

# Cortex™-A8

Revision: r3p1

## Technical Reference Manual



# Cortex-A8

## Technical Reference Manual

Copyright © 2006-2009 ARM Limited. All rights reserved.

### Release Information

The following changes have been made to this book.

<b>Change history</b>			
<b>Date</b>	<b>Issue</b>	<b>Confidentiality</b>	<b>Change</b>
18 July 2006	A	Confidential	First release for r1p0
13 December 2006	B	Non-Confidential	First release for r1p1
13 July 2007	C	Non-Confidential	First release for r2p0
16 November 2007	D	Non-Confidential	First release for r2p1
14 March 2008	E	Non-Confidential	First release for r2p2
06 October 2008	F	Non-Confidential	First release for r2p3
24 October 2008	G	Non-Confidential Restricted Access	First release for r3p0
03 December 2008	H	Non-Confidential Unrestricted Access	Second release for r3p0
30 January 2009	I	Non-Confidential Unrestricted Access	First release for r3p1

### Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited, 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.

Some material in this document is based on *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic* and on *IEEE Std. 1500-2005, IEEE Standard Testability Method for Embedded Core-based Integrated Circuits*. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

**Confidentiality Status**

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

**Product Status**

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

**Web Address**

<http://www.arm.com>



# Contents

## Cortex-A8 Technical Reference Manual

### Preface

About this manual .....	xxviii
Feedback .....	xxxiv

### Chapter 1

#### Introduction

1.1	About the processor .....	1-2
1.2	ARMv7-A architecture .....	1-3
1.3	Components of the processor .....	1-4
1.4	External interfaces of the processor .....	1-8
1.5	Debug .....	1-9
1.6	Power management .....	1-10
1.7	Configurable options .....	1-11
1.8	Product documentation and architecture .....	1-12
1.9	Product revisions .....	1-14

### Chapter 2

#### Programmers Model

2.1	About the programmers model .....	2-3
2.2	Thumb-2 instruction set .....	2-4
2.3	ThumbEE instruction set .....	2-6
2.4	Jazelle Extension .....	2-10
2.5	Security Extensions architecture .....	2-13
2.6	Advanced SIMD architecture .....	2-15

2.7	VFPv3 architecture .....	2-16
2.8	Processor operating states .....	2-17
2.9	Data types .....	2-18
2.10	Memory formats .....	2-19
2.11	Addresses in a processor system .....	2-21
2.12	Operating modes .....	2-22
2.13	Registers .....	2-23
2.14	The program status registers .....	2-27
2.15	Exceptions .....	2-35
2.16	Software consideration for Security Extensions .....	2-44
2.17	Hardware consideration for Security Extensions .....	2-45
2.18	Control coprocessor .....	2-48
<b>Chapter 3</b>	<b>System Control Coprocessor</b>	
3.1	About the system control coprocessor .....	3-2
3.2	System control coprocessor registers .....	3-9
<b>Chapter 4</b>	<b>Unaligned Data and Mixed-endian Data Support</b>	
4.1	About unaligned and mixed-endian data .....	4-2
4.2	Unaligned data access support .....	4-3
4.3	Mixed-endian access support .....	4-5
<b>Chapter 5</b>	<b>Program Flow Prediction</b>	
5.1	About program flow prediction .....	5-2
5.2	Predicted instructions .....	5-3
5.3	Nonpredicted instructions .....	5-6
5.4	Guidelines for optimal performance .....	5-7
5.5	Enabling program flow prediction .....	5-8
5.6	Operating system and predictor context .....	5-9
<b>Chapter 6</b>	<b>Memory Management Unit</b>	
6.1	About the MMU .....	6-2
6.2	Memory access sequence .....	6-3
6.3	16MB supersection support .....	6-4
6.4	MMU interaction with memory system .....	6-5
6.5	External aborts .....	6-6
6.6	TLB lockdown .....	6-7
6.7	MMU software-accessible registers .....	6-8
<b>Chapter 7</b>	<b>Level 1 Memory System</b>	
7.1	About the L1 memory system .....	7-2
7.2	Cache organization .....	7-3
7.3	Memory attributes .....	7-6
7.4	Cache debug .....	7-9
7.5	Data cache features .....	7-10
7.6	Instruction cache features .....	7-11

7.7	Hardware support for virtual aliasing conditions .....	7-13
7.8	Parity detection .....	7-14
<b>Chapter 8</b>	<b>Level 2 Memory System</b>	
8.1	About the L2 memory system .....	8-2
8.2	Cache organization .....	8-3
8.3	Enabling and disabling the L2 cache controller .....	8-5
8.4	L2 PLE .....	8-6
8.5	Synchronization primitives .....	8-11
8.6	Locked access .....	8-13
8.7	Parity and error correction code .....	8-14
<b>Chapter 9</b>	<b>External Memory Interface</b>	
9.1	About the external memory interface .....	9-2
9.2	AXI control signals in the processor .....	9-4
9.3	AXI instruction transactions .....	9-7
9.4	AXI data read/write transactions .....	9-8
<b>Chapter 10</b>	<b>Clock, Reset, and Power Control</b>	
10.1	Clock domains .....	10-2
10.2	Reset domains .....	10-5
10.3	Power control .....	10-10
<b>Chapter 11</b>	<b>Design for Test</b>	
11.1	MBIST .....	11-2
11.2	ATPG test features .....	11-37
<b>Chapter 12</b>	<b>Debug</b>	
12.1	Debug systems .....	12-2
12.2	About the debug unit .....	12-4
12.3	Debug register interface .....	12-7
12.4	Debug register descriptions .....	12-17
12.5	Management registers .....	12-54
12.6	Debug events .....	12-70
12.7	Debug exception .....	12-74
12.8	Debug state .....	12-78
12.9	Cache debug .....	12-87
12.10	External debug interface .....	12-89
12.11	Using the debug functionality .....	12-94
12.12	Debugging systems with energy management capabilities .....	12-116
<b>Chapter 13</b>	<b>NEON and VFP Programmers Model</b>	
13.1	About the NEON and VFP programmers model .....	13-2
13.2	General-purpose registers .....	13-4
13.3	Short vectors .....	13-6
13.4	System registers .....	13-12

13.5	Modes of operation .....	13-21
13.6	Compliance with the IEEE 754 standard .....	13-23
<b>Chapter 14</b>	<b>Embedded Trace Macrocell</b>	
14.1	About the ETM .....	14-2
14.2	ETM configuration .....	14-6
14.3	ETM register summary .....	14-8
14.4	ETM register descriptions .....	14-10
14.5	Precision of TraceEnable and ViewData .....	14-23
14.6	Exact match bit .....	14-26
14.7	Context ID tracing .....	14-28
14.8	Instrumentation instructions .....	14-29
14.9	Idle state control .....	14-30
14.10	Interaction with the Performance Monitoring Unit .....	14-32
<b>Chapter 15</b>	<b>Cross Trigger Interface</b>	
15.1	About the CTI .....	15-2
15.2	Trigger inputs and outputs .....	15-6
15.3	Connecting asynchronous channel interfaces .....	15-8
15.4	About the CTI programmers model .....	15-9
15.5	CTI register summary .....	15-10
15.6	CTI register descriptions .....	15-13
15.7	CTI Integration Test Registers .....	15-24
15.8	CTI CoreSight defined registers .....	15-30
<b>Chapter 16</b>	<b>Instruction Cycle Timing</b>	
16.1	About instruction cycle timing .....	16-2
16.2	Instruction-specific scheduling for ARM instructions .....	16-3
16.3	Dual-instruction issue restrictions .....	16-16
16.4	Other pipeline-dependent latencies .....	16-17
16.5	Advanced SIMD instruction scheduling .....	16-21
16.6	Instruction-specific scheduling for Advanced SIMD instructions .....	16-23
16.7	VFP instructions .....	16-44
16.8	Scheduling example .....	16-49
<b>Chapter 17</b>	<b>AC Characteristics</b>	
17.1	About setup and hold times .....	17-2
17.2	AXI interface .....	17-4
17.3	ATB and CTI interfaces .....	17-6
17.4	APB interface and miscellaneous debug signals .....	17-7
17.5	L1 and L2 MBIST interfaces .....	17-9
17.6	L2 preload interface .....	17-10
17.7	DFT interface .....	17-11
17.8	Miscellaneous signals .....	17-12



<b>Appendix A</b>	<b>Signal Descriptions</b>	
A.1	AXI interface .....	A-2
A.2	ATB interface .....	A-3
A.3	MBIST and DFT interface .....	A-4
A.4	Preload engine interface .....	A-7
A.5	APB interface .....	A-8
A.6	Miscellaneous signals .....	A-10
A.7	Miscellaneous debug signals .....	A-14
A.8	Miscellaneous ETM and CTI signals .....	A-17
<b>Appendix B</b>	<b>Instruction Mnemonics</b>	
B.1	Advanced SIMD data-processing instructions .....	B-2
B.2	VFP data-processing instructions .....	B-5
<b>Appendix C</b>	<b>Revisions</b>	
	<b>Glossary</b>	



# List of Tables

## Cortex-A8 Technical Reference Manual

	Change history .....	ii
Table 1-1	Cortex-A8 configurable options .....	1-11
Table 2-1	ThumbEE Configuration Register bit functions .....	2-7
Table 2-2	ThumbEE HandlerBase Register bit functions .....	2-8
Table 2-3	Access to ThumbEE registers .....	2-9
Table 2-4	Jazelle Identity Register bit functions .....	2-10
Table 2-5	Jazelle Main Configuration Register bit functions .....	2-11
Table 2-6	Jazelle OS Control Register bit functions .....	2-12
Table 2-7	Address types in the processor system .....	2-21
Table 2-8	Mode structure .....	2-22
Table 2-9	Register mode identifiers .....	2-24
Table 2-10	GE[3:0] settings .....	2-30
Table 2-11	PSR mode bit values .....	2-32
Table 2-12	Exception entry and exit .....	2-35
Table 2-13	Exception priorities .....	2-42
Table 3-1	System control coprocessor register functions .....	3-3
Table 3-2	CP15 registers affected by CP15SDISABLE .....	3-6
Table 3-3	Summary of CP15 registers and operations .....	3-9
Table 3-4	Main ID Register bit functions .....	3-25
Table 3-5	Results of access to the Main ID Register .....	3-26
Table 3-6	Cache Type Register bit functions .....	3-27
Table 3-7	Results of access to the Cache Type Register .....	3-27
Table 3-8	Results of access to the TCM Type Register .....	3-28

Table 3-9	TLB Type Register bit functions .....	3-29
Table 3-10	Results of access to the TLB Type Register .....	3-29
Table 3-11	Results of access to the Multiprocessor ID Register .....	3-30
Table 3-12	Processor Feature Register 0 bit functions .....	3-31
Table 3-13	Results of access to the Processor Feature Register 0 .....	3-31
Table 3-14	Processor Feature Register 1 bit functions .....	3-32
Table 3-15	Results of access to Processor Feature Register 1 .....	3-32
Table 3-16	Debug Feature Register 0 bit functions .....	3-33
Table 3-17	Results of access to Debug Feature Register 0 .....	3-34
Table 3-18	Results of access to Auxiliary Feature Register 0 .....	3-35
Table 3-19	Memory Model Feature Register 0 bit functions .....	3-36
Table 3-20	Results of access to Memory Model Feature Register 0 .....	3-36
Table 3-21	Memory Model Feature Register 1 bit functions .....	3-37
Table 3-22	Results of access to Memory Model Feature Register 1 .....	3-39
Table 3-23	Memory Model Feature Register 2 bit functions .....	3-40
Table 3-24	Results of access to Memory Model Feature Register 2 .....	3-41
Table 3-25	Memory Model Feature Register 3 bit functions .....	3-42
Table 3-26	Results of access to Memory Model Feature Register 3 .....	3-42
Table 3-27	Instruction Set Attributes Register 0 bit functions .....	3-43
Table 3-28	Results of access to Instruction Set Attributes Register 0 .....	3-44
Table 3-29	Instruction Set Attributes Register 1 bit functions .....	3-45
Table 3-30	Results of access to Instruction Set Attributes Register 1 .....	3-46
Table 3-31	Instruction Set Attributes Register 2 bit functions .....	3-47
Table 3-32	Results of access to Instruction Set Attributes Register 2 .....	3-48
Table 3-33	Instruction Set Attributes Register 3 bit functions .....	3-49
Table 3-34	Results of access to Instruction Set Attributes Register 3 .....	3-50
Table 3-35	Instruction Set Attributes Register 4 bit functions .....	3-51
Table 3-36	Results of access to Instruction Set Attributes Register 4 .....	3-51
Table 3-37	Cache Level ID Register bit functions .....	3-52
Table 3-38	Results of access to the Cache Level ID Register .....	3-53
Table 3-39	Silicon ID Register bit functions .....	3-54
Table 3-40	Results of access to the Silicon ID Register .....	3-54
Table 3-41	Cache Size Identification Register bit functions .....	3-55
Table 3-42	Encodings of the Cache Size Identification Register .....	3-56
Table 3-43	Results of access to the Cache Size Identification Register .....	3-56
Table 3-44	Cache Size Selection Register bit functions .....	3-57
Table 3-45	Results of access to the Cache Size Selection Register .....	3-57
Table 3-46	Control Register bit functions .....	3-59
Table 3-47	Results of access to the Control Register .....	3-60
Table 3-48	Behavior of the processor when enabling caches .....	3-61
Table 3-49	Auxiliary Control Register bit functions .....	3-63
Table 3-50	Results of access to the Auxiliary Control Register .....	3-67
Table 3-51	Coprocessor Access Control Register bit functions .....	3-68
Table 3-52	Results of access to the Coprocessor Access Control Register .....	3-69
Table 3-53	Secure Configuration Register bit functions .....	3-70
Table 3-54	Operation of the FW and FIQ bits .....	3-71
Table 3-55	Operation of the AW and EA bits .....	3-72

