as being Shared results in Unpredictable behavior.

Writes to Shared Normal memory might not be atomic. That is, all observers might not see the writes occurring at the same time. To preserve coherence where two writes are made to the same location, the order of those writes must be seen to be the same by all observers. Reads to Shared Normal memory that are aligned in memory to the size of the access are atomic.

Non-Shared Normal memory

The Non-Shared Normal memory attribute describes normal memory that can be accessed only by a single processor.

A region of memory marked as Non-Shared Normal does not have any requirement to make the effect of a cache transparent.

Cachable Write-Through, Cachable Write-Back, and Noncachable

In addition to marking a region of Normal memory as being Shared or Non-Shared, a region of memory marked as Normal can also be marked on a per-page basis in an MMU as being one of:

- Cachable Write-Through
- Cachable Write-Back
- Noncachable.

This marking is independent of the marking of a region of memory as being Shared or Non-Shared, and indicates the required handling of the data region for reasons other than those to handle the requirements of shared data. As a result, it is acceptable for a region of memory that is marked as being Cachable and Shared not to be held in the cache in an implementation that handles Shared regions as not caching the data.

The marking of the same memory locations as having different Cachable attributes, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

6.7.2 Device memory attribute

The Device memory attribute is defined for memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Memory-mapped peripherals and I/O locations are typical examples of areas of memory that you must mark as Device. The marking of a region of memory as Device is performed on a per-page basis in the MMU.

Accesses to memory-mapped locations that have side effects that apply to memory locations that are Normal memory might require Memory Barriers to ensure correct execution. An example where this might be an issue is the programming of the control registers of a memory controller while accesses are being made to the memories controlled by the controller.

Instruction fetches must not be performed to areas of memory containing read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses. As a result, instruction fetches from such devices can result in Unpredictable behavior. Up to 64 bytes can be prefetched sequentially ahead of the current instruction being executed. To enable this, read-sensitive devices must be located in memory in such a way to allow for this prefetching.

Explicit accesses from the processor to regions of memory marked as Device occur at the size and order defined by the instruction. The number of location accesses is specified by the program. Repeat accesses to such locations when there is only one access in the program, that is the accesses are not restartable, are not possible in the ARM1136JF-S processor. An example of where a repeat access might be required is before and after an interrupt to enable the interrupt to abandon a slow access. You must ensure these optimizations are not performed on regions of memory marked as Device.

If a memory operation that causes multiple transactions (such as an LDM or an unaligned memory access) crosses a 4KB address boundary, then it can perform more accesses than are specified by the program, regardless of one or both of the areas being marked as Device. For this reason, accesses to volatile memory devices must not be made using single instructions that cross a 4KB address boundary. This restriction is expected to cause restrictions to the placing of such devices in the memory map of a system, rather than to cause a compiler to be aware of the alignment of memory accesses.

In addition, address locations marked as Device are not held in a cache.

6.7.3 Shared memory attribute

Regions of Memory marked as Device are further distinguished by the Shared attribute in the MMU. These memory regions can be marked as:

- Shared Device
- Non-Shared Device.

Explicit accesses to memory with each of the sets of attributes occur in program order relative to other explicit accesses to the same set of attributes.

All explicit accesses to memory marked as Device must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page 6-22.

The marking of the same memory location as being Shared Device and Non-Shared Device in an MMU, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

An example of an implementation where the Shared attribute is used to distinguish memory accesses is an implementation that supports a local bus for its private peripherals, while system peripherals are situated on the main system bus. Such a system can have more predictable access times for local peripherals such as watchdog timers or interrupt controllers.

For shared device memory, the data of a write is visible to all observers before the end of a Drain Write Buffer memory barrier. For non-shared device memory, the data of a write is visible to the processor before the end of a Drain Write Buffer memory barrier (see *Explicit Memory Barriers* on page 6-24).

6.7.4 Strongly Ordered memory attribute

A further memory attribute, Strongly Ordered, is defined on a per-page basis in the MMU. Accesses to memory marked as Strongly Ordered have a strong memory-ordering model with respect to all explicit memory accesses from that processor. An access to memory marked as Strongly Ordered acts as a memory barrier to all other explicit accesses from that processor, until the point at which the access is complete (that is, has changed the state of the target location or data has been returned). In addition, an access to memory marked as Strongly Ordered must complete before the end of a Memory Barrier (see *Explicit Memory Barriers* on page 6-24).

To maintain backwards compatibility with ARMv5 architecture, any ARMv5 instructions that implicitly or explicitly change the interrupt masks in the CSPR that appear in program order after a Strongly Ordered access must wait for the Strongly Ordered memory access to complete. These instructions are MSR with the control field mask bit set, and the flag setting variants of arithmetic and logical instructions whose destination register is r15, which copies the SPSR to CSPR. This requirement exists only for backwards compatibility with previous versions of the ARM architecture, and the behavior is deprecated in ARMv6. Programs must not rely on this behavior, but instead include an explicit Memory Barrier (see *Explicit Memory Barriers* on page 6-24) between the memory access and the following instruction.

The ARM1136JF-S processor does not require an explicit memory barrier in this situation, but for future compatibility it is recommended that programmers insert a memory barrier.

Explicit accesses from the processor to memory marked as Strongly Ordered occur at their program size, and the number of accesses that occur to such locations is the number that are specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program (that is, the accesses are not restartable).

If a memory operation that causes multiple transactions (such as LDM or an unaligned memory access) crosses a 4KB address boundary, then it might perform more accesses than are specified by the program regardless of one or both of the areas being marked as Strongly Ordered. For this reason, it is important that accesses to volatile memory devices are not made using single instructions that cross a 4KB address boundary.

Address locations marked as Strongly Ordered are not held in a cache, and are treated as Shared memory locations.

For Strongly Ordered memory, the data and side effects of a write are visible to all observers before the end of a Drain Write Buffer memory barrier (see *Explicit Memory Barriers* on page 6-24).

6.7.5 Ordering requirements for memory accesses

The various memory types defined in this section have restrictions in the memory orderings that are allowed.

Ordering requirements for two accesses

The order of any two explicit architectural memory accesses where one or more are to memory marked as Non-Shared must obey the ordering requirements shown in Table 6-6 on page 6-23.

Table 6-6 on page 6-23 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order.

The symbols used in the table are as follows:

- < Accesses must occur strictly in program order. That is, A1 must occur strictly before A2. It must be impossible to tell otherwise from observation of the read/write values and side effects caused by the memory accesses.
- ? Accesses can occur in any order, provided that the requirements of uniprocessor semantics are met, for example respecting dependencies between instructions within a single processor.

Table 6-6 Memory ordering restrictions

A1	A2							
	Normal read	Device read		Strongly Ordered read	Normal write	Device write		Strongly Ordered write
		Non-Shared	Shared			Non-Shared	Shared	
Normal read	<	?	<	<	? ^a	?	<	<
Device read (Non-Shared)	?	<	?	<	?	<	?	<
Device read (Shared)	<	?	<	<	?	?	<	<
Strongly Ordered read	<	<	<	<	<	<	<	<
Normal write	?	?	?	<	<	?	<	<
Device write (Non-Shared)	?	<	?	<	?	<	?	<
Device write (Shared)	<	?	<	<	<	?	<	<
Strongly Ordered write	<	<	<	<	<	<	<	<

a. ARM1136JF-S processor orders the normal read ahead of normal write.

There are no ordering requirements for implicit accesses to any type of memory.

Definition of program order of memory accesses

The program order of instruction execution is defined as the order of the instructions in the control flow trace.

Two explicit memory accesses in an execution can either be:

Ordered Denoted by <. If the accesses are Ordered, then they must occur strictly in order.

Weakly Ordered Denoted by <=. If the accesses are Weakly Ordered, then they must occur in order or simultaneously.

The rules for determining this for two accesses A1 and A2 are:

1. If A1 and A2 are generated by two different instructions, then:
 - $A1 < A2$ if the instruction that generates A1 occurs before the instruction that generates A2 in program order.
 - $A2 < A1$ if the instruction that generates A2 occurs before the instruction that generates A1 in program order.
2. If A1 and A2 are generated by the same instruction, then:
 - If A1 and A2 are the load and store generated by a SWP or SWPB instruction, then:
 - $A1 < A2$ if A1 is the load and A2 is the store
 - $A2 < A1$ if A2 is the load and A1 is the store.
 - If A1 and A2 are two word loads generated by an LDC, LDRD, or LDM instruction, or two word stores generated by an STC, STRD, or STM instruction, but excluding LDM or STM instructions whose register list includes the PC, then:
 - $A1 \leq A2$ if the address of A1 is less than the address of A2
 - $A2 \leq A1$ if the address of A2 is less than the address of A1.
 - If A1 and A2 are two word loads generated by an LDM instruction whose register list includes the PC or two word stores generated by an STM instruction whose register list includes the PC, then the program order of the memory operations is not defined.

Multiple load and store instructions (such as LDM, LDRD, STM, and STRD) generate multiple word accesses, each being a separate access to determine ordering.

6.7.6 Explicit Memory Barriers

Two explicit Memory Barrier operations are described in this section:

- Data Memory Barrier
- Drain Write Buffer.

In addition, to ensure correct operation where the processor writes code, an explicit Flush Prefetch Buffer operation is provided.

These operations are implemented by writing to the CP15 Cache operation register *c7*. For details on how to use this register see *Cache Operations Register* on page 3-17.

Data Memory Barrier

This memory barrier ensures that all explicit memory transactions occurring in program order before this instruction are completed. No explicit memory transactions occurring in program order after this instruction are started until this instruction completes. Other instructions can complete out of order with the Data Memory Barrier instruction.

Drain Write Buffer

This memory barrier completes when all explicit memory transactions occurring in program order before this instruction are completed. No explicit memory transactions occurring in program order after this instruction are started until this instruction completes. In fact, no instructions occurring in program order after the Drain Write Buffer complete, or change the interrupt masks, until this instruction completes.

For Shared Device and Normal memory, the data of a write is visible to all observers before the end of a Drain Write Buffer memory barrier. For Strongly Ordered memory, the data and the side effects of a write are visible to all observers before the end of a Drain Write Buffer memory barrier. For Non-Shared Device and Normal memory, the data of a write is visible to the processor before the end of a Drain Write Buffer memory barrier.

Flush Prefetch Buffer

The Flush Prefetch Buffer instruction flushes the pipeline in the processor, so that all instructions following the pipeline flush are fetched from memory, including the cache, after the instruction has been completed. Combined with Drain Write Buffer, and potentially invalidating the Instruction Cache, this ensures that any instructions written by the processor are executed. This guarantee is required as part of the mechanism for handling self-modifying code. The execution of a Drain Write Buffer instruction and the invalidation of the Instruction Cache and Branch Target Cache are also required for the handling of self-modifying code.

The Flush Prefetch Buffer is guaranteed to perform this function, while alternative methods of performing the same task, such as a branch instruction, can be optimized in the hardware to avoid the pipeline flush (for example, by using a branch predictor).

Memory synchronization primitives

Memory synchronization primitives exist to ensure synchronization between different processes, which might be running on the same processor or on different processors. You can use memory synchronization primitives in regions of memory marked as

Shared and Non-Shared when the processes to be synchronized are running on the same processor. You must only use them in Shared areas of memory when the processes to be synchronized are running on different processors.

6.7.7 Backwards compatibility

The ARMv6 memory attributes are significantly different from those in previous versions of the architecture. Table 6-7 shows the interpretation of the earlier memory types in the light of this definition.

Table 6-7 Memory region backwards compatibility

Previous architectures	ARMv6 attribute
NCNB (Noncachable, Non Bufferable)	Strongly Ordered ^a
NCB (Noncachable, Bufferable)	Shared Device ^a
Write-Through Cachable, Bufferable	Non-Shared Normal (Write-Through Cachable)
Write-Back Cachable, Bufferable	Non-Shared Normal (Write-Back Cachable)

a. Memory locations contained within the TCMs are treated as being Noncachable, rather than Strongly Ordered or Shared Device.

6.8 MMU aborts

Mechanisms that can cause the ARM1136JF-S processor to take an exception because of a memory access are:

MMU fault	The MMU detects a restriction and signals the processor.
Debug abort	Monitor mode debug is enabled and a breakpoint or a watchpoint has been detected.
External abort	The external memory system signals an illegal or faulting memory access.

Collectively these are called *aborts*. Accesses that cause aborts are said to be aborted. If the memory request that aborts is an instruction fetch, then a Prefetch Abort exception is raised if and when the processor attempts to execute the instruction corresponding to the aborted access.

If the aborted access is a data access or a cache maintenance operation, a Data Abort exception is raised.

All Data Aborts, and aborts caused by cache maintenance operations, cause the *Data Fault Status Register* (DFSR) to be updated so that you can determine the cause of the abort.

For all aborts, excluding external aborts, other than on translation, the *Fault Address Register* (FAR) is updated with the address that caused the abort. External Data Aborts, other than on translation, can all be imprecise and therefore the FAR does not contain the address of the abort. See *Imprecise Data Abort mask in the CPSR/SPSR* on page 2-37 for more details on imprecise Data Aborts.

For all Data Aborts that update the FAR, the instruction FAR is also updated with the address of the instruction that caused the abort. For the precise value stored in the IFAR see *Instruction Fault Address Register* on page 3-67.

Note

The IFAR contains the virtual address of the instruction that caused the abort, not the modified virtual address.

For instruction aborts the value of r14 is used by the abort handler to determine the address that caused the abort.

6.8.1 External aborts

External memory errors are defined as those that occur in the memory system other than those that are detected by an MMU. External memory errors are expected to be extremely rare and are likely to be fatal to the running process. An example of an event that can cause an external memory error is an uncorrectable parity or ECC failure on a level two memory structure.

External abort on instruction fetch

Externally generated errors during an instruction prefetch are precise in nature, and are only recognized by the processor if it attempts to execute the instruction fetched from the location that caused the error. The resulting failure is reported in the Instruction Fault Status Register if no higher priority abort (including a Data Abort) has taken place.

The Fault Address Register is not updated on an external abort on instruction fetch.

External abort on data read/write

Externally generated errors during a data read or write can be imprecise. This means that `r14_abt` on entry into the abort handler on such an abort might not hold an address that is related to the instruction that caused the exception. Correspondingly, external aborts can be unrecoverable. See *Aborts* on page 2-35 for more details.

The Fault Address Register is not updated on an imprecise external abort on a data access.

External abort on a hardware page table walk

An external abort occurring on a hardware page table access must be returned with the page table data. Such aborts are precise. The Fault Address Register is updated on an external abort on a hardware page table walk on a data access, but not on an instruction access. The appropriate Fault Status Register indicates that this has occurred.

6.9 MMU fault checking

During the processing of a section or page, the MMU behaves differently because it is checking for faults. The MMU generates four types of fault:

- *Alignment fault* on page 6-31
- *Translation fault* on page 6-31
- *Domain fault* on page 6-31
- *Permission fault* on page 6-31.

Aborts that are detected by the MMU are taken before any external memory access takes place.

Alignment fault checking is enabled by the A bit in the Control Register CP15 c1. Alignment fault checking is independent of the MMU being enabled. Translation, domain, and permission faults are only generated when the MMU is enabled.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU aborts the access and signals the fault condition to the processor. The MMU retains status and address information about faults generated by data accesses in DFSR and FAR, see *Fault status and address* on page 6-33. The MMU does not retain status about faults generated by instruction fetches.

An access violation for a given memory access inhibits any corresponding external access, and an abort is returned to the ARM1136JF-S processor.

6.9.1 Fault checking sequence

Figure 6-1 on page 6-30 shows the fault checking sequence for translation table managed TLB modes.

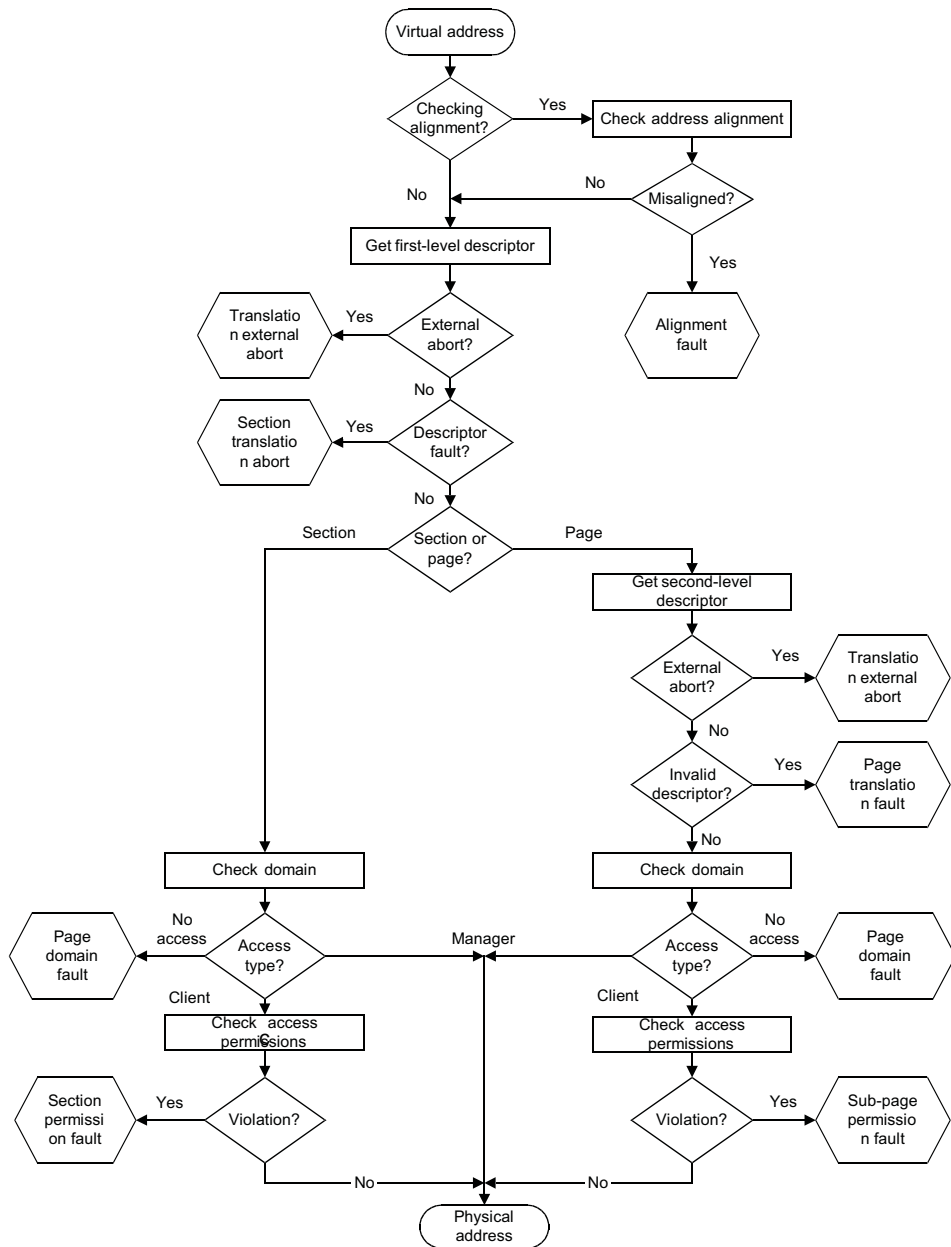


Figure 6-1 Translation table managed TLB fault checking sequence

6.9.2 Alignment fault

An alignment fault occurs if the ARM1136JF-S processor has attempted to access a particular data memory size at an address location that is not aligned with that size.

The conditions for generating Alignment faults are described in *Operation of unaligned accesses* on page 4-18.

Alignment checks are performed with the MMU both enabled and disabled.

6.9.3 Translation fault

There are two types of translation fault:

Section A section translation fault occurs if the first-level translation table descriptor is marked as invalid, bits [1:0] = b00.

Page A page translation fault occurs if the second-level translation table descriptor is marked as invalid, bits [1:0] = b00.

6.9.4 Domain fault

There are two types of domain fault:

Section For a section the domain is checked when the first-level descriptor is returned.

Page For a page the domain is checked when the second-level descriptor is returned.

For each type, the first-level descriptor indicates the domain in CP15 c3, the Domain Access Control Register, to select. If the selected domain has bit 0 set to 0 indicating either no access or reserved, then a domain fault occurs.

6.9.5 Permission fault

If the two-bit domain field returns Client, the access permission check is performed on the access permission field in the TLB entry. A permission fault occurs if the access permission check fails.

6.9.6 Debug event

When monitor mode debug is enabled an abort can be taken caused by a breakpoint on an instruction access or a watchpoint on a data access. In both cases the memory system completes the access before the abort is taken. If an abort is taken when in monitor mode debug then the appropriate FSR (IFSR or DFSR) is updated to indicate a debug abort.

If a watchpoint is taken the IFAR is set to the address that caused the watchpoint. Watchpoints are not taken precisely because following instructions can run underneath load and store multiples. The debugger must read the IFAR to determine which instruction caused the debug event.

6.10 Fault status and address

The encodings for the Fault Status Register are shown in Table 6-8.

Table 6-8 Fault Status Register encoding

Priority	Sources		FSR[10,3:0]	Domain	FAR	
Highest	Alignment		b00001	Invalid	Valid	
	Instruction cache maintenance ^a operation fault		b10100	Invalid	Valid	
	External abort on translation	first-level	b01100	Invalid	Valid	
		second-level	b01110	Valid	Valid	
	Translation	Section	b00101	Invalid	Valid	
		Page	b00111	Valid	Valid	
	Domain	Section	b01001	Valid	Valid	
		Page	b01011	Valid	Valid	
	Permission	Section	b01101	Valid	Valid	
		Page	b01111	Valid	Valid	
	Precise external abort		b01010	Valid	Valid	
	Imprecise external abort		b10110	Invalid	Invalid	
	Lowest	Instruction debug event		b00010	Valid	Valid

a. These aborts cannot be signaled with the IFSR because they do not occur on the instruction side.

Note

All other Fault Status encodings are reserved.

If a translation abort occurs during a Data Cache maintenance operation by virtual address, then a Data Abort is taken and the DFSR indicates the reason. The FAR indicates the faulting address, and the IFAR indicates the address of the instruction causing the abort.

If a translation abort occurs during an Instruction Cache maintenance operation by virtual address, then a Data Abort is taken, and an Instruction Cache Maintenance Operation Fault is indicated in the DFSR. The IFSR indicates the reason. The FAR indicates the faulting address, and the IFAR indicates the address of the instruction causing the abort.

Domain and fault address information is only available for data accesses. For instruction aborts r14 must be used to determine the faulting address. You can determine the domain information by performing a TLB lookup for the faulting address and extracting the domain field.

A summary of which abort vector is taken, and which of the Fault Status and Fault Address Registers are updated for each abort type is shown in Table 6-9.

Table 6-9 Summary of aborts

Abort type	Abort taken	Precise?	Register updated?			
			IFSR	IFAR	DFSR	FAR
Instruction MMU fault	Prefetch Abort	Yes	Yes	No	No	No
Instruction debug abort	Prefetch Abort	Yes	Yes	No	No	No
Instruction external abort on translation	Prefetch Abort	Yes	Yes	No	No	No
Instruction external abort	Prefetch Abort	Yes	Yes	No	No	No
Instruction cache maintenance operation	Data Abort	Yes	Yes	Yes ^a	Yes	Yes
Data MMU fault	Data Abort	Yes	No	Yes ^a	Yes	Yes
Data debug abort	Data Abort	No	No	Yes	Yes	Yes
Data external abort on translation	Data Abort	Yes	No	Yes ^a	Yes	Yes
Data external abort	Data Abort	No ^b	No	No	Yes	No
Data cache maintenance operation	Data Abort	Yes	No	Yes ^a	Yes	Yes

a. Although the IFAR is updated by the processor the behavior is architecturally Unpredictable.

b. Data Aborts can be precise, see *External aborts* on page 6-28 for more details.

6.11 Hardware page table translation

The ARM1136JF-S MMU implements the hardware page table walking mechanism from ARMv4 and ARMv5 cached processors with the exception of the fine page table descriptor.

A hardware page table walk occurs whenever there is a TLB miss. ARM1136JF-S hardware page table walks do not cause a read from the level one Unified/Data Cache, or the TCM. The P, RGN, S, and C bits in the Translation Table Base Registers determine the memory region attributes for the page table walk.

Two formats of page tables are supported:

- A backwards-compatible format supporting subpage access permissions. These have been extended so that certain page table entries support extended region types.
- ARMv6 format, not supporting sub-page access permissions, but with support for ARMv6 MMU features. These features are:
 - extended region types
 - global and process specific pages
 - more access permissions
 - marking of Shared and Non-Shared regions
 - marking of Execute-Never regions.

Additionally two translation table base registers are provided. On a TLB miss, the Translation Table Base Control Register, CP15 c2, and the top bits of the virtual address determine if the first or second translation table base is used. See *Translation Table Base Control Register* on page 3-79 for details. The first-level descriptor indicates whether the access is to a section or to a page table. If the access is to a page table, the ARM1136JF-S MMU fetches a second-level descriptor.

A page table holds 256 32-bit entries 4KB in size. You can determine the page type by examining bits [1:0] of the second-level descriptor.

For both first and second level descriptors if bits [1:0] are b00, the associated virtual addresses are unmapped, and attempts to access them generate a translation fault. Software can use bits [31:2] for its own purposes in such a descriptor, because they are ignored by the hardware. Where appropriate, ARM Limited recommends that bits [31:2] continue to hold valid access permissions for the descriptor.

6.11.1 Backwards-compatible page table translation (subpage AP bits enabled)

When the CP15 Control Register c1 bit 23 is set to 0, the subpage AP bits are enabled and the page table formats are backwards-compatible with ARMv4 and ARMv5 MMU architectures.

All mappings are treated as global, and executable (XN = 0). All Normal memory is Non-Shared. Device memory can be Shared or Non-Shared as determined by the TEX bits and the C and B bits.

For large and small pages, there can be four subpages defined with different access permissions. For a large page, the subpage size is 16KB and is accessed using bits [15:14] of the page index of the virtual address. For a small page, the subpage size is 1KB and is accessed using bits [11:10] of the page index of the virtual address.

The use of subpage AP bits where AP3, AP2, AP1, and AP0 contain different values is deprecated.

Backwards-compatible page table format

Figure 6-2 shows a backwards-compatible format first-level descriptor.

	31	24	23	20	19	18	17	15	14	12	11	10	9	8	5	4	3	2	1	0	
Translation fault	Ignored																		0	0	
Coarse page table	Coarse page table base address														P	Domain		SBZ		0	1
Section (1MB)	Section base address				S	B	Z	0	SBZ	TEX	AP	P	Domain		0	C	B	1	0		
Supersection (16MB)	Supersection base address			SBZ		1	SBZ	TEX	AP	P	Ignored		0	C	B	1	0				
Reserved																			1	1	

Figure 6-2 Backwards-compatible first-level descriptor format

If the P bit is supported and set for the memory region, it indicates to the system memory controller that this memory region has ECC enabled.

Figure 6-3 on page 6-37 shows a backwards-compatible format second-level descriptor for a page table.

	31		16	15		12	11	10	9	8	7	6	5	4	3	2	1	0	
Translation fault	Ignored																	0	0
Large page (64KB)	Large page table base address				TEX	AP3	AP2	AP1	AP0	C	B	0	1						
Small page (4KB)	Small page table base address					AP3	AP2	AP1	AP0	C	B	1	0						
	Extended small page table base address					SBZ	TEX	AP	C	B	1	1							

Figure 6-3 Backwards-compatible second-level descriptor format

For extended small page table entries without a TEX field you must use the value b000.

For details of TEX encodings see *C and B bit, and type extension field encodings* on page 6-14.

Figure 6-4 on page 6-38 shows an overview of the section, supersection, and page translation process using backwards-compatible descriptors.

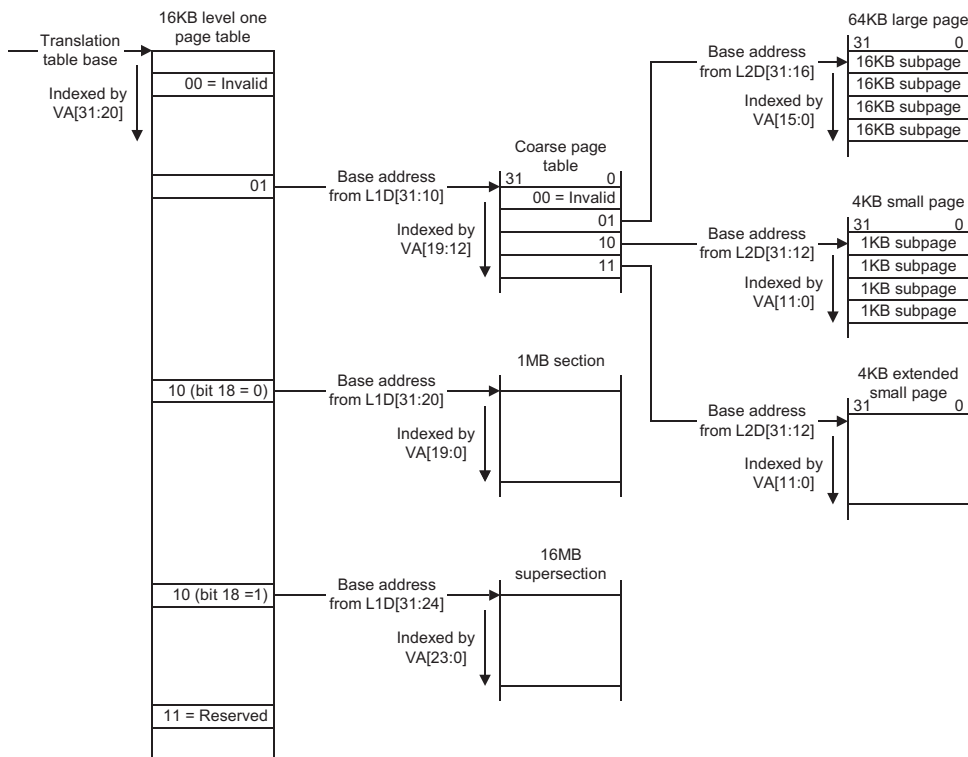


Figure 6-4 Backwards-compatible section, supersection, and page translation

6.11.2 ARMv6 page table translation (subpage AP bits disabled)

When the CP15 Control Register c1 Bit 23 is set to 1, the subpage AP bits are disabled and the page tables have support for ARMv6 MMU features. Four new page table bits are added to support these features:

- The Not-Global (nG) bit, determines if the translation is marked as global (0), or process-specific (1) in the TLB. For process-specific translations the translation is inserted into the TLB using the current ASID, from the ContextID Register, CP15 c13.
- The Shared (S) bit, determines if the translation is for Non-Shared (0), or Shared (1) memory. This only applies to Normal memory regions. Device memory can be Shared or Non-Shared as determined by the TEX bits and the C and B bits.

- The Execute-Never (XN) bit, determines if the region is Executable (0) or Not-executable (1).
- Three access permission bits. The access permissions extension (APX) bit, provides an extra access permission bit.

All ARMv6 page table mappings support the TEX field.

ARMv6 page table format

Figure 6-5 shows the format of an ARMv6 first-level descriptor when subpages are enabled.

	31	24	23	20	19	18	17	15	14	12	11	10	9	8	5	4	3	2	1	0	
Translation fault	Ignored																		0	0	
Coarse page table	Coarse page table base address													P	Domain			SBZ		0	1
Section (1MB)	Section base address					S	B	0	SBZ		TEX	AP	P	Domain			0	C	B	1	0
Supersection (16MB)	Supersection base address			SBZ		1	SBZ		TEX	AP	P	Ignored			0	C	B	1	0		
Reserved																			1	1	

Figure 6-5 ARMv6 first-level descriptor formats with subpages enabled

Figure 6-6 on page 6-40 shows the format of an ARMv6 first-level descriptor when subpages are disabled.

	31	24	23	20	19	18	17	16	15	14	12	11	10	9	8	5	4	3	2	1	0	
Translation fault	Ignored																				0	0
Coarse page table	Coarse page table base address														P	Domain			SBZ		0	1
Section (1MB)	Section base address				S B Z	0	n G	S	A P X	TEX	AP	P	Domain			X N	C	B	1	0		
Supersection (16MB)	Supersection base address			SBZ		1	n G	S	A P X	TEX	AP	P	Ignored			X N	C	B	1	0		
Translation fault	Reserved																				1	1

Figure 6-6 ARMv6 first-level descriptor formats with subpages disabled

If the P bit is supported and set for the memory region, it indicates to the system memory controller that this memory region has ECC enabled.

In addition to the invalid translation, bits [1:0] = b00, translations for the reserved entry, bits [1:0] = b11, result in a translation fault.

Bit 18 of the first-level descriptor selects between a 1MB section and a 16MB supersection. For details of supersections see *Supersections* on page 6-6.

Figure 6-7 shows the format of an ARMv6 second-level descriptor.

	31	16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0	
Translation fault	Ignored																0	0
Large page (64KB)	Large page table base address						X N	TEX	n G	S	A P X	SBZ		AP	C	B	0	1
Small page (4KB)	Extended small page table base address								n G	S	A P X	TEX	AP	C	B	1	X N	

Figure 6-7 ARMv6 second-level descriptor format

Figure 6-8 on page 6-41 shows an overview of the section, supersection, and page translation process using ARMv6 descriptors.

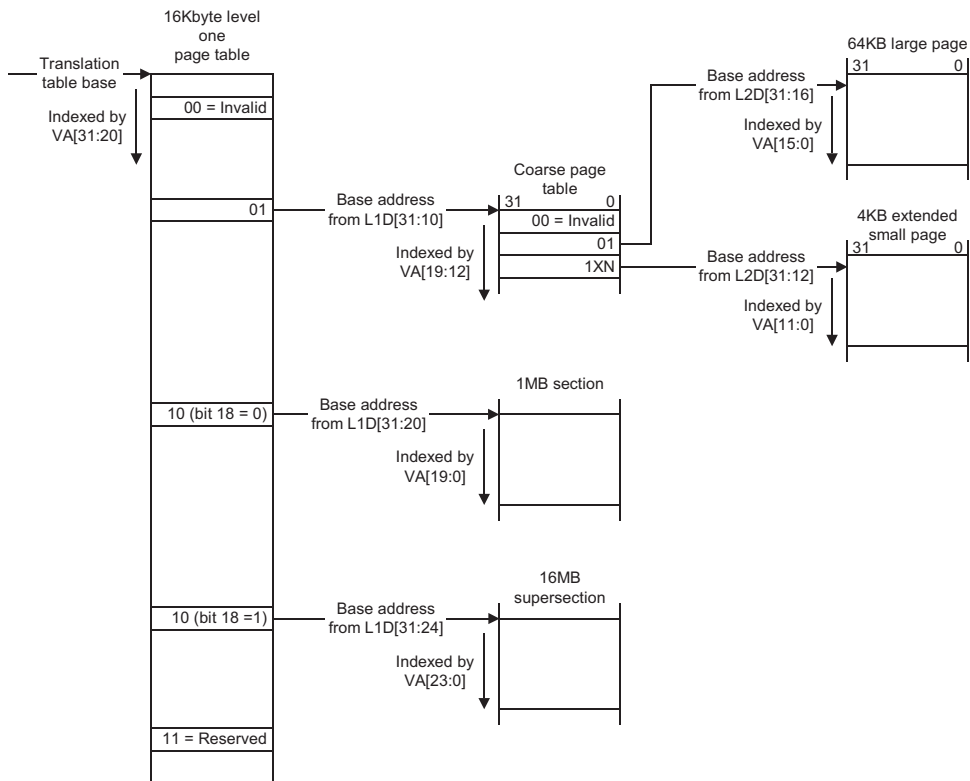


Figure 6-8 ARMv6 section, supersection, and page translation

6.11.3 Restrictions on page table mappings

The ARM1136JF-S processor uses virtually indexed, physically addressed caches. To prevent alias problems where cache sizes greater than 16KB have been implemented, you must restrict the mapping of pages that remap virtual address bits [13:12]. Bits 11 and 23, the P bits for the Instruction and Data Caches in the Cache Type Register CP15 c0, indicate if this is necessary.

This restriction enables these bits of the virtual address to be used to index into the cache without requiring hardware support to avoid alias problems.

For pages marked as Non-Shared, if bit 11 or bit 23 of the Cache Type Register is set, the restriction applies to pages that remap virtual address bits [13:12] and might cause aliasing problems when 4KB pages are used. To prevent this you must ensure the following restrictions are applied:

1. If multiple virtual addresses are mapped onto the same physical address then for all mappings of bits [13:12] the virtual addresses must be equal and the same as bits [13:12] of the physical address. The same physical address can be mapped by TLB entries of different page sizes, including page sizes over 4KB.
2. Alternatively, if all mappings to a physical address are of a page size equal to 4KB, then the restriction that bits [13:12] of the virtual address must equal bits [13:12] of the physical address is not necessary. Bits [13:12] of all virtual address aliases must still be equal.

There is no restriction on the more significant bits in the virtual address equalling those in the physical address.

6.12 MMU descriptors

To support sections and pages, the ARM1136JF-S MMU uses a two-level descriptor definition. The first-level descriptor indicates whether the access is to a section or to a page table. If the access is to a page table, the ARM1136JF-S MMU determines the page table type and fetches a second-level descriptor.

6.12.1 First-level descriptor address

The *Translation Table Base Control Register* (TTBCR) selects between the two possible first-level descriptor addresses created by the two *Translation Table Base Registers* (TTBR0 and TTBR1) and the virtual address from the ARM1136JF-S processor. Figure 6-9 on page 6-44 shows the creation of a first-level descriptor address.

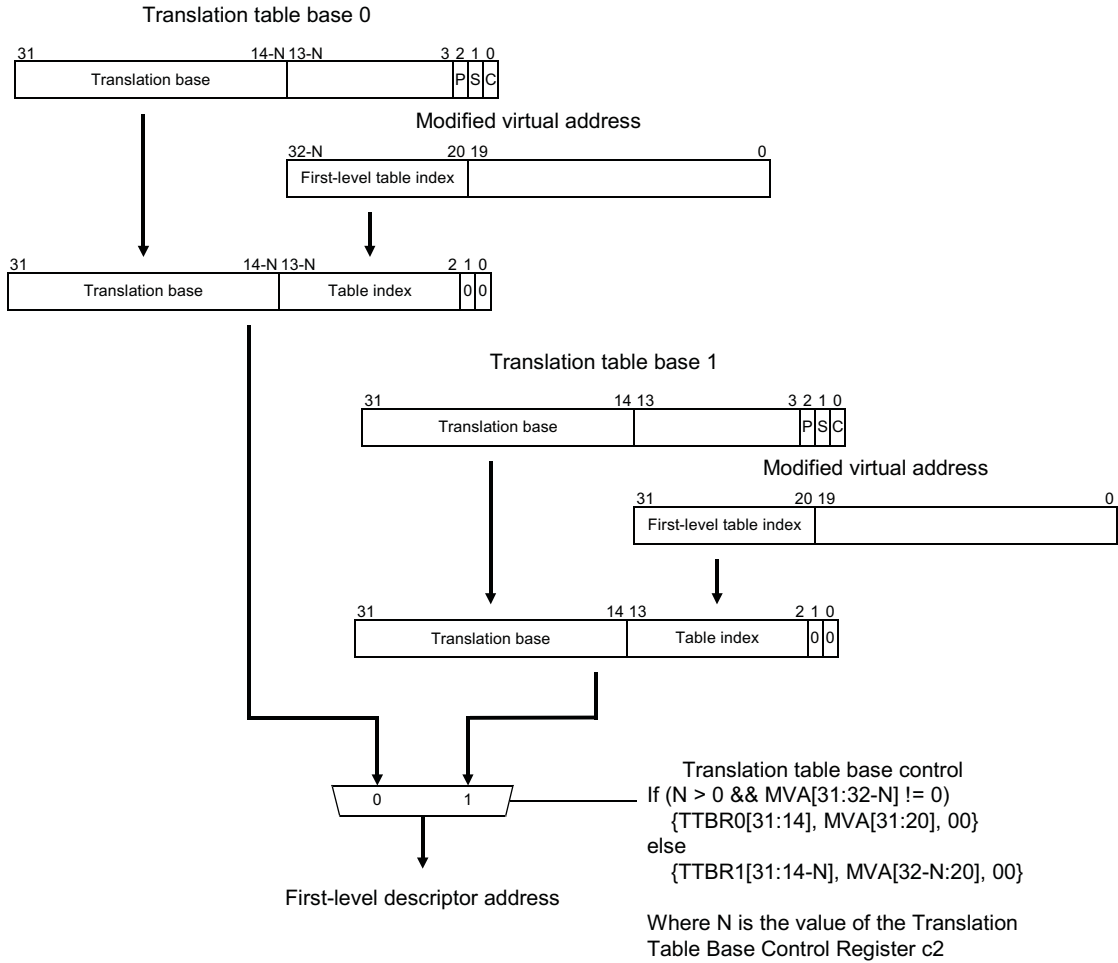


Figure 6-9 Creating a first-level descriptor address

6.12.2 First-level descriptor

Using the first-level descriptor address, a request is made to external memory. This returns the first-level descriptor. By examining bits [1:0] of the first-level descriptor, the access type is indicated as shown in Table 6-10.

Table 6-10 Access types from first-level descriptor bit values

Bit values	Access type
b00	Translation fault
b01	Page table base address
b10	Section base address
b11	Reserved, results in translation fault

First-level translation fault

If bits [1:0] of the first-level descriptor are b00 or b11, a translation fault is generated. This causes either a Prefetch Abort or Data Abort in the ARM1136JF-S processor. Prefetch Aborts occur in the instruction MMU. Data Aborts occur in the data MMU.

First-level page table address

If bits [1:0] of the first-level descriptor are b01, then a page table walk is required. This process is described in *Second-level page table walk* on page 6-47.

First-level section base address

If bits [1:0] of the first-level descriptor are b10, a request to a section memory block has occurred. Figure 6-10 on page 6-46 shows the translation process for a 1MB section using ARMv6 format (AP bits disabled).

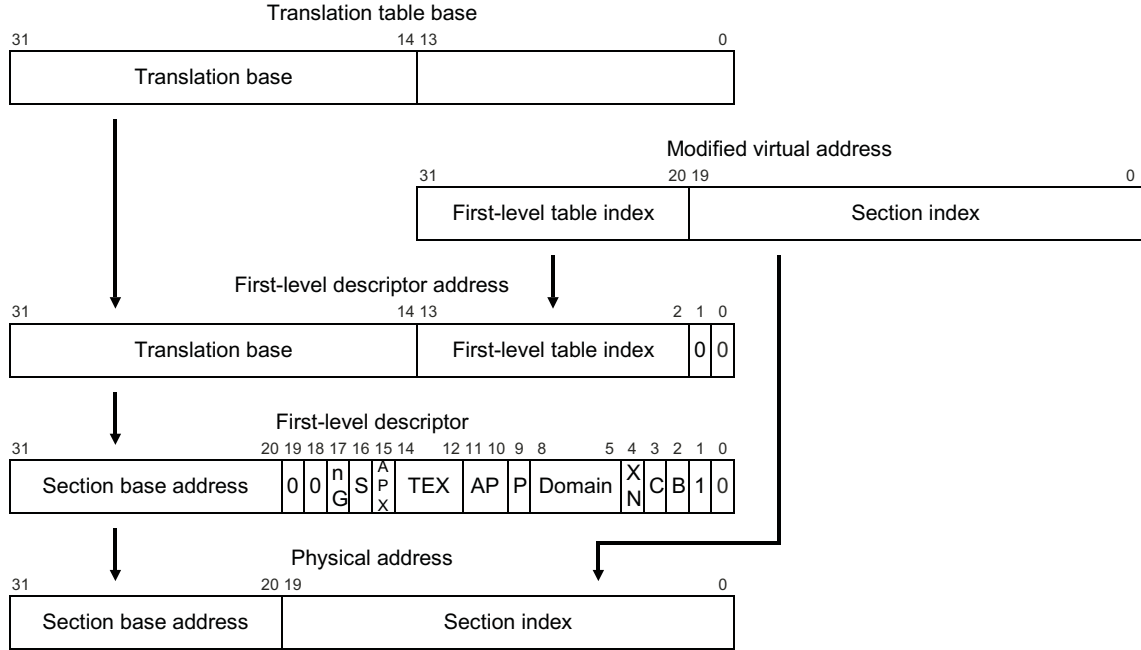


Figure 6-10 Translation for a 1MB section, ARMv6 format

Following the first-level descriptor translation, the physical address is used to transfer to and from external memory the data requested from and to the ARM1136JF-S processor. This is done only after the domain and access permission checks are performed on the first-level descriptor for the section. These checks are described in *Memory access control* on page 6-11.

Figure 6-11 on page 6-47 shows the translation process for a 1MB section using backwards-compatible format (AP bits enabled).

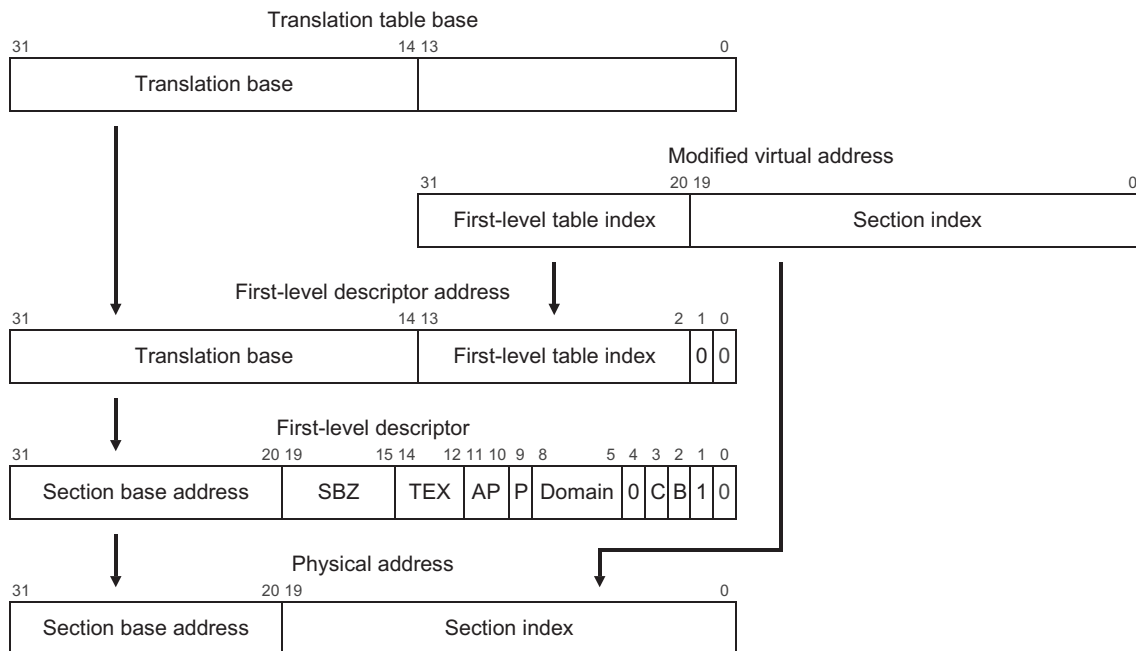


Figure 6-11 Translation for a 1MB section, backwards-compatible format

6.12.3 Second-level page table walk

If bits [1:0] of the first-level descriptor bits are b01, then a page table walk is required. The MMU requests the second-level page table descriptor from external memory. Figure 6-12 on page 6-48 shows how the second-level page table address is generated.

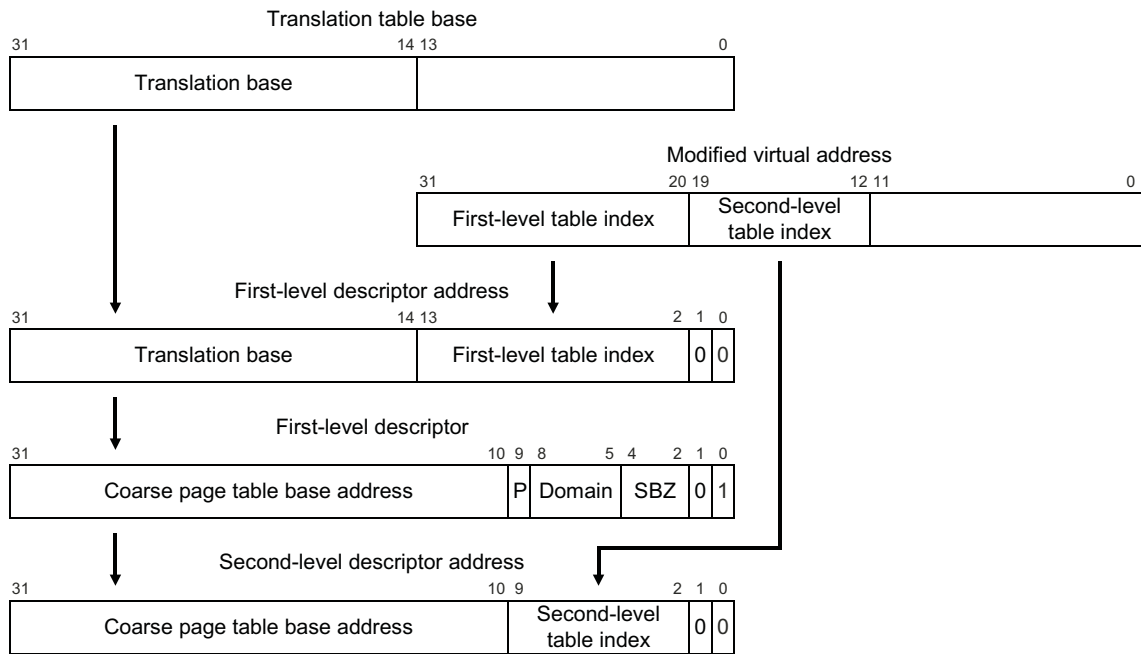


Figure 6-12 Generating a second-level page table address

When the page table address is generated, a request is made to external memory for the second-level descriptor.

By examining bits [1:0] of the second-level descriptor, the access type is indicated as shown in Table 6-11.

Table 6-11 Access types from second-level descriptor bit values

Descriptor format	Bit values	Access type
Both	b00	Translation fault
Backwards-compatible	b01	64KB large page
ARMv6	b01	64KB large page

Table 6-11 Access types from second-level descriptor bit values (continued)

Descriptor format	Bit values	Access type
Backwards-compatible	b10	4KB small page
ARMv6	b1XN	4KB extended small page
Backwards-compatible	b11	4KB extended small page

Second-level translation fault

If bits [1:0] of the second-level descriptor are b00, then a translation fault is generated. This generates an abort to the ARM1136JF-S processor, either a Prefetch Abort for the instruction side or a Data Abort for the data side.

Second-level large page base address

If bits [1:0] of the second-level descriptor are b01, then a large page table walk is required. Figure 6-13 on page 6-50 shows the translation process for a 64KB large page using ARMv6 format (AP bits disabled).

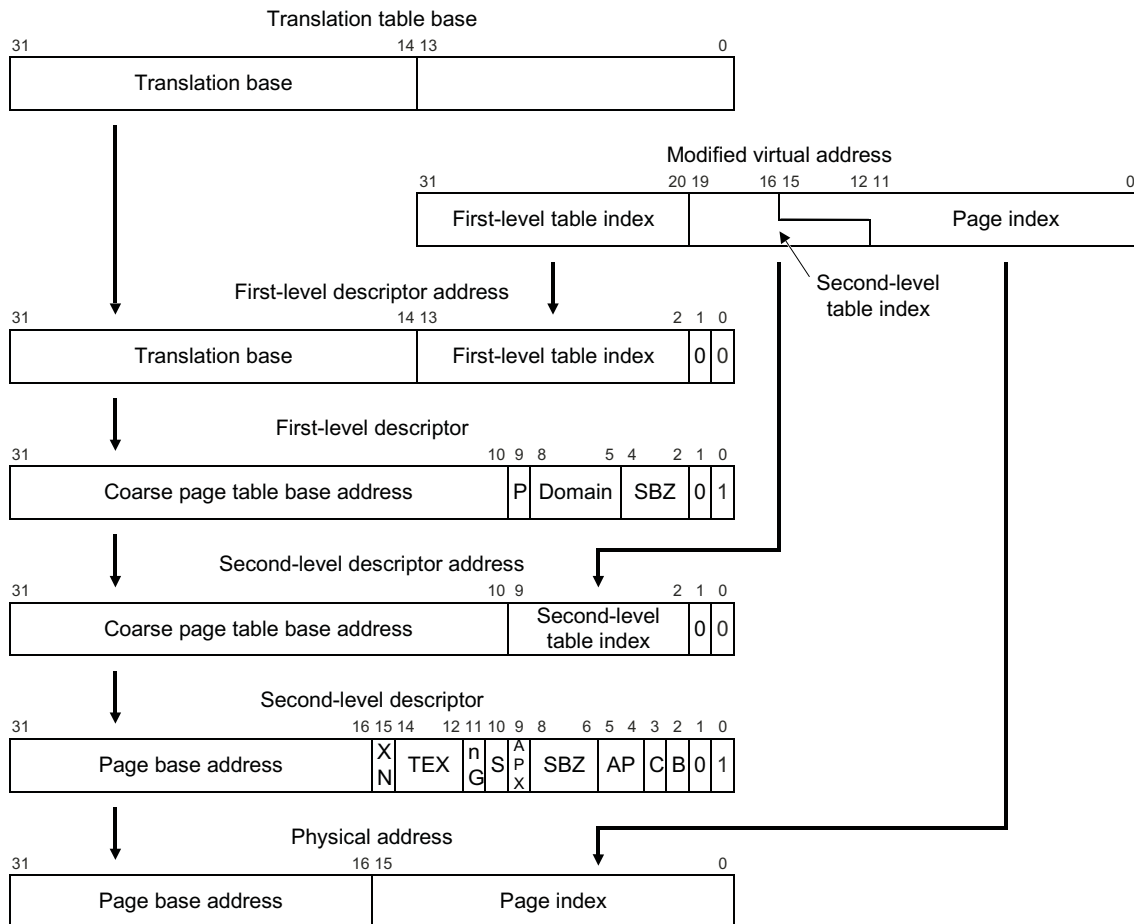


Figure 6-13 Large page table walk, ARMv6 format

Figure 6-14 on page 6-51 shows the translation process for a 64KB large page, or a 16KB large page subpage, using backwards-compatible format (AP bits enabled).

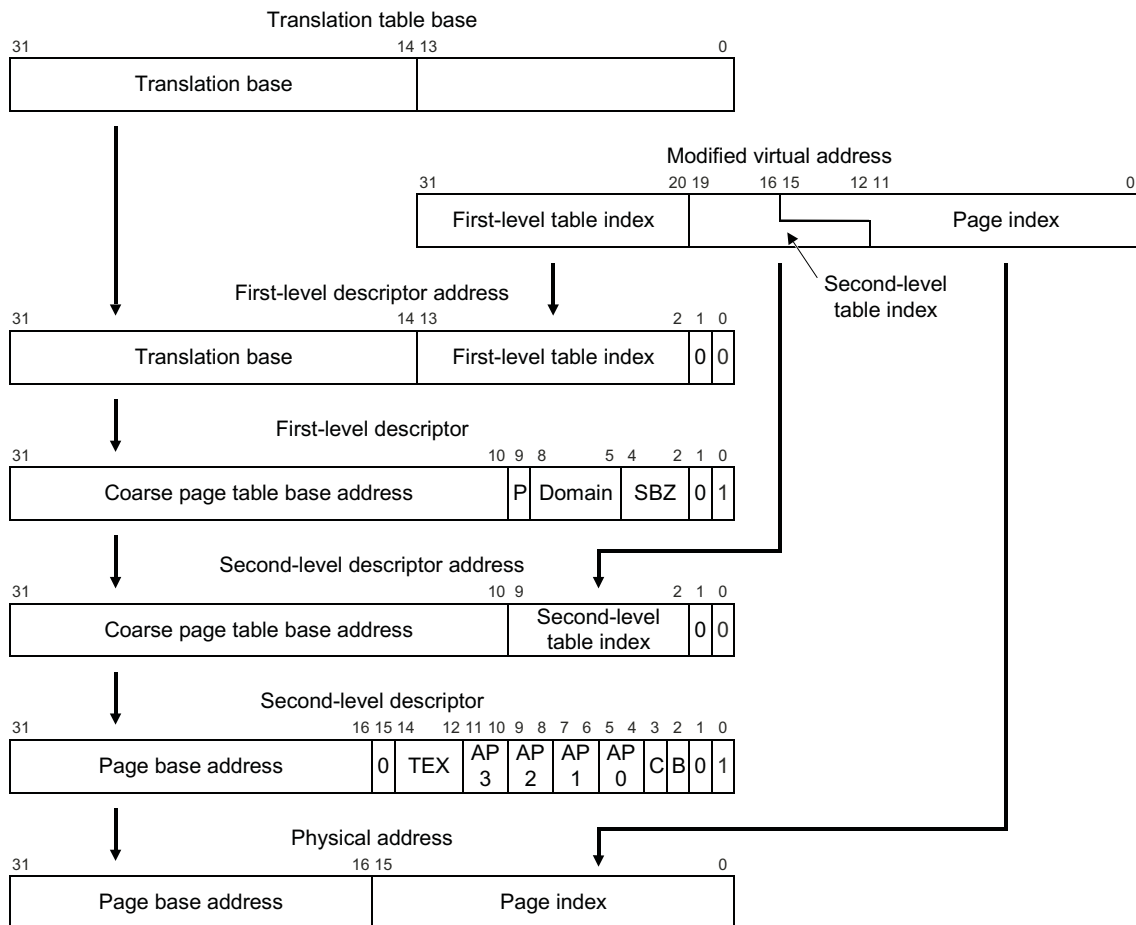


Figure 6-14 Large page table walk, backwards-compatible format

Using backwards-compatible format descriptors, the 64KB large page is generated by setting all of the AP bit pairs to the same values, $AP_3=AP_2=AP_1=AP_0$. If any one of the pairs are different, then the 64KB large page is converted into four 16KB large page subpages. The subpage access permission bits are chosen using the virtual address bits [15:14].

Second-level small page table walk

If bits [1:0] of the second-level descriptor are b10 for backwards-compatible format, then a small page table walk is required.

Figure 6-15 shows the translation process for a 4KB small page or a 1KB small page subpage using backwards-compatible format descriptors (AP bits enabled).

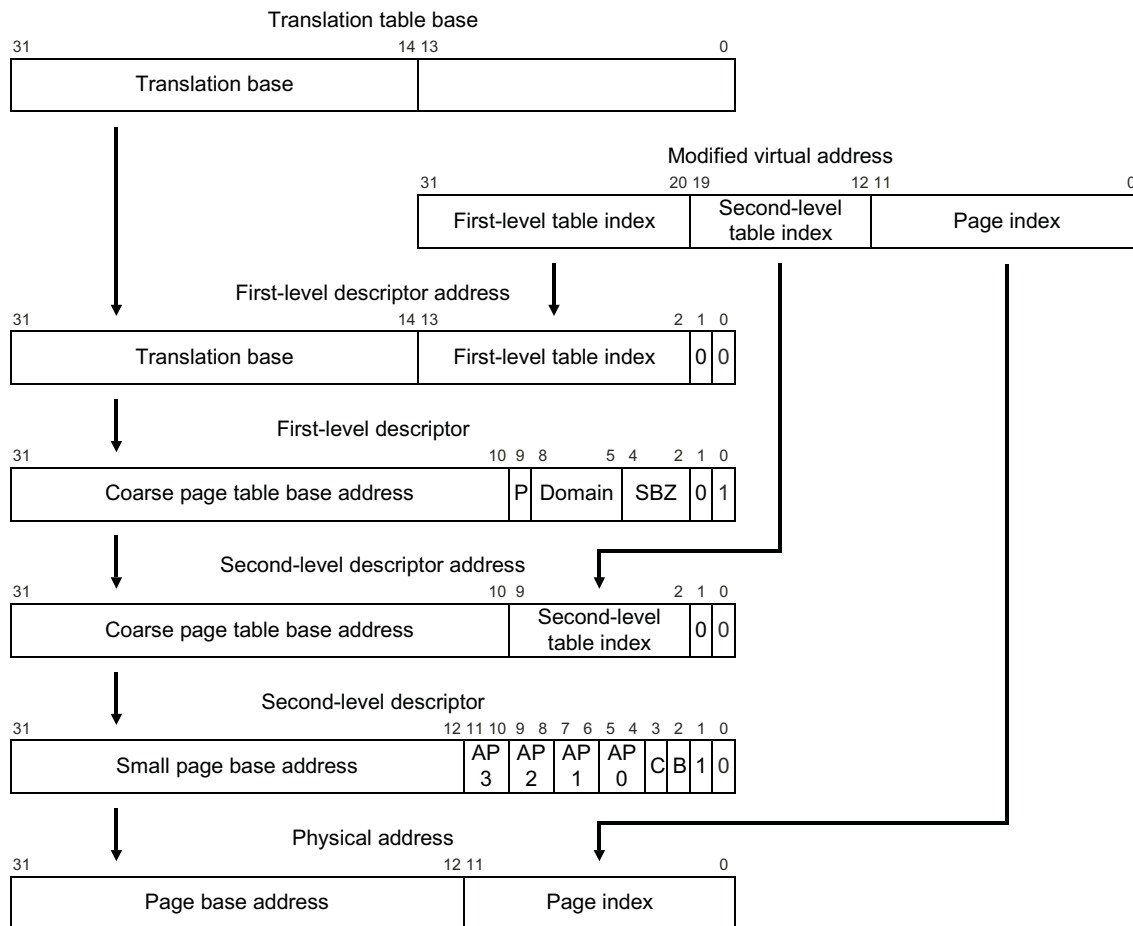


Figure 6-15 4KB small page or 1KB small subpage translations, backwards-compatible format

Using backwards-compatible descriptors, the 4KB small page is generated by setting all of the AP bit pairs to the same values, $AP_3=AP_2=AP_1=AP_0$. If any one of the pairs are different, then the 4KB small page is converted into four 1KB small page subpages. The subpage access permission bits are chosen using the virtual address bits [11:10].

Second-level extended small page table walk

If bits [1:0] of the second-level descriptor are b1XN for ARMv6 format descriptors, or b11 for backwards-compatible descriptors, then an extended small page table walk is required. Figure 6-16 shows the translation process for a 4KB extended small page using ARMv6 format descriptors (AP bits disabled).

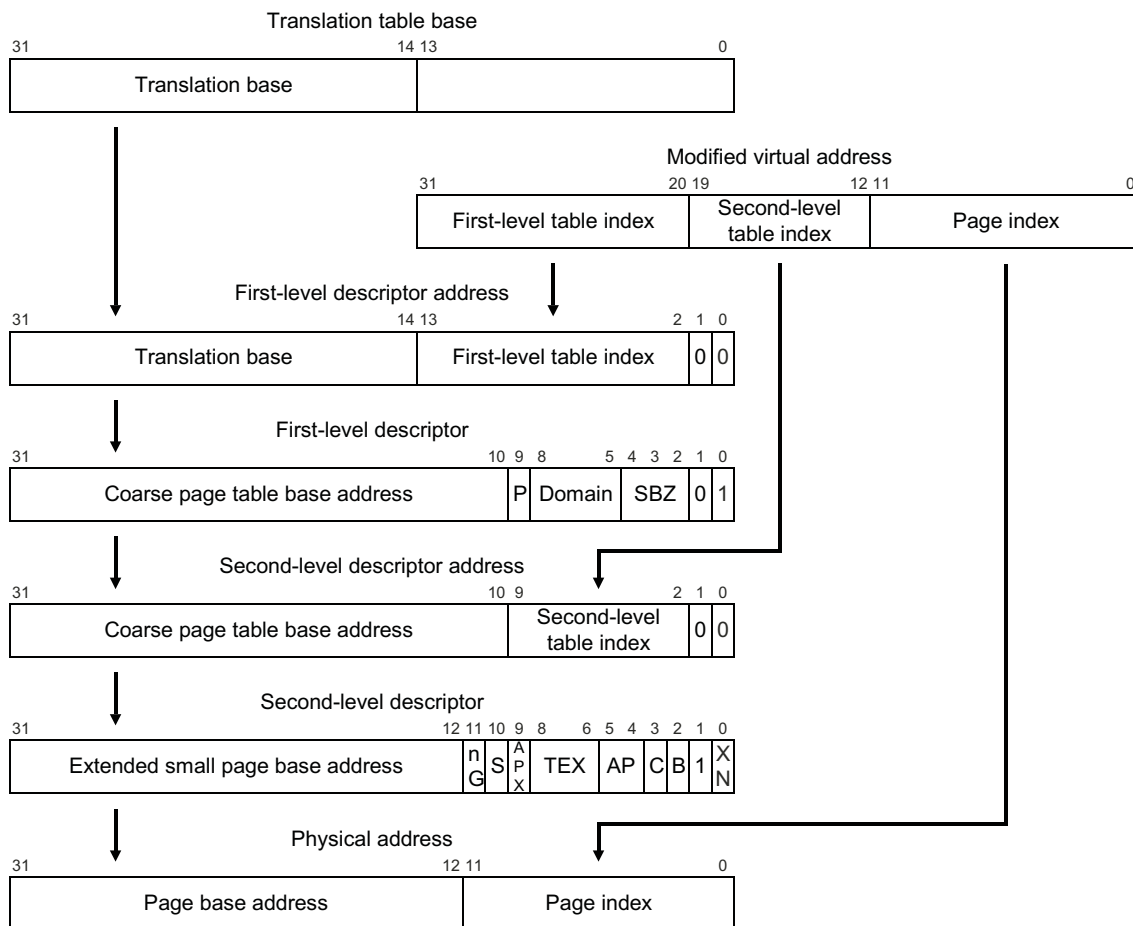


Figure 6-16 4KB extended small page translations, ARMv6 format

Figure 6-17 on page 6-54 shows the translation process for a 4KB extended small page or a 1KB extended small page subpage using backwards-compatible format descriptors (AP bits enabled).

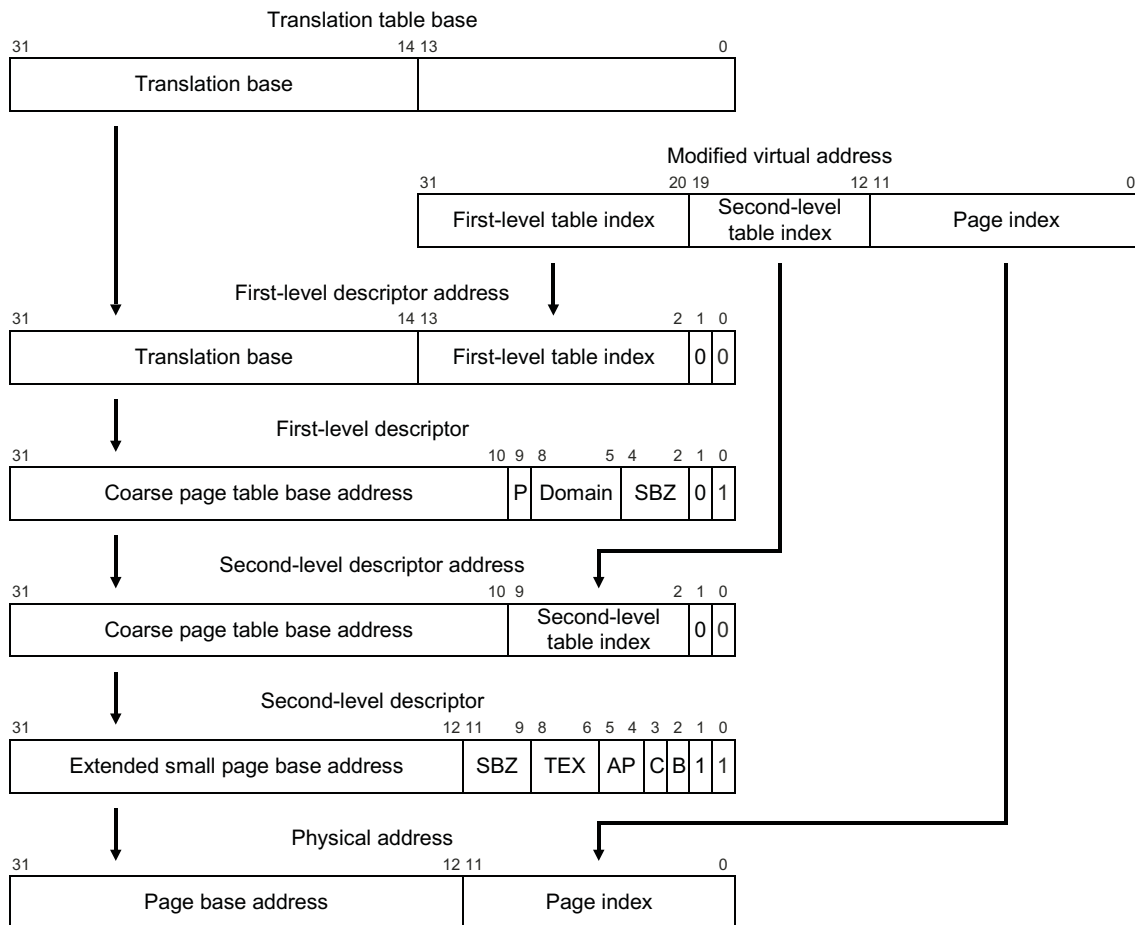


Figure 6-17 4KB extended small page or 1KB extended small subpage translations, backwards-compatible format

Using backwards-compatible descriptors, the 4KB extended small page is generated by setting all of the AP bit pairs to the same values, $AP_3=AP_2=AP_1=AP_0$. If any one of the pairs are different, then the 4KB extended small page is converted into four 1KB extended small page subpages. The subpage access permission bits are chosen using the virtual address bits [11:10].

6.13 MMU software-accessible registers

The MMU is controlled by the system control coprocessor (CP15) registers, shown in Table 6-12, in conjunction with page table descriptors stored in memory.

You can access all the registers with instructions of the form:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

```
MCR p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

Where CRn is the system control coprocessor register. Unless specified otherwise, CRm and Opcode_2 Should Be Zero.

Table 6-12 CP15 register functions

Register	Number	Bits	Description
TLB Type Register	0	[23:16] ILSize, [15:8] DLSIZE, [0] U	The number of the TLB entries for the lockable TLB partitions is specified by the DLSIZE and ILSIZE fields respectively. See <i>TLB Type Register</i> on page 3-74. U bit, unified or separate TLBs: 0 = unified TLB 1 = separate instruction and data TLBs.
Control Register	1	[0] M, [1] A, [3] W, [8] S, [9] R, [23] XP	M bit, MMU enable/disable: 0 = MMU disabled 1 = MMU enabled. A bit, strict data address alignment fault enable/disable: 0 = Strict data address alignment fault checking disabled 1 = Strict data address alignment fault checking enabled. W bit, write buffer enable/disable. If implemented: 0 = write buffer disabled 1 = write buffer enabled. If not implemented, this bit reads as 1, writes ignored. S bit, system protection bit. Only applies when subpage AP bits are enabled (XP = 0). See <i>Control Register</i> on page 3-96. R bit, ROM protection. Only applies when subpage AP bits are enabled (XP = 0). See <i>Control Register</i> on page 3-96. XP bit, extended page table configuration: 0 = subpage AP bits enabled (backwards-compatible format descriptors used) 1 = subpage AP bits disabled, hardware translation tables support additional ARMv6 features (ARMv6 descriptors used).

Table 6-12 CP15 register functions (continued)

Register	Number	Bits	Description
Translation Table Base Register 0	2	[31:14-N] TTBR0	Pointer to the first-level translation table base 0 address for accessing page tables for process-specific addresses. N is the value of the Translation Table Base Control Register 2. It determines the boundary address of the translation table: If N = 0, the page table must reside on a 16KB boundary If N = 1, the page table must reside on a 8KB boundary ... If N = 7, the page table must reside on a 128-byte boundary. See <i>Translation Table Base Register 0</i> on page 3-80.
Translation Table Base Register 1	2	[31:14] TTBR1	Pointer to the first-level translation table base 0 address for accessing page tables for system and I/O addresses. See <i>Translation Table Base Register 1</i> on page 3-81.
Translation Table Base Control Register	2	[2:0] N	Translation table base register control: 0 = use TTBR0. Backwards compatible with ARMv5. 1 = if VA 31 = b0, use TTBR0, otherwise use TTBR1 2 = if VA[31:30] = b00, use TTBR0, otherwise use TTBR1 ... 7 = if VA[31:25] = b0000000, use TTBR0, otherwise use TTBR1. See <i>Translation Table Base Control Register</i> on page 3-79.
Domain Access Control Register	3	[31:0] D15-D0	Comprises 16 2-bit fields. Each field defines the access control attributes for one of 16 domains, D15–D0. See <i>Domain Access Control Register</i> on page 3-67.
Data Fault Status Register (DFSR)	5	[7:4] Domain, [3:0] Status	Indicates the cause of a Data Abort and the domain number of the aborted access, when a Data Abort occurs. Bits [7:4] specify which of the 16 domains (D15–D0) was being accessed when a fault occurred. Bits [3:0] indicate the type of access being attempted. The value of all other bits is Unpredictable. The encoding of these bits is shown in <i>Data Fault Status Register</i> on page 3-66.
Instruction Fault Status Register (IFSR)	5	[3:0] Status	Bits [3:0] indicate the type of access being attempted. The value of all other bits is Unpredictable. The encoding of these bits is shown in <i>Instruction Fault Status Register</i> on page 3-68.

Table 6-12 CP15 register functions (continued)

Register	Number	Bits	Description
Fault Address Register (FAR)	6	[31:0] Data fault address	Holds the modified virtual address associated with the access that caused the data fault. See <i>MMU fault checking</i> on page 6-29 for instructions needed to address the FAR. See <i>Fault Address Register</i> on page 3-65 for details of the address stored for each type of fault.
Instruction Fault Address Register (IFAR)	6	[31:0] Instruction fault address	Holds the modified virtual address associated with the access that caused the instruction fault, or the virtual address of the instruction that caused a debug event. See <i>MMU fault checking</i> on page 6-29 for instructions needed to address the IFAR. See <i>Instruction Fault Address Register</i> on page 3-67 for details of the address stored for each type of fault.
Cache Operations Register	7	[31:5] MVA or [31:30] Index, [S+2:3] Set Where S is \log_2 of the Size Field in the <i>Cache Type Register</i> on page 3-28. [0] selects TCM when set to 1 or cache when set to 0.	A write-only register that you can use to control Instruction Cache, Data Cache, and Write Buffer operations. Also used to control operations on prefetch buffers, and branch target caches, if they are implemented. Instructions to this register are in one of two formats: <ul style="list-style-type: none"> • MVA format • Index/Set format. See <i>Cache Operations Register</i> on page 3-17 for details.
TLB Operations Register	8	[31:10] MVA, [7:0] ASID	Writing to this register causes the MMU to perform TLB maintenance operations. Three functions are provided, selected by the value of the <i>Opcode_2</i> field: b000 = invalidate all the (unpreserved) entries in a TLB b001 = invalidate a specific entry b010 = invalidate entry on ASID match. Reading from this register is Unpredictable. See <i>TLB Operations Register</i> on page 3-75.
TLB Lockdown Register	10	[31:29] SBZ [28:26] Victim [0] P	The Victim field specifies which TLB entry in the lockdown region is replaced by the translation table walk result generated by the next TLB miss. Any translation table walk results written to TLB entries while P = 1 are protected from being invalidated by r8 Invalidate TLB operations. Translation table walk results written to TLB entries while P = 0 are invalidated normally by r8 Invalidate TLB operations.

Table 6-12 CP15 register functions (continued)

Register	Number	Bits	Description
FCSE PID Register	13	[31:25] FCSE PID	This register controls the fast context switch extension. See <i>FCSE PID Register</i> on page 3-100.
ContextID Register	13	[31:8] ProcID, [7:0] ASID	The bottom eight bits of this register contain the ASID of the currently running process. The ProcID bits extend the ASID. See <i>Context ID Register</i> on page 3-95.

Note

All the CP15 MMU registers, except CP15 c7 and CP15 c8, contain state that you read using MRC instructions and written to using MCR instructions. Registers c5 and c6 are also written by the MMU. Reading CP15 c7 and c8 is Unpredictable.

6.14 MMU and Write Buffer

During any translation table walk the MMU has access to external memory. Before the table walk occurs, the write buffer has to be flushed of any related writes to avoid read-after-write hazards.

When either the instruction MMU or data MMU contains valid TLB entries that are being modified, those TLB entries must be invalidated by software, and the write buffer drained using the Drain Write Buffer instruction before the new section or page is accessed.

Chapter 7

Level One Memory System

This chapter describes the ARM1136JF-S level one memory system. It contains the following sections:

- *About the level one memory system* on page 7-2
- *Cache organization* on page 7-3
- *Tightly-coupled memory* on page 7-8
- *DMA* on page 7-11
- *TCM and cache interactions* on page 7-13
- *Cache debug* on page 7-17
- *Write Buffer* on page 7-18.

7.1 About the level one memory system

The ARM1136JF-S level one memory system consists of:

- separate Instruction and Data Caches in a Harvard arrangement
- separate Instruction and Data *Tightly-Coupled Memory* (TCM) areas
- a DMA system for accessing the TCM
- a Write Buffer
- two MicroTLBs, backed by a main TLB.

In parallel with each of the caches is an area of dedicated RAM on both the instruction and data sides. These regions are referred to as TCM. You can implement 0 or 1 TCM on each of the Instruction and Data sides.

Each TCM has a dedicated base address that you can place anywhere in the physical address map, and does not have to be backed by memory implemented externally. The Instruction and Data TCMs have separate base addresses.

Each TCM can optionally support a SmartCache mode of operation. In this mode of operation, the TCM behaves as a large contiguous area of cache, starting at the base address.

Each TCM not configured to operate as SmartCache can be accessed by a DMA mechanism to enable this memory to be loaded from or stored to another location in memory while the processor core is running.

The MMU provides the facilities required by sophisticated operating systems to deliver protected virtual memory environments and demand paging. It also supports real-time tasks with features that provide predictable execution time.

Address translation is handled in a full MMU for each of the instruction and data sides. The MMU is responsible for protection checking, address translation, and memory attributes, some of which can be passed to the level two memory system.

The memory translations are cached in MicroTLBs for each of the instruction and data sides and for the DMA, with a single main TLB backing the MicroTLBs.

7.2 Cache organization

Each cache is implemented as a four-way set associative cache of configurable size. They are virtually indexed and physically addressed. The cache sizes are configurable with sizes in the range of 4 to 64KB. Both the Instruction Cache and the Data Cache are capable of providing two words per cycle for all requesting sources.

Each cache way is architecturally limited to 16KB in size, because of the limitations of the virtually indexed, physically addressed implementation. The number of cache ways is fixed at four, but the cache way size can be varied between 1KB and 16KB in powers of 2. The line length is not configurable and is fixed at eight words per line.

Write operations must occur after the Tag RAM reads and associated address comparisons have completed. A three-entry Write Buffer is included in the cache to enable the written words to be held until there is a gap in cache usage to enable them to be written. One or two words can be written in a single store operation. The addresses of these outstanding writes provide an additional input into the Tag RAM comparison for reads.

To avoid a critical path from the Tag RAM comparison to the enable signals for the data RAMs, there is a minimum of one cycle of latency between the determination of a hit to a particular way, and the start of writing to the data RAM of that way. This requires the Cache Write Buffer to be able to hold three entries, for back-to-back writes. Accesses that read the dirty bits must also check the Cache Write Buffer for pending writes that result in dirty bits being set. The cache dirty bits for the Data Cache are updated when the Cache Write Buffer data is written to the RAM. This requires the dirty bits to be held as a separate storage array (significantly, the tag arrays cannot be written, because the arrays are not accessed during the data RAM writes), but permits the dirty bits to be implemented as a small RAM.

The other main operations performed by the cache are cache line refills and Write-Back. These occur to particular cache ways, which are determined at the point of the detection of the cache miss by the victim selection logic.

To reduce overall power consumption, the number of full cache reads is reduced by the sequential nature of many cache operations, especially on the instruction side. On a cache read that is sequential to the previous cache read, only the data RAM set that was previously read is accessed, if the read is within the same cache line. The Tag RAM is not accessed at all during this sequential operation.

To reduce unnecessary power consumption further, only the addressed words within a cache line are read at any time. With the required 64-bit read interface, this is achieved by disabling half of the RAMs on occasions when only a 32-bit value is required. The implementation uses two 32-bit wide RAMs to implement the cache data RAM shown

in Figure 7-1, with the words of each line folded into the RAMs on an odd and even basis. This means that cache refills can take several cycles, depending on the cache line lengths. The cache line length is eight words.

The control of the level one memory system and the associated functionality, together with other system wide control attributes are handled through the system control coprocessor, CP15. This is described in *Summary of control coprocessor CP15 registers* on page 3-5.

The block diagram of the cache subsystem is as shown in Figure 7-1. This diagram does not show the cache refill paths.

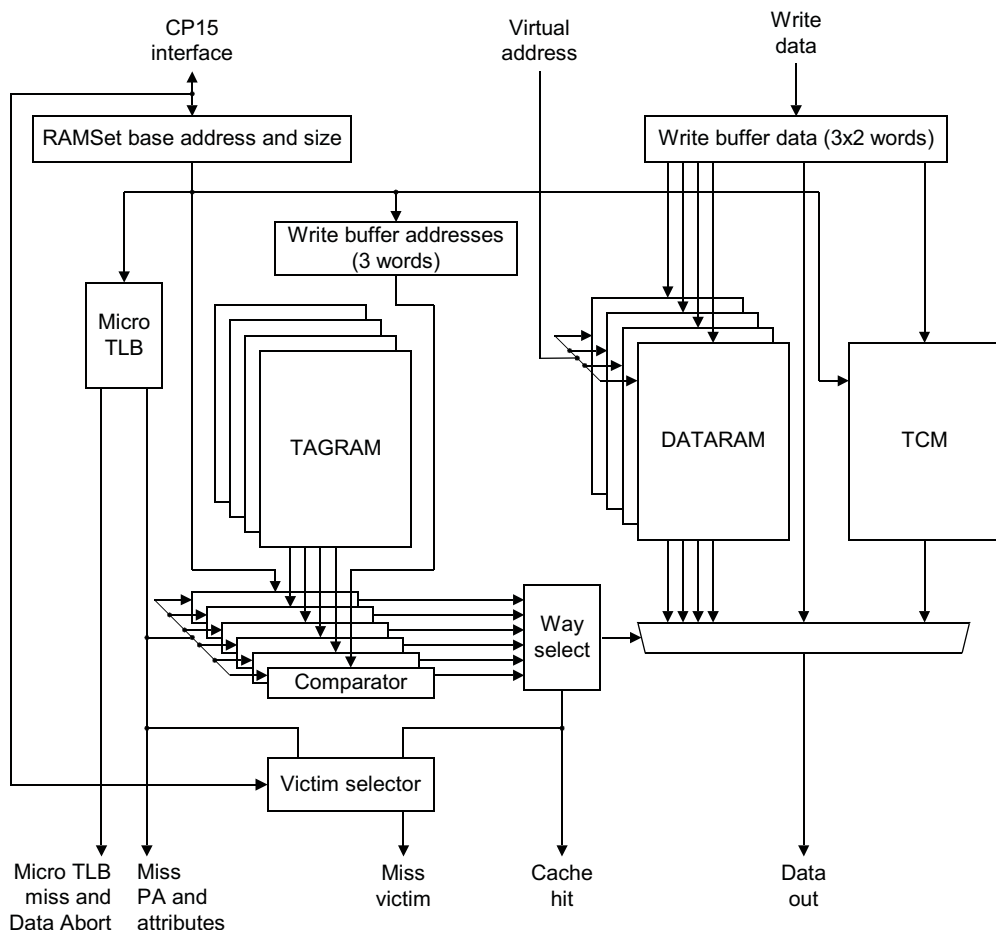


Figure 7-1 Level one cache block diagram

7.2.1 Features of the cache system

The level one cache system has the following features:

- The cache is a Harvard implementation.
- The caches are lockable at a granularity of a cache way, using Format C lockdown. See *Cache Lockdown Registers* on page 3-15.
- Cache replacement policies are Pseudo-Random or Round-Robin, as controlled by the RR bit in CP15 register c1. Round-Robin uses a single counter for all sets, that selects the way used for replacement.
- Cache line allocation uses the cache replacement algorithm when all cache lines are valid. If one or more lines is invalid, then the invalid cache line with the lowest way number is allocated to in preference to replacing a valid cache line. This mechanism does not allocate to locked cache ways unless all cache ways are locked. See *Cache miss handling when all ways are locked down* on page 7-7.
- Cache lines can be either Write-Back or Write-Through, determined by the MicroTLB entry.
- Only read allocation is supported.
- The cache can be disabled independently from the TCM, under control of the appropriate bits in CP15 c1.
- Data cache misses are nonblocking with a single outstanding Data Cache miss being supported.
- Streaming of sequential data from LDM and LDRD operations, and for sequential instruction fetches is supported.

7.2.2 Cache functional description

The cache and TCM exist to perform associative reads and writes on requested addresses. The steps involved in this for reads are as follows:

1. The lower bits of the virtual address are used as the virtual index for the tag and RAM blocks, including the TCM.
2. In parallel the MicroTLB is accessed to perform the virtual to physical address translation.
3. The physical addresses read from the Tag RAMs and the TCM base address register, and the Write Buffer address registers, are compared with the physical address from the MicroTLB to form hit signals for each of the cache ways

4. The hit signals are used to select the data from the cache way that has a hit. Any bytes contained in both the data RAMs and the Write Buffer entries are taken from the Write Buffer. If two or three Write Buffer entries are to the same bytes, the most recently written bytes are taken.

The steps for writes are as follows:

1. The lower bits of the virtual address are used as the virtual index for the tag blocks.
2. In parallel, the MicroTLB is accessed to perform the virtual to physical address translation.
3. The physical addresses read from the Tag RAMs and the TCM base address register are compared with the physical address from the MicroTLB to form hit signals for each of the cache ways.
4. If a cache way, or the TCM, has recorded a hit, then the write data is written to an entry in the Cache Write Buffer, along with the cache way, or TCM, that it must take place to.
5. The contents of the Cache Write Buffer are held until a cycle that is not performing a cache read. At this point the oldest entry in the Cache Write Buffer is written into the cache.

7.2.3 Cache control operations

The cache control operations that are supported by the ARM1136JF-S processor are described in *Cache Operations Register* on page 3-17. ARM1136JF-S processors support all the block cache control operations in hardware.

7.2.4 Cache miss handling

A cache miss results in the requests required to do the line fill being made to the level two interface, with a Write-Back occurring if the line to be replaced contains dirty data.

The Write-Back data is transferred to the Write Buffer, which is arranged to handle this data as a sequential burst. Because of the requirement for nonblocking caches, additional write transactions can occur during the transfer of Write-Back data from the cache to the Write Buffer. These transactions do not interfere with the burst nature of the Write-Back data. The Write Buffer is responsible for handling the potential *Read After Write* (RAW) data hazards that might exist from a Data Cache line Write-Back. The caches perform critical word-first cache refilling. The internal bandwidth from the level two data read port to the Data Caches is eight bytes per cycle, and supports streaming.

Cache miss handling when all ways are locked down

The ARM architecture describes the behavior of the cache as being Unpredictable when all ways in the cache are locked down. However, for ARM1136JF-S processors a cache miss is serviced as if Way 0 is not locked.

7.2.5 Cache disabled behavior

If the cache is disabled, then the cache is not accessed for reads or for writes. This ensures that maximum power savings can be achieved. It is therefore important that before the cache is disabled, all of the entries are cleaned to ensure that the external memory has been updated. In addition, if the cache is enabled with valid entries in it, then it is possible that the entries in the cache contain old data. Therefore the cache must be disabled with clean and invalid entries.

Cache maintenance operations can be performed even if the cache is disabled.

7.2.6 Unexpected hit behavior

An unexpected hit is where the cache reports a hit on a memory location that is marked as Noncachable or Shared. The unexpected hit behavior is that these hits are ignored and a level two access occurs. The unexpected hit is ignored because the cache hit signal is qualified by the cachability.

For writes, an unexpected cache hit does not result in the cache being updated. Therefore, writes appear to be Noncachable accesses. For a data access, if it lies in the range of memory specified by the Instruction TCM configured as Local RAM, then the access is made to that RAM rather than to level two memory. This applies to both writes and reads.

7.3 Tightly-coupled memory

The TCM is designed to provide low-latency memory that can be used by the processor without the unpredictability that is a feature of caches.

You can use such memory to hold critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache is highly undesirable. In addition you can use it to hold scratch pad data, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks.

You can configure the TCM in several ways:

- one TCM on the instruction side and one on the data side
- one TCM on the instruction or data side only
- no TCM on either side.

The TCM Status Register in CP15 c0 describes what TCM options and TCM sizes can be implemented, see *TCM Status Register* on page 3-83.

Each TCM can optionally support a SmartCache mode of operation, see *SmartCache behavior* on page 7-9. In this mode the RAM behaves as a large contiguous area of cache, starting at the base address. As a result, the corresponding memory locations must also exist in the external memory system.

When a TCM is configured as a SmartCache it has the same:

- behavior as cache
- unexpected hit behavior as cache, see *Unexpected hit behavior* on page 7-7.

If a TCM is not configured to operate as SmartCache, then it behaves as Local RAM, see *Local RAM behavior* on page 7-9. Each Data TCM is implemented in parallel with the Data Cache and the Instruction TCM is implemented in parallel with the Instruction Cache. Each TCM has a single movable base address, specified in CP15 register c9, (see *Data TCM Region Register* on page 3-83 and *Instruction TCM Region Register* on page 3-85).

The size of each TCM can be different to the size of a cache way, but forms a single contiguous area of memory. The entire level one memory system is shown in Figure 7-1 on page 7-4.

You can disable each TCM to avoid an access being made to it. This gives a reduction in the power consumption. You can disable each TCM independently from the enabling of the associated cache, as determined by CP15 register c9.

The disabling of a TCM invalidates the base address, so there is no unexpected hit behavior for the TCM when configured as Local RAM.

7.3.1 SmartCache behavior

Instruction and Data TCMs support SmartCache in this implementation.

When a TCM is configured as SmartCache it forms a contiguous area of cache, with the contents of memory backed by external memory. Each line of the TCM, which is of the same length as the cache line (indicated in the Cache Type Register for the equivalent cache), can be individually set as being Valid or Invalid. Writing the RAM Region Register causes the valid information for each line to be cleared (marked as Invalid). When a read access is made to an Invalid line, the line is fetched from the level two memory system in exactly the same way as for a cache miss, and the fetched line is then marked as Valid.

For the TCM to exhibit SmartCache behavior, areas of memory that are covered by a TCM operating as SmartCache must be marked as Cachable. For a memory access to a memory location that is marked as Noncachable but is in an area covered by a TCM, if the corresponding SmartCache line is marked as Invalid, then the memory access does not cause the location to be fetched from external memory and marked as Valid. If the corresponding SmartCache line is marked as Valid, then the access is made to external memory.

If a TCM region configured as SmartCache covers an area of memory that is Shared, then the SmartCache is not loaded on a miss.

7.3.2 Local RAM behavior

When a TCM is configured as Local RAM it forms a continuous area of memory that is always valid if the TCM is enabled. Therefore it does not use the Valid bits for each line that is used for SmartCache. The TCM configured as Local RAM is used as part of the physical memory map of the system, and is not backed by a level of external memory with the same physical addresses. For this reason, the TCM behaves differently from the caches for regions of memory that are marked as being Write-Through Cachable. In such regions, no external writes occur in the event of a write to memory locations contained in the TCM.

The DMA only operates to an area of TCM that is configured as Local RAM, to prevent any requirement of interactions between the cache refill and DMA operations. Attempting to perform a DMA to an area of TCM that is configured as SmartCache result in an internal DMA error (TCM DMA out of range).

7.3.3 Restriction on page table mappings

The TCMs are implemented in a physically indexed, physically addressed manner, giving the following behavior:

- the entries in the TCM do not have to be cleaned and/or invalidated by software for different virtual to physical mappings
- aliases to the same physical address can exist in memory regions that are held in the TCM.

As a result, the page mapping restrictions for the TCM are less restrictive than for the cache, as described in *Restrictions on accesses to different types of memory* on page 6-26.

7.3.4 Restriction on page table attributes

The page table entries that describe areas of memory that are handled by the TCM can be described as being Cachable or Noncachable, but must not be marked as Shared. If they are marked as either Device or Strongly Ordered, or have the Shared attribute set then the locations that are contained within the TCM are treated as being Non-Shared, Noncachable.

7.4 DMA

The level one DMA provides a background route to transfer blocks of data to or from the TCMs. Its used to move large blocks, rather than individual words or small structures.

The level one DMA is initiated and controlled by accessing the appropriate CP15 registers and instructions, see *DMA registers* on page 3-51. The process specifies the internal start and end addresses and external start address, together with the direction of the DMA. The addresses specified are Virtual Addresses, and the level one DMA hardware includes translation of Virtual Addresses to Physical Addresses and checking of protection attributes.

The TLB, described in *TLB organization* on page 6-4 is used to hold the page table entries for the DMA, and ensures that the entries in a TLB used by the DMA are consistent with the page tables. Errors, arising from protection checks, are signaled to the processor using an interrupt.

Completion of the DMA can also be configured by software to signal the processor with an interrupt using the same interrupt to the processor that the error uses.

The status of the DMA is read from the CP15 registers associated with the DMA.

The DMA controller is programmed using the CP15 coprocessor. DMA accesses can only be to or from the TCM, configured as Local RAM, and must not be from areas of memory that can be contained in the caches. That is, no coherency support is provided in the caches.

The ARM1136JF-S processor implements two DMA channels. Only one channel can be active at a time. The key features of the DMA system are:

- the DMA system runs in the background of processor operations
- DMA progress is accessible from software
- DMA is programmed with virtual addresses, with a MicroTLB dedicated to the DMA function
- you can configure the DMA to work to either the instruction or data RAMs
- DMA is allocated by a privileged process, enabling User access to control the DMA.

For some DMA events an interrupt is generated. If this happens the nDMAIRQ signal of the ARM1136JF-S processor is asserted. You can route this output pin to an external interrupt controller for prioritization and masking. This is the only mechanism by which the interrupt is signaled to the core.

Each DMA channel has its own set of Control and Status Registers. The maximum number of DMA channels that can be defined is architecturally limited to 2. Only 1 DMA channel can be active at a time. If the other DMA channel has been started, it is queued to start performing memory operations after the currently active channel has completed.

The level one DMA behaves as a distinct master from the rest of the processor, and the same mechanisms for handling Shared memory regions must be used if the external addresses being accessed by the level one DMA system are also accessed by the rest of the processor. These are described in *Memory attributes and types* on page 6-17. If a User mode DMA transfer is performed using an external address that is not marked as Shared, an error is signaled by the DMA channel.

There is no ordering requirement of memory accesses caused by the level one DMA relative to those generated by reads and writes by the processor, while a channel is running. When a channel has completed running, all its transactions are visible to all other observers in the system. All memory accesses caused by the DMA occur in the order specified by the DMA channel, regardless of the memory type.

If a DMA is performed to Strongly Ordered memory (see *Memory attributes and types* on page 6-17), then a transaction caused by the DMA prevents any further transactions being generated by the DMA until the point at which the access is complete. A transaction is complete when it has changed the state of the target location or data has been returned to the DMA.

If the FCSE PID, the Domain Access Control Register, or the page table mappings are changed, or the TLB is flushed, while a DMA channel is in the Running or Queued state, then it is Unpredictable when the effect of these changes is seen by the DMA.

7.5 TCM and cache interactions

In the event that a TCM and a cache both contain the requested address, it is architecturally Unpredictable which memory the instruction data is returned from. It is expected that such an event only arises from a failure to invalidate the cache when the base register of the TCM is changed, and so is clearly a programming error.

For a Harvard arrangement of caches and TCM, data reads and writes can access any Instruction TCM configured as local memory for both reads and writes. This ensures that accesses to literal pools, Undefined instructions, and SWI numbers are possible, and aids debugging. For this reason, an Instruction TCM configured as local memory must behave as a unified TCM, but can be optimized for instruction fetches. This requirement only exists for the TCMs when configured as Local RAM.

You must not program an Instruction TCM to the same base address as a Data TCM and, if the two RAMs are different sizes, the regions in physical memory of the two RAMs must not be overlapped unless each TCM is configured to operate as SmartCache. This is because the resulting behavior is architecturally Unpredictable.

If a Data and an Instruction TCM overlap, and either is not configured as SmartCache, it is Unpredictable which memory the instruction data is returned from.

In these cases, you must not rely on the behavior of ARM1136JF-S processor that is intended to be ported to other ARM platforms.

7.5.1 DMA and core access arbitration

DMA and core accesses to both the Instruction TCM and the Data TCM can occur in parallel. So as not to disrupt the execution of the core, core-generated accesses have priority over those requested by the DMA engine.

7.5.2 Instruction accesses to TCM

If the Instruction TCM and the Instruction Cache both contain the requested instruction address, the ARM1136JF-S processor returns data from the TCM. The instruction prefetch port of the ARM1136JF-S processor cannot access the Data TCM. If an instruction prefetch misses the Instruction TCM and Instruction Cache but hits the Data TCM, then the result is an access to the level two memory.

An IMB must be inserted between a write to an Instruction TCM and the instructions being written being relied upon. In addition, any branch prediction mechanism must be invalidated or disabled if a branch in the Instruction TCM is overwritten.

7.5.3 Data and instruction accesses to TCM

If the Data TCM and the Data Cache both contain the requested data address for a read, the ARM1136JF-S processor returns data from the Data TCM. For a write, the write occurs to the Data TCM. The majority of data accesses are expected to go to the Data Cache or to the Data TCM, but it is necessary for the Instruction TCM to be read or written on occasion.

The Instruction TCM base addresses are read by the ARM1136JF-S processor data port as a possible source for data for all memory accesses. This increases the data comparisons associated with the data, compared with the number required for the instruction memory lookup, for the level one memory hit generation. This functionality is required for reading literal values and for debug purposes, such as setting software breakpoints.

SWP and other memory synchronization operations, such as load-exclusive and stored-exclusive, to instruction TCM are not supported, and result in Unpredictable behavior. Access to the Instruction TCM involves a delay of at least two cycles in the reading or writing of the data. This delay enables the Instruction TCM access to be scheduled to take place only when the presence of a hit to the Instruction TCM is known. This saves power and avoids unnecessary delays being inserted into the instruction-fetch side. This delay is applied to all accesses in a multiple operation in the case of an LDM, an LDCL, an STM, or an STCL.

It is not required for instruction port(s) to be able to access the Data TCM. An attempt to access addresses in the range covered by a Data TCM from an instruction port does not result in an access to the Data TCM. In this case, the instruction is fetched from main memory. It is anticipated that such accesses can result in external aborts in some systems, because the address range might not be supported in main memory.

Table 7-1 on page 7-15 summarizes the results of data accesses to TCM and the cache. This also embodies the unexpected hit behavior for the cache described in *Unexpected hit behavior* on page 7-7. In Table 7-1 on page 7-15, if the Data Cache or Data TCM are operating as SmartCache, they can only be hit if the memory location being accessed is marked as being Cachable and Not Sharable.

The hit to the Data TCM and Instruction TCM refers to hitting an address in the range covered by that TCM.

Table 7-1 Summary of data accesses to TCM and caches

Data TCM	Data cache	Instruction TCM (Local RAM)	Read behavior	Write behavior
Hit (local RAM)	Hit	Hit	Read from Data TCM.	Write to Data TCM. No write to the Instruction TCM. No write to level two, even if marked as Write-Through.
Hit (SmartCache)	Hit	Hit	Read from Data TCM.	Write to Data TCM if line valid. No write to Instruction TCM. If Write-Through, write to level two.
Hit (Local RAM)	Hit	Miss	Read from Data TCM.	Write to Data TCM. No write to level two even if marked as Write-Through.
Hit (SmartCache)	Hit	Miss	Read from Data TCM.	Write to Data TCM if line valid. If Write-Through write to level two.
Hit (Local RAM)	Miss	Hit	Read from Data TCM. No linefill to Data Cache fill even if marked Cachable.	Write to Data TCM. No write to Instruction TCM. No write to level two even if marked as Write-Through.
Hit (SmartCache)	Miss	Hit	Read from Data TCM if line valid. Linefill to SmartCache if line invalid. No linefill to Data Cache even if location is marked as Cachable.	Write to Data TCM if line valid. No write to Instruction TCM if Write-Back. If Write-Through or Data TCM invalid, write to Instruction TCM.
Hit (Local RAM)	Miss	Miss	Read from Data TCM. No linefill to Data Cache even if marked Cachable.	Write to Data TCM. No write to level two even if marked as Write-Through.
Hit (SmartCache)	Miss	Miss	Read from Data TCM. Linefill to SmartCache if line invalid. No linefill to Data Cache even if location is marked as Cachable.	Write to Data TCM if line valid. If Write-Through, or Data TCM line invalid, write to level two.
Miss	Hit	Hit	If Cachable, read from Data Cache. If Noncachable, read from Instruction TCM.	Write to Data Cache. If Write-Through, write to Instruction TCM.

Table 7-1 Summary of data accesses to TCM and caches (continued)

Data TCM	Data cache	Instruction TCM (Local RAM)	Read behavior	Write behavior
Miss	Hit	Miss	If Cachable, read from Data Cache. If Noncachable, read from level two.	Write to Data Cache. If Write-Through, write to level two.
Miss	Miss	Hit	Read from Instruction TCM. No cache fill even if marked Cachable.	Write to Instruction TCM. No write to level two even if marked as Write-Through.
Miss	Miss	Miss	If Cachable and cache enabled, cache linefill. If Noncachable or cache disabled, read to level two.	Write to level two.

Table 7-2 summarizes the results of instruction accesses to TCM and the cache. This also embodies the unexpected hit behavior for the cache described in *Unexpected hit behavior* on page 7-7. In Table 7-2, the Instruction Cache, and the Instruction TCM if operating as SmartCache, can only be hit if the memory location being accessed is marked as being Cachable and not shareable. The hit to the Instruction TCM refers to hitting an address in the range covered by that TCM.

Table 7-2 Summary of instruction accesses to TCM and caches

Instruction TCM	Instruction cache	Data TCM	Read behavior
Hit	Hit	Don't care	Unpredictable.
Hit (Local RAM)	Miss	Don't care	Read from Instruction TCM. No linefill to Instruction Cache even if marked Cachable.
Hit (SmartCache)	Miss	Don't care	Read from Instruction TCM if line valid. Linefill to SmartCache if line invalid. No linefill to Instruction Cache even if marked Cachable.
Miss	Hit	Don't care	Read from Instruction Cache.
Miss	Miss	Don't care	If Cachable and cache enabled, cache linefill. If Noncachable or cache disabled, read to level two.

7.6 Cache debug

The debug architecture for the ARM1136JF-S processor is described in Chapter 13 *Debug*. The External Debug Interface is based on JTAG, and is as described in Chapter 14 *Debug Test Access Port*. The debug architecture enables the cache debug to be defined by the implementation. This functionality is defined here.

It is desirable for the debugger to examine the contents of the instruction and Data Caches during debug operations. This is achieved in two stages:

1. Reading the Tag RAM entries for each cache location.
2. Reading the data values for those addresses.

The debugger determines which valid addresses are stored in the cache. This is done by reading the Instruction and Data Cache Tag arrays using a CP15 instruction executed using the Instruction Transfer Register. The Instruction Transfer Register is accessed using scan chain 4 as described in *Scan chain 4, instruction transfer register (ITR)* on page 14-13. The debugger must do this for each entry of each set within the cache. This access is performed by an MCR that transfers from the ARM register the Set and Index of the required line in the Tag RAM array. The contents of the line are then returned to the Instruction or Data Debug Cache Register as appropriate.

7.7 Write Buffer

All memory writes take place using the Write Buffer. To ensure that the Write Buffer is not drained on reads, the following features are implemented:

- The Write Buffer is a FIFO of outstanding writes to memory. It consists of a set of addresses and a set of data words (together with their size information).
- If a sequence of data words is contained in the Write Buffer, these are denoted as applying to the same address by the Write Buffer storing the size of the store multiple. This reduces the number of address entries that need to be stored in the Write Buffer.
- In addition to this, a separate FIFO of Write-Back addresses and data words is implemented. Having a separate structure avoids complications associated with performing an external write while the write-through is being handled.
- The address of a new read access is compared against the addresses in the Write Buffer. If a read is to a location that is already in the Write Buffer, the read is blocked until the Write Buffer has drained sufficiently far for that location to be no longer in the Write Buffer. The sequential marker only applies to words in the same 8 word (8 word aligned) block, and the address comparisons are based on 8 word aligned addresses.

The ordering of memory accesses is described in *Memory access control* on page 6-11.

Chapter 8

Level Two Interface

The ARM1136JF-S processor is designed to be used within larger chip designs using *Advanced Microcontroller Bus Architecture (AMBA)*. The ARM1136JF-S processor uses the level two interface as its interface to memory and peripherals.

This chapter describes the features of the level two interface not covered in the *AMBA Specification*. The chapter contains the following sections:

- *About the level two interface* on page 8-2
- *Synchronization primitives* on page 8-7
- *AHB-Lite control signals in the ARM1136JF-S processor* on page 8-9
- *Instruction Fetch Interface AHB-Lite transfers* on page 8-20
- *Data Read Interface AHB-Lite transfers* on page 8-24
- *Data Write Interface AHB-Lite transfers* on page 8-49
- *DMA Interface AHB-Lite transfers* on page 8-64
- *Peripheral Interface AHB-Lite transfers* on page 8-66
- *AHB-Lite* on page 8-69.

8.1 About the level two interface

The level two memory interface exists to provide a high-bandwidth interface to second level caches, on-chip RAM, peripherals, and interfaces to external memory.

It is a key feature in ensuring high system performance, providing a higher bandwidth mechanism for filling the caches in a cache miss than has existed on previous ARM processors.

The ARM1136JF-S processor level two interconnect system uses the following 64-bit wide AHB-Lite interfaces:

- Instruction Fetch Interface
- Data Read Interface
- Data Write Interface
- DMA Interface.

Another interface is also provided. The Peripheral Interface is a 32-bit AHB-Lite interface.

The level two interconnect interfaces are shown in Figure 8-1.

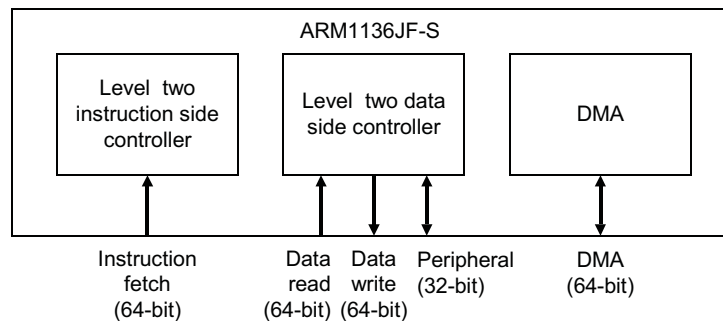


Figure 8-1 Level two interconnect interfaces

These interfaces provide for several simultaneous outstanding transactions, giving the potential for high performance from level two memory systems that support parallelism, and also for high utilization of pipelined memories such as SDRAM.

Each of the four wide interfaces is an AHB-lite interface, with additional signals to support additional features for the level two memory system:

- shared memory synchronization primitives
- multi-level cache support
- unaligned and mixed-endian data access.

8.1.1 Level two interface clocking

In addition to the ARM1136JF-S clock **CLKIN**, the level two interfaces are clocked using:

- **HCLKIRW** for the instruction read, data read, and data write ports
- **HCLKPD** for the peripheral and DMA ports.

The two clocks used by each port can be either synchronous or asynchronous. Input pins on the ARM1136JF-S processor control selection between synchronous and asynchronous clocking, and ensure that the latency penalty for any synchronization is only applied when it is required.

Figure 8-2 compares the performance lost through synchronization penalty with the performance lost through reducing the core frequency to be an integer multiple of the bus frequency.

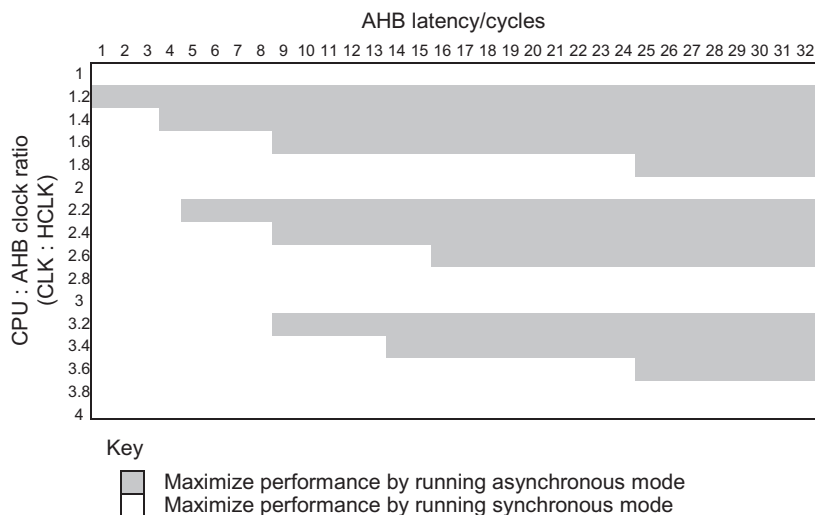


Figure 8-2 Synchronization penalty

You can independently configure **HCLKIRW** and **HCLKPD** to be synchronous or asynchronous. See Chapter 9 *Clocking and Resets* for more details.

8.1.2 Level two instruction-side controller

The level two instruction-side controller contains the level two Instruction Fetch Interface. See *Instruction Fetch Interface* on page 8-4.

The level two instruction-side controller handles all instruction-side cache misses including those for Noncachable locations. It is responsible for the sequencing of cache operations for Instruction Cache linefills, making requests for the individual stores through the *Prefetch Unit* (PU) to the Instruction Cache. The decoupling involved means that the level two instruction-side controller contains some buffering.

Instruction Fetch Interface

The Instruction Fetch Interface is a read-only interface that services the Instruction Cache on cache misses, including the fetching of instructions for the PU that are held in memory marked as Noncachable. The interface is optimized for cache linefills rather than individual requests.

8.1.3 Level two data-side controller

The level two data-side controller is responsible for the level two:

- Data Read Interface
- Data Write Interface
- Peripheral Interface.

The level two data-side controller handles:

- All external access requests from the Load Store Unit, including cache misses, data Write-Through operations, and Noncachable data.
- SWP instructions and semaphore operations. It schedules all reads and writes on the two interfaces, which are closely related.

The level two data-side controller also handles the Peripheral Interface.

The level two data-side controller contains the Refill and Write-Back engines for the Data Cache. These make requests through the Load Store Unit for the individual cache operations that are required. The decoupling involved means that the level two data-side controller contains some buffering. The write buffer is an integral part of the level two data-side controller.

A separate block within the level two data-side controller also schedules the reads required for hardware page table walks, and returns the appropriate page table information to the main TLB.

Data Read Interface

The Data Read Interface performs reads and swap writes. It services the Data Cache on cache misses, handles TLB misses on hardware page table walks, and reads uncachable locations. While cache miss handling is important, the latency between outstanding uncachable loads is minimized. The same address never appears on the Data Read Interface and the Data Write Interface simultaneously.

Data Write Interface

The Data Write Interface is a write-only interface that services the writes out of the Write Buffer. Multiple writes can be queued up as part of this interface.

Peripheral Interface

The Peripheral Interface is a bidirectional AHB-Lite interface that services peripheral devices. The bus is a single master bus with the Peripheral Interface being the master. In ARM1136JF-S processors, the Peripheral Interface is used for peripherals that are private to the ARM1136JF-S processor, such as the Vectored Interrupt Controller or Watchdog Timer. Accesses to regions of memory that are marked as Device and Non-Shared are routed to the Peripheral Interface in preference to the Data Read Interface or Data Write Interface.

Peripheral Port Memory Remap Register

The Peripheral Port Memory Remap Register enables regions to be remapped to the Peripheral Interface when the MMU is disabled. For details of the Peripheral Port Memory Remap Register see *Remapping the peripheral port when the MMU is disabled* on page 3-72.

8.1.4 DMA

The DMA is responsible for:

- Performing all external memory transactions required by the DMA engine, and for requesting accesses from the Instruction TCM and Data TCM as required.
- Queuing the two DMA channels as required. The DMA Interface contains several registers that are CP15 registers dedicated for DMA use, see *DMA control* on page 3-51 for details.

The DMA contains:

- Its own MicroTLB that is backed up by the main TLB. The DMA uses the PU and the *Load Store Unit* (LSU) to schedule its accesses to the TCMs.

- Buffering to enable the decoupling of internal and external requests. This is because of variable latency between internal and external accesses.

DMA Interface

The DMA Interface is a bidirectional interface that services the DMA subsystem for writing and reading the TCMs. Although the DMA Interface is bidirectional, it is able to produce a stream of successive accesses that are in the same direction, followed by either a further stream in the same direction, or a stream in the opposite direction. Correspondingly the direction turnaround is not significantly optimized.

8.2 Synchronization primitives

On previous architectures support for shared memory synchronization has been with the read-locked-write operations that swap register contents with memory, the SWP and SWPB instructions. These support basic busy and free semaphore mechanisms. For details of the swap instructions, and how to use them to implement semaphores, see the *ARM Architecture Reference Manual*.

ARMv6 describes support for more comprehensive shared-memory synchronization primitives that scale for multiple-processor system designs. Two instructions are introduced that support multiple-processor and shared-memory inter-process communication:

- load-exclusive, LDREX
- store-exclusive, STREX.

The exclusive-access instructions rely on the ability to tag a physical address as exclusive-access for a particular processor. This tag is later used to determine if an exclusive store to an address occurs. For memory regions that:

- Have the Shared TLB attribute, any attempt to modify that address by any processor clears this tag.
- Do not have the Shared TLB attribute, any attempt to modify that address by the same processor that marked it as exclusive-access clears this tag. In both cases other events might cause the tag to be cleared. In particular, for memory regions that are not shared, it is Unpredictable whether a store by another processor to a tagged physical address causes the tag to be cleared.

An external abort on either a load-exclusive or store-exclusive puts the processor into Abort mode.

Note

An external abort on a load-exclusive can leave the ARM1136JF-S internal monitor in its exclusive state and might affect your software. If it does you must ensure that a store-exclusive to an unused location is executed in your abort handler to clear the ARM1136JF-S internal monitor to an open state.

8.2.1 Load-exclusive instruction

Load-exclusive performs a load from memory and causes the physical address of the access to be tagged as exclusive-access for the requesting processor. This causes any other physical address that has been tagged by the requesting processor to no longer be tagged as exclusive-access.

8.2.2 Store-exclusive instruction

Store-exclusive performs a conditional store to memory. The store only takes place if the physical address is tagged as exclusive-access for the requesting processor. This operation returns a status value. If the store updates memory the return value is 0, otherwise it is 1. In both cases, the physical address is no longer tagged as exclusive-access for any processor.

8.2.3 Example of LDREX and STREX usage

Given below is an example of typical usage. Suppose you are trying to claim a lock:

```

Lock address : LockAddr
Lock free   : 0x00
Lock taken  : 0xFF
    MOV R1, #0xFF ; load the 'lock taken' value
try LDREXR0, [LockAddr]; load the lock value
    CMP R0, #0 ; is the lock free?
    STREXEQR1, R0, [LockAddr]; try and claim the lock
    CMPEQR0, #0 ; did this succeed?
    BNE try ; no - try again. . . .
; yes - we have the lock

```

The typical case, where the lock is free and you have exclusive-access, is six instructions.

8.3 AHB-Lite control signals in the ARM1136JF-S processor

This section describes the ARM1136JF-S processor implementation of the AHB-Lite control signals:

- *HTRANS[1:0]*
- *HSIZE[2:0]* on page 8-10
- *HBURST[2:0]* on page 8-10
- *HPROT[4:0]* on page 8-11
- *HPROT[5]* and *HRESP[2]* on page 8-13
- *HBSTRB[7:0]* and *HUNALIGN* on page 8-15.

For additional information about AHB, see the *AMBA Specification Rev 2.0*.

8.3.1 Signal name suffixes

The signal name for each of the interfaces denotes the interface that it applies to. The following suffixes are used:

I	Instruction Fetch Interface.
D	DMA Interface.
R	Data Read Interface.
W	Data Write Interface.
P	Peripheral Interface.

For example, **HTRANS[1:0]** is called **HTRANSI[1:0]** in the Instruction Fetch Interface.

8.3.2 HTRANS[1:0]

Table 8-1 shows the settings used to indicate the type of transfer on the interface.

Table 8-1 HTRANS[1:0] settings

HTRANS[1:0] settings	type of transfer
b00	Idle
b10	Nonsequential
b11	Sequential
b01	Busy is not used

8.3.3 HSIZE[2:0]

The ARM1136JF-S processor has 64-bit buses. **HSIZE** cannot be greater than 64 bits. The encodings of **HSIZE[2:0]** are shown in Table 8-2.

Table 8-2 HSIZE[2:0] encoding

HSIZE[2]	HSIZE[1]	HSIZE[0]	Size	Description
0	0	0	8 bits	Byte
0	0	1	16 bits	Halfword
0	1	0	32 bits	Word
0	1	1	64 bits	Doubleword

8.3.4 HBURST[2:0]

Table 8-3 shows the settings used to indicate the type of transfer on the interface.

Table 8-3 HBURST[2:0] settings

HBURST[2:0] settings	type of transfer
b000	Single
b001	Incr
b010	Wrap4
b011	Incr4

8.3.5 HPROT[4:0]

The values of the **HPROT[1:0]** bits that can be used in level two caches are shown in Table 8-4.

Table 8-4 HPROT[1:0] encoding

Value	Meaning
HPROT[0]	0 = Instruction cache linefill or core instruction fetch. 1 = Data cache linefill, core Load or Store operation, or page table walks.
HPROT[1]	0 User mode. 1 Privileged mode.

To support the addition of on-chip second level caching the ARMv6 AHB-Lite extensions include an additional **HPROT[4]** bit that is used to extend the definition of the **HPROT[3:2]** bits. The additional bit provides information about the caching policy that is used for the transfer that is being performed.

Table 8-5 shows the how various combinations of **HPROT[4:2]** signals are encoded.

Table 8-5 HPROT[4:2] encoding

HPROT[4] Allocate	HPROT[3] Cachable	HPROT[2] Bufferable	Description
0	0	0	Strongly Ordered, cannot be buffered
0	0	1	Device, can be buffered
0	1	0	Cachable (Outer Noncachable, do not allocate on reads or writes)
1	1	0	Cachable Write-Through (allocate on reads only, No Allocate on Write)
0	1	1	Cachable Write-Back (allocate on reads and writes)
1	1	1	Cachable Write-Back (allocate on reads only, No Allocate on Write)

The timing of **HPROT[4]** is identical to the timing of the other **HPROT** signals, so it is an address phase signal and must remain constant throughout a burst.

The Allocate bit, **HPROT[4]**, is used to provide additional information on the allocation scheme that must be used for the transfer. When the transfer is Noncachable (**HPROT[3]** is LOW) then the Allocate bit is not used and must also be driven LOW by a master.

For transfers that are indicated as Cachable (**HPROT[3]** is HIGH) the combination of the Allocate bit and the Bufferable bit are used to indicate which of the following cases is required:

Allocate = 0, Bufferable = 0

Indicates that the transfer can be treated as Cachable, but it is recommended that this transfer is not cached. This scheme can typically be used for an address region that is memory (as opposed to peripheral space) but which does not benefit from being cached in a level two cache. This might be because:

- The memory region is going to be cached in a level one cache.
- The contents of the memory region have characteristics that mean there is no benefit in caching the region. For example, it is data that is used only once.

Marking that a region that must not be cached as Cachable enables improvements in overall system performance. Certain system components, such as bus bridges, can improve performance when accessing cachable regions by executing speculative accesses.

Allocate = 1, Bufferable = 0

Indicates that a region must be treated as Write-Through. A read transfer must cause the memory region to be loaded in to the cache. If a write occurs to an address that is already cached, then the cache must be updated and a write must occur to update the original memory location at the same time. This strategy enables a cache line to be later evicted from the cache without the requirement to first update any memory regions that have changed.

Allocate = 0, Bufferable = 1

Indicates a Write-Back Cachable region. A read transfer causes the memory region to be loaded in to the cache. If a write occurs to an address that is already cached then the cache must be updated. If the address is not already cached then it must be loaded in to the cache. The write to update the original memory location must not occur until the cache line is evicted from the cache.

Allocate = 1, Bufferable = 1

Indicates a Write-Back Cachable region, but with No Allocate on Write. In this instance if a write occurs to an address that is not already in the cache, then that address must not be loaded in to the cache. Instead, the write to the original address location must occur.

8.3.6 HPROT[5] and HRESP[2]

Two additional signals are provided in the ARMv6 AHB-Lite extensions to support an exclusive access mechanism. The exclusive access mechanism is used to provide additional functionality over and above that provided by the lock mechanism on AHB v2.0. The exclusive access mechanism enables the implementation of semaphore type operations without requiring the entire bus to remain locked to a particular master for the duration of the operation.

The advantage of this approach is that semaphore type operations do not impact on the critical bus access latency and they do not impact on the maximum achievable bandwidth.

The additional signals are:

HPROT[5] Exclusive bit, indicates that an access is part of an exclusive operation.

HRESP[2] Exclusive response, which indicates if the write part of an exclusive operation has succeeded or failed:

- **HRESP[2] = 0** indicates that the exclusive operation has succeeded
- **HRESP[2] = 1** indicates that the exclusive operation has failed.

The exclusive access mechanism operates as follows:

1. A master performs an exclusive read from an address location.
2. At some later point in time the master attempts to complete the exclusive access by performing an exclusive write to the same address location.
3. The write access of the master is signaled as successful if no other master has updated (written to) the location between the read and write accesses.

If, however, another master has updated the location between the read and write accesses then the exclusive access is signaled as having failed and the address location is not updated.

The following points apply to the exclusive access mechanism:

- If a master attempts an exclusive write without first performing an exclusive read then the write is signaled as failing.
- A master can attempt an exclusive read to new location without first completing the read or write sequence to a another location that has previously been exclusively read from. In this instance the second exclusive sequence must continue as described in step 1. to step 3. above.

If the write portion of the earlier sequence does eventually occur then it is acceptable that the access is indicated as successful only if the sequence has truly succeeded. That is, the location has not been updated since the exclusive read from that master. Alternatively, the access can be automatically signaled as failing.

It is important that repeated occurrences of incomplete exclusive accesses, where only the read portion of the access happens, does not cause a lock up situation.

Exclusive access protocol

The protocols for exclusive accesses are summarized as follows:

- All AHB-Lite control signals must remain constant throughout a burst, this includes **HPROT[5]**. This means that a burst of accesses must not include an exclusive access as one item within the burst.
- A response on **HRESP[2]** indicating failure of the exclusive write access is a two-cycle response, as is the case for any other non-Okay value on **HRESP**.
- The mnemonic for a response indicating the failure of an exclusive write is Xfail. Table 8-6 shows the valid responses on **HRESP[2:0]** with associated mnemonics. All other values of **HRESP[2:0]** are reserved.

Table 8-6 HRESP[2:0] mnemonics

HRESP[2:0]	Mnemonic
b000	Okay
b001	Error
b010	Retry
b011	Split
b100	Xfail

- It is not possible to indicate a combination of either Error, Retry, or Split with Xfail. The values b101, b110, and b111 are not valid responses. The Xfail response indicates that an exclusive write has not been transmitted to the destination because the exclusive access monitor knows that another domain has already over-written it. Therefore, because the access is not attempted, there can be no associated Error, Retry, or Split information.

- The master cannot cancel the next access for an Xfail response if it is already indicating one on AHB-Lite. This is unlike:
 - Split or Retry responses for which the master must cancel the next access
 - Error responses for which the master might cancel the next access.

If a master does have to wait for the response to the exclusive write before issuing the next access, then it is recommended that it issues either:

- Idle cycles
 - deassert request line, **HBUSREQ**
 - Idle cycles and deassert request line, **HBUSREQ**.
- An Error response to an exclusive read indicates that the data read back cannot be trusted. That is, the read is invalid and must be tried again after the reason for the error has been resolved.
 - An Error response to an exclusive write indicates that the data has not been written, but does not necessarily mean that another process has written to that memory location, or that the data is not the most recent data. The exclusive write can be tried again at a later time, after the reason for the error has been resolved, and the success or failure of the exclusive write is determined by whether or not an Xfail response is eventually received.

8.3.7 HBSTRB[7:0] and HUNALIGN

To handle unaligned accesses and mixed-endian accesses the AHB-Lite extensions enable the use of byte lane strobes to indicate which byte lanes are active in a transfer. One **HBSTRB** signal is required for each byte lane on the data bus. That is, one **HBSTRB** bit for every eight bits of the data bus.

The **HBSTRB** signal is asserted in the same cycles as the other address and control signal of the transfer that it applies to. In other words it is an address phase signal.

HADDR and **HSIZE** are used to define the container within which the byte lane strobes can be active. The size of the transaction is sufficient to cover all the bytes being written and covers more bytes in the case of a mis-aligned transfer. **HADDR** is aligned to the size of transfer, as indicated by **HSIZE**, so that the address of the transfer is rounded down to the nearest boundary of the size of the transaction.

Byte strobes are required for both read and write transfers. Read sensitive devices must not be accessed using unaligned transfers so a master can choose, for a read transfer, to activate all byte strobes within the AHB v2.0 container (as defined by **HADDR** and **HSIZE**).

For forwards compatibility, if an AHB v2.0 master does not generate byte strobe signals then these can be generated directly from the **HADDR** and **HSIZE** signals. This generation process must take into account the endianness of the transfer.

For backwards compatibility, an additional **HUNALIGN** signal is provided by a master that can produce unaligned accesses. This signal is only provided to assist with backwards compatibility and indicates when a single unaligned transfer occurs that requires more than one AHB v2.0 transfer (without byte strobes). The **HUNALIGN** signal has address phase timing and must be asserted HIGH for unaligned transfers and LOW for AHB v2.0 compatible aligned transfers.

The mapping of byte strobes to data bus bits is fixed and is not dependent on the endianness of the access. The mapping of **HBSTRB** to the write data bus for a 64-bit interface is shown in Table 8-7.

Table 8-7 Mapping of HBSTRB to HWDATA bits for a 64-bit interface

Byte strobe	⇒	Data bus bits
HBSTRB[0]	⇒	HWDATA[7:0]
HBSTRB[1]	⇒	HWDATA[15:8]
HBSTRB[2]	⇒	HWDATA[23:16]
HBSTRB[3]	⇒	HWDATA[31:24]
HBSTRB[4]	⇒	HWDATA[39:32]
HBSTRB[5]	⇒	HWDATA[47:40]
HBSTRB[6]	⇒	HWDATA[55:48]
HBSTRB[7]	⇒	HWDATA[63:56]

———— **Note** ————

Not all possible combinations of byte lane strobes are generated by the ARM1136JF-S processor. The slaves that support these extensions must enable all possible combinations to provide compatibility with future AMBA components (for example, masters containing merging write buffers).

Example uses of byte lane strobes

This section gives some example ARMv6 transfers on AHB-Lite and shows the byte strobe signals that are produced. Table 8-8 on page 8-17 shows examples of transfers that can be produced by an ARMv6 architecture processor.

The examples assume a 64-bit data bus.

————— **Note** —————

When an access straddles a 32-bit data boundary then two transfers are required.

Table 8-8 Byte lane strobes for example ARMv6 transfers

Transfer description	HADDR	HSIZE[2:0]	HBSTRB[7:0]	HUNALIGN
8-bit access to 0x1000	0x1000	0x0	b00000001	0
8-bit access to 0x1003	0x1003	0x0	b00001000	0
8-bit access to 0x1007	0x1007	0x0	b10000000	0
16-bit access to 0x1000	0x1000	0x1	b00000011	0
16-bit access to 0x1005	0x1004	0x2	b01100000	1
16-bit access to 0x1007	0x1007	0x0	b10000000	0
	0x1008	0x0	b00000001	0
32-bit access to 0x1000	0x1000	0x2	b00001111	0
32-bit access to 0x1002	0x1002	0x1	b00001100	0
	0x1004	0x1	b00110000	0
32-bit access to 0x1003	0x1003	0x0	b00001000	0
	0x1004	0x2	b01110000	1
32-bit access to 0x1007	0x1007	0x0	b10000000	0
	0x1008	0x2	b00000111	1
64-bit access to 0x1000	0x1000	0x3	b11111111	0

8.3.8 Exclusive access timing

Figure 8-3 on page 8-18 shows the basic operation of an exclusive read, followed at some arbitrary time later by an exclusive write. The exclusive write receives an Okay response indicating that the operation has been successful.

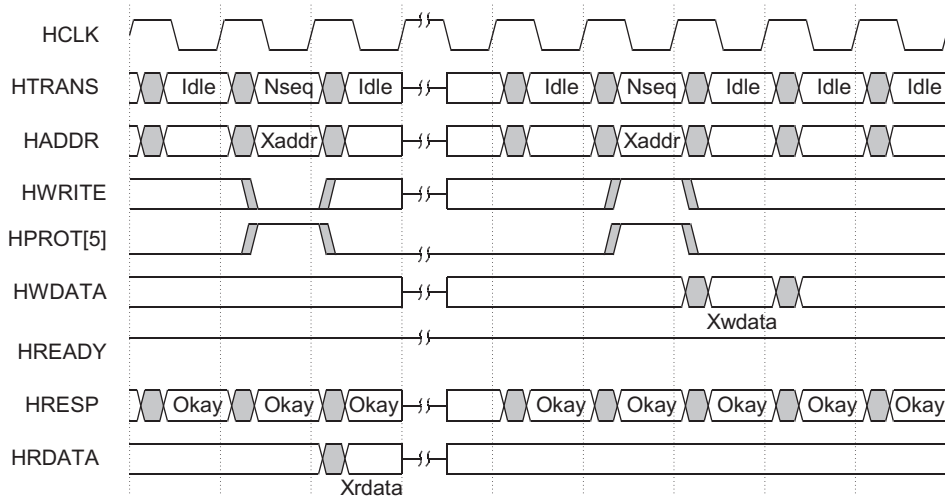


Figure 8-3 Exclusive access read and write with Okay response

Figure 8-4 shows an exclusive access that receives an Xfail response. Although **HWDATA** is shown asserted for the write access, the target location must not be updated within the slave.

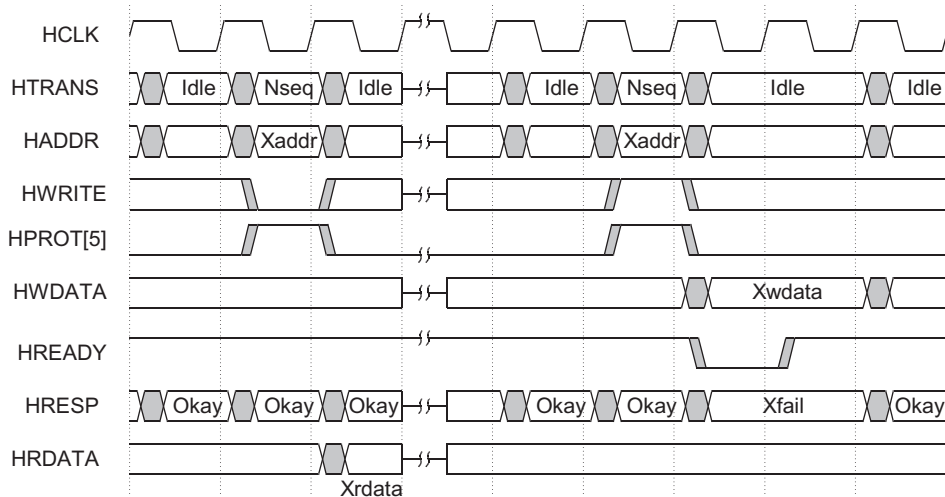


Figure 8-4 Exclusive access read and write with Xfail response

Figure 8-5 shows an exclusive access that receives an Xfail response, but this time the master has already placed the next transfer (a read from address Naddr) onto the AHB-Lite address and control pins. If the two-cycle response were a Split or Retry, then the master has to force **HTRANS** to Idle after time T_{17} , or has the option to do so if the response is Error. For the Xfail response, the master must continue with the transfer indicated after T_{16} .

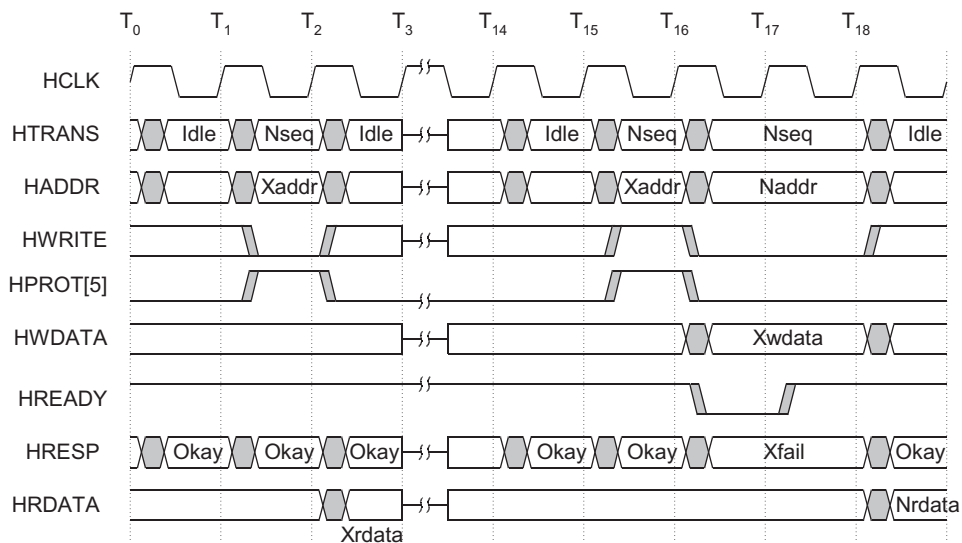


Figure 8-5 Exclusive access read and write with Xfail response and following transfer

8.4 Instruction Fetch Interface AHB-Lite transfers

The tables in this section describe the AHB-Lite interface behavior for instruction side fetches to either Cachable or Noncachable regions of memory for the following interface signals:

- **HBURSTI[2:0]**
- **HTRANSI[1:0]**
- **HADDRI[31:0]**
- **HBSTRBI[7:0]**
- **HUNALIGNI**.

See *Other AHB-Lite signals for Cachable and Noncachable instruction fetches* on page 8-22 for details of the other AHB-Lite signals.

8.4.1 Cachable fetches

The values of **HTRANSI**, **HADDRI**, **HBURSTI**, **HBSTRBI**, and **HUNALIGNI** for Cachable fetches from words 0-7 are shown in Table 8-9.

Table 8-9 AHB-Lite signals for Cachable fetches

Address[4:0]	HTRANSI	HADDRI	HBURSTI	HBSTRBI	HUNALIGNI
0x00 (word 0)	Nseq	0x00	Incr4	b11111111	0
0x04 (word 1)	Seq	0x08			
		0x10			
		0x18			
0x08 (word 2)	Nseq	0x08	Wrap4	b11111111	0
0x0C (word 3)	Seq	0x10			
		0x18			
		0x00			
0x10 (word 4)	Nseq	0x10	Wrap4	b11111111	0
0x14 (word 5)	Seq	0x18			
		0x00			
		0x08			

Table 8-9 AHB-Lite signals for Cachable fetches (continued)

Address[4:0]	HTRANSI	HADDRI	HBURSTI	HBSTRBI	HUNALIGNI
0x18 (word 6)	Nseq	0x18	Wrap4	b11111111	0
0x1C (word 7)	Seq	0x00			
		0x08			
		0x10			

8.4.2 Noncachable fetches

The values of **HTRANSI**, **HADDRI**, **HBURSTI**, **HBSTRBI**, and **HUNALIGNI** for Noncachable fetches from words 0-7 are shown in Table 8-10.

Table 8-10 AHB-Lite signals for Noncachable fetches

Address[4:0]	HTRANSI	HADDRI	HBURSTI	HBSTRBI	HUNALIGNI
0x00 (word 0)	Nseq	0x00	Incr4	b11111111	0
	Seq	0x08			
		0x10			
		0x18			
0x04 (word 1)	Nseq	0x00	Incr4	b11110000	1
	Seq	0x08		b11111111	0
		0x10			
		0x18			
0x08 (word 2)	Nseq	0x08	Incr	b11111111	0
	Seq	0x10			
		0x18			
0x0C (word 3)	Nseq	0x08	Incr	b11110000	1
	Seq	0x10		b11111111	0
		0x18			

Table 8-10 AHB-Lite signals for Noncachable fetches (continued)

Address[4:0]	HTRANSI	HADDRI	HBURSTI	HBSTRBI	HUNALIGNI
0x10 (word 4)	Nseq	0x10	Incr	b11111111	0
	Seq	0x18			
0x14 (word 5)	Seq	0x10	Incr	b11110000	1
		0x18		b11111111	0
0x18 (word 6)	Nseq	0x18	Single	b11111111	0
0x1C (word 7)	Nseq	0x18	Single	b11110000	1

8.4.3 Other AHB-Lite signals for Cachable and Noncachable instruction fetches

The other AHB-Lite signals used in the Instruction Fetch Interface are:

HWRITEI	Static 0, indicating a read.
HSIZEI[2:0]	Static b011, indicating a size of 64 bits.
HPROTI[5]	Static 0, indicating a non-exclusive transfer.
HPROTI[4:2]	These bits encode the memory region attributes, as shown in Table 8-11.

Table 8-11 HPROTI[4:2] encoding

HPROTI[4:2]	Memory region attribute
b000	Strongly Ordered
b001	Device
b010	Outer Noncachable
b110	Outer Write-Through, No Allocate on Write
b111	Outer Write-Back, No Allocate on Write
b011	Outer Write-Back, Write Allocate

HPROTI[1] Encodes the CPSR state, as shown in Table 8-12.

Table 8-12 HPROTI[1] encoding

HPROTI[1]	CPSR state
0	User mode access
1	Privileged mode access

HPROTI[0] Statically 0, indicating an Opcode Fetch.

HSIDEBANDI[3:1] Encodes the Inner Cachable TLB attributes, as shown in Table 8-13.

Table 8-13 HSIDEBANDI[3:1] encoding

HSIDEBAND[3:1]	Attribute
b000	Strongly ordered
b001	Device
b010	Inner Noncachable
b110	Inner Cachable

HSIDEBANDI[0] The TLB Sharable bit.

HMASTLOCKI Static 0, indicating an unlocked transfer.

8.5 Data Read Interface AHB-Lite transfers

The tables in this section describe the AHB-Lite interface behavior for Data Read Interface transfers for the following interface signals:

- **HBURSTR[2:0]**
- **HTRANSR[1:0]**
- **HADDRR[31:0]**
- **HBSTRBR[7:0]**
- **HSIZER[2:0]**.

8.5.1 Linefills

A linefill comprises four accesses to the Data Cache if there is no external abort returned. In the event of an external abort, the doubleword and subsequent doublewords are not written into the Data Cache and the line is never marked as Valid. The four accesses are:

- Write Tag and data doubleword
- Write data doubleword
- Write data doubleword
- Write Valid = 1, Dirty = 0, and data doubleword.

The linefill can only progress to attempt to write a doubleword if it does not contain dirty data. This is determined in one of two ways:

- if the victim cache line is not valid, then there is no danger and the linefill progresses
- if the victim line is valid a signal encodes which doublewords are clean (either because they were not dirty or they have been cleaned).

The order of words written into the cache is critical-word first, wrapping at the upper cache line boundary.

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for linefills are shown in Table 8-14.

Table 8-14 Linefills

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x00-0x07	Nseq	0x00	Incr4	64-bit	b11111111
	Seq	0x08			
		0x10			
		0x18			
0x08-0x0F	Nseq	0x08	Wrap4	64-bit	b11111111
	Seq	0x10			
		0x18			
		0x00			
0x10-0x17	Nseq	0x10	Wrap4	64-bit	b11111111
	Seq	0x18			
		0x00			
		0x08			
0x18-0x1F	Nseq	0x18	Wrap4	64-bit	b11111111
	Seq	0x00			
		0x08			
		0x10			

8.5.2 Noncachable LDRB

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDRBs from bytes 0-7 are shown in Table 8-15.

Table 8-15 Noncachable LDRB

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x00 (byte 0)	Nseq	0x00	Single	8-bit	b00000001
0x01 (byte 1)	Nseq	0x01	Single	8-bit	b00000010
0x02 (byte 2)	Nseq	0x02	Single	8-bit	b00000100
0x03 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
0x04 (byte 4)	Nseq	0x04	Single	8-bit	b00010000
0x05 (byte 5)	Nseq	0x05	Single	8-bit	b00100000
0x06 (byte 6)	Nseq	0x06	Single	8-bit	b01000000
0x07 (byte 7)	Nseq	0x07	Single	8-bit	b10000000

8.5.3 Noncachable LDRH

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDRHs from bytes 0-7 are shown in Table 8-16.

Table 8-16 Noncachable LDRH

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x0 (byte 0)	Nseq	0x00	Single	16-bit	b00000011
0x1 (byte 1)	Nseq	0x00	Single	32-bit	b00000110 ^a
0x2 (byte 2)	Nseq	0x02	Single	16-bit	b00001100
0x3 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
		0x04			b00010000
0x4 (byte 4)	Nseq	0x04	Single	16-bit	b00110000
0x5 (byte 5)	Nseq	0x04	Single	32-bit	b01100000 ^a

Table 8-16 Noncacheable LDRH (continued)

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x6 (byte 6)	Nseq	0x06	Single	32-bit	b11000000
0x7 (byte 7)	Nseq	0x07	Single	8-bit	b10000000
		0x08			

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

8.5.4 Noncacheable LDR or LDM1

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDR or LDM1s are shown in Table 8-17.

Table 8-17 Noncacheable LDR or LDM1

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x00 (byte 0) (word 0)	Nseq	0x00	Single	32-bit	b00001111
0x01 (byte 1)	Nseq	0x00	Single	32-bit	b00001110 ^a
		0x04		8-bit	b00010000
0x02 (byte 2)	Nseq	0x02	Single	16-bit	b00001100
		0x04			b00110000
0x03 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
		0x04		32-bit	b01110000 ^a
0x04 (byte 4) (word 1)	Nseq	0x04	Single	32-bit	b11110000
		0x05 (byte 5)		0x04	Single
0x06 (byte 6)	Nseq		0x08	Single	8-bit
		0x06	Single		16-bit
0x07 (byte 7)	Nseq	0x08	Single	8-bit	b00000011
		0x07		Single	8-bit
0x08 (word 2)	Nseq	0x08	Single	32-bit	b00000111 ^a
				32-bit	b00001111

Table 8-17 Noncacheable LDR or LDM1 (continued)

Address[4:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x0C (word 3)	Nseq	0x0C	Single	32-bit	b11110000
0x10 (word 4)	Nseq	0x10	Single	32-bit	b00001111
0x14 (word 5)	Nseq	0x14	Single	32-bit	b11110000
0x18 (word 6)	Nseq	0x18	Single	32-bit	b00001111
0x1C (word 7)	Nseq	0x1C	Single	32-bit	b11110000

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

8.5.5 Noncacheable LDM2

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM2s are shown in Table 8-18 to Table 8-25 on page 8-29.

Table 8-18 Noncacheable LDM2 from word 0

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Single	64-bit	b11111111

Table 8-19 Noncacheable LDM2 from word 1

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr	32-bit	b11110000
Seq	0x08			b00001111

Table 8-20 Noncacheable LDM2 from word 2

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Single	64-bit	b11111111

Table 8-21 Noncachable LDM2 from word 3

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x0C	Incr	32-bit	b11110000
Seq	0x10			b00001111

Table 8-22 Noncachable LDM2 from word 4

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x10	Single	64-bit	b11111111

Table 8-23 Noncachable LDM2 from word 5

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x14	Incr	32-bit	b11110000
Seq	0x18			b00001111

Table 8-24 Noncachable LDM2 from word 6

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x18	Single	64-bit	b11111111

Table 8-25 Noncachable LDM2 from word 7

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x1C	Single	32-bit	b11110000

Plus an LDR from 0x00 (byte 0).

8.5.6 Noncacheable LDM3

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM3s are shown in Table 8-26 to Table 8-38 on page 8-33.

Table 8-26 Noncacheable LDM3 from word 0, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	32-bit	b00001111
Seq	0x04			b11110000
	0x08			b00001111

Table 8-27 Noncacheable LDM3 from word 0, Noncacheable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11111111 ^a
Seq	0x08			b00001111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-28 Noncacheable LDM3 from word 1, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr	32-bit	b11110000
Seq	0x08			b00001111
	0x0C			b11110000

**Table 8-29 Noncachable LDM3 from word 1,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11110000 ^a
Seq	0x08			b11111111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-30 Noncachable LDM3 from word 2,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	32-bit	b00001111
Seq	0x0C			b11110000
	0x10			b00001111

**Table 8-31 Noncachable LDM3 from word 2,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11111111 ^a
Seq	0x10			b00001111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-32 Noncachable LDM3 from word 3,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x0C	Incr	32-bit	b11110000
Seq	0x10			b00001111
	0x14			b11110000

**Table 8-33 Noncacheable LDM3 from word 3,
Noncacheable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11110000 ^a
Seq	0x10			b11111111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-34 Noncacheable LDM3 from word 4,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x10	Incr	32-bit	b00001111
Seq	0x14			b11110000
	0x18			b00001111

**Table 8-35 Noncacheable LDM3 from word 4,
Noncacheable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x10	Incr	64-bit	b11111111 ^a
Seq	0x18			b00001111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-36 Noncacheable LDM3 from word 5,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x14	Incr	32-bit	b11110000
Seq	0x18			b00001111
	0x1C			b11110000

**Table 8-37 Noncachable LDM3 from word 5,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x10	Incr	64-bit	b11110000 ^a
Seq	0x18			b11111111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-38 Noncachable LDM3 from word 6 or 7,
Noncachable memory or cache disabled**

Address[4:0]	Operations
0x18	LDM2 from 0x18 + LDR from 0x00
0x1C	LDR from 0x1C + LDM2 from 0x00

8.5.7 Noncachable LDM4

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM4s are shown in Table 8-39 to Table 8-46 on page 8-35.

Table 8-39 Noncachable LDM4 from word 0

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11111111
Seq	0x08			

**Table 8-40 Noncachable LDM4 from word 1,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr4	32-bit	b11110000
Seq	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111

**Table 8-41 Noncachable LDM4 from word 1,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11110000 ^a
Seq	0x08			b11111111 ^a
	0x10			b00001111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-42 Noncachable LDM4 from word 2

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11111111
Seq	0x10			

**Table 8-43 Noncachable LDM4 from word 3,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x0C	Incr4	32-bit	b11110000
Seq	0x10			b00001111
	0x14			b11110000
	0x18			b00001111

**Table 8-44 Noncachable LDM4 from word 3,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11110000 ^a
Seq	0x10			b11111111 ^a
	0x18			b00001111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-45 Noncacheable LDM4 from word 4

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x10	Incr	64-bit	b11111111
Seq	0x18			

Table 8-46 Noncacheable LDM4 from word 5, 6, or 7

Address[4:0]	Operations
0x14 (word 5)	LDM3 from 0x14 + LDR from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM2 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM3 from 0x00

8.5.8 Noncacheable LDM5

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM5s are shown in Table 8-47 to Table 8-55 on page 8-38.

Table 8-47 Noncacheable LDM5 from word 0, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	32-bit	b00001111
Seq	0x04			b11110000
	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111

Table 8-48 Noncacheable LDM5 from word 0, Noncacheable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11111111 ^a
Seq	0x08			b11111111 ^a
	0x10			b00001111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-49 Noncachable LDM5 from word 1, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr	32-bit	b11110000
Seq	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111
	0x14			b11110000

Table 8-50 Noncachable LDM5 from word 1, Noncachable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11110000 ^a
Seq	0x08			b11111111 ^a
	0x10			

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-51 Noncachable LDM5 from word 2, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	32-bit	b00001111
Seq	0x0C			b11110000
	0x10			b00001111
	0x14			b11110000
	0x18			b00001111

**Table 8-52 Noncachable LDM5 from word 2,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11111111 ^a
Seq	0x10			b11111111 ^a
	0x18			b00001111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

**Table 8-53 Noncachable LDM5 from word 3,
Strongly Ordered or Device memory**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x0C	Incr	32-bit	b11110000
Seq	0x10			b00001111
	0x14			b11110000
	0x18			b00001111
	0x1C			b11110000

**Table 8-54 Noncachable LDM5 from word 3,
Noncachable memory or cache disabled**

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11110000 ^a
Seq	0x10			b11111111 ^a
	0x18			

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-55 Noncacheable LDM5 from word 4, 5, 6, or 7

Address[4:0]	Operations
0x10 (word 4)	LDM4 from 0x10 + LDR from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM2 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM3 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM4 from 0x00

8.5.9 Noncacheable LDM6

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM6s are shown in Table 8-56 to Table 8-60 on page 8-39.

Table 8-56 Noncacheable LDM6 from word 0

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	64-bit	b11111111
Seq	0x08			
	0x10			

Table 8-57 Noncacheable LDM6 from word 1, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr	32-bit	b11110000
Seq	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111
	0x14			b11110000
	0x18			b00001111

Table 8-58 Noncachable LDM6 from word 1, Noncachable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr4	64-bit	b11110000 ^a
Seq	0x08			b11111111 ^a
	0x10			b11111111 ^a
	0x18			b00001111 ^a

a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-59 Noncachable LDM6 from word 2

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x08	Incr	64-bit	b11111111
Seq	0x10			
	0x18			

Table 8-60 Noncachable LDM6 from word 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x0C (word 3)	LDM5 from 0x0C + LDR from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM2 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM3 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM4 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM5 from 0x00

8.5.10 Noncacheable LDM7

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM7s are shown in Table 8-61 to Table 8-65 on page 8-41.

Table 8-61 Noncacheable LDM7 from word 0, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr	32-bit	b00001111
Seq	0x04			b11110000
	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111
	0x14			b11110000
	0x18			b00001111

Table 8-62 Noncacheable LDM7 from word 0, Noncacheable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr4	64-bit	b11111111 ^a
Seq	0x08			
	0x10			
	0x18			b00001111 ^a

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-63 Noncachable LDM7 from word 1, Strongly Ordered or Device memory

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x04	Incr	32-bit	b11110000
Seq	0x08			b00001111
	0x0C			b11110000
	0x10			b00001111
	0x14			b11110000
	0x18			b00001111
	0x1C			b11110000

Table 8-64 Noncachable LDM7 from word 1, Noncachable memory or cache disabled

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr4	64-bit	b11110000 ^a
Seq	0x08			b11111111
	0x10			
	0x18			

- a. Denotes that **HUNALIGNR** is asserted for that transfer. This is only for ARMv6 unaligned loads and loads to normal memory, where reading more data than is necessary is possible.

Table 8-65 Noncachable LDM7 from word 2, 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x08 (word 2)	LDM6 from 0x08 + LDR from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM2 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM3 from 0x00

Table 8-65 Noncachable LDM7 from word 2, 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x14 (word 5)	LDM3 from 0x14 + LDM4 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM5 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM6 from 0x00

8.5.11 Noncachable LDM8

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM8s are shown in Table 8-66 and Table 8-67.

Table 8-66 Noncachable LDM8 from word 0

HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
Nseq	0x00	Incr4	64-bit	b11111111
Seq	0x08			
	0x10			
	0x18			

Table 8-67 Noncachable LDM8 from word 1, 2, 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x04 (word 1)	LDM7 from 0x04 + LDR from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM2 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM3 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM4 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM5 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM6 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM7 from 0x00

8.5.12 Noncachable LDM9

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM9s are shown in Table 8-68.

Table 8-68 Noncachable LDM9

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDR from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM2 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM3 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM4 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM5 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM6 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM7 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00

8.5.13 Noncachable LDM10

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM10s are shown in Table 8-69.

Table 8-69 Noncachable LDM10

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM2 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM3 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM4 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM5 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM6 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM7 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDR from 0x00

8.5.14 Noncachable LDM11

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM11s are shown in Table 8-70.

Table 8-70 Noncachable LDM11

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM3 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM4 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM5 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM6 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM7 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDR from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM2 from 0x00

8.5.15 Noncachable LDM12

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM12s are shown in Table 8-71.

Table 8-71 Noncachable LDM12

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM4 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM5 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM6 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM7 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM8 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00 + LDR from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDM2 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM3 from 0x00

8.5.16 Noncachable LDM13

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM13s are shown in Table 8-72.

Table 8-72 Noncachable LDM13

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM5 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM6 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM7 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM8 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM8 from 0x00 + LDR from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00 + LDM2 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDM3 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM4 from 0x00

8.5.17 Noncachable LDM14

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncachable LDM14s are shown in Table 8-73.

Table 8-73 Noncachable LDM14

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM6 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM7 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM8 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM8 from 0x00 + LDR from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM8 from 0x00 + LDM2 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00 + LDM3 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDM4 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM5 from 0x00

8.5.18 Noncacheable LDM15

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM15s are shown in Table 8-74.

Table 8-74 Noncacheable LDM15

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM7 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM8 from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM8 from 0x00 + LDR from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM8 from 0x00 + LDM2 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM8 from 0x00 + LDM3 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00 + LDM4 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDM5 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM6 from 0x00

8.5.19 Noncacheable LDM16

The values of **HTRANSR**, **HADDRR**, **HBURSTR**, **HSIZER**, and **HBSTRBR** for Noncacheable LDM16s are shown in Table 8-75.

Table 8-75 Noncacheable LDM16

Address[4:0]	Operations
0x00 (word 0)	LDM8 from 0x00 + LDM8 from 0x00
0x04 (word 1)	LDM7 from 0x04 + LDM8 from 0x00 + LDR from 0x00
0x08 (word 2)	LDM6 from 0x08 + LDM8 from 0x00 + LDM2 from 0x00
0x0C (word 3)	LDM5 from 0x0C + LDM8 from 0x00 + LDM3 from 0x00
0x10 (word 4)	LDM4 from 0x10 + LDM8 from 0x00 + LDM4 from 0x00
0x14 (word 5)	LDM3 from 0x14 + LDM8 from 0x00 + LDM5 from 0x00
0x18 (word 6)	LDM2 from 0x18 + LDM8 from 0x00 + LDM6 from 0x00
0x1C (word 7)	LDR from 0x1C + LDM8 from 0x00 + LDM7 from 0x00

8.5.20 SWP instructions

Cachable and Noncachable SWP instructions over the Data Read Interface are shown in Table 8-76 and Table 8-77 respectively.

Table 8-76 Cachable swap

Swap operation	AHB-Lite operations
Swap read	LDR or LDRB from the Data Read Interface
Swap write	STR or STRB from the Data Write Interface

Table 8-77 Noncachable swap

Swap operation	AHB-Lite operations
Swap read	LDR or LDRB from the Data Read Interface
Swap write	STR or STRB from the Data Read Interface (HWRITER = 1)

8.5.21 Page table walks

Page table walks over the Data Read Interface are shown in Table 8-78.

Table 8-78 Page table walks

Address[2:0]	HTRANSR	HADDRR	HBURSTR	HSIZER	HBSTRBR
0x0	Nseq	0x0	Single	32-bit	b00001111
0x4	Nseq	0x4	Single	32-bit	b11110000

8.5.22 Other AHB-Lite signals for Data Read ports

The other AHB-Lite signals for Data Read ports are:

HSIDEBAND[3:1] Encodes the Inner Cachable TLB attributes, as shown in Table 8-79.

Table 8-79 HSIDEBAND[3:1] encoding

HSIDEBAND[3:1]	Attribute
b000	Strongly ordered
b001	Device
b010	Inner Noncachable
b110	Inner Write-Through
bx11	Inner Write-Back

HSIDEBAND[0] The TLB Sharable bit.

8.6 Data Write Interface AHB-Lite transfers

The tables in this section describe the AHB-Lite interface behavior for Data Write Interface transfers for the following interface signals:

- **HBURSTW[2:0]**
- **HTRANSW[1:0]**
- **HADDRW[31:0]**
- **HBSTRBW[7:0]**
- **HSIZEW[2:0]**.

8.6.1 Stores on the AHB-Lite interface

The values of **HTRANSW**, **HADDRW**, **HBURSTW**, **HSIZEW**, and **HBSTRBW** for stores over the Data Write Interface are shown in Table 8-80 to Table 8-104 on page 8-60.

Table 8-80 Cachable or Noncachable Write-Through STRB

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (byte 0)	Nseq	0x00	Single	8-bit	b00000001
0x01 (byte 1)	Nseq	0x01	Single	8-bit	b00000010
0x02 (byte 2)	Nseq	0x02	Single	8-bit	b00000100
0x03 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
0x04 (byte 4)	Nseq	0x04	Single	8-bit	b00010000
0x05 (byte 5)	Nseq	0x05	Single	8-bit	b00100000
0x06 (byte 6)	Nseq	0x06	Single	8-bit	b01000000
0x07 (byte 7)	Nseq	0x07	Single	8-bit	b10000000

Table 8-81 Cachable or Noncachable Write-Through STRH

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (byte 0)	Nseq	0x00	Single	16-bit	b00000011
0x01 (byte 1)	Nseq	0x00	Single	32-bit	b00000110 ^a
0x02 (byte 2)	Nseq	0x02	Single	16-bit	b00001100

Table 8-81 Cachable or Noncachable Write-Through STRH (continued)

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x03 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
		0x04			b00010000
0x04 (byte 4)	Nseq	0x04	Single	16-bit	b00110000
0x05 (byte 5)	Nseq	0x04	Single	32-bit	b01100000 ^a
0x06 (byte 6)	Nseq	0x06	Single	32-bit	b11000000
0x07 (byte 7)	Nseq	0x07	Single	8-bit	b10000000
	Nseq	0x08			b00000001

a. Denotes that HUNALIGNW is asserted for that transfer. This is only used for ARMv6 unaligned stores.

Table 8-82 Cachable or Noncachable Write-Through STR or STM1

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (byte 0) (word 0)	Nseq	0x00	Single	32-bit	b00001111
0x01 (byte 1)	Nseq	0x00	Single	32-bit	b00001110 ^a
		0x04		8-bit	b00010000
0x02 (byte 2)	Nseq	0x02	Single	16-bit	b00001100
		0x04			b00110000
0x03 (byte 3)	Nseq	0x03	Single	8-bit	b00001000
		0x04		32-bit	b01110000 ^a
0x04 (byte 4) (word 1)	Nseq	0x04	Single	32-bit	b11110000
0x05 (byte 5)	Nseq	0x04	Single	32-bit	b11100000 ^a
		0x08		8-bit	b00000001
0x06 (byte 6)	Nseq	0x06	Single	16-bit	b11000000
		0x08			b00000011

Table 8-82 Cacheable or Noncacheable Write-Through STR or STM1 (continued)

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x07 (byte 7)	Nseq	0x07	Single	8-bit	b10000000
		0x08		32-bit	b00000111 ^a
0x08 (byte 8) (word 2)	Nseq	0x08	Single	32-bit	b00001111
0x0C (word 3)	Nseq	0x08	Single	32-bit	b11110000
0x10 (word 4)	Nseq	0x10	Single	32-bit	b00001111
0x14 (word 5)	Nseq	0x14	Single	32-bit	b11110000
0x18 (word 6)	Nseq	0x18	Single	32-bit	b00001111
0x1C (word 7)	Nseq	0x1C	Single	32-bit	b11110000

a. Denotes that HUNALIGNW is asserted for that transfer. This is only used for ARMv6 unaligned stores.

Table 8-83 Cacheable or Noncacheable Write-Through STM2 to words 0, 1, 2, 3, 4, 5, or 6

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (word 0)	Nseq	0x00	Single	64-bit	b11111111
0x04 (word 1)	Nseq	0x04	Incr	32-bit	b11110000
	Seq	0x08	Incr	32-bit	b00001111
0x08 (word 2)	Nseq	0x08	Single	64-bit	b11111111
0x0C (word 3)	Nseq	0x0C	Incr	32-bit	b11110000
	Seq	0x10	Incr	32-bit	b00001111
0x10 (word 4)	Nseq	0x10	Single	64-bit	b11111111
0x14 (word 5)	Nseq	0x14	Incr	32-bit	b11110000
	Seq	0x18	Incr	32-bit	b00001111
0x18 (word 6)	Nseq	0x18	Single	64-bit	b11111111

Table 8-84 Cachable or Noncachable Write-Through STM2 to word 7

Address[4:0]	Operations
0x1C	STR to 0x1C + STR to 0x00

Table 8-85 Cachable or Noncachable Write-Through STM3 to words 0, 1, 2, 3, 4, or 5

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (word 0)	Nseq	0x00	Incr	32-bit	b00001111
	Seq	0x04			b11110000
		0x08			b00001111
0x04 (word 1)	Nseq	0x04	Incr	32-bit	b11110000
	Seq	0x08			b00001111
		0x0C			b11110000
0x08 (word 2)	Nseq	0x08	Incr	32-bit	b00001111
	Seq	0x0C			b11110000
		0x10			b00001111
0x0C (word 3)	Nseq	0x0C	Incr	32-bit	b11110000
	Seq	0x10			b00001111
		0x14			b11110000
0x10 (word 4)	Nseq	0x10	Incr	32-bit	b00001111
	Seq	0x14			b11110000
		0x18			b00001111
0x14 (word 5)	Nseq	0x14	Incr	32-bit	b11110000
	Seq	0x18			b00001111
		0x1C			b11110000

Table 8-86 Cachable or Noncachable Write-Through STM3 to words 6 or 7

Address[4:0]	Operations
0x18 (word 6)	STM2 to 0x18 + STR to 0x00
0x1C (word 7)	STR to 0x1C + STM2 to 0x00

Table 8-87 Cachable or Noncachable STM4 to word 0, 1, 2, 3, or 4

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW	
0x00 (word 0)	Nseq	0x00	Incr	64-bit	b11111111	
	Seq	0x08				
0x04 (word 1)	Nseq	0x04	Incr4	32-bit	b11110000	
	Seq	0x08				b00001111
		0x0C				b11110000
		0x10				b00001111
0x08 (word 2)	Nseq	0x08	Incr	64-bit	b11111111	
	Seq	0x10				
0x0C (word 3)	Nseq	0x0C	Incr4	32-bit	b11110000	
	Seq	0x10				b00001111
		0x14				b11110000
		0x18				b00001111
0x10 (word 4)	Nseq	0x10	Incr	64-bit	b11111111	
	Seq	0x18				

Table 8-88 Cachable or Noncachable STM4 to word 5, 6, or 7

Address[4:0]	Operations
0x14 (word 5)	STM3 to 0x14 + STR to 0x00
0x18 (word 6)	STM2 to 0x18 + STM2 to 0x00
0x1C (word 7)	STR to 0x1C + STM3 to 0x00

Table 8-89 Cachable or Noncachable STM5 to word 0, 1, 2, or 3

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (word 0)	Nseq	0x00	Incr	32-bit	b00001111
	Seq	0x04			b11110000
		0x08			b00001111
		0x0C			b11110000
		0x10			b00001111
0x04 (word 1)	Nseq	0x04	Incr	32-bit	b11110000
	Seq	0x08			b00001111
		0x0C			b11110000
		0x10			b00001111
		0x14			b11110000
0x08 (word 2)	Nseq	0x08	Incr	32-bit	b00001111
	Seq	0x0C			b11110000
		0x10			b00001111
		0x14			b11110000
		0x18			b00001111
0x0C (word 3)	Nseq	0x0C	Incr	32-bit	b11110000
	Seq	0x10			b00001111
		0x14			b11110000
		0x18			b00001111
		0x1C			b11110000

Table 8-90 Cachable or Noncachable STM5 to word 4, 5, 6, or 7

Address[4:0]	Operations
0x10 (word 4)	STM4 to 0x10 + STR to 0x00
0x14 (word 5)	STM3 to 0x14 + STM2 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM3 to 0x00
0x1C (word 7)	STR to 0x1C + STM4 to 0x00

Table 8-91 Cachable or Noncachable STM6 to word 0, 1, or 2

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (word 0)	Nseq	0x00	Incr	64-bit	b11111111
	Seq	0x08			b11111111
		0x10			b11111111
0x04 (word 1)	Nseq	0x04	Incr	32-bit	b11110000
	Seq	0x08			b00001111
		0x0C			b11110000
		0x10			b00001111
		0x14			b11110000
		0x18			b00001111
0x08 (word 2)	Nseq	0x08	Incr	64-bit	b11111111
	Seq	0x10			b11111111
		0x18			b11111111

Table 8-92 Cachable or Noncachable STM6 to word 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x0C (word 3)	STM5 to 0x0C + STR to 0x00
0x10 (word 4)	STM4 to 0x10 + STM2 to 0x00

Table 8-92 Cachable or Noncachable STM6 to word 3, 4, 5, 6, or 7 (continued)

Address[4:0]	Operations
0x14 (word 5)	STM3 to 0x14 + STM3 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM4 to 0x00
0x1C (word 7)	STR to 0x1C + STM5 to 0x00

Table 8-93 Cachable or Noncachable STM7 to word 0 or 1

Address[4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00 (word 0)	Nseq	0x00	Incr	32-bit	b00001111
	Seq	0x04			b11110000
		0x08			b00001111
		0x0C			b11110000
		0x10			b00001111
		0x14			b11110000
		0x18			b00001111
0x04 (word 1)	Nseq	0x04	Incr	32-bit	b11110000
	Seq	0x08			b00001111
		0x0C			b11110000
		0x10			b00001111
		0x14			b11110000
		0x18			b00001111
		0x1C			b11110000

Table 8-94 Cachable or Noncachable STM7 to word 2, 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x08 (word 2)	STM6 to 0x08 + STR to 0x00
0x0C (word 3)	STM5 to 0x0C + STM2 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM3 to 0x00

Table 8-94 Cachable or Noncachable STM7 to word 2, 3, 4, 5, 6, or 7 (continued)

Address[4:0]	Operations
0x14 (word 5)	STM3 to 0x14 + STM4 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM5 to 0x00
0x1C (word 7)	STR to 0x1C + STM6 to 0x00

Table 8-95 Cachable or Noncachable STM8 to word 0

HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
Nseq	0x00	Incr4	64-bit	b11111111
Seq	0x08			
	0x10			
	0x18			

Table 8-96 Cachable or Noncachable STM8 to word 1, 2, 3, 4, 5, 6, or 7

Address[4:0]	Operations
0x04 (word 1)	STM7 to 0x04 + STR to 0x00
0x08 (word 2)	STM6 to 0x08 + STM2 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM3 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM4 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM5 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM6 to 0x00
0x1C (word 7)	STR to 0x1C + STM7 to 0x00

Table 8-97 Cachable or Noncachable STM9

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STR to 0x00
0x04 (word 1)	STM7 to 0x04 + STM2 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM3 to 0x00

Table 8-97 Cachable or Noncachable STM9 (continued)

Address[4:0]	Operations
0x0C (word 3)	STM5 to 0x0C + STM4 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM5 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM6 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM7 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00

Table 8-98 Cachable or Noncachable STM10

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM2 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM3 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM4 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM5 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM6 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM7 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STR to 0x00

Table 8-99 Cachable or Noncachable STM11

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM3 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM4 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM5 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM6 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM7 to 0x00

Table 8-99 Cachable or Noncachable STM11 (continued)

Address[4:0]	Operations
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STR to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM2 to 0x00

Table 8-100 Cachable or Noncachable STM12

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM4 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM5 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM6 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM7 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM8 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00 + STR to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STM2 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM3 to 0x00

Table 8-101 Cachable or Noncachable STM13

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM5 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM6 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM7 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM8 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM8 to 0x00 + STR to 0x00
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00 + STM2 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STM3 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM4 to 0x00

Table 8-102 Cachable or Noncachable STM14

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM6 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM7 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM8 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM8 to 0x00 + STR to 0x00
0x10 (word 4)	STM4 to 0x10 + STM8 to 0x00 + STM2 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00 + STM3 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STM4 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM5 to 0x00

Table 8-103 Cachable or Noncachable STM15

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM7 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM8 to 0x00
0x08 (word 2)	STM6 to 0x08 + STM8 to 0x00 + STR to 0x00
0x0C (word 3)	STM5 to 0x0C + STM8 to 0x00 + STM2 to 0x00
0x10 (word 4)	STM4 to 0x10 + STM8 to 0x00 + STM3 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00 + STM4 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STM5 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM6 to 0x00

Table 8-104 Cachable or Noncachable STM16

Address[4:0]	Operations
0x00 (word 0)	STM8 to 0x00 + STM8 to 0x00
0x04 (word 1)	STM7 to 0x04 + STM8 to 0x00 + STR to 0x00
0x08 (word 2)	STM6 to 0x08 + STM8 to 0x00 + STM2 to 0x00
0x0C (word 3)	STM5 to 0x0C + STM8 to 0x00 + STM3 to 0x00

Table 8-104 Cachable or Noncachable STM16 (continued)

Address[4:0]	Operations
0x10 (word 4)	STM4 to 0x10 + STM8 to 0x00 + STM4 to 0x00
0x14 (word 5)	STM3 to 0x14 + STM8 to 0x00 + STM5 to 0x00
0x18 (word 6)	STM2 to 0x18 + STM8 to 0x00 + STM6 to 0x00
0x1C (word 7)	STR to 0x1C + STM8 to 0x00 + STM7 to 0x00

8.6.2 Half-line Write-Back

The values of **HTRANSW**, **HADDRW**, **HBURSTW**, **HSIZEW**, and **HBSTRBW** for half-line Write-Backs over the Data Write Interface are shown in Table 8-105.

Table 8-105 Half-line Write-Back

Read address [4:0]	Description	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00-0x07	Evicted cache line valid and lower half dirty	Nseq	0x00	Incr	64-bit	b11111111
		Seq	0x08			
	Evicted cache line valid and upper half dirty	Nseq	0x10	Incr		
		Seq	0x18			
0x08-0x0F	Evicted cache line valid and lower half dirty	Nseq	0x08	Single	64-bit	b11111111
		Seq	0x00			
	Evicted cache line valid and upper half dirty	Nseq	0x10	Incr		
		Seq	0x18			
0x10-0x17	Evicted cache line valid and lower half dirty	Nseq	0x00	Incr	64-bit	b11111111
		Seq	0x08			
	Evicted cache line valid and upper half dirty	Nseq	0x10	Incr		
		Seq	0x18			

Table 8-105 Half-line Write-Back (continued)

Read address [4:0]	Description	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x18-0x1F	Evicted cache line valid and lower half dirty	Nseq	0x00	Incr	64-bit	b11111111
		Seq	0x08			
0x18-0x1F	Evicted cache line valid and upper half dirty	Nseq	0x18	Single	64-bit	b11111111
		Seq	0x10			

8.6.3 Full-line Write-Back

The values of **HTRANSW**, **HADDRW**, **HBURSTW**, **HSIZEW**, and **HBSTRBW** for full-line Write-Backs, evicted cache line valid and both halves dirty, over the Data Write Interface are shown in Table 8-105 on page 8-61.

Table 8-106 Full-line Write-Back

Read address [4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x00-0x07	Nseq	0x00	Incr4	64-bit	b11111111
	Seq	0x08			
		0x10			
		0x18			
0x08-0x0F	Nseq	0x08	Wrap4	64-bit	b11111111
	Seq	0x10			
		0x18			
		0x00			
0x10-0x17	Nseq	0x10	Wrap4	64-bit	b11111111
	Seq	0x18			
		0x00			
		0x08			

Table 8-106 Full-line Write-Back (continued)

Read address [4:0]	HTRANSW	HADDRW	HBURSTW	HSIZEW	HBSTRBW
0x18-0x1F	Nseq	0x18	Wrap4	64-bit	b11111111
	Seq	0x00			
	Seq	0x08			
	Seq	0x10			

8.6.4 Store-exclusive

Store-exclusive is described in *HPROT[5]* and *HRESP[2]* on page 8-13.

8.6.5 Other AHB-Lite signals for Data Write port

The other AHB-Lite signals for the Data Write port are:

HSIDEBANDW[3:1]

Encodes the Inner Cachable TLB attributes, as shown in Table 8-107.

Table 8-107 HSIDEBANDW[3:1] encoding

HSIDEBAND[3:1]	Attribute
b000	Strongly ordered
b001	Device
b010	Inner Noncachable
b110	Inner Write-Through
bx11	Inner Write-Back

HSIDEBANDW[0] The TLB Sharable bit.

8.7 DMA Interface AHB-Lite transfers

AHB-Lite reads or writes over the DMA Interface use the standard AHB-Lite signals. The following AHB-Lite signals are also used:

HBURSTD[2:0]	Statically set to Single. Only single transfers are supported.
HTRANS[1:0]	Normally set to Idle, set to Nonseq to start a transfer.
HRESPD[0]	There is only one response because Retry and Split are not supported.
HUNALIGND	Set if an unaligned transfer is to be carried out.
HBSTRBD[7:0]	One byte lane for each byte in the 64-bit word to be transferred. Each bit is set to indicate that the corresponding byte lane in HRDATAD and HWDATAD is in use.
HSIZED[2:0]	8, 16, 32, or 64 bits.
HPROTD[4:2]	These bits encode the memory region attributes, as shown in Table 8-108.

Table 8-108 HPROTD[4:2] encoding

HPROTD[4:2]	Memory region attribute
b000	Strongly Ordered
b001	Device
b010	Outer Noncachable
b110	Outer Write-Through, No Allocate on Write
b111	Outer Write-Back, No Allocate on Write
b011	Outer Write-Back, Write Allocate

HPROTD[1] Encodes the CPSR state, as shown in Table 8-109.

Table 8-109 HPROTD[1] encoding

HPROTD[1]	CPSR state
0	User mode access
1	privileged mode access

HPROTD[0] Indicates that the transfer is an opcode fetch or data access, as shown in Table 8-109.

Table 8-110 HPROTD[0] encoding

HPROTD[0]	Attribute
0	Instruction
1	Data

HSIDEBANDD[3:1]

Encodes the Inner Cachable TLB attributes, as shown in Table 8-111.

Table 8-111 HSIDEBANDD[3:1] encoding

HSIDEBANDD[3:1]	Attribute
b000	Strongly ordered
b001	Device
b010	Inner Noncachable
b110	Inner Write-Through, No Allocate on Write
b111	Inner Write-Back, No Allocate on Write
b011	Inner Write-Back, Write Allocate

HSIDEBANDD[0] Set if the addressed memory region is Sharable.

8.8 Peripheral Interface AHB-Lite transfers

The tables in this section describe the Peripheral Interface behavior for reads and writes for the following interface signals:

- **HTRANSP[1:0]**
- **HADDRP[31:0]**
- **HBSTRBP[7:0]**
- **HSIZEP[2:0]**.

See *Other AHB-Lite signals for Peripheral Interface reads and writes* on page 8-67 for details of the other AHB-Lite signals.

8.8.1 Reads and writes

The values of **HTRANSP**, **HADDRP**, **HBURSTP**, and **HSIZEP** for example Peripheral Interface reads and writes are shown in Table 8-112.

Table 8-112 Example Peripheral Interface reads and writes

Example transfer (read or write)	HTRANSP	HADDRP	HBURSTP	HSIZEP
Words 0-7	Nseq	0x00	Incr	Word
	Seq	0x04		
	Nseq	0x08		
	Seq	0x0C		
	Nseq	0x10		
	Seq	0x14		
	Nseq	0x18		
	Seq	0x1C		
Words 0-3	Nseq	0x00	Incr	Word
	Seq	0x04		
	Nseq	0x08		
	Seq	0x0C		

Table 8-112 Example Peripheral Interface reads and writes (continued)

Example transfer (read or write)	HTRANSF	HADDRP	HBURSTP	HSIZEP
Words 0-2	Nseq	0x00	Incr	Word
	Seq	0x04		
	Nseq	0x08		
Words 0-1	Nseq	0x00	Incr	Word
	Seq	0x04		
Word 2	Nseq	0x08	Single	Word
Word 0, bytes 0 and 1	Nseq	0x00	Single	Halfword
Word 1, bytes 2 and 3	Nseq	0x06	Single	Halfword
Word 2, byte 3	Nseq	0x0B	Single	Byte

8.8.2 Other AHB-Lite signals for Peripheral Interface reads and writes

The other AHB-Lite signals used in the Peripheral Interface are:

HWRITEP	When HIGH indicates a write transfer, when LOW indicates a read.
HPROTP[4:0]	HPROTP[4:2] encodes the memory region attributes, as shown in Table 8-113.

Table 8-113 HPROTP[4:2] encoding

HPROTP[4:2]	Memory region attribute
b000	Strongly Ordered
b001	Device
b010	Outer Noncacheable
b110	Outer Write-Through, No Allocate on Write
b111	Outer Write-Back, No Allocate on Write
b011	Outer Write-Back, Write Allocate

HPROTP[1] encodes the CPSR state, as shown in Table 8-114.

Table 8-114 HPROTP[1] encoding

HPROTP[1]	CPSR state
0	User mode access
1	Privileged mode access

HPROTP[0] statically 1 indicating a data access.

HSIDEBANDP[4:0] Statically set to b0010 to indicate a Non-shared Device access.

8.9 AHB-Lite

AHB-Lite is a subset of the full AHB specification for use in designs where only a single bus master is used. This can either be a simple single-master system, as shown in Figure 8-6, or a multi-layer AHB-Lite system where there is only one AHB master per layer.

Figure 8-6 shows a block diagram of a single-master system.

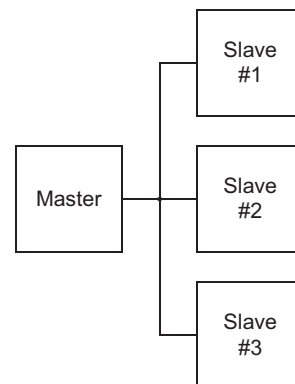


Figure 8-6 AHB-Lite single-master system

AHB-Lite simplifies the AHB specification by removing the protocol required for multiple bus masters, which includes the Request or Grant protocol to the arbiter and the Split or Retry responses from slaves.

Masters designed to the AHB-Lite interface specification are significantly simpler in terms of interface design, than a full AHB master. AHB-Lite enables faster design and verification of these masters, and you can add a standard off-the-shelf bus mastering wrapper to convert an AHB-Lite master for use in a full AHB system.

Any master that is already designed to the full AHB specification can be used in an AHB-Lite system with no modification.

The majority of AHB slaves can be used interchangeably in either an AHB or AHB-Lite system. This is because AHB slaves that do not use either the Split or Retry response are automatically compatible with both the full AHB and the AHB-Lite specification. It is only existing AHB slaves that do use Split or Retry responses that require you to use an additional standard off-the-shelf wrapper in your AHB-Lite system.

Any slave designed for use in an AHB-Lite system works in both a full AHB and an AHB-Lite design.

8.9.1 Specification

The AHB-Lite specification differs from the full AHB specification in the following ways:

- Only one master. There is only one source of address, control, and write data, so no master-to-slave multiplexor is required.
- No arbiter. None of the signals associated with the arbiter are used.
- The master has no **HBUSREQ** output. If such an output exists on a master, it is left unconnected.
- The master has no **HGRANT** input. If such an input exists on a master, it is tied HIGH.
- Slaves must not produce either a Split or Retry response.
- The AHB-Lite lock signal is the same as **HMASTLOCK** and it has the same timing as the address bus and other control signals. If a master has an **HLOCK** output, it can be retimed to generate **HMASTLOCK**.
- The AHB-Lite lock signal must remain stable throughout a burst of transfers, in the same way that other control signals must remain constant throughout a burst.

8.9.2 Compatibility

Table 8-115 shows how masters and slaves designed for use in either full AHB or AHB-Lite can be used interchangeably in different systems.

Table 8-115 AHB-Lite interchangeability

Component	Full AHB system	AHB-Lite system
Full AHB master	Yes	Yes
AHB-Lite master	Use standard AHB master wrapper	Yes
AHB slave (no Split or Retry)	Yes	Yes
AHB slave with Split or Retry	Yes	Use standard AHB slave wrapper

8.9.3 AHB-Lite master interface

An AHB-Lite master has the same signal interface as a full AHB bus master, except that it does not support **HBUSREQx** and **HGRANTx**.

The lock functionality is still required because the master might be performing a transfer to a multi-interface slave. The slave must be given an indication that no other transfer must occur to the slave when the master requires locked access.

An AHB-Lite master is not required to support either the Split or Retry response and only the Okay and Error responses are required, so the AHB-Lite master interface does not require the **HRESP[1]** input.

8.9.4 AHB-Lite advantages

The advantage of using the AHB-Lite protocol is that the bus master does not have to support the following cases:

- Losing ownership of the bus. The clock enable for the master can be derived from the **HREADY** signal on the bus.
- Early terminated bursts. There is no requirement for the master to rebuild a burst due to early termination, because the master always has access to the bus.
- Split or Retry transfer responses. There is no requirement for the master to retain the address of the last transfer to be able to restart a previous transfer.

8.9.5 AHB-Lite conversion to full AHB

A standard wrapper is available to convert an AHB-Lite master to make it a full AHB master. This wrapper adds support for the features described above.

Because the AHB-Lite master has no bus request signal available, the wrapper generates this directly from the **HTRANS** signals.

8.9.6 AHB-Lite slaves

AHB slaves that do not use either the Split or Retry response can be used in either a full AHB or AHB-Lite system.

You can use any slave that does use Split or Retry responses in an AHB-Lite system by adding a standard wrapper. This wrapper provides the ability to store the previous transfer in the case of a Split or Retry response and restart the transfer when appropriate. This wrapper is very similar to that required to convert an AHB-Lite master for use in a full AHB system.

For compatibility with Multi-layer AHB, it is required that all AHB-Lite slaves still retain support for early terminated bursts.

8.9.7 Block diagram

Figure 8-7 shows a more detailed block diagram, including decoder and slave-to-master multiplexor connections.

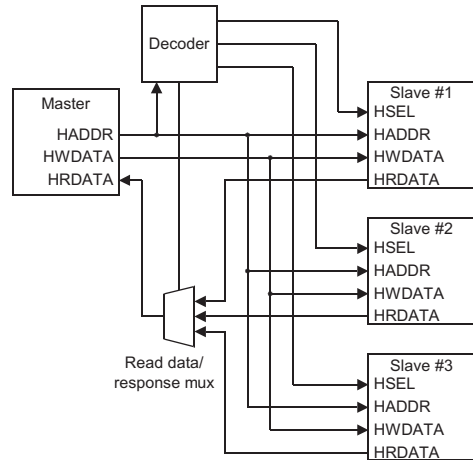


Figure 8-7 AHB-Lite block diagram

Chapter 9

Clocking and Resets

This chapter describes the clocking and reset options available for ARM1136JF-S processors. It contains the following sections:

- *ARM1136JF-S clocking* on page 9-2
- *Reset* on page 9-7
- *Reset modes* on page 9-8.

9.1 ARM1136JF-S clocking

The ARM1136JF-S processor has six functional clock inputs. These are paired into three clock domains. Externally to ARM1136JF-S, you must connect together **CLKIN** and **FREECLKIN**. The same is true of:

- **HCLKIRW** and **FREEHCLKIRW**
- **HCLKPD** and **FREEHCLKPD**.

For information on how the clock domains are implemented see *ARM1136 Implementation Guide*.

For the purposes of this chapter, you can ignore **FREECLKIN**, **FREEHCLKIRW**, and **FREEHCLKPD** clock domains. Logically, the clock domains are:

Table 9-1 AHB clock domains

Logical blocks	Clock	Domain
Core	CLKIN	Core
Peripheral port DMA port	HCLKPD	PD
Instruction Fetch port Data Read port Data Write port	HCLKIRW	IRW

All clocks can be stopped indefinitely without loss of state.

You can preconfigure the ARM1136JF-S processor so that each clock domain can operate synchronously or asynchronously to the core clock domain.

9.1.1 Synchronous clocking

The benefit of synchronous clocking is that it is possible to reduce the read and write latency by removing the synchronization register in the external request path. However, due to the integer relationship of the clocks, it might not be possible to get the maximum performance from the core due to constraints placed on the bus frequency by components such as SDRAM controllers. It is not possible to run the core slower than the bus.

9.1.2 Asynchronous clocking

The main benefit of asynchronous clocking is that the core performance can be maximized, while running the bus at a fixed system frequency. Additionally, in sleep-mode situations when the core is not required to do much work, the frequency can be lowered to reduce power consumption.

For low-power operation, if the ARM1136JF-S processor is configured asynchronously, it can be operated with the core clock slower than the bus clock. See Chapter 10 *Power Control* for details of other aspects of power management.

9.1.3 Synchronization

For each AHB clock domain the ARM1136JF-S processor provides an AHB clock and two control inputs that you can use to configure for synchronous or asynchronous operation, see Table 9-2.

Table 9-2 Clock domain control signals

Clock domain	Control signals
IRW	SYNCENIRW HSYNCENIRW
PD	SYNCENPD HSYNCENPD

These are state inputs that select a bypass path for every synchronization register, if they are tied HIGH, to enable synchronous operation.

Figure 9-1 on page 9-4 shows the synchronization between AHB and core clock domains.

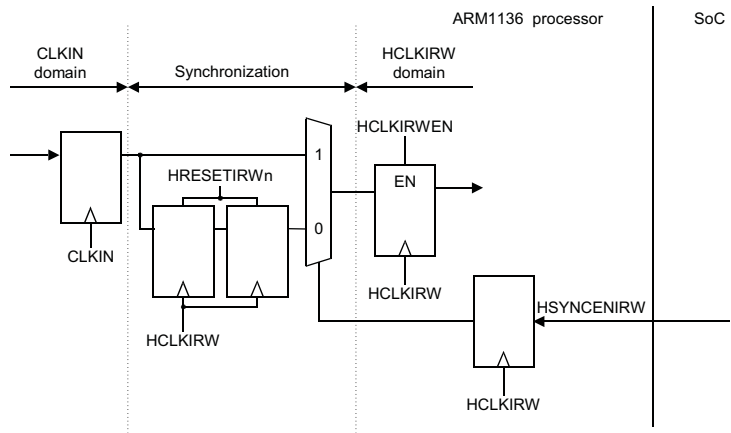


Figure 9-1 Synchronization between AHB and core clock domains

Figure 9-2 shows the synchronization between core clock and AHB domains.

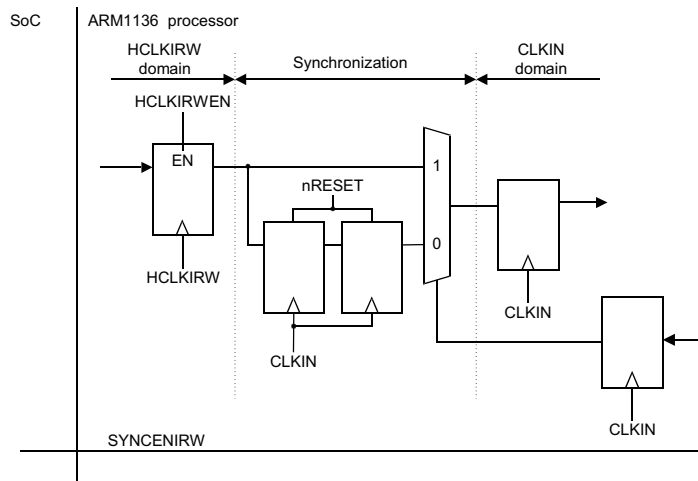


Figure 9-2 Synchronization between core clock and AHB domains

There are two synchronizer control signals per port to provide a clean static-timing view of the interface. Logically these must be held at the same level.

For a given AHB clock domain, if synchronous operation is selected then the clock inputs for that domain must be connected to the same logical input as **CLKIN**. In this case, the AHB-Lite interfaces in the given clock domain can run at n:1 (AHB:Core) ratio to **CLKIN** using the enable signals, see Table 9-3.

Table 9-3 Synchronous mode clock enable signals

Domain	AHB port	Enable signals
IRW	Instruction Fetch	HCLKIRWEN
	Data Read	
	Data Write	
PD	DMA	HCLKDEN
	Peripheral	HCLKPEN

9.1.4 Read latency penalty for synchronous operation

The Nonsequential Noncacheable read-latency for synchronous 1:1 clocking with zero-wait-state AHB is a six-cycle penalty over a cache hit (where data is returned in the DC2 cycle), on the data side, and a five-cycle penalty over a cache hit on the instruction side.

In the first cycle after the data cache miss, a read-after-write hazard check is performed against the contents of the Write Buffer. This prevents stalling while waiting for the Write Buffer to drain. Following that, a request is made to the AHB-Lite interface, and subsequently a transfer is started on the AHB. In the next cycle data is returned to the AHB-Lite interface, from where it is returned first to the level one clock domain before being forwarded to the core. This is shown in Figure 9-3.

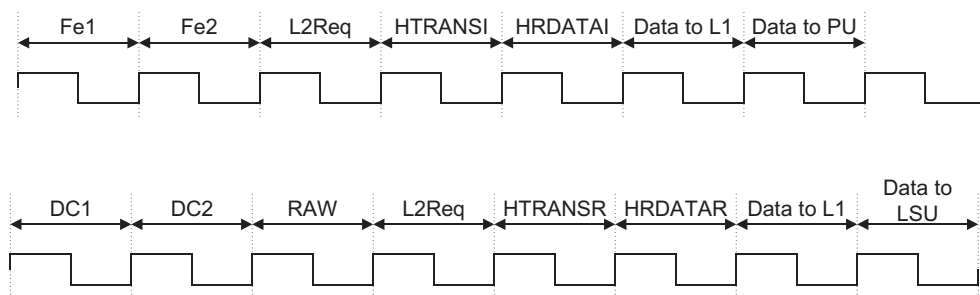


Figure 9-3 Read latency for synchronous 1:1 clocking

The same sequence appears on the I-Side, except that there is less to do in the equivalent RAW cycle.

9.2 Reset

The ARM1136JF-S processor has the following reset inputs:

HRESETPDn	The HRESETPDn is the reset signal for the PD domain.
HRESETIRWn	The HRESETIRWn is the reset signal for the IRW domain.
nRESETIN	The nRESETIN signal is the main processor reset that initializes the majority of the ARM1136JF-S logic.
DBGnTRST	The DBGnTRST signal is the DBGTAP reset.
nPORESETIN	The nPORESETIN signal is the power-on reset that initializes the CP14 debug logic. See <i>CP14 registers reset</i> on page 13-24 for details.

All of these are active LOW signals that reset logic in the ARM1136JF-S processor. You must take care when designing the logic to drive these reset signals.

9.3 Reset modes

The reset signals present in the ARM1136JF-S processor design to enable you to reset different parts of the design independently. The reset signals, and the combinations and possible applications that you can use them in, are shown in Table 9-4.

Table 9-4 Reset modes

Reset mode	nRESETIN	DBGnTRST	nPORESETIN	Application
Power-on reset	0	x	0	Reset at power up, full system reset. Hard reset or cold reset.
Processor reset	0	x	1	Reset of processor core only, watchdog reset. Soft reset or warm reset.
DBGTAP reset	1	0	1	Reset of DBGTAP logic.
Normal	1	x	1	No reset. Normal run mode.

9.3.1 Power-on reset

You must apply power-on or *cold* reset to the ARM1136JF-S processor when power is first applied to the system. In the case of power-on reset, the leading (falling) edge of the reset signals, **nRESETIN** and **nPORESETIN**, does not have to be synchronous to **CLKIN**. Because the **nRESETIN** and **nPORESETIN** signals are synchronized within the ARM1136JF-S processor, you do not have to synchronize these signals. Figure 9-4 shows the application of power-on reset.

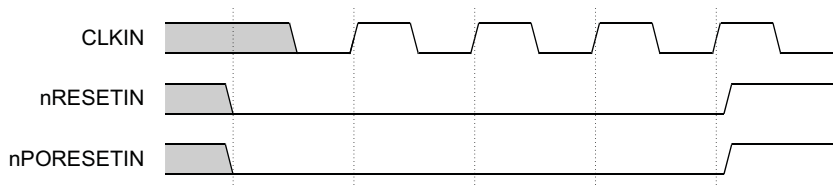


Figure 9-4 Power-on reset

It is recommended that you assert the reset signals for at least three **CLKIN** 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.

It is not necessary to assert **DBGnTRST** on power-up.

9.3.2 CP14 debug logic

Because the **nPORESETIN** signal is synchronized within the ARM1136JF-S processor, you do not have to synchronize this signal.

9.3.3 Processor reset

A processor or *warm* reset initializes the majority of the ARM1136JF-S processor, excluding the ARM1136JF-S DBGTAP controller and the EmbeddedICE-RT logic. Processor reset is typically used for resetting a system that has been operating for some time, for example, watchdog reset.

Because the **nRESETIN** signal is synchronized within the ARM1136JF-S processor, you do not have to synchronize this signal.

9.3.4 DBGTAP reset

DBGTAP reset initializes the state of the ARM1136JF-S DBGTAP controller. DBGTAP reset is typically used by the RealView™ ICE module for hot connection of a debugger to a system.

DBGTAP reset enables initialization of the DBGTAP controller without affecting the normal operation of the ARM1136JF-S processor.

Because the **DBGnTRST** signal is synchronized within the ARM1136JF-S processor, you do not have to synchronize this signal.

9.3.5 Normal operation

During normal operation, neither processor reset nor power-on reset is asserted. If the DBGTAP port is not being used, the value of **DBGnTRST** does not matter.

Chapter 10

Power Control

This chapter describes the ARM1136JF-S power control functions. It contains the following sections:

- *About power control* on page 10-2
- *Power management* on page 10-3.

10.1 About power control

The features of the ARM1136JF-S processor that improve energy efficiency include:

- accurate branch and return prediction, reducing the number of incorrect instruction fetch and decode operations
- use of physically addressed caches, which reduces the number of cache flushes and refills, saving energy in the system
- the use of MicroTLBs reduces the power consumed in translation and protection look-ups each cycle
- the caches use sequential access information to reduce the number of accesses to the TagRAMs and to unwanted Data RAMs.

In the ARM1136JF-S processor extensive use is also made of gated clocks and gates to disable inputs to unused functional blocks. Only the logic actively in use to perform a calculation consumes any dynamic power.

10.2 Power management

ARM1136JF-S processors support three levels of power management:

- *Run mode*
- *Standby mode*
- *Shutdown mode* on page 10-4
- plus partial support for a fourth level, *Dormant mode* on page 10-4.

10.2.1 Run mode

Run mode is the normal mode of operation in which all of the functionality of the core is available.

10.2.2 Standby mode

Standby mode disables most of the clocks of the device, while keeping the design powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the standby state.

The transition from Standby mode to Run mode is caused by the arrival of an interrupt (whether masked or unmasked), a debug request (whether debug is enabled or disabled) or reset.

The debug request can be generated by an externally generated debug request, using the **EDBGRQ** pin on the ARM1136JF-S processor, or from a Debug Halt instruction issued to the ARM1136JF-S processor through the debug scan chains.

Entry into Standby Mode is performed by executing the Wait For Interrupt CP15 operation. To ensure that the memory system is not affected by the entry into the Standby state, the following operations are performed:

- A Drain Write Buffer operation ensures that all explicit memory accesses occurring in program order before the Wait For Interrupt have completed. This avoids any possible deadlocks that could be caused in a system where memory access triggers or enables an interrupt that the core is waiting for. This might require some TLB page table walks to take place as well.
- The DMA continues running during a Wait For Interrupt and any queued DMA operations are executed as normal. This enables an application using the DMA to set up the DMA to signal an interrupt once the DMA has completed, and then for the application to issue a Wait For Interrupt instruction. The degree of power-saving while the DMA is running is less than is the case if the DMA is not running.

- Any other memory accesses that have been started at the time that the Wait For Interrupt instruction is executed are completed as normal. This ensures that the level two memory system does not see any disruption caused by the Wait For Interrupt.
- The debug channel remains active throughout a Wait For Interrupt. You must tie the **DBGTCKEN** signal to VSS to avoid clocking unnecessary logic to ensure best power-saving when not using debug.

Systems using the VIC interface must ensure that the VIC is not masking any interrupts that are required for restarting the ARM1136JF-S processor when in this mode of operation.

After the processor clocks have been stopped the signal **STANBYWFI** is asserted to indicate that the ARM1136JF-S processor is in Standby mode.

10.2.3 Shutdown mode

Shutdown mode has the entire device powered down, and you must externally save all state, including cache and TCM state. The processor is returned to Run mode by the assertion of Reset. This state saving is performed with interrupts disabled, and finishes with a Drain Write Buffer operation. When all the state of the ARM1136JF-S processor is saved the ARM1136JF-S processor executes a Wait For Interrupt instruction. The signal **STANBYWFI** is asserted to indicate that the processor can enter Shutdown mode.

10.2.4 Dormant mode

Dormant mode enables the core to be powered down, leaving the caches and the *Tightly-Coupled Memory* (TCM) powered up and maintaining their state.

The software visibility of the Valid bits is provided to enable an implementation to be extended for Dormant mode, but some hardware modification of the RAM blocks during implementation to include an input clamp is required for the full implementation of Dormant mode.

Considerations for Dormant mode

Dormant mode is partially supported on ARM1136JF-S processors, because care is required in implementing this on a standard synthesizable flow. The RAM blocks that are to remain powered up must be implemented on a separate power domain, and there is a requirement to clamp all of the inputs to the RAMs to a known logic level (with the chip enable being held inactive). This clamping is not implemented in gates as part of the default synthesis flow because it contributes to a tight critical path.

Designers wanting to implement Dormant mode must add these clamps around the RAMs, either as explicit gates in the RAM power domain, or as pull-down transistors that clamp the values while the core is powered down.

The RAM blocks that remain must be powered up in Dormant mode, if it is implemented, are:

- all Data RAMs associated with the cache and tightly-coupled memories
- all TagRAMs associated with the cache
- all Valid RAMs and Dirty RAMs associated with the cache.

The state of the Branch Target Address Cache is not maintained on entry into Dormant mode.

Implementations of the ARM1136JF-S processor can optionally disable the RAMs associated with the main TLB, so that a trade-off can be made between Dormant mode leakage power and the recovery time.

Before entering Dormant mode, the state of the ARM1136JF-S processor, excluding the contents of the RAMs that remain powered up in dormant mode, must be saved to external memory. These state saving operations must ensure that the following occur:

- All ARM registers, including CPSR and SPSR registers are saved.
- Any DMA operations in progress are stopped.
- All CP15 registers are saved, including the DMA state.
- All VFP registers are saved if the VFP contains defined state.
- Any locked entries in the main TLB are saved.
- All debug-related state are saved.
- The Master Valid bits for the cache and SmartCache are saved. These are accessed using CP15 register c15 as described in *Cache and main TLB Master Valid Registers* on page 3-37.
- If the main TLB is powered down on entry into the Dormant mode, then the Valid bits of the main TLB are saved. These are accessed using CP15 register c15 as described in *Cache and main TLB Master Valid Registers* on page 3-37.
- A Drain Write Buffer instruction is executed to ensure that all state saving has been completed.

A Wait For Interrupt CP15 operation is then executed, enabling the signal **STANBYWFI** to indicate that the ARM1136JF-S processor can enter Dormant mode.

- On entry into Dormant mode, the Reset signal to the ARM1136JF-S processor must be asserted by the external power control mechanism.

Transition from Dormant state to Run state is triggered by the external power controller asserting Reset to the ARM1136JF-S processor until the power to the processor is restored. When power has been restored the core leaves reset and, by interrogating the external power control, can determine that the saved state must be restored.

10.2.5 Communication to the Power Management Controller

The Power Management Controller performs the powering up and powering down of the power domains of the processor. The communication mechanism between the ARM1136JF-S processor and the Power Management Controller is a memory-mapped controller, which is accessed by the processor performing Strongly-Ordered accesses to it.

The Power Management Controller is informed of what powerdown state to be in on seeing the **STANBYWFI** signal from the ARM1136JF-S processor.

The **STANBYWFI** signal can also be used to signal that the ARM1136JF-S processor is ready to have its power state changed. **STANBYWFI** is asserted in response to a Wait For Interrupt operation.

Chapter 11

Coprocessor Interface

This chapter describes the ARM1136JF-S coprocessor interface. It contains the following sections:

- *About the ARM1136JF-S coprocessor interface* on page 11-2
- *Coprocessor pipeline* on page 11-3
- *Token queue management* on page 11-12
- *Token queues* on page 11-16
- *Data transfer* on page 11-20
- *Operations* on page 11-25
- *Multiple coprocessors* on page 11-28.

11.1 About the ARM1136JF-S coprocessor interface

The ARM1136JF-S processor supports the connection of on-chip coprocessors through an external coprocessor interface. All types of coprocessor instruction are supported.

The ARM instruction set supports the connection of 16 coprocessors, numbered 0-15, to an ARM processor. In ARM1136JF-S processors, the following coprocessor numbers are reserved:

CP10	VFP control
CP11	VFP control
CP14	Debug and ETM control
CP15	System control.

You can use CP0-9, CP12, and CP13 for your own external coprocessors.

The ARM1136JF-S processor is designed to pass instructions to several coprocessors and exchange data with them. These coprocessors are intended to run in step with the core and are pipelined in a similar way to the core. Instructions are passed out of the Fetch stage of the core pipeline to the coprocessor and decoded. The decoded instruction is passed down its own pipeline. Coprocessor instructions can be canceled by the core if a condition code fails, or the entire coprocessor pipeline can be flushed in the event of a mispredicted branch. Load and store data are also required to pass between the core *Logic Store Unit* (LSU) and the coprocessor pipeline.

The coprocessor interface operates over a two-cycle delay. Any signal passing from the core to the coprocessor, or from the coprocessor to the core, is given a whole clock cycle to propagate from one to the other. This means that a signal crossing the interface is clocked out of a register on one side of the interface and clocked directly into another register on the other side. No combinatorial process must intervene. This constraint exists because the core and coprocessor can be placed a considerable distance apart and generous timing margins are necessary to cover signal propagation times. This delay in signal propagation makes it difficult to maintain pipeline synchronization, ruling out a tightly-coupled synchronization method.

ARM1136JF-S processors implement a token-based pipeline synchronization method that allows some slack between the two pipelines, while ensuring that the pipelines are correctly aligned for crucial transfers of information.

11.2 Coprocessor pipeline

The coprocessor interface achieves loose synchronization between the two pipelines by exchanging tokens from one pipeline to the other. These tokens pass down queues between the pipelines and can carry additional information. In most cases the primary purpose of the queue is to carry information about the instruction being processed, or to inform one pipeline of events occurring in the other.

Tokens are generated whenever a coprocessor instruction passes out of a pipeline stage associated with a queue into the next stage. These tokens are picked up by the partner stage in the other pipeline, and used to enable the corresponding instruction in that stage to move on. The movement of coprocessor instructions down each pipeline is matched exactly by the movement of tokens along the various queues that connect the pipelines.

If a pipeline stage has no associated queue, the instruction contained within it moves on in the normal way. The coprocessor interface is data-driven rather than control-driven.

11.2.1 Coprocessor instructions