Table 3-56	Secure Debug Enable Register bit functions .....	3-73
Table 3-57	Results of access to the Secure Debug Enable Register .....	3-73
Table 3-58	Nonsecure Access Control Register bit functions .....	3-74
Table 3-59	Results of access to the Auxiliary Control Register .....	3-75
Table 3-60	Translation Table Base Register 0 bit functions .....	3-76
Table 3-61	Results of access to the Translation Table Base Register 0 .....	3-77
Table 3-62	Translation Table Base Register 1 bit functions .....	3-78
Table 3-63	Results of access to the Translation Table Base Register 1 .....	3-78
Table 3-64	Translation Table Base Control Register bit functions .....	3-80
Table 3-65	Results of access to the Translation Table Base Control Register .....	3-81
Table 3-66	Domain Access Control Register bit functions .....	3-82
Table 3-67	Results of access to the Domain Access Control Register .....	3-82
Table 3-68	Data Fault Status Register bit functions .....	3-83
Table 3-69	Instruction Fault Status Register bit functions .....	3-85
Table 3-70	Results of access to the Auxiliary Fault Status Registers .....	3-87
Table 3-71	Results of access to the Data Fault Address Register .....	3-88
Table 3-72	Results of access to the Instruction Fault Address Register .....	3-89
Table 3-73	Register c7 cache and prefetch buffer maintenance operations .....	3-90
Table 3-74	Functional bits of c7 for set and way .....	3-91
Table 3-75	Values of A, L, and S for L1 cache sizes .....	3-92
Table 3-76	Values of A, L, and S for L2 cache sizes .....	3-92
Table 3-77	Functional bits of c7 for MVA .....	3-93
Table 3-78	PA Register for successful translation bit functions .....	3-94
Table 3-79	PA Register for unsuccessful translation bit functions .....	3-96
Table 3-80	Results of access to the data synchronization barrier operation .....	3-99
Table 3-81	Results of access to the data memory barrier operation .....	3-99
Table 3-82	Performance Monitor Control Register bit functions .....	3-102
Table 3-83	Results of access to the Performance Monitor Control Register .....	3-103
Table 3-84	Count Enable Set Register bit functions .....	3-104
Table 3-85	Results of access to the Count Enable Set Register .....	3-104
Table 3-86	Count Enable Clear Register bit functions .....	3-105
Table 3-87	Results of access to the Count Enable Clear Register .....	3-105
Table 3-88	Overflow Flag Status Register bit functions .....	3-106
Table 3-89	Results of access to the Overflow Flag Status Register .....	3-107
Table 3-90	Software Increment Register bit functions .....	3-108
Table 3-91	Results of access to the Software Increment Register .....	3-108
Table 3-92	Performance Counter Selection Register bit functions .....	3-109
Table 3-93	Results of access to the Performance Counter Selection Register .....	3-109
Table 3-94	Results of access to the Cycle Count Register .....	3-110
Table 3-95	Event Selection Register bit functions .....	3-111
Table 3-96	Results of access to the Event Selection Register .....	3-111
Table 3-97	Values for predefined events .....	3-112
Table 3-98	Results of access to the Performance Monitor Count Registers .....	3-116
Table 3-99	Signal settings for the Performance Monitor Count Registers .....	3-116
Table 3-100	User Enable Register bit functions .....	3-117
Table 3-101	Results of access to the User Enable Register .....	3-118
Table 3-102	Interrupt Enable Set Register bit functions .....	3-119

Table 3-103	Results of access to the Interrupt Enable Set Register .....	3-119
Table 3-104	Interrupt Enable Clear Register bit functions .....	3-120
Table 3-105	Results of access to the Interrupt Enable Clear Register .....	3-120
Table 3-106	L2 Cache Lockdown Register bit functions .....	3-122
Table 3-107	Results of access to the L2 Cache Lockdown Register .....	3-123
Table 3-108	L2 Cache Auxiliary Control Register bit functions .....	3-125
Table 3-109	Results of access to the L2 Cache Auxiliary Control Register .....	3-127
Table 3-110	TLB Lockdown Register bit functions .....	3-129
Table 3-111	Results of access to the TLB Lockdown Register .....	3-130
Table 3-112	Application of remapped registers on memory access .....	3-132
Table 3-113	Primary Region Remap Register bit functions .....	3-133
Table 3-114	Encoding for the remapping of the primary memory type .....	3-134
Table 3-115	Normal Memory Remap Register bit functions .....	3-134
Table 3-116	Remap encoding for inner or outer cacheable attributes .....	3-136
Table 3-117	Results of access to the memory region remap registers .....	3-136
Table 3-118	PLE Identification and Status Register bit functions .....	3-138
Table 3-119	Opcode_2 values for PLE Identification and Status Register functions .....	3-138
Table 3-120	Results of access to the PLE Identification and Status Registers .....	3-139
Table 3-121	PLE User Accessibility Register bit functions .....	3-140
Table 3-122	Results of access to the PLE User Accessibility Register .....	3-140
Table 3-123	PLE Channel Number Register bit functions .....	3-141
Table 3-124	Results of access to the PLE User Accessibility Register .....	3-142
Table 3-125	Results of access to the PLE enable commands .....	3-143
Table 3-126	PLE Control Register bit functions .....	3-144
Table 3-127	Writing to UM bit [26] .....	3-145
Table 3-128	Results of access to the PLE Control Registers .....	3-146
Table 3-129	Results of access to the PLE Internal Start Address Register .....	3-147
Table 3-130	Maximum transfer size for various L2 cache sizes .....	3-148
Table 3-131	Results of access to the PLE Internal End Address Register .....	3-149
Table 3-132	PLE Channel Status Register bit functions .....	3-150
Table 3-133	Results of access to the PLE Channel Status Register .....	3-151
Table 3-134	PLE Context ID Register bit functions .....	3-152
Table 3-135	Results of access to the PLE Context ID Register .....	3-152
Table 3-136	Secure or Nonsecure Vector Base Address Register bit functions .....	3-153
Table 3-137	Results of access to the Secure or Nonsecure Vector Base Address Register ...	3-154
Table 3-138	Monitor Vector Base Address Register bit functions .....	3-155
Table 3-139	Results of access to the Monitor Vector Base Address Register .....	3-155
Table 3-140	Interrupt Status Register bit functions .....	3-156
Table 3-141	Results of access to the Interrupt Status Register .....	3-157
Table 3-142	FCSE PID Register bit functions .....	3-158
Table 3-143	Results of access to the FCSE PID Register .....	3-158
Table 3-144	Context ID Register bit functions .....	3-160
Table 3-145	Results of access to the Context ID Register .....	3-160
Table 3-146	Results of access to the Thread and Process ID Registers .....	3-162
Table 3-147	Functional bits of I-L1 or D-L1 Data 0 Register for a TLB CAM operation .....	3-164
Table 3-148	Functional bits of I-L1 or D-L1 Data 1 Register for a TLB CAM operation .....	3-164
Table 3-149	Functional bits of I-L1 or D-L1 Data 0 Register for a TLB ATTR operation .....	3-165

Table 3-150	Functional bits of I-L1 or D-L1 Data 0 Register for a TLB PA array operation .....	3-165
Table 3-151	Functional bits of I-L1 or D-L1 Data 0 Register for an HVAB array operation .....	3-166
Table 3-152	Functional bits of I-L1 or D-L1 Data 0 Register for an L1 tag array operation .....	3-166
Table 3-153	Functional bits of I-L1 or D-L1 Data 0 Register for L1 data array operation .....	3-166
Table 3-154	Functional bits of I-L1 or D-L1 Data 1 Register for L1 data array operation .....	3-167
Table 3-155	Functional bits of I-L1 Data 0 Register for a BTB array operation .....	3-167
Table 3-156	Functional bits of I-L1 Data 1 Register for a BTB array operation .....	3-167
Table 3-157	Functional bits of I-L1 Data 0 Register for a GHB array operation .....	3-168
Table 3-158	Functional bits of L2 Data 0 Register for an L2 parity/ECC operation .....	3-179
Table 3-159	Functional bits of L2 Data 0 Register for a tag RAM operation .....	3-179
Table 3-160	Functional bits of L2 Data 0 Register for a data RAM operation .....	3-180
Table 3-161	Functional bits of L2 Data 1 Register for a data RAM operation .....	3-180
Table 3-162	Functional bits of L2 Data 2 Register for a data RAM operation .....	3-180
Table 3-163	Address field values .....	3-181
Table 4-1	NEON normal memory alignment qualifiers .....	4-4
Table 6-1	CP15 register functions .....	6-8
Table 7-1	Memory types affecting L1 and L2 cache flows .....	7-7
Table 8-1	L2 cache transfer policy .....	8-4
Table 8-2	Cacheable and noncacheable memory region types .....	8-9
Table 9-1	Read address channel AXI ID .....	9-4
Table 9-2	Write address channel AXI ID .....	9-5
Table 9-3	AXI master interface attributes .....	9-5
Table 9-4	A64n128 encoding .....	9-6
Table 9-5	AXI address channel for instruction transactions .....	9-7
Table 9-6	Number of transfers on AXI write channel for an eviction .....	9-8
Table 9-7	AXI address channel for data transactions - excluding load/store multiples .....	9-10
Table 9-8	AXI address channel for data transactions for load/store multiples .....	9-16
Table 10-1	Reset inputs .....	10-5
Table 10-2	Valid power domains .....	10-17
Table 11-1	MBIST register summary .....	11-3
Table 11-2	Selecting a test pattern with pttm[5:0] .....	11-4
Table 11-3	Selecting the L1 arrays to test with L1_array_sel[22:0] .....	11-5
Table 11-4	L1_config[14:0] .....	11-6
Table 11-5	Configuring the number of L1 array rows with L1_config[14:0] .....	11-7
Table 11-6	Selecting L2 RAMs for test with L2_ram_sel[4:0] .....	11-8
Table 11-7	L2_config[22:0] .....	11-9
Table 11-8	Selecting L2 data array latency with L2DLat[3:0] .....	11-9
Table 11-9	Selecting L2 tag array latency with L2TLat[1:0] .....	11-10
Table 11-10	Selecting the L2 RAMs with L2Rows[11:0] .....	11-10
Table 11-11	Configuring the number of L2 RAM rows with L2Rows[11:0] .....	11-11
Table 11-12	Valid L2 array row numbers .....	11-11
Table 11-13	Selecting the L2ValSer test type .....	11-12
Table 11-14	Selecting L2 RAMs for LSB control .....	11-12
Table 11-15	Selecting counting sequence of L2 RAM column address LSBs .....	11-12
Table 11-16	GNG[10:0] field .....	11-13
Table 11-17	L2 cache way grouping .....	11-16
Table 11-18	Identifying failing L2 bits with failing_bits[32:0] .....	11-17

Table 11-19	Summary of MBIST patterns .....	11-25
Table 12-1	Access to CP14 debug registers .....	12-7
Table 12-2	CP14 debug registers summary .....	12-8
Table 12-3	Debug memory-mapped registers .....	12-9
Table 12-4	Processor reset effect on debug and ETM logic .....	12-12
Table 12-5	APB interface access with relation to software lock .....	12-14
Table 12-6	Debug registers access with relation to power-down event .....	12-15
Table 12-7	Power management registers access with relation to power-down event .....	12-15
Table 12-8	ETM and CTI registers access with relation to power-down event .....	12-16
Table 12-9	Terms used in register descriptions .....	12-17
Table 12-10	CP14 debug registers .....	12-17
Table 12-11	Debug ID Register bit functions .....	12-19
Table 12-12	Debug ROM Address Register bit functions .....	12-20
Table 12-13	Debug Self Address Offset Register bit functions .....	12-21
Table 12-14	Debug Status and Control Register bit functions .....	12-23
Table 12-15	Data Transfer Register bit functions .....	12-30
Table 12-16	Watchpoint Fault Address Register bit functions .....	12-30
Table 12-17	Vector Catch Register bit functions .....	12-32
Table 12-18	Event Catch Register bit functions .....	12-34
Table 12-19	Debug State Cache Control Register bit functions .....	12-35
Table 12-20	Instruction Transfer Register bit functions .....	12-36
Table 12-21	Debug Run Control Register bit functions .....	12-37
Table 12-22	Breakpoint Value Registers bit functions .....	12-38
Table 12-23	Breakpoint Control Registers bit functions .....	12-39
Table 12-24	Meaning of BVR bits [22:20] .....	12-41
Table 12-25	Watchpoint Value Registers bit functions .....	12-42
Table 12-26	Watchpoint Control Registers bit functions .....	12-44
Table 12-27	OS Lock Access Register bit functions .....	12-47
Table 12-28	OS Lock Status Register bit functions .....	12-48
Table 12-29	OS Save and Restore Register bit functions .....	12-49
Table 12-30	PRCR bit functions .....	12-51
Table 12-31	PRSR bit functions .....	12-52
Table 12-32	Management registers .....	12-54
Table 12-33	Processor Identifier Registers .....	12-55
Table 12-34	Integration Internal Output Control Register bit functions .....	12-57
Table 12-35	Integration External Output Control Register bit functions .....	12-59
Table 12-36	Integration Input Status Register bit functions .....	12-60
Table 12-37	Integration Mode Control Register bit functions .....	12-61
Table 12-38	Claim Tag Set Register bit functions .....	12-62
Table 12-39	Claim Tag Clear Register bit functions .....	12-62
Table 12-40	Lock Access Register bit functions .....	12-63
Table 12-41	Lock Status Register bit functions .....	12-64
Table 12-42	Authentication Status Register bit functions .....	12-65
Table 12-43	Device Type Register bit functions .....	12-66
Table 12-44	Peripheral Identification Registers .....	12-66
Table 12-45	Fields in the Peripheral Identification Registers .....	12-67
Table 12-46	Peripheral ID Register 0 bit functions .....	12-67