Each coprocessor can only execute a subset of all possible coprocessor instructions. Coprocessors reject those instructions they cannot handle. Table 11-1 lists all the coprocessor instructions supported by ARM1136JF-S processors and gives a brief description of each. For more details of coprocessor instructions, see the *ARM Architecture Reference Manual*.

Table 11-1 Coprocessor instructions

Instruction	Data transfer	Vectored	Description
CDP	None	No	Processes information already held within the coprocessor
MRC	Store	No	Transfers information from the coprocessor to the core registers
MCR	Load	No	Transfers information from the core registers to the coprocessor
MRRC	Store	No	Transfers information from the coprocessor to a pair of registers in the core

Table 11-1 Coprocessor instructions (continued)

Instruction	Data transfer	Vectored	Description
MCRR	Load	No	Transfers information from a pair of registers in the core to the coprocessor
STC	Store	Yes	Transfers information from the coprocessor to memory and might be iterated to transfer a vector
LDC	Load	Yes	Transfers information from memory to the coprocessor and might be iterated to transfer a vector

The coprocessor instructions fall into three groups:

- loads
- stores
- processing instructions.

The load and store instructions enable information to pass between the core and the coprocessor. Some of them might be vectored. This enables several values to be transferred in a single instruction. This typically involves the transfer of several words of data between a set of registers in the coprocessor and a contiguous set of locations in memory.

Other instructions, for example MCR and MRC, transfer data between core and coprocessor registers. The CDP instruction controls the execution of a specified operation on data already held within the coprocessor, writing the result back into a coprocessor register, or changing the state of the coprocessor in some other way. Opcode fields within the CDP instruction determine which operation is to be carried out.

The core pipeline handles both core and coprocessor instructions. The coprocessor, on the other hand, only deals with coprocessor instructions, so the coprocessor pipeline is likely to be empty for most of the time.

11.2.2 Coprocessor control

The coprocessor communicates with the core using several signals. Most of these signals control the synchronizing queues that connect the coprocessor pipeline to the core pipeline. The signals used for general coprocessor control are shown in Table 11-2.

Table 11-2 Coprocessor control signals

Signal	Description
CLKIN	This is the clock signal from the core.
RESET	This is the reset signal from the core.
ACPNUM[3:0]	This is the fixed number assigned to the coprocessor, and is in the range 0-13. Coprocessor numbers 10, 11, 14, and 15 are reserved for system control coprocessors.
ACPENABLE	When set, enables the coprocessor to respond to signals from the core.
ACPPRIV	When asserted, indicates that the core is in privileged mode. This might affect the execution of certain coprocessor instructions.

11.2.3 Pipeline synchronization

Figure 11-1 on page 11-6 shows an outline of the core and coprocessor pipelines and the synchronizing queues that communicate between them. Each queue is implemented as a very short *First In First Out* (FIFO) buffer.

No explicit flow control is required for the queues, because the pipeline lengths between the queues limits the number of items any queue can hold at any time. The geometry used means that only three slots are required in each queue.

The only status information required is a flag to indicate when the queue is empty. This is monitored by the receiving end of the queue, and determines if the associated pipeline stage can move on. Any information carried by the queue can also be read and acted upon at the same time.

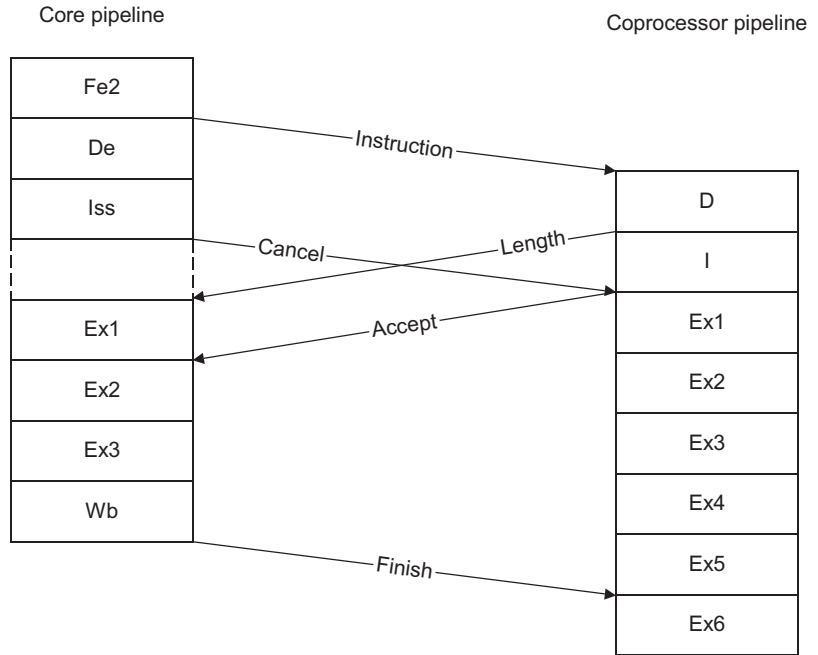


Figure 11-1 Core and coprocessor pipelines

Figure 11-2 on page 11-7 provides a more detailed picture of the pipeline and the queues maintained by the coprocessor.

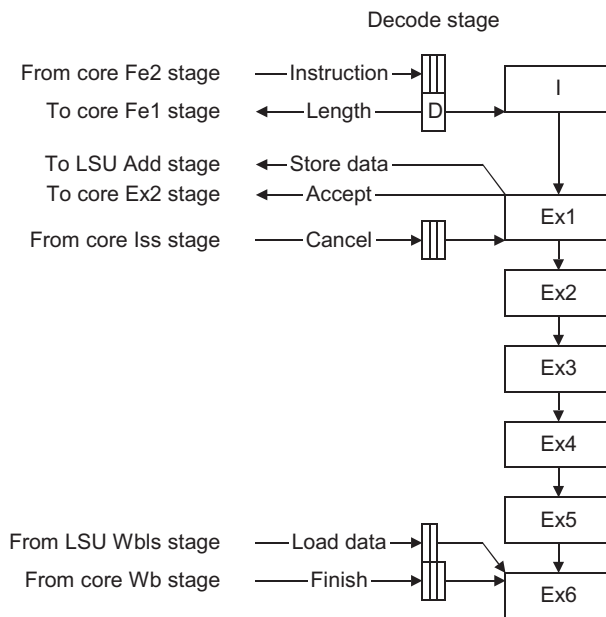


Figure 11-2 Coprocessor pipeline and queues

The instruction queue incorporates the instruction decoder and returns the length to the Ex1 stage of the core, using the length queue, which is maintained by the core. The coprocessor I stage sends a token to the core Ex2 stage through the accept queue, which is also maintained by the core. This token indicates to the core if the coprocessor is accepting the instruction in its I stage, or bouncing it.

The core can cancel an instruction currently in the coprocessor Ex1 stage by sending a signal with the token passed down the cancel queue. When a coprocessor instruction reads the Ex6 stage it might retire. How it retires depends on the instruction:

- Load instructions retire when they find load data available in the load data queue, see *Loads* on page 11-21
- Store instructions retire as soon as they leave the Ex1 stage, and are removed from the pipeline, see *Stores* on page 11-23
- CDP instructions retire when they read a token passed by the core down the finish queue.

Data transfer uses the load data and store data queues, which are shown in Figure 11-2 and explained in *Data transfer* on page 11-20.

11.2.4 Pipeline control

The coprocessor pipeline is very similar to the core pipeline, but lacks the fetch stages. Instructions are passed from the core directly into the Decode stage of the coprocessor pipeline, which takes the form of a FIFO queue.

The Decode stage then decodes the instruction, rejecting non-coprocessor instructions and any coprocessor instructions containing a nonmatching coprocessor number.

The length of any vectored data transfer is also decided at this point and sent back to the core. The decoded instruction then passes into the issue (I) stage. This stage decides if this particular instance of the instruction can be accepted. If it cannot, because it addresses a non-existent register, the instruction is bounced, informing the core that it cannot be accepted.

If the instruction is both valid and executable, it then passes down the execution pipeline, Ex1 to Ex6. At the bottom of the pipeline, in Ex6, the instruction waits for retirement, which it can do when it receives a matching token from another queue fed by the core.

Figure 11-3 on page 11-9 shows the coprocessor pipeline, the main fields within each stage, and the main control signals. Each stage controls the flow of information from the previous stage in the pipeline by passing its Enable signal back. When a pipeline stage is not enabled, it cannot accept information from the previous stage.

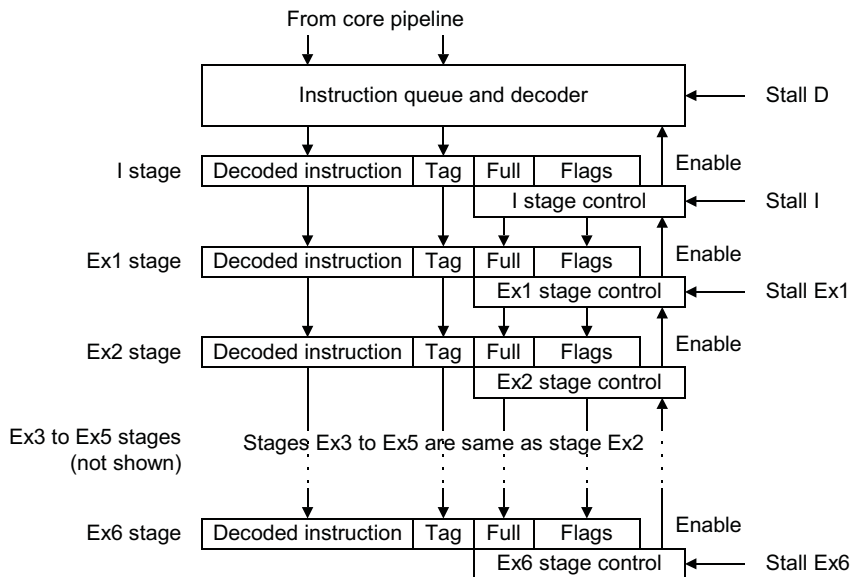


Figure 11-3 Coprocessor pipeline

Each pipeline stage contains a decoded instruction, and a tag, plus a few status flags:

- Full flag** This flag is set whenever the pipeline stage contains an instruction.
- Dead flag** This flag is set to indicate that the instruction in the stage is a phantom. See *Cancel operations* on page 11-25.
- Tail flag** This flag is set to indicate that the instruction is the tail of an iterated instruction. See *Loads* on page 11-21.

There might also be other flags associated with the decoding of the instruction.

Each stage is controlled not only by its own state, but also by external signals and signals from the following stage, as follows:

- Stall** This signal prevents the stage from accepting a new instruction or passing its own instruction on, and only affects the D, I, Ex1, and Ex6 stages.
- Iterate** This signal indicates that the instruction in the stage must be iterated in order to implement a multiple load/store and only applies to the I stage.
- Enable** This signal indicates that the next stage in the pipeline is ready to accept data from the current stage.

These signals are combined with the current state of the pipeline to determine if the stage can accept new data, and what the new state of the stage is going to be. Table 11-3 shows how the new state of the pipeline stage is derived.

Table 11-3 Pipeline stage update

Stall	Enable input	Iterate	State	Enable	To next stage	Remarks
0	0	X	Empty	1	None	Bubble closing
0	0	X	Full	0	-	Stalled by next stage
0	1	0	Empty	1	None	Normal pipeline movement
0	1	0	Full	1	Current	Normal pipeline movement
0	1	1	Empty	-	-	Impossible
0	1	1	Full	0	Current	Iteration (I stage only)
1	X	X	X	0	None	Stalled (D, I, Ex1, and Ex6 only)

The Enable input comes from the next stage in the pipeline and indicates if data can be passed on. In general, if this signal is unasserted the pipeline stage cannot receive new data or pass on its own contents. However, if the pipeline stage is empty it can receive new data without passing any data on to the next stage. This is known as *bubble closing*, because it has the effect of filling up empty stages in the pipeline by enabling them to move on while lower stages are stalled.

11.2.5 Instruction tagging

It is sometimes necessary for the core to be able to identify instructions in the coprocessor pipeline. This is necessary for flushing (see *Flush operations* on page 11-26) so that the core can indicate to the coprocessor which instructions are to be flushed. The core therefore gives each instruction sent to the coprocessor a tag, which is drawn from a pool of values large enough so that all the tags in the pipeline at any moment are unique. Sixteen tags are sufficient to achieve this, requiring a four-bit tag field. Each time a tag is assigned to an instruction, the tag number is incremented modulo 16 to generate the next tag.

The flushing mechanism is simplified because successive coprocessor instructions have contiguous tags. The core manages this by only incrementing the tag number when the instruction passed to the coprocessor is a coprocessor instruction. This is done after sending the instruction, so the tag changes after a coprocessor instruction is sent, rather than before. It is not possible to increment the tag before sending the instruction because

the core has not yet had time to decode the instruction to determine what kind of instruction it is. When the coprocessor Decode stage removes the non-coprocessor instructions, it is left with an instruction stream carrying contiguous tags.

The tags can also be used to verify that the sequence of tokens moving down the queues matches the sequence of instructions moving down the core and coprocessor pipelines.

11.2.6 Flush broadcast

If a branch has been mispredicted, it might be necessary for the core to flush both pipelines. Because this action potentially affects the entire pipeline, it is not passed across in a queue but is broadcast from the core to the coprocessor, subject to the same timing constraints as the queues. When the flush signal is received by the coprocessor, it causes the pipeline and the instruction queue to be cleared up to the instruction triggering the flush. This is explained in more detail in *Flush operations* on page 11-26.

11.3 Token queue management

The token queues, all of which are three slots long and function identically, are implemented as short FIFOs. An example implementation of the queues is described in:

- *Queue implementation*
- *Queue modification*
- *Queue flushing* on page 11-14.

11.3.1 Queue implementation

The queue FIFOs are implemented as three registers, with the current output selected by using multiplexors. Figure 11-4 shows this arrangement.

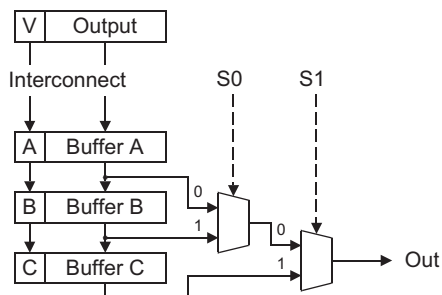


Figure 11-4 Token queue buffers

The queue consists of three registers, each of which is associated with a flag that indicates if the register contains valid data. New data are moved into the queue by being written into buffer A and continue to move along the queue if the next register is empty, or is about to become empty. If the queue is full, the oldest data, and therefore the first to be read from the queue, occupies buffer C and the newest occupies buffer A.

The multiplexors also select the current flag, which then indicates if the selected output is valid.

11.3.2 Queue modification

The queue is written to on each cycle. Buffer A accepts the data arriving at the interface, and the buffer A flag accepts the valid bit associated with the data. If the queue is not full, this results in no loss of data because the contents of buffer A are moved to buffer B during the same cycle.

If the queue is full, then the loading of buffer A is inhibited to prevent loss of data. In any case, no valid data is presented by the interface when the queue is full, so no data loss ensues.

The state of the three buffer flags is used to decide which buffer provides the queue output during each cycle. The output is always provided by the buffer containing the oldest data. This is buffer C if it is full, or buffer B or, if that is empty, buffer A.

A simple priority encoder, looking at the three flags, can supply the correct multiplexor select signals. The state of the three flags can also determine how data are moved from one buffer to another in the queue. Table 11-4 shows how the three flags are decoded.

Table 11-4 Addressing of queue buffers

Flag C	Flag B	Flag A	S1	S0	Remarks
0	0	0	X	X	Queue is empty
0	0	1	0	0	B = A
0	1	0	0	1	C = B
0	1	1	0	1	C = B, B = A
1	0	0	1	X	-
1	0	1	1	X	B = A
1	1	0	1	X	-
1	1	1	1	X	Queue is full. Input inhibited

New data can be moved into buffer A, provided the queue is not full, even if its flag is set, because the current contents of buffer A are moved to buffer B.

When the queue is read, the flag associated with the buffer providing the information must be cleared. This operation can be combined with an input operation so that the buffer is overwritten at the end of the cycle during which it provides the queue output. This can be implemented by using the read enable signal to mask the flag of the selected stage, making it available for input. Figure 11-5 on page 11-14 shows reading and writing a queue.

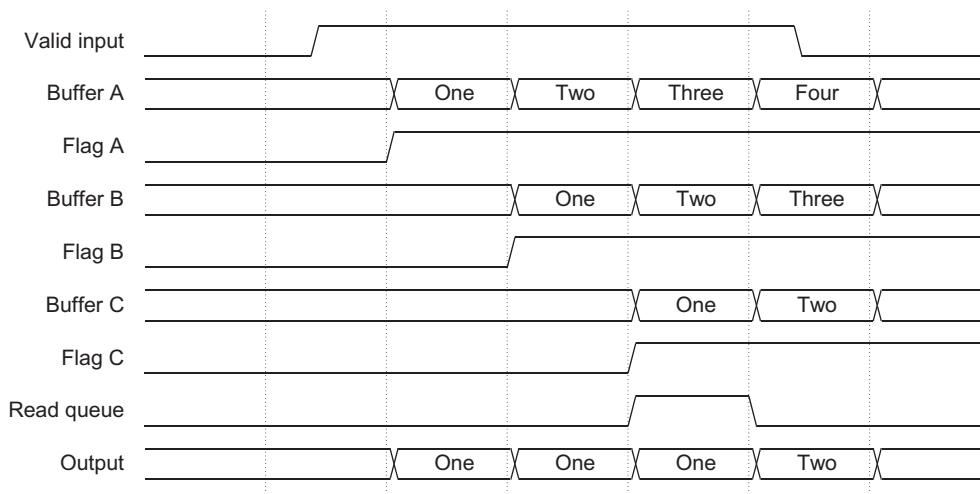


Figure 11-5 Queue reading and writing

Four valid inputs (labeled One, Two, Three, and Four) are written into the queue, and are clocked into buffer A as they arrive. Figure 11-5 shows how these inputs are clocked from buffer to buffer until the first input reaches buffer C. At this point a read from the queue is required. Because buffer C is full, it is chosen to supply the data. Because it is being read, it is free to accept more input, and so it receives the value Two from buffer B, which in turn receives the value Three from buffer A. Because buffer A is being emptied by writing to buffer B, it can accept the value Four from the input.

11.3.3 Queue flushing

When the coprocessor pipeline is flushed, in response to a command from the core, some of the queues might also need flushing. There are two possible ways of flushing the queue:

- the entire queue is cleared
- the queue is flushed from a selected buffer, along with all data in the queue newer than the data in the selected buffer.

The method used depends on the point at which flushing begins in the coprocessor pipeline. See *Flush operations* on page 11-26 for more details.

A flush command has associated with it a tag value that indicates where the queue flushing starts. This is matched with the tag carried by every instruction.

If the queue is to be flushed from a selected buffer, the buffer is chosen by looking for a matching tag. When this is found, the flag associated with that buffer is cleared, and every flag newer than the selected one is also cleared. Figure 11-6 shows queue flushing.

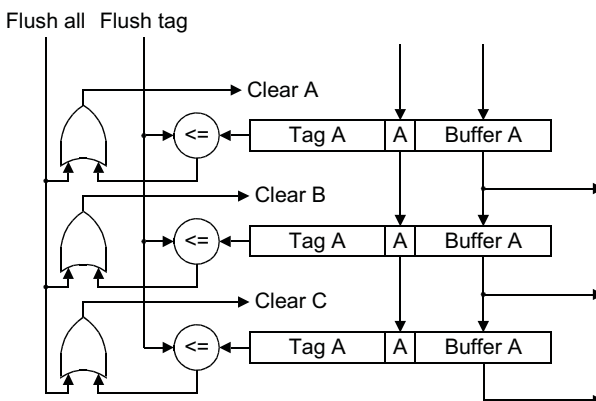


Figure 11-6 Queue flushing

Each buffer in the queue has a tag comparator associated with it. The flush tag is presented to each comparator, to be compared with the tag belonging to each valid instruction held in the queue. The flush tag is compared with each tag in the queue. If the flush tag is the same as, or older than, any tag then that queue entry has its Full flag cleared. This indicates that it is empty. A less-than-or-equal-to comparison is used to identify tags that are to be flushed. If a tag in the pipeline later than the queue matches, the Flush all signal is asserted to clear the entire queue.

11.4 Token queues

Each of the synchronizing queues is discussed in the following sections:

- *Instruction queue*
- *Length queue* on page 11-17
- *Accept queue* on page 11-18
- *Cancel queue* on page 11-18
- *Finish queue* on page 11-19.

11.4.1 Instruction queue

The core passes every instruction fetched from memory across the coprocessor interface, where it enters the instruction queue. Ideally it only passes on the coprocessor instructions, but has not, at this stage, had time to decode the instruction.

The coprocessor decodes the instruction on arrival in its own Decode stage and rejects the non-coprocessor instructions. The core does not require any acknowledgement of the removal of these instructions because each instruction type is determined within the coprocessors Decode stage. This means that the instruction received from the core must be decoded as soon as it enters the instruction queue. The instruction queue is a modified version of the standard queue, which incorporates an instruction decoder. Figure 11-7 shows an instruction queue implementation.

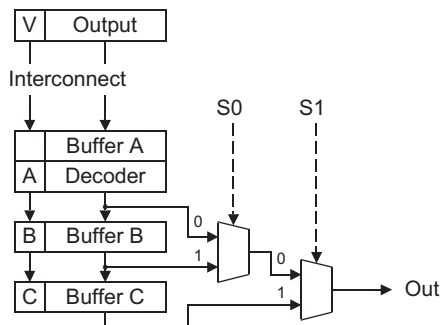


Figure 11-7 Instruction queue

The decoder decodes the instruction written into buffer A as soon as it arrives. The subsequent buffers, B and C, receive the decoded version of the instruction in buffer A.

The A flag now indicates that the data in buffer A are valid and represent a coprocessor instruction. This means that non-coprocessor or unrecognized instructions are immediately dropped from the instruction queue and are never passed on.

The coprocessor must also compare the coprocessor number field in a coprocessor instruction and compare it with its own number, given by **ACPNUM**. If the number does not match, the instruction is invalid.

The instruction queue provides an interface to the core through the following signals, which are all driven by the core:

- ACPINSTRV** This signal is asserted when valid data are available from the core. It must be clocked directly into the buffer A flag, unless the queue is full, in which case it is ignored.
- ACPINSTR[31:0]** This is the instruction being passed to the coprocessor from the core, and must be clocked into buffer A.
- ACPINSTRT[3:0]** This is the flush tag associated with the instruction in **ACPINSTR**, and must be clocked into the tag associated with buffer A.

The instruction queue feeds the issue stage of the coprocessor pipeline, providing a new input to the pipeline, in the form of a decoded instruction and its associated tag, whenever the queue is not empty.

11.4.2 Length queue

When a coprocessor has decoded an instruction it knows how long a vectored load/store operation is. This information is sent with the synchronizing token down the length queue, as the relevant instruction leaves the instruction queue to enter the issue stage of the pipeline. The length queue is maintained by the core and the coprocessor communicates with the queue using the following signals:

CPALENGTH[3:0]

This is the length of a vectored data transfer to or from the coprocessor. It is determined by the decoder in the instruction queue and asserted as the decoded instruction moves into the issue stage. If the current instruction does not represent a vectored data transfer, the length value is set to zero.

CPALENGTHT[3:0]

This is the tag associated with the instruction leaving the instruction queue, and is copied from the queue buffer supplying the instruction.

CPALENGTHHOLD

This is deasserted when the instruction queue is providing valid information to the core length queue. Otherwise, the signal is asserted to indicate that no valid data are available.

11.4.3 Accept queue

The coprocessor must decide in the issue stage if it can accept an otherwise valid coprocessor instruction. It passes this information with the synchronizing token down the accept queue, as the relevant instruction passes from the issue stage to Ex1.

If an instruction cannot be accepted by the coprocessor it is said to have been bounced. If the coprocessor bounces an instruction it does not remove the instruction from its pipeline, but converts it to a phantom. This is explained in more detail in *Bounce operations* on page 11-25.

The accept queue is maintained by the core and the coprocessor communicates with the queue using the following signals, which are all driven by the coprocessor:

CPAACCEPT

This is set to indicate that the instruction leaving the coprocessor issue stage has been accepted.

CPAACCEPTT[3:0]

This is the tag associated with the instruction leaving the issue stage.

CPAACCEPTHOLD

This is deasserted when the issue stage is passing an instruction on to the Ex1 stage, whether it has been accepted or not. Otherwise, the signal is asserted to indicate that no valid data are available.

11.4.4 Cancel queue

The core might want to cancel an instruction that it has already passed on to the coprocessor. This can happen if the instruction fails its condition codes, which requires the instruction to be removed from the instruction stream in both the core and the coprocessor.

The queue, which is a standard queue as described in *Token queue management* on page 11-12, is maintained by the coprocessor and is read by the coprocessor Ex1 stage.

The cancel queue provides an interface to the core through the following signals, which are all driven by the core:

ACPCANCELV

This signal is asserted when valid data are available from the core. It must be clocked directly into the buffer A flag, unless the queue is full, in which case it is ignored.

ACPCANCEL

This is the cancel command being passed to the coprocessor from the core, and must be clocked into buffer A.

ACPCANCELT[3:0]

This is the flush tag associated with the cancel command, and must be clocked into the tag associated with buffer A.

The cancel queue is read by the coprocessor Ex1 stage, which acts on the value of the queued **ACPCANCEL** signal by removing the instruction from the Ex1 stage if the signal is set, and not passing it on to the Ex2 stage.

11.4.5 Finish queue

The finish queue maintains synchronism at the end of the pipeline by providing permission for CDP instructions in the coprocessor pipeline to retire. The queue, which is a standard queue as described in *Token queue management* on page 11-12, is maintained by the coprocessor and is read by the coprocessor Ex6 stage.

The finish queue provides an interface to the core using the **ACPFINISHV** signal, which is driven by the core.

This signal is asserted to indicate that the instruction in the coprocessor Ex6 stage can retire. It must be clocked directly into the buffer A flag, unless the queue is full, in which case it is ignored.

The finish queue is read by the coprocessor Ex6 stage, which can retire a CDP instruction if the finish queue is not empty.

11.5 Data transfer

Data transfers are managed by the LSU on the core side, and the pipeline itself on the coprocessor side. Transfers can be a single value or a vector. In the latter case, the coprocessor effectively converts a multiple transfer into a series of single transfers by iterating the instruction in the issue stage. This creates an instance of the load/store instruction for each item to be transferred.

The instruction stays in the coprocessor issue stage while it iterates, creating copies of itself that move down the pipeline. Figure 11-9 on page 11-21 illustrates this process for a load instruction.

The first of the iterated instructions, shown in uppercase, is the head and the others (shown in lowercase) are the tails. In the example shown the vector length is four so there is one head and three tails. At the first iteration of the instruction, the tail flag is set so that subsequent iterations send tail instructions down the pipeline. In the example shown in Figure 11-9 on page 11-21, instruction B has stalled in the Ex1 stage (which might be caused by the cancel queue being empty), so that instruction C does not iterate during its first cycle in the issue stage, but only starts to iterate after the stall has been removed.

Figure 11-8 shows the extra paths required for passing data to and from the coprocessor.

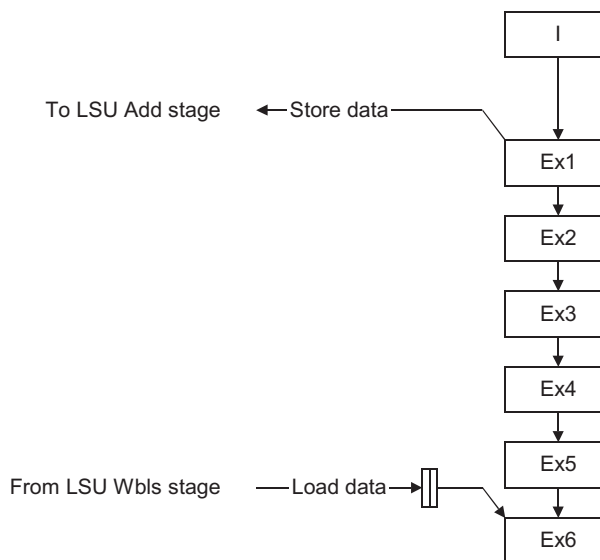


Figure 11-8 Coprocessor data transfer

Two data paths are required:

- One passes store data from the coprocessor to the core, and this requires a queue, which is maintained by the core.
- The other passes load data from the core to the coprocessor and requires no queue, only two pipeline registers.

Figure 11-9 shows instruction iteration for loads.

I	A	B	[C]	C	c	c	c	D						
Ex1		A	[B]	B	C	c	c	c	D					
Ex2			A		B	C	c	c	c	D				
Ex3				A		B	C	c	c	c	D			
Ex4					A		B	C	c	c	c	D		
Ex5						A		B	C	c	c	c	D	
Ex6							A		B	C	c	c	c	D
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 11-9 Instruction iteration for loads

Only the head instruction is involved in token exchange with the core pipeline, which does not iterate instructions in this way, the tail instructions passing down the pipeline silently.

When an iterated load/store instruction is cancelled or flushed, all the tail instructions (bearing the same tag) must be removed from the pipeline. Only the head instruction becomes a phantom when cancelled. Any tail instruction can be left intact in the pipeline because it has no further effect.

Because the cancel token is received in the coprocessor Ex1 stage, a cancelled iterated instruction always consists of a head instruction in Ex1 and a single tail instruction in the issue stage.

11.5.1 Loads

Load data emerge from the WBIs stage of the core LSU and are received by the coprocessor Ex6 stage. Each item in a vectored load is picked up by one instance of the iterated load instruction.

The pipeline timing is such that a load instruction is always ready, or just arrived, in Ex6 to pick up each data item. If a load instruction has arrived in Ex6, but the load information has not yet appeared, the load instruction must stall in Ex6, stalling the rest of the coprocessor pipeline.

The following signals are driven by the core to pass load data across to the coprocessor:

ACPLDVALID

This signal, when set, indicates that the associated data are valid.

ACPLDDATA[63:0]

This is the information passed from the core to the coprocessor.

Load buffers

To achieve correct alignment of the load data with the load instruction in the coprocessor Ex6 stage, the data must be double buffered when they arrive at the coprocessor. Figure 11-10 shows an example.

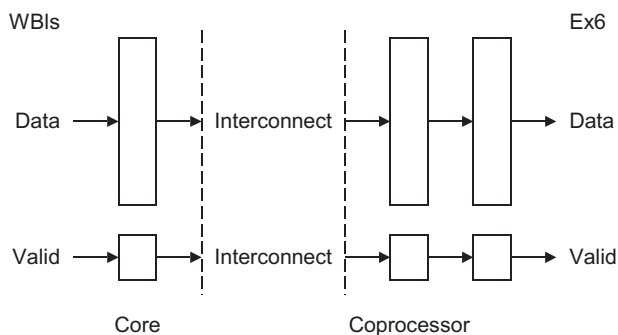


Figure 11-10 Load data buffering

The load data buffers function as pipeline registers and so require no flow control and do not need to carry any tags. Only the data and a valid bit are required. For load transfers to work:

- instructions must always arrive in the coprocessor Ex6 stage coincident with, or before, the arrival of the corresponding instruction in the core WBIs stage
- finish tokens from the core must arrive at the same time as the corresponding load data items arrive at the end of the load data pipeline buffers

- the LSU must see the token from the accept queue before it enables a load instruction to move on from its Add stage.

Loads and flushes

If a flush does not involve the core WBIs stage it cannot affect the load data buffers, and the load transfer completes normally. If a flush is initiated by an instruction in the core WBIs stage, this is not a load instruction because load instructions cannot trigger a flush. Any coprocessor load instructions behind the flush point find themselves stalled if they get as far as the Ex6 stage, for the lack of a finish token, so no data transfers can have taken place. Any data in the load data buffers expires naturally during the flush dead period while the pipeline reloads.

Loads and cancels

If a load instruction is canceled both the head and any tails must be removed. Because the cancellation happens in the coprocessor Ex1 stage, no data transfers can have taken place and therefore no special measures are required to deal with load data.

Loads and retirement

When a load instruction reaches the bottom of the coprocessor pipeline it must find a data item at the end of the load data buffer. This applies to both head and tail instructions. Load instructions do not use finish queue.

11.5.2 Stores

Store data emerge from the coprocessor issue stage and are received by the core LSU DC1 stage. Each item of a vectored store is generated because the store instruction iterates in the coprocessor issue stage. The iterated store instructions then pass down the pipeline but have no further use, except to act as place markers for flushes and cancels.

The following signals control the transfer of store data across the coprocessor interface:

CPASTDATAV

This signal is asserted when valid data is available from the coprocessor.

CPASTDATAT[3:0]

This is the tag associated with the data being passed to the core.

CPASTDATA[63:0]

This is the information passed from the coprocessor to the core.

ACPSTSTOP

This signal from the core prevents additional transfers from the coprocessor to the core, and is raised when the store queue, maintained by the core, can no longer accept any more data. When the signal is deasserted, data transfers can resume.

When **ACPSTSTOP** is asserted, the data previously placed onto **CPASTDATA** must be left there, until new data can be transferred. This enables the core to leave data on **CPASTDATA** until there is sufficient space in the store data queue.

Store data queue

Because the store data transfer can be stopped at any time by the LSU, a store data queue is required. Additionally, because store data vectors can be of arbitrary length, flow control is required. A queue length of three slots is sufficient to enable flow control to be used without loss of data.

Stores and flushes

When a store instruction is involved in a flush, the store data queue must be flushed by the core. Because the queue continues to fill for two cycles after the core notifies the coprocessor of the flush (because of the signal propagation delay) the core must delay for two cycles before carrying out the store data queue flush. The dead period after the flush extends sufficiently far to enable this to be done.

Stores and cancels

If the core cancels a store instruction, the coprocessor must ensure that it sends no store data for that instruction. It can achieve this by either:

- delaying the start of the store data until the corresponding cancel token has been received in the Ex1 stage
- looking ahead into the cancel queue and start the store data transfer when the correct token is seen.

Stores and retirement

Because store instructions do not use the finish token queue they are retired as soon as they leave the Ex1 stage of the pipeline.

11.6 Operations

This section describes the various operations that can be performed and events that can take place.

11.6.1 Normal operation

In normal operation the core passes all instructions across to the coprocessor, and then increments the tag if the instruction was a coprocessor instruction. The coprocessor decodes the instruction and throws it away if it is not a coprocessor instruction or if it contains the wrong coprocessor number.

Each coprocessor instruction then passes down the pipeline, sending a token down the length queue as it moves into the issue stage. The instruction then moves into the Ex1 stage, sending a token down the accept queue, and remains there until it has received a token from the cancel queue.

If the cancel token does not request that the instruction is cancelled, and is not a Store instruction, it moves on to the Ex2 stage. The instruction then moves down the pipeline until it reaches the Ex6 stage. At this point it waits to receive a token from the finish queue, which enables it to retire, unless it is either:

- a store instruction, in which case it requires no token from the finish queue
- a load instruction, in which case it must wait until load data are available.

Store instructions are removed from the pipeline as soon as they leave the Ex1 stage.

11.6.2 Cancel operations

When the coprocessor instruction reaches the Ex1 stage it looks for a token in the cancel queue. If the token indicates that the instruction is to be cancelled, it is removed from the pipeline and does not pass to Ex2. Any tail instruction in the I stage is also removed.

11.6.3 Bounce operations

The coprocessor can reject an instruction by bouncing it when it reaches the issue stage. This can happen to an instruction that has been accepted as a valid coprocessor instruction by the decoder, but that is found to be unexecutable by the issue stage, perhaps because it refers to a non-existent register or operation.

When the bounced instruction leaves the issue stage to move into Ex1, the token sent down the accept queue has its bounce bit set. This causes the instruction to be removed from the core pipeline.

When the instruction moves into Ex1 it has its dead bit set, turning it into a phantom. This enables the instruction to remain in the pipeline to match tokens in the cancel queue.

The core posts a token for the bounced instruction before the coprocessor can bounce it, so the phantom is required to pick up the token for the bounced instruction. The instruction is otherwise inert, and has no other effect.

The core might already have decided to cancel the instruction being bounced. In this case, the cancel token just causes the phantom to be removed from the pipeline. If the core does not cancel the phantom it continues to the bottom of the pipeline.

11.6.4 Flush operations

A flush can be triggered by the core in any stage from issue to WBIs inclusive. When this happens a broadcast signal is received by the coprocessor, passing it the tag associated with the instruction triggering the flush.

Because the tag is changed by the core after each new coprocessor instruction, the tag matches the first coprocessor instruction following the instruction causing the flush. The coprocessor must then find the first instruction that has a matching tag, working from the bottom of the pipeline upwards, and remove all instructions from that point upwards.

Unlike tokens passing down a queue, a flush signal has a fixed delay so that the timing relationship between a flush in the core and a flush in the coprocessor is known precisely.

Most of the token queues also need flushing and this can also be done using the tags attached to each instruction. If a match has been found before the stage at the receiving end of a token queue is passed, then the token queue is just cleared.

Otherwise, it must be properly flushed by matching the tags in the queue. This operation must be performed on all the queues except the finish queue, which is updated in the normal way. Therefore, the coprocessor must flush the instruction and cancel queues.

The flushing operation can be carried out by the coprocessor as soon as the flush signal is received. The flushing operation is simplified because the instruction and cancel queues cannot be performing any other operation. This means that flushing does not need to be combined with queue updates for these queues.

There is a single cycle following a flush in which nothing happens that affects the flushed queues, and this provides a good opportunity to carry out the queue flushing operation.

The following signals provide the flush broadcast signal from the core:

ACPFLUSH

This signal is asserted when a flush is to be performed.

ACPFLUSHT[3:0]

This is the tag associated with the first instruction to be flushed.

11.6.5 Retirement operations

When an instruction reaches the bottom of the coprocessor pipeline it is retired. How it retires depends on the kind of instruction it is and if it is iterated, as shown in Table 11-5.

Table 11-5 Retirement conditions

Instruction	Type	Retirement conditions
CDP	-	Must find a token in the finish queue.
MRC	Store	No conditions. Immediate retirement on leaving Ex1.
MCR	Load	All load instructions must find data in the load data pipeline from the core.
MRRC	Store	No conditions. Immediate retirement on leaving Ex1.
MCRR	Load	All load instructions must find data in the load data pipeline from the core.
STC	Store	No conditions. Immediate retirement on leaving Ex1.
LDC	Load	Must find data in the load data pipeline from the core.

Table 11-5 lists the conditions for each coprocessor instruction:

- all store instructions retire unconditionally on leaving Ex1 because no token is required in the finish queue
- CDP instructions require a token in the finish queue
- all load instructions must pick up data from the load pipeline
- phantom load instructions retire unconditionally.

11.7 Multiple coprocessors

There might be more than one coprocessor attached to the core, and so some means is required for dealing with multiple coprocessors. It is important, for reasons of economy, to ensure that as little of the coprocessor interface is duplicated. In particular, the coprocessors must share the length, accept, and store data queues, which are maintained by the core.

If these queues are to be shared, only one coprocessor can use the queues at any time. This is achieved by enabling only one coprocessor to be active at any time. This is not a serious limitation because only one coprocessor is in use at any time.

Typically, a processor is driven through driver software, which drives just one coprocessor. Calls to the driver software, and returns from it, ensure that there are several core instructions between the use of one coprocessor and the use of a different coprocessor.

11.7.1 Interconnect considerations

If only one coprocessor is allowed to communicate with the core at any time, all coprocessors can share the coprocessor interface signals from the core. Signals from the coprocessors to the core can be ORed together, provided that every coprocessor holds its outputs to zero when it is inactive.

11.7.2 Coprocessor selection

Coprocessors are enabled by a signal **ACPENABLE** from the core. There are 16 of these signals, one for each coprocessor. Only one can be active at any time. In addition, instructions to the coprocessor include the coprocessor number, enabling coprocessors to reject instructions that do not match their own number. Core instructions are also rejected.

11.7.3 Coprocessor switching

When the core decodes a coprocessor instruction destined for a different coprocessor to that last addressed, it stalls this instruction until the previous coprocessor instruction has been retired. This ensures that all activity in the currently selected coprocessor has ceased.

The coprocessor selection is switched, disabling the last active coprocessor and activating the new coprocessor. The coprocessor that should have received the new coprocessor instruction must have ignored it, being disabled. Therefore, the instruction is resent by the core, and is now accepted by the newly activated coprocessor.

A coprocessor is disabled by the core by setting **ACPENABLE LOW** for the selected coprocessor. The coprocessor responds by ceasing all activity and setting all its output signals **LOW**.

When the coprocessor is enabled, which is signaled by setting **ACPENABLE HIGH**, it must immediately set the signals **CPALENGTHHOLD** and **CPAACCEPHOLD HIGH**, and **CPASTDATAV LOW**, because the pipeline is empty at this point. The coprocessor can then start normal operation.

Chapter 12

Vectored Interrupt Controller Port

This chapter describes the ARM1136JF-S vectored interrupt controller port. It contains the following sections:

- *About the PL192 Vectored Interrupt Controller* on page 12-2
- *About the ARM1136JF-S VIC port* on page 12-3
- *Timing of the VIC port* on page 12-6
- *Interrupt entry flowchart* on page 12-9.

12.1 About the PL192 Vectored Interrupt Controller

An interrupt controller is a peripheral that is used to handle multiple interrupt sources. Features usually found in an interrupt controller are:

- multiple interrupt request inputs, one for each interrupt source, and one interrupt request output for the processor interrupt request input
- software can mask out particular interrupt requests
- prioritization of interrupt sources for interrupt nesting.

In a system with an interrupt controller having the above features, software is still required to:

- determine which interrupt source is requesting service
- determine where the service routine for that interrupt source is loaded.

A *Vectored Interrupt Controller (VIC)* does both things in hardware. It supplies the starting address (vector address) of the service routine corresponding to the highest priority requesting interrupt source.

The PL192 VIC is an *Advanced Microcontroller Bus Architecture (AMBA)* compliant, *System-on-Chip (SoC)* peripheral that is developed, tested, and licensed by ARM Limited for use in ARM1136JF-S designs.

The ARM1136JF-S VIC port and the Peripheral Interface enable you to connect a PL192 VIC to an ARM1136JF-S processor. See *ARM PrimeCell Vectored Interrupt Controller (PL192) Technical Reference Manual* for more details.

12.2 About the ARM1136JF-S VIC port

Figure 12-1 shows the VIC port and the Peripheral Interface connecting a PL192 VIC and an ARM1136JF-S processor.

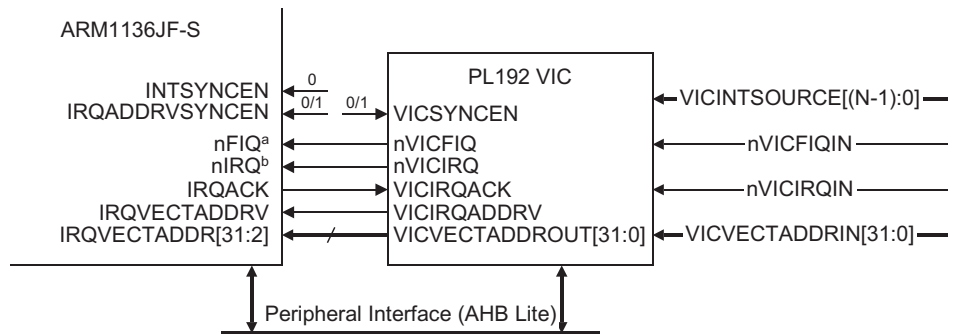


Figure 12-1 Connection of a PL192 VIC to an ARM1136JF-S processor

The VIC port enables the processor to read the vector address as part of the IRQ interrupt entry. That is, the ARM1136JF-S processor takes a vector address from this interface instead of using the legacy `0x00000018` or `0xFFFF0018`.

The VIC port does not support the reading of FIQ vector addresses.

The interrupt interface is designed to handle interrupts asserted by a controller that is clocked either synchronously or asynchronously to the ARM1136JF-S processor clock. This capability ensures that the controller can be used in systems that have either a synchronous or asynchronous interface between the core clock and the AHB clock.

The VIC port consists of the signals shown in Table 12-1.

Table 12-1 VIC port signals

Signal name	Direction	Description
nFIQ	Input	Active LOW fast interrupt request signal
nIRQ	Input	Active LOW normal interrupt request signal
INTSYNCEN	Input	If this signal is asserted, the internal nFIQ and nIRQ synchronizers are bypassed
IRQADDRVSYNCEN	Input	If this signal is asserted, the internal IRQADDRV synchronizer is bypassed

Table 12-1 VIC port signals (continued)

Signal name	Direction	Description
IRQACK	Output	Active HIGH IRQ acknowledge
IRQADDRV	Input	Active HIGH valid signal for the IRQ interrupt vector address below
IRQADDR[31:2]	Input	IRQ interrupt vector address. IRQADDR[31:2] holds the address of the first ARM state instruction in the IRQ handler

IRQACK is driven by the ARM1136JF-S processor to indicate to an external VIC that the processor wants to read the **IRQADDR** input.

IRQADDRV is driven by a VIC to tell the ARM1136JF-S processor that the address on the **IRQADDR** bus is valid and being held, and so it is safe for the processor to sample it.

IRQACK and **IRQADDRV** together implement a four-phase handshake between the ARM1136JF-S processor and a VIC. See *Timing of the VIC port* on page 12-6 for more details.

12.2.1 Synchronization of the VIC port signals

The peripheral port clock signal **HCLK** can run at any frequency, synchronously or asynchronously to the ARM1136JF-S processor clock signal, **CLKIN**. The ARM1136JF-S processor VIC port can cope with any clocking mode.

nFIQ and **nIRQ** can be connected to either synchronous or asynchronous sources. Synchronizers are provided internally for the case of asynchronous sources. Pins **INTSYNCEN** is also provided to enable SoC designers to bypass the synchronizers if required. Similarly, a synchronizer is provided inside the ARM1136JF-S processor for the **IRQADDRV** signal. If this signal is known to be synchronous, the synchronizer can be bypassed by pulling **IRQADDRVSYNCEN** HIGH.

These signals enable SoC designers to reduce interrupt latency if it is known that the **nFIQ**, **nIRQ**, or **IRQADDRV** input is always driven by a synchronous source.

When connecting the PL192 VIC to the ARM1136JF-S processor, **INTSYNCEN** must be tied LOW regardless of the Peripheral Port clocking mode. This is because the PL192 **nVICIRQ** and **nVICFIQ** outputs are completely asynchronous, because there are combinational paths that cross this device through to these outputs. However, **IRQADDRVSYNCEN** must be set depending on the clocking mode.

12.2.2 Interrupt handler exit

The software acknowledges an IRQ interrupt handler exit to a VIC by issuing a write to the vector address register.

12.3 Timing of the VIC port

Figure 12-2 shows a timing example of VIC port operation. In this example **IRQC** is received followed by **IRQB** having a higher priority. The waveforms in Figure 12-2 show an asynchronous relationship between **CLKIN** and **HCLK**, and the delays marked **Sync** cater for the delay of the synchronizers. When this interface is used synchronously, these delays are reduced to being a single cycle of the receiving clock.

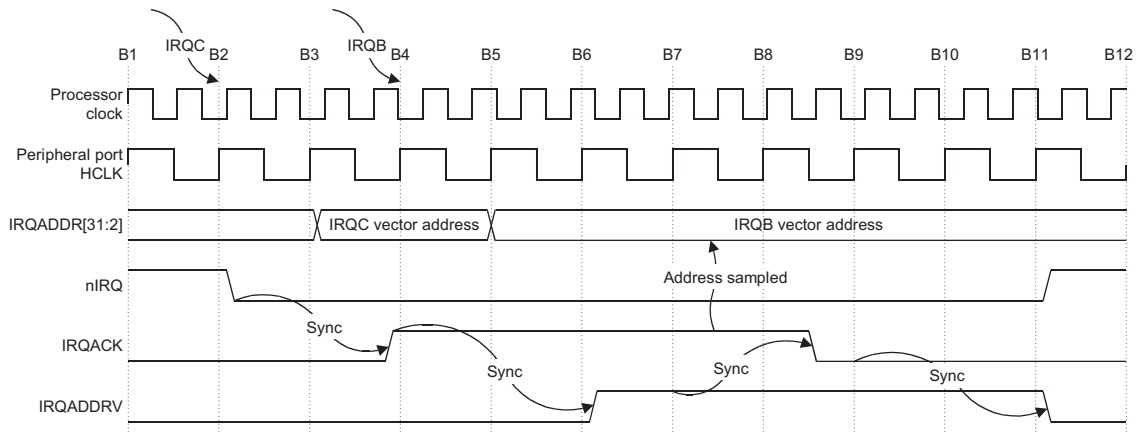


Figure 12-2 VIC port timing example

Figure 12-2 illustrates the basic handshake mechanism that operates between an ARM1136JF-S processor and a PL192 VIC:

1. An **IRQC** interrupt request occurs causing the PL192 VIC to set the processor **nIRQ** input.
2. The processor samples the **nIRQ** input LOW and initiates an interrupt entry sequence.
3. Another **IRQB** interrupt request of higher priority than **IRQC** occurs.
4. Between B3 and B4, the processor decides that the pending interrupt is an **IRQ** rather than a **FIQ** and asserts the **IRQACK** signal.
5. At B4 the VIC samples **IRQACK** HIGH and starts generating **IRQADDRV**. The VIC can still change **IRQADDR** to the **IRQB** vector address while **IRQADDRV** is LOW.

6. At B6 the VIC asserts **IRQADDRV** while **IRQADDR** is set to the IRQB vector address. **IRQADDR** is held until the processor acknowledges it has sampled it, even if a higher priority interrupt is received while the VIC is waiting.
7. Around B8 the processor samples the value of the **IRQADDR** input bus and deasserts **IRQACK**
8. When the VIC samples **IRQACK** LOW, it stacks the priority of the IRQB interrupt and deasserts **IRQADDRV**. It also deasserts **nIRQ** if there are no higher priority interrupts pending.
9. When the processor samples **IRQADDRV** LOW, it knows it can sample the **nIRQ** input again. Therefore, if the VIC requires some time for deasserting **nIRQ**, it must ensure that **IRQADDRV** stays HIGH until **nIRQ** has been deasserted.

The clearing of the interrupt is handled in software by the interrupt handling routine. This enables multiple interrupt sources to share a single interrupt priority. In addition, the interrupt handling routine must communicate to the VIC that the interrupt currently being handled is complete, using the memory-mapped or coprocessor-mapped interface, to enable the interrupt masking to be unwound.

12.3.1 PL192 VIC timing

As its part of the handshake mechanism, the PL192 VIC:

1. Synchronizes **IRQACK** on its way in if the peripheral port clocking mode is asynchronous or bypasses the synchronizers if it is in synchronous mode.
2. Asserts **IRQADDRV** when an address is ready at **IRQADDR**, and holds that address until **IRQACK** is sampled LOW, even if higher priority interrupts come along.
3. Stacks the priority that corresponds to the vector address present at **IRQADDR** when it samples the **IRQACK** signal LOW (while **IRQADDRV** is HIGH).
4. Clears **IRQADDRV** so the processor can recognize another interrupt. If **nIRQ** is also to be deasserted at this point because there are no higher priority interrupts pending, it is deasserted before or at the same time as **IRQADDRV** to ensure that the processor does not take the same interrupt again.

12.3.2 Core timing

As its part of the handshake mechanism, the core:

1. Starts an interrupt entry sequence when it samples the **nIRQ** signal asserted.

2. Determines if an FIQ or an IRQ is going to be taken. This happens after the interrupt entry sequence is started. If it decides that an IRQ is going to be taken, it starts the VIC port handshake by asserting **IRQACK**. If it decides that the interrupt is an FIQ, then it does not assert **IRQACK** and the VIC port handshake is not initiated.
3. Ignores the value of the **nFIQ** input until the IRQ interrupt entry sequence is completed if it has decided that the interrupt is an IRQ.
4. Samples the **IRQADDR** input bus when both **IRQACK** and **IRQADDRV** are sampled asserted. The interrupt entry sequence proceeds with this value of **IRQADDR**.
5. Ignores the **nIRQ** signal while **IRQADDRV** is HIGH. This gives the VIC time to deassert the **nIRQ** signal if there is no higher priority interrupt pending.
6. Ignores the **nFIQ** signal while **IRQADDRV** is HIGH.

12.4 Interrupt entry flowchart

Figure 12-3 is a flowchart for ARM1136JF-S interrupt recognition. It shows all the decisions and actions that have to be taken to complete interrupt entry.

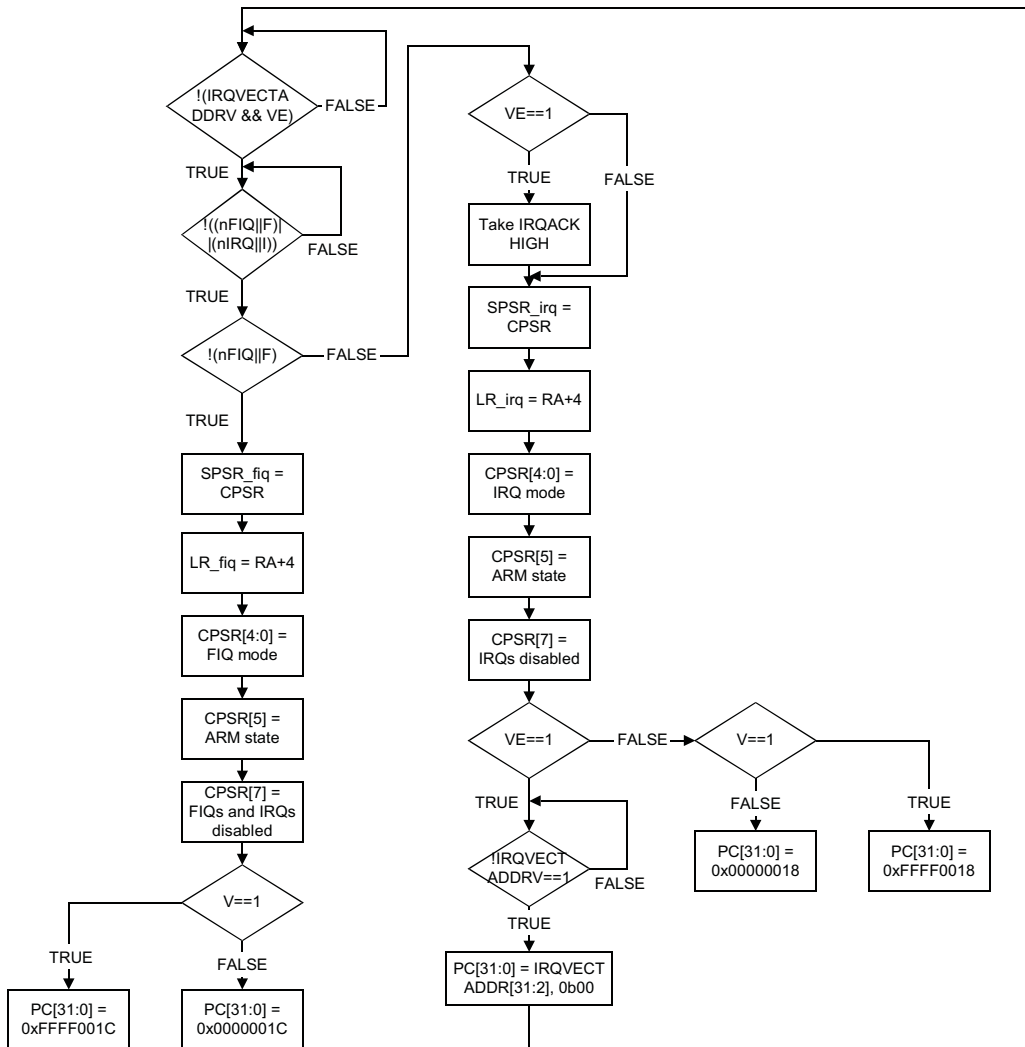


Figure 12-3 Interrupt entry sequence

Chapter 13

Debug

This chapter contains details of the ARM1136JF-S debug unit. These features assist the development of application software, operating systems, and hardware. This chapter contains the following sections:

- *Debug systems* on page 13-2
- *About the debug unit* on page 13-4
- *Debug registers* on page 13-7
- *CP14 registers reset* on page 13-24
- *CP14 debug instructions* on page 13-25
- *Debug events* on page 13-28
- *Debug exception* on page 13-32
- *Debug state* on page 13-34
- *Debug communications channel* on page 13-38
- *Debugging in a cached system* on page 13-39
- *Debugging in a system with TLBs* on page 13-40
- *Monitor mode debugging* on page 13-41
- *Halt mode debugging* on page 13-47
- *External signals* on page 13-49.

13.1 Debug systems

The ARM1136JF-S processor forms one component of a debug system that interfaces from the high-level debugging performed by you, to the low-level interface supported by the ARM1136JF-S processor. A typical system is shown in Figure 13-1.

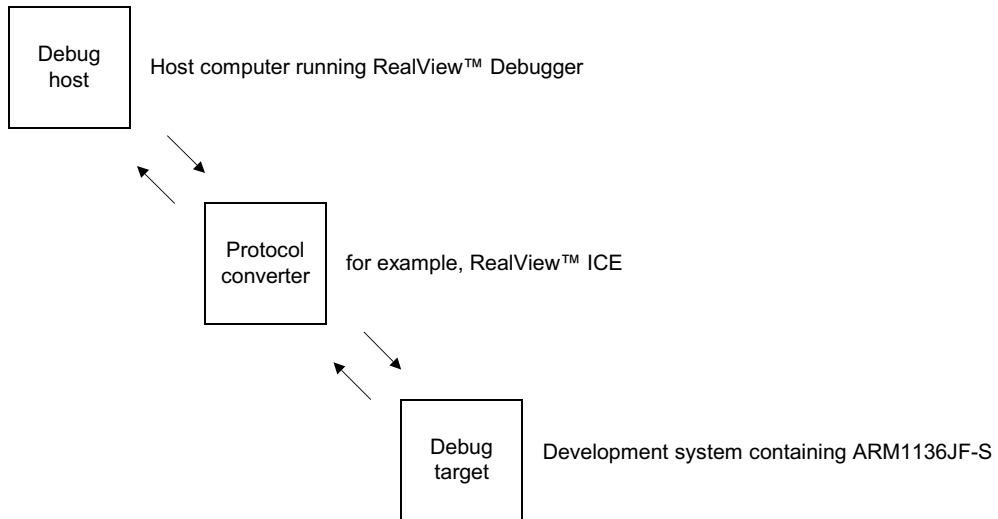


Figure 13-1 Typical debug system

This typical system has three parts:

- *The debug host*
- *The protocol converter* on page 13-3
- *The ARM1136JF-S processor* on page 13-3.

13.1.1 The debug host

The debug host is a computer, for example a personal computer, running a software debugger such as RealView™ Debugger. The debug host enables you to issue high-level commands such as *set breakpoint at location XX*, or *examine the contents of memory from 0x0-0x100*.

13.1.2 The protocol converter

The debug host is connected to the ARM1136JF-S development system using an interface, for example an RS232. The messages broadcast over this connection must be converted to the interface signals of the ARM1136JF-S processor. This function is performed by a protocol converter, for example, RealView ICE.

13.1.3 The ARM1136JF-S processor

The ARM1136JF-S processor, with debug unit, is the lowest level of the system. The debug extensions enable you to:

- stall program execution
- examine its internal state and the state of the memory system
- resume program execution.

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

13.2 About the debug unit

The ARM1136JF-S debug unit assists in debugging software running on the ARM1136JF-S processor. You can use an ARM1136JF-S debug unit, in combination with a software debugger program, to debug:

- application software
- operating systems
- ARM processor based hardware systems.

The debug unit enables you to:

- stop program execution
- examine and alter processor and coprocessor state
- examine and alter memory and input/output peripheral state
- restart the processor core.

you can debug the ARM1136JF-S processor in the following ways:

- *Halt mode debugging*
- *Monitor mode debugging* on page 13-5
- Trace debugging. See Chapter 15 *Trace Interface Port* for interfacing with an ETM.

The ARM1136JF-S debug interface is based on the *IEEE Standard Test Access Port and Boundary-Scan Architecture*.

13.2.1 Halt mode debugging

When the ARM1136JF-S debug unit is in Halt mode, the processor halts when a debug event, such as a breakpoint, occurs. When the core is halted, an external host can examine and modify its state using the DBGTAP.

In Halt mode you can examine and alter all processor state (processor registers), coprocessor state, memory, and input/output locations through the DBGTAP. This mode is intentionally invasive to program execution. Halt mode requires :

- external hardware to control the DBGTAP
- a software debugger to provide the user interface to the debug hardware.

See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10 to learn how to set the ARM1136JF-S debug unit into Halt mode.

13.2.2 Monitor mode debugging

When the ARM1136JS-S debug unit is in Monitor mode, the processor takes a Debug exception instead of halting. A special piece of software, a monitor target, can then take control to examine or alter the processor state. Monitor mode is essential in real-time systems where the core cannot be halted to collect information. For example, engine controllers and servo mechanisms in hard drive controllers that cannot stop the code without physically damaging the components.

When debugging in Monitor mode the processor stops execution of the current program and starts execution of a monitor target. The state of the processor is preserved in the same manner as all ARM exceptions (see the *ARM Architecture Reference Manual* on exceptions and exception priorities). The monitor target communicates with the debugger to access processor and coprocessor state, and to access memory contents and input/output peripherals. Monitor mode requires a debug monitor program to interface between the debug hardware and the software debugger.

When debugging in Monitor mode, you can program new debug events through CP14. This coprocessor is the software interface of all the debug resources such as the breakpoint and watchpoint registers. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10 to learn how to set the ARM1136JS-S debug unit into Monitor mode.

13.2.3 Virtual addresses and debug

Unless otherwise stated, all addresses in this chapter are *Virtual Addresses (VA)* as described in the *ARM Architecture Reference Manual*. For example, the *Breakpoint Value Registers (BVR)* and *Watchpoint Value Registers (WVR)* must be programmed with VAs.

The terms *Instruction Virtual Address (IVA)* and *Data Virtual Address (DVA)*, where used, mean the VA corresponding to an instruction address and the VA corresponding to a data address respectively.

13.2.4 Programming the debug unit

The ARM1136JF-S debug unit is programmed using *CoProcessor 14 (CP14)*. CP14 provides:

- instruction address comparators for triggering breakpoints
- data address comparators for triggering watchpoints
- a bidirectional *Debug Communication Channel (DCC)*
- all other state information associated with ARM1136JF-S debug.

CP14 is accessed using coprocessor instructions in Monitor mode, and certain debug scan chains in Halt mode, see Chapter 14 *Debug Test Access Port* to learn how to access the ARM1136JF-S debug unit using scan chains.

13.3 Debug registers

Table 13-1 shows definitions of terms used in register descriptions.

Table 13-1 Terms used in register descriptions

Term	Description
R	Read-only. Written values are ignored. However, it is written as 0 or preserved by writing the same value previously read from the same fields on the same processor.
W	Write-only. This bit cannot be read. Reads return an Unpredictable value.
RW	Read or write.
C	Cleared on read. This bit is cleared whenever the register is read.
UNP/SBZP	Unpredictable or <i>Should Be Zero or Preserved</i> (SBZP). A read to this bit returns an Unpredictable value. It is written as 0 or preserved by writing the same value previously read from the same fields on the same processor. These bits are usually reserved for future expansion.
Core view	This column defines the core access permission for a given bit.
External view	This column defines the DBGTAP debugger view of a given bit.
Read/write attributes	This is used when the core and the DBGTAP debugger view are the same.

On a power-on reset, all the CP14 debug registers take the values indicated by the Reset value column in the register bit field definition tables (Table 13-4 on page 13-11, Table 13-6 on page 13-15, Table 13-9 on page 13-18, Table 13-11 on page 13-21, and Table 13-12 on page 13-22). In these tables, - means an Undefined reset value.

13.3.1 Accessing debug registers

To access the CP14 debug registers you must set Opcode_1 and CRn to 0. The Opcode_2 and CRm fields of the coprocessor instructions are used to encode the CP14 debug register number, where the register number is {<Opcode2>, <CRm>}.

Table 13-2 on page 13-8 shows the CP14 debug register map. All of these registers are also accessible as scan chains from the DBGTAP.

Table 13-2 CP14 debug register map

Binary address		Register number	CP14 debug register name	Abbreviation
Opcode_2	CRm			
b000	b0000	c0	Debug ID Register	DIDR
b000	b0001	c1	Debug Status and Control Register	DSCR
b000	bb0010 b0100	c2-c4	Reserved	-
b000	b0101	c5	Data Transfer Register	DTR
b000	b0110	c6	Reserved	-
b000	b0111	c7	Vector Catch Register	VCR
b000	b1000-b1111	c8-c15	Reserved	-
b001-b011	b0000-b1111	c16-c63	Reserved	-
b100	b0000-b0101	c64-c69	Breakpoint Value Registers	BVRy ^a
	b0110-b1111	c70-c79	Reserved	-
b101	b0000-b0101	c80-c85	Breakpoint Control Registers	BCRy ^a
	b0110-b1111	c86-c95	Reserved	-
b110	b0000-b0001	c96-c97	Watchpoint Value Registers	WVRy ^a
	b0010-b1111	c98-c111	Reserved	-
b111	b0000-b0001	c112-c113	Watchpoint Control Registers	WCRy ^a
	b0010-b1111	c114-c127	Reserved	-

a. y is the decimal representation for the binary number CRm.

———— **Note** —————

All the debug resources required for Monitor mode debugging are accessible through CP14 registers. For Halt mode debugging some additional resources are required. See Chapter 14 *Debug Test Access Port*.

13.3.2 CP14 c0, Debug ID Register (DIDR)

The Debug ID Register is a read-only register that defines the configuration of debug registers in a system. The format of the Debug ID Register is shown in Figure 13-2.

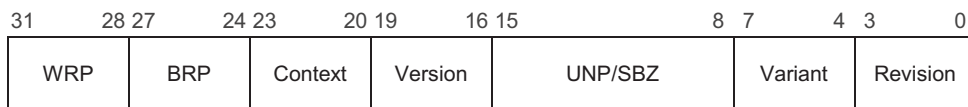


Figure 13-2 Debug ID Register format

The ARM1136JF-S r0p0 processor has 0x1511xx00 in this register.

The bit field definitions for the Debug ID Register are shown in Table 13-3.

Table 13-3 Debug ID Register bit field definition

Bits	Read/write attributes	Description
[31:28] WRP	R	Number of Watchpoint Register Pairs: b0000 = 1 WRP b0001 = 2 WRPs ... b1111 = 16 WRPs. For the ARM1136JF-S processor these bits are b0001 (2 WRPs).
[27: 24] BRP	R	Number of Breakpoint Register Pairs: b0000 = Reserved. The minimum number of BRPs is 2. b0001 = 2 BRPs b0010 = 3 BRPs ... b1111 = 16 BRPs. For the ARM1136JF-S processor these bits are b0101 (6 BRPs).
[23: 20] Context	R	Number of Breakpoint Register Pairs with context ID comparison capability: b0000 = 1 BRP has context ID comparison capability b0001 = 2 BRPs have context ID comparison capability ... b1111 = 16 BRPs have context ID comparison capability. For the ARM1136JF-S processor these bits are b0001 (2 BRPs).

Table 13-3 Debug ID Register bit field definition (continued)

Bits	Read/write attributes	Description
[19:16] Version	R	Debug architecture version.
[15:8]	UNP/SBZP	Reserved.
[7: 4] Variant	R	Implementation-defined variant number. This number is incremented on functional changes.
[3: 0] Revision	R	Implementation-defined revision number. This number is incremented on bug fixes.

The values of the following fields of the Debug ID Register agree with the values in CP15 c0, ID Register:

- D IDR[3:0] is the same as CP15 c0 bits [3:0]
- D IDR[7:4] is the same as CP15 c0 bits [23:20].

See *ID Code Register* on page 3-102 for a description of CP15 c0, ID Register.

The reason for duplicating these fields here is that the Debug ID Register is accessible through scan chain 0. This enables an external debugger to determine the variant and revision numbers without stopping the core.

13.3.3 CP14 c1, Debug Status and Control Register (DSCR)

The Debug Status And Control Register contains status and configuration information about the state of the debug system. The format of the Debug Status And Control Register is shown in Figure 13-3 on page 13-11.

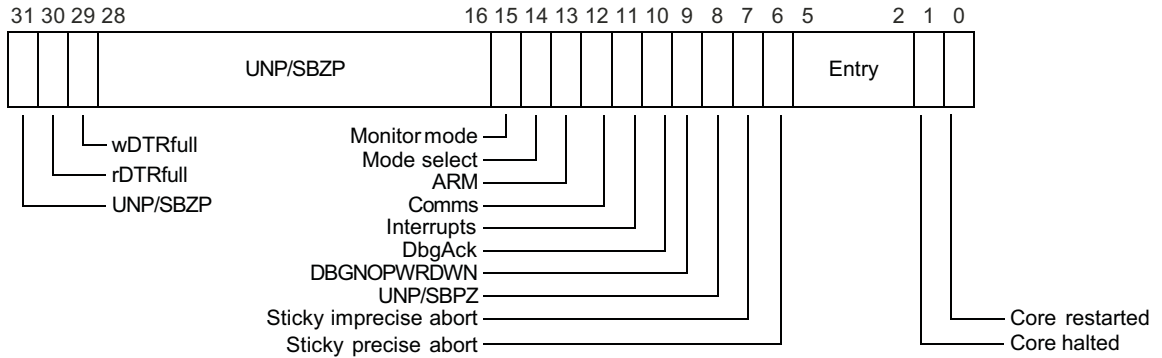


Figure 13-3 Debug Status And Control Register format

The bit field definitions for the Debug Status And Control Register are shown in Table 13-4.

Table 13-4 Debug Status And Control Register bit field definitions

Bits	Core view	External view	Reset value	Description
[31]	UNP/SBZP	UNP/SBZP	-	Reserved.
[30]	R	R	0	The rDTRfull flag: 0 = rDTR empty 1 = rDTR full. This flag is automatically set on writes by the DBGTAP debugger to the rDTR and is cleared on reads by the core of the same register. No writes to the rDTR are enabled if the rDTRfull flag is set.
[29]	R	R	0	The wDTRfull flag: 0 = wDTR empty 1 = wDTR full. This flag is automatically cleared on reads by the DBGTAP debugger of the wDTR and is set on writes by the core to the same register.
[28:16]	UNP/SBZP	UNP/SBZP	-	Reserved.
[15]	RW	R	0	The Monitor mode enable bit: 0 = Monitor mode disabled 1 = Monitor mode enabled. For the core to take a debug exception, Monitor mode has to be both selected and enabled (bit 14 clear and bit 15 set).

Table 13-4 Debug Status And Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[14]	R	RW	0	Mode select bit: 0 = Monitor mode selected 1 = Halt mode selected and enabled.
[13]	R	RW	0	Execute ARM instruction enable bit: 0 = Disabled 1 = Enabled. If this bit is set, the core can be forced to execute ARM instructions in debug state using the Debug Test Access Port. If this bit is set when the core is not in debug state, the behavior of the ARM1136JF-S processor is Unpredictable.
[12]	RW	R	0	User mode access to comms channel control bit: 0 = User mode access to comms channel enabled 1 = User mode access to comms channel disabled. If this bit is set and a User mode process tries to access the DIDR, DSCR, or the DTR, the Undefined instruction exception is taken. Because accessing the rest of CP14 debug registers is never possible in User page mode (see <i>Executing CP14 debug instructions</i> on page 13-26, setting this bit means that a User mode process cannot access any CP14 debug register.
[11]	R	RW	0	Interrupts bit: 0 = Interrupts enabled 1 = Interrupts disabled. If this bit is set, the IRQ and FIQ input signals are inhibited. ^a
[10]	R	RW	0	DbgAck bit. If this bit is set, the DBGACK output signal (see <i>External signals</i> on page 13-49) is forced HIGH, regardless of the processor state. ^a
[9]	R	RW	0	Powerdown disable: 0 = DBGNOPWRDWN is LOW 1 = DBGNOPWRDWN is HIGH. See <i>External signals</i> on page 13-49.
[8]	UNP/SBZP	UNP/SBZP	-	Reserved.
[7]	R	RC	0	Sticky imprecise Data Aborts bit: 0 = No imprecise Data Aborts occurred since the last time this bit was cleared 1 = An imprecise Data Abort has occurred since the last time this bit was cleared. It is cleared on reads of a DBGTAP debugger to the DSCR.

Table 13-4 Debug Status And Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[6]	R	RC	0	<p>Sticky precise Data Abort bit:</p> <p>0 = No precise Data Abort occurred since the last time this bit was cleared</p> <p>1 = An precise Data Abort has occurred since the last time this bit was cleared.</p> <p>This flag is meant to detect Data Aborts generated by instructions issued to the processor using the Debug Test Access Port. Therefore, if the DSCR[13] execute ARM instruction enable bit is a 0, the value of the sticky precise Data Abort bit is Unpredictable. It is cleared on reads of a DBGTAP debugger to the DSCR.</p>
[5:2]	RW	R	b0000	<p>Method of entry bits:</p> <p>b0000 = a Halt DBGTAP instruction occurred</p> <p>b0001 = a breakpoint occurred</p> <p>b0010 = a watchpoint occurred</p> <p>b0011 = a BKPT instruction occurred</p> <p>b0100 = an EDBGRQ signal activation occurred</p> <p>b0101 = a vector catch occurred</p> <p>b0110 = a data-side abort occurred</p> <p>b0111 = an instruction-side abort occurred</p> <p>b1xxx = reserved.</p>
[1]	R	R	1	<p>Core restarted bit:</p> <p>0 = the processor is exiting debug state</p> <p>1 = the processor has exited debug state.</p> <p>The DBGTAP debugger can poll this bit to determine when the processor has exited debug state. See <i>Debug state</i> on page 13-34 for a definition of debug state.</p>
[0]	R	R	0	<p>Core halted bit:</p> <p>0 = the processor is in normal state</p> <p>1 = the processor is in debug state.</p> <p>The DBGTAP debugger can poll this bit to determine when the processor has entered debug state. See <i>Debug state</i> on page 13-34 for a definition of debug state.</p>

- a. Bits DSCR[11:10] can be controlled by a DBGTAP debugger to execute code in normal state as part of the debugging process. For example, if the DBGTAP debugger has to execute an OS service to bring a page from disk into memory, and then return to the application to see the effect this change of state produces, it is undesirable that interrupts are serviced during execution of this routine.

Bits [5:2] are set to indicate:

- the reason for jumping to the Prefetch or Data Abort vector
- the reason for entering debug state.

Using bits [5:2], a Prefetch Abort or a Data Abort handler determines if it must jump to the monitor target. Additionally, a DBGTAP debugger or monitor target can determine the specific debug event that caused the debug state or debug exception entry.

13.3.4 CP14 c5, Data Transfer Registers (DTR)

This register consists of two separate physical registers:

- the rDTR (Read Data Transfer Register)
- the wDTR (Write Data Transfer Register).

The register accessed is dependent on the instruction used:

- writes, MCR and LDC instructions, access the wDTR
- reads, MRC and STC instructions, access the rDTR.

———— **Note** ————

Read and write refer to the core view.

For details of the use of these registers with the rDTRfull flag and wDTRfull flag see *Debug communications channel* on page 13-38. The format of both the rDTR and wDTR is shown in Figure 13-4.



Figure 13-4 DTR format

The bit field definitions for rDTR and wDTR are shown in Table 13-5.

Table 13-5 Data Transfer Register bit field definitions

Bits	Core view	External view	Description
[31:0]	R	W	Read data transfer register (read-only)
[31:0]	W	R	Write data transfer register (write-only)

13.3.5 CP14 c7, Vector Catch Register (VCR)

The ARM1136JF-S processor supports efficient exception vector catching. This is controlled by the VCR, as shown in Figure 13-5.

7	6	5	4	3	2	1	0
FIQ	IRQ	Reserved	Data Abort	Prefetch Abort	SWI	Undefined	Reset

Figure 13-5 Vector Catch Register format

If one of the bits in this register is set and the corresponding vector is committed for execution, then a Debug exception or debug state entry might be generated, depending on the value of the DSCR[15:14] bits (see *Behavior of the processor on debug events* on page 13-29). Under this model, any kind of fetch of an exception vector can trigger a vector catch, not just the ones due to exception entries.

The update of the VCR might occur several instruction after the corresponding MCR instruction. It only takes effect by the next *Instruction Memory Barrier (IMB)*.

Bits [31:8] and bit 5 are reserved.

The bit field definitions for the Vector Catch Register are shown in Table 13-6.

Table 13-6 Vector Catch Register bit field definitions

Bits	Read/write attributes	Reset value	Description	Normal address	High vector address
[31:8]	UNP/SBZP	-	Reserved	-	-
[7]	RW	0	Vector catch enable, FIQ	0x0000001C	0xFFFF001C
[6]	RW	0	Vector catch enable, IRQ	Most recent ^a IRQ address	Most recent ^a IRQ address
[5]	UNP/SBZP	-	Reserved	-	-
[4]	RW	0	Vector catch enable, Data Abort	0x00000010	0xFFFF0010
[3]	RW	0	Vector catch enable, Prefetch Abort	0x0000000C	0xFFFF000C
[2]	RW	0	Vector catch enable, SWI	0x00000008	0xFFFF0008
[1]	RW	0	Vector catch enable, Undefined Instruction	0x00000004	0xFFFF0004
[0]	RW	0	Vector catch enable, Reset	0x00000000	0xFFFF0000

- a. You can configure the ARM1136JF-S processor so that the IRQ uses vector exceptions other than 0x00000008 and 0xFFFF0008. See *Changes to existing interrupt vectors* on page 2-23 for more details.

13.3.6 CP14 c64-c69, Breakpoint Value Registers (BVR)

Each BVR is associated with a BCR register. BCR_y is the corresponding control register for BVR_y.

A pair of breakpoint registers, BVR_y/BCR_y, is called a *Breakpoint Register Pair* (BRP). BVR0-5 are paired with BCR0-5 to make BRP0-5.

The BVR of a BRP is loaded with an IVA and then its contents can be compared against the IVA bus of the processor.

The breakpoint value contained in the BVR corresponds to either an IVA or a context ID. Breakpoints can be set on:

- an IVA
- a context ID
- an IVA/context ID pair.

The ARM1136JF-S processor supports thread-aware breakpoints and watchpoints. A context ID can be loaded into the BVR and the BCR can be configured so this BVR value is compared against the CP15 context ID register, c13, instead of the IVA bus. Another register pair loaded with an IVA or DVA can then be linked with the context ID holding BRP. A breakpoint or watchpoint debug event is only generated if both the address and the context ID match at the same time. This means that unnecessary hits can be avoided when debugging a specific thread within a task.

Breakpoint debug events generated on context ID matches only are also supported. However, if the match occurs while the processor is running in a privileged mode and the debug logic in Monitor mode, it is ignored. This is to avoid the processor ending in an unrecoverable state.

The ARM1136JF-S processor implements the breakpoint and watchpoint registers shown in Table 13-7.

Table 13-7 ARM1136JF-S breakpoint and watchpoint registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b100	b0000-b0011	c64-c67	Breakpoint Value Registers 0-3	BVR0-3	No
	b0100-b0101	c68-c69	Breakpoint Value Registers 4-5	BVR4-5	Yes
	b0110-b1111	c70-c79	Reserved	-	-
b101	b0000-b0011	c80-c83	Breakpoint Control Registers 0-3	BCR0-3	No
	b0100-b0101	c84-c85	Breakpoint Control Registers 4-5	BCR4-5	Yes
	b0110-b1111	c86-c95	Reserved	-	-
b110	b0000-b0001	c96-c97	Watchpoint Value Registers 0-1	WVR0-1	-
	b0010-b1111	c98-c111	Reserved	-	-
b111	b0000-b0001	c112-c113	Watchpoint Control Registers 0-1	WCR0-1	-
	b0010-b1111	c114-c127	Reserved	-	-

The bit field definitions for context ID and non context ID Breakpoint Value Registers are shown in Table 13-8.

Table 13-8 Breakpoint Value Registers, bit field definition

Context ID capable?	Bits	Read/write attributes	Description
No	[31:2]	RW	Breakpoint address
Yes	[31:0]	RW	Breakpoint address

When a context ID capable BRP is set for IVA comparison, BVR bits [1:0] are ignored.

13.3.7 CP14 c80-c85, Breakpoint Control Registers (BCR)

These registers contain the necessary control bits for setting:

- breakpoints
- linked breakpoints.

The format of the Breakpoint Control Registers is shown in Figure 13-6.

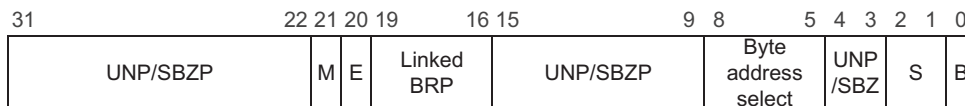


Figure 13-6 Breakpoint Control Registers, format

Bit field definitions for the Breakpoint Control Registers are shown in Table 13-9.

Table 13-9 Breakpoint Control Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:22]	UNP/SBZP	-	Reserved.
[21]	RW (Read as 0)	- (-)	Meaning of BVR: 0 = Instruction Virtual Address. The corresponding BVR is compared against the IVA bus. 1 = Context ID. The corresponding BVR is compared against the CP15 context ID (register 13). If this BRP does not have context ID comparison capability, this control bit does not apply and the corresponding bit is read as 0. See Table 13-10 on page 13-20 for details.
[20]	RW	-	Enable linking: 0 = Linking disabled 1 = Linking enabled. When this bit is set HIGH, the corresponding BRP is linked. See Table 13-10 on page 13-20 for details.
[19:16]	RW	-	Linked BRP number. The binary number encoded here indicates another BRP to link this one with. If a BRP is linked with itself, it is Unpredictable if a breakpoint debug event is generated.
[15:9]	UNP/SBZP	-	Reserved.

Table 13-9 Breakpoint Control Registers, bit field definitions (continued)

Bits	Read/write attributes	Reset value	Description
[8:5]	RW	-	<p>Byte address select. The BVR is programmed with a word address. You can use this field to program the breakpoint so it hits only if certain byte addresses are accessed.</p> <p>b0000 = The breakpoint never hits</p> <p>bxxx1= If the byte at address BVR[31:2]+0 is accessed, the breakpoint hits</p> <p>bxx1x = If the byte at address BVR[31:2]+1 is accessed, the breakpoint hits</p> <p>bx1xx = If the byte at address BVR[31:2]+2 is accessed, the breakpoint hits</p> <p>b1xxx = If the byte at address BVR[31:2]+3 is accessed, the breakpoint hits.</p> <p>This field must be set to b1111 when this BRP is programmed for context ID comparison, that is BCR[21:20] set to b1x. Otherwise breakpoint or watchpoint debug events might not be generated as expected.</p> <p style="text-align: center;">Note</p> <p>These are little-endian byte addresses. This ensures that a breakpoint is triggered regardless of the endianness of the instruction fetch.</p> <p>For example, if a breakpoint is set on a certain Thumb instruction by doing BCR[8:5] = b0011, it is triggered if in little-endian and IVA[1:0] is b00 or if big-endian and IVA[1:0] is b10.</p>
[4:3]	UNP/SBZP	-	Reserved
[2:1]	RW	-	<p>Supervisor Access. The breakpoint can be conditioned to the privilege of the access being done:</p> <p>b00 = Reserved</p> <p>b01= Privileged</p> <p>b10 = User</p> <p>b11 = Either.</p> <p>If this BRP is programmed for context ID comparison and linking (BCR[21:20] is set b11), then the BCR[2:1] field of the IVA-holding BRP takes precedence and it is Undefined whether this field is included in the comparison or not.</p> <p>Therefore, it must be set to either.</p> <p>The WCR[2:1] field of a WRP linked with this BRP also takes precedence over this field.</p>
[0]	RW	0	<p>Breakpoint enable:</p> <p>0 = Breakpoint disabled</p> <p>1 = Breakpoint enabled.</p>

Table 13-10 summarizes the meaning of BCR bits [21:20].

Table 13-10 Meaning of BCR[21:20] bits

BCR[21:20]	Meaning
b00	The corresponding BVR is compared against the IVA bus. This BRP is not linked with any other one. It generates a breakpoint debug event on an IVA match.
b01	The corresponding BVR is compared against the IVA bus. This BRP is linked with the one indicated by BCR[19:16] linked BRP field. They generate a breakpoint debug event on a joint IVA and context ID match.
b10	The corresponding BVR is compared against CP15 Context Id Register, c13. This BRP is not linked with any other one. It generates a breakpoint debug event on a context ID match.
b11	The corresponding BVR is compared against CP15 Context Id Register, c13. Another BRP (of the BCR[21:20]=b01 type), or WRP (with WCR[20]=b1), is linked with this BRP. They generate a breakpoint or watchpoint debug event on a joint IVA or DVA and context ID match.

———— **Note** —————

The BCR[8:5] and BCR[2:1] fields still apply when a BRP is set for context ID comparison. See *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-41 for detailed programming sequences for linked breakpoints and linked watchpoints.

The following rules apply to the ARM1136JF-S processor for breakpoint debug event generation:

- The update of a BVR or a BCR can take effect several instructions after the corresponding MCR. It takes effect by the next IMB.
- Updates of the CP15 Context ID Register c13, can take effect several instructions after the corresponding MCR. However, the write takes place by the end of the exception return. This is to ensure that a User mode process, switched in by a processor scheduler, can break at its first instruction.
- Any BRP (holding an IVA) can be linked with any other one with context ID capability. Several BRPs (holding IVAs) can be linked with the same context ID capable one.

- If a BRP (holding an IVA) is linked with one that is not configured for context ID comparison and linking, it is Unpredictable whether a breakpoint debug event is generated or not. BCR[21:20] fields of the second BRP must be set to b11.
- If a BRP (holding an IVA) is linked with one that is not implemented, it is Unpredictable if a breakpoint debug event is generated or not.
- If a BRP is linked with itself, it is Unpredictable if a breakpoint debug event is generated or not.
- If a BRP (holding an IVA) is linked with another BRP (holding a context ID value), and they are not both enabled (both BCR[0] bits set), the first one does not generate any breakpoint debug event.

13.3.8 CP14 c96-c97, Watchpoint Value Registers (WVR)

Each WVR is associated with a WCR register. WCR_y is the corresponding register for WVR_y.

A pair of watchpoint registers, WVR_y and WCR_y, is called a *Watchpoint Register Pair* (WRP). WVR0-1 are paired with WCR0-1 to make WRP0-1.

The watchpoint value contained in the WVR always corresponds to a DVA. Watchpoints can be set on:

- a DVA
- a DVA/context ID pair.

For the second case a WRP and a BRP with context ID comparison capability have to be linked. A debug event is generated when both the DVA and the context ID pair match simultaneously. Table 13-11 shows the bit field definitions for the Watchpoint Value Registers.

Table 13-11 Watchpoint Value Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:2]	RW	-	Watchpoint address

13.3.9 CP14 c112-c113, Watchpoint Control Registers (WCR)

These registers contain the necessary control bits for setting:

- watchpoints
- linked watchpoints.

The format of the Watchpoint Control Registers is shown in Figure 13-7.

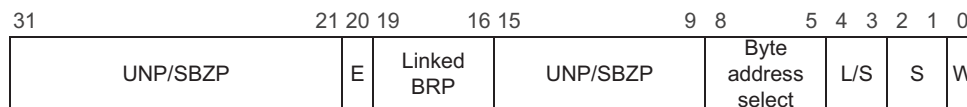


Figure 13-7 Watchpoint Control Registers, format

Bit field definitions for the Watchpoint Control Registers are shown in Table 13-12.

Table 13-12 Watchpoint Control Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:21]	UNP/SBZP	-	Reserved.
[20]	RW	-	Enable linking bit: 0 = Linking disabled 1 = Linking enabled. When this bit is set, this watchpoint is linked with the context ID holding BRP selected by the linked BRP field.
[19:16]	RW	-	Linked BRP. The binary number encoded here indicates a context ID holding BRP to link this WRP with.
[15:9]	SBZ	-	Reserved.
[8:5]	RW	-	Byte address select. The WVR is programmed with a word address. This field can be used to program the watchpoint so it hits only if certain byte addresses are accessed. b0000 = The watchpoint never hits bxxx1 = If the byte at address WVR[31:2]+0 is accessed, the watchpoint hits bxx1x = If the byte at address WVR[31:2]+1 is accessed, the watchpoint hits bx1xx = If the byte at address WVR[31:2]+2 is accessed, the watchpoint hits b1xxx = If the byte at address WVR[31:2]+3 is accessed, the watchpoint hits.
			Note
			These are little-endian byte addresses. This ensures that a watchpoint is triggered regardless of the way it is accessed.
			For example, if a watchpoint is set on a certain byte in memory by doing WCR[8:5] = b0001. LDRB r0, #0x0 it triggers the watchpoint in little-endian mode, as does LDRB r0, #x3 in legacy big-endian mode (B bit of CP15 c1 set).

Table 13-12 Watchpoint Control Registers, bit field definitions (continued)

Bits	Read/write attributes	Reset value	Description
[4:3]	RW	-	Load/store access. The watchpoint can be conditioned to the type of access being done: b00 = Reserved b01 = Load b10 = Store b11 = Either. A SWP triggers on Load, Store, or Either. A load exclusive instruction, LDREX, triggers on Load or Either. A store exclusive instruction, STREX, triggers on Store or Either, whether it succeeded or not.
[2:1]	RW	-	Supervisor Access. The watchpoint can be conditioned to the privilege of the access being done: b00 = Reserved b01 = Privileged b10 = User b11 = Either.
[0]	RW	0	Watchpoint enable: 0 = Watchpoint disabled 1 = Watchpoint enabled.

In addition to the rules for breakpoint debug event generation, see *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 13-17, the following rules apply to the ARM1136JF-S processor for watchpoint debug event generation:

- The update of a WVR or a WCR can take effect several instructions after the corresponding MCR. It is only guaranteed to have taken effect by the next IMB.
- Any WRP can be linked with any BRP with context ID comparison capability. Several BRPs (holding IVAs) and WRPs can be linked with the same context ID capable BRP.
- If a WRP is linked with a BRP that is not configured for context ID comparison and linking, it is Unpredictable if a watchpoint debug event is generated or not. BCR[21:20] fields of the BRP must be set to b11.
- If a WRP is linked with a BRP that is not implemented, it is Unpredictable if a watchpoint debug event is generated or not.
- If a WRP is linked with a BRP and they are not both enabled (BCR[0] and WCR[0] set), it does not generate a watchpoint debug event.

13.4 CP14 registers reset

The CP14 debug registers are all reset by the ARM1136JF-S processor power-on reset signal, **nPORESETIN**, see *Power-on reset* on page 9-8.

This ensures that a vector catch set on the reset vector is taken when **nRESETIN** is deasserted. It also ensure that the DBGTAP debugger can be connected when the processor is running without clearing CP14 debug setting, because **DBGnTRST** does not reset these registers.

13.5 CP14 debug instructions

The CP14 debug instructions are shown in Table 13-13.

Table 13-13 CP14 debug instructions

Binary address		Register number	Abbreviation	Legal instructions
Opcode_2	CRm			
b000	b0000	0	DIDR	MRC p14, 0, <Rd>, c0, c0, 0 ^a
b000	b0001	1	DSCR	MRC p14, 0, <Rd>, c0, c1, 0 ^a MRC p14, 0, R15, c0, c1, 0 MCR p14, 0, <Rd>, c0, c1, 0 ^a
b000	b0101	5	DTR (rDTR/wDTR)	MRC p14, 0, <Rd>, c0, c5, 0 ^a MCR p14, 0, <Rd>, c0, c5, 0 ^a STC p14, c5, <addressing mode> LDC p14, c5, <addressing mode>
b000	b0111	7	VCR	MRC p14, 0, <Rd>, c0, c7, 0 ^a MCR p14, 0, <Rd>, c0, c7, 0 ^a
b100	b0000-b1111	64-79	BVR	MRC p14, 0, <Rd>, c0, cy, 4 ^{ab} MCR p14, 0, <Rd>, c0, cy, 4 ^{ab}
b101	b0000-b1111	80-95	BCR	MRC p14, 0, <Rd>, c0, cy, 5 ^{ab} MCR p14, 0, <Rd>, c0, cy, 5 ^{ab}
b110	b0000-b1111	96-111	WVR	MRC p14, 0, <Rd>, 0, cy, 6 ^{ab} MCR p14, 0, <Rd>, 0, cy, 6 ^{ab}
b111	b0000-b1111	112-127	WCR	MRC p14, 0, <Rd>, c0, cy, 7 ^{ab} MCR p14, 0, <Rd>, c0, cy, 7 ^{ab}

a. <Rd> is any of R0-14 ARM registers.

b. y is the decimal representation for the binary number CRm.

In Table 13-13, MRC p14, 0, <Rd>, c0, c5, 0 and STC p14, c5, <addressing mode> refer to the rDTR and MCR p14, 0, <Rd>, c0, c5, 0 and LDC p14, c5, <addressing mode> refer to the wDTR. See *CP14 c5, Data Transfer Registers (DTR)* on page 13-14 for more details.

The MRC p14, 0, R15, c0, c1, 0 instruction sets the CPSR flags as follows:

- N flag = DSCR[31]. This is an Unpredictable value.
- Z flag = DSCR[30]. This is the value of the rDTRfull flag.
- C flag = DSCR[29]. This is the value of the wDTRfull flag.

- V flag = DSCR[28]. This is an Unpredictable value.

Instructions that follow the MRC instruction can be conditioned to these CPSR flags.

13.5.1 Executing CP14 debug instructions

If the core is in debug state (see *Debug state* on page 13-34), you can execute any CP14 debug instruction regardless of the processor mode.

If the processor tries to execute a CP14 debug instruction that either is not in Table 13-13 on page 13-25, or is targeted to a reserved register, such as a non-implemented BVR, the Undefined instruction exception is taken.

You can access the DCC (read DIDR, read DSCR and read/write DTR) in User mode. All other CP14 debug instructions are privileged. If the processor tries to execute one of these in User mode, the Undefined instruction exception is taken.

If the User mode access to DCC disable bit, DSCR[12], is set, all CP14 debug instructions are considered as privileged, and all attempted User mode accesses to CP14 debug registers generate an Undefined instruction exception.