Table 12-47	Peripheral ID Register 1 bit functions .....	12-68
Table 12-48	Peripheral ID Register 2 bit functions .....	12-68
Table 12-49	Peripheral ID Register 3 bit functions .....	12-68
Table 12-50	Peripheral ID Register 4 bit functions .....	12-69
Table 12-51	Component Identification Registers .....	12-69
Table 12-52	Processor behavior on debug events .....	12-72
Table 12-53	Values in Link Register after exceptions .....	12-75
Table 12-54	Read PC value after debug state entry .....	12-79
Table 12-55	Permitted updates to the CPSR in debug state .....	12-82
Table 12-56	Accesses to CP15 and CP14 registers in debug state .....	12-83
Table 12-57	Authentication signal restrictions .....	12-91
Table 12-58	Values to write to BCR for a simple breakpoint .....	12-98
Table 12-59	Values to write to WCR for a simple watchpoint .....	12-100
Table 12-60	Example byte address masks for watchpointed objects .....	12-101
Table 13-1	Single-precision three-operand register usage .....	13-10
Table 13-2	Single-precision two-operand register usage .....	13-11
Table 13-3	Double-precision three-operand register usage .....	13-11
Table 13-4	Double-precision two-operand register usage .....	13-11
Table 13-5	NEON and VFP system registers .....	13-12
Table 13-6	Accessing NEON and VFP system registers .....	13-12
Table 13-7	FPSID Register bit functions .....	13-14
Table 13-8	FPSCR Register bit functions .....	13-15
Table 13-9	Vector length and stride combinations .....	13-17
Table 13-10	Floating-Point Exception Register bit functions .....	13-18
Table 13-11	MVFR0 Register bit functions .....	13-19
Table 13-12	MVFR1 Register bit functions .....	13-20
Table 13-13	Default NaN values .....	13-23
Table 13-14	QNaN and SNaN handling .....	13-24
Table 14-1	ETM implementation .....	14-6
Table 14-2	ETM register summary .....	14-8
Table 14-3	ID Register bit functions .....	14-11
Table 14-4	Configuration Code Register bit functions .....	14-12
Table 14-5	Configuration Code Extension Register bit functions .....	14-13
Table 14-6	Peripheral Identification Registers bit functions .....	14-14
Table 14-7	Component Identification Registers bit functions .....	14-16
Table 14-8	Output signals that can be controlled by the Integration Test Registers .....	14-17
Table 14-9	Input signals that can be read by the Integration Test Registers .....	14-17
Table 14-10	ITMISCOUT Register bit functions .....	14-18
Table 14-11	ITMISCIN Register bit functions .....	14-19
Table 14-12	ITTRIGGER Register bit functions .....	14-19
Table 14-13	ITATBDATA0 Register bit functions .....	14-20
Table 14-14	ITATBCTR2 Register bit functions .....	14-21
Table 14-15	ITATBCTR1 Register bit functions .....	14-21
Table 14-16	ITATBCTR0 Register bit functions .....	14-22
Table 14-17	PMU event number mappings .....	14-32
Table 14-18	PMU event cycle mappings .....	14-34
Table 15-1	Trigger inputs .....	15-6

Table 15-2	Trigger outputs .....	15-6
Table 15-3	CTI register summary .....	15-10
Table 15-4	CTI Control Register bit functions .....	15-13
Table 15-5	CTI Interrupt Acknowledge Register bit functions .....	15-14
Table 15-6	CTI Application Trigger Set Register bit functions .....	15-15
Table 15-7	CTI Application Trigger Clear Register bit functions .....	15-16
Table 15-8	CTI Application Pulse Register bit functions .....	15-16
Table 15-9	CTI Trigger to Channel Enable Registers bit functions .....	15-17
Table 15-10	CTI Channel to Trigger Enable Registers bit functions .....	15-18
Table 15-11	CTI Trigger In Status Register bit functions .....	15-19
Table 15-12	CTI Trigger Out Status Register bit functions .....	15-19
Table 15-13	CTI Channel In Status Register bit functions .....	15-20
Table 15-14	CTI Channel Gate Register bit functions .....	15-21
Table 15-15	ASIC Control Register bit functions .....	15-22
Table 15-16	CTI Channel Out Status Register bit functions .....	15-23
Table 15-17	CTI Integration Test Registers .....	15-24
Table 15-18	ITTRIGINACK Register bit functions .....	15-25
Table 15-19	ITCHOUT Register bit functions .....	15-25
Table 15-20	ITTRIGOUT Register bit functions .....	15-26
Table 15-21	ITTRIGOUT connections to other integration test registers .....	15-26
Table 15-22	ITTRIGOUTACK Register bit functions .....	15-27
Table 15-23	ITTRIGOUTACK connections to other integration test registers .....	15-27
Table 15-24	ITCHIN Register bit functions .....	15-28
Table 15-25	ITTRIGIN Register bit functions .....	15-28
Table 15-26	ITTRIGIN connections to other integration test registers .....	15-29
Table 15-27	Authentication Status Register bit functions .....	15-30
Table 15-28	Device ID Register bit functions .....	15-30
Table 15-29	Device Type Identifier Register bit functions .....	15-31
Table 15-30	Peripheral Identification Registers bit functions .....	15-31
Table 15-31	Component Identification Registers bit functions .....	15-33
Table 16-1	Data-processing instructions with a destination .....	16-5
Table 16-2	Data-processing instructions without a destination .....	16-6
Table 16-3	MOV and MOVN instructions .....	16-6
Table 16-4	Multiply instructions .....	16-7
Table 16-5	Parallel arithmetic instructions .....	16-8
Table 16-6	Extended instructions .....	16-8
Table 16-7	Miscellaneous data-processing instructions .....	16-8
Table 16-8	Status register access instructions .....	16-9
Table 16-9	Load instructions .....	16-9
Table 16-10	Store instructions .....	16-10
Table 16-11	Branch instructions .....	16-12
Table 16-12	Nonpipelined CP14 instructions .....	16-12
Table 16-13	Nonpipelined CP15 instructions .....	16-13
Table 16-14	CP15 instructions affected when ACTRL bit[20] = 0 .....	16-15
Table 16-15	Dual-issue restrictions .....	16-16
Table 16-16	Memory system effects on instruction timings .....	16-18
Table 16-17	ThumbEE instructions .....	16-19

Table 16-18	Advanced SIMD integer ALU instructions .....	16-25
Table 16-19	Advanced SIMD integer multiply instructions .....	16-28
Table 16-20	Advanced SIMD integer shift instructions .....	16-31
Table 16-21	Advanced SIMD floating-point instructions .....	16-32
Table 16-22	Advanced SIMD byte permute instructions .....	16-34
Table 16-23	Advanced SIMD load/store instructions .....	16-36
Table 16-24	Advanced SIMD register transfer instructions .....	16-43
Table 16-25	VFP Instruction cycle counts .....	16-45
Table 17-1	Format of timing parameter tables .....	17-3
Table 17-2	Timing parameters of AXI interface .....	17-4
Table 17-3	Timing parameters of ATB and CTI interfaces .....	17-6
Table 17-4	Timing parameters of APB interface and miscellaneous debug signals .....	17-7
Table 17-5	Timing parameters of the L1 and L2 MBIST interface .....	17-9
Table 17-6	Timing parameters of the L2 preload interface .....	17-10
Table 17-7	Timing parameters of the DFT interface .....	17-11
Table 17-8	Timing parameters of miscellaneous signals .....	17-12
Table A-1	AXI interface .....	A-2
Table A-2	ATB interface .....	A-3
Table A-3	MBIST interface .....	A-4
Table A-4	DFT and additional MBIST pin requirements .....	A-5
Table A-5	Preload engine interface .....	A-7
Table A-6	APB interface .....	A-8
Table A-7	Miscellaneous signals .....	A-10
Table A-8	Miscellaneous debug signals .....	A-14
Table A-9	Miscellaneous ETM and CTI signals .....	A-17
Table B-1	Advanced SIMD mnemonics .....	B-2
Table B-2	VFP data-processing mnemonics .....	B-5
Table C-1	Differences between issue F and issue G .....	C-1
Table C-2	Differences between issue G and issue H .....	C-2
Table C-3	Differences between issue H and issue I .....	C-3



# List of Figures

## Cortex-A8 Technical Reference Manual

	Key to timing diagram conventions .....	xxxi
Figure 1-1	Cortex-A8 block diagram .....	1-4
Figure 2-1	32-bit ARM Thumb-2 instruction format .....	2-4
Figure 2-2	ThumbEE Configuration Register format .....	2-7
Figure 2-3	ThumbEE HandlerBase Register format .....	2-8
Figure 2-4	Jazelle Identity Register format .....	2-10
Figure 2-5	Jazelle Main Configuration Register format .....	2-11
Figure 2-6	Jazelle OS Control Register format .....	2-12
Figure 2-7	Secure and Nonsecure states .....	2-14
Figure 2-8	Big-endian addresses of bytes within words .....	2-19
Figure 2-9	Little-endian addresses of bytes within words .....	2-20
Figure 2-10	Register organization in ARM state .....	2-25
Figure 2-11	Processor register set showing banked registers .....	2-26
Figure 2-12	Program status register .....	2-27
Figure 3-1	Main ID Register format .....	3-25
Figure 3-2	Cache Type Register format .....	3-26
Figure 3-3	TLB Type Register format .....	3-28
Figure 3-4	Processor Feature Register 0 format .....	3-30
Figure 3-5	Processor Feature Register 1 format .....	3-32
Figure 3-6	Debug Feature Register 0 format .....	3-33
Figure 3-7	Memory Model Feature Register 0 format .....	3-35
Figure 3-8	Memory Model Feature Register 1 format .....	3-37
Figure 3-9	Memory Model Feature Register 2 format .....	3-39

Figure 3-10	Memory Model Feature Register 3 format .....	3-42
Figure 3-11	Instruction Set Attributes Register 0 format .....	3-43
Figure 3-12	Instruction Set Attributes Register 1 format .....	3-45
Figure 3-13	Instruction Set Attributes Register 2 format .....	3-46
Figure 3-14	Instruction Set Attributes Register 3 format .....	3-48
Figure 3-15	Instruction Set Attributes Register 4 format .....	3-50
Figure 3-16	Cache Level ID Register format .....	3-52
Figure 3-17	Silicon ID Register format .....	3-54
Figure 3-18	Cache Size Identification Register format .....	3-55
Figure 3-19	Cache Size Selection Register format .....	3-57
Figure 3-20	Control Register bit assignments .....	3-58
Figure 3-21	Auxiliary Control Register format .....	3-62
Figure 3-22	Coprocessor Access Control Register format .....	3-68
Figure 3-23	Secure Configuration Register format .....	3-70
Figure 3-24	Secure Debug Enable Register format .....	3-72
Figure 3-25	Nonsecure Access Control Register format .....	3-74
Figure 3-26	Translation Table Base Register 0 format .....	3-76
Figure 3-27	Translation Table Base Register 1 format .....	3-78
Figure 3-28	Translation Table Base Control Register format .....	3-79
Figure 3-29	Domain Access Control Register format .....	3-82
Figure 3-30	Data Fault Status Register format .....	3-83
Figure 3-31	Instruction Fault Status Register format .....	3-85
Figure 3-32	c7 format for set and way .....	3-91
Figure 3-33	c7 format for MVA .....	3-92
Figure 3-34	PA Register format for successful translation .....	3-94
Figure 3-35	PA Register format for unsuccessful translation .....	3-94
Figure 3-36	TLB Operations MVA and ASID format .....	3-100
Figure 3-37	TLB Operations ASID format .....	3-101
Figure 3-38	Performance Monitor Control Register format .....	3-101
Figure 3-39	Count Enable Set Register format .....	3-103
Figure 3-40	Count Enable Clear Register format .....	3-105
Figure 3-41	Overflow Flag Status Register format .....	3-106
Figure 3-42	Software Increment Register format .....	3-107
Figure 3-43	Performance Counter Selection Register format .....	3-109
Figure 3-44	Event Selection Register format .....	3-111
Figure 3-45	User Enable Register format .....	3-117
Figure 3-46	Interrupt Enable Set Register format .....	3-118
Figure 3-47	Interrupt Enable Clear Register format .....	3-120
Figure 3-48	L2 Cache Lockdown Register format .....	3-121
Figure 3-49	L2 Cache Auxiliary Control Register format .....	3-125
Figure 3-50	TLB Lockdown Register format .....	3-129
Figure 3-51	Primary Region Remap Register format .....	3-132
Figure 3-52	Normal Memory Remap Register format .....	3-134
Figure 3-53	PLE identification and Status Registers format .....	3-137
Figure 3-54	PLE User Accessibility Register format .....	3-139
Figure 3-55	PLE Channel Number Register format .....	3-141
Figure 3-56	PLE Control Register format .....	3-144

Figure 3-57	PLE Internal Start Address Register bit format .....	3-147
Figure 3-58	PLE Internal End Address Register format .....	3-148
Figure 3-59	PLE Channel Status Register format .....	3-149
Figure 3-60	PLE Context ID Register format .....	3-151
Figure 3-61	Secure or Nonsecure Vector Base Address Register format .....	3-153
Figure 3-62	Monitor Vector Base Address Register format .....	3-154
Figure 3-63	Interrupt Status Register format .....	3-156
Figure 3-64	FCSE PID Register format .....	3-158
Figure 3-65	Address mapping with the FCSE PID Register .....	3-159
Figure 3-66	Context ID Register format .....	3-160
Figure 3-67	Instruction and Data side Data 0 Registers format .....	3-163
Figure 3-68	Instruction and Data side Data 1 Registers format .....	3-164
Figure 3-69	L1 TLB CAM read operation format .....	3-168
Figure 3-70	L1 TLB CAM write operation format .....	3-169
Figure 3-71	L1 HVAB array read operation format .....	3-171
Figure 3-72	L1 HVAB array write operation format .....	3-171
Figure 3-73	L1 tag array read operation format .....	3-172
Figure 3-74	L1 tag array write operation format .....	3-173
Figure 3-75	L1 data array read operation format .....	3-174
Figure 3-76	L1 data array write operation format .....	3-174
Figure 3-77	BTB array read operation format .....	3-175
Figure 3-78	BTB array write operation format .....	3-176
Figure 3-79	GHB array read operation format .....	3-177
Figure 3-80	GHB array write operation format .....	3-177
Figure 3-81	L2 Data 0 Register format .....	3-178
Figure 3-82	L2 Data 1 Register format .....	3-178
Figure 3-83	L2 Data 2 Register format .....	3-179
Figure 3-84	L2 parity/ECC array read operation format .....	3-181
Figure 3-85	L2 parity/ECC array write operation format .....	3-182
Figure 3-86	L2 tag array read operation format .....	3-183
Figure 3-87	L2 tag array write operation format .....	3-183
Figure 3-88	L2 data RAM array read operation format .....	3-184
Figure 3-89	L2 data RAM array write operation format .....	3-184
Figure 6-1	16MB supersection descriptor format .....	6-4
Figure 8-1	L2 cache bank structure .....	8-4
Figure 10-1	CLK duty cycle .....	10-2
Figure 10-2	CLK-to-ACLK ratio of 4:1 .....	10-3
Figure 10-3	Changing the CLK-to-ACLK ratio from 4:1 to 1:1 .....	10-3
Figure 10-4	Changing the PCLK-to-internal-PCLK ratio from 4:1 to 1:1 .....	10-4
Figure 10-5	Changing the ATCLK-to-internal-ATCLK ratio from 4:1 to 1:1 .....	10-4
Figure 10-6	Power-on reset timing .....	10-6
Figure 10-7	Soft reset timing .....	10-7
Figure 10-8	PRESETn and ATRESETn assertion .....	10-8
Figure 10-9	STANDBYWFI deassertion .....	10-12
Figure 10-10	CLKSTOPREQ and CLKSTOPACK .....	10-12
Figure 10-11	Power domains .....	10-16
Figure 10-12	Voltage domains .....	10-18

Figure 10-13	Retention power domains .....	10-27
Figure 11-1	L1 MBIST Instruction Register bit assignments .....	11-3
Figure 11-2	L2 MBIST Instruction Register bit assignments .....	11-8
Figure 11-3	L1 and L2 MBIST GO-NOGO Instruction Registers bit assignments .....	11-13
Figure 11-4	L1 MBIST GO-NOGO Instruction Register example with two patterns .....	11-14
Figure 11-5	L1 MBIST Datalog Register bit assignments .....	11-14
Figure 11-6	L2 MBIST Datalog Register bit assignments .....	11-15
Figure 11-7	Timing of MBIST instruction load .....	11-20
Figure 11-8	Timing of MBIST custom GO-NOGO instruction load .....	11-21
Figure 11-9	Timing of MBIST at-speed execution .....	11-22
Figure 11-10	Timing of MBIST end-of-test datalog retrieval .....	11-23
Figure 11-11	Timing of MBIST start of bitmap datalog retrieval .....	11-23
Figure 11-12	Timing of MBIST end of bitmap datalog retrieval .....	11-24
Figure 11-13	Physical array after pass 1 of CKBD .....	11-27
Figure 11-14	Physical array after pass 1 of COLBAR .....	11-28
Figure 11-15	Physical array after pass 1 of ROWBAR .....	11-29
Figure 11-16	Row 1 column 2 state during pass 2 of RWXMARCH .....	11-30
Figure 11-17	Row 1 column 2 state during pass 2 of RWYMARCH .....	11-30
Figure 11-18	Row 1 column 2 state during pass 2 of RWRXMARCH .....	11-31
Figure 11-19	Row 1 column 2 state during pass 2 of RWRYMARCH .....	11-31
Figure 11-20	Row 1 column 2 state during pass 2 of XMARCHC .....	11-32
Figure 11-21	Row 1 column 2 state during pass 2 of YMARCHC .....	11-33
Figure 11-22	XADDRBAR array accessing and data .....	11-33
Figure 11-23	YADDRBAR array accessing and data .....	11-34
Figure 11-24	WRITEBANG .....	11-35
Figure 11-25	READBANG .....	11-35
Figure 11-26	Input wrapper boundary register cell control logic .....	11-38
Figure 11-27	Output wrapper boundary register cell control logic .....	11-38
Figure 11-28	IEEE 1500-compliant input wrapper boundary register cell .....	11-39
Figure 11-29	Reset handling .....	11-40
Figure 11-30	Safe shift RAM signal .....	11-40
Figure 12-1	Typical debug system .....	12-2
Figure 12-2	Debug ID Register format .....	12-18
Figure 12-3	Debug ROM Address Register format .....	12-20
Figure 12-4	Debug Self Address Offset Register format .....	12-21
Figure 12-5	Debug Status and Control Register format .....	12-22
Figure 12-6	DTR Register format .....	12-30
Figure 12-7	Vector Catch Register format .....	12-31
Figure 12-8	Event Catch Register format .....	12-33
Figure 12-9	Debug State Cache Control Register format .....	12-34
Figure 12-10	ITR format .....	12-35
Figure 12-11	Debug Run Control Register format .....	12-36
Figure 12-12	Breakpoint Control Registers format .....	12-38
Figure 12-13	Watchpoint Control Registers format .....	12-43
Figure 12-14	OS Lock Access Register format .....	12-46
Figure 12-15	OS Lock Status Register format .....	12-47
Figure 12-16	OS Save and Restore Register format .....	12-48



Figure 12-17	PRCR format .....	12-50
Figure 12-18	PRSR format .....	12-52
Figure 12-19	Integration Internal Output Control Register format .....	12-57
Figure 12-20	Integration External Output Control Register format .....	12-58
Figure 12-21	Integration Input Status Register format .....	12-60
Figure 12-22	Integration Mode Control Register format .....	12-61
Figure 12-23	Claim Tag Set Register format .....	12-62
Figure 12-24	Claim Tag Clear Register format .....	12-62
Figure 12-25	Lock Access Register format .....	12-63
Figure 12-26	Lock Status Register format .....	12-64
Figure 12-27	Authentication Status Register format .....	12-65
Figure 12-28	Device Type Register format .....	12-66
Figure 12-29	Timing of core power-down and power-up sequences .....	12-90
Figure 13-1	NEON and VFP register bank .....	13-5
Figure 13-2	Register banks .....	13-7
Figure 13-3	Floating-Point System ID Register format .....	13-13
Figure 13-4	Floating-Point Status and Control Register format .....	13-15
Figure 13-5	Floating-Point Exception Register format .....	13-18
Figure 13-6	MVFR0 Register format .....	13-18
Figure 13-7	MVFR1 Register format .....	13-19
Figure 14-1	Example CoreSight debug environment .....	14-4
Figure 14-2	ID Register format .....	14-10
Figure 14-3	Configuration Code Register format .....	14-12
Figure 14-4	Configuration Code Extension Register format .....	14-13
Figure 14-5	Mapping between the Component ID Registers and the component ID value .....	14-16
Figure 14-6	ITMISCOUT Register format .....	14-18
Figure 14-7	ITMISCIN Register format .....	14-18
Figure 14-8	ITTRIGGER Register format .....	14-19
Figure 14-9	ITATBDATA0 Register format .....	14-20
Figure 14-10	ITATBCTR2 Register format .....	14-20
Figure 14-11	ITATBCTR1 Register format .....	14-21
Figure 14-12	ITATBCTR0 Register format .....	14-22
Figure 15-1	Debug system components .....	15-2
Figure 15-2	Cross Trigger Interface channels .....	15-4
Figure 15-3	Asynchronous to synchronous converter .....	15-8
Figure 15-4	CTI Control Register format .....	15-13
Figure 15-5	CTI Interrupt Acknowledge Register format .....	15-14
Figure 15-6	CTI Application Trigger Set Register format .....	15-14
Figure 15-7	CTI Application Trigger Clear Register format .....	15-15
Figure 15-8	CTI Application Pulse Register format .....	15-16
Figure 15-9	CTI Trigger to Channel Enable Registers format .....	15-17
Figure 15-10	CTI Channel to Trigger Enable Registers format .....	15-18
Figure 15-11	CTI Trigger In Status Register format .....	15-18
Figure 15-12	CTI Trigger Out Status Register format .....	15-19
Figure 15-13	CTI Channel In Status Register format .....	15-20
Figure 15-14	CTI Channel Gate Register format .....	15-21
Figure 15-15	ASIC Control Register format .....	15-22

Figure 15-16	CTI Channel Out Status Register format .....	15-23
Figure 15-17	ITTRIGINACK Register format .....	15-24
Figure 15-18	ITCHOUT Register format .....	15-25
Figure 15-19	ITTRIGOUT Register format .....	15-26
Figure 15-20	ITTRIGOUTACK Register format .....	15-27
Figure 15-21	ITCHIN Register format .....	15-28
Figure 15-22	ITTRIGIN Register format .....	15-28
Figure 15-23	Mapping between the Component ID Registers and the component ID value .....	15-33
Figure 17-1	Input timing parameters .....	17-2
Figure 17-2	Output timing parameters .....	17-2

# Preface

This preface introduces the *Cortex-A8 Technical Reference Manual (TRM)*. It contains the following sections:

- *About this manual* on page xxviii
- *Feedback* on page xxxiv.

## About this manual

This is the technical reference manual for the Cortex-A8 processor. In this manual the generic term processor means the Cortex-A8 processor.

## Product revision status

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

**rn** Identifies the major revision of the product.

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

## Intended audience

This document is written for hardware and software engineers who want to design or develop products based on the Cortex-A8 processor.

## Using this manual

This manual is organized into the following chapters:

### **Chapter 1** *Introduction*

Read this for an introduction to the processor and descriptions of the major functional blocks.

### **Chapter 2** *Programmers Model*

Read this for a description of the processor registers and programming details.

### **Chapter 3** *System Control Coprocessor*

Read this for a description of the system control coprocessor CP15 registers and programming information.

### **Chapter 4** *Unaligned Data and Mixed-endian Data Support*

Read this for a description of the processor support for unaligned and mixed-endian data accesses. It also describes Advanced *Single Instruction Multiple Data* (SIMD) data access and alignment.

### **Chapter 5** *Program Flow Prediction*

Read this for a description of branch prediction, including guidelines for optimal performance, and how to enable program flow prediction.

**Chapter 6 *Memory Management Unit***

Read this for a description of the *Memory Management Unit* (MMU) and the address translation process, including a list of CP15 registers that control the MMU.

**Chapter 7 *Level 1 Memory System***

Read this for a description of the Level 1 memory system that consists of separate instruction and data caches.

**Chapter 8 *Level 2 Memory System***

Read this for a description of the Level 2 memory system, including the *L2 PreLoad Engine* (PLE).

**Chapter 9 *External Memory Interface***

Read this for a description of the external memory interface including AXI control signals in the processor.

**Chapter 10 *Clock, Reset, and Power Control***

Read this for a description of the clocking modes and the reset signals. This chapter also describes the power control facilities that include the different clock gating levels to control power and skew.

**Chapter 11 *Design for Test***

Read this for a description of the *Design For Test* (DFT) features of the processor.

**Chapter 12 *Debug***

Read this for a description of the debug support.

**Chapter 13 *NEON and VFP Programmers Model***

Read this for an overview of the NEON and *Vector Floating-Point* (VFP) coprocessor and a description of the NEON and VFP registers and programming details.

**Chapter 14 *Embedded Trace Macrocell***

Read this for an overview of the *Embedded Trace Macrocell* (ETM).

**Chapter 15 *Cross Trigger Interface***

Read this for a description of the *Cross Trigger Interface* (CTI).

**Chapter 16 *Instruction Cycle Timing***

Read this for a description of the instruction cycle timing and for details of the interlocks.

### **Chapter 17 AC Characteristics**

Read this for a description of the timing parameters applicable to the processor.

### **Appendix A Signal Descriptions**

Read this for a summary of the processor signals.

### **Appendix B Instruction Mnemonics**

Read this for a list of the *Unified Assembler Language* (UAL) equivalents of the legacy Advanced SIMD data-processing and VFP data-processing assembly language mnemonics used in this manual.

### **Appendix C Revisions**

Read this for a list of the technical changes between released issues of this book.

**Glossary** Read the Glossary for definitions of terms used in this manual.

## **Conventions**

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xxxi
- *Signals* on page xxxi

### **Typographical**

The typographical conventions are:

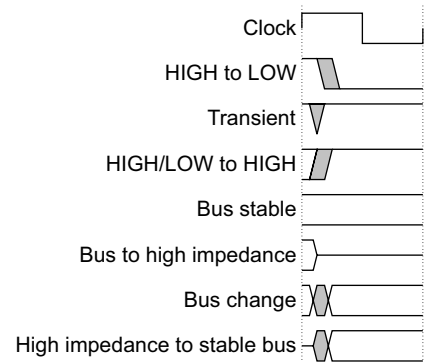
<b><i>italic</i></b>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
<b>bold</b>	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
monospace	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
<u>monospace</u>	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

<i>monospace italic</i>	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
<b>monospace bold</b>	Denotes language keywords when used outside example code.
<b>&lt; and &gt;</b>	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: <ul style="list-style-type: none"> <li>MRC p15, 0 &lt;Rd&gt;, &lt;CRn&gt;, &lt;CRm&gt;, &lt;Opcode_2&gt;</li> <li>The Opcode_2 value selects which register is accessed.</li> </ul>

## Timing diagrams

The figure named *Key to timing diagram conventions* explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

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.