When DSCR bit 14 is set (Halt mode selected and enabled), if the software running on the processor tries to access any register other than the DIDR, the DSCR, or the DTR, the core takes the Undefined instruction exception. The same thing happens if the core is not in any Debug mode (DSCR[15:14]=b00).

This lockout mechanism ensures that the software running on the core cannot modify the settings of a debug event programmed by the DBGTAP debugger.

Table 13-14 on page 13-27 shows the results of executing CP14 debug instructions.

Table 13-14 Debug instruction execution

State when executing CP14 debug instruction:				Results of CP14 debug instruction execution:		
Processor mode	Debug state	DSCR[15:14] (Mode enabled and selected)	DSCR[12] (DCC User accesses disabled)	Read DIDR, read DSCR and read/write DTR	Write DSCR	Read/write other registers
x	Yes	xx	x	Proceed	Proceed	Proceed
User	No	xx	0	Proceed	Undefined exception	Undefined exception
User	No	xx	1	Undefined exception	Undefined exception	Undefined exception
Privileged	No	b00 (None)	x	Proceed	Proceed	Undefined exception
Privileged	No	b01 (Halt)	x	Proceed	Proceed	Undefined exception
Privileged	No	b10 (Monitor)	x	Proceed	Proceed	Proceed
Privileged	No	b11 (Halt)	x	Proceed	Proceed	Undefined exception

13.6 Debug events

A debug event is any of the following:

- *Software debug event*
- *External debug request signal* on page 13-29
- *Halt DBGTAP instruction* on page 13-29.

13.6.1 Software debug event

A software debug event is any of the following:

- A watchpoint debug event. This occurs when:
 - the DVA present in the data bus matches the watchpoint value
 - all the conditions of the WCR match
 - the watchpoint is enabled
 - the linked contextID-holding BRP (if any) is enabled and its value matches the context ID in CP15 c13.
- A breakpoint debug event. This occurs when:
 - an instruction was fetched and the IVA present in the instruction bus matched the breakpoint value
 - at the same time the instruction was fetched, all the conditions of the BCR matched
 - the breakpoint was enabled
 - at the same time the instruction was fetched, the linked contextID-holding BRP (if any) was enabled and its value matched the context ID in CP15 c13
 - the instruction is now committed for execution.
- A breakpoint debug event also occurs when:
 - an instruction was fetched and the CP15 Context ID (register 13) matched the breakpoint value
 - at the same time the instruction was fetched, all the conditions of the BCR matched
 - the breakpoint was enabled
 - the instruction is now committed for execution.
- A software breakpoint debug event. This occurs when a BKPT instruction is committed for execution.

- A vector catch debug event. This occurs when:
 - The instruction at a vector location was fetched. This includes any kind of prefetches, not just the ones due to exception entry.
 - At the same time the instruction was fetched, the corresponding bit of the VCR was set (vector catch enabled).
 - The instruction is now committed for execution.

13.6.2 External debug request signal

The ARM1136JF-S processor has an external debug request input signal, **EDBGRQ**. When this signal is HIGH it causes the processor to enter debug state when execution of the current instruction has completed. When this happens, the DSCR[5:2] method of entry bits are set to b0100.

This signal can be driven by the ETM to signal a trigger to the core. For example, if the processor is in Halt mode and a memory permission fault occurs, an external Trace analyzer can collect trace information around this trigger event at the same time that the processor is stopped to examine its state. See the *Chapter 15 Trace Interface Port* for more details. A DBGTAP debugger can also drive this signal.

13.6.3 Halt DBGTAP instruction

The Halt mechanism is used by the Debug Test Access Port to force the core into debug state. When this happens, the DSCR[5:2] method of entry bits are set to b0000.

13.6.4 Behavior of the processor on debug events

This section describes how the processor behaves on debug events while not in debug state. See *Debug state* on page 13-34 for information on how the processor behaves while in debug state.

When a software debug event occurs and Monitor mode is selected and enabled then a Debug exception is taken. However, Prefetch Abort and Data Abort Vector catch debug events are ignored. This is to avoid the processor ending in an unrecoverable state on certain combinations of exceptions and vector catches. Unlinked context ID breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor mode is selected and enabled.

The external debug request signal and the Halt DBGTAP instruction are ignored when Monitor mode is selected and enabled.

When a debug event occurs and Halt mode is selected and enabled then the processor enters debug state.

When neither Halt nor Monitor mode is selected and enabled, all debug events are ignored, although the BKPT instruction generates a Prefetch Abort exception.

Table 13-15 Behavior of the processor on debug events

DSCR[15:14]	Mode selected and enabled	Action on software debug event	Action on external debug request signal activation	Action on Halt DBGTAP
b00	None	Ignore/Prefetch Abort ^a	Ignore	Ignore
b01	Halt	Debug state entry	Debug state entry	Debug state entry
b10	Monitor	Debug exception/Ignore ^b	Ignore	Ignore
b11	Halt	Debug state entry	Debug state entry	Debug state entry

- a. When debug is disabled, a BKPT instruction generates a Prefetch Abort exception instead of being ignored.
- b. Prefetch Abort and Data Abort vector catch debug events are ignored in Monitor mode. Unlinked context ID breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor mode is selected and enabled.

13.6.5 Effect of a debug event on CP15 registers

The four CP15 registers that can be set on a debug event are:

- *Instruction Fault Status Register (IFSR)*
- *Data Fault Status Register (DFSR)*
- *Fault Address Register (FAR)*
- *Instruction Fault Address Register (IFAR).*

They are set under the following circumstances:

- The IFSR is set whenever a breakpoint, software breakpoint, or vector catch debug event generates a Debug exception entry. It is set to indicate the cause for the Prefetch Abort vector fetch.
- The DFSR is set whenever a watchpoint debug event generates a Debug exception entry. It is set to indicate the cause for the Data Abort vector fetch.
- The ARM1136JF-S processor updates the FAR on debug exception entry because of watchpoints, although this is architecturally Unpredictable. It is set to the *Modified Virtual Address (MVA)* that triggered the watchpoint.
- The IFAR is set whenever a watchpoint debug event generates either a Debug exception or debug state entry. It is set to the VA of the instruction that caused the Watchpoint debug event, plus an offset dependent on the processor state. These offsets are the same as the ones shown in Table 13-18 on page 13-35.

Table 13-16 shows the setting of CP15 registers on debug events.

Table 13-16 Setting of CP15 registers on debug events

Register	Debug exception taken due to:		Debug state entry due to:	
	A breakpoint, software breakpoint, or vector catch debug event	A watchpoint debug event	A debug event other than a watchpoint	A watchpoint debug event
IFSR	Cause of Prefetch Abort exception handler entry	Unchanged	Unchanged	Unchanged
DFSR	Unchanged	Cause of Data Abort exception handler entry	Unchanged	Unchanged
FAR	Unchanged	Watchpointed address	Unchanged	Unchanged
IFAR	Unchanged	Address of the instruction causing the watchpoint debug event	Unchanged	Address of the instruction causing the watchpoint debug event

You must take care when setting a breakpoint or software breakpoint debug event inside the Prefetch Abort or Data Abort exception handlers, or when setting a watchpoint debug event on a data address that might be accessed by any of these handlers. These debug events overwrite the `r14_abt`, `SPRS_abt` and the CP15 registers listed in this section, leading to an unpredictable software behavior if the handlers did not have the chance of saving the registers.

13.7 Debug exception

When a Software debug event occurs and Monitor mode is selected and enabled then a Debug exception is taken. Prefetch Abort and Data Abort Vector catch debug events are ignored though. Unlinked context ID breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor mode is selected and enabled.

If the cause of the Debug exception is a watchpoint debug event, the processor performs the following actions:

- The DSCR[5:2] method of entry bits are set to indicate that a watchpoint occurred.
- The CP15 DFSR, FAR, and IFAR, are set as described in *Effect of a debug event on CP15 registers* on page 13-30.
- The same sequence of actions as in a Data Abort exception is performed. This includes setting the r14_abt, base register and destination registers to the same values as if this was a Data Abort.

The Data Abort handler is responsible for checking the DFSR or DSCR[5:2] bit to determine if the routine entry was caused by a debug exception or a Data Abort exception. On entry:

1. It must first check for the presence of a monitor target.
2. If present, the handler must disable the active watchpoints. This is necessary to prevent corruption of the FAR because of an unexpected watchpoint debug event whilst servicing a Data Abort exception.
3. If the cause is a Debug exception the Data Abort handler branches to the monitor target.

———— **Note** —————

- the watchpointed address can be found in the FAR
 - the address of the instruction that caused the watchpoint debug event can be found in the IFAR
 - the address of the instruction to restart at plus 0x08 can be found in the r14_abt register.
-

If the cause of the Debug exception is a breakpoint, software breakpoint or vector catch debug event, the processor performs the following actions:

- the DSCR[5:2] method of entry bits are set appropriately

- the CP15 IFSR register is set as described in *Effect of a debug event on CP15 registers* on page 13-30.
- the same sequence of actions as in a Prefetch Abort exception is performed.

The Prefetch Abort handler is responsible for checking the IFSR or DSCR[5:2] bits to find out if the routine entry is caused by a Debug exception or a Prefetch Abort exception. If the cause is a Debug exception it branches to the monitor target.

———— **Note** —————

The address of the instruction causing the Software debug event plus 0x04 can be found in the r14_abt register.

Table 13-17 shows the values in the link register after exceptions.

Table 13-17 Values in the link register after exceptions

Cause of the fault	ARM	Thumb	Java	Return address (RA^a) meaning
Breakpoint	RA+4	RA+4	RA+4	Breakpointed instruction address
Watchpoint	RA+8	RA+8	RA+8	Address of the instruction where the execution resumes (a number of instructions after the one that hit the watchpoint)
BKPT instruction	RA+4	RA+4	RA+4	BKPT instruction address
Vector catch	RA+4	RA+4	RA+4	Vector address
Prefetch Abort	RA+4	RA+4	RA+4	Address of the instruction where the execution resumes
Data Abort	RA+8	RA+8	RA+8	Address of the instruction where the execution resumes

- a. This is the address of the instruction that the processor first executes on debug state exit. Watchpoints can be imprecise. RA is not the address of the instruction just after the one that hit the watchpoint, the processor might stop a number of instructions later. The address of the instruction that hit the watchpoint is in the CP15 IFAR.

13.8 Debug state

When the conditions in *Behavior of the processor on debug events* on page 13-29 are met then the processor switches to debug state. While in debug state, the processor behaves as follows:

- The DSCR[0] core halted bit is set.
- The **DBGACK** signal is asserted, see *External signals* on page 13-49.
- The DSCR[5:2] method of entry bits are set appropriately.
- The CP15 IFSR, DFSR, and FAR registers are set as described in *Effect of a debug event on CP15 registers* on page 13-30. The IFAR is set to an Unpredictable value.
- The processor is halted. The pipeline is flushed and no instructions are fetched.
- The processor does not change the execution mode. The CPSR is not altered.
- The DMA engine keeps on running. The DBGTAP debugger can stop it and restart it using CP15 operations. See Chapter 7 *Level One Memory System* for details.
- Interrupts and exceptions are treated as described in *Interrupts* on page 13-36 and *Exceptions* on page 13-36.
- Software debug events are ignored.
- The external debug request signal is ignored.
- Debug state entry request commands are ignored.
- There is a mechanism, using the Debug Test Access Port, where the core is forced to execute an ARM state instruction. This mechanism is enabled using DSCR[13] execute ARM instruction enable bit.
- The core executes the instruction as if it is in ARM state, regardless of the actual value of the T and J bits of the CPSR. If you do set both the J and T bits the behavior is Unpredictable.
- In this state the core can execute any ARM state instruction, as if in a privileged mode. For example, if the processor is in User mode then the MSR instruction updates the PSRs and all the CP14 debug instructions can be executed. However, the processor still accesses the register bank and memory as indicated by the CPSR mode bits. For example, if the processor is in User mode then it sees the User mode register bank, and accesses the memory without any privilege.

- The PC behaves as described in *Behavior of the PC in debug state*.
- A DBGTAP debugger can force the processor out of debug state by issuing a Restart instruction, see Table 14-1 on page 14-6. The Restart command clears the DSCR[1] core restarted flag. When the processor has actually exited debug state, the DSCR[1] core restarted bit is set and the DSCR[0] core halted bit and **DBGACK** signal are cleared.

13.8.1 Behavior of the PC in debug state

In debug state:

- The PC is frozen on entry to debug state. That is, it does not increment on the execution of ARM instructions. However, branches and instructions that modify the PC directly do update it.
- If the PC is read after the processor has entered debug state, it returns a value as described in Table 13-18, depending on the previous state and the type of debug event.
- If a sequence for writing a certain value to the PC is executed while in debug state, and then the processor is forced to restart, execution starts at the address corresponding to the written value. However, the CPSR has to be set to the return ARM, Thumb, or Java state before the PC is written to, otherwise the processor behavior is Unpredictable.
- If the processor is forced to restart without having performed a write to the PC, the restart address is Unpredictable.
- If the PC or CPSR are written to while in debug state, subsequent reads to the PC return an Unpredictable value.
- If a conditional branch is executed and it fails its condition code, an Unpredictable value is written to the PC.

Table 13-18 shows the read PC value after debug state entry for different debug events.

Table 13-18 Read PC value after debug state entry

Debug event	ARM	Thumb	Java	Return address (RA ^a) meaning
Breakpoint	RA+8	RA+4	RA	Breakpointed instruction address
Watchpoint	RA+8	RA+4	RA	Address of the instruction where the execution resumes (several instructions after the one that hit the watchpoint)
BKPT instruction	RA+8	RA+4	RA	BKPT instruction address

Table 13-18 Read PC value after debug state entry (continued)

Debug event	ARM	Thumb	Java	Return address (RA^a) meaning
Vector catch	RA+8	RA+4	RA	Vector address
External debug request signal activation	RA+8	RA+4	RA	Address of the instruction where the execution resumes
Debug state entry request command	RA+8	RA+4	RA	Address of the instruction where the execution resumes

- a. This is the address of the instruction that the processor first executes on debug state exit. Watchpoints can be imprecise. RA is not the address of the instruction just after the one that hit the watchpoint, the processor might stop a number of instructions later. The address of the instruction that hit the watchpoint is in the CP15 IFAR.

13.8.2 Interrupts

Interrupts are ignored regardless of the value of the I and F bits of the CPSR, although these bits are not changed because of the debug state entry.

13.8.3 Exceptions

Exceptions are handled as follows while in debug state:

Reset This exception is taken as in a normal processor state, ARM, Thumb, or Java. This means the processor leaves debug state as a result of the system reset.

Prefetch Abort

This exception cannot occur because no instructions are prefetched while in debug state.

Debug This exception cannot occur because software debug events are ignored while in debug state.

SWI and Undefined exceptions

If one of these exception occurs while in debug state the behavior of the ARM1136JF-S processor is Unpredictable.

Data abort

When a Data Abort occurs in debug state, the behavior of the core is as follows:

- The PC, CPSR, and SPSR_abt are set as for a normal processor state exception entry.

- If the debugger has not written to the PC or the CPSR while in debug state, R14_abt is set as described in the *ARM Architecture Reference Manual*.
- If the debugger has written to the PC or the CPSR while in debug state, R14_abt is set to an Unpredictable value.
- The processor remains in debug state and does not fetch the exception vector.
- The DFSR, and FAR are set as for a normal processor state exception entry. The IFAR is set to an Unpredictable value.
- The DSCR[6] sticky precise Data Abort bit, or the DSCR[7] sticky imprecise Data Aborts bit are set.
- The DSCR[5:2] method of entry bits are set to b0110.

If it is an imprecise Data Abort and the debugger has not written to the PC or CPSR, R14_abt is set as described in the *Architecture Reference Manual*. Therefore the processor is in the same state as if the exception was taken on the instruction that was cancelled by the debug state entry sequence. This is necessary because it is not possible to guarantee that the debugger reads the PC before an imprecise Data Abort exception is taken.

13.9 Debug communications channel

There are two ways that a DBGTAP debugger can send data to or receive data from the core:

- The debug communications channel, when the core is not in debug state. It is defined as the set of resources used for communicating between the DBGTAP debugger and a piece of software running on the core.
- The mechanism for forcing the core to execute ARM instructions, when the core is in debug state. For details see *Executing instructions in debug state* on page 14-24.

At the core side, the debug communications channel resources are:

- CP14 Debug Register c5 (DTR). Data coming from a DBGTAP debugger can be read by an MRC or STC instruction addressed to this register. The core can write to this register any data intended for the DBGTAP debugger, using an MCR or LDC instruction. Because the DTR comprises both a read (rDTR) and a write portion (wDTR), a data item written by the core can be held in this register at the same time as one written by the DBGTAP debugger.
- Some flags and control bits of CP14 Debug Register c1 (DSCR):
 - User mode access to comms channel disable, DSCR[12]. If this bit is set, only privileged software is able to access the debug communications channel. That is, access the DSCR and the DTR.
 - wDTRfull flag, DSCR bit 29. When clear, this flag indicates to the core that the wDTR is ready to receive data. It is automatically cleared on reads of the wDTR by the DBGTAP debugger, and is set on writes by the core to the same register. If this bit is set and the core attempts to write to the wDTR, the register contents are overwritten and the wDTRfull flag remains set.
 - rDTRfull flag, DSCR bit 30. When set, this flag indicates to the core that there is data available to read at the rDTR. It is automatically set on writes to the rDTR by the DBGTAP debugger, and is cleared on reads by the core of the same register.

The DBGTAP debugger side of the debug communications channel is described in *Monitor mode debugging* on page 14-50.

13.10 Debugging in a cached system

Debugging must be non-intrusive in a cached system. In ARM1136JF-S systems, you can preserve the contents of the cache so the state of the target application is not altered, and to maintain memory coherency during debugging.

To preserve the contents of the level one cache, you can disable the Instruction Cache and Data Cache line fills so read misses from main memory do not update the caches. You can put the caches in this mode by programming the operation of the caches during debug using CP15 c15. See *Cache Debug Control Register* on page 3-34. This facility is accessible from both the core and DBGTAP debugger sides.

In debug state, the caches behave as follows, for memory coherency purposes:

- Cache reads behave as for normal operation.
- Writes are covered in *Data cache writes*.
- ARMv6 includes CP15 instructions for cleaning and invalidating the cache content, See *Cache Operations Register* on page 3-17. These instructions enable you to reset the processor memory system to a known safe state, and are accessible from both the core and the DBGTAP debugger side.

13.10.1 Data cache writes

The problem with Data Cache writes is that, while debugging, you might want to write some instructions to memory, either some code to be debugged or a BKPT instruction. This poses coherency issues on the Instruction Cache.

In ARM1136JF-S systems, CP15 c15, the Cache Debug Control Register, enables you to use the following features:

- You can put the processor in a state where data writes work as if the cache is enabled and every region of memory is Write-Through. This facility is accessible from both the core and the DBGTAP debugger side. See *Cache Debug Control Register* on page 3-34.
- ARMv6 architecture provides CP15 instructions for invalidating the Instruction Cache, described in *Cache Operations Register* on page 3-17 to ensure that, after a write, there are no out-of-date words in the Instruction Cache.

13.11 Debugging in a system with TLBs

Debugging in a system with TLBs has to be as non-intrusive as possible. There has to be a way to put the TLBs in a state where their contents are not affected by the debugging process. This facility has to be accessible from both the core and the DBGTAP debugger side. The ARM1136JF-S processor enables you to put the TLBs in this mode using CP15 c15. See *Control of main TLB and MicroTLB loading and matching* on page 3-41.

The ARM1136JF-S processor also enables you to read the state of the MicroTLBs and main TLB with no side effects. This facility is accessible through CP15 c15 operations. See *MMU debug operations* on page 3-38 for more details.

13.12 Monitor mode debugging

Monitor mode debugging is essential in real-time systems when the integer unit cannot be halted to collect information. Engine controllers and servo mechanisms in hard drive controllers are examples of systems that might not be able to stop the code without physically damaging components. These are typical systems that can be debugged using Monitor mode.

For situations that can only tolerate a small intrusion into the instruction stream, Monitor mode is ideal. Using this technique, code can be suspended with an exception long enough to save off state information and important variables. The code continues when the exception handler is finished. The *Method Of Entry* (MOE) bits in the DSCR can be read to determine what caused the exception.

When in Monitor mode, all breakpoint and watchpoint registers can be read and written with MRC and MCR instructions from a privileged processing mode.

13.12.1 Entering the monitor target

Monitor mode is the default mode on power-on reset. Only a DBGTAP debugger can change the mode bit in the DSCR. When a software debug event occurs (as described in *Software debug event* on page 13-28) and Monitor mode is selected and enabled, then a Debug exception is taken, although Prefetch Abort and Data Abort vector catch debug events are ignored. Debug exception entry is described in *Debug exception* on page 13-32. The Prefetch Abort handler can check the IFSR or the DSCR[5:2] bits, and the Data Abort handler can check the DFSR or the DSCR[5:2] bits, to find out the cause of the exception. If the cause was a Debug exception, the handler branches to the monitor target.

When the monitor target is running, it can determine and modify the processor state and new software debug events can be programmed.

13.12.2 Setting breakpoints, watchpoints, and vector catch debug events

When the monitor target is running, breakpoints, watchpoints, and vector catch debug events can be set. This can be done by executing MCR instructions to program the appropriate CP14 debug registers. The monitor target can only program these registers if the processor is in a privileged mode and Monitor mode is selected and enabled, see *Debug Status And Control Register bit field definitions* on page 13-11.

You can program a vector catch debug event using CP14 Debug Vector Catch Register.

You can program a breakpoint debug event using CP14 debug breakpoint value registers and CP14 Debug Breakpoint Control Registers, see *CP14 c64-c69, Breakpoint Value Registers (BVR)* on page 13-16 and *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 13-17.

You can program a watchpoint debug event using CP14 Debug Watchpoint Value Registers and CP14 Debug Watchpoint Control Registers, see *CP14 c96-c97, Watchpoint Value Registers (WVR)* on page 13-21, and *CP14 c112-c113, Watchpoint Control Registers (WCR)* on page 13-21.

Setting a simple breakpoint on an IVA

You can set a simple breakpoint on an IVA as follows:

1. Read the BCR.
2. Clear the BCR[0] enable breakpoint bit in the read word and write it back to the BCR. Now the breakpoint is disabled.
3. Write the IVA to the BVR register.
4. Write to the BCR with its fields set as follows:
 - BCR[21] meaning of BVR bit cleared, to indicate that the value loaded into BVR is to be compared against the IVA bus.
 - BCR[20] enable linking bit cleared, to indicate that this breakpoint is not to be linked.
 - BCR[8:5] byte address select BCR field as required.
 - BCR[2:1] supervisor access BCR field as required.
 - BCR[0] enable breakpoint bit set.

————— Note —————

Any BVR can be compared against the IVA bus.

Setting a simple breakpoint on a context ID value

A simple breakpoint on a context ID value can be set, using one of the context ID capable BRPs, as follows:

1. Read the BCR.
2. Clear the BCR[0] enable breakpoint bit in the read word and write it back to the BCR. Now the breakpoint is disabled.

3. Write the context ID value to the BVR register.
4. Write to the BCR with its fields set as follows:
 - BCR[21] meaning of BVR bit set, to indicate that the value loaded into BVR is to be compared against the CP15 Context Id Register c13.
 - BCR[20] enable linking bit cleared, to indicate that this breakpoint is not to be linked.
 - BCR[8:5] byte address select BCR field set to b1111.
 - BCR[2:1] supervisor access BCR field as required.
 - BCR[0] enable breakpoint bit set.

Note

Any BVR can be compared against the IVA bus.

Setting a linked breakpoint

In the following sequence b is any of the breakpoint registers pairs with context ID comparison capability, and a is any of the implemented breakpoints different from b.

You can link IVA holding and contextID-holding breakpoints register pairs as follows:

1. Read the BCRa and BCRb.
2. Clear the BCRa[0] and BCRb[0] enable breakpoint bits in the read words and write them back to the BCRs. Now the breakpoints are disabled.
3. Write the IVA to the BVRa register.
4. Write the context ID to the BVRb register.
5. Write to the BCRb with its fields set as follows:
 - BCRb[21] meaning of BVR bit set, to indicate that the value loaded into BVRb is to be compared against the CP15 context ID register 13
 - BCRb[20] enable linking bit, set
 - BCRb[8:5] byte address select set to b1111
 - BCRb[2:1] supervisor access set to b11
 - BCRb[0] enable breakpoint bit set.
6. Write to the BCRa with its fields set as follows:
 - BCRa[21] meaning of BVR bit cleared, to indicate that the value loaded into BVRa is to be compared against the IVA bus

- BCRa[20] enable linking bit set, in order to link this BRP with the one indicated by BCRa[19:16] (BRPb in this example)
- binary representation of b into BCR[19:6] linked BRP field
- BCRa[8:5] byte address select field as required
- BCRa[2:1] supervisor access field as required
- BCRa[0] enable breakpoint set.

Setting a simple watchpoint

You can set a simple watchpoint as follows:

1. Read the WCR.
2. Clear the WCR[0] enable watchpoint bit in the read word and write it back to the WCR. Now the watchpoint is disabled.
3. Write the DVA to the WVR register.
4. Write to the WCR with its fields set as follows:
 - WCR[20] enable linking bit cleared, to indicate that this watchpoint is not to be linked
 - WCR byte address select, load/store access, and supervisor access fields as required
 - WCR[0] enable watchpoint bit set.

Note

Any WVR can be compared against the DVA bus.

Setting a linked watchpoint

In the following sequence b is any of the BRPs with context ID comparison capability. You can use any of the WRP.

You can link WRP and contextID-holding BRPs as follows:

1. Read the WCR and BCRb.
2. Clear the WCR[0] Enable Watchpoint and the BCRb[0] Enable breakpoint bits in the read words and write them back to the WCR and BCRb. Now the watchpoint and the breakpoint are disabled.
3. Write the DVA to the WVR register.

4. Write the context ID to the BVRb register.
5. Write to the WCR with its fields set as follows:
 - WCR[20] enable linking bit set, in order to link this WRP with the BRP indicated by WCR[19:16] (BRPb in this example)
 - Binary representation of b into WCR[19:6] linked BRP field
 - WCR byte address select, load/store access, and supervisor access fields as required
 - WCR[0] enable watchpoint bit set.
6. Write to the BCRb with its fields set as follows:
 - BCRb[21] meaning of BVR bit set, to indicate that the value loaded into BVRb is to be compared against the CP15 Context ID Register.
 - BCRb[20] enable linking bit, set
 - BCRb[8:5] byte address select set to b1111
 - BCRb[2:1] supervisor access set to b11
 - BCRb[0] enable breakpoint bit set.

13.12.3 Setting software breakpoint debug events (BKPT)

To set a software breakpoint on a particular virtual address, the monitor target must perform the following steps:

1. Read memory location and save actual instruction.
2. Write BKPT instruction to the memory location.
3. Read memory location again to check that the BKPT instruction has been written.
4. If it has not been written, determine the reason.

————— **Note** —————

Cache coherency issues might arise when writing a BKPT instruction. See *Debugging in a cached system* on page 13-39.

13.12.4 Using the debug communications channel

To read a word sent by a DBGTAP debugger:

1. Read the DSCR register.
2. If DSCR[30] rDTRfull flag is clear, then go to 1.

3. Read the word from the rDTR, CP14 Debug Register c5.

To write a word for a DBGTAP debugger:

1. Read the DSCR register.
2. If DSCR[29] wDTRfull flag is set, then go to 1.
3. Write the word to the wDTR, CP14 Debug Register c5.

13.13 Halt mode debugging

Halt mode is used to debug the ARM1136JF-S processor using external hardware connected to the DBGTAP. The external hardware provides an interface to a DBGTAP debugger application. You can only select Halt mode by setting the halt bit (bit 14) of the DSCR, which is only writable through the Debug Test Access Port. See Chapter 14 *Debug Test Access Port*.

In Halt mode the processor stops executing instructions if one of the following events occurs:

- a breakpoint hits
- a watchpoint hits
- a BKPT instruction is executed
- the **EDBGRQ** signal is asserted
- a Halt instruction has been scanned into the DBGTAP instruction register
- an vector catch occurs.

When the processor is halted, it is controlled by sending instructions to the integer unit through the DBGTAP. Any valid instruction can be scanned into the processor, and the effect of the instruction upon the integer unit is as if it was executed under normal operation. Also accessible through the DBGTAP is a register to transfer data between CP14 and the DBGTAP debugger.

The integer unit is restarted by executing a DBGTAP Restart instruction.

13.13.1 Entering debug state

When a debug event occurs and Halt mode is selected and enabled then the processor enters debug state as defined in *Debug state* on page 13-34.

When the core is in debug state, the DBGTAP debugger can determine and modify the processor state and new debug events can be programmed.

13.13.2 Exiting debug state

You can force the processor out of debug state using the DBGTAP Restart instruction. See *Exiting debug state* on page 14-5. The DSCR[1] core restarted bit indicates if the core has already returned to normal operation.

13.13.3 Programming debug events

In Halt mode debugging you can program the following debug events:

- *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-48

- *Setting software breakpoints (BKPT)*
- *Reading and writing to memory.*

Setting breakpoints, watchpoints, and vector catch debug events

For setting breakpoints, watchpoints, and vector catch debug events when in Halt mode, the debug host has to use the same CP14 debug registers and the same sequence of operations as in Monitor mode debugging (see *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-41). The only difference is that the CP14 debug registers are accessed using the DBGTAP scan chains, see *The DBGTAP port and debug registers* on page 14-6.

———— Note —————

A DBGTAP debugger can access the CP14 debug registers whether the processor is in debug state or not, so these debug events can be programmed while the processor is in ARM, Thumb, or Java state.

Setting software breakpoints (BKPT)

To set a software breakpoint, the DBGTAP debugger must perform the same steps as the monitor target (described in *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-41). The difference is that CP14 debug registers are accessed using the DBGTAP scan chains, see Chapter 14 *Debug Test Access Port*.

Reading and writing to memory

See *Debug sequences* on page 14-34 for memory access sequences using the ARM1136JF-S Debug Test Access Port.

13.14 External signals

The following external signals are used by debug:

- DBGACK** Debug acknowledge signal. The processor asserts this output signal to indicate the system has entered Debug state. See *Debug state* on page 13-34 for a definition of the Debug state.
- DBGEN** Debug enable signal. When this signal is LOW, DSCR[15:14] is read as 0 and the processor behaves as if in debug disabled mode.
- EDBGRQ** External debug request signal. As described in *External debug request signal* on page 13-29, this input signal forces the core into Debug state if the Debug logic is in Halt mode.
- DBGNOPWRDWN** Powerdown disable signal generated from DSCR[9]. When this signal is HIGH, the system power controller is forced into Emulate mode. This is to avoid losing CP14 debug state that can only be written through the DBGTAP. Therefore, DSCR[9] must only be set if Halt mode debugging is necessary.

Chapter 14

Debug Test Access Port

This chapter introduces the Debug Test Access Port built into ARM1136JF-S processor. It contains the following sections:

- *Debug Test Access Port and Halt mode* on page 14-2
- *Synchronizing RealView™ ICE* on page 14-3
- *Entering debug state* on page 14-4
- *Exiting debug state* on page 14-5
- *The DBG TAP port and debug registers* on page 14-6
- *Debug registers* on page 14-8
- *Using the Debug Test Access Port* on page 14-24
- *Debug sequences* on page 14-34
- *Programming debug events* on page 14-48
- *Monitor mode debugging* on page 14-50.

14.1 Debug Test Access Port and Halt mode

JTAG-based hardware debug using Halt mode provides access to the ARM1136JF-S processor and debug unit. Access is through scan chains and the *Debug Test Access Port* (DBGTAP). The *DBGTAP state Machine* (DBGTAPM) is illustrated in Figure 14-1.

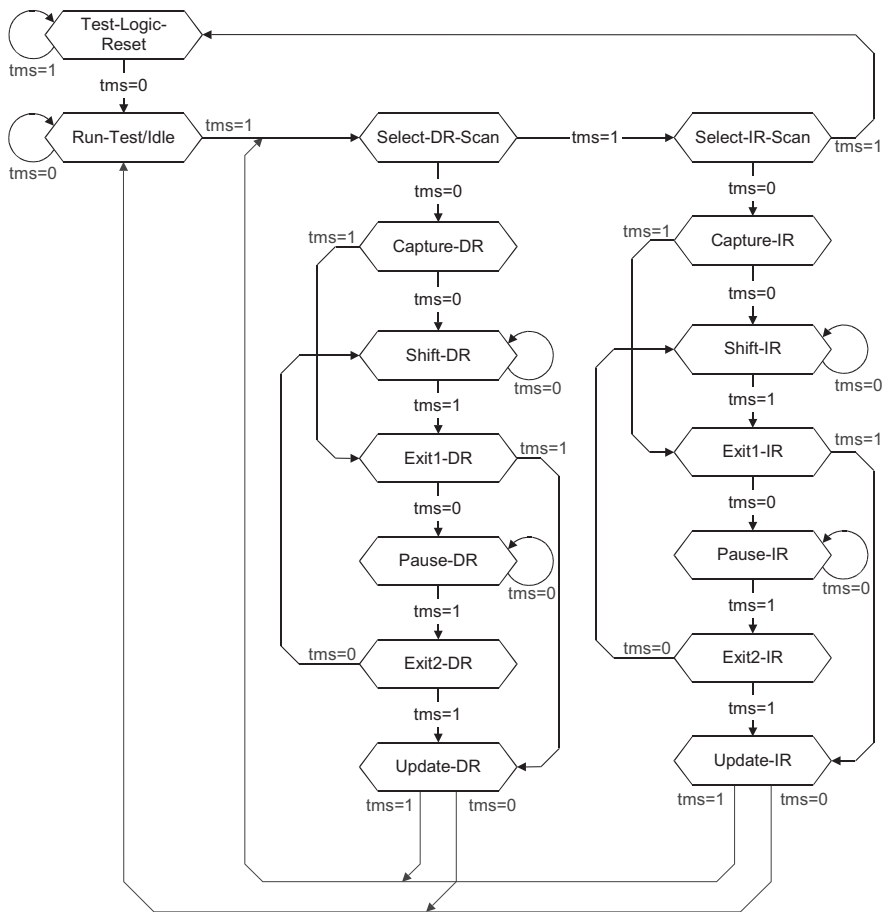


Figure 14-1 JTAG DBGTAP state machine diagram¹

1. From IEEE Std 1149.1-1990. Copyright 2002 IEEE. All rights reserved.

14.2 Synchronizing RealView™ ICE

The system and test clocks must be synchronized externally to the macrocell. The ARM RealView ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM1136 processor you must use a three-stage synchronizer. The off-chip device (for example, RealView 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** edge until after an **RTCK** edge is received. Figure 14-2 shows this synchronization.

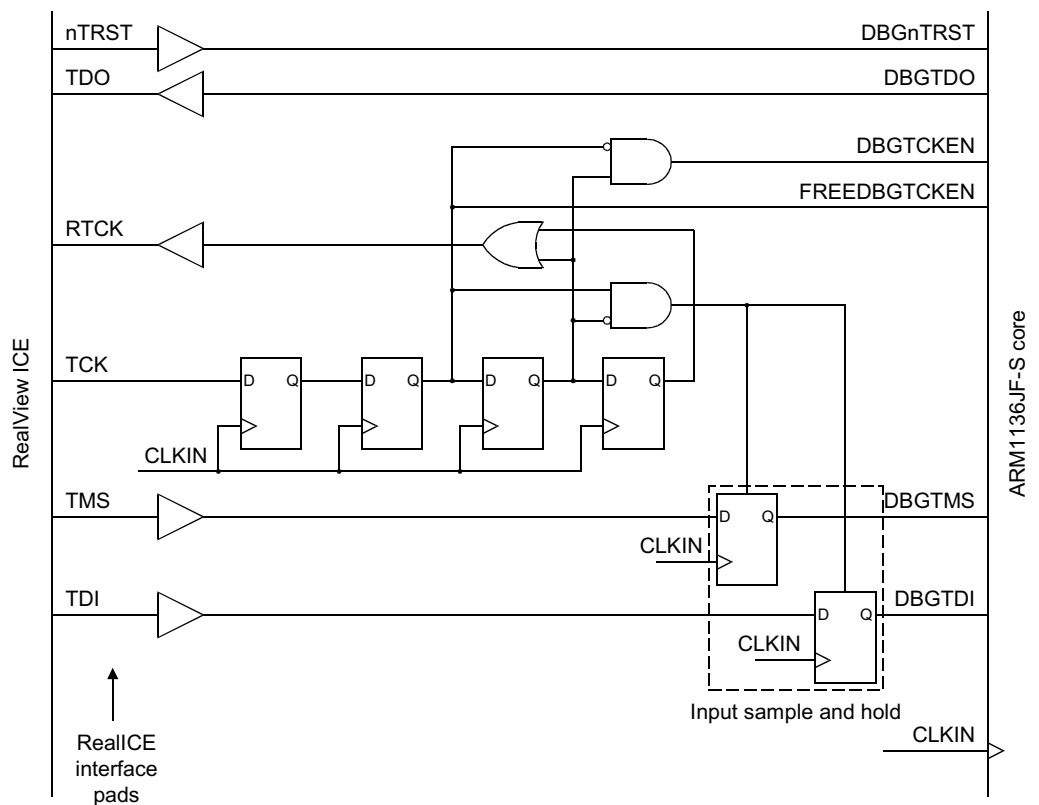


Figure 14-2 Clock synchronization

Note

All of the D types are reset by **DBGnTRST**.

14.3 Entering debug state

Halt mode is enabled by writing a 1 to bit 14 of the DSCR, see *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10. This can only be done by a DBGTAP debugger hardware such as RealView ICE. When this mode is enabled the processor halts, instead of taking an exception in software, if one of the following events occurs:

- A Halt instruction has been scanned in through the DBGTAP. The DBGTAP controller must pass through Run-Test/Idle to issue the Halt command to the ARM.
- A vector catch occurs.
- A breakpoint hits.
- A watchpoint hits.
- A BKPT instruction is executed.
- **EDBGRQ** is asserted.

The core halted bit in the DSCR is set when debug state is entered. At this point, the debugger determines why the integer unit was halted and preserves the processor state. The MSR instruction can be used to change modes and gain access to all banked registers in the machine. While in debug state:

- the PC is not incremented
- interrupts are ignored
- all instructions are read from the instruction transfer register (scan chain 4).

Debug state is described in *Debug state* on page 13-34.

14.4 Exiting debug state

To exit from debug state, scan in the Restart instruction through the ARM1136JF-S DBGTAP. You might want to adjust the PC before restarting, depending on the way the integer unit entered debug state. When the state machine enters the Run-Test/Idle state, normal operations resume. The delay, waiting until the state machine is in Run-Test/Idle, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. When Run-Test/Idle state is entered, all the processors resume operation simultaneously. The core restarted bit is set when the Restart sequence is complete.

14.5 The DBGTAP port and debug registers

The ARM1136JF-S DBGTAP controller is the part of the debug unit that enables access through the DBGTAP to the on-chip debug resources, such as breakpoint and watchpoint registers. The DBGTAP controller is based on the IEEE 1149.1 standard and supports:

- a device ID register
- a bypass register
- a five-bit instruction register
- a five-bit scan chain select register.

In addition, the public instructions listed in Table 14-1 are supported.

Table 14-1 Supported public instructions

Binary code	Instruction	Description
b00000	EXTEST	This instruction connects the selected scan chain between DBGTDI and DBGTDO . When the instruction register is loaded with the EXTEST instruction, the debug scan chains can be written. See <i>Scan chains</i> on page 14-11.
b00001	-	Reserved.
b00010	Scan_N	Selects the scan chain select register (SCREG). This instruction connects SCREG between DBGTDI and DBGTDO . See <i>Scan chain select register (SCREG)</i> on page 14-10.
b00011	-	Reserved.
b00100	Restart	Forces the processor to leave debug state. This instruction is used to exit from debug state. The processor restarts when the Run-Test/Idle state is entered.
b00101	-	Reserved.
b00110	-	Reserved.
b00111	-	Reserved.
b01000	Halt	Forces the processor to enter debug state. This instruction is used to stop the integer unit and put it into debug state. The core can only be put into debug state if Halt mode is enabled.
b01001	-	Reserved.
b01010-b01011	-	Reserved.
b01100	INTEST	This instruction connects the selected scan chain between DBGTDI and DBGTDO . When the instruction register is loaded with the INTEST instruction, the debug scan chains can be read. See <i>Scan chains</i> on page 14-11.

Table 14-1 Supported public instructions (continued)

Binary code	Instruction	Description
b01101-b11100	-	Reserved.
b11101	ITRsel	When this instruction is loaded into the IR (Update-DR state), the DBGTAP controller behaves as if IR=EXTEST and SCREG=4. The ITRsel instruction makes the DBGTAP controller behave as if EXTEST and scan chain 4 are selected. It can be used to speed up certain debug sequences. See <i>Using the ITRsel IR instruction</i> on page 14-25 for the effects of using this instruction.
b11110	IDcode	See IEEE 1149.1. Selects the DBGTAP controller device ID code register. The IDcode instruction connects the device identification register (or ID register) between DBGTDI and DBGTDO . The ID register is a 32-bit register that enables you to determine the manufacturer, part number, and version of a component using the DBGTAP. See <i>Device ID code register</i> on page 14-9 for details of selecting and interpreting the ID register value.
b11111	Bypass	See IEEE 1149.1. Selects the DBGTAP controller bypass register. The Bypass instruction connects a 1-bit shift register (the bypass register) between DBGTDI and DBGTDO . The first bit shifted out is a 0. All unused DBGTAP controller instruction codes default to the Bypass instruction. See <i>Bypass register</i> on page 14-8.

———— **Note** ————

Sample/Preload, Clamp, HighZ, and ClampZ instructions are not implemented because the ARM1136JF-S DBGTAP controller does not support the attachment of external boundary scan chains.

All unused DBGTAP controller instructions default to the Bypass instruction.

14.6 Debug registers

You can connect the following debug registers between **DBGTDI** and **DBGTDO**:

- *Bypass register*
- *Device ID code register* on page 14-9
- *Instruction register* on page 14-9
- *Scan chain select register (SCREG)* on page 14-10
- *Scan chain 0, debug ID register (DIDR)* on page 14-12
- *Scan chain 1, debug status and control register (DSCR)* on page 14-12
- *Scan chain 4, instruction transfer register (ITR)* on page 14-13
- *Scan chain 5* on page 14-15.
- *Scan chain 6* on page 14-19.
- *Scan chain 7* on page 14-19.

14.6.1 Bypass register

Purpose	Bypasses the device by providing a path between DBGTDI and DBGTDO .
Length	1 bit.
Operating mode	When the bypass instruction is the current instruction in the instruction register, serial data is transferred from DBGTDI to DBGTDO in the Shift-DR state with a delay of one TCK cycle. There is no parallel output from the bypass register. A logic 0 is loaded from the parallel input of the bypass register in the Capture-DR state. Nothing happens at the Update-DR state.
Order	The order of bits in the bypass register is shown in Figure 14-3.

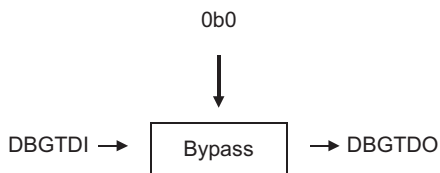


Figure 14-3 Bypass register bit order

14.6.2 Device ID code register

Purpose	Device identification. To distinguish the ARM1136JF-S processor from other processors, the DBGTAP controller ID is unique for each. This means that a DBGTAP debugger such as RealView ICE can easily see which processor it is connected to. The Device ID register version and manufacturer ID fields are routed to the edge of the chip so that partners can create their own Device ID numbers by tying the pins to HIGH or LOW values. The default manufacturer ID for the ARM1136JF-S processor is b11110000111. The part number field is hard-wired inside the ARM1136JF-S to 0x7B36. All ARM semiconductor partner-specific devices must be identified by manufacturer ID numbers of the form shown in <i>ID Code Register</i> on page 3-102.
Length	32 bits
Operating mode	When the ID code instruction is current, the shift section of the device 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 ID code is loaded into this shift section during the Capture-DR state. This is shifted out during Shift-DR (least significant bit first) while a <i>don't care</i> value is shifted in. The shifted-in data is ignored in the Update-DR state.
Order	The order of bits in the ID code register is shown in Figure 14-4.

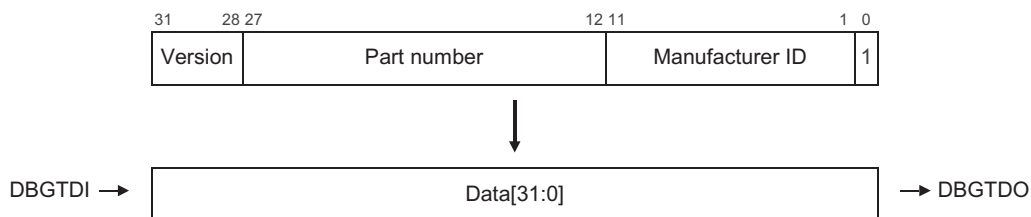


Figure 14-4 Device ID code register bit order

14.6.3 Instruction register

Purpose	Holds the current DBGTAP controller instruction.
Length	5 bits.

Operating mode When in Shift-IR state, the shift section of the instruction register is selected as the serial path between **DBGTDI** and **DBGTDO**. At the Capture-IR state, the binary value b00001 is loaded into this shift section. This is shifted out during Shift-IR (least significant bit first), while a new instruction is shifted in (least significant bit first). At the Update-IR state, the value in the shift section is loaded into the instruction register so it becomes the current instruction. On DBGTAP reset, the IDcode becomes the current instruction.

Order The order of bits in the instruction register is shown in Figure 14-5.

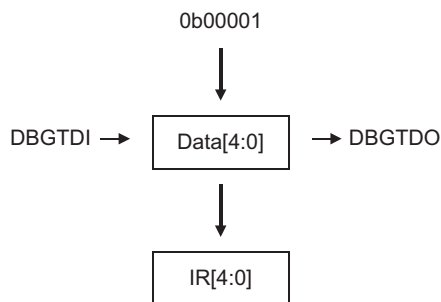


Figure 14-5 Instruction register bit order

14.6.4 Scan chain select register (SCREG)

Purpose Holds the currently active scan chain number.

Length 5 bits.

Operating mode After Scan_N has been selected as the current instruction, when in Shift-DR state, the shift section of the scan chain select register is selected as the serial path between **DBGTDI** and **DBGTDO**. At the Capture-DR state, the binary value b10000 is loaded into this shift section. This is shifted out during Shift-DR (least significant bit first), while a new value is shifted in (least significant bit first). At the Update-DR state, the value in the shift section is loaded into the Scan Chain Select Register to become the current active scan chain. All further instructions such as INTEST then apply to that scan chain. The currently selected scan chain only changes when

a Scan_N or ITRsel instruction is executed, or a DBGTAP reset occurs. On DBGTAP reset, scan chain 3 is selected as the active scan chain.

Order

The order of bits in the scan chain select register is shown in Figure 14-6.

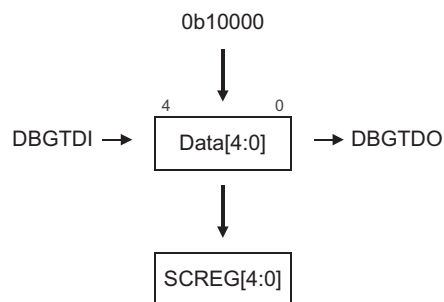


Figure 14-6 Scan chain select register bit order

14.6.5 Scan chains

To access the debug scan chains you must:

1. Load the Scan_N instruction into the IR. Now SCREG is selected between **DBGTDI** and **DBGTDO**.
2. Load the number of the desired scan chain. For example, load b00101 to access scan chain 5.
3. Load either INTEST or EXTEST into the IR.
4. Go through the DR leg of the DBGTAPSM to access the scan chain.

You must use INTEST and EXTEST as follows:

INTEST Use INTEST for reading the active scan chain. Data is captured into the shift register at the Capture-DR state. The previous value of the scan chain is shifted out during the Shift-DR state, while a new value is shifted in. The scan chain is not updated during Update-DR. Those bits or fields that are defined as cleared on read are only cleared if INTEST is selected, even when EXTEST also captures their values.

EXTEST Use EXTEST for writing the active scan chain. Data is captured into the shift register at the Capture-DR state. The previous value of the scan chain is shifted out during the Shift-DR state, while a new value is shifted in. The scan chain is updated with the new value during Update-DR.

Scan chain 0, debug ID register (DIDR)

Purpose Debug.

Length 8 + 32 = 40 bits.

Description Debug identification. This scan chain accesses CP14 debug register 0, the debug ID register. Additionally, the eight most significant bits of this scan chain contain an implementor code. This field is hardwired to 0x41, the implementor code for ARM Limited, as specified in the *ARM Architecture Reference Manual*. This register is read-only. Therefore, EXTEST has the same effect as INTEST.

Order The order of bits in scan chain 0 is shown in Figure 14-7.

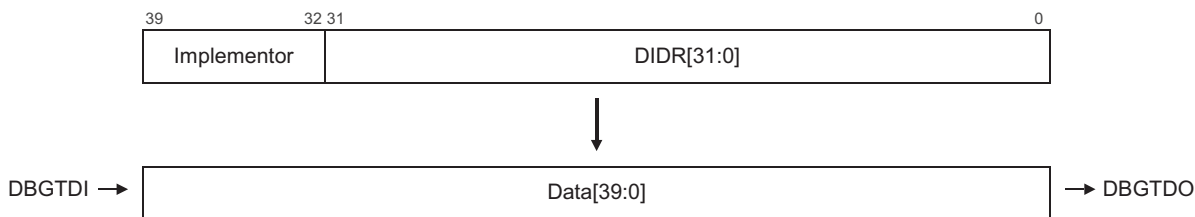


Figure 14-7 Scan chain 0 bit order

Scan chain 1, debug status and control register (DSCR)

Purpose Debug.

Length 32 bits.

Description This scan chain accesses CP14 register 1, the DSCR. This is mostly a read/write register, although certain bits are read-only for the Debug Test Access Port. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10 for details of DSCR bit definitions, and for read/write attributes for each bit. Those bits defined as cleared on read are only cleared if INTEST is selected.

Order The order of bits in scan chain 1 is shown in Figure 14-8 on page 14-13.



Figure 14-8 Scan chain 1 bit order

The following DSCR bits affect the operation of other scan chains:

- DSCR[30:29]** rDTRfull and wDTRfull flags. These indicate the status of the rDTR and wDTR registers. They are copies of the rDTRempty (NOT rDTRfull) and wDTRfull bits that the DBGTAP debugger sees in scan chain 5.
- DSCR[13]** Execute ARM instruction enable bit. This bit enables the mechanism used for executing instructions in debug state. It changes the behavior of the rDTR and wDTR registers, the sticky precise Data Abort bit, rDTRempty, wDTRfull, and InstCompl flags. See *Scan chain 5* on page 14-15.
- DSCR[6]** Sticky precise Data Abort flag. If the core is in debug state and the DSCR[13] execute ARM instruction enable bit is HIGH, then this flag is set on precise Data Aborts. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10.

———— **Note** ————

Unlike DSCR[6], DSCR [7] sticky imprecise Data Aborts flag does not affect the operation of the other scan chains.

Scan chain 4, instruction transfer register (ITR)

- Purpose** Debug
- Length** 1 + 32 = 33 bits

Description This scan chain accesses the *Instruction Transfer Register (ITR)*, used to send instructions to the core through the *Prefetch Unit (PU)*. It consists of 32 bits of information, plus an additional bit to indicate the completion of the instruction sent to the core (InstCompl). The InstCompl bit is read-only.

While in debug state, an instruction loaded into the ITR can be issued to the core by making the DBGTAPSM go through the Run-Test/Idle state. The InstCompl flag is cleared when the instruction is issued to the core and set when the instruction completes.

For an instruction to be issued when going through Run-Test/Idle state, you must ensure the following conditions are met:

- The processor must be in debug state.
- The DSCR[13] execute ARM instruction enable bit must be set. For details of the DSCR see *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10.
- Scan chain 4 or 5 must be selected.
- INTEST or EXTEST must be selected.
- Ready flag must be captured set. That is, the last time the DBGTAPSM went through Capture-DR the InstCompl flag must have been set.
- The DSCR[6] sticky precise Data Abort flag must be clear. This flag is set on precise Data Aborts.

For an instruction to be loaded into the ITR when going through Update-DR, you must ensure the following conditions are met:

- The processor can be in any state.
- The value of DSCR[13] execute ARM instruction enable bit does not matter.
- Scan chain 4 must be selected.
- EXTEST must be selected.
- Ready flag must be captured set. That is, the last time the DBGTAPSM went through Capture-DR the InstCompl flag must have been set.
- The value of DSCR[6] sticky precise Data Abort flag does not matter.

Order The order of bits in scan chain 4 is shown in Figure 14-9 on page 14-15.

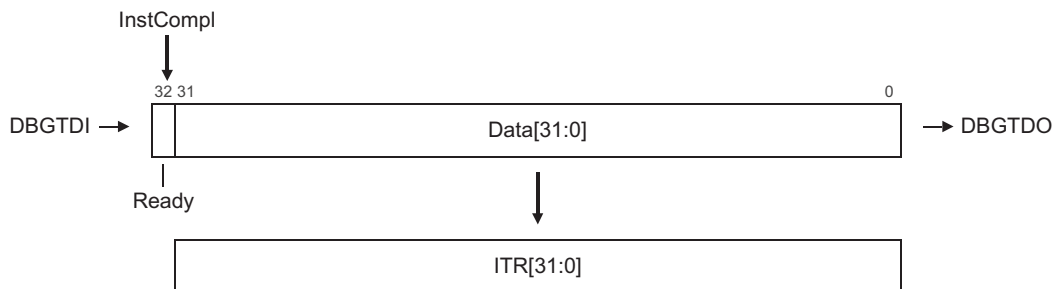


Figure 14-9 Scan chain 4 bit order

It is important to distinguish between the InstCompl flag and the Ready flag:

- The InstCompl flag signals the completion of an instruction.
- The Ready flag is the captured version of the InstCompl flag, captured at the Capture-DR state. The Ready flag conditions the execution of instructions and the update of the ITR.

The following points apply to the use of scan chain 4:

- When an instruction is issued to the core in debug state, the PC is not incremented. It is only changed if the instruction being executed explicitly writes to the PC. For example, branch instructions and move to PC instructions.
- If CP14 debug register c5 is a source register for the instruction to be executed, the DBGTap debugger must set up the data in the rDTR before issuing the coprocessor instruction to the core. See *Scan chain 5*.
- Setting DSCR[13] the execute ARM instruction enable bit when the core is not in debug state leads to Unpredictable behavior.
- The ITR is write-only. When going through the Capture-DR state, an Unpredictable value is loaded into the shift register.

Scan chain 5

Purpose Debug.

Length $1 + 1 + 32 = 34$ bits.

Description This scan chain accesses CP14 register c5, the data transfer registers, rDTR and wDTR. The rDTR is used to transfer words from the DBGTap debugger to the core, and is read-only to the core and write-only to the

DBGTAP debugger. The wDTR is used to transfer words from the core to the DBGTAP debugger, and is read-only to the DBGTAP debugger and write-only to the core.

The DBGTAP controller only sees one (read/write) register through scan chain 5, and the appropriate register is chosen depending on the instruction used. INTEST selects the wDTR, and EXTEST selects the rDTR.

Additionally, scan chain 5 contains some status flags. These are nRetry, Valid, and Ready, which are the captured versions of the rDTRempty, wDTRfull, and InstCompl flags respectively. All are captured at the Capture-DR state.

Order

The order of bits in scan chain 5 with EXTEST selected is shown in Figure 14-10. The order of bits in scan chain 5 with INTEST selected is shown in Figure 14-11 on page 14-17.

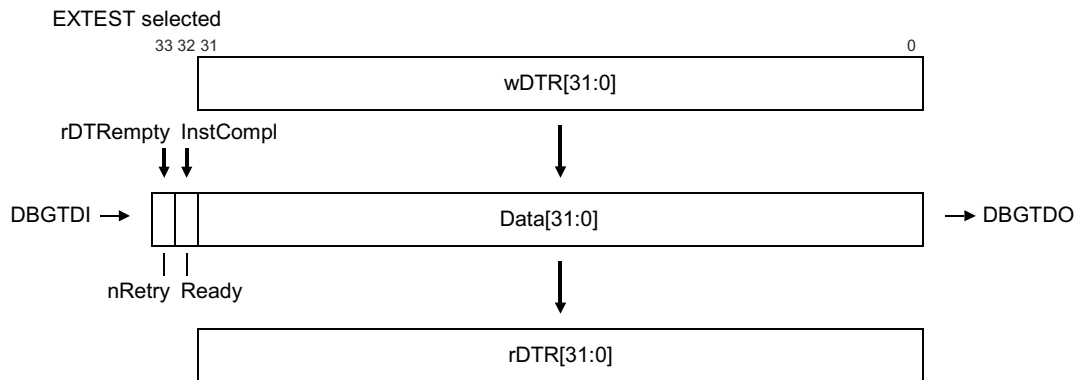


Figure 14-10 Scan chain 5 bit order, EXTEST selected

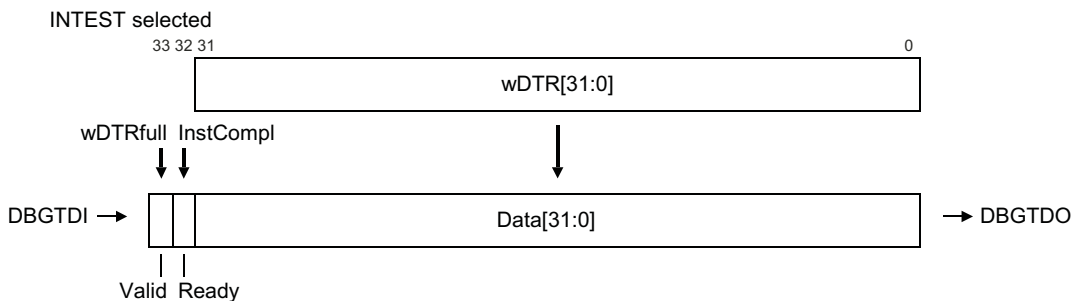


Figure 14-11 Scan chain 5 bit order, INTEST selected

You can use scan chain 5 for two purposes:

- As part of the *Debug Communications Channel* (DCC). The DBGTAP debugger uses scan chain 5 to exchange data with software running on the core. The software accesses the rDTR and wDTR using coprocessor instructions.
- For examining and modifying the processor state while the core is halted. For example, to read the value of an ARM register:
 1. Issue a MCR cp14, 0, Rd, c0, c5, 0 instruction to the core to transfer the register contents to the CP14 debug c5 register.
 2. Scan out the wDTR.

The DBGTAP debugger can use the DSCR[13] execute ARM instruction enable bit to indicate to the core that it is going to use scan chain 5 as part of the DCC or for examining and modifying the processor state. DSCR[13] = 0 indicates DCC use. The behavior of the rDTR and wDTR registers, the sticky precise Data Abort, rDTRempty, wDTRfull, and InstCompl flags changes accordingly:

- DSCR[13] = 0:
 - The wDTRfull flag is set when the core writes a word of data to the DTR and cleared when the DBGTAP debugger goes through the Capture-DR state with INTEST selected. Valid indicates the state of the wDTR register, and is the captured version of wDTRfull. Although the value of wDTR is captured into the shift register, regardless of INTEST or EXTEST, wDTRfull is only cleared if INTEST is selected.
 - The rDTR empty flag is cleared when the DBGTAP debugger writes a word of data to the rDTR, and set when the core reads it. nRetry is the captured version of rDTRempty.

- rDTR overwrite protection is controlled by the nRetry flag. If the nRetry flag is sampled clear, meaning that the rDTR is full, when going through the Capture-DR state, then the rDTR is not updated at the Update-DR state.
- The InstCompl flag is always set.
- The sticky precise Data Abort flag is Unpredictable. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-10.
- DSCR[13] = 1:
 - The wDTR Full flag behaves as if DSCR[13] is clear. However, the Ready flag can be used for handshaking in this mode.
 - The rDTR Empty flag status behaves as if DSCR[13] is clear. However, the Ready flag can be used for handshaking in this mode.
 - rDTR overwrite protection is controlled by the Ready flag. If the InstCompl flag is sampled clear when going through Capture-DR, then the rDTR is not updated at the Update-DR state. This prevents an instruction that uses the rDTR as a source operand from having it modified before it has time to complete.
 - The InstCompl flag changes from 1 to 0 when an instruction is issued to the core, and from 0 to 1 when the instruction completes execution.
 - The sticky precise Data Abort flag is set on precise Data Aborts.

The behavior of the rDTR and wDTR registers, the sticky precise Data Abort, rDTRempty, wDTRfull, and InstCompl flags when the core changes state is as follows:

- The DSCR[13] execute ARM instruction enable bit must be clear when the core is not in debug state. Otherwise, the behavior of the rDTR and wDTR registers, and the flags, is Unpredictable.
- When the core enters debug state, none of the registers and flags are altered.
- When the DSCR[13] execute ARM instruction enable bit is changed from 0 to 1:
 1. None of the registers and flags are altered.
 2. Ready flag can be used for handshaking.
- The InstCompl flag must be set when the DSCR[13] execute ARM instruction enable bit is changed from 1 to 0. Otherwise, the behavior of the core is Unpredictable. If the DSCR[13] flag is cleared correctly, none of the registers and flags are altered.
- When the core leaves debug state, none of the registers and flags are altered.

Scan chain 6

Purpose Embedded Trace Macrocell.

Length $1 + 7 + 32 = 40$ bits.

Description This scan chain accesses the register map of the Embedded Trace Macrocell. See the description in the programmer's model chapter in the *Embedded Trace Macrocell Specification* for details of register allocation.

Order The order of bits in scan chain 6 is shown in Figure 14-12.

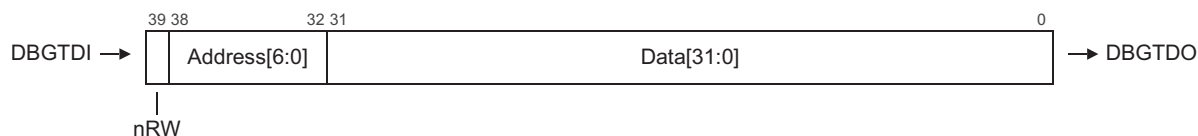


Figure 14-12 Scan chain 6 bit order

Scan chain 7

Purpose Debug.

Length $7 + 32 + 1 = 40$ bits.

Description Scan chain 7 accesses the VCR, PC, BRPs, and WRPs. The accesses are performed with the help of read or write request commands. A read request copies the data held by the addressed register into scan chain 7. A write request copies the data held by the scan chain into the addressed register. When a request is finished the ReqCompl flag is set. The DBGTAP debugger must poll it and check it is set before another request can be issued.

The exact behavior of the scan chain is as follows:

- Either EXTEST or INTEST have to be selected. They have the same meaning in this scan chain.
- If the value captured by the Ready/nRW bit at the Capture-DR state is 1, the data that is being shifted in generates a request at the Update-DR state. The Address field indicates the register being accessed (see Table 14-2 on page 14-21), the Data field contains

the data to be written and the Ready/nRW bit holds the read/write information (0=read and 1=write). If the request is a read, the Data field is ignored.

- When a request is placed, the Address and Data sections of the scan chain are frozen. That is, their contents are not shifted until the request is completed. This means that, if the value captured in the Ready/nRW field at the Capture-DR state is 0, the shifted-in data is ignored and the shifted-out value is all 0s.
- After a read request has been placed, if the DBGTAPSM goes through the Capture-DR state and a logic 1 is captured in the Ready/nRW field, this means that the shift register has also captured the requested register contents. Therefore, they are shifted out at the same time as the Ready/nRW bit. The Data field is corrupted as new data is shifted in.
- After a write request has been placed, if the DBGTAPSM goes through the Capture-DR state and a logic 1 is captured in the Ready/nRW field, this means that the requested write has completed successfully.
- If the Address field is all 0s (address of the NULL register) at the Update-DR state, then no request is generated.
- A request to a reserved register generates Unpredictable behavior.

Order The order of bits in scan chain 7 is shown in Figure 14-13.

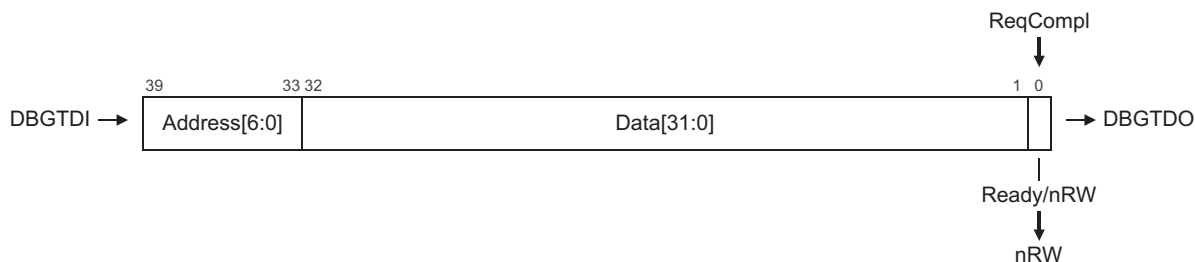


Figure 14-13 Scan chain 7 bit order

A typical sequence for writing registers is as follows:

1. Scan in the address of a first register, the data to write, and a 1 to indicate that this is a write request.

2. Scan in the address of a second register, the data to write, and a 1 to indicate that this is a write request.

Scan out 40 bits. If Ready/nRW is 0 repeat this step. If Ready/nRW is 1, the first write request has completed successfully and the second has been placed.

3. Scan in the address 0. The rest of the fields are not important.

Scan out 40 bits. If Ready/nRW is 0 repeat this step. If Ready/nRW is 1, the second write request has completed successfully. The scanned-in null request has avoided the generation of another request.

A typical sequence for reading registers is as follows:

1. Scan in the address of a first register and a 0 to indicate that this is a read request. The Data field is not important.

2. Scan in the address of a second register and a 0 to indicate that this is a read request.

Scan out 40 bits. If Ready/nRW is 0 then repeat this step. If Ready/nRW is 1, the first read request has completed successfully and the next scanned-out 32 bits are the requested value. The second read request was placed at the Update-DR state.

3. Scan in the address 0 (the rest of the fields are not important).

Scan out 40 bits. If Ready/nRW is 0 then repeat this step. If Ready/nRW is 1, the second read request has completed successfully and the next scanned-out 32 bits are the requested value. The scanned-in null request has avoided the generation of another request.

The register map is similar to the one of CP14 debug, and is shown in Table 14-2.

Table 14-2 Scan chain 7 register map

Address[6:0]	Register number	Abbreviation	Register name
b0000000	0	NULL	No request register
b0000001-b0000110	1-6	-	Reserved
b0000111	7	VCR	Vector catch register
b0001000	8	PC	Program counter
b0010011-b0111111	19-63	-	Reserved
b1000000-b1000101	64-69	BVRy ^a	Breakpoint value registers
b1000110-b1001111	70-79	-	Reserved

Table 14-2 Scan chain 7 register map (continued)

Address[6:0]	Register number	Abbreviation	Register name
b1010000-b1010101	80-85	BCR _y ^a	Breakpoint control registers
b1010110-b1011111	86-95	-	Reserved
b1100000-b1100001	96-97	WVR _y ^a	Watchpoint value registers
b1100010-b1011111	98-111	-	Reserved
b1110000-b1110001	112-113	WCR _y ^a	Watchpoint control registers
b1110010-b1111111	114-127	-	Reserved

a. _y is the decimal representation for the binary number Address[3:0]

The following points apply to the use of scan chain 7:

- Every time there is a request to read the PC, a sample of its value is copied into scan chain 7. Writes are ignored. The sampled value can be used for profiling of the code. See *Interpreting the PC samples* for details of how to interpret the sampled value.
- When accessing registers using scan chain 7, the processor can be either in debug state or in normal state. This implies that breakpoints, watchpoints, and vector traps can be programmed through the Debug Test Access Port even if the processor is running. However, although a PC read can be requested in debug state, the result is Undefined.

Interpreting the PC samples

The PC values read correspond to instructions committed for execution, including those that failed their condition code. However, these values are offset as described in Table 13-15 on page 13-30. These offsets are different for different processor states, so additional information is required:

- If a read request to the PC completes and Data[1:0] equals b00, the read value corresponds to an ARM state instruction whose 30 most significant bits of the offset address (instruction address + 8) are given in Data[31:2].
- If a read request to the PC completes and Data[0] equals b1, the read value corresponds to a Thumb state instruction whose 31 most significant bits of the offset address (instruction address + 4) are given in Data[31:1].

- If a read request to the PC completes and Data[1:0] equals b10, the read value corresponds to a Java state instruction whose 30 most significant bits of its address are given in Data[31:2] (the offset is 0). Because of the state encoding, the lower two bits of the Java address are not sampled. However, the information provided is enough for profiling the code.
- If the PC is read while the processor is in Debug state, the result is Unpredictable.

Scan chains 8-15

These scan chains are reserved.

Scan chains 16-31

These scan chains are unassigned.

14.6.6 Reset

The DBGTAP is reset either by asserting **DBGnTRST**, or by clocking it while DBGTAPSM is in the Test-Logic-Reset state. The processor, including CP14 debug logic, is not affected by these events. See *Reset modes* on page 9-8 and *CP14 registers reset* on page 13-24 for details.

14.7 Using the Debug Test Access Port

This section contains the following subsections:

- *Entering and leaving debug state*
- *Executing instructions in debug state*
- *Using the ITRsel IR instruction on page 14-25*
- *Transferring data between the host and the core on page 14-27*
- *Using the debug communications channel on page 14-27*
- *Target to host debug communications channel sequence on page 14-28*
- *Host to target debug communications channel on page 14-29*
- *Transferring data in debug state on page 14-29*
- *Example sequences on page 14-30.*

14.7.1 Entering and leaving debug state

These debug sequences are described in detail in *Debug sequences* on page 14-34.

14.7.2 Executing instructions in debug state

When the ARM1136 JF-S processor is in debug state, it can be forced to execute ARM state instructions using the DBGTAP. Two registers are used for this purpose, the *Instruction Transfer Register (ITR)* and the *Data Transfer Register (DTR)*.

The ITR is used to insert an instruction into the processor pipeline. An ARM state instruction can be loaded into this register using scan chain number 4. When the instruction is loaded, and INTEST or EXTEST is selected, and scan chain 4 or 5 is selected, the instruction can be issued to the core by making the DBGTAPSM go through the Run-Test/Idle state, provided certain conditions are met (described in this section). This mechanism enables re-executing the same instruction over and over without having to reload it.

The DTR can be used in conjunction with the ITR to transfer data in and out of the core. For example, to read out the value of an ARM register:

1. issue an MCR p14, 0, Rd, c0, c5, 0 instruction to the core to transfer the <Rd> contents to the c5 register
2. scan out the wDTR.

The DSCR[13] execute ARM instruction enable bit controls the activation of the ARM instruction execution mechanism. If this bit is cleared, no instruction is issued to the core when the DBGTAPSM goes through Run-Test/Idle. Setting this bit while the core is not in debug state leads to Unpredictable behavior. If the core is in debug state and

this bit is set, the Ready and the sticky precise Data Abort flags condition the updates of the ITR and the instruction issuing as described in *Scan chain 4, instruction transfer register (ITR)* on page 14-13.

As an example, this sequence stores out the contents of the ARM register r0:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This action clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. EXTEST into the IR.
6. Scan in the previously read value with the DSCR[13] execute ARM instruction enable bit set.
7. Scan_N into the IR.
8. 4 into the SCREG.
9. EXTEST into the IR.
10. Scan the MCR p14,0,R0,c0,c5,0 instruction into the ITR.
11. Go through the Run-Test/Idle state of the DBGTAPSM.
12. Scan_N into the IR.
13. 5 into the SCREG.
14. INTEST into the IR.
15. Scan out 34 bits. The 33rd bit indicates if the instruction has completed. If the bit is clear, repeat this step again.
16. The least significant 32 bits hold the contents of r0.

14.7.3 Using the ITRsel IR instruction

When the ITRsel instruction is loaded into the IR, at the Update-IR state, the DBGTAP controller behaves as if EXTEST and scan chain 4 are selected, but SCREG retains its value. It can be used to speed up certain debug sequences.

Figure 14-14 on page 14-26 shows the effect of the ITRsel IR instruction.

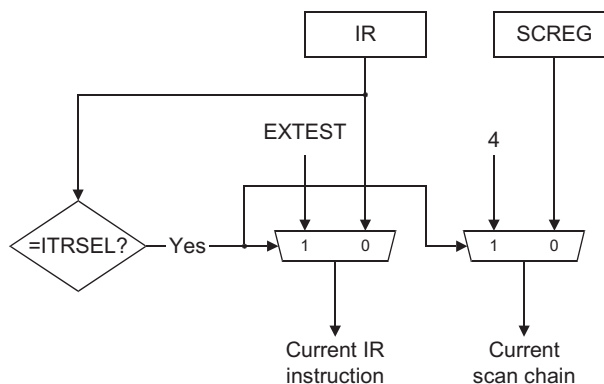


Figure 14-14 Behavior of the ITRsel IR instruction

Consider for example the preceding sequence to store out the contents of ARM register r0. This is the same sequence using the ITRsel instruction:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This action clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. EXTEST into the IR.
6. Scan in the previously read value with the DSCR[13] execute ARM instruction enable bit set.
7. Scan_N into the IR.
8. 5 into the SCREG.
9. ITRsel into the IR. Now the DBG TAP controller works as if EXTEST and scan chain 4 is selected.
10. Scan the MCR p14,0,R0,c0,c5,0 instruction into the ITR.
11. Go through the Run-Test/Idle state of the DBG TAPSM.
12. INTEST into the IR. Now INTEST and scan chain 5 are selected.

13. Scan out 34 bits. The 33rd bit indicates if the instruction has completed. If the bit is clear, repeat this step again.
14. The least significant 32 bits hold the contents of r0.

The number of steps has been reduced from 16 to 14. However, the bigger reduction comes when reading additional registers. Using the ITRsel instruction there are 6 extra steps (9 to 14), compared with 10 extra steps (7 to 16) in the first sequence.

14.7.4 Transferring data between the host and the core

There are two ways in which a DBGTAP debugger can send or receive data from the core:

- using the DCC, when the ARM1136JF-S processor is not in debug state
- using the instruction execution mechanism described in *Executing instructions in debug state* on page 14-24, when the core is in debug state.

This is described in:

- *Using the debug communications channel.*
- *Target to host debug communications channel sequence* on page 14-28
- *Host to target debug communications channel* on page 14-29
- *Transferring data in debug state* on page 14-29
- *Example sequences* on page 14-30.

14.7.5 Using the debug communications channel

The DCC is defined as the set of resources that the external DBGTAP debugger uses to communicate with a piece of software running on the core.

The DCC in the ARM1136JF-S processor is implemented using the two physically separate DTRs and a full/empty bit pair to augment each register, creating a bidirectional data port. One register can be read from the DBGTAP and is written from the processor. The other register is written from the DBGTAP and read by the processor. The full/empty bit pair for each register is automatically updated by the debug unit hardware, and is accessible to both the DBGTAP and to software running on the processor.

At the core side, the DCC resources are the following:

- CP14 debug register c5 (DTR). Data coming from a DBGTAP debugger can be read by an MRC or STC instruction addressed to this register. The core can write to this register any data intended for the DBGTAP debugger, using an MCR or

LDC instruction. Because the DTR comprises both a read (rDTR) and a write portion (wDTR), a piece of data written by the core and another coming from the DBGTAP debugger can be held in this register at the same time.

- Some flags and control bits at CP14 debug register c1 (DSCR):

DSCR[12]	User mode access to DCC disable bit. If this bit is set, only privileged software can access the DCC. That is, access the DSCR and the DTR.
DSCR[29]	The wDTRfull flag. When clear, this flag indicates to the core that the wDTR is ready to receive data from the core.
DSCR[30]	The rDTRfull flag. When set, this flag indicates to the core that there is data available to read at the DTR.

At the DBGTAP side, the resources are the following:

- Scan chain 5 (see *Scan chain 5* on page 14-15). The only part of this scan chain that it is not used for the DCC is the Ready flag. The rest of the scan chain is to be used in the following way:

rDTR	When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 5 selected, and the nRetry flag set, the contents of the Data field are loaded into the rDTR. This is how the DBGTAP debugger sends data to the software running on the core.
wDTR	When the DBGTAPSM goes through the Capture-DR state with INTEST and scan chain 5 selected, the contents of the wDTR are loaded into the Data field of the scan chain. This is how the DBGTAP debugger reads the data sent by the software running on the core.
Valid flag	When set, this flag indicates to the DBGTAP debugger that the contents of the wDTR that it has just captured are valid.
nRetry flag	When set, this flag indicates to the DBGTAP debugger that the scanned-in Data field has been successfully written into the rDTR at the Update-DR state.

14.7.6 Target to host debug communications channel sequence

The DBGTAP debugger can use the following sequence for receiving data from the core:

1. Scan_N into the IR.
2. 5 into the SCREG.

3. INTEST into the IR.
4. Scan out 34 bits of data. If the Valid flag is clear repeat this step again.
5. The least significant 32 bits hold valid data.
6. Go to step 4 again for reading out more data.

14.7.7 Host to target debug communications channel

The DBGTAP debugger can use the following sequence for sending data to the core:

1. Scan_N into the IR.
2. 5 into the SCREG.
3. EXTEST into the IR.
4. Scan in 34 bits, the least significant 32 holding the word to be sent. At the same time, 34 bits were scanned out. If the nRetry flag is clear repeat this step again.
5. Now the data has been written into the rDTR. Go to step 4 again for sending in more data.

14.7.8 Transferring data in debug state

When the core is in debug state, the DBGTAP debugger can transfer data in and out of the core using the instruction execution facilities described in *Executing instructions in debug state* on page 14-24 in addition to scan chain 5. You must ensure that the DSCR[13] execute ARM instruction enable bit is set for the instruction execution mechanism to work. When it is set, the interface for the DBGTAP debugger consists of the following:

- Scan chain 4 (see *Scan chain 4, instruction transfer register (ITR)* on page 14-13). It is used for loading an instruction and for monitoring the status of the execution:

ITR When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 4 selected, and the Ready flag set, the ITR is loaded with the least significant 32 bits of the scan chain.

InstCompl flag When clear, this flag indicates to the DBGTAP debugger that the last issued instruction has not yet completed execution. While Ready (captured version of InstCompl) is clear, no updates of the ITR and the rDTR occur and the instruction execution mechanism is disabled. No instruction is issued when going through Run-Test/Idle.

- Scan chain 5 (see *Scan chain 5* on page 14-15). It is used for writing in or reading out the data and for monitoring the state of the execution:

rDTR When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 5 selected, and the Ready flag set, the contents of the Data field are loaded into the rDTR.

wDTR When the DBGTAPSM goes through the Capture-DR state with INTEST or EXTEST selected, the contents of the wDTR are loaded into the Data field of the scan chain.

InstCompl flag When clear, this flag indicates to the DBGTAP debugger that the last issued instruction has not yet completed execution. While Ready (captured version of InstCompl) is clear, no updates of the ITR and the rDTR occur and the instruction execution mechanism is disabled. No instruction is issued when going through Run-Test/Idle.

- Some flags and control bits at CP14 debug register c1 (DSCR):

DSCR[13] Execute ARM instruction enable bit. This bit must be set for the instruction execution mechanism to work.

Sticky precise Data Abort flag

DSCR[6]. When set, this flag indicates to the DBGTAP debugger that a precise Data Abort occurred while executing an instruction in debug state. While this bit is set, the instruction execution mechanism is disabled. When this flag is set InstCompl stays HIGH, and additional attempts to execute an instruction appear to succeed but do not execute.

Sticky imprecise Data Abort flag

DSCR[7]. When set, this flag indicates to the DBGTAP debugger that an imprecise Data Abort occurred while executing an instruction in debug state. This flag does not disable the debug state instruction execution.

14.7.9 Example sequences

This section includes some example sequences to illustrate how to transfer data between the DBGTAP debugger and the core when it is in debug state. The examples are related to accessing the processor memory.

Target to host transfer

The DBGTAP debugger can use the following sequence for reading data from the processor memory system. The sequence assumes that the ARM register r0 contains a pointer to the address of memory at which the read has to start:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. Scan_N into the IR.
6. 4 into the SCREG.
7. EXTEST into the IR.
8. Scan in the LDC p14, c5, [R0], #4 instruction into the ITR.
9. Scan_N into the IR.
10. 5 into the SCREG.
11. INTEST into the IR.
12. Go through Run-Test/Idle state. The instruction loaded into the ITR is issued to the processor pipeline.
13. Scan out 34 bits of data. If the Ready flag is clear repeat this step again.
14. The instruction has completed execution. Store the least significant 32 bits.
15. Go to step 12 again for reading out more data.
16. Scan_N into the IR.
17. 1 into the SCREG.
18. INTEST into the IR.
19. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be read, and after the cause for the abort has been fixed the sequence resumes at step 5.

Note

If the sticky imprecise Data Aborts flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and r0 is reloaded.

Host to target transfer

The DBGTAP debugger can use the following sequence for writing data to the processor memory system. The sequence assumes that the ARM register r0 contains a pointer to the address of memory at which the write has to start:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. Scan_N into the IR.
6. 4 into the SCREG.
7. EXTEST into the IR.
8. Scan in the STC p14,c5,[R0],#4 instruction into the ITR.
9. Scan_N into the IR.
10. 5 into the SCREG.
11. EXTEST into the IR.
12. Scan in 34 bits, the least significant 32 holding the word to be sent. At the same time, 34 bits are scanned out. If the Ready flag is clear, repeat this step.
13. Go through Run-Test/Idle state.
14. Go to step 12 again for writing in more data.
15. Scan in 34 bits. All the values are don't care. At the same time, 34 bits are scanned out. If the Ready flag is clear, repeat this step. The don't care value is written into the rDTR (Update-DR state) just after Ready is seen set (Capture-DR state). However, the STC instruction is not re-issued because the DBGTAPSM does not go through Run-Test/Idle.

16. Scan_N into the IR.
17. 1 into the SCREG.
18. INTEST into the IR.
19. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be written, and after the cause for the abort has been fixed the sequences resumes at step 5.

———— **Note** —————

If the sticky imprecise Data Abort flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and c0 is reloaded.

14.8 Debug sequences

This section describes how to debug a program running on the ARM1136JF-S processor using a DBG TAP debugger device such as RealView ICE.

In Halt mode, the processor stops when a debug event occurs enabling the DBG TAP debugger to do the following:

1. Determine and modify the current state of the processor and memory.
2. Set up breakpoints, watchpoints, and vector traps.
3. Restart the processor.

You enable this mode by setting CP14 debug DSCR[14] bit, which can only be done by the DBG TAP debugger.

From here it is assumed that the debug unit is in Halt mode. Monitor mode debugging is described in *Monitor mode debugging* on page 14-50.

14.8.1 Debug macros

The debug code sequences in this section are written using a fixed set of macros. The mapping of each macro into a debug scan chain sequence is given in this section.

Scan_N <n>

Select scan chain register number <n>:

1. Scan the Scan_N instruction into the IR.
2. Scan the number <n> into the DR.

INTEST

1. Scan the INTEST instruction into the IR.

EXTEST

1. Scan the EXTEST instruction into the IR.

ITRsel

1. Scan the ITRsel instruction into the IR.

Restart

1. Scan the Restart instruction into the IR.
2. Go to the DBGTAP controller Run-Test/Idle state so that the processor exits debug state.

INST <instr> [stateout]

Go through Capture-DR, go to Shift-DR, scan in an ARM instruction to be read and executed by the core and scan out the Ready flag, go through Update-DR. The ITR (scan chain 4) and EXTEST must be selected when using this macro.

1. Scan in:
 - Any value for the InstCompl flag. This bit is read-only.
 - 32-bit assembled code of the instruction (instr) to be executed, for ITR[31:0].
2. The following data is scanned out:
 - The value of the Ready flag, to be stored in stateout.
 - 32 bits to be ignored. The ITR is write-only.

DATA <datain> [<stateout> [dataout]]

Go through Capture-DR, go to Shift-DR. Scan in a data item and scan out another one, go through Update-DR. Either the DTR (scan chain 5) or the DSCR (scan chain 1) must be selected when using this macro.

1. If scan chain 5 is selected, scan in:
 - Any value for the nRetry or Valid flag. These bits are read-only.
 - Any value for the InstCompl flag. This bit is read-only.
 - 32-bit datain value for rDTR[31:0].
2. The following data is scanned out:
 - The contents of wDTR[31:0], to be stored in dataout.
 - If the DSCR[13] execute ARM instruction enable bit is set, the value of the Ready flag is stored in stateout.
 - If the DSCR[13] execute ARM instruction enable bit is clear, the nRetry or Valid flag (depending on whether EXTEST or INTEST is selected) is stored in stateout.

3. If scan chain 1 is selected, scan in:
 - 32-bit datain value for DSCR[31:0].Stateout and dataout fields are not used in this case.

DATAOUT <dataout>

1. Scan out a data value. DSCR (scan chain 1) and INTEST must be selected when using this macro.
2. If scan chain 1 is selected, scan out the contents of the DSCR, to be stored in dataout.
3. The scanned-in value is discarded, because INTEST is selected.

REQ <address> <data> <nR/W> [<stateout> [dataout]]

Go through Capture-DR, go to Shift-DR, scan in a request and scan out the result of the former one, go through Update-DR. Scan chain 7, and either INTEST or EXTEST, must be selected when using this macro.

1. Scan in:
 - 7-bit address value for Address[6:0]
 - 32-bit data value for Data[31:0]
 - 1-bit nR/W value (0 for read and 1 for write) for the Ready/nRW field.
2. Scan out:
 - the value of the Ready/nRW bit, to be stored in stateout
 - the contents of the Data field, to be stored in dataout.

RTI

1. Go through Run-Test/Idle DBGTAPSM state. This forces the execution of the instruction currently loaded into the ITR, provided the execute ARM instruction enable bit (DSCR[13]) is set, the Ready flag was captured as set, and the sticky precise Data Abort flag is cleared.

14.8.2 General setup

You must setup the following control bits before DBGTAP debugging can take place:

- DSCR[14] Halt/Monitor mode bit must be set to 1. It resets to 0 on power-up.

- DSCR[6] sticky precise Data Abort flag must be cleared down, so that aborts are not detected incorrectly immediately after startup.

The DSCR must be read, the DSCR[14] bit set, and the new value written back. The action of reading the DSCR automatically clears the DSCR[6] sticky precise Data Abort flag.

All individual breakpoints, watchpoints, and vector catches reset disabled on power-up.

14.8.3 Forcing the processor to halt

Scan the Halt instruction into the DBGTAP controller IR and go through Run-Test/Idle.

14.8.4 Entering debug state

To enter debug state you must:

1. Check whether the core has entered debug state, as follows:

```
SCAN_N 1                ; select DSCR
INTEST
LOOP
    DATAOUT readDSCR
UNTIL  readDSCR[0]==1    ; until Core Halted bit is set
```

2. Save DSCR, as follows:

```
DATAOUT readDSCR
Save value in readDSCR
```

3. Save wDTR (in case it contains some data), as follows:

```
SCAN_N 5                ; select DTR
INTEST
DATA    0x00000000 Valid wDTR
If Valid==1 then Save value in wDTR
```

4. Set the DSCR[13] execute ARM instruction enable bit, so instructions can be issued to the core from now:

```
SCAN_N 1                ; select DSCR
EXTEST
DATA modifiedDSCR        ; modifiedDSCR equals readDSCR with bit
                        ; DSCR[13] set
```

5. Before executing any instruction in debug state you have to drain the write buffer. This ensures that no imprecise Data Aborts can return at a later point:

```
SCAN_N 4                ; select DTR
INST  MRC p14,0,Rd,c5,c10,0 ; drain write buffer
LOOP
```

```

        LOOP
            SCAN_N 4                ; select DTR
            RTI
            INST 0x0 Ready
        Until Ready == 1
        SCAN_N 1
        DATAOUT readDSCR
        Until readDSCR[7]==1
        SCAN_N 4
        INST NOP                    ;NOP takes the
        RTI                          ;imprecise Data Aborts
        LOOP
            INST 0 Ready
        Until Ready == 1
        SCAN_N 1
        DATAOUT readDSCR            ;clears DSCR[7]

```

6. Store out r0. It is going to be used to save the rDTR. Use the standard sequence of *Reading a current mode ARM register in the range r0-r14* on page 14-40. Scan chain 5 and INTEST are now selected.
7. Save the rDTR and the rDTRempty bit in three steps:
 - a. The rDTRempty bit is the inverted version of DSCR[30] (saved in step 2). If DSCR[30] is clear (register empty) there is no requirement to read the rDTR, go to 7.
 - b. Transfer the contents of rDTR to r0:

```

                ITRSEL                ; select the ITR and EXTEST
                INST   MRC p14,0,R0,c0,c5,0    ; instruction to copy CP14's debug
                                                ; register c5 into R0

                RTI
                LOOP
                    INST 0x00000000 Ready
                UNTIL   Ready==1                ; wait until the instruction ends

```
 - c. Read r0 using the standard sequence of *Reading a current mode ARM register in the range r0-r14* on page 14-40.
8. Store out CPSR using the standard sequence of *Reading the CPSR/SPSR* on page 14-41.
9. Store out PC using the standard sequence of *Reading the PC* on page 14-42.
10. Adjust the PC to enable you to resume execution later:
 - subtract 0x8 from the stored value if the processor was in ARM state when entering debug state
 - subtract 0x4 from the stored value if the processor was in Thumb state when entering debug state

- subtract $0x0$ from the stored value if the processor was in Java state when entering debug state.

These values are not dependent on the debug state entry method, (see *Behavior of the PC in debug state* on page 13-35). The entry state can be determined by examining the T and J bits of the CPSR.

11. Cache and MMU preservation measures must also be taken here. This includes saving all the relevant CP15 registers using the standard coprocessor register reading sequence described in *Coprocessor register reads and writes* on page 14-46.

14.8.5 Leaving debug state

To leave debug state:

1. Restore standard ARM registers for all modes, except r0, PC, and CPSR.
2. Cache and MMU restoration must be done here. This includes writing the saved registers back to CP15.

3. Ensure that rDTR and wDTR are empty:

```
ITRSEL                ; select the ITR and EXTEST
INST    MCR p14,0,R0,c0,c5,0 ; instruction to copy R0 into
                                           ; CP14 debug register c5
```

```
RTI
LOOP
```

```
    INST 0x00000000 Ready
```

```
UNTIL Ready==1           ; wait until the instruction ends
```

```
SCAN_N 5
```

```
INTEST
```

```
DATA    0x0 Valid wDTR
```

4. If the wDTR did not contain any valid data on debug state entry go to step 5. Otherwise, restore wDTRfull and wDTR (uses r0 as a temporary register) in two steps.
 - a. Load the saved wDTR contents into r0 using the standard sequence of *Writing a current mode ARM register in the range r0-r14* on page 14-41. Now scan chain 5 and EXTEST are selected

- b. Transfer r0 into wDTR:

```
ITRSEL                ; select the ITR and EXTEST
INST    MCR p14,0,R0,c0,c5,0 ; instruction to copy R0 into
                                           ; CP14 debug register c5
```

```
RTI
LOOP
```

```
    INST 0x00000000 Ready
```

```
UNTIL Ready==1           ; wait until the instruction ends
```

5. Restore CPSR using the standard CPSR writing sequence described in *Writing the CPSR/SPSR* on page 14-42.
6. Restore the PC using the standard sequence of *Writing the PC* on page 14-42.
7. Restore r0 using the standard sequence of *Writing a current mode ARM register in the range r0-r14* on page 14-41. Now scan chain 5 and EXTEST are selected.
8. Restore the DSCR with the DSCR[13] execute ARM instruction enable bit clear, so no more instructions can be issued to the core:

```
SCAN_N 1                ; select DSCR
EXTEST
DATA modifiedDSCR      ; modifiedDSCR equals the saved contents
                        ; of the DSCR with bit DSCR[13] clear
```

9. If the rDTR did not contain any valid data on debug state entry go to step 10. Otherwise, restore the rDTR and rDTRempty flag:

```
SCAN_N 5                ; select DTR
EXTEST
DATA Saved_rDTR        ; rDTRempty bit is automatically cleared
                        ; as a result of this action
```

10. Restart processor:

```
RESTART
```

11. Wait until the core is restarted:

```
SCAN_N 1                ; select DSCR
INTEST
LOOP
    DATAOUT readDSCR
UNTIL readDSCR[1]==1    ; until Core Restarted bit is set
```

14.8.6 Reading a current mode ARM register in the range r0-r14

Use the following sequence to read a current mode ARM register in the range r0-r14:

```
SCAN_N 5                ; select DTR
ITRSEL                  ; select the ITR and EXTEST
INST MCR p14,0,Rd,c0,c5,0 ; instruction to copy Rd into CP14 debug
                        ; register c5

RTI
INTEST                  ; select the DTR and INTEST
LOOP
    DATA 0x00000000 Ready readData
UNTIL Ready==1          ; wait until the instruction ends
Save value in readData
```

Note

Register r15 cannot be read in this way because the effect of the required MCR is to take an Undefined exception.

14.8.7 Writing a current mode ARM register in the range r0-r14

Use the following sequence to write a current mode ARM register in the range r0-r14:

```

SCAN_N  5                               ; select DTR
ITRSEL                                  ; select the ITR and EXTEST
INST    MRC p14,0,Rd,c0,c5,0           ; instruction to copy CP14 debug
                                              ; register c5 into Rd
EXTEST                                  ; select the DTR and EXTEST
DATA    Data2Write
RTI
LOOP
    DATA 0x00000000 Ready
UNTIL   Ready==1                       ; wait until the instruction ends

```

Note

Register r15 cannot be written in this way because the MRC instruction used would update the CPSR flags rather than the PC.

14.8.8 Reading the CPSR/SPSR

Here r0 is used as a temporary register:

1. Move the contents of CPSR/SPSR to r0.

```

SCAN_N  5                               ; select DTR
ITRSEL                                  ; select the ITR and EXTEST
INST    MRS R0,CPSR                     ; or SPSR
RTI
LOOP
    INST 0x00000000 Ready
UNTIL   Ready==1                       ; wait until the instruction ends

```

2. Perform the read of r0 using the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 14-40. Scan chain 5 and ITRsel are already selected.

14.8.9 Writing the CPSR/SPSR

Here r0 is used as a temporary register:

1. Load the desired value into r0 using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 14-41. Now scan chain 5 and EXTEST are selected.
2. Move the contents of r0 to CPRS/SPRS:

```

ITRSEL                                ; select the ITR and EXTEST
INST  MSR CPSR,R0                     ; or SPSR
RTI
LOOP
    INST 0x00000000 Ready
UNTIL  Ready==1                       ; wait until the instruction ends

```

It is not a problem to write to the T and J bits because they have no effect in the execution of instructions while in Debug state.

The CPSR mode and control bits can be written in User mode when the core is in debug state. This is essential so that the debugger can change mode and then get at the other banked registers.

14.8.10 Reading the PC

Here r0 is used as a temporary register:

1. Move the contents of the PC to r0:


```

ITRSEL                                ; select the ITR and EXTEST
INST  MOV R0,PC
RTI
LOOP
    INST 0x00000000 Ready
UNTIL  Ready==1                       ; wait until the instruction ends

```
2. Read the contents of r0 using the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 14-40.

14.8.11 Writing the PC

Here r0 is used as a temporary register:

1. Load r0 with the address to resume using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 14-41. Now scan chain 5 and EXTEST are selected.
2. Move the contents of r0 to the PC:

```

ITRSEL                                ; select the ITR and EXTEST
INST  MOV PC,R0
RTI
LOOP
    INST 0x00000000 Ready
UNTIL  Ready==1                        ; wait until the instruction ends

```

14.8.12 General notes about reading and writing memory

On the ARM1136JF-S processor, an abort occurring in debug state causes an Abort exception entry sequence to start, and so changes mode to Abort mode, and writes to r14_abt and SPSR_abt. This means that the Abort mode registers must be saved before performing a debug state memory access.

The word-based read and write sequences are substantially more efficient than the halfword and byte sequences. This is because the ARM LDC and STC instructions always perform word accesses, and this can be used for efficient access to word width memory. Halfword and byte accesses must be done with a combination of loads or stores, and coprocessor register transfers, which is much less efficient.

When writing data, the Instruction Cache might become incoherent. In those cases, either a line or the whole Instruction Cache must be invalidated. In particular, the Instruction Cache must be invalidated before setting a software breakpoint or downloading code.

14.8.13 Reading memory as words

This sequence is optimized for a long sequential read.

This sequence assumes that r0 has been set to the address to load data from prior to running this sequence. r0 is post-incremented so that it can be used by successive reads of memory.

1. Load and issue the LDC instruction:

```

SCAN_N  5                                ; select DTR
ITRSEL                                ; select the ITR and EXTEST
INST  LDC p14,c5,[R0],#4                ; load the content of the position of
                                        ; memory pointed by R0 into wDTR and
                                        ; increment R0 by 4
RTI

```

2. The DTR is selected in order to read the data:

```

INTEST                                ; select the DTR and INTEST

```

3. This loop keeps on reading words, but it stops before the latest read. It is skipped if there is only one word to read:

```

FOR(i=1; i <= (Words2Read-1); i++) DO
    LOOP
        DATA 0x00000000 Ready readData ; gets the result of
                                           ; the previous read
        RTI ; issues the next read
    UNTIL Ready==1 ; wait until the instruction ends
    Save value in readData
ENDFOR

```

4. Wait for the last read to finish:

```

LOOP
    DATA 0x00000000 Ready readData
    UNTIL Ready==1 ; wait until instruction ends
    Save value in readData

```

5. Now check whether an abort occurred:

```

SCAN_N 1 ; select DSCR
INTEST
DATAOUT DSCR ; this action clears the DSCR[6] flag

```

6. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be written, and after the cause for the abort has been fixed the sequences resumes at step 1.

————— **Note** —————

If the sticky imprecise Data Aborts flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and r0 is reloaded.

14.8.14 Writing memory as words

This sequence is optimized for a long sequential write.

This sequence assumes that r0 has been set to the address to store data to prior to running this sequence. Register r0 is post-incremented so that it can be used by successive writes to memory:

1. The instruction is loaded:

```

SCAN_N 5 ; select DTR
ITRSEL ; select the ITR and EXTEST
INST STC p14,c5,[R0],#4 ; store the contents of rDTR into the
                          ; position of memory pointed by R0 and
                          ; increment it by 4

```


- ```

EXTEST ; select the DTR and EXTEST

```
- This loop writes all the words:

```

FOR (i=1; i <= Words2Write; i++) DO
 LOOP
 DATA Data2Write Ready
 RTI
 UNTIL Ready==1 ; wait until instruction ends
ENDFOR

```
  - Wait for the last write to finish:

```

LOOP
 DATA 0x00000000 Ready
 UNTIL Ready==1 ; wait until instruction ends

```
  - Check for aborts, as described in *Reading memory as words* on page 14-43.

#### 14.8.15 Reading memory as halfwords or bytes

The above sequences cannot be used to transfer halfwords or bytes because LDC and STC instructions always transfer whole words. Two operations are needed to complete a halfword or byte transfer, from memory to ARM register and from ARM register to CP14 debug register. Therefore, performance is decreased because the load instruction cannot be kept in the ITR.

This sequence assumes that r0 has been set to the address to load data from prior to running the sequence. Register r0 is post-incremented so that it can be used by successive reads of memory. Register r1 is used as a temporary register:

- Load and issue the LDRH or LDRB instruction:

```

ITRSEL ; select the ITR and EXTEST
INST LDRH R1,[R0],#2 ; LDRB R1,[R0],#1 for byte reads
RTI
LOOP
 INST 0x00000000 Ready
 UNTIL Ready==1 ; wait until instruction ends

```
- Use the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 14-40 on register r1. Now scan chain 5 and INTEST are selected.
- If there are more halfwords or bytes to be read go to 1.
- Check for aborts, as described in *Reading memory as words* on page 14-43.

### 14.8.16 Writing memory as halfwords/bytes

This sequence assumes that r0 has been set to the address to store data to prior to running this sequence. Register r0 is post-incremented so that it can be used by successive writes to memory. Register r1 is used as a temporary register:

1. Write the halfword/byte onto r1 using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 14-41. Scan chain 5 and EXTEST are selected.
2. Write the contents of r1 to memory:
 

```

ITRSEL ; select the ITR and EXTEST
INST STRH R1, [R0], #2 ; STRB R1, [R0], #1 for byte writes
RTI
LOOP
 INST 0x00000000 Ready
UNTIL Ready==1 ; wait until instruction ends

```
3. If there are more halfwords or bytes to be read go to 1.
4. Now check for aborts as described in *Reading memory as words* on page 14-43.

### 14.8.17 Coprocessor register reads and writes

The ARM1136JF-S processor can execute coprocessor instructions while in debug state. Therefore, the straightforward method to transfer data between a coprocessor and the DBGTAP debugger is using an ARM register temporarily. For this method to work, the coprocessor must be able to transfer all its registers to the core using coprocessor transfer instructions.

### 14.8.18 Reading coprocessor registers

1. Load the value into ARM register r0:
 

```

ITRSEL ; select the ITR and EXTEST
INST MRC px,y,R0,ca,cb,z
RTI
LOOP
 INST 0x00000000 Ready
UNTIL Ready==1 ; wait until instruction ends

```
2. Use the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 14-40.

### 14.8.19 Writing coprocessor registers

1. Write the value onto r0, using the standard sequence. See *Writing a current mode ARM register in the range r0-r14* on page 14-41 for more details. Scan chain 5 and EXTEST are selected.

2. Transfer the contents of r0 to a coprocessor register:

```

ITRSEL ; select the ITR and EXTEST
INST MCR px,y,R0,ca,cb,z
RTI
LOOP
 INST 0x00000000 Ready
UNTIL Ready==1 ; wait until instruction ends

```

## 14.9 Programming debug events

The following operations are described:

- *Reading registers using scan chain 7*
- *Writing registers using scan chain 7*
- *Setting breakpoints, watchpoints and vector traps on page 14-49*
- *Setting software breakpoints on page 14-49.*

### 14.9.1 Reading registers using scan chain 7

A typical sequence for reading registers using scan chain 7 is as follows:

```

SCAN_N 7 ; select ITR
EXTTEST
REQ 1stAddr2Rd 0 0 ; read request for register 1stAddr2read
FOR(i=2; i <= Words2Read; i++) DO
 LOOP
 REQ ithAddr2Rd 0 0 Ready readData
 ; ith read request while waiting
 UNTIL Ready==1 ; wait until the previous request completes
 Save value in readData
 ENDFOR
 LOOP
 REQ 0 0 0 Ready readData ; null request while waiting
 UNTIL Ready==1 ; wait until last request completes
 Save value in readData

```

### 14.9.2 Writing registers using scan chain 7

A typical sequence for writing to a register using scan chain 7 is as follows:

```

SCAN_N 7 ; select ITR
EXTTEST
REQ 1stAddr2Wr 1stData2Wr 0b1 ; write request for register 1stAddr2write
FOR(i=2; i <= Words2Write; i++) DO
 LOOP
 REQ ithAddr2Wr ithData2Wr 1 Ready
 ; ith write request while waiting
 UNTIL Ready==1 ; wait until the previous request completes
 ENDFOR
 LOOP
 REQ 0 0 0 Ready ; null request while waiting
 UNTIL Ready==1 ; wait until last request completes

```

### 14.9.3 Setting breakpoints, watchpoints and vector traps

You can program a vector catch debug event by writing to CP14 debug vector catch register.

You can program a breakpoint debug event by writing to CP14 debug 64-69 breakpoint value registers and CP14 debug 80-84 breakpoint control registers.

You can program a watchpoint debug event by writing to CP14 debug 96-97 watchpoint value registers and CP14 debug 112-113 watchpoint control registers.

---

**Note**

---

An External Debugger can access the CP14 debug registers whether the processor is in debug state or not, so these debug events can be programmed on-the-fly (while the processor is in ARM/Thumb/Java state).

---

See *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-41 for the sequences of register accesses needed to program these software debug events. See *Writing registers using scan chain 7* on page 14-48 to learn how to access CP14 debug registers using scan chain 7.

### 14.9.4 Setting software breakpoints

To set a software breakpoint on a certain Virtual Address, a debugger must go through the following steps:

1. Read memory location and save actual instruction.
2. Write the BKPT instruction to the memory location.
3. Read memory location again to check that the BKPT instruction got written.
4. If it is not written, determine the reason.

All of these can be done using the previously described sequences.

---

**Note**

---

Cache coherency issues might arise when writing a BKPT instruction. See *Debugging in a cached system* on page 13-39.

---

## 14.10 Monitor mode debugging

If DSCR[14] Halt/Monitor mode bit is clear, then the processor takes an exception (rather than halting) when a software debug event occurs. See *Halt mode debugging* on page 13-47 for details.

When the exception is taken, the handler uses the DCC to transmit status information to, and receive commands from the host using a DBGTAP debugger. Monitor mode is essential in real-time systems when the core cannot be halted to collect information.

### 14.10.1 Receiving data from the core

```

SCAN_N 5 ; select DTR
INTEST
FOREACH Data2Read
 LOOP
 DATA 0x00000000 Valid readData
 UNTIL Valid==1 ; wait until instruction ends
 Save value in readData
END

```

### 14.10.2 Sending data to the core