**Key to timing diagram conventions**

## Signals

The signal conventions are:

<b>Signal level</b>	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.
<b>Lower-case n</b>	Denotes an active-LOW signal

<b>Prefix A</b>	Denotes <i>Advanced eXtensible Interface</i> (AXI) global and address channel signals.
<b>Prefix AR</b>	Denotes AXI read address channel signals.
<b>Prefix AW</b>	Denotes AXI write address channel signals.
<b>Prefix B</b>	Denotes AXI write response channel signals.
<b>Prefix C</b>	Denotes AXI low-power interface signals.
<b>Prefix H</b>	Denotes <i>Advanced High-performance Bus</i> (AHB) signals.
<b>Prefix n</b>	Denotes active-LOW signals except in the case of AXI, AHB, or <i>Advanced Peripheral Bus</i> (APB) reset signals.
<b>Prefix P</b>	Denotes APB signals.
<b>Prefix R</b>	Denotes AXI read channel signals.
<b>Prefix W</b>	Denotes AXI write channel signals.
<b>Suffix n</b>	Denotes AXI, AHB, and APB reset signals.

## Additional reading

This section lists publications by ARM and by third parties.

See <http://infocenter.arm.com> for access to ARM documentation.

### ARM publications

This manual contains information that is specific to the processor. See the following documents for other relevant information:

- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *AMBA<sup>®</sup> AHB Specification* (ARM IHI 0011)
- *AMBA AXI Protocol Specification* (ARM IHI 0022)
- *AMBA 3 APB Protocol Specification* (ARM IHI 0024)
- *CoreSight<sup>™</sup> Architecture Specification* (ARM IHI 0029)
- *CoreSight Design Kit Technical Reference Manual* (ARM DDI 0314)
- *CoreSight Design Kit Implementation and Integration Manual* (ARM DII 0092)
- *CoreSight Technology System Design Guide* (ARM DGI 0012)



- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition (ARM DDI 0406)*
- *RealView™ Compilation Tools Developer Guide (ARM DUI 0203)*
- *Cortex-A8 Release Notes (PRDC 007834).*

### **Other publications**

This section list relevant documents published by third parties:

- *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*
- *IEEE Std. 1500-2005, IEEE Standard Testability Method for Embedded Core-based Integrated Circuits.*

## Feedback

ARM welcomes feedback on this product and its documentation.

### Feedback on the product

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

- The product name
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms if appropriate.

### Feedback on this manual

If you have any comments on this manual, send e-mail to [errata@arm.com](mailto:errata@arm.com) giving:

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

ARM also welcomes general suggestions for additions and improvements.

# Chapter 1

## Introduction

This chapter introduces the processor and its features. It contains the following sections:

- *About the processor* on page 1-2
- *ARMv7-A architecture* on page 1-3
- *Components of the processor* on page 1-4
- *External interfaces of the processor* on page 1-8
- *Debug* on page 1-9
- *Power management* on page 1-10
- *Configurable options* on page 1-11
- *Product documentation and architecture* on page 1-12
- *Product revisions* on page 1-14.

## 1.1 About the processor

The Cortex-A8 processor is a high-performance, low-power, cached application processor that provides full virtual memory capabilities. The features of the processor include:

- full implementation of the ARM architecture v7-A instruction set
- configurable 64-bit or 128-bit high-speed *Advanced Microprocessor Bus Architecture* (AMBA) with *Advanced Extensible Interface* (AXI) for main memory interface supporting multiple outstanding transactions
- a pipeline for executing ARM integer instructions
- a NEON pipeline for executing Advanced SIMD and VFP instruction sets
- dynamic branch prediction with branch target address cache, global history buffer, and 8-entry return stack
- *Memory Management Unit* (MMU) and separate instruction and data *Translation Look-aside Buffers* (TLBs) of 32 entries each
- Level 1 instruction and data caches of 16KB or 32KB configurable size
- Level 2 cache of 0KB, 128KB through 1MB configurable size
- Level 2 cache with parity and *Error Correction Code* (ECC) configuration option
- *Embedded Trace Macrocell* (ETM) support for non-invasive debug
- static and dynamic power management including *Intelligent Energy Management* (IEM)
- ARMv7 debug with watchpoint and breakpoint registers and a 32-bit *Advanced Peripheral Bus* (APB) slave interface to a CoreSight debug system.

## 1.2 ARMv7-A architecture

The processor implements ARMv7-A that includes the following features:

- ARM Thumb<sup>®</sup>-2 instruction set for overall code density comparable with Thumb and performance comparable with ARM instructions.
- *Thumb Execution Environment* (ThumbEE) to provide execution environment acceleration.
- Security Extensions architecture for enhanced security features that facilitate the development of secure applications.
- Advanced SIMD architecture extension to accelerate the performance of multimedia applications such as 3-D graphics and image processing.

---

**Note**

- The Advanced SIMD architecture extension, its associated implementations, and supporting software, are commonly referred to as NEON technology.
  - This document uses the older assembler language instruction mnemonics. See Appendix B *Instruction Mnemonics* for information about the *Unified Assembler Language* (UAL) equivalents of the Advanced SIMD instruction mnemonics. See the *ARM Architecture Reference Manual* for more information on the UAL syntax.
- 

- *Vector Floating Point v3* (VFPv3) architecture for floating-point computation that is fully compliant with the IEEE 754 standard.

---

**Note**

This document uses the older assembler language instruction mnemonics. See Appendix B *Instruction Mnemonics* for information about the *Unified Assembler Language* (UAL) equivalents of the VFP data-processing instruction mnemonics. See the *ARM Architecture Reference Manual* for more information on the UAL syntax.

---

See the *ARM Architecture Reference Manual* for more information on the ARMv7-A architecture.

### 1.3 Components of the processor

The main components of the processor are:

- *Instruction fetch* on page 1-5
- *Instruction decode* on page 1-5
- *Instruction execute* on page 1-5
- *Load/store* on page 1-6
- *L2 cache* on page 1-6
- *NEON* on page 1-6
- *ETM* on page 1-6.

Figure 1-1 shows the structure of the Cortex-A8 processor.

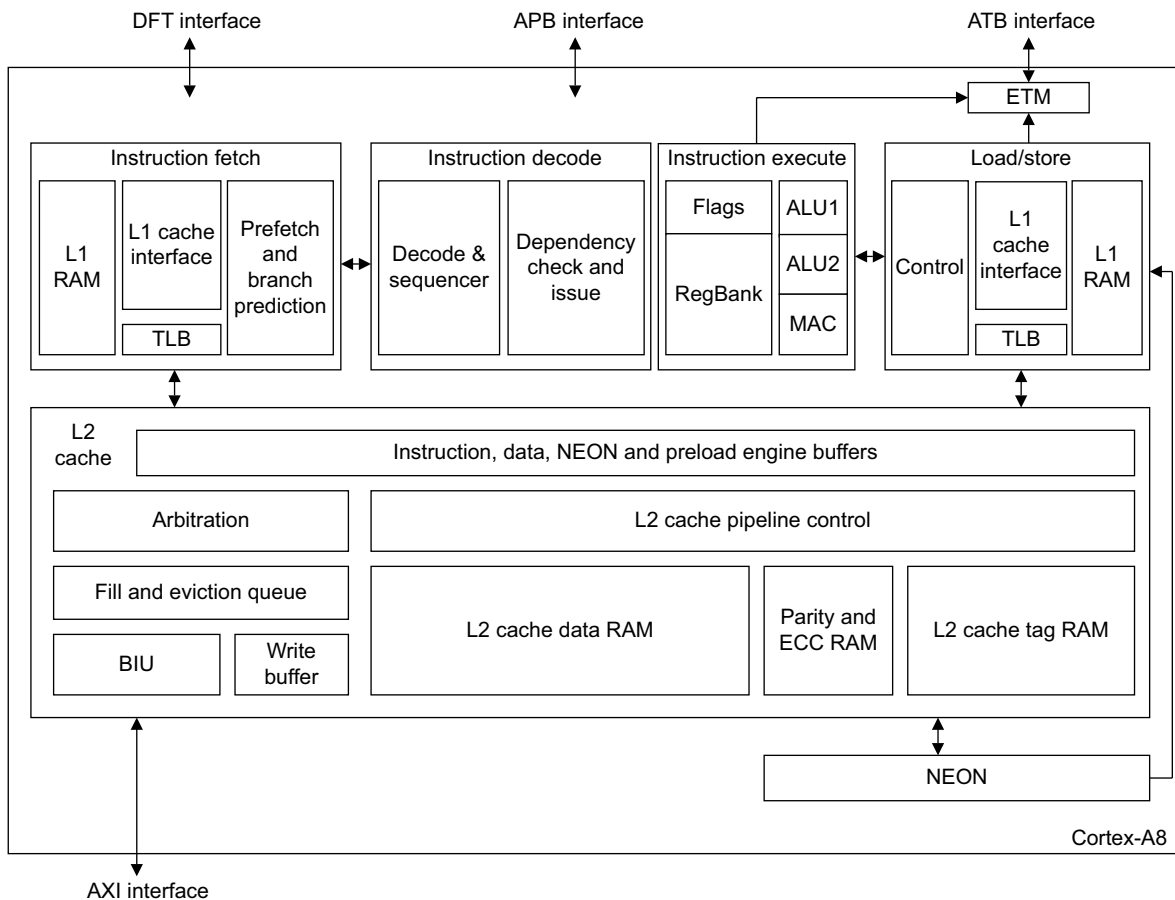


Figure 1-1 Cortex-A8 block diagram

### 1.3.1 Instruction fetch

The instruction fetch unit predicts the instruction stream, fetches instructions from the L1 instruction cache, and places the fetched instructions into a buffer for consumption by the decode pipeline. The instruction fetch unit also includes the L1 instruction cache.

### 1.3.2 Instruction decode

The instruction decode unit decodes and sequences all ARM and Thumb-2 instructions including debug control coprocessor, CP14, instructions and system control coprocessor, CP15, instructions. See Chapter 12 *Debug* for information on the CP14 coprocessor and Chapter 3 *System Control Coprocessor* for information on the CP15 coprocessor.

The instruction decode unit handles the sequencing of:

- exceptions
- debug events
- reset initialization
- *Memory Built-In Self Test* (MBIST)
- wait-for-interrupt
- other unusual events.

See Chapter 16 *Instruction Cycle Timing* for more information on how the processor sequences instructions.

### 1.3.3 Instruction execute

The instruction execute unit consists of two symmetric *Arithmetic Logical Unit* (ALU) pipelines, an address generator for load and store instructions, and the multiply pipeline. The execute pipelines also perform register write back.

The instruction execute unit:

- executes all integer ALU and multiply operations including flag generation
- generates the virtual addresses for loads and stores and the base write-back value, when required
- supplies formatted data for stores and also forwards data and flags
- processes branches and other changes of instruction stream and evaluates instruction condition codes.

### 1.3.4 Load/store

The load/store unit encompasses the entire L1 data side memory system and the integer load/store pipeline. This includes:

- the L1 data cache
- the data side TLB
- the integer store buffer
- the NEON store buffer
- the integer load data alignment and formatting
- the integer store data alignment and formatting.

The pipeline accepts one load or store per cycle that can be present in either pipeline 0 or pipeline 1. This gives the processor flexibility when scheduling load and store instructions. See Chapter 7 *Level 1 Memory System* for more information.

### 1.3.5 L2 cache

The L2 cache unit includes the L2 cache and the *Buffer Interface Unit (BIU)*. It services L1 cache misses from both the instruction fetch unit and the load/store unit. See Chapter 8 *Level 2 Memory System* and Chapter 9 *External Memory Interface* for more information.

### 1.3.6 NEON

The NEON unit includes the full 10-stage NEON pipeline that decodes and executes the Advanced SIMD media instruction set. The NEON unit includes:

- the NEON instruction queue
- the NEON load data queue
- two pipelines of NEON decode logic
- three execution pipelines for Advanced SIMD integer instructions
- two execution pipelines for Advanced SIMD floating-point instructions
- one execution pipeline for Advanced SIMD and VFP load/store instructions
- the VFP engine for full execution of the VFPv3 data-processing instruction set.

See Chapter 13 *NEON and VFP Programmers Model* for programming information on the NEON and VFP coprocessor.

### 1.3.7 ETM

The ETM unit is a non-intrusive trace macrocell that filters and compresses an instruction and data trace for use in system debugging and system profiling.



The ETM unit has an external interface outside of the processor called the *Advanced Trace Bus (ATB)* interface. See Chapter 14 *Embedded Trace Macrocell* for more information.

## 1.4 External interfaces of the processor

The processor has the following external interfaces:

- *AMBA AXI interface*
- *AMBA APB interface*
- *AMBA ATB interface*
- *DFT interface.*

### 1.4.1 AMBA AXI interface

The AXI bus interface is the main interface to the system bus. It performs L2 cache fills and noncacheable accesses for both instructions and data. The AXI interface supports 64-bit or 128-bit wide input and output data buses. It also supports multiple outstanding requests on the AXI bus. The AXI signals are synchronous to the **CLK** input. A wide range of bus clock to core clock ratios is possible through the use of the AXI clock enable signal **ACLKEN**. See the *AMBA AXI Protocol Specification* for more information.

### 1.4.2 AMBA APB interface

The Cortex-A8 processor implements an APB slave interface that enables access to the ETM, CTI, and the debug registers. The APB interface is compatible with the CoreSight architecture which is the ARM architecture for multi-processor trace and debug. See the *CoreSight Architecture Specification* for more information.

### 1.4.3 AMBA ATB interface

The Cortex-A8 processor implements an ATB interface that outputs trace information used for debugging. The ATB interface is compatible with the CoreSight architecture. See the *CoreSight Architecture Specification* for more information.

### 1.4.4 DFT interface

The *Design For Test* (DFT) interface provides support for manufacturing testing of the core using *Memory Built-In Self Test* (MBIST) and *Automatic Test Pattern Generation* (ATPG). See Chapter 11 *Design for Test* for more information.

## 1.5 Debug

The processor implements the ARMv7 Debug architecture that includes support for Security Extensions and CoreSight. To get full access to the processor debug capability, you can access the debug register map through the *Advanced Peripheral Bus* (APB) slave interface. See Chapter 12 *Debug* for more information on the Cortex-A8 implementation of debug. See the *ARM Architecture Reference Manual* for more information on the ARMv7 Debug architecture.

## 1.6 Power management

The processor provides both dynamic and static power control mechanisms. See *Dynamic power management* on page 10-10 for more information. Static power control is implementation-specific. See *Static or leakage power management* on page 10-14 for more information.

## 1.7 Configurable options

Table 1-1 lists the configurable options for the Cortex-A8 processor.

**Table 1-1 Cortex-A8 configurable options**

Feature	Options
AXI bus width	64-bit or 128-bit bus width
L1 RAM	L1 cache size: <ul style="list-style-type: none"> <li>• 16KB</li> <li>• 32KB.</li> </ul>
L2 RAM	L2 cache size: <ul style="list-style-type: none"> <li>• 0KB</li> <li>• 128KB</li> <li>• 256KB</li> <li>• 512KB</li> <li>• 1MB.</li> </ul>
L2 parity/ECC	Yes or No
ETM	Yes or No
NEON	Yes or No
	<p style="text-align: center;"><b>Note</b></p> <p>When you configure the processor without the NEON options, all attempted Advanced SIMD and VFP instructions result in an Undefined Instruction exception.</p>
IEM	Support: <ul style="list-style-type: none"> <li>• all power domains and retention</li> <li>• no power domain or retention</li> <li>• level-shifting only</li> <li>• debug <b>PCLK</b>, <b>ETM CLK</b>, and <b>ETM ATCLK</b> power domain</li> <li>• NEON power domain</li> <li>• L1 data RAMs and L2 RAMs retention</li> <li>• L2 RAMs retention.</li> </ul>

## 1.8 Product documentation and architecture

This section describes the content of the product documents and the relevant architectural standards and protocols.

———— **Note** ————

See *Additional reading* on page xxxii for more information about the documentation described in this section.

### 1.8.1 Documentation

The following books describe the processor:

#### **Technical Reference Manual**

The *Technical Reference Manual* (TRM) describes the processor functionality and the effects of functional options on the behavior of the processor. It is required at all stages of the design flow. Some behavior described in the TRM might not be relevant, because of the way the processor has been implemented and integrated. If you are programming the processor, contact the implementer to determine the build configuration of the implementation, and the integrator to determine the pin configuration of the SoC that you are using.

#### **Configuration and Sign-Off Guide**

The *Configuration and Sign-Off Guide* (CSG) describes:

- the available build configuration options and related issues in selecting them
- how to configure the *Register Transfer Level* (RTL) with the build configuration options
- the processes to sign off the configured RTL and final macrocell.

The ARM product deliverables include reference scripts and information about using them to implement your design. Reference methodology documentation from your EDA tools vendor complements the CSG. The CSG is a confidential book that is only available to licensees.

#### **Methodology Guide**

The Methodology Guide (MG) describes the major requirements and considerations for each step in the implementation flow. This guide does not provide complete details about the tools and commands used or procedural steps for specific implementation flows. The details of the flow depend on the particular *Electronic Design Automation* (EDA) tools,

process technology, and standard cell libraries that the implementation team uses. The MG is a confidential book that is only available to licensees.

## 1.8.2 Architectural information

The Cortex-A8 processor conforms to, or implements, the following specifications:

### ARM Architecture

This describes:

- The behavior and encoding of the instructions that the processor can execute.
- The modes and states that the processor can be in.
- The various data and control registers that the processor must contain.
- The properties of memory accesses.
- The debug architecture you can use to debug the processor. The TRM gives more information about the implemented debug features.

The Cortex-A8 processor implements the ARMv7-A architecture profile.

### Advanced Microcontroller Bus Architecture protocol

*Advanced Microcontroller Bus Architecture* (AMBA) is an open standard, on-chip bus specification that defines the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It facilitates development of embedded processors with multiple peripherals.

### IEEE 754 IEEE Standard for Binary Floating Point Arithmetic.

An architecture specification typically defines a number of versions, and includes features that are either optional or partially specified. The TRM describes which architectures are used, including which version is implemented, and the architectural choices made for the implementation. The TRM does not provide detailed information about the architecture, but some architectural information is included to give an overview of the implementation or, in the case of control registers, to make the manual easier to use. See the appropriate specification for more information about the implemented architectural features.

## 1.9 Product revisions

This section summarizes the differences in functionality between the releases of this processor.

———— **Note** ————

The first released version of the Cortex-A8 processor was r1p0.

### 1.9.1 r1p0-r1p1

The following changes have been made in this release:

- ID Register values changed to reflect product revision status:  
**Main ID Register** 0x411FC081  
**FPSID Register** 0x410330C1
- The L2EN bit of the Auxiliary Control Register is banked between Nonsecure and Secure states.
- **SAFESHIFTRAM** top-level pin added for ATPG test.
- ETM and NEON configurability support added.

### 1.9.2 r1p1-r1p2

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x411FC082

**FPSID Register** 0x410330C1

### 1.9.3 r1p2-r1p3

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x411FC083

**FPSID Register** 0x410330C1

### 1.9.4 r1p3-r1p7

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x411FC087



**FPSID Register** 0x410330C1

### 1.9.5 r1p1-r2p0

The following changes have been made in this release:

- ID Register values changed to reflect product revision status:  
**Main ID Register** 0x412FC080  
**FPSID Register** 0x410330C2
- **CLKSTOPREQ** and **CLKSTOPACK** functionality added to stop and restart the processor clocks without relying on software to execute WFI instruction.
- The **SAFESHIFTRAM** signal is replaced with the **SAFESHIFTRAMIF**, **SAFESHIFTRAMLS**, and **SAFESHIFTRAML2** signals for ATPG test.
- *Intelligent Energy Management (IEM)* multiple power domain support added.
- 1-way and 4-way L2 tag bank removed.
- 64KB and 2MB L2 cache sizes removed.
- **ETMPWRDWNREQ** and **ETMPWRDWNACK** are no longer required because debug and the ETM use the same power domain. **ETMPWRDWNREQ** must be tied to 0. See *Static or leakage power management* on page 10-14 for information on the supported Cortex-A8 power domain configurations.

### 1.9.6 r2p0-r2p1

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x412FC081

**FPSID Register** 0x410330C2

### 1.9.7 r2p1-r2p2

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x412FC082

**FPSID Register** 0x410330C2

### 1.9.8 r2p1-r2p5

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x412FC085

**FPSID Register** 0x410330C2

### 1.9.9 r2p2-r2p3

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x412FC083

**FPSID Register** 0x410330C2

### 1.9.10 r2p2-r2p6

The ID Register values changed to reflect product revision status:

**Main ID Register** 0x412FC086

**FPSID Register** 0x410330C2

### 1.9.11 r2p2-r3p0

The following changes have been made in this release:

- ID Register values changed to reflect product revision status:  
**Main ID Register** 0x413FC080  
**FPSID Register** 0x410330C3
- Improved performance for Cache Maintenance operations.
- Addition of Auxiliary Control Register bit[20] accessible in Secure state only for controlling the performance of Cache Maintenance operations.
- Changed the PLE to perform clean-and-invalidate when DT=1 from clean.

### 1.9.12 r3p0-r3p1

The following changes have been made in this release:

- ID Register values changed to reflect product revision status:  
**Main ID Register** 0x413FC081. See *c0, Main ID Register* on page 3-25.

**FPSID Register** 0x410330C3. See *Floating-Point System ID Register, FPSID* on page 13-13.

- Changed the name for trigger input 0 from **DBGTRIGGER** to Debug entry. This trigger is a pulse asserted on debug state entry. See Table 15-1 on page 15-6 and Table 15-26 on page 15-29.



# Chapter 2

## Programmers Model

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

- *About the programmers model* on page 2-3
- *Thumb-2 instruction set* on page 2-4
- *ThumbEE instruction set* on page 2-6
- *Jazelle Extension* on page 2-10
- *Security Extensions architecture* on page 2-13
- *Advanced SIMD architecture* on page 2-15
- *VFPv3 architecture* on page 2-16
- *Processor operating states* on page 2-17
- *Data types* on page 2-18
- *Memory formats* on page 2-19
- *Addresses in a processor system* on page 2-21
- *Operating modes* on page 2-22
- *Registers* on page 2-23
- *The program status registers* on page 2-27
- *Exceptions* on page 2-35
- *Software consideration for Security Extensions* on page 2-44

- *Hardware consideration for Security Extensions* on page 2-45
- *Control coprocessor* on page 2-48.

## 2.1 About the programmers model

This section describes the processor at the level required to write functional code. It does not include internal microarchitecture details.

The processor implements the ARMv7-A architecture. This includes:

- the 32-bit ARM instruction set
- the 16-bit and 32-bit Thumb-2 instruction set
- the ThumbEE instruction set
- the Security Extensions architecture
- the Advanced SIMD architecture.

See the *ARM Architecture Reference Manual* for more information on the ARMv7-A architecture.

## 2.2 Thumb-2 instruction set

Thumb-2 is an enhancement to the 16-bit Thumb instruction set. It adds 32-bit instructions that can be freely intermixed with 16-bit instructions in a program. The additional 32-bit instructions enable Thumb-2 to cover the functionality of the ARM instruction set. The 32-bit instructions enable Thumb-2 to combine the code density of earlier versions of Thumb, with performance of the ARM instruction.

The most important difference between the Thumb-2 instruction set and the ARM instruction set is that most 32-bit Thumb instructions are unconditional, whereas most ARM instructions can be conditional. Thumb-2 introduces a conditional execution instruction, IT, that is a logical if-then-else function that you can apply to following instructions to make them conditional.

Thumb-2 instructions are accessible as were Thumb instructions when the processor is in Thumb state, that is, the T bit in the CPSR is 1 and the J bit in the CPSR is 0.

In addition to the 32-bit Thumb instructions, there are several 16-bit Thumb instructions and a few 32-bit ARM instructions, introduced as part of the Thumb-2 architecture.

The main enhancements are:

- 32-bit instructions added to the Thumb instruction set to:
  - provide support for exception handling in Thumb state
  - provide access to coprocessors
  - include *Digital Signal Processing (DSP)* and media instructions
  - improve performance in cases where a single 16-bit instruction restricts functions available to the compiler.
- addition of a 16-bit IT instruction that enables one to four following Thumb instructions, the IT block, to be conditional
- addition of a 16-bit *Compare with Zero and Branch (CZB)* instruction to improve code density by replacing two-instruction sequence with a single instruction.

The 32-bit ARM Thumb-2 instructions are added in the space occupied by the Thumb BL and BLX instructions. Figure 2-1 shows the 32-bit ARM Thumb-2 instruction format.

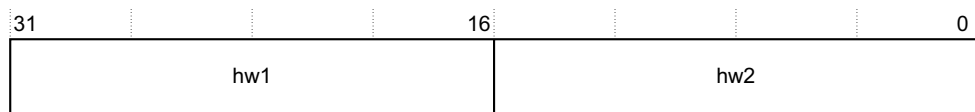


Figure 2-1 32-bit ARM Thumb-2 instruction format



The *first halfword* (hw1) determines the instruction length and functionality. If the processor decodes the instruction as 32-bit long, then the processor fetches the *second halfword* (hw2) of the instruction from the instruction address plus two.

The availability of both 16-bit Thumb and 32-bit instructions in the Thumb-2 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, you can code critical loops for applications such as fast interrupts and DSP algorithms using the 32-bit media instructions in Thumb-2 and use the smaller 16-bit classic Thumb instructions for the rest of the application. This is for code density and does not require any mode change.

See the *ARM Architecture Reference Manual* for information on the ARM, Thumb, and Thumb-2 instruction set.

## 2.3 ThumbEE instruction set

ThumbEE is a variant of the Thumb-2 instruction set. It is designed as a target for dynamically generated code. This is code compiled on the device either shortly before or during execution from a portable bytecode or other intermediate or native representation. It is particularly suited to languages that employ managed pointers or array types. ThumbEE provides increased code density for the compiled binary compared with the compiled code for the ARM or Thumb-2 instruction set. ThumbEE introduces a new processor state, the ThumbEE state, indicated by both the T bit and the J bit in the CPSR Register being set to 1.

See the *ARM Architecture Reference Manual* for information on the ARM, Thumb, and ThumbEE instruction sets.

### 2.3.1 Instructions

In ThumbEE state, the processor uses almost the same instruction set as Thumb-2 although some instructions behave differently, and a few are removed, or added.

The key differences are:

- additional state changing instructions in both Thumb state and ThumbEE state
- new instructions to branch to handlers
- null pointer checking on loads and stores
- an additional instruction in ThumbEE state to check array bounds
- some other modifications to the load, store, and branch instructions.

ThumbEE instructions are accessible when the processor is in ThumbEE state.

### 2.3.2 Configuration

Two registers provide ThumbEE configuration:

- ThumbEE Configuration Register. This contains a single bit, the ThumbEE configuration control bit, XED.
- ThumbEE HandlerBase Register. This contains the base address for ThumbEE handlers.

A handler is a short, commonly executed, sequence of instructions. It is typically, but not always, associated directly with one or more bytecodes or other intermediate language elements.

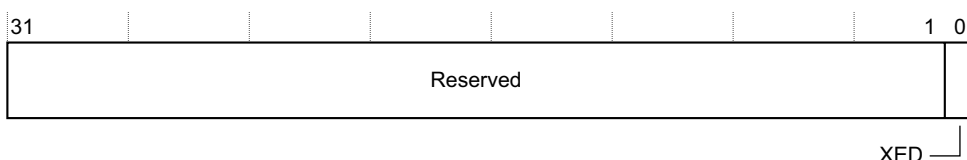
## ThumbEE Configuration Register

The purpose of the ThumbEE Configuration Register is to control access to the ThumbEE HandlerBase Register.

The ThumbEE Configuration Register is:

- in CP14 register c0
- a 32-bit register, with access rights that depend on the current privilege:
  - the result of an unprivileged write to the register is Undefined
  - unprivileged reads, and privileged reads and writes, are permitted.

Figure 2-2 shows the bit arrangement of the ThumbEE Configuration Register.