```

SCAN_N 5 ; select DTR
EXTTEST
FOREACH Data2Write
 LOOP
 DATA Data2Write nRetry
 UNTIL nRetry==1 ; wait until instruction ends
END

```

# Chapter 15

## Trace Interface Port

This chapter gives a brief description of the *Embedded Trace Macrocell* (ETM) support for the ARM1136JF-S processor. It contains the following section:

- *About the ETM interface* on page 15-2.

## 15.1 About the ETM interface

The ARM1136JF-S trace interface port enables simple connection of an ETM to an ARM1136JF-S processor. The ARM *Embedded Trace Macrocell* (ETM) provides instruction and data trace for the ARM1136JF-S family of processors.

All inputs are registered immediately inside the ETM unless specified otherwise. All outputs are driven directly from a register unless specified otherwise. All signals are relative to **CLKIN** unless specified otherwise.

The ETM interface includes the following groups of signals:

- an instruction interface
- a data address interface
- a pipeline advance interface
- a data value interface
- a coprocessor interface
- other connections to the core.

### 15.1.1 Instruction interface

The primary sampling point for these signals is on entry to Write-Back. See *Typical pipeline operations* on page 1-28. This ensures that instructions are traced correctly before any data transfers associated with them, as required by the ETM protocol.

The instruction interface signals are shown in Table 15-1.

**Table 15-1 Instruction interface signals**

| Signal name           | Description                                                                                                             | Qualified by     |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------|------------------|
| <b>ETMICTL[17:0]</b>  | Instruction interface control signals                                                                                   | -                |
| <b>ETMIA[31:0]</b>    | This is the address for:<br>ARM executed instruction + 8<br>Thumb executed instruction + 4<br>Java executed instruction | <b>IValid</b>    |
| <b>ETMIARET[31:0]</b> | Address to return to if branch is incorrectly predicted                                                                 | <b>IABpValid</b> |

**ETMIA** is used for branch target address calculation.



Other than this the ETM must know, for each cycle, the current address of the instruction in execute and the address of any branch phantom progressing through the pipeline. The ARM1136JF-S processor does not maintain the address of branch phantoms, instead it maintains the address to return to if the branch proves to be incorrectly predicted.

The instruction interface can trace a branch phantom without an associated normal instruction.

In the case of a branch that is predicted taken, the return address (for when the branch is not taken) is one instruction after the branch. Therefore, the branch address is:

$$\text{ETMIABP} = \text{ETMIARET} - \langle \text{size} \rangle$$

When the instruction is predicted not taken, the return address is the target of the branch. However, because the branch was not taken, it must precede the normal instruction. Therefore, the branch address is:

$$\text{ETMIABP} = \text{ETMIA} - \langle \text{size} \rangle$$

The **ETMIACTL[17:0]** instruction interface control signals are shown in Table 15-2.

**Table 15-2 ETMIACTL[17:0]**

| Bits    | Reference name     | Description                                                                                                                                    | Qualified by       |
|---------|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|
| [17]    | <b>IASlotKill</b>  | Kill outstanding slots.                                                                                                                        | <b>IAException</b> |
| [16]    | <b>IADAbort</b>    | Data Abort.                                                                                                                                    | <b>IAException</b> |
| [15]    | <b>IAExCancel</b>  | Exception canceled previous instruction.                                                                                                       | <b>IAException</b> |
| [12:14] | <b>IAExInt</b>     | b001 = IRQ<br>b101 = FIQ<br>b100 = Java exception<br>b110 = Precise Data Abort<br>b000 = Other exception.                                      | <b>IAException</b> |
| [11]    | <b>IAException</b> | Instruction is an exception vector.                                                                                                            | None <sup>a</sup>  |
| [10]    | <b>IABounce</b>    | Kill the data slot associated with this instruction. There is only ever one of these instructions. Used for bouncing coprocessor instructions. | <b>IADataInst</b>  |
| [9]     | <b>IADataInst</b>  | Instruction is a data instruction. This includes any load, store, or CPRT, but does not include preloads.                                      | <b>IAInstValid</b> |
| [8]     | <b>IAContextID</b> | Instruction updates context ID.                                                                                                                | <b>IAInstValid</b> |

Table 15-2 ETMICTL[17:0] (continued)

| Bits | Reference name      | Description                                                                                                  | Qualified by       |
|------|---------------------|--------------------------------------------------------------------------------------------------------------|--------------------|
| [7]  | <b>IAIndBr</b>      | Instruction is an indirect branch.                                                                           | <b>IAInstValid</b> |
| [6]  | <b>IABpCCFail</b>   | Branch phantom failed its condition codes.                                                                   | <b>IABpValid</b>   |
| [5]  | <b>IAInstCCFail</b> | Instruction failed its condition codes.                                                                      | <b>IAInstValid</b> |
| [4]  | <b>IAJBit</b>       | Instruction executed in Java state.                                                                          | <b>IAValid</b>     |
| [3]  | <b>IATBit</b>       | Instruction executed in Thumb state.                                                                         | <b>IAValid</b>     |
| [2]  | <b>IABpValid</b>    | Branch phantom executed this cycle.                                                                          | <b>IAValid</b>     |
| [1]  | <b>IAInstValid</b>  | (Non-phantom) instruction executed this cycle.                                                               | <b>IAValid</b>     |
| [0]  | <b>IAValid</b>      | Signals on the instruction interface are valid this cycle.<br>This is kept LOW when the ETM is powered down. | None               |

- a. The exception signals become valid when the core takes the exception and remain valid until the next instruction is seen at the exception vector.

### 15.1.2 Data address interface

Data addresses are sampled at the ADD stage because they are guaranteed to be in order at this point. These are assigned a slot number for identification on retirement.

The data address interface signals are shown in Table 15-3.

Table 15-3 Data address interface signals

| Signal name           | Description                            | Qualified by                           |
|-----------------------|----------------------------------------|----------------------------------------|
| <b>ETMDACTL[17:0]</b> | Data address interface control signals | -                                      |
| <b>ETMDA[31:3]</b>    | Address for data transfer              | <b>DASlot</b> != 00 AND <b>!DACPRT</b> |

The ETMDACTL[17:0] signals are described in Table 15-4.

**Table 15-4 ETMDACTL[17:0]**

| Bits    | Reference name     | Description                                                                                                                                                                                                                                                                                                                                                                                                     | Qualified by        |
|---------|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| [17]    | <b>DANSeq</b>      | The data transfer is nonsequential from the last. This signal must be asserted on the first cycle of each instruction, in addition to the second transfer of a SWP or LDM pc, because the address of these transfers is not one word greater than the previous transfer, and therefore the transfer must have its address re-output.<br>This signal is only valid on the first transfer of an unaligned access. | <b>DASlot</b> != 00 |
| [16]    | <b>DALast</b>      | The data transfer is the last for this data instruction. This signal is asserted for both halves of an unaligned access.<br>A related signal, DAFirst, can be implied from this signal, because the next transfer must be the first transfer of the next data instruction.                                                                                                                                      | <b>DASlot</b> != 00 |
| [15]    | <b>DACPRT</b>      | The data transfer is a CPRT.                                                                                                                                                                                                                                                                                                                                                                                    | <b>DASlot</b> != 00 |
| [14]    | <b>DASwizzle</b>   | Words must be byte swizzled for ARM big-endian mode. This signal is only valid on the first transfer of an unaligned access.                                                                                                                                                                                                                                                                                    | <b>DASlot</b> != 00 |
| [13:12] | <b>DARot</b>       | Number of bytes to rotate right each word by. This signal is only valid on the first transfer of an unaligned access.                                                                                                                                                                                                                                                                                           | <b>DASlot</b> != 00 |
| [11]    | <b>DAUnaligned</b> | First transfer of an unaligned access.<br>The next transfer must be the second half, for which this signal is not asserted.                                                                                                                                                                                                                                                                                     | <b>DASlot</b> != 00 |
| [10:3]  | <b>DABLSel</b>     | Byte lane selects.                                                                                                                                                                                                                                                                                                                                                                                              | <b>DASlot</b> != 00 |
| [2]     | <b>DAWrite</b>     | Read or write.<br>This signal is only valid on the first transfer of an unaligned access.                                                                                                                                                                                                                                                                                                                       | <b>DASlot</b> != 00 |
| [1:0]   | <b>DASlot</b>      | Slot occupied by data item. b00 indicates that no slot is in use in this cycle. This is kept at b00 when the ETM is powered down.                                                                                                                                                                                                                                                                               | None                |

### 15.1.3 Data value interface

The data values are sampled at the WBIs stage. Here the load, store, MCR, and MRC data is combined. The memory view of the data is presented, which must be converted back to the register view depending on the alignment and endianness.

Data is not returned for at least two cycles after the address. However, it is not necessary to pipeline the address because the slot does not return data for a previous address during this time. Data values are defined to correspond to the most recent data addresses

with the same slot number, starting from the previous cycle. In other words, data can correspond to an address from the previous cycle, but not to an address from the same cycle.

The data value interface signals are shown in Table 15-5.

**Table 15-5 Data value interface signals**

| Signal name          | Description                                                  | Qualified by        |
|----------------------|--------------------------------------------------------------|---------------------|
| <b>ETMDDCTL[3:0]</b> | Data value interface control signals                         | -                   |
| <b>ETMDD[63:0]</b>   | Contains the data for a load, store, MRC, or MCR instruction | <b>DDSlot</b> != 00 |

The **ETMDDCTL[3:0]** signals are described in Table 15-6.

**Table 15-6 ETMDDCTL[3:0]**

| Bits  | Reference name    | Description                                                                                                                 | Qualified by        |
|-------|-------------------|-----------------------------------------------------------------------------------------------------------------------------|---------------------|
| [3]   | <b>DDImpAbort</b> | Imprecise Data Aborts on this slot. Data is ignored.                                                                        | <b>DDSlot</b> != 00 |
| [2]   | <b>DDFail</b>     | STREX data write failed.                                                                                                    | <b>DDSlot</b> != 00 |
| [1:0] | <b>DDSlot</b>     | Slot occupied by data item. b00 indicates that no slot is in use this cycle. This is kept b00 when the ETM is powered down. | None                |

#### 15.1.4 Pipeline advance interface

There are three points in the ARM1136JF-S pipeline at which signals are produced for the ETM. These signals must be realigned by the ETM, so pipeline advance signals are provided.

The pipeline advance signals indicate when a new instruction enters pipeline stages Ex3, Ex2, and ADD, see *Typical pipeline operations* on page 1-28.

The **ETMPADV[2:0]** pipeline advance interface signals are shown in Table 15-7.

**Table 15-7 ETMPADV[2:0]**

| Bits | Reference name           | Description                     | Qualified by |
|------|--------------------------|---------------------------------|--------------|
| [2]  | <b>PAEx3<sup>a</sup></b> | Instruction entered Ex3         | -            |
| [1]  | <b>PAEx2<sup>a</sup></b> | Instruction entered Ex2         | -            |
| [0]  | <b>PAAdd<sup>a</sup></b> | Instruction entered Ex1 and ADD | -            |

a. This is kept LOW when the ETM is powered down.

The pipeline advance signals present in other interfaces are:

|                     |                             |
|---------------------|-----------------------------|
| <b>IInvalid</b>     | Instruction entered WBEx.   |
| <b>DASlot != 00</b> | Data transfer entered DC1.  |
| <b>DDSlot != 00</b> | Data transfer entered WBIs. |

### 15.1.5 Coprocessor interface

This interface enables software to access ETM registers as registers in CP14. Rather than using the external coprocessor interface, the core provides a dedicated, cut-down coprocessor interface similar to that used by the debug logic.

The coprocessor interface signals are described in Table 15-8.

**Table 15-8 Coprocessor interface signals**

| Signal name        | Direction | Description                                                                                                                             | Qualified by          | Reg bound              |
|--------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------|-----------------------|------------------------|
| <b>ETMCPENABLE</b> | Output    | Interface enable. <b>ETMCPWRITE</b> and <b>ETMCPADDRESS</b> are valid this cycle, and the remaining signals are valid two cycles later. | None                  | No (late) <sup>a</sup> |
| <b>ETMCPCOMMIT</b> | Output    | Commit. If this signal is LOW two cycles after <b>ETMCPENABLE</b> is asserted, the transfer is canceled and must not take any effect.   | <b>ETMCPENABLE</b> +2 | No (late) <sup>a</sup> |
| <b>ETMCPWRITE</b>  | Output    | Read or write. Asserted for write.                                                                                                      | <b>ETMCPENABLE</b>    | Yes                    |

Table 15-8 Coprocessor interface signals (continued)

| Signal name        | Direction | Description      | Qualified by | Reg bound |
|--------------------|-----------|------------------|--------------|-----------|
| ETMCPADDRESS[14:0] | Output    | Register number. | ETMCPENABLE  | Yes       |
| ETMCPADATA[31:0]   | Input     | Read data.       | ETMCPCOMMIT  | Yes       |
| ETMCPWDATA[31:0]   | Output    | Write value.     | ETMCPCOMMIT  | Yes       |

a. Used as a clock enable for coprocessor interface logic.

A complete transaction takes three cycles. The first and last cycles can overlap, giving a sustained rate of one every two cycles.

The ETM coprocessor interface also catches writes to the Context ID Register, CP15 c13 (see *Context ID Register* on page 3-95). This enables the state of this register to be shadowed even when the core interface is powered down.

Only the following instructions are presented by the coprocessor interface:

MRC p14, 1, <Rd>, c0, <reg[3:0]>, <reg[6:4]> ; Read ETM register

MCR p14, 1, <Rd>, c0, <reg[3:0]>, <reg[6:4]> ; Write ETM register

MCR p15, 0, <Rd>, c13, c0, 1 ; Write Context ID Register

Where <reg[3:0]> and <reg[6:4]> are bits in the ETM register to be accessed.

The format of the **ETMCPADDRESS[14:0]** signals are shown in Figure 15-1.

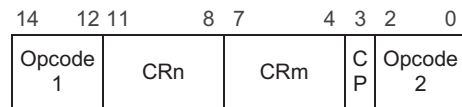


Figure 15-1 ETMCPADDRESS format

In Figure 15-1, the CP bit is 0 for CP14 or 1 for CP15.

Non-ETM instructions are not presented on this interface.

In contrast to the debug logic, the core makes no attempt to decode if a given ETM register exists or not. If a register does not exist, the write is silently ignored. For more details see the *Embedded Trace Macrocell Specification*.

### 15.1.6 Other connections to the core

The signals shown in Table 15-9 are also connected to the core.

**Table 15-9 Other connections**

| Signal name           | Direction | Description                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>EVNTBUS[19:0]</b>  | Output    | Gives the status of the performance monitoring events. See <i>System performance monitoring</i> on page 3-87.                                                                                                                                                                                                                                                                                      |
| <b>ETMEXTOUT[1:0]</b> | Input     | Provides feedback to the core of the <b>EVNTBUS</b> signals after being passed through ETM triggering facilities and comparators. This enables the performance monitoring facilities provide by ARM1136JF-S processors to be conditioned in the same way as ETM events. For more details see <i>System performance monitoring</i> on page 3-87 and the <i>ETM11RV Technical Reference Manual</i> . |
| <b>ETMPWRUP</b>       | Input     | Indicates that the ETM is active. When LOW the Trace Interface must be clock gated to conserve power.                                                                                                                                                                                                                                                                                              |





# Chapter 16

## Cycle Timings and Interlock Behavior

This chapter describes the cycle timings and interlock behavior of integer instructions on the ARM1136JF-S and ARM1136J-S processors. This chapter contains the following sections:

- *About cycle timings and interlock behavior* on page 16-2
- *Register interlock examples* on page 16-7
- *Data processing instructions* on page 16-8
- *QADD, QDADD, QSUB, and QDSUB instructions* on page 16-11
- *ARMv6 media data-processing* on page 16-12
- *ARMv6 Sum of Absolute Differences (SAD)* on page 16-14
- *Multiplies* on page 16-15
- *Branches* on page 16-17
- *Processor state updating instructions* on page 16-18
- *Single load and store instructions* on page 16-19
- *Load and Store Double instructions* on page 16-22
- *Load and Store Multiple Instructions* on page 16-24
- *RFE and SRS instructions* on page 16-27
- *Synchronization instructions* on page 16-28.
- *Coprocessor instructions* on page 16-29
- *SWI, BKPT, Undefined, Prefetch Aborted instructions* on page 16-30
- *Thumb instructions* on page 16-31.

## 16.1 About cycle timings and interlock behavior

Complex instruction dependencies and memory system interactions make it impossible to describe briefly the exact cycle timing behavior for all instructions in all circumstances. The timings described in this chapter are accurate in most cases. If precise timings are required you must use a cycle-accurate model of the ARM1136JF-S processor.

Unless stated otherwise cycle counts and result latencies described in this chapter are best case numbers. They assume:

- no outstanding data dependencies between the current instruction and a previous instruction
- the instruction does not encounter any resource conflicts
- all data accesses hit in the MicroTLB and Data Cache, and do not cross protection region boundaries
- all instruction accesses hit in the Instruction Cache.

This section describes:

- *Changes in instruction flow overview*
- *Instruction execution overview* on page 16-3
- *Conditional instructions* on page 16-4
- *Opposite condition code checks* on page 16-5
- *Definition of terms* on page 16-6.

### 16.1.1 Changes in instruction flow overview

To minimize the number of cycles, because of changes in instruction flow, the ARM1136JFS processor includes a:

- dynamic branch predictor
- static branch predictor
- return stack.

The dynamic branch predictor is a 128-entry direct-mapped branch predictor using VA bits [9:3]. The prediction scheme uses a two-bit saturating counter for predictions that are:

- Strongly Not Taken
- Weakly Not Taken
- Weakly Taken
- Strongly Taken.

Only branches with a constant offset are predicted. Branches with a register-based offset are not predicted. A dynamically predicted branch can be folded out of the instruction stream if the following instruction arrives while the branch is within the prefetch instruction buffer. A dynamically predicted branch takes one cycle or zero cycles if folded out.

The static branch predictor operates on branches with a constant offset that are not predicted by the dynamic branch predictor. Static predictions are issued from the Iss stage of the main pipeline, consequently a statically predicted branch takes four cycles.

The return stack consists of three entries, and as with static predictions, issues a prediction from the Iss stage of the main pipeline. The return stack mispredicts if the value taken from the return stack is not the value that is returned by the instruction. Only unconditional returns are predicted. A conditional return pops an entry from the return stack but is not predicted. If the return stack is empty a return is not predicted. Items are placed on the return stack from the following instructions:

- BL #<immed>
- BLX #<immed>
- BLX Rx

Items are popped from the return stack by the following types of instruction:

- BX lr
- MOV pc, lr
- LDR pc, [sp], #cns
- LDMIA sp!, {...,pc}

A correctly predicted return stack pop takes four cycles.

## 16.1.2 Instruction execution overview

The instruction execution pipeline is constructed from three parallel four-stage pipelines, see Table 16-1. For a complete description of these pipeline stages see *Pipeline stages* on page 1-26.

**Table 16-1 Pipeline stages**

| Pipeline   | Stages |      |      |      |
|------------|--------|------|------|------|
| ALU        | Sh     | ALU  | Sat  | WBex |
| Multiply   | MAC1   | MAC2 | MAC3 |      |
| Load/Store | ADD    | DC1  | DC2  | WBls |

The ALU and multiply pipelines operate in a lock-step manner, causing all instructions in these pipelines to retire in order. The load/store pipeline is a decoupled pipeline enabling subsequent instructions in the ALU and multiply pipeline to complete underneath outstanding loads.

Extensive forwarding to the Sh, MAC1, ADD, ALU, MAC2, and DC1 stages enables many dependent instruction sequences to run without pipeline stalls. General forwarding occurs from the ALU, Sat, WBex and WBls pipeline stages. In addition, the multiplier contains an internal multiply accumulate forwarding path.

Most instructions do not require a register until the ALU stage. All result latencies are given as the number of cycles until the register is required by a following instruction in the ALU stage.

The following sequence takes four cycles:

```
LDR r1, [r2] ;Result latency three
ADD r3, r3, r1 ;Register r1 required by ALU
```

If a subsequent instruction requires the register at the start of the Sh, MAC1, or ADD stage then an extra cycle must be added to the result latency of the instruction producing the required register. Instructions that require a register at the start of these stages are specified by describing that register as an Early Reg. The following sequence, requiring an Early Reg, takes five cycles:

```
LDR r1, [r2] ;Result latency three plus one
ADD r3, r3, r1 LSL#6 ;plus one since Register r1 is required by Sh
```

Finally, some instructions do not require a register until their second execution cycle. If a register is not required until the ALU, MAC1, or Dc1 stage for the second execution cycle, then a cycle can be subtracted from the result latency for the instruction producing the required register. If a register is not required until this later point, it is specified as a Late Reg. The following sequence where r1 is a Late Reg takes four cycles:

```
LDR r1, [r2] ;Result latency three minus one
ADD r3, r3, r1, r4 LSL#5 ;minus one since Register r1 is a Late Reg
 ;This ADD is a two issue cycle instruction
```

### 16.1.3 Conditional instructions

Most instructions execute in one or two cycles. If these instructions fail their condition codes then they take one and two cycles respectively.

Multiplies, MSR, and some CP14 and CP15 coprocessor instructions are the only instructions that require more than two cycles to execute. If one of these instructions fails its condition codes, then it takes a variable number of cycles to execute. The number of cycles is dependent on:

- the length of the operation
- the number of cycles between the setting of the flags and the start of the dependent instruction.

The worst-case number of cycles for a condition code failing multicycle instruction is five.

The following algorithm describes the number of cycles taken for multi-cycle instructions which condition-code fail:

$\text{Min}(\text{NonFailingCycleCount}, \text{Max}(5 - \text{FlagCycleDistance}, 3))$

Where:

**Max (a,b)** returns the maximum of the two values a,b.

**Min (a,b)** returns the minimum of the two values a,b.

**NonFailingCycleCount**

is the number of cycles that the failing instruction would have taken had it passed.

**FlagCycDistance**

is the number of cycles between the instruction that sets the flags and the conditional instruction, including interlocking cycles. For example:

- The following sequence has a FlagCycleDistance of 0 because the instructions are back-to-back with no interlocks:  

```

ADDS r1, r2, r3
MULEQ r4, r5, r6

```
- The following sequence has a FlagCycleDistance of one:  

```

ADDS r1, r2, r3
MOV r0, r0
MULEQ r4, r5, r6

```

#### 16.1.4 Opposite condition code checks

If instruction A and instruction B both write the same register the pipeline must ensure that the register is written in the correct order. Therefore interlocks might be required to correctly resolve this pipeline hazard.

The only useful sequences where two instructions write the same register without an instruction reading its value in between are when the two instructions have opposite sets of condition codes. The ARM1136JF-S processor optimizes these sequences to prevent unnecessary interlocks. For example:

- The following sequences take two cycles to execute:
  - `ADDNE r1, r5, r6`  
`LDREQ r1, [r8]`
  - `LDREQ r1, [r8]`  
`ADDNE r1, r5, r6`
- The following sequence also takes two cycles to execute, because the STR instruction does not store the value of r1 produced by the QDADDNE instruction:
  - `QDADDNE r1, r5, r6`
  - `STREQ r1, [r8]`

### 16.1.5 Definition of terms

Table 16-2 gives descriptions of cycle timing terms used in this chapter.

**Table 16-2 Definition of cycle timing terms**

| Term                  | Description                                                                                                                                                                                                                                                                                                              |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cycles                | This is the minimum number of cycles required by an instruction.                                                                                                                                                                                                                                                         |
| Result Latency        | This is the number of cycles before the result of this instruction is available for a following instruction requiring the result at the start of the ALU, MAC2, and DC1 stage. This is the normal case. Exceptions to this mark the register as an Early Reg.                                                            |
|                       | <p style="text-align: center;">———— <b>Note</b> —————</p> <p>The result latency is the number of cycles from the first cycle of an instruction.</p>                                                                                                                                                                      |
| Register Lock Latency | For STM and STRD instructions only. This is the number of cycles that a register is write locked for by this instruction, preventing subsequent instructions that want to write the register from starting. This lock is required to prevent a following instruction from writing to a register before it has been read. |
| Early Reg             | The specified registers are needed at the start of the Sh, MAC1, and ADD stage. Add one cycle to the result latency of the instruction producing this register for interlock calculations.                                                                                                                               |
| Late Reg              | The specified registers are not needed until the start of the ALU, MAC1, and DC1 stage for the second execution. Subtract one cycle from the result latency of the instruction producing this register for interlock calculations.                                                                                       |
| FlagsCycleDistance    | The number of cycles between an instruction that sets the flags and the conditional instruction.                                                                                                                                                                                                                         |

## 16.2 Register interlock examples

Table 16-3 shows register interlock examples using LDR and ADD instructions.

LDR instructions take one cycle, have a result latency of three, and require their base register as an Early Reg.

ADD instructions take one cycle and have a result latency of one.

**Table 16-3 Register interlock examples**

| <b>Instruction sequence</b>      | <b>Behavior</b>                                                                                 |
|----------------------------------|-------------------------------------------------------------------------------------------------|
| LDR r1, [r2]<br>ADD r6, r5, r4   | Takes two cycles because there are no register dependencies                                     |
| ADD r1, r2, r3<br>ADD r9, r6, r1 | Takes two cycles because ADD instructions have a result latency of one                          |
| LDR r1, [r2]<br>ADD r6, r5, r1   | Takes four cycles because of the result latency of r1                                           |
| ADD r2, r5, r6<br>LDR r1, [r2]   | Takes three cycles because of the use of the result of r1 as an Early Reg                       |
| LDR r1, [r2]<br>LDR r5, [r1]     | Takes five cycles because of the result latency and the use of the result of r1 as an Early Reg |

## 16.3 Data processing instructions

This section describes the cycle timing behavior for the AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, CMN, ORR, MOV, BIC, MVN, TST, TEQ, CMP, and CLZ instructions.

### 16.3.1 Cycle counts if destination is not PC

Table 16-4 shows the cycle timing behavior for data processing instructions if its destination is not the PC. You can substitute ADD with any of the data processing instructions identified in the opening paragraph of this section.

**Table 16-4 Data Processing Instruction cycle timing behavior if destination is not PC**

| Example Instruction                | Cycles | Early Reg | Late Reg | Result Latency | Comment                                                                                                                                                                                                                     |
|------------------------------------|--------|-----------|----------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ADD <Rd>, <Rn>, <Rm>.              | 1      | -         | -        | 1              | Normal case.                                                                                                                                                                                                                |
| ADD <Rd>, <Rn>, <Rm>, LSL #<immed> | 1      | <Rm>      | -        | 1              | Requires a shifted source register.                                                                                                                                                                                         |
| ADD <Rd>, <Rn>, <Rm>, LSL <Rs>     | 2      | <Rs>      | <Rn>     | 2              | Requires a register controlled shifted source register. Instruction takes two issue cycles. In the first cycle the shift distance Rs is sampled. In the second cycle the actual shift of Rm and the ADD instruction occurs. |

### 16.3.2 Cycle counts if destination is the PC

Table 16-5 shows the cycle timing behavior for data processing instructions if its destination is the PC. You can substitute ADD with any data processing instruction except for a MOV and CLZ. A CLZ with the PC as the destination is an Unpredictable instruction.

The timings for a MOV instruction are given separately in the table.

For condition code failing cycle counts, the cycles for the non-PC destination variants must be used.

**Table 16-5 Data Processing Instruction cycle timing behavior if destination is the PC**

| Example Instruction | Cycles | Early Reg | Late Reg | Result Latency | Comment                                     |
|---------------------|--------|-----------|----------|----------------|---------------------------------------------|
| MOV pc, 1r          | 4      | -         | -        | -              | Correctly return stack predicted MOV pc, 1r |



**Table 16-5 Data Processing Instruction cycle timing behavior if destination is the PC (continued)**

| Example Instruction                     | Cycles           | Early Reg | Late Reg | Result Latency | Comment                                                       |
|-----------------------------------------|------------------|-----------|----------|----------------|---------------------------------------------------------------|
| MOV pc, lr                              | 7                | -         | -        | -              | Incorrectly return stack predicted MOV pc, lr                 |
| MOV <cond> pc, lr                       | 5-7 <sup>a</sup> | -         | -        | -              | Conditional return, or return when return stack is empty      |
| MOV pc, <Rd>                            | 5                | -         | -        | -              | MOV to PC, no shift required                                  |
| MOV <cond> pc, <Rd>                     | 5-7 <sup>a</sup> | -         | -        | -              | Conditional MOV to PC, no shift required                      |
| MOV pc, <Rn>, <Rm>, LSL #<immed>        | 6                | <Rm>      | -        | -              | Conditional MOV to PC, with a shifted source register         |
| MOV <cond> pc, <Rn>, <Rm>, LSL #<immed> | 6-7 <sup>a</sup> | -         | -        | -              | Conditional MOV to PC, with a shifted source register         |
| MOV pc, <Rn>, <Rm>, LSL <Rs>            | 7                | <Rs>      | <Rn>     | -              | MOV to pc, with a register controlled shifted source register |
| ADD pc, <Rd>, <Rm>                      | 7                | -         | -        | -              | Normal case to PC                                             |
| ADD pc, <Rn>, <Rm>, LSL #<immed>        | 7                | <Rm>      | -        | -              | Requires a shifted source register                            |
| ADD pc, <Rn>, <Rm>, LSL <Rs>            | 8                | <Rs>      | <Rn>     | -              | Requires a register controlled shifted source register        |

- a. If the instruction is conditional and passes conditional checks it takes  $\text{MAX}(\text{MaxCycles} - \text{FlagCycleDistance}, \text{MinCycles})$ . If the instruction is unconditional it takes Min Cycles.

### 16.3.3 Example interlocks

Most data processing instructions are single-cycle and can be executed back-to-back without interlock cycles, even if there are data dependencies between them. The exceptions to this are when the Shifter or Register controlled shifts are used.

#### Shifter

The shifter is in a separate pipeline stage from the ALU. A register required by the shifter is an Early Reg and requires an additional cycle of result availability before use. For example, the following sequence introduces a one-cycle interlock, and takes three cycles to execute:

```
ADD r1, r2, r3
ADD r4, r5, r1 LSL #1
```

The second source register, which is not shifted, does not incur an extra data dependency check. Therefore, the following sequence takes two cycles to execute:

```
ADD r1, r2, r3
ADD r4, r1, r9 LSL #1
```

### Register controlled shifts

Register controlled shifts take two cycles to execute:

- the register containing the shift distance is read in the first cycle
- the shift is performed in the second cycle
- The final operand is not required until the ALU stage for the second cycle.

Because a shift distance is required, the register containing the shift distance is an Early Reg and incurs an extra interlock penalty. For example, the following sequence takes four cycles to execute:

```
ADD r1, r2, r3
ADD r4, r2, r4, LSL r1
```

## 16.4 QADD, QDADD, QSUB, and QDSUB instructions

This section describes the cycle timing behavior for the QADD, QDADD, QSUB, and QDSUB instructions.

These instructions perform saturating arithmetic. Their result is produced during the Sat stage, consequently they have a result latency of two. The QDADD and QDSUB instructions must double and saturate the register <Rn> before the addition. This operation occurs in the Sh stage of the pipeline, consequently this register is an Early Reg.

Table 16-6 shows the cycle timing behavior for QADD, QDADD, QSUB, and QDSUB instructions.

**Table 16-6 QADD, QDADD, QSUB, and QDSUB instruction cycle timing behavior**

| Instructions | Cycles | Early Reg | Result Latency |
|--------------|--------|-----------|----------------|
| QADD, QSUB   | 1      | -         | 2              |
| QDADD, QDSUB | 1      | <Rn>      | 2              |

## 16.5 ARMv6 media data-processing

Table 16-7 shows ARMv6 media data-processing instructions and gives their cycle timing behavior.

All ARMv6 media data-processing instructions are single-cycle issue instructions. These instructions produce their results in either the ALU or Sat stage, and have result latencies of one or two accordingly. Some of the instructions require an input register to be shifted before use and therefore are marked as requiring an Early Reg.

**Table 16-7 ARMv6 media data-processing instructions cycle timing behavior**

| Instructions                        | Cycles | Early Reg  | Result Latency |
|-------------------------------------|--------|------------|----------------|
| SADD16, SSUB16, SADD8, SSUB8        | 1      | -          | 1              |
| USAD8, USADA8                       | 1      | <Rm>, <Rs> | 3              |
| UADD16, USUB16, UADD8, USUB8        | 1      | -          | 1              |
| SEL                                 | 1      | -          | 1              |
| QADD16, QSUB16, QADD8, QSUB8        | 1      | -          | 2              |
| SHADD16, SHSUB16, SHADD8, SHSUB8    | 1      | -          | 2              |
| UQADD16, UQSUB16, UQADD8, UQSUB8    | 1      | -          | 2              |
| UHADD16, UHSUB16, UHADD8, UHSUB8    | 1      | -          | 2              |
| SSAT16, USAT16                      | 1      | -          | 2              |
| SADDSUBX, SSUBADDX                  | 1      | <Rm>       | 1              |
| UADDSUBX, USUBADDX                  | 1      | <Rm>       | 1              |
| SADD8TO16, SADD8TO32, SADD16TO32    | 1      | <Rm>       | 1              |
| SUNPK8TO16, SUNPK8TO32, SUNPK16TO32 | 1      | <Rm>       | 1              |
| UUNPK8TO16, UUNPK8TO32, UUNPK16TO32 | 1      | <Rm>       | 1              |
| UADD8TO16, UADD8TO32, UADD16TO32    | 1      | <Rm>       | 1              |
| REV, REV16, REVSH                   | 1      | <Rm>       | 1              |
| PKHBT, PKHTB                        | 1      | <Rm>       | 1              |
| SSAT, USAT                          | 1      | <Rm>       | 2              |
| QADDSUBX, QSUBADDX                  | 1      | <Rm>       | 2              |

**Table 16-7 ARMv6 media data-processing instructions cycle timing behavior**

| <b>Instructions</b>  | <b>Cycles</b> | <b>Early Reg</b> | <b>Result Latency</b> |
|----------------------|---------------|------------------|-----------------------|
| SHADDSUBX, SHSUBADDX | 1             | <Rm>             | 2                     |
| UQADDSUBX, UQSUBADDX | 1             | <Rm>             | 2                     |
| UHADDSUBX, UHSUBADDX | 1             | <Rm>             | 2                     |

## 16.6 ARMv6 Sum of Absolute Differences (SAD)

Table 16-8 shows ARMv6 SAD instructions and gives their cycle timing behavior.

**Table 16-8 ARMv6 sum of absolute differences instruction timing behavior**

| Instructions | Cycles | Early Reg  | Result Latency |
|--------------|--------|------------|----------------|
| USAD8        | 1      | <Rm>, <Rs> | 3 <sup>a</sup> |
| USADA8       | 1      | <Rm>, <Rs> | 3              |

a. Result latency is one less if the destination is the accumulate for a subsequent USADA8.

### 16.6.1 Example interlocks

Table 16-9 shows interlock examples using USAD8 and USADA8 instructions.

**Table 16-9 Example interlocks**

| Instruction sequence                                           | Behavior                                                                                                                             |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| USAD8 r1, r2, r3<br>ADD r5, r6, r1                             | Takes four cycles because USAD8 has a Result Latency of three, and the ADD requires the result of the USAD8 instruction.             |
| USAD8 r1, r2, r3<br>MOV r9, r9<br>MOV r9, r9<br>ADD r5, r6, r1 | Takes four cycles. The MOV instructions are scheduled during the Result Latency of the USAD8 instruction.                            |
| USAD8 r1, r2, r3<br>USADA8 r1, r4, r5, r1                      | Takes three cycles. The Result Latency is one less because the result is used as the accumulate for a subsequent USADA8 instruction. |

## 16.7 Multiplies

The multiplier consists of a three-cycle pipeline with early result forwarding not possible other than to the internal accumulate path. For a subsequent multiply accumulate the result is available one cycle earlier than for all other uses of the result.

Certain multiplies require:

- more than one cycle to execute.
- more than one pipeline issue to produce a result.

Multiplies with 64-bit results take and require two cycles to write the results, consequently they have two result latencies with the low half of the result always available first. The multiplicand and multiplier are required as Early Regs because they are both required at the start of MAC1.

Table 16-10 shows the cycle timing behavior of example multiply instructions.

**Table 16-10 Example multiply instruction cycle timing behavior**

| Example Instruction | Cycles | Cycles if sets flags | Early Reg  | Late Reg | Result Latency |
|---------------------|--------|----------------------|------------|----------|----------------|
| MUL(S)              | 2      | 5                    | <Rm>, <Rs> | -        | 4              |
| MLA(S)              | 2      | 5                    | <Rm>, <Rs> | <Rn>     | 4              |
| SMULL(S)            | 3      | 6                    | <Rm>, <Rs> | -        | 4/5            |
| UMULL(S)            | 3      | 6                    | <Rm>, <Rs> | -        | 4/5            |
| SMLAL(S)            | 3      | 6                    | <Rm>, <Rs> | <RdLo>   | 4/5            |
| UMLAL(S)            | 3      | 6                    | <Rm>, <Rs> | <RdLo>   | 4/5            |
| SMULxy              | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMLAxy              | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMULWy              | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMLAWy              | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMLALxy             | 2      | -                    | <Rm>, <Rs> | <RdHi>   | 3/4            |
| SMUAD, SMUADX       | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMLAD, SMLADX       | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMUSD, SMUSDx       | 1      | -                    | <Rm>, <Rs> | -        | 3              |
| SMLSD, SMLSDx       | 1      | -                    | <Rm>, <Rs> | -        | 3              |

**Table 16-10 Example multiply instruction cycle timing behavior (continued)**

| <b>Example Instruction</b> | <b>Cycles</b> | <b>Cycles if sets flags</b> | <b>Early Reg</b> | <b>Late Reg</b> | <b>Result Latency</b> |
|----------------------------|---------------|-----------------------------|------------------|-----------------|-----------------------|
| SMMUL, SMMULR              | 2             | -                           | <Rm>, <Rs>       | -               | 4                     |
| SMMLA, SMMLAR              | 2             | -                           | <Rm>, <Rs>       | <Rn>            | 4                     |
| SMMLS, SMMLSR              | 2             | -                           | <Rm>, <Rs>       | <Rn>            | 4                     |
| SMLALD, SMLALDX            | 2             | -                           | <Rm>, <Rs>       | <RdHi>          | 3/4                   |
| SMLSLD, SMLSLDX            | 2             | -                           | <Rm>, <Rs>       | <RdHi>          | 3/4                   |
| UMAAL                      | 3             | -                           | <Rm>, <Rs>       | <RdLo>          | 4/5                   |

**Note**

Result Latency is one less if the result is used as the accumulate register for a subsequent multiply accumulate.



## 16.8 Branches

This section describes the cycle timing behavior for the B, BL, and BLX instructions.

Branches are subject to dynamic, static and return stack predictions. Table 16-11 shows example branch instructions and their cycle timing behavior.

**Table 16-11 Branch instruction cycle timing behavior**

| Example instruction               | Cycles           | Comment                             |
|-----------------------------------|------------------|-------------------------------------|
| B <immed>                         | 0                | Folded dynamic prediction           |
| B<immed>, BL<immed>, BLX<immed>   | 1                | Not-folded dynamic prediction       |
| B<immed>, BL<immed>, BLX<immed>   | 1                | Correct not-taken static prediction |
| B<immed>, BL<immed>, BLX<immed>   | 4                | Correct taken static prediction     |
| B<immed>, BL<immed>, BLX<immed>   | 5-7 <sup>a</sup> | Incorrect dynamic/static prediction |
| BX r14                            | 4                | Correct return stack prediction     |
| BX r14                            | 7                | Incorrect return stack prediction   |
| BX r14                            | 5                | Empty return stack                  |
| BX <cond> r14                     | 5-7 <sup>a</sup> | Conditional return                  |
| BX <cond> <reg>, BLX <cond> <reg> | 1                | If not taken                        |
| BX <cond> <reg>, BLX <cond> <reg> | 5-7 <sup>a</sup> | If taken                            |

- a. Mispredicted branches, including taken unpredicted branches, takes a varying number of cycles to execute depending on their distance from a flag setting instruction. The timing behavior is  

$$\text{Cycle} = \text{MAX}(\text{MaxCycles} - \text{FlagCycleDistance}, \text{MinCycles}).$$

## 16.9 Processor state updating instructions

This section describes the cycle timing behavior for the MSR, MRS, CPS, and SETEND instructions. Table 16-12 shows processor state updating instructions and their cycle timing behavior.

**Table 16-12 Processor state updating instructions cycle timing behavior**

| <b>instruction</b>             | <b>Cycles</b> | <b>Comments</b>            |
|--------------------------------|---------------|----------------------------|
| MRS                            | 1             | All MRS instructions       |
| MSR CPSR_f                     | 1             | MSR to CPSR flags only     |
| MSR                            | 4             | All other MSRs to the CPSR |
| MSR SPSR                       | 5             | All MSRs to the SPSR       |
| CPS <effect> <iflags>          | 1             | Interrupt masks only       |
| CPS <effect> <iflags>, #<mode> | 2             | Mode changing              |
| SETEND                         | 1             | -                          |

## 16.10 Single load and store instructions

This section describes the cycle timing behavior for LDR, LDRT, LDRB, LDRBT, LDRSB, LDRH, LDRSH, STR, STRT, STRB, STRBT, STRH, and PLD instructions.

Table 16-13 shows the cycle timing behavior for stores and loads, other than loads to the PC. You can replace LDR with any of the above single load or store instructions. The following rules apply:

- They are single-cycle issue if a constant offset is used or if a register offset with no shift, or shift by 2 is used. Both the base and any offset register are Early Regs.
- They are two-cycle issue if either a negative register offset or a shift other than LSL #2 is used. Only the offset register is an Early Reg.
- If ARMv6 unaligned support is enabled then accesses to addresses not aligned to the access size generates two memory accesses, and so consume the load/store unit for an additional cycle. This extra cycle is required if the base or the offset is not aligned to the access size, consequently the final address is potentially unaligned, even if the final address turns out to be aligned.
- If ARMv6 unaligned support is enabled and the final access address is unaligned there is an extra cycle of result latency.
- PLD (data preload hint instructions) have cycle timing behavior as for load instructions. Because they have no destination register, the result latency is not-applicable for such instructions.
- The updated base register has a result latency of one. For back-to-back load/store instructions with base write back, the updated base is available to the following load/store instruction with a result latency of 0.

**Table 16-13 Cycle timing behavior for stores and loads, other than loads to the PC**

| Example instruction                     | Cycles | Memory cycles | Result Latency | Comments                             |
|-----------------------------------------|--------|---------------|----------------|--------------------------------------|
| LDR <Rd>, <addr_md_1cycle> <sup>a</sup> | 1      | 1             | 3              | Legacy access / ARMv6 aligned access |
| LDR <Rd>, <addr_md_2cycle> <sup>a</sup> | 2      | 2             | 4              | Legacy access / ARMv6 aligned access |
| LDR <Rd>, <addr_md_1cycle> <sup>a</sup> | 1      | 2             | 3              | Potentially ARMv6 unaligned access   |
| LDR <Rd>, <addr_md_2cycle> <sup>a</sup> | 2      | 3             | 4              | Potentially ARMv6 unaligned access   |
| LDR <Rd>, <addr_md_1cycle> <sup>a</sup> | 1      | 2             | 4              | ARMv6 unaligned access               |
| LDR <Rd>, <addr_md_2cycle> <sup>a</sup> | 1      | 2             | 4              | ARMv6 unaligned access               |

a. See Table 16-15 on page 16-21 for an explanation of <addr\_md\_1cycle> and <addr\_md\_2cycle>.

Table 16-14 shows the cycle timing behavior for loads to the PC.

**Table 16-14 Cycle timing behavior for loads to the PC**

| Example instruction                   | Cycles | Memory cycles | Result Latency | Comments                                  |
|---------------------------------------|--------|---------------|----------------|-------------------------------------------|
| LDR pc, [sp, #cns] (!)                | 4      | 1             | -              | Correctly return stack predicted          |
| LDR pc, [sp], #cns                    | 4      | 1             | -              | Correctly return stack predicted          |
| LDR pc, [sp, #cns] (!)                | 9      | 1             | -              | Return stack mispredicted                 |
| LDR pc, [sp], #cns                    | 9      | 1             | -              | Return stack mispredicted                 |
| LDR <cond> pc, [sp, #cns] (!)         | 8      | 1             | -              | Conditional return, or empty return stack |
| LDR <cond> pc, [sp], #cns             | 8      | 1             | -              | Conditional return, or empty return stack |
| LDR pc, <addr_md_1cycle> <sup>a</sup> | 8      | 1             | -              | -                                         |
| LDR pc, <addr_md_2cycle> <sup>a</sup> | 9      | 2             | -              | -                                         |

a. Table 16-15 on page 16-21 for an explanation of <addr\_md\_1cycle> and <addr\_md\_2cycle>.

Only cycle times for aligned accesses are given because Unaligned accesses to the PC are not supported.

ARM1136JF-S processor includes a three-entry return stack that can predict procedure returns. Any load to the pc with an immediate offset, and the stack pointer r13 as the base register is considered a procedure return.

For condition code failing cycle counts, you must use the cycles for the non-PC destination variants.

Table 16-15 on page 16-21 shows the explanation of <addr\_md\_1cycle> and <addr\_md\_2cycle> used in Table 16-13 on page 16-19 and Table 16-14.

**Table 16-15 <addr\_md\_1cycle> and <addr\_md\_2cycle>  
LDR example instruction explanation**

| Example instruction                   | Early Reg  | Comment                                                                                                    |
|---------------------------------------|------------|------------------------------------------------------------------------------------------------------------|
| <b>&lt;addr_md_1cycle&gt;</b>         |            |                                                                                                            |
| LDR <Rd>, [<Rn>, #cns] (!)            | <Rn>       | If an immediate offset, or a positive register offset with no shift or shift LSL #2, then one-issue cycle. |
| LDR <Rd>, [<Rn>, <Rm>] (!)            | <Rn>, <Rm> |                                                                                                            |
| LDR <Rd>, [<Rn>, <Rm>, LSL #2] (!)    | <Rn>, <Rm> |                                                                                                            |
| LDR <Rd>, [<Rn>], #cns                | <Rn>       |                                                                                                            |
| LDR <Rd>, [<Rn>], <Rm>                | <Rn>, <Rm> |                                                                                                            |
| LDR <Rd>, [<Rn>], <Rm>, LSL #2        | <Rn>, <Rm> |                                                                                                            |
| <b>&lt;addr_md_2cycle&gt;</b>         |            |                                                                                                            |
| LDR <Rd>, [<Rn>, -<Rm>] (!)           | <Rm>       | If negative register offset, or shift other than LSL #2 then two-issue cycles.                             |
| LDR <Rd>, [Rm, -<Rm> <shf> <cns>] (!) | <Rm>       |                                                                                                            |
| LDR <Rd>, [<Rn>], -<Rm>               | <Rm>       |                                                                                                            |
| LDR <Rd>, [<Rn>], -<Rm> <shf> <cns>   | <Rm>       |                                                                                                            |

### 16.10.1 Base register update

The base register update for load or store instructions occurs in the ALU pipeline. To prevent an interlock for back-to-back load or store instructions reusing the same base register, there is a local forwarding path to recycle the updated base register around the ADD stage.

For example, the following instruction sequence take three cycles to execute:

```
LDR r5, [r2, #4]!
LDR r6, [r2, #0x10]!
LDR r7, [r2, #0x20]!
```

## 16.11 Load and Store Double instructions

This section describes the cycle timing behavior for the LDRD and STRD instructions

The LDRD and STRD instructions:

- Are two-cycle issue if either a negative register offset or a shift other than LSL #2 is used. Only the offset register is an Early Reg.
- Are single-cycle issue if either a constant offset is used or if a register offset with no shift, or shift by 2 is used. Both the base and any offset register are Early Regs.
- Take only one memory cycle if the address is doubleword aligned.
- Take two memory cycles if the address is not doubleword aligned.

The updated base register has a result latency of one. For back-to-back load/store instructions with base write back, the updated base is available to the following load/store instruction with a result latency of 0.

To prevent instructions after a STRD from writing to a register before it has stored that register, the STRD registers have a lock latency that determines how many cycles it is before a subsequent instruction which writes to that register can start.

Table 16-16 shows the cycle timing behavior for LDRD and STRD instructions.

**Table 16-16 Load and Store Double instructions cycle timing behavior**

| Example instruction                    | Cycles | Memory cycles | Result Latency (LDRD) | Register lock latency (STRD) |
|----------------------------------------|--------|---------------|-----------------------|------------------------------|
| <b>Address is double-word aligned</b>  |        |               |                       |                              |
| LDRD r1, <addr_md_1cyc1e> <sup>a</sup> | 1      | 1             | 3/3                   | 1,2                          |
| LDRD r1, <addr_md_2cyc1e> <sup>a</sup> | 2      | 2             | 4/4                   | 2,3                          |
| <b>Address not double-word aligned</b> |        |               |                       |                              |
| LDRD r1, <addr_md_1cyc1e> <sup>a</sup> | 1      | 2             | 3/4                   | 1,2                          |
| LDRD r1, <addr_md_2cyc1e> <sup>a</sup> | 2      | 3             | 4/5                   | 2,3                          |

a. Table 16-17 on page 16-23 for an explanation of <addr\_md\_1cyc1e> and <addr\_md\_2cyc1e>.

Table 16-17 on page 16-23 shows the explanation of <addr\_md\_1cyc1e> and <addr\_md\_2cyc1e> used in Table 16-16.

**Table 16-17 <addr\_md\_1cycle> and <addr\_md\_2cycle>  
LDRD example instruction explanation**

| Example instruction                    | Early Reg  | Comment                                                                                                    |
|----------------------------------------|------------|------------------------------------------------------------------------------------------------------------|
| <hr/>                                  |            |                                                                                                            |
| <addr_md_1cycle>                       |            |                                                                                                            |
| LDRD <Rd>, [<Rn>, #cns] (!)            | <Rn>       | If an immediate offset, or a positive register offset with no shift or shift LSL #2, then one-issue cycle. |
| LDRD <Rd>, [<Rn>, <Rm>] (!)            | <Rn>, <Rm> |                                                                                                            |
| LDRD <Rd>, [<Rn>, <Rm>, LSL #2] (!)    | <Rn>, <Rm> |                                                                                                            |
| LDRD <Rd>, [<Rn>], #cns                | <Rn>       |                                                                                                            |
| LDRD <Rd>, [<Rn>], <Rm>                | <Rn>, <Rm> |                                                                                                            |
| LDRD <Rd>, [<Rn>], <Rm>, LSL #2        | <Rn>, <Rm> |                                                                                                            |
| <hr/>                                  |            |                                                                                                            |
| <addr_md_2cycle>                       |            |                                                                                                            |
| LDRD <Rd>, [<Rn>, -<Rm>] (!)           | <Rm>       | If negative register offset, or shift other than LSL #2 then two-issue cycles.                             |
| LDRD Rd, [<Rm>, -<Rm> <shf> <cns>] (!) | <Rm>       |                                                                                                            |
| LDRD <Rd>, [<Rn>], -<Rm>               | <Rm>       |                                                                                                            |
| LDRD< Rd>, [Rn], -<Rm> <shf> <cns>     | <Rm>       |                                                                                                            |
| <hr/>                                  |            |                                                                                                            |

## 16.12 Load and Store Multiple Instructions

This section describes the cycle timing behavior for the LDM and STM instructions.

These instructions take one cycle to issue but then use multiple memory cycles to load/store all the registers. Because the memory datapath is 64-bits wide, two registers can be loaded or stored on each cycle. Following non-dependent, non-memory instructions can execute in the integer pipeline while these instructions complete. A dependent instruction is one that either:

- writes a register that has not yet been stored
- reads a register that has not yet been loaded.

Before a load or store multiple can begin all the registers in the register list must be available. For example, a STM cannot begin until all outstanding loads for registers in the register list have completed.

To prevent instructions after a store multiple from writing to a register before a store multiple has stored that register, the register list has a lock latency that determines how many cycles it is before a subsequent instruction which writes to that register can start.

### 16.12.1 Load and Store Multiples, other than load multiples including the PC

In all cases the base register, Rx, is an Early Reg.

Table 16-18 shows the cycle timing behavior of load and store multiples including the PC.

**Table 16-18 Cycle timing behavior of Load and Store Multiples, other than load multiples including the PC**

| Example Instruction                    | Cycles | Memory cycles | Result Latency (LDM) | Register Lock Latency (STM) |
|----------------------------------------|--------|---------------|----------------------|-----------------------------|
| <b>First address 64-bit aligned</b>    |        |               |                      |                             |
| LDMIA Rx, {r1}                         | 1      | 1             | 3                    | 1                           |
| LDMIA Rx, {r1, r2}                     | 1      | 1             | 3,3                  | 1,2                         |
| LDMIA Rx, {r1, r2, r3}                 | 1      | 2             | 3,3,4                | 1,2,2                       |
| LDMIA Rx, {r1, r2, r3, r4}             | 1      | 2             | 3,3,4,4              | 1,2,2,3                     |
| LDMIA Rx, {r1, r2, r3, r4, r5}         | 1      | 3             | 3,3,4,4,5            | 1,2,2,3,3                   |
| LDMIA Rx, {r1, r2, r3, r4, r5, r6}     | 1      | 3             | 3,3,4,4,5,5          | 1,2,2,3,3,4                 |
| LDMIA Rx, {r1, r2, r3, r4, r5, r6, r7} | 1      | 4             | 3,3,4,4,5,5,6        | 1,2,2,3,3,4,4               |



**Table 16-18 Cycle timing behavior of Load and Store Multiples, other than load multiples including the PC (continued)**

| Example Instruction                     | Cycles | Memory cycles | Result Latency (LDM) | Register Lock Latency (STM) |
|-----------------------------------------|--------|---------------|----------------------|-----------------------------|
| <b>First address not 64-bit aligned</b> |        |               |                      |                             |
| LDMIA Rx, {r1}                          | 1      | 1             | 3                    | 1                           |
| LDMIA Rx, {r1, r2}                      | 1      | 2             | 3,4                  | 1,2                         |
| LDMIA Rx, {r1, r2, r3}                  | 1      | 2             | 3,4,4                | 1,2,2                       |
| LDMIA Rx, {r1, r2, r3, r4}              | 1      | 3             | 3,4,4,5              | 1,2,2,3                     |
| LDMIA Rx, {r1, r2, r3, r4, r5}          | 1      | 3             | 3,4,4,5,5            | 1,2,2,3,4                   |
| LDMIA Rx, {r1, r2, r3, r4, r5, r6}      | 1      | 4             | 3,4,4,5,5,6          | 1,2,2,3,4,4                 |
| LDMIA Rx, {r1, r2, r3, r4, r5, r6, r7}  | 1      | 4             | 3,4,4,5,5,6,6        | 1,2,2,3,4,4,5               |

### 16.12.2 Load Multiples, where the PC is in the register list

If a LDM loads the PC then the PC access is performed first to accelerate the branch, followed by the rest of the register loads. The cycle timings and all register load latencies for LDMs with the pc in the list are one greater than the cycle times for the same LDM without the PC in the list.

ARM1136JF-S processor includes a three-entry return stack which can predict procedure returns. Any LDM to the pc with the stack point (r13) as the base register, and which does not restore the SPSR to the CPSR, is predicted as a procedure return.

For condition code failing cycle counts, the cycles for the non-PC destination variants must be used. These are all single-cycle issue, consequently a condition code failing LDM to the PC takes one cycle.

In all cases the base register, Rx, is an Early Reg, and requires an extra cycle of result latency to provide its value.

Table 16-19 shows the cycle timing behavior of Load Multiples, where the PC is in the register list.

**Table 16-19 Cycle timing behavior of Load Multiples, where the PC is in the register list**

| Example instruction        | Cycles | Memory Cycles    | Result Latency | Comments                                  |
|----------------------------|--------|------------------|----------------|-------------------------------------------|
| LDMIA sp!, {...,pc}        | 4      | 1+n <sup>a</sup> | 4,...          | Correctly return stack predicted          |
| LDMIA sp!, {...,pc}        | 9      | 1+n <sup>a</sup> | 4,...          | Return stack mispredicted                 |
| LDMIA <cond> sp!, {...,pc} | 9      | 1+n <sup>a</sup> | 4,...          | Conditional return, or empty return stack |
| LDMIA rx, {...,pc}         | 8      | 1+n <sup>a</sup> | 4,...          | Not return stack predicted                |

a. Where n is the number of memory cycles for this instruction if the pc had not been in the register list.

### 16.12.3 Example Interlocks

The following sequence that has an LDM instruction take five cycles, because r3 has a result latency of four cycles:

```
LDMIA r0, {r1-r7}
ADD r10, r10, r3
```

The following that has an STM instruction takes five cycles to execute, because r6 has a register lock latency of four cycles:

```
STMIA r0, {r1-r7}
ADD r6, r10, r11
```

## 16.13 RFE and SRS instructions

This section describes the cycle timing for the RFE and SRS instructions.

These instructions return from an exception and save exception return state respectively. The RFE instruction always requires two memory cycles. It first loads the SPSR value from the stack, and then the return address. The SRS instruction takes one or two memory cycles depending on double-word alignment first address location.

In all cases the base register is an Early Reg, and requires an extra cycle of result latency to provide its value.

Table 16-20 shows the cycle timing behavior for RFE and SRS instructions.

**Table 16-20 RFE and SRS instructions cycle timing behavior**

| <b>Example Instruction</b>             | <b>Cycles</b> | <b>Memory Cycles</b> |
|----------------------------------------|---------------|----------------------|
| <b>Address double-word aligned</b>     |               |                      |
| RFEIA <Rn>                             | 9             | 2                    |
| SRSIA #<mode>                          | 1             | 1                    |
| <b>Address not double-word aligned</b> |               |                      |
| RFEIA <Rn>                             | 9             | 2                    |
| SRSIA #<mode>                          | 1             | 2                    |

## 16.14 Synchronization instructions

This section describes the cycle timing behavior for the SWP, SWPB, LDREX, and STREX instructions

In all cases the base register, Rn, is an Early Reg, and requires an extra cycle of result latency to provide its value. Table 16-21 shows the synchronization instructions cycle timing behavior.

**Table 16-21 Synchronization Instructions cycle timing behavior**

| Instruction            | Cycles | Memory Cycles | Result Latency |
|------------------------|--------|---------------|----------------|
| SWP Rd, <Rm>, [Rn]     | 2      | 2             | 3              |
| SWPB Rd, <Rm>, [Rn]    | 2      | 2             | 3              |
| LDREX <Rd>, [Rn]       | 1      | 1             | 3              |
| STREX, Rd>, <Rm>, [Rn] | 1      | 1             | 3              |

## 16.15 Coprocessor instructions

This section describes the cycle timing behavior for the CDP, LDC, STC, LDCL, STCL, MCR, MRC, MCRR, and MRRC instructions.

The precise timing of coprocessor instructions is tightly linked with the behavior of the relevant coprocessor. The numbers below are best case numbers. For LDC/STC instructions the coprocessor can determine how many words are required. Table 16-22 shows the coprocessor instructions cycle timing behavior.

**Table 16-22 Coprocessor Instructions cycle timing behavior**

| Instruction | Cycles | Memory cycles | Result Latency |
|-------------|--------|---------------|----------------|
| MCR         | 1      | 1             | -              |
| MCRR        | 1      | 1             | -              |
| MRC         | 1      | 1             | 3              |
| MRRC        | 1      | 1             | 3/3            |
| LDC/LDCL    | 1      | As required   | -              |
| STC/STCL    | 1      | As required   | -              |
| CDP         | 1      | 1             | -              |

## 16.16 SWI, BKPT, Undefined, Prefetch Aborted instructions

This section describes the cycle timing behavior for SWI, Undefined Instruction, BKPT and Prefetch Abort.

In all cases the exception is taken in the WBex stage of the pipeline. SWI and most Undefined instructions which fail their condition codes take one cycle. A small number of undefined instructions which fail their condition codes take two cycles. Table 16-23 shows the SWI, BKPT, undefined, prefetch aborted instructions cycle timing behavior.

**Table 16-23 SWI, BKPT, undefined, prefetch aborted instructions cycle timing behavior**

| <b>Instruction</b>    | <b>Cycles</b> |
|-----------------------|---------------|
| SWI                   | 8             |
| BKPT                  | 8             |
| Prefetch Abort        | 8             |
| Undefined Instruction | 8             |

## **16.17 Thumb instructions**

The cycle timing behavior for Thumb instructions follow the ARM equivalent instruction cycle timing behavior.

Thumb BL instructions that are encoded as two Thumb instructions, can be dynamically predicted. The prediction occurs on the second part of the BL pair, consequently a correct prediction takes two cycles.





# Chapter 17

## AC Characteristics

This chapter gives the timing diagrams and timing parameters for the ARM1136JF-S processor. This chapter contains the following sections:

- *ARM1136JF-S timing diagrams* on page 17-2
- *ARM1136JF-S timing parameters* on page 17-3.

## 17.1 ARM1136JF-S timing diagrams

The AMBA bus interface of the ARM1136JF-S processor conforms to the *AMBA Specification*. Refer to this document for the relevant timing diagrams.

## 17.2 ARM1136JF-S timing parameters

The maximum timing parameter or constraint delay for each ARM1136JF-S processor signal applied to the SoC is given as a percentage in Table 17-1 to Table 17-8 on page 17-7. The input delay columns provide the maximum and minimum time as a percentage of the ARM1136JF-S processor clock cycle given to the SoC for that signal.

———— **Note** —————

The maximum delay timing parameter or constraint allowed for all ARM1136JF-S processor output signals enables 60% of the ARM1136JF-S processor clock cycle to the SoC.

Table 17-1 shows the AHB-Lite bus interface timing parameters.

**Table 17-1 AHB-Lite bus interface timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>   |
|-------------------------|-------------------------|----------------------|
| Clock uncertainty       | 40%                     | <b>HCLKIRWEN</b>     |
| Clock uncertainty       | 40%                     | <b>HCLKDEN</b>       |
| Clock uncertainty       | 40%                     | <b>HCLKPEN</b>       |
| Clock uncertainty       | 70%                     | <b>HSYNCENIRW</b>    |
| Clock uncertainty       | 70%                     | <b>HSYNCENPD</b>     |
| Clock uncertainty       | 70%                     | <b>SYNCENIRW</b>     |
| Clock uncertainty       | 70%                     | <b>SYNCENPD</b>      |
| Clock uncertainty       | 50%                     | <b>HREADYI</b>       |
| Clock uncertainty       | 70%                     | <b>HRESPI</b>        |
| Clock uncertainty       | 70%                     | <b>HRDATAI[63:0]</b> |
| Clock uncertainty       | 50%                     | <b>HREADYR</b>       |
| Clock uncertainty       | 70%                     | <b>HRESPR</b>        |
| Clock uncertainty       | 70%                     | <b>HRDATAR[63:0]</b> |
| Clock uncertainty       | 50%                     | <b>HREADYW</b>       |
| Clock uncertainty       | 70%                     | <b>HRESPW[2:0]</b>   |
| Clock uncertainty       | 50%                     | <b>HREADYP</b>       |

**Table 17-1 AHB-Lite bus interface timing parameters (continued)**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>   |
|-------------------------|-------------------------|----------------------|
| Clock uncertainty       | 70%                     | <b>HRESPP</b>        |
| Clock uncertainty       | 70%                     | <b>HRDATAP[31:0]</b> |
| Clock uncertainty       | 50%                     | <b>HREADYD</b>       |
| Clock uncertainty       | 70%                     | <b>HRESPD</b>        |
| Clock uncertainty       | 70%                     | <b>HRDATAD[63:0]</b> |

Table 17-2 shows the coprocessor port timing parameters

**Table 17-2 Coprocessor port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>      |
|-------------------------|-------------------------|-------------------------|
| Clock uncertainty       | 70%                     | <b>CPALENGTHHOLD</b>    |
| Clock uncertainty       | 70%                     | <b>CPAACCEPT</b>        |
| Clock uncertainty       | 70%                     | <b>CPAACCEPTHOLD</b>    |
| Clock uncertainty       | 70%                     | <b>CPASTDATAV</b>       |
| Clock uncertainty       | 70%                     | <b>CPALENGTH[3:0]</b>   |
| Clock uncertainty       | 70%                     | <b>CPALENGTHT[3:0]</b>  |
| Clock uncertainty       | 70%                     | <b>CPAACCEPTT[3:0]</b>  |
| Clock uncertainty       | 70%                     | <b>CPASTDATA[63:0]</b>  |
| Clock uncertainty       | 70%                     | <b>CPASTDATAT[3:0]</b>  |
| Clock uncertainty       | 70%                     | <b>CPAPRESENT[11:0]</b> |

Table 17-3 shows the ETM interface port timing parameters

**Table 17-3 ETM interface port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>      |
|-------------------------|-------------------------|-------------------------|
| Clock uncertainty       | 60%                     | <b>ETMPWRUP</b>         |
| Clock uncertainty       | 60%                     | <b>nETMWFIREADY</b>     |
| Clock uncertainty       | 60%                     | <b>ETMEXTOUT[1:0]</b>   |
| Clock uncertainty       | 60%                     | <b>ETMCPADATA[31:0]</b> |

Table 17-4 shows the interrupt port timing parameters

**Table 17-4 Interrupt port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>    |
|-------------------------|-------------------------|-----------------------|
| Clock uncertainty       | 60%                     | <b>nFIQ</b>           |
| Clock uncertainty       | 60%                     | <b>nIRQ</b>           |
| Clock uncertainty       | 60%                     | <b>INTSYNCEN</b>      |
| Clock uncertainty       | 60%                     | <b>IRQADDRV</b>       |
| Clock uncertainty       | 60%                     | <b>IRQADDRVSYNCEN</b> |
| Clock uncertainty       | 60%                     | <b>IRQADDR[31:2]</b>  |

Table 17-5 shows the debug timing parameters

**Table 17-5 Debug timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>     |
|-------------------------|-------------------------|------------------------|
| Clock uncertainty       | 40%                     | <b>DBGTCKEN</b>        |
| Clock uncertainty       | 40%                     | <b>FREEDBGTCKEN</b>    |
| Clock uncertainty       | 50%                     | <b>DBGMANID[10:0]</b>  |
| Clock uncertainty       | 50%                     | <b>DBGTDI</b>          |
| Clock uncertainty       | 50%                     | <b>DBGTMS</b>          |
| Clock uncertainty       | 50%                     | <b>DBGVERSION[3:0]</b> |

**Table 17-5 Debug timing parameters (continued)**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b> |
|-------------------------|-------------------------|--------------------|
| Clock uncertainty       | 60%                     | <b>DBGnTRST</b>    |
| Clock uncertainty       | 60%                     | <b>EDBGRQ</b>      |
| Clock uncertainty       | 60%                     | <b>DBGEN</b>       |

Table 17-6 shows the test port timing parameters

**Table 17-6 test port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b>     |
|-------------------------|-------------------------|------------------------|
| Clock uncertainty       | 20%                     | <b>SCANMODE</b>        |
| Clock uncertainty       | 20%                     | <b>SE</b>              |
| Clock uncertainty       | 20%                     | <b>SI<sup>*a</sup></b> |
| Clock uncertainty       | 20%                     | <b>MUXINSEL</b>        |
| Clock uncertainty       | 20%                     | <b>MUXOUTSEL</b>       |
| Clock uncertainty       | 60%                     | <b>MBISTADDR[12:0]</b> |
| Clock uncertainty       | 60%                     | <b>MBISTCE[22:0]</b>   |
| Clock uncertainty       | 60%                     | <b>MBISTDIN[63:0]</b>  |
| Clock uncertainty       | 60%                     | <b>MBISTWE</b>         |
| Clock uncertainty       | 60%                     | <b>MTESTON</b>         |

a. The asterisk in

Table 17-7 shows the static configuration signal port timing parameters

**Table 17-7 Static configuration signal port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b> |
|-------------------------|-------------------------|--------------------|
| Clock uncertainty       | 60%                     | <b>BIGENDINIT</b>  |

**Table 17-7 Static configuration signal port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b> |
|-------------------------|-------------------------|--------------------|
| Clock uncertainty       | 60%                     | <b>UBITINIT</b>    |
| Clock uncertainty       | 60%                     | <b>INITRAM</b>     |
| Clock uncertainty       | 60%                     | <b>VINITHI</b>     |

Table 17-8 shows the reset port timing parameters.

**Table 17-8 Reset port timing parameters**

| <b>Input delay Min.</b> | <b>Input delay Max.</b> | <b>Signal name</b> |
|-------------------------|-------------------------|--------------------|
| Clock uncertainty       | 20%                     | nRESETIN           |
| Clock uncertainty       | 20%                     | nPORESETIN         |
| Clock uncertainty       | 20%                     | HRESETIRWn         |
| Clock uncertainty       | 20%                     | HRESETPDn          |





# Appendix A

## Signal Descriptions

This appendix lists and describes the ARM1136JF-S signals. It contains the following sections:

- *Global signals* on page A-2
- *Static configuration signals* on page A-3
- *Interrupt signals (including VIC interface)* on page A-4
- *AHB interface signals* on page A-5
- *Coprocessor interface signals* on page A-14
- *Coprocessor interface signals* on page A-14
- *Debug interface signals (including JTAG)* on page A-16
- *ETM interface signals* on page A-17
- *Test signals* on page A-18.

———— **Note** —————

The output signals shown in Table A-1 on page A-2 to Table A-14 on page A-18 are set to 0 on reset unless otherwise stated.

## A.1 Global signals

Table A-1 lists the ARM1136JF-S global signals.

Free clocks are the free running clocks with minimal insertion delay for clocking the clock gating circuitry. Free clocks must be balanced with the incoming clock signal, but not with the clocks clocking the core logic.

**Table A-1 Global signals**

| <b>Name</b>                  | <b>Direction</b> | <b>Description</b>                                                                |
|------------------------------|------------------|-----------------------------------------------------------------------------------|
| <b>CLKIN</b>                 | Input            | Core clock                                                                        |
| <b>FREECLKIN</b>             | Input            | Free version of the core clock                                                    |
| <b>FREEHCLKIRW</b>           | Input            | Free version of <b>HCLKIRW</b>                                                    |
| <b>FREEHCLKPD</b>            | Input            | Free version of <b>HCLKPD</b>                                                     |
| <b>HCLKDEN</b>               | Input            | Clock enable for the DMA port to enable it to be clocked at a reduced rate        |
| <b>HCLKIRW</b>               | Input            | <b>HCLK</b> for the I/R/W ports                                                   |
| <b>HCLKIRWEN</b>             | Input            | <b>HCLKEN</b> for the I/R/W ports                                                 |
| <b>HCLKPD</b>                | Input            | <b>HCLK</b> for the P/D ports                                                     |
| <b>HCLKPEN</b>               | Input            | Clock enable for the peripheral port to enable it to be clocked at a reduced rate |
| <b>HRESETIRW<sub>n</sub></b> | Input            | <b>HRESET<sub>n</sub></b> for the I/R/W ports                                     |
| <b>HRESETPD<sub>n</sub></b>  | Input            | <b>HRESET<sub>n</sub></b> for the P/D ports                                       |
| <b>HSYNCENIRW</b>            | Input            | Synchronous control <b>HCLK</b> domain for I/R/W ports                            |
| <b>HSYNCENPD</b>             | Input            | Synchronous control <b>HCLK</b> domain for P/D ports                              |
| <b>nPORESETIN</b>            | Input            | Power on reset (resets debug logic)                                               |
| <b>nRESETIN</b>              | Input            | Core reset                                                                        |
| <b>STANDBYWFI</b>            | Output           | Indicates that the ARM1136JF-S processor is in Standby mode                       |
| <b>SYNCENIRW</b>             | Input            | Synchronous control <b>CLKIN</b> domain for IRW ports                             |
| <b>SYNCENPD</b>              | Input            | Synchronous control <b>CLKIN</b> domain for PD ports                              |

## A.2 Static configuration signals

Table A-2 lists the ARM1136JF-S static configuration signals.

**Table A-2 Static configuration signals**

| <b>Name</b>          | <b>Direction</b> | <b>Description</b>                                                  |
|----------------------|------------------|---------------------------------------------------------------------|
| <b>COREASID[7:0]</b> | Output           | ASID used by the integer processor exported to memory system        |
| <b>DMAASID[7:0]</b>  | Output           | ASID used by the DMA exported to memory system                      |
| <b>BIGENDINIT</b>    | Input            | When HIGH, indicates v5 Bigendian mode                              |
| <b>CFGBIGENDIRW</b>  | Output           | Current state of the CP15 Bigend bit synchronized to <b>HCLKIRW</b> |
| <b>CFGBIGENDPD</b>   | Output           | Current state of the CP15 Bigend bit synchronized to <b>HCLKPD</b>  |
| <b>INTRAM</b>        | Input            | When HIGH, indicates ITCM enabled at address 0x0                    |
| <b>UBITINIT</b>      | Input            | When HIGH, indicates ARMv6 Unaligned behavior                       |
| <b>VINITHI</b>       | Input            | When HIGH, indicates High-Vecs mode                                 |

### A.3 Interrupt signals (including VIC interface)

Table A-3 lists the interrupt signals, including those used with the VIC interface.

**Table A-3 Interrupt signals**

| <b>Name</b>           | <b>Direction</b> | <b>Description</b>                                                  |
|-----------------------|------------------|---------------------------------------------------------------------|
| <b>INTSYNCEN</b>      | Input            | Indicates that VIC interface is asynchronous.                       |
| <b>IRQACK</b>         | Output           | Interrupt acknowledge.                                              |
| <b>IRQADDR[31:2]</b>  | Input            | Address of the IRQ.                                                 |
| <b>IRQADDRV</b>       | Input            | Indicates <b>IRQADDR</b> is valid.                                  |
| <b>IRQADDRVSYNCEN</b> | Input            | Indicates that VIC <b>IRQADDRV</b> requires synchronizer.           |
| <b>nFIQ</b>           | Input            | Fast interrupt request.                                             |
| <b>nIRQ</b>           | Input            | Interrupt request.                                                  |
| <b>nDMAIRQ</b>        | Output           | Interrupt request by DMA. On reset this pin is set to 1.            |
| <b>nPMUIRQ</b>        | Output           | Interrupt request by system metrics. On reset this pin is set to 1. |

## A.4 AHB interface signals

The AHB interface ports operate using standard AHB-lite signals, extended for ARMv6.

This extension includes the following signals:

|                       |                                                                                                                                                                                                                                                                                       |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HRESP[2]</b>       | Signals an exclusive access failure.                                                                                                                                                                                                                                                  |
| <b>HPROT[4:2]</b>     | Used to signal the memory types.                                                                                                                                                                                                                                                      |
| <b>HPROT[5]</b>       | Signals that the access is an exclusive access.                                                                                                                                                                                                                                       |
| <b>HUNALIGN</b>       | Indicates that the access is unaligned and requires <b>HBSTRB</b> information.                                                                                                                                                                                                        |
| <b>HBSTRB[7:0]</b>    | Byte lane strobes.                                                                                                                                                                                                                                                                    |
| <b>HSIDEBAND[0]</b>   | Sharable bit for that access                                                                                                                                                                                                                                                          |
| <b>HSIDEBAND[3:1]</b> | Inner memory system attributes. Can be used to replace <b>HPROT[4:2]</b> if the level two system requires inner cache attributes. The encoding of <b>HSIDEBAND[3:1]</b> is the same as <b>HPROT[4:2]</b> , but refers to inner cache attributes as opposed to outer cache attributes. |

The signal names have a one or two-letter suffix that designates the appropriate port as shown in Table A-4.

**Table A-4 Port signal name suffixes**

| Port                    | Prefix | Comment                 |
|-------------------------|--------|-------------------------|
| Instruction fetch       | I      | Read-only               |
| Data read               | R      | Read-only               |
| Data write              | W      | Write only              |
| Data read or data write | RW     | Read-only or write-only |
| DMA                     | D      | Bidirectional           |
| Peripheral              | P      | Bidirectional           |

### A.4.1 Instruction fetch port signals

The instruction fetch port is a 64-bit wide AHB-lite port that is read-only.

Table A-5 lists the ARM1136JF-S instruction fetch port signals.

**Table A-5 Instruction fetch port signals**

| <b>Name</b>            | <b>Direction</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HADDRI[31:0]</b>    | Output           | The 32-bit system instruction fetch port address bus.                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>HBSTRBI[7:0]</b>    | Output           | Indicates which byte lanes are valid.                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>HBURSTI[2:0]</b>    | Output           | Indicates if the transfer forms part of a burst. Four-beat bursts are supported and the burst can be either incrementing or wrapping.                                                                                                                                                                                                                                                                                                            |
| <b>HMASTLOCKI</b>      | Output           | Instruction fetch port lock signal                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>HPROTI[5:0]</b>     | Output           | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. The signals indicate if: <ul style="list-style-type: none"> <li>• the transfer is an opcode fetch or data access</li> <li>• the transfer is a Supervisor mode access or User mode access</li> <li>• the current access is Cachable or Bufferable.</li> </ul> |
| <b>HRDATAI[63:0]</b>   | Input            | The read data bus is used to transfer data and instructions from bus slaves to the bus master during read operations.                                                                                                                                                                                                                                                                                                                            |
| <b>HREADYI</b>         | Input            | When HIGH the <b>HREADYI</b> signal indicates that a transfer has finished on the bus. You can drive this signal LOW to extend a transfer.                                                                                                                                                                                                                                                                                                       |
| <b>HRESPI</b>          | Input            | The transfer response provides additional information on the status of a transfer. Two responses are provided:<br>0 = Okay<br>1 = Error.<br>Connects to <b>HRESP[0]</b> .                                                                                                                                                                                                                                                                        |
| <b>HSIDEBANDI[3:0]</b> | Output           | Signals shareable and inner cachable                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>HSIZE[2:0]</b>      | Output           | Indicates the size of the instruction fetch port transfer: <ul style="list-style-type: none"> <li>• byte (8-bit)</li> <li>• halfword (16-bit)</li> <li>• word (32-bit)</li> <li>• doubleword (64-bit).</li> </ul> The protocol enables larger transfer sizes up to a maximum of 1024 bits. On reset these pins are set to b011.                                                                                                                  |

Table A-5 Instruction fetch port signals (continued)

| Name                | Direction | Description                                                                                                                                                                 |
|---------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HTRANSI[1:0]</b> | Output    | Indicates the type of the current transfer on the instruction fetch port, which can be:<br>b00 = Idle<br>b10 = Nonsequential<br>b11 = Sequential<br>b01 = Busy is not used. |
| <b>HUNALIGNI</b>    | Output    | When HIGH, indicates that the access is unaligned and that <b>HBSTRBI</b> information is required.                                                                          |
| <b>HWRITEI</b>      | Output    | When HIGH this signal indicates a write transfer on the instruction fetch port, and when LOW a read transfer.                                                               |

#### A.4.2 Data read port signals

The data read port is a 64-bit wide AHB-lite port that is read/write.

For AHB protocol reasons, locked reads and writes of SWP or SWPB instructions must occur on the same bus. Because of this, the data read port can perform writes of SWP and SWPB instructions.

Table A-6 lists the ARM1136JF-S data read port signals.

Table A-6 Data read port signals

| Name                 | Direction | Description                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HADDRR[31:0]</b>  | Output    | The 32-bit system data read port address bus.                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>HBSTRBR[7:0]</b>  | Output    | Indicates which byte lanes are valid.                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>HBURSTR[2:0]</b>  | Output    | Indicates if the transfer forms part of a burst. Four-beat bursts are supported and the burst can be either incrementing or wrapping.                                                                                                                                                                                                                                                                                                            |
| <b>HMASTLOCKR</b>    | Output    | Data read port lock signal.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>HPROTR[5:0]</b>   | Output    | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. The signals indicate if: <ul style="list-style-type: none"> <li>the transfer is an opcode fetch or data access</li> <li>if the transfer is a Supervisor mode access or User mode access</li> <li>if the current access is Cachable or Bufferable.</li> </ul> |
| <b>HRDATAR[63:0]</b> | Input     | Data read port read data bus.                                                                                                                                                                                                                                                                                                                                                                                                                    |

Table A-6 Data read port signals (continued)

| Name                   | Direction | Description                                                                                                                                                                                                                                                                        |
|------------------------|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HREADYR</b>         | Input     | Data read port address ready.                                                                                                                                                                                                                                                      |
| <b>HRESPR</b>          | Input     | The transfer response provides additional information on the status of a transfer. Two responses are provided:<br>0 = Okay<br>1 = Error.<br>Connects to <b>HRESP[0]</b> .                                                                                                          |
| <b>HSIDEBANDR[3:0]</b> | Output    | Signals shareable and inner cachable                                                                                                                                                                                                                                               |
| <b>HSIZER[2:0]</b>     | Output    | Indicates the size of the data read port transfer: <ul style="list-style-type: none"> <li>• byte (8-bit)</li> <li>• halfword (16-bit)</li> <li>• word (32-bit)</li> <li>• doubleword (64-bit).</li> </ul> The protocol enables larger transfer sizes up to a maximum of 1024 bits. |
| <b>HTRANSR[1:0]</b>    | Output    | Indicates the type of the current transfer on the data read port:<br>b00 = Idle<br>b10 = Nonsequential<br>b11 = Sequential<br>b01 = Busy is not used.                                                                                                                              |
| <b>HUNALIGNR</b>       | Output    | When HIGH, indicates that the access is unaligned and that <b>HBSTRBR</b> information is required.                                                                                                                                                                                 |
| <b>HWDATAR[63:0]</b>   | Output    | The data read port write data bus is used to transfer data from the bus master to the bus slave during write operations for SWP and SWPB instructions.                                                                                                                             |
| <b>HWRITER</b>         | Output    | When HIGH this signal indicates a write transfer of a SWP or SWPB instruction on the data read port, and when LOW a read transfer.                                                                                                                                                 |

### A.4.3 Data write port

The data write port is a 64-bit wide AHB-lite port that is write-only.

Table A-7 on page A-9 lists the data write port signals.



Table A-7 Data write port signals

| Name                   | Direction | Description                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HADDRW[31:0]</b>    | Output    | The 32-bit system data write port address bus.                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>HBSTRBW[7:0]</b>    | Output    | Indicates which byte lanes are valid.                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>HBURSTW[2:0]</b>    | Output    | Indicates if the transfer forms part of a burst. Four-beat bursts are supported and the burst can be either incrementing or wrapping.                                                                                                                                                                                                                                                                                                       |
| <b>HMASTLOCKW</b>      | Output    | Data write port lock signal.                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>HPROTW[5:0]</b>     | Output    | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection. The signals indicate if: <ul style="list-style-type: none"> <li>the transfer is an opcode fetch or data access</li> <li>the transfer is a Supervisor mode access or User mode access</li> <li>the current access is Cachable or Bufferable.</li> </ul> |
| <b>HREADYW</b>         | Input     | When HIGH the <b>HREADYW</b> signal indicates that a transfer has finished on the data write port bus. You can drive this signal LOW to extend a transfer.                                                                                                                                                                                                                                                                                  |
| <b>HRESPW[2:0]</b>     | Input     | The transfer response provides additional information on the status of a transfer. Five responses are provided: <ul style="list-style-type: none"> <li>b000 = Okay</li> <li>b001 = Error</li> <li>b010 = Retry</li> <li>b011 = Split</li> <li>b100 = Xfail.</li> </ul>                                                                                                                                                                      |
| <b>HSIDEBANDW[3:0]</b> | Output    | Signals shareable and inner cachable                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>HSIZEW[2:0]</b>     | Output    | Indicates the size of the data write port transfer: <ul style="list-style-type: none"> <li>byte (8-bit)</li> <li>halfword (16-bit)</li> <li>word (32-bit)</li> <li>doubleword (64-bit).</li> </ul> The protocol enables larger transfer sizes up to a maximum of 1024 bits.                                                                                                                                                                 |

Table A-7 Data write port signals (continued)

| Name                 | Direction | Description                                                                                                                                                          |
|----------------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HTRANSW[1:0]</b>  | Output    | Indicates the type of the current transfer on the data write port, which can be:<br>b00 = Idle<br>b10 = Nonsequential<br>b11 = Sequential<br>b01 = Busy is not used. |
| <b>HUNALIGNW</b>     | Output    | When HIGH, indicates that the access is unaligned and that <b>HBSTRBW</b> information is required.                                                                   |
| <b>HWDATAW[63:0]</b> | Output    | The data write port write data bus is used to transfer data from the bus master to the bus slave during write operations.                                            |
| <b>HWRITEW</b>       | Output    | When HIGH this signal indicates a write transfer on the data write port, and when LOW a read transfer. On reset this pin is set to 1.                                |
| <b>WRITEBACK</b>     | Output    | Indicates that the current transaction is a cache line eviction.                                                                                                     |

#### A.4.4 Peripheral port signals

The peripheral port is a 32-bit wide AHB-lite port that is read/write.

Table A-8 lists the peripheral port signals.

Table A-8 Peripheral port signals

| Name                | Direction | Description                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HADDRP[31:0]</b> | Output    | The 32-bit system peripheral port address bus.                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>HBSTRBP[7:0]</b> | Output    | Indicates which byte lanes are valid.                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>HBURSTP[2:0]</b> | Output    | Indicates if the transfer forms part of a burst. Four-beat bursts are supported and the burst can be either incrementing or wrapping.                                                                                                                                                                                                                                                                                                      |
| <b>HMASTLOCKP</b>   | Output    | Peripheral port lock signal.                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>HPROTP[5:0]</b>  | Output    | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. The signals indicate if: <ul style="list-style-type: none"> <li>the transfer is an opcode fetch or data access</li> <li>the transfer is a Supervisor mode access or User mode access</li> <li>the current access is Cachable or Bufferable.</li> </ul> |

Table A-8 Peripheral port signals (continued)

| Name                   | Direction | Description                                                                                                                                                                                                                                                                                             |
|------------------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HRDATAP[31:0]</b>   | Input     | The read data bus is used to transfer data and instructions from bus slaves to the bus master during read operations.                                                                                                                                                                                   |
| <b>HREADYP</b>         | Input     | When HIGH the <b>HREADYP</b> signal indicates that a transfer has finished on the peripheral port data bus. You can drive this signal LOW to extend a transfer.                                                                                                                                         |
| <b>HRESPP</b>          | Input     | The transfer response provides additional information on the status of a transfer. Two responses are provided:<br>0 = Okay<br>1 = Error                                                                                                                                                                 |
| <b>HSIDEBANDP[3:0]</b> | Output    | Signals shareable and inner cachable. On reset <b>HSIDEBANDP[3:0]</b> is set to b0010.                                                                                                                                                                                                                  |
| <b>HSIZEP[2:0]</b>     | Output    | Indicates the size of the peripheral port transfer, which is typically: <ul style="list-style-type: none"> <li>• byte (8-bit)</li> <li>• halfword (16-bit)</li> <li>• word (32-bit)</li> <li>• doubleword (64-bit).</li> </ul> The protocol enables larger transfer sizes up to a maximum of 1024 bits. |
| <b>HTRANSP[1:0]</b>    | Output    | Indicates the type of the current transfer on the peripheral port, which can be:<br>b00 = Idle<br>b10 = Nonsequential<br>b11 = Sequential<br>b01 = Busy is not used.                                                                                                                                    |
| <b>HUNALIGNP</b>       | Output    | When HIGH, indicates that the access is unaligned and that <b>HBSTRBP</b> information is required.                                                                                                                                                                                                      |
| <b>HWDATAP[31:0]</b>   | Output    | The peripheral port write data bus is used to transfer data from the bus master to the bus slave during write operations.                                                                                                                                                                               |
| <b>HWRITEP</b>         | Output    | When HIGH this signal indicates a write transfer on the peripheral port, and when LOW a read transfer.                                                                                                                                                                                                  |

## A.4.5 DMA port signals

The DMA port is a 64-bit wide AHB-lite port that is read/write.

Table A-9 lists the DMA port signals.

**Table A-9 DMA port signals**

| Name                   | Direction | Description                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HADDRD[31:0]</b>    | Output    | The 32-bit system DMA port address bus.                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>HBSTRBD[7:0]</b>    | Output    | Indicates which byte lanes are valid.                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>HBURSTD[2:0]</b>    | Output    | Indicates if the transfer forms part of a burst. Four-beat bursts are supported and the burst can be either incrementing or wrapping.                                                                                                                                                                                                                                                                                                      |
| <b>HMASTLOCKD</b>      | Output    | DMA port lock signal.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>HPROTD[5:0]</b>     | Output    | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. The signals indicate if: <ul style="list-style-type: none"> <li>the transfer is an opcode fetch or data access</li> <li>the transfer is a Supervisor mode access or User mode access</li> <li>the current access is Cachable or Bufferable.</li> </ul> |
| <b>HRDATAD[63:0]</b>   | Input     | The read data bus is used to transfer data and instructions from bus slaves to the bus master during DMA read operations.                                                                                                                                                                                                                                                                                                                  |
| <b>HREADYD</b>         | Input     | When HIGH the <b>HREADYD</b> signal indicates that a transfer has finished on the bus. You can drive this signal LOW to extend a transfer.                                                                                                                                                                                                                                                                                                 |
| <b>HRESPD</b>          | Input     | The transfer response provides additional information on the status of a transfer. Two responses are provided:<br>0 = Okay<br>1 = Error.                                                                                                                                                                                                                                                                                                   |
| <b>HSIDEBANDD[3:0]</b> | Output    | Signals shareable and inner cachable                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>HSIZED[2:0]</b>     | Output    | Indicates the size of the DMA port transfer: <ul style="list-style-type: none"> <li>byte (8-bit)</li> <li>halfword (16-bit)</li> <li>word (32-bit)</li> <li>doubleword (64-bit).</li> </ul> <p>The protocol enables larger transfer sizes up to a maximum of 1024 bits.</p>                                                                                                                                                                |

Table A-9 DMA port signals (continued)

| Name                 | Direction | Description                                                                                                                                     |
|----------------------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HTRANSID[1:0]</b> | Output    | Indicates the type of the current transfer on the DMA port:<br>b00 = Idle<br>b10 = Nonsequential<br>b11 = Sequential<br>b01 = Busy is not used. |
| <b>HUNALIGND</b>     | Output    | When HIGH, indicates that the access is unaligned and that <b>HBSTRBD</b> information is required.                                              |
| <b>HWDATAD[63:0]</b> | Output    | The DMA port write data bus is used to transfer data from the bus master to the bus slave during DMA write operations.                          |
| <b>HWRITED</b>       | Output    | When HIGH this signal indicates a write transfer on the DMA port, and when LOW a read transfer.                                                 |

## A.5 Coprocessor interface signals

The interface signals from the core to the coprocessor are listed in Table A-10.

**Table A-10 Core to coprocessor signals**

| Name                    | Direction | Description                                                                                              |
|-------------------------|-----------|----------------------------------------------------------------------------------------------------------|
| <b>ACPCANCEL</b>        | Output    | Asserted to indicate that the instruction is to be canceled.                                             |
| <b>ACPCANCELT [3:0]</b> | Output    | The tag accompanying the cancel signal in <b>ACPCANCEL</b> .                                             |
| <b>ACPCANCELV</b>       | Output    | Asserted to indicate that <b>ACPCANCEL</b> is valid.                                                     |
| <b>ACPENABLE[11:0]</b>  | Output    | Enables the coprocessor when this is asserted. All lines driven by the coprocessor must be held to zero. |
| <b>ACPFINISHV</b>       | Output    | The finish token from the core WBIs stage to the coprocessor Ex6 stage.                                  |
| <b>ACPFLUSH</b>         | Output    | Flush broadcast from the core.                                                                           |
| <b>ACPFLUSHT[3:0]</b>   | Output    | The tag to be flushed from.                                                                              |
| <b>ACPINSTR [31:0]</b>  | Output    | The instruction passed from the core Fe2 stage to the coprocessor Decode stage.                          |
| <b>ACPINSTRT [3:0]</b>  | Output    | The tag accompanying the instruction in <b>ACPINSTR</b> .                                                |
| <b>ACPINSTRV</b>        | Output    | Asserted to indicate that <b>ACPINSTR</b> carries a valid instruction.                                   |
| <b>ACPLDDATA [63:0]</b> | Output    | The load data from the core to the coprocessor.                                                          |
| <b>ACPLDVALID</b>       | Output    | Asserted to indicate that the data in <b>ACPLDATA</b> is valid.                                          |
| <b>ACPSTSTOP</b>        | Output    | Asserted by the core to tell the coprocessor to stop sending store data.                                 |
| <b>ACPPRIV</b>          | Output    | Asserted to indicate that the core is in Supervisor mode.                                                |

The interface signals from the coprocessor to the core are listed in Table A-11 on page A-15.

If no coprocessor is connected, the following control signals must be driven LOW:

- **CPALENGTHHOLD**
- **CPAACCEPT**
- **CPAACCEPTHOLD**.

**Table A-11 Coprocessor to core signals**

| <b>Name</b>             | <b>Direction</b> | <b>Description</b>                                                                 |
|-------------------------|------------------|------------------------------------------------------------------------------------|
| <b>CPAACCEPT</b>        | Input            | The bounce signal from the coprocessor issue stage to the core Ex2 stage.          |
| <b>CPAACCEPTHOLD</b>    | Input            | Asserted to indicate that the bounce information in <b>CPAACCEPT</b> is not valid. |
| <b>CPAACCEPTT [3:0]</b> | Input            | The tag accompanying the bounce signal in <b>CPAACCEPT</b> .                       |
| <b>CPALENGTH [3:0]</b>  | Input            | The length information from the coprocessor Decode stage to the core Ex1 stage.    |
| <b>CPALENGTHHOLD</b>    | Input            | Asserted to indicate that the length information in <b>CPALENGTH</b> is not valid. |
| <b>CPALENGTHT [3:0]</b> | Input            | The tag accompanying the length signal in <b>CPALENGTH</b> .                       |
| <b>CPAPRESENT[11:0]</b> | Input            | Indicates which coprocessors are present.                                          |
| <b>CPASTDATA [63:0]</b> | Input            | The store data passing from the coprocessor to the core.                           |
| <b>CPASTDATAT [3:0]</b> | Input            | The tag accompanying the store data in <b>CPASTDATA</b> .                          |
| <b>CPASTDATAV</b>       | Input            | Indicates that the store data to the core is valid.                                |

## A.6 Debug interface signals (including JTAG)

Table A-12 lists the debug interface signals including JTAG.

**Table A-12 Debug interface signals**

| <b>Name</b>            | <b>Direction</b> | <b>Description</b>                                                     |
|------------------------|------------------|------------------------------------------------------------------------|
| <b>DBGTCKEN</b>        | Input            | Debug clock enable.                                                    |
| <b>DBGnTRST</b>        | Input            | Debug <b>nTRST</b> .                                                   |
| <b>DBGTDI</b>          | Input            | Debug <b>TDI</b> .                                                     |
| <b>DBGTMS</b>          | Input            | Debug <b>TMS</b> .                                                     |
| <b>EDBGRQ</b>          | Input            | External debug request.                                                |
| <b>DBGEN</b>           | Input            | Debug enable.                                                          |
| <b>DBGVERSION[3:0]</b> | Input            | JTAG ID version field.                                                 |
| <b>DBGMANID[10:0]</b>  | Input            | JTAG ID manufacturer field.                                            |
| <b>DBGTDO</b>          | Output           | Debug <b>TDO</b> .                                                     |
| <b>DBGnTDOEN</b>       | Output           | Debug <b>nTDOEN</b> .                                                  |
| <b>COMMTX</b>          | Output           | Comms channel transmit.<br>On reset this pin is set to 1.              |
| <b>COMMRX</b>          | Output           | Comms channel receive.                                                 |
| <b>DBGACK</b>          | Output           | Debug acknowledge.                                                     |
|                        | output           | Debugger has requested that ARM1136JF-S processor is not powered down. |
| <b>FREEDBGTCKEN</b>    | Input            | Debug clock enable for the <b>FRECLK</b> domain.                       |



## A.7 ETM interface signals

Table A-13 lists the ETM interface signals.

**Table A-13 ETM interface signals**

| Name               | Direction | Description                                                                                                                |
|--------------------|-----------|----------------------------------------------------------------------------------------------------------------------------|
| ETMDA[31:3]        | Output    | ETM data address.                                                                                                          |
| ETMDACTL[17:0]     | Output    | ETM data control (address phase).                                                                                          |
| ETMDD[63:0]        | Output    | ETM data data.                                                                                                             |
| ETMDDCTL[3:0]      | Output    | ETM data control (data phase).                                                                                             |
| ETMEXTOUT[1:0]     | Input     | ETM external event to be monitored.                                                                                        |
| ETMIA[31:0]        | Output    | ETM instruction address.                                                                                                   |
| ETMIACTL[17:0]     | Output    | ETM instruction control.                                                                                                   |
| ETMIARET[31:0]     | Output    | ETM return instruction address.                                                                                            |
| ETMPADV[2:0]       | Output    | ETM pipeline advance.                                                                                                      |
| ETMPWRUP           | Input     | When HIGH, indicates that the ETM is powered up. When LOW, logic supporting the ETM must be clock gated to conserve power. |
| nETMWFIREADY       | Input     | When HIGH, indicates ETM can accept Wait For Interrupt.                                                                    |
| ETMCPADDRESS[14:0] | Output    | Coprocessor CP14 address.                                                                                                  |
| ETMCPCOMMIT        | Output    | Coprocessor CP14 commit.                                                                                                   |
| ETMCPENABLE        | Output    | Coprocessor CP14 interface enable.                                                                                         |
| ETMCPRDATA[31:0]   | Input     | Coprocessor CP14 read data.                                                                                                |
| ETMCPWDATA[31:0]   | Output    | Coprocessor CP14 write data.                                                                                               |
| ETMCPWRITE         | Output    | Coprocessor CP14 write control.                                                                                            |
| EVNTBUS[19:0]      | Output    | System metrics event bus.                                                                                                  |
| WFIPENDING         | Output    | Indicates a Pending Wait For Interrupt. Handshakes with nETMWFIREADY.                                                      |

## A.8 Test signals

Table A-14 lists the test signals.

**Table A-14 Test signals**

| <b>Name</b>            | <b>Direction</b> | <b>Description</b>                                                                                   |
|------------------------|------------------|------------------------------------------------------------------------------------------------------|
| <b>SCANMODE</b>        | Input            | In scan test mode.                                                                                   |
| <b>SE</b>              | Input            | Scan enable.                                                                                         |
| <b>MBISTADDR[12:0]</b> | Input            | <i>Memory Built-In Self Test</i> (MBIST) address.                                                    |
| <b>MBISTCE[22:0]</b>   | Input            | MBIST chip enable.                                                                                   |
| <b>MBISTDIN[63:0]</b>  | Input            | MBIST data in.                                                                                       |
| <b>MBISTDOUT[63:0]</b> | Output           | MBIST data out.                                                                                      |
| <b>MBISTWE</b>         | Input            | MBIST write enable.                                                                                  |
| <b>MTESTON</b>         | Input            | BIST test is enabled.                                                                                |
| <b>nVALFIQ</b>         | Output           | Request for a Fast Interrupt. On reset this pin is set to 1.                                         |
| <b>nVALIRQ</b>         | Output           | Request for an Interrupt. On reset this pin is set to 1.                                             |
| <b>nVALRESET</b>       | Output           | Request for a Reset. On reset this pin is set to 1.                                                  |
| <b>VALEDBGREQ</b>      | Output           | Request for an external debug request.                                                               |
| <b>MUXINSEL</b>        | Input            | These are the test wrapper enable signals. See <i>ARM1136 Implementation Guide</i> for more details. |
| <b>MUXOUTSEL</b>       | Input            |                                                                                                      |

# Glossary

This glossary describes some of the terms used in this manual. Where terms can have several meanings, the meaning presented here is intended.

- Abort** A mechanism that indicates to a core that it must halt execution of an attempted illegal memory access. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a prefetch abort, a Data Abort, or an external abort. *See also* Data Abort, External Abort and Prefetch Abort.
- Abort model** An abort model is the defined behavior of an ARM processor in response to a Data Abort exception. Different abort models behave differently with regard to load and store instructions that specify base register Write-Back.
- Advanced Microcontroller Bus Architecture (AMBA)** The ARM open standard for on-chip buses. AHB conforms to this standard.
- Aligned** Refers to data items stored so that their address is divisible by the highest power of two that divides their size. Aligned words and halfwords therefore have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore refer to addresses that are divisible by four and two respectively. Other related terms are defined similarly.
- ALU** *See* Arithmetic Logic Unit.

|                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>AMBA</b>                                           | <i>See</i> Advanced Microcontroller Bus Architecture.                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Arithmetic Logic Unit (ALU)</b>                    | The part of a processor core that performs arithmetic and logic operations.                                                                                                                                                                                                                                                                                                                                                              |
| <b>Application Specific Integrated Circuit (ASIC)</b> | An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.                                                                                                                                                                                                                                                                                                        |
| <b>ARM state</b>                                      | A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.                                                                                                                                                                                                                                                                                                                                          |
| <b>ASIC</b>                                           | <i>See</i> Application Specific Integrated Circuit.                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Banked registers</b>                               | Those physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.                                                                                                                                                                                                                                                                                                                         |
| <b>Base register</b>                                  | A register specified by a load/store instruction that is used to hold the base value for the instruction's address calculation.                                                                                                                                                                                                                                                                                                          |
| <b>Big-endian</b>                                     | Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory. <i>See also</i> Little-endian and Endianness.                                                                                                                                                                                                                                                                |
| <b>Breakpoint</b>                                     | A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested. <i>See also</i> Watchpoint. |
| <b>Byte</b>                                           | An 8-bit data item.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Byte invariant</b>                                 | Refers to the way of switching between little-endian and big-endian operation that leaves byte accesses entirely unchanged. Accesses to other data sizes are necessarily affected by such endianness switches.                                                                                                                                                                                                                           |
| <b>Cache</b>                                          | A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.                                                                                                                                |
| <b>Cache contention</b>                               | When the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity increases and performance decreases.                                                                                                                                                                                                                                |
| <b>Cache hit</b>                                      | A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.                                                                                                                                                                                                                                                                                                      |
| <b>Cache line index</b>                               | The number associated with each cache line in a cache set. Within each cache set, the cache lines are numbered from 0 to (set associativity) -1.                                                                                                                                                                                                                                                                                         |

|                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cache lockdown</b>                    | To fix a line in cache memory so that it cannot be overwritten. Cache lockdown enables critical instructions and/or data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to the instructions/data concerned are cache hits, and therefore complete as quickly as possible.                                                                                                            |
| <b>Cache miss</b>                        | A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.                                                                                                                                                                                                                                                                                                                            |
| <b>Central Processing Unit (CPU)</b>     | The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.                                                                                                                                                                                                                                                                                                                      |
| <b>Clock gating</b>                      | Gating a clock signal for a macrocell with a control signal (such as <b>PWRDOWN</b> ) and using the modified clock that results to control the operating state of the macrocell.                                                                                                                                                                                                                                                                                                      |
| <b>Communications channel</b>            | The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel. In an ARMv6 compliant core, the communications channel includes the Data Transfer Register, some bits of the Data Status and Control Register, and the external debug interface controller, such as the DBGTAP controller in the case of the JTAG interface. |
| <b>Condition field</b>                   | A 4-bit field in an instruction that is used to specify a condition under which the instruction can execute.                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Coprocessor</b>                       | A processor that supplements the main processor. It carries out additional functions that the main processor cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.                                                                                                                                                                                                                                                              |
| <b>Coprocessor Data Processing (CDP)</b> | For the VFP coprocessor, CDP operations are arithmetic operations rather than load/store operations.                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Copy back</b>                         | <i>See</i> Write-Back.                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Data Abort</b>                        | An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A Data Abort is attempting to access invalid data memory. <i>See also</i> Abort, External Abort, and Prefetch Abort.                                                                                                                                                                                                                                                  |
| <b>Data Cache</b>                        | A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.                                                                                                                                                                                                             |
| <b>DBGTAP</b>                            | <i>See</i> Debug Test Access Port.                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

**DeBuG Test Access Port (DBGTAP)**

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **DBGTDI**, **DBGTDO**, **DBGTMS**, and **TCK**. The optional terminal is **TRST**.

**Debugger**

A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

**Domain**

A collection of sections, large pages and small pages of memory, which can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register c3).

**Doubleword**

A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

**Endianness**

Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. *See also* Little-endian and Big-endian.

**Exception vector**

One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt service routine.

**External Abort**

An indication from an external memory system to a core that it must halt execution of an attempted illegal memory access. An external abort is caused by the external memory system as a result of attempting to access invalid memory. *See also* Abort, Data Abort and Prefetch Abort

**Fast Context Switch Extension (FCSE)**

This enables cached processors with an MMU to present different addresses to the rest of the memory system for different software processes even when those processes are using identical addresses.

**FCSE**

*See* Fast Context Switch Extension.

**Halfword**

A 16-bit data item.

**Halt mode**

One of two mutually exclusive debug modes. In halt mode all processor execution halts when a breakpoint or watchpoint is encountered. All processor state, coprocessor state, memory and input/output locations can be examined and altered by the JTAG interface. *See also* Monitor mode.

**Hit-Under-Miss (HUM)**

A buffer that enables program execution to continue, even though there has been a data miss in the cache.

**HUM**

*See* Hit-Under-Miss.

|                                       |                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Instruction Cache</b>              | A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.                                                                                           |
| <b>Joint Test Action Group (JTAG)</b> | The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.                                                                                                                                                  |
| <b>JTAG</b>                           | <i>See</i> Joint Test Action Group.                                                                                                                                                                                                                                                                                                                                         |
| <b>Little-endian</b>                  | Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory. <i>See also</i> Big-endian and Endianness.                                                                                                                                                                                                      |
| <b>Macrocell</b>                      | A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as an ARM1136JFS, an ETM11RV, and a memory block) plus application-specific logic.                                                                                                                                                                    |
| <b>MCR</b>                            | For the FPS this includes instructions that transfer data or control registers between an ARM register and a FPS register. Only 32 bits of information can be transferred using a single MCR class instruction.                                                                                                                                                             |
| <b>Modified Virtual Address (MVA)</b> | A virtual address produced by the ARM1136JF-S processor can be changed by the current Process ID to provide a <i>Modified Virtual Address</i> (MVA) for the MMUs and caches. <i>See also</i> FCSE.                                                                                                                                                                          |
| <b>Monitor mode</b>                   | One of two mutually exclusive debug modes. In monitor mode the ARM1136JF-S processor enables a software abort handler provided by the debug monitor or operating system debug task. When a breakpoint or watchpoint is encountered, this enables vital system interrupts to continue to be serviced while normal program execution is suspended. <i>See also</i> Halt mode. |
| <b>MRC</b>                            | For the FPS this includes instructions that transfer data or control registers between the FPS and an ARM register. Only 32 bits of information can be transferred using a single MRC class instruction.                                                                                                                                                                    |
| <b>MVA</b>                            | <i>See</i> Modified Virtual Address.                                                                                                                                                                                                                                                                                                                                        |
| <b>PA</b>                             | <i>See</i> Physical Address.                                                                                                                                                                                                                                                                                                                                                |
| <b>Prefetch Abort</b>                 | An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A prefetch abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory. <i>See also</i> Data Abort, External Abort and Abort                                                               |

**Physical Address (PA)**

The MMU performs a translation on *Modified Virtual Addresses* (MVA) to produce the *Physical Address* (PA) which is given to AHB to perform an external access. The PA is also stored in the Data Cache to avoid needing address translation when data is cast out of the cache. *See also* FCSE.

**Processor**

A contraction of microprocessor. A processor includes the processor or core, plus additional components such as memory, and interfaces. These are combined as a single macrocell, that can be fabricated on an integrated circuit.

**Read**

Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP. Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

**RealView ICE**

RealView ICE is a system for debugging embedded processor cores that uses a JTAG interface.

**Region**

A partition of instruction or data memory space.

**Register**

A temporary storage location used to hold binary data until it is ready to be used.

**Remapping**

Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM when the initialization has been done.

**Reserved**

A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and are to be read as 0.

**SBO**

*See* Should Be One.

**SBZ**

*See* Should Be Zero.

**Should Be One (SBO)**

Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 produces Unpredictable results.

**Should Be Zero (SBZ)**

Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.



**Synchronization primitive**

The memory synchronization primitive instructions are those instructions that are used to ensure memory synchronization. That is, the LDREX, STREX, SWP, and SWPB instructions.

**Thumb state**

A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.

**TLB**

See Translation Look-aside Buffer.

**Translation Lookaside Buffer (TLB)**

A cache of recently used page table entries that avoid the overhead of page table walking on every memory access. Part of the Memory Management Unit.

**Undefined**

Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more information on ARM exceptions.

**Unpredictable**

For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

**VA**

See Virtual Address.

**Vector operation**

An operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination.

**Virtual Address (VA)**

The MMU uses its page tables to translate a Virtual Address into a Physical Address. The processor executes code at the Virtual Address, which might be located elsewhere in physical memory. *See also* FCSE, MVA, and PA.

**Watchpoint**

A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to allow inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. *See also* Breakpoint.

**WB**

See Write-Back.

**Word**

A 32-bit data item.

**Write**

Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH. Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

- Write-Back (WB)** In a Write-Back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. (Also known as copyback).
- Write buffer** A block of high-speed memory, arranged as a FIFO buffer, between the Data Cache and main memory, whose purpose is to optimize stores to main memory. Each entry in the write buffer can contain the address of a data item to be stored to main memory, the data for that item, and a sequential bit that indicates if the next store is sequential or not.
- Write completion** The memory system indicates to the processor that a write has been completed at a point in the transaction where the memory system is able to guarantee that the effect of the write is visible to all processors in the system. This is not the case if the write is associated with a memory synchronization primitive, or is to a Device or Strongly Ordered region. In these cases the memory system might only indicate completion of the write when the access has affected the state of the target, unless it is impossible to distinguish between having the effect of the write visible and having the state of target updated.
- This stricter requirement for some types of memory ensures that any side-effects of the memory access can be guaranteed by the processor to have taken place. You can use this to prevent the starting of a subsequent operation in the program order until the side-effects are visible.
- Write-Through (WT)** In a Write-Through cache, data is written to main memory at the same time as the cache is updated.
- WT** *See* Write-Through.

# Index

The items in this index are listed in alphabetical order. The references given are to page numbers.

## A

- A bit 2-20, 3-99
- Abbreviations 3-4
- Abort 2-35
  - mode 2-9
  - Prefetch 2-36
  - summary 6-34
- Access permission 3-46, 6-12
- Accessing CP15 registers 3-3
- Address mapping 3-101
- Addressing mode 2 1-46
- Addressing mode 5 1-49
- AHB 8-9
- AHB-Lite 8-69
  - advantages 8-71
  - block diagram 8-72
  - compatibility with full AHB 8-70
  - conversion to full AHB 8-71
  - master interface 8-71
  - slaves 8-71
  - specification 8-70
- ALU pipeline stage 1-26

- AMBA 12-2
- AP bits 3-46, 3-48, 6-12
- AP field encoding 3-47
- APX bit 6-12
- ARM
  - exception, entering 2-26
  - exception, leaving 2-26
  - instruction set 1-34
  - instruction set summary 1-38
- ARM state 1-34, 2-3
  - register set 2-10
  - to Java state 2-3
  - to Thumb state 2-3
- ARM11 core 1-5
- ARM1136JF-S
  - architecture 1-34
  - clocking 9-2
  - cycle timing behavior 16-2
  - instruction set 1-36
  - interlock behavior 16-2
  - pipeline 1-26
- Asynchronous clocking 9-3
- Auxiliary Control Register 3-93

## B

- B bit 3-96, 3-99, 6-14
- Banked registers 2-10
- Big-endian format 2-6
- BIGENDINIT 3-8, 3-96
- BKPT 2-39
- Block Address Register format 3-27
- Block transfer operations 3-25
- Block Transfer Status Register format 3-28
- Branch folding 5-6
- Branch prediction 5-4
  - CP15 r1 5-6
  - IMB instruction 5-9
  - incorrect prediction 5-7
  - static 5-6
  - Z bit 5-6
- Breakpoint instruction 2-39
- Bubble closing 11-10
- Bypass 14-7
- Byte 2-5

- Byte lane strobe 8-15
  - example use 8-16
- Bytecode, Java 1-34
  
- C**
- C bit 3-89, 3-99, 6-14
- C flag 2-16
- Cachable
  - fetches 8-20
  - Write-Back 6-19
  - Write-Through 6-19
- Cache
  - associativity encoding 3-31
  - block diagram 7-4
  - cleaning and invalidating,
    - SmartCache 3-23
  - debug 7-17
  - Debug Control Register 3-34
  - Dirty Status Register format 3-25
  - disabled behavior 7-7
  - functional description 7-5
  - miss handling 7-6
  - Operations Register 3-17
  - organization 7-3
  - size encoding 3-30
  - system features 7-5
  - Type Register 3-28
- CCNT 3-92
- Change Processor State 2-24
- Changes in instruction flow 16-2
  - dynamic branch predictor 16-2
  - return stack 16-2
  - static branch predictor 16-2
- Clamp 14-7
- ClampZ 14-7
- Clean
  - Data Cache Line
    - (using index) 3-20, 3-22
  - Data Cache Line (using MVA) 3-20
  - Data Cache Range 3-26
  - Entire Data Cache 3-20, 3-24, 3-93
  - Range 3-25
- Clean and Invalidate
  - Data Cache Line
    - (using MVA) 3-20
  - Data Cache Line (using index) 3-20
  - Data Cache Range 3-27
- Clean and Invalidate (Continued)
  - Entire Data Cache 3-20, 3-24
  - Range 3-25
- Clear 3-60
- Client 6-11
- Clocking
  - ARM1136JF-S 9-2
  - asynchronous 9-3
  - synchronous 9-2
- Code density 1-34
- Cold reset 9-8
- Complete or Error 3-55
- Compression, instruction 1-34
- Condition code flags 1-50, 2-16
- Conditional execution 1-5
- Conditional instructions 16-4
  - CP14 instructions 16-5
  - CP15 instructions 16-5
  - MSR 16-5
  - Multiplies 16-5
- Configurable options 1-25
- Context ID Register 3-95
- Control bits 2-20
- Control Register 3-96
- Coprocessor Access Control Register
  - 3-94
- Count
  - Register Reset on Write 3-89
  - Register 0 3-91
  - Register 1 3-91
- CPS 2-24
- CPSR 2-10, 2-13, 2-16
  - E bit 4-27
- CPU reset 9-9
- CP15 registers arranged
  - by function 3-9
  - numerically 3-12
- Current Program Status Register 2-10, 2-13, 2-16
- Cycle timing behavior
  - Data processing instructions 16-8
  - data processing instructions 16-8
- Cycle count divider 3-88
- Cycle Counter Register 3-92
  - Reset on Write 3-89
- Cycle counts
  - if destination is not PC 16-8
  - if destination is the PC 16-8
- Cycle timing behavior 16-11
  - ARMv6 Media data-processing 16-12
  - Multiplies 16-15
  - register interlock examples 16-7
  - saturating arithmetic 16-11
- Cycle timings
  - complex instruction dependencies 16-2
  
- D**
- D bit 3-88
- DABLSel 15-5
- DACPRT 15-5
- DALast 15-5
- DANSeq 15-5
- DARot 15-5
- DASlot 15-5
- DASwizzle 15-5
- Data
  - alignment 4-3
  - Debug Cache Register 3-35
  - Memory Remap Register 3-69
  - SmartCache Master Valid Register 3-37
  - TCM Region Register 3-83
  - Transfer Register (DTR) 14-15
  - types 2-5
- Data Cache 7-2
  - Lockdown Register 3-15
  - Master Valid Register 3-37
- Data Memory Barrier 3-20, 6-24, 6-25
- Data MicroTLB
  - Attribute Register 3-47
  - PA Register 3-45
  - VA Register 3-44
- Data processing instructions
  - AND 16-8
- Data Read Interface 8-2, 8-5
- Data Write Interface 8-2, 8-5
  - AHB transfers 8-49
- DAUnaligned 15-5
- DAWrite 15-5
- DB bit 3-94

- DBGTAP
  - reset 9-9
  - state machine 14-2
- DDFail 15-6
- DDSLOT 15-6
- De pipeline stage 1-26
- Debug
  - abort 6-27
  - event 6-31
  - scan chains 14-8
  - system 13-2
- Default memory region 3-72
- Definition of cycle timing terms 16-6
- Device memory 6-17, 6-19
- Direction of transfer 3-57
- DMA
  - and core access arbitration 7-13
  - Channel Number Register 3-53
  - Channel Status Register 3-53
  - Context ID Register 3-55
  - Control Register 3-56
  - Enable Register 3-59
  - External Start Address Register 3-61
  - Identification and Status Register 3-61
  - interface 8-2, 8-6
  - interface, AHB transfers 8-64
  - Internal End Address Register 3-63
  - Internal Start Address Register 3-63
  - Memory Remap Register 3-69
  - registers 3-51
  - User Accessibility Register 3-64
- Domain 6-11
  - Access Control Register 3-67
  - access permission 3-67
  - bits 3-49
  - fault 6-31
- Domains
  - clients 6-11
  - managers 6-11
  - memory access control 6-11
- Dormant mode 10-4
- Drain Write Buffer 3-20, 6-24, 6-25
- DSP instructions 1-6
- DT bit 3-57, 3-98
- Dynamic branch
  - prediction enable 3-94
  - predictor 5-5
- Dynamic branch predictor 16-2
  - prediction scheme 16-2
- E**
  - E bit 3-89, 4-27
  - En bit 3-84, 3-86
  - Enable 3-89
  - Enable Export 3-88
  - Enabling/disabling
    - Instruction Cache 3-98
    - MMU protection 3-99
  - End Address 3-27
  - Endianness 2-6
  - Error response 8-14
  - ES bits 3-54
  - ETM
    - coprocessor interface 15-7
    - core connections 15-9
    - data address interface 15-4
    - data value interface 15-5
    - interface 15-2
    - pipeline advance interface 15-6
  - ETMCPAddress 15-8
  - ETMCPCommit 15-7
  - ETMCPEnable 15-7
  - ETMCPRData 15-8
  - ETMCPWData 15-8
  - ETMCPWrite 15-7
  - ETMDA 15-4
  - ETMDACTL 15-4
  - ETMDDD 15-6
  - ETMDDDCTL 15-6
  - ETMEXTOUT 15-9
  - ETMPADV 15-7
  - ETMPWRDOWN 15-9
  - Events, performance monitoring 3-89
  - EVNTBUS 3-88, 15-9
  - EvtCount1 3-88
  - EvtCount2 3-88
  - Example interlocks 16-9
- Exception 2-23
  - entry and exit 2-25
  - entry, ARM and Java states 2-26
  - entry, Thumb state 2-26
  - FIQ 2-27
  - IRQ 2-28
  - priority 2-40
  - vector location 3-98
  - vectors 2-39
- Exclusive access
  - instructions 8-7
  - protocol 8-14
  - timing 8-17
- Execute never 3-49, 6-13
- Explicit Memory Barrier 6-24
- Extended page table configuration 3-98
- Extended region type 3-46
- Extended small page translation
  - ARMv6 6-53
  - backwards-compatible 6-54
- Extended small subpage translation
  - backwards-compatible 6-54
- External abort 6-27, 6-28
  - on data read/write 6-28
  - on hardware page table walk 6-28
  - on instruction fetch 6-28
- External Address Error Status bits 3-54
- Extest 14-6
- F**
  - F bit, FIQ disable 2-20
  - Fast Context Switch Extension 3-100
  - Fast interrupt configuration 3-98
  - Fault
    - Address Register 3-65
    - checking sequence 6-29
    - Status Register 6-33
  - FCSE 3-100
    - PID Register 3-95
  - Fe1 pipeline stage 1-26
  - Fe2 pipeline stage 1-26
  - FI bit 2-28, 3-98
  - Fields 1-50

- FIQ
    - disable, F bit 2-20
    - exception 2-27
    - handler 2-32
    - mode 2-9
  - First-level descriptor 6-36, 6-39, 6-40, 6-45
    - address 6-43
  - First-level translation fault 6-45
  - Flag bit 3-88
  - Flags 2-16
  - Flush
    - Branch Target Cache Entry 3-19, 3-23
    - Entire Branch Target Cache 3-19
    - Prefetch Buffer 3-19, 6-24, 6-25
  - FT bit 3-58
  - Full Transfer 3-58
  - Full-line Write-Back 8-62
- G**
- Global
    - Data TCM enable 3-98
    - Instruction TCM enable 3-98
- H**
- HADDR 8-15
  - Half-line Write-Back 8-61
  - Halfword 2-5
  - Halt 14-6
  - Hardware page table translation 6-35
  - HBSTRB 8-15
  - High registers 2-14
  - HighZ 14-7
  - HPROT 8-11, 8-13
  - HRESP 8-13
  - HSIZE 8-10, 8-15
  - HUNALIGN 8-15
- I**
- I bit 2-20, 3-98
  - IABpCCFail 15-3
  - IABpValid 15-4
  - IAContextID 15-3
  - IADAbort 15-3
  - IADDataInst 15-3
  - IAExCancel 15-3
  - IAException 15-3
  - IAExInt 15-3
  - IAIndBr 15-3
  - IAInstCCFail 15-3
  - IAInstValid 15-4
  - IAJBit 15-3
  - IATBit 15-3
  - IAValid 15-4
  - IC bit 3-57
  - ID Code Register 3-102
  - IDcode 14-7
  - Idle 3-55
  - IE bit 3-58
  - IMB instruction 5-9
  - Imprecise Data Abort 2-37
  - Imprecise data abort mask 2-37
  - Incorrect prediction 5-7
  - Inner 6-15
  - Inner region remap encoding 3-71
  - Instruction
    - compression 1-34
    - Debug Cache Register 3-35
    - Fault Address Register 3-67
    - length 2-4
    - Memory Remap Register 3-69
    - SmartCache Master Valid Register 3-37
    - TCM Region Register 3-85
  - Instruction Cache 7-2
    - enabling/disabling 3-98
    - Lockdown Register 3-15
    - Master Valid Register 3-37
  - Instruction execution overview 16-3
    - ALU and multiply pipeline operation 16-4
  - Instruction Fetch Interface 8-2, 8-4
    - AHB transfers 8-20
  - Instruction Memory Barrier 3-95
    - See IMB instruction
  - Instruction MicroTLB
    - Attribute Register 3-47
    - PA Register 3-45
    - VA Register 3-44
  - Instruction set
    - ARM 1-34, 1-38
    - summary 1-36
    - Thumb 1-6, 1-34
  - Instruction sets 1-5
    - branch 1-5
    - coprocessor 1-5
    - data processing 1-5
    - load and store processing 1-5
  - Instruction Transfer Register (ITR)
    - scan chain 4 14-14
  - IntEn bits 3-88
  - Interlock behavior 16-2
  - Internal Address Error Status bits 3-55
  - Interrupt
    - Enable 3-88
    - entry flowchart 12-9
    - handler exit 12-5
    - latency 2-28
    - latency, example 2-29
    - on Completion 3-57
    - on Error 3-58
  - Interrupting 3-62
  - Interworking 2-3
  - Invalidate
    - Both Caches 3-19
    - Data Cache Line
      - (using index) 3-19
      - (using MVA) 3-19
    - Data Cache Range 3-26
    - Entire Data Cache 3-19
    - Entire Instruction Cache 3-19
    - Instruction Cache Line
      - (using index) 3-19
      - (using MVA) 3-19
    - Instruction Cache Range 3-26
    - Range 3-25
    - TLB 3-75, 3-77
    - TLB Single Entry 3-75, 3-77
    - TLB Single Entry on ASID Match 3-75, 3-77
  - IRQ
    - disable, I bit 2-20
    - exception 2-28
    - mode 2-9
  - IS bits 3-55
  - Iss pipeline stage 1-26
  - IT bit 3-98

**J**

- J bit 2-18
- Java bytecode 1-34
- Java state 1-34
  - byte-aligned instructions 2-3
  - to ARM state 2-3
- JTAG diagram 14-2
- JTAG public instructions
  - Bypass 14-7
  - Clamp 14-7
  - ClampZ 14-7
  - Extest 14-6
  - HighZ 14-7
  - IDcode 14-7
  - Sample/Preload 14-7
  - Scan\_N 14-6

**L**

- L bit 3-16
- Large page 6-7
- Large page table walk
  - Armv6 6-50
  - backwards-compatible 6-51
- LDM 8-27
- LDR 8-27
- LDRB 8-26
- LDREX 8-7
  - example 8-8
- LDRH 8-26
- Level one cache block diagram 7-4
- Level two
  - data-side controller 8-4
  - instruction-side controller 8-3
  - interface 8-2
  - interface clocking 8-3
- Line length encoding 3-32
- Linefill 8-24
- Link Register 2-10, 2-13
- Little-endian 4-23
  - format 2-6
- Load
  - coprocessor 4-16
  - double 4-16
  - multiple 4-16
  - signed byte 4-7
  - signed halfword 4-10

- unsigned byte 4-7
  - unsigned halfword 4-8, 4-9
  - word 4-12, 4-13
- Load-exclusive 8-7
- Loads to PC set T bit 3-98
- Local RAM 7-9
- Location of exception vectors 3-98
- Low interrupt latency 2-28
- Low registers 2-14
- L4 bit 3-98

**M**

- M bit 3-100
  - MMU 6-55
- Main TLB 6-5
  - Attribute Register 3-47
  - debug 3-40
  - debug operations 3-39
  - Master Valid Register 3-37
  - VA Register 3-44
- Manager 6-11
- MCR, accessing CP15 3-3
- Media instructions 1-6
- Memory
  - access sequence 6-7
  - access, program order 6-23
  - attributes 6-17
  - formats 2-6
  - ordering requirements 6-22
  - ordering restrictions 6-23
  - region attributes 6-14
  - Region Remap Register fields 3-70
  - Region Remap Registers 3-69
  - region, default 3-72
  - space identifier format 3-45
  - synchronization primitives 6-25
  - types 6-17
- Memory access
  - control
    - domains 6-11
- Memory access control 6-11
  - domains 6-11
- Memory management
  - unit
    - memory access permission
      - control 6-3
      - memory region attributes 6-3

- Memory Management Unit 6-2
- Memory management unit 6-2
  - translation lookaside buffer 6-2
- MicroTLB 6-4
  - debug 3-39
  - debug operations 3-39
  - Index format 3-40
  - loading and matching 3-41
- Mixed-endian
  - data access 4-23
  - support 4-22
- MMU
  - abort 6-27
  - debug 3-38
  - descriptors 6-43
  - disabling 6-9, 6-55
  - enable 3-100
  - enabling 6-9, 6-55
  - fault 6-27
  - fault checking 6-29
  - microTLB debug 3-39
  - protection 3-99
  - software-accessible registers 6-55
- Mode
  - abort 2-9
  - bits 2-21
  - FIQ 2-9
  - identifier 2-11
  - IRQ 2-9
  - privileged 2-9
  - PSR bit values 2-21
  - supervisor 2-9
  - System 2-9
  - Undefined 2-9
  - User 2-9
- Modes and exceptions 1-6
- Modified Virtual Address format 3-23
- MRC and MCR bit pattern 3-3
- MRC, accessing CP15 3-3

**N**

- N flag 2-16
- nDBGTRST 9-7
- Noncachable 6-19
  - fetches 8-21
  - LDM 8-27
  - LDR 8-27

## Noncacheable (Continued)

- LDRB 8-26
- LDRH 8-26
- Non-Shared Normal memory 6-19
- Normal memory 6-17, 6-18
- nPORESET 9-7
- nRESET 9-7

## O

- Okay response 8-14
- Operand2 1-49
- Operating state 2-3
  - ARM 2-3
  - Java 2-3
  - T bit 2-20
  - Thumb 2-3
- Opposite condition code checks 16-5
- Ordered 6-23
- Outer 6-15
- Outer region remap encoding 3-71

## P

- P bit 3-78, 3-89
- PAAdd 15-7
- PAEx2 15-7
- PAEx3 15-7
- Page table
  - attributes, restriction 7-10
  - format 6-14
  - format, ARMv6 6-39
  - mappings, restrictions 7-10
  - translation 6-35
  - translation, ARMv6 6-38
  - translation, backwards-compatible 6-36, 6-38
  - walk 8-47
- Page translation 6-38, 6-41
- Page-based attributes 6-6
- PC (Program Counter) 2-13
- Performance Monitor Control Register 3-87
- Performance monitoring events 3-89
- Peripheral Interface 8-2, 8-5
  - AHB transfers 8-66

## Peripheral Port Memory Remap Register 3-72, 8-5

- Permission fault 6-31
- Physical page number 3-46
- Pipeline operations 1-28
  - ALU 1-29
  - LDM/STM 1-32
  - LDR that misses 1-33
  - LDR/STR 1-31
  - multiply 1-30
- Pipeline stages 1-26
- PMN0 3-91
- PMN1 3-91
- Power management 10-3
  - controller, communication to 10-6
- Power-on reset 9-8
- PPN bits 3-46
- Predictions, incorrect 5-7
- Prefetch
  - Abort 2-36
  - Data Cache Range 3-27
  - Instruction Cache Line 3-20, 3-23
  - Instruction Cache Range 3-27
  - Range 3-25
- Present 3-62
- Preserve bit 3-78
- Priority of exceptions 2-40
- Privileged modes 2-9
- Process 3-44
- Process Identifier Registers 3-95
- ProcID 3-101
- Program Counter 2-10, 2-13
  - not incremented in debug 14-4
- Program flow prediction 5-2
  - enable 3-99, 5-5
- Program status registers 2-16
- PSR
  - control bits 2-20
  - mode bit values 2-21
  - reserved bits 2-22

## Q

- Q flag 2-17
- Queued 3-55, 3-62

## R

- R bit 3-99, 6-12
- Read
  - Cache Debug Control Register 3-34
  - Data Debug Cache Register 3-34
  - Data MicroTLB Attribute Register 3-39
  - Data MicroTLB Entry Operation 3-39
  - Data MicroTLB PA Register 3-39
  - Data MicroTLB VA Register 3-39
  - Data Tag Register 3-34
  - Instruction Cache Data RAM Register 3-34
  - Instruction Debug Cache Register 3-34
  - Instruction MicroTLB Attribute Register 3-39
  - Instruction MicroTLB Entry Operation 3-39
  - Instruction MicroTLB PA Register 3-39
  - Instruction MicroTLB VA Register 3-39
  - Instruction Tag RAM Read Operation 3-34
  - Main TLB Attribute Register 3-39
  - Main TLB Entry Register 3-39
  - Main TLB PA Register 3-39
  - Main TLB VA Register 3-39
  - TLB Debug Control Register 3-39
- Region size 3-46
- Region type bits 3-49
- Register
  - Auxiliary Control 3-93
  - banked 2-10
  - Cache Debug Control 3-34
  - Cache Operations 3-17
  - Cache Type 3-28
  - Context ID 3-95
  - Control 3-96
  - Coprocessor Access Control 3-94
  - Count 0 3-91
  - Count 1 3-91
  - Current Program Status 2-10
  - Cycle Counter 3-92
  - Data Cache Lockdown 3-15
  - Data Cache Master Valid 3-37



- Register (Continued)
  - Data Debug Cache 3-35
  - Data Memory Remap 3-69
  - Data Micro TLB PA 3-45
  - Data MicroTLB 3-44
  - Data MicroTLB Attribute 3-47
  - Data SmartCache Master Valid 3-37
  - Data TCM Region 3-83
  - DMA Channel Number 3-53
  - DMA Channel Status 3-53
  - DMA Context ID 3-55
  - DMA Control 3-56
  - DMA Enable 3-59
  - DMA External Start Address 3-61
  - DMA Identification and Status 3-61
  - DMA Internal End Address 3-63
  - DMA Internal Start Address 3-63
  - DMA Memory Remap 3-69
  - DMA User Accessibility 3-64
  - Domain Access Control 3-67
  - Fault Address 3-65
  - Fault Status 6-33
  - FCSE PID 3-95
    - general-purpose 2-10
    - high 2-14
  - ID Code 3-102
  - Instruction Cache Lockdown 3-15
  - Instruction Cache Master Valid 3-37
  - Instruction Debug Cache 3-35
  - Instruction Fault Address 3-67
  - Instruction Memory Remap 3-69
  - Instruction MicroTLB 3-44
  - Instruction MicroTLB Attribute 3-47
  - Instruction MicroTLB PA 3-45
  - Instruction TCM Region 3-85
  - InstructionSmart Cache Master Valid 3-37
  - Link 2-10
  - Main TLB Attribute 3-47
  - Main TLB Master Valid 3-37
  - Main TLB VA 3-44
  - Performance Monitor Control 3-87
  - Peripheral Port Memory Remap 3-72, 8-5
    - program status 2-16
  - Saved Program Status 2-10
  - status 2-10
  - TCM Status 3-83
  - TLB Debug Control 3-42
  - TLB Lockdown 3-77
  - TLB Operations 3-75
  - Translation Table Base 3-80, 3-81
  - Translation Table Base Control 3-79
  - Register controlled shifts 16-10
  - Register interlock examples 16-7, 16-14
  - Register interlocks 16-7
    - ADR instructions 16-7
    - LDR instructions 16-7
  - Register organization
    - in ARM state 2-12
    - in Thumb state 2-14
  - Registers 1-5
  - Registers, software-accessible by MMU 6-55
  - Replacement algorithms
    - RR bit 3-98
  - Reserved bits, PSR 2-22
  - Reset 2-27
    - CPU 9-9
    - DBGTAP 9-9
    - modes 9-8
    - power-on 9-8
    - warm 9-9
  - Retry response 8-14
  - Return From Exception 2-24
  - Return stack 5-8, 16-3
    - enable 3-94
  - Reverse bytes 4-26
  - RFE 2-24
  - RGN bits 3-49
  - ROM protection 3-99
  - RR bit 3-98
  - RS bit 3-94
  - Run mode 10-3
  - Running 3-55, 3-62
- S**
  - S bit 3-49, 3-99, 6-12, 6-55
  - Sample/Preload
    - JTAG public instructions 14-7
  - Sat pipeline stage 1-26
  - Saved Program Status Register 2-10
  - SB bit 3-94
  - SC bit 3-84, 3-86
  - Scan chain 4
    - Instruction Transfer Register (ITR) 14-14
  - Scan chains
    - debug 14-8
    - scan chain 4 14-14
  - Scan\_N 14-6
  - Second-level
    - descriptor 6-36, 6-40
    - page table address 6-48
    - page table walk 6-47
    - small page table walk 6-51
    - translation fault 6-49
  - Section 6-7, 6-38, 6-41
  - Section translation
    - ARMv6 6-46
    - backwards-compatible 6-47
  - Set/Index format 3-21
  - Sh pipeline stage 1-26
  - Shareable 6-16
  - Shared attribute bit 3-49
  - Shared memory 6-20
  - Shared Normal memory 6-18
  - Shifter 16-9
  - Should Be One 3-4
  - Should Be Zero 3-4
  - Shutdown mode 10-4
  - Size field 3-84, 3-86
  - Small page 6-8
  - Small page translation
    - backwards-compatible 6-52
  - Small subpage translation
    - backwards-compatible 6-52
  - SmartCache 3-23, 3-84, 3-86, 7-2
    - behavior 7-9
  - Software interrupt 2-38
  - Software-accessible registers 6-55
  - SP 2-13
  - Speculative prefetching 5-9
  - Split response 8-14
  - SPSR 2-10, 2-14, 2-16
  - SPV bit 3-48
  - SRS 2-24
  - ST bit 3-58
  - Stack Pointer 2-13
  - Standby mode 10-3

- Start 3-59
  - Start Address 3-27
  - State
    - ARM 1-34
    - switching 2-3
    - Thumb 1-34
  - Static branch prediction enable 3-94
  - Static branch predictor 5-6, 16-3
  - Status
    - bits 3-55
    - registers 2-10
  - Sticky Overflow flag 2-17
  - STM 8-49
  - Stop 3-60
    - Prefetch Range 3-28
  - Store 8-49
    - byte 4-8
    - coprocessor 4-17
    - double 4-17
    - halfword 4-11, 4-12
    - multiple 4-17
    - Return State 2-24
    - word 4-14, 4-15
  - Stored Program Status Register 2-14, 2-16
  - Store-exclusive 8-7, 8-8
  - STR 8-49
  - STREX 8-7
    - example 8-8
  - Strict data address alignment enable 3-99
  - Stride 3-58
  - Strongly Ordered memory 6-17, 6-21
  - Subpage
    - access permission 3-48
    - valid bit 3-48
  - Summary of aborts 6-34
  - Supersection 6-6, 6-7, 6-38, 6-41
  - Supervisor mode 2-9
  - SWI 2-38
    - handler 5-10
  - Switching states 2-3
  - SWP 8-47
  - Synchronization 9-3
  - Synchronization primitives 6-25, 8-7
  - Synchronous clocking 9-2
  - System
    - mode 2-9
    - protection 3-99
  - System metrics 3-87
  - SZ bits 3-46
- ## T
- T bit 2-20
  - TCM 7-2, 7-8
    - and cache interactions 7-13
    - data accesses to 7-14
    - instruction accesses to 7-13
    - Status Register 3-83
  - TEX 6-14
  - Thumb
    - instruction set 1-6, 1-34
    - register set 2-13
    - state 1-34, 2-3, 2-13
    - to ARM state 2-3
  - Tightly-coupled memory
    - See* TCM
  - TLB Attribute Registers 3-47
  - TLB control operations 6-5
  - TLB debug control operations 3-39
  - TLB Debug Control Register 3-42
  - TLB loading and matching 3-41
  - TLB Lockdown Register 3-77
  - TLB match 6-7
  - TLB Operations Register 3-75
    - ASID format 3-76
    - Virtual Address format 3-76
  - TLB organization 6-4
  - TLB PA Register 3-45
  - TLB VA Registers 3-44
  - TR bit 3-57
  - Transaction Size 3-58
  - Translation fault 6-31, 6-45
  - Translation lookaside buffer 2-8
  - Translation Table Base Control Register 3-79
  - Translation Table Base Register 0 3-80
  - Translation Table Base Register 1 3-81
  - TS bit 3-58
  - Two-bit saturating counter 16-2
- ## U
- U bit 3-64, 3-98
  - UM bit 3-58
- Unaligned access
    - model 4-4
  - Unaligned data access
    - enable 3-98
    - restrictions 4-5
    - specification 4-7
    - support in ARMv6 4-5
  - Undefined 3-4
    - instruction 2-38
    - mode 2-9
  - Unexpected hit behavior 7-7
  - Unpredictable 3-4
  - User mode 2-9, 3-58
- ## V
- V bit 3-46, 3-96, 3-98
  - V flag 2-16
  - Valid bit 3-37, 3-46
  - VE bit 3-97
  - Vectored interrupt 3-97
  - Vectored Interrupt Controller 12-2
  - Vector, exception 2-39
  - VIC interface 12-3
  - VIC port 12-3
  - VIC port synchronization 12-4
  - VIC port timing 12-6
  - Victim field 3-78
  - VINITHI 3-8, 3-96
  - Virtual page number 3-44
  - Virtual to physical translation mapping
    - restrictions 6-8
  - VPN 3-44
- ## W
- W bit 3-99
  - Wait For Interrupt 3-19
  - Warm reset 9-9
  - WBex pipeline stage 1-26
  - WCache enable 3-99
  - Weakly Ordered 6-23
  - Word 2-5
  - Write
    - Cache Debug Control Register 3-34
    - Main TLB Attribute Register 3-39
    - Main TLB Entry Register 3-39

## Write (Continued)

- Main TLB PA Register 3-39

- Main TLB VA Register 3-39

- TLB Debug Control Register 3-39

Write allocation policy 6-16

Write Buffer 6-59, 7-18

- enable 3-99

Write-Back 8-61

## X

X bit 3-88

Xfail response 8-14

XN bit 3-49, 6-13

XP bit 3-98

XRGN bits 3-46

XRGN field encoding 3-49

## Z

Z bit 3-99

- branch prediction 5-6

- CP15 register 1 5-6

Z flag 2-16