**Figure 2-2 ThumbEE Configuration Register format**

Table 2-1 shows how the bit values correspond with the ThumbEE Configuration Register.

**Table 2-1 ThumbEE Configuration Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved. <i>Unpredictable (UNP), Should-Be-Zero (SBZ).</i>
[0]	XED	eXecution Environment Disable bit. Controls unprivileged access to the ThumbEE HandlerBase Register: 0 = Unprivileged access permitted. See <i>Access to ThumbEE registers</i> on page 2-9 for details. 1 = Unprivileged access disabled. The reset value of this bit is 0.

Any change to this register is only guaranteed to be visible to subsequent instructions after the execution of an ISB instruction. However, a read of this register always returns the last value written to the register.

To access the ThumbEE Configuration Register, read or write CP14 with:

MRC p14, 6, <Rd>, c0, c0, 0 ; Read ThumbEE Configuration Register  
MCR p14, 6, <Rd>, c0, c0, 0 ; Write ThumbEE Configuration Register

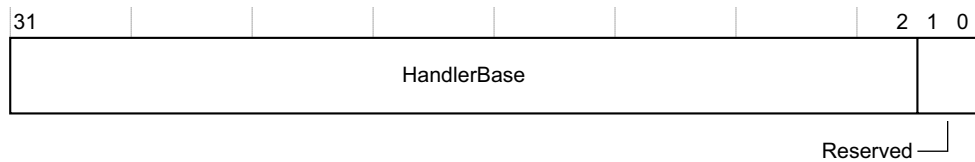
## ThumbEE HandlerBase Register

The purpose of the ThumbEE HandlerBase Register is to hold the base address for ThumbEE handlers.

The ThumbEE HandlerBase Register is:

- in CP14 register c0
- a 32-bit read/write register, with unprivileged access that depends on the value of the ThumbEE Configuration Register. See *Access to ThumbEE registers* on page 2-9.

Figure 2-3 shows the bit arrangement of the ThumbEE HandlerBase Register.



**Figure 2-3 ThumbEE HandlerBase Register format**

Table 2-2 shows how the bit values correspond with the ThumbEE HandlerBase Register.

**Table 2-2 ThumbEE HandlerBase Register bit functions**

Bits	Field	Function
[31:2]	HandlerBase	The address of the ThumbEE Handler_00 implementation. This is the address of the first of the ThumbEE handlers. The reset value of this field is Unpredictable.
[1:0]	-	Reserved. UNP, SBZ.

Any change to this register is only guaranteed to be visible to subsequent instructions after the execution of an ISB instruction. However, a read of this register always returns the last value written to the register.

To access the ThumbEE HandlerBase Register, read or write CP14 with:

MRC p14, 6, <Rd>, c1, c0, 0 ; Read ThumbEE HandlerBase Register  
MCR p14, 6, <Rd>, c1, c0, 0 ; Write ThumbEE HandlerBase Register

## Access to ThumbEE registers

Table 2-3 shows the access permissions for the ThumbEE registers, and how unprivileged access to the ThumbEE HandlerBase Register depends on the value of the ThumbEE Configuration Register.

**Table 2-3 Access to ThumbEE registers**

Register	Unprivileged access		Privileged access
	XED == 0 <sup>a</sup>	XED == 1 <sup>a</sup>	
ThumbEE Configuration	Read access permitted, write access Undefined	Read access permitted, write access Undefined	Read and write access permitted
ThumbEE HandlerBase	Read and write access permitted	Read and write access Undefined	Read and write access permitted

a. Value of XED bit in the ThumbEE Configuration Register, see *ThumbEE Configuration Register* on page 2-7.

## 2.4 Jazelle Extension

The Cortex-A8 processor provides a trivial implementation of the Jazelle Extension. This means that the processor does not accelerate the execution of any bytecodes, and all bytecodes are executed by software routines.

In the implementation of the Jazelle Extension:

- Jazelle state is not supported
- The BXJ instruction behaves as a BX instruction.

See the *ARM Architecture Reference Manual* for information on Jazelle Extension.

The processor provides three registers for the implementation of the Jazelle Extension:

- the *Jazelle Identity Register*
- the *Jazelle Main Configuration Register* on page 2-11
- the *Jazelle OS Control Register* on page 2-11.

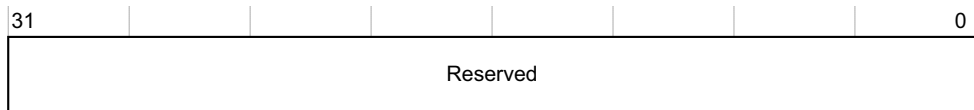
### 2.4.1 Jazelle Identity Register

The Jazelle Identity Register enables software to determine the implementation of the Jazelle Extension provided by the processor.

The Jazelle Identity Register is:

- in CP14 register c0
- a 32-bit read-only register, accessible in all processor modes and security states.

Figure 2-4 shows the bit arrangement of the Jazelle Identity Register.



**Figure 2-4 Jazelle Identity Register format**

Table 2-4 shows how the bit values correspond with the Jazelle Identity Register.

**Table 2-4 Jazelle Identity Register bit functions**

Bits	Field	Function
[31:0]	-	<i>Read-As-Zero (RAZ)</i>

To access this register, read CP14 with:

MRC p14, 7, <Rd>, c0, c0, 0 ; Read Jazelle Identity Register

## 2.4.2 Jazelle Main Configuration Register

The Jazelle Main Configuration Register controls features of the Jazelle Extension.

The Jazelle Main Configuration Register is:

- in CP14 register c0
- a 32-bit register, with access rights that depend on the current privilege:
  - Write-only (WO) in User mode
  - Read/Write (R/W) in Privileged modes.

Figure 2-5 shows the bit arrangement of the Jazelle Main Configuration Register.



**Figure 2-5 Jazelle Main Configuration Register format**

Table 2-5 shows how the bit values correspond with the Jazelle Main Configuration Register.

**Table 2-5 Jazelle Main Configuration Register bit functions**

Bits	Field	Function
[31:0]	-	RAZ

To access this register, read or write CP14 with:

MRC p14, 7, <Rd>, c2, c0, 0 ; Read Main Configuration Register  
 MCR p14, 7, <Rd>, c2, c0, 0 ; Write Main Configuration Register

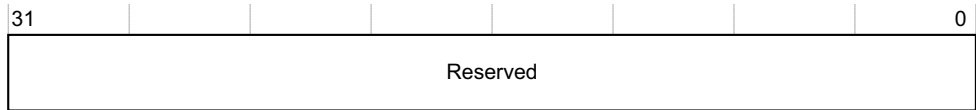
## 2.4.3 Jazelle OS Control Register

The Jazelle OS Control Register enables operating systems to control access to Jazelle Extension hardware.

The Jazelle OS Control Register is:

- in CP14 register c0
- a 32-bit register, with access rights that depend on the current privilege:
  - the result of an access in User mode is an Undefined Instruction exception
  - the register is Read/Write (R/W) in Privileged modes.

Figure 2-6 on page 2-12 shows the bit arrangement of the Jazelle OS Control Register.



**Figure 2-6 Jazelle OS Control Register format**

Table 2-6 shows how the bit values correspond with the Jazelle OS Control Register.

**Table 2-6 Jazelle OS Control Register bit functions**

Bits	Field	Function
[31:0]	-	RAZ

To access this register, read or write CP14 with:

```
MRC p14, 7, <Rd>, c1, c0, 0 ; Read OS Control Register
MCR p14, 7, <Rd>, c1, c0, 0 ; Write OS Control Register
```



## 2.5 Security Extensions architecture

The processor implements the TrustZone Security Extensions architecture to facilitate the development of secure applications.

Security Extensions are based on these fundamental principles:

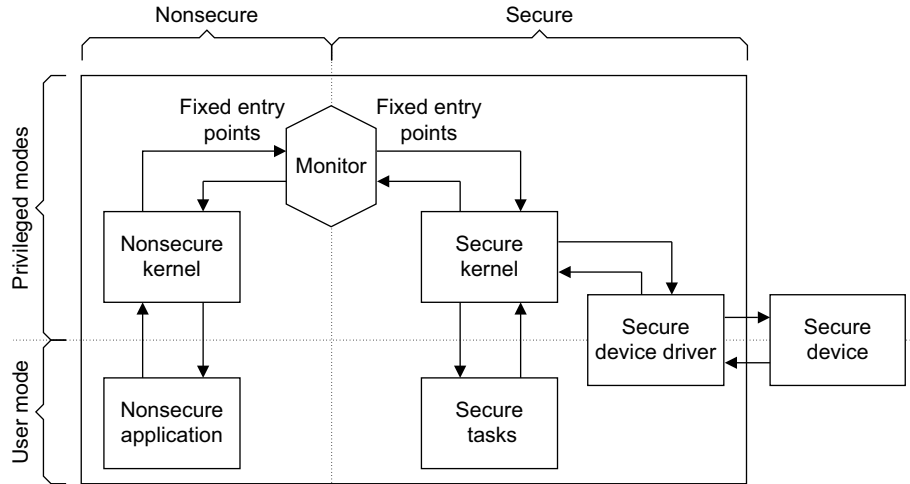
- The extensions define a class of core operation that you can switch between Secure and Nonsecure state. Most code runs in Nonsecure state. Only trusted code runs in Secure state.
- The extensions define some memory as secure memory. When the core is in Secure state, it can access secure memory.
- Entry into Secure state is strictly controlled.
- Exit from Secure state can only occur at programmed points.
- Debug is strictly controlled.
- The processor enters Secure state on reset.

Exceptions are generally handled in a similar way to other ARM architectures. Support is available for some exceptions handled only by code running in Secure state.

See the *ARM Architecture Reference Manual* for information on the Security Extensions.

### 2.5.1 Security Extensions model

The basis of the Security Extensions model is that the computing environment splits into two isolated states, the Secure state and the Nonsecure state, with no leakage of secure data to the Nonsecure state. Software Secure Monitor code, running in the Monitor mode, links the two states and acts as a gatekeeper to manage program flow. The system can have both secure and nonsecure peripherals that is suitable to secure and nonsecure device drivers control. Figure 2-7 on page 2-14 shows the relationship between the Secure and Nonsecure states. The *Operating System* (OS) splits into the secure OS, that includes the secure kernel, and the nonsecure OS, that includes the nonsecure kernel. For details on modes of operation, see *Operating modes* on page 2-22.



**Figure 2-7 Secure and Nonsecure states**

In normal nonsecure operation, the OS runs tasks in the usual way. When a User process requires secure execution it makes a request to the secure kernel, that operates in privileged mode. This then calls the Secure Monitor to transfer execution to the Secure state.

This approach to secure systems means that the platform OS that works in the Nonsecure state, has only a few fixed entry points into the Secure state through the Secure Monitor. The trusted code base for the Secure state, that includes the secure kernel and secure device drivers, is small and therefore much easier to maintain and verify.

See *Software consideration for Security Extensions* on page 2-44 and *Hardware consideration for Security Extensions* on page 2-45 for more details.

## 2.6 Advanced SIMD architecture

Advanced SIMD architecture is a media and signal processing architecture that adds instructions targeted primarily at audio, video, 3-D graphics, image, and speech processing. Advanced SIMD instructions are available in both ARM and Thumb states.

The NEON coprocessor provides a register bank that is distinct from the ARM integer core register bank. Both the Advanced SIMD instructions and the VFP instructions use this register bank.

The Advanced SIMD instructions perform packed SIMD operations. These operations process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel. Instructions operate on vectors held in 64-bit or 128-bit registers.

The elements can be:

- 32-bit single-precision floating-point numbers
- 8-bit, 16-bit, 32-bit, or 64-bit signed or unsigned integers
- 8-bit, 16-bit, 32-bit, or 64-bit bitfields
- 8-bit or 16-bit polynomials with 1-bit coefficients.

See the *ARM Architecture Reference Manual* for information on the Advanced SIMD architecture.

## 2.7 VFPv3 architecture

The VFP architecture v3 is an enhancement to the VFP architecture v2. The main changes are:

- the doubling of the number of double-precision registers to 32
- the introduction of an instruction that places a floating-point constant in a register, and instructions that perform conversions between fixed-point and floating-point numbers
- the introduction of VFP architecture v3 variant, that does not trap floating-point exceptions.

VFPv3 is backward compatible with VFPv2 except for the capability of trapping floating-point exceptions.

See the *ARM Architecture Reference Manual* for information on the VFPv3 architecture.

## 2.8 Processor operating states

The processor has the following operating states controlled by the T bit and J bit in the CPSR.

<b>ARM state</b>	32-bit, word-aligned ARM instructions are executed in this state. T bit is 0 and J bit is 0.
<b>Thumb state</b>	16-bit and 32-bit, halfword-aligned Thumb-2 instructions. T bit is 1 and J bit is 0.
<b>ThumbEE state</b>	16-bit and 32-bit, halfword-aligned variant of the Thumb-2 instruction set designed as a target for dynamically generated code. This is code compiled on the device either shortly before or during execution from a portable bytecode or other intermediate or native representation. T bit is 1 and J bit is 1.

---

**Note**

- The processor does not support Jazelle state. This means there is no processor state where the T bit is 0 and J bit is 1.
  - Transition between ARM and Thumb states does not affect the processor mode or the register contents. See the *ARM Architecture Reference Manual* for information on entering and exiting ThumbEE state.
- 

### 2.8.1 Switching state

You can switch the operating state of the 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*.
- Thumb state and ThumbEE state using the ENTERX and LEAVEX instructions.

Exceptions cause the processor to enter ARM or Thumb state according to the value held in the TE bit within the system control coprocessor. Normally, on exiting an exception handler, the processor restores the original contents of the T and J bits.

### 2.8.2 Interworking ARM and Thumb state

The processor enables you to mix ARM Thumb-2 code. See Chapter 4 *Interworking ARM and Thumb* in the *RealView Compilation Tools Developer Guide* for details.

## 2.9 Data types

The processor supports the following data types:

- doubleword, 64-bit
- 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 align with 4-byte boundaries
- halfword quantities must align with 2-byte boundaries
- byte quantities can be placed on any byte boundary.

The processor provides mixed-endian and unaligned access support. See Chapter 4 *Unaligned Data and Mixed-endian Data Support* for details.

---

**Note**

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

---

## 2.10 Memory formats

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

The processor can treat words in memory as either:

- *Byte-invariant big-endian format*
- *Little-endian format.*

Additionally, the processor supports mixed-endian and unaligned data accesses. See Chapter 4 *Unaligned Data and Mixed-endian Data Support* for details.

———— **Note** —————

Instructions are always treated as little-endian.

### 2.10.1 Byte-invariant big-endian format

In byte-invariant big-endian format, the 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 as Figure 2-8 shows.

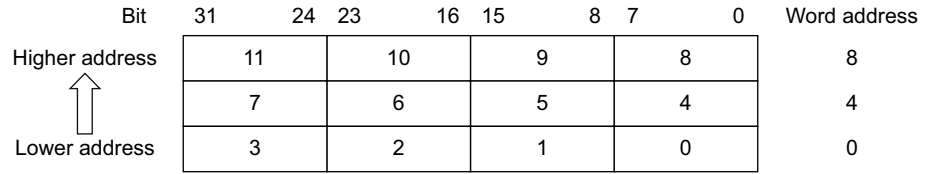
Bit	31	24	23	16	15	8	7	0	Word address
Higher address	8	9	10	11					8
↑	4	5	6	7					4
Lower address	0	1	2	3					0

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

**Figure 2-8 Big-endian addresses of bytes within words**

### 2.10.2 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 2-9 on page 2-20.



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

**Figure 2-9 Little-endian addresses of bytes within words**



## 2.11 Addresses in a processor system

Three distinct types of address exist in the processor system:

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

When the core is in the Secure or Nonsecure state, the VA is Secure or Nonsecure respectively. To get the VA to PA translation, the core uses secure translation tables while it is in Secure state. Otherwise it uses the nonsecure translation tables.

Table 2-7 shows the address types in the processor system.

**Table 2-7 Address types in the processor system**

Processor	Caches	TLBs	AXI bus
Virtual Address	Virtual index physical tag <sup>a</sup>	Translates Virtual Address to Physical Address	Physical Address <sup>b</sup>

a. L1 cache is virtual index physical tag.

b. L2 cache is physical address physical tag.

This is an example of the address manipulation that occurs when the processor requests an instruction.

1. The processor issues the VA of the instruction as the Secure or Nonsecure VA according to the state of the processor.
2. The lower bits of the VA indexes the instruction cache. The VA is translated using the Secure or Nonsecure Process ID, CP15 c13, to the MVA, and then to PA in the *Translation Lookaside Buffer (TLB)*. The TLB performs the translation in parallel with the cache lookup. The translation uses secure descriptors if the core is in the Secure state. Otherwise it uses the nonsecure ones.
3. If the TLB performs a successful protection check on the MVA, and the PA tag is in the instruction cache, the instruction data is returned to the processor. For information on unsuccessful protection checks, see *Aborts* on page 2-37.
4. The PA is passed to the L2 cache. If the L2 cache contains the physical address of the requested instruction, the L2 cache supplies the instruction data.
5. The PA is passed to the AXI bus interface to perform an external access, in the event of a cache miss. The external access is always Nonsecure when the core is in the Nonsecure state. In the Secure state, the external access is Secure or Nonsecure according to the NS attribute value in the selected descriptor.

## 2.12 Operating modes

There are eight 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 OS
- Abort mode is entered after a data abort or prefetch abort
- System mode is a privileged user mode for the OS
- Undefined mode is entered when an Undefined Instruction exception occurs
- Monitor mode is a Secure mode for the Security Extensions Secure Monitor code.

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. Table 2-8 shows the mode structure for the processor.

**Table 2-8 Mode structure**

Modes	Mode type	Security state of core	
		NS bit = 1	NS bit = 0
User	User	Nonsecure	Secure
FIQ	Privileged	Nonsecure	Secure
IRQ	Privileged	Nonsecure	Secure
Supervisor	Privileged	Nonsecure	Secure
Abort	Privileged	Nonsecure	Secure
Undefined	Privileged	Nonsecure	Secure
System	Privileged	Nonsecure	Secure
Monitor	Privileged	Secure	Secure

## 2.13 Registers

The processor has a total of 40 registers:

- 33 general-purpose 32-bit registers
- seven 32-bit status registers.

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

### 2.13.1 The state register set

In ARM state, 16 data registers and one or two status registers are accessible at any time. In privileged modes, mode-specific banked registers become available. Figure 2-10 on page 2-25 shows which registers are available in each mode.

Thumb and ThumbEE state give access to the same set of registers as ARM state. However, the 16-bit instructions provide only limited access to some of the registers. No such limitations exist for 32-bit Thumb-2 and ThumbEE instructions.

Registers r0 through r13 are general-purpose registers used to hold either data or address values.

Registers r14 and r15 have the following special functions:

- Link Register** Register r14 is used as the subroutine *Link Register* (LR). Register r14 receives the return address when the processor executes a *Branch with Link* (BL or BLX) instruction. You can treat r14 as a general-purpose register at all other times. Similarly, the corresponding banked registers r14\_mon, r14\_svc, r14\_irq, r14\_fiq, r14\_abt, and r14\_und hold the return values when the processor receives interrupts and exceptions, or when it executes the BL or BLX instructions 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 ThumbEE state, this is halfword-aligned.

One of the status registers, the *Current Program Status Register* (CPSR), contains condition code flags, status bits, and current mode bits.

In privileged modes, another register, one of the *Saved Program Status Registers* (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. Typically, this is used when returning after handling an exception.

Banked registers have a mode identifier that indicates the mode that they relate to. Table 2-9 shows these mode identifiers.

**Table 2-9 Register mode identifiers**

Mode	Mode identifier
User	usr
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usr
Undefined	und
Monitor	mon
















The `usr` mode 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`, that is, `r8_fiq` through `r14_fiq`. As a result many FIQ handlers do not have to save any registers.







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

Figure 2-10 on page 2-25 shows the ARM state registers.

### ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined	Secure monitor
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	 r8_fiq	r8	r8	r8	r8	r8
r9	 r9_fiq	r9	r9	r9	r9	r9
r10	 r10_fiq	r10	r10	r10	r10	r10
r11	 r11_fiq	r11	r11	r11	r11	r11
r12	 r12_fiq	r12	r12	r12	r12	r12
r13	 r13_fiq	r13_svc	 r13_abt	 r13_irq	 r13_und	 r13_mon
r14	 r14_fiq	r14_svc	 r14_abt	 r14_irq	 r14_und	 r14_mon
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

### ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und	 SPSR_mon

 = banked register

**Figure 2-10 Register organization in ARM state**

Figure 2-11 on page 2-26 shows an alternative view of the ARM registers.

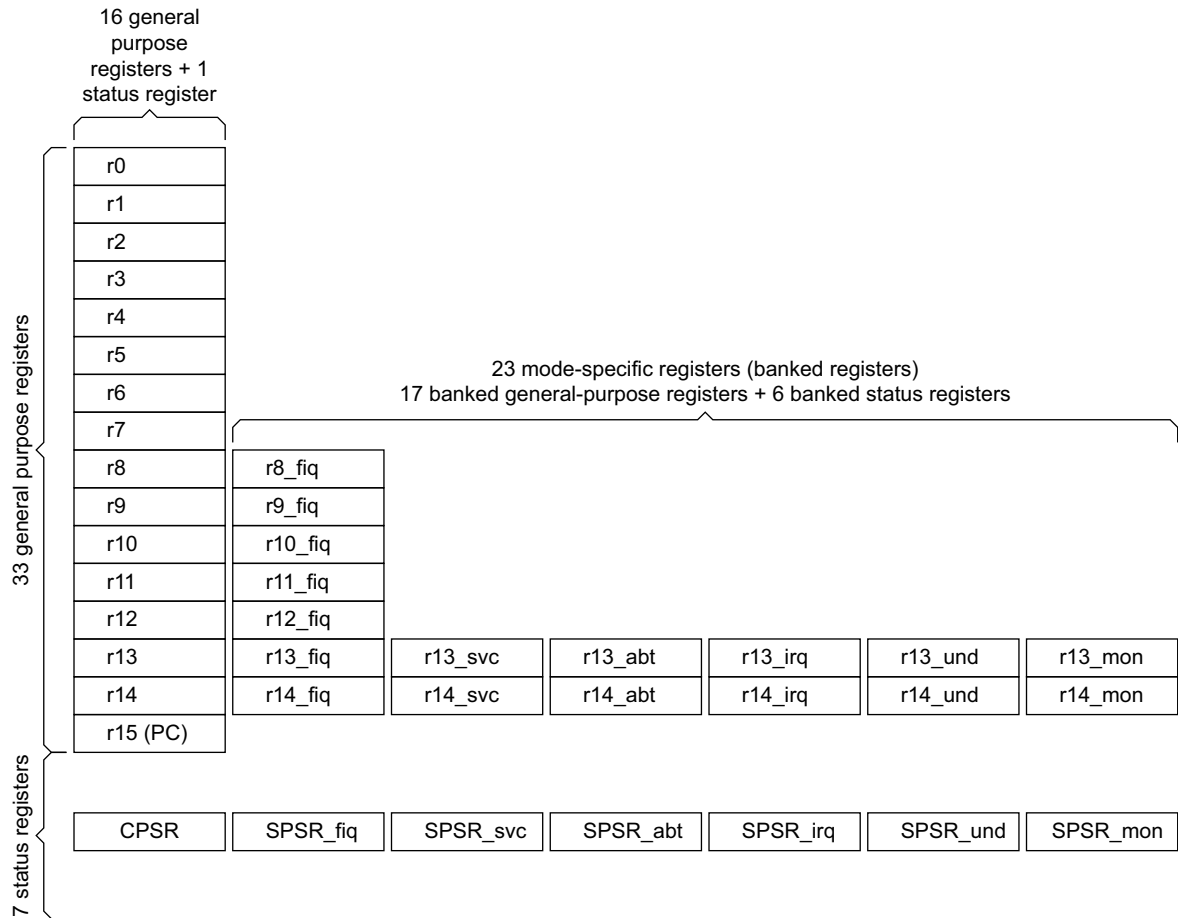


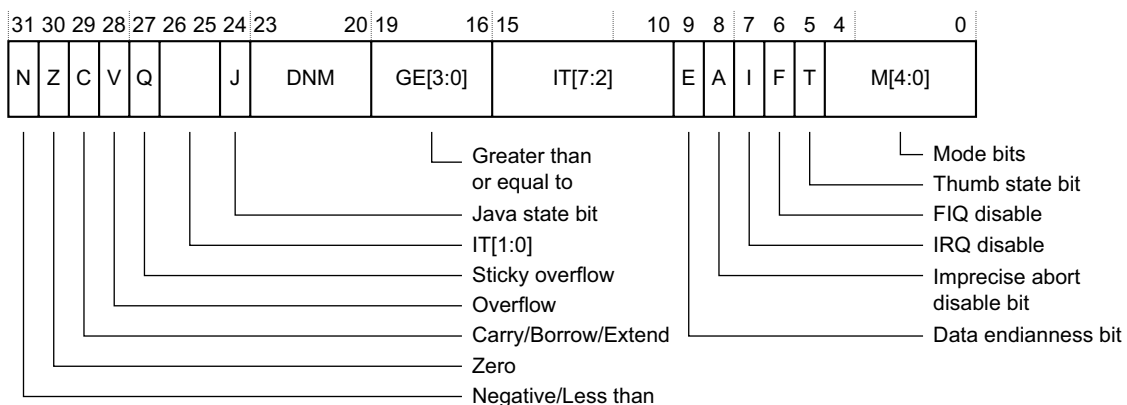
Figure 2-11 Processor register set showing banked registers

## 2.14 The program status registers

The processor contains one CPSR, and six SPSRs for exception handlers to use. The program status registers:

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

Figure 2-12 shows the bit arrangements of the program status registers.



**Figure 2-12 Program status register**

**Note**

The bits identified in Figure 2-12 as *Do Not Modify* (DNM) 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 you change the CPSR.

### 2.14.1 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 processor tests these flags to determine whether to execute an instruction.

In ARM state, you can execute most instructions conditionally on the state of the N, Z, C, and V bits. In Thumb state, you can execute fewer instructions conditionally. However, you can make most instructions conditional with the IT instruction.

See the *ARM Architecture Reference Manual* for more information about conditional executions.

### 2.14.2 The Q flag

You can set the Sticky Overflow, Q flag, to 1 by executing 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 to 1 by an instruction, it remains set until explicitly cleared to 0 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. See the individual instruction definitions in the *ARM Architecture Reference Manual* for details of how you can set and clear the Q flag.

### 2.14.3 The IT execution state bits

IT[7:5] encodes the base condition code for the current IT block, if any. It contains b000 when no IT block is active.

IT[4:0] encodes the number of instructions that are to be conditionally executed, and whether the condition for each is the base condition code or the inverse of the base condition code. It contains b00000 when no IT block is active.



When the processor executes an IT instruction, it sets these bits according to the condition in the instruction, and the *Then* and *Else* (T and E) parameters in the instruction. During execution of an IT block, IT[4:0] is shifted:

- to reduce the number of instructions to be conditionally executed by one
- to move the next bit into position to form the least significant bit of the condition code.

See the *ARM Architecture Reference Manual* for more information on the operation of the IT execution state bits.

#### 2.14.4 The J bit

The J bit in the CPSR indicates when the processor is in ThumbEE state.

When T=1:

**J = 0**            The processor is in Thumb state.

**J = 1**            The processor is in ThumbEE state.

———— **Note** —————

- You cannot set the J bit to 1 when the T bit is 0. The J bit is written as 0 when the T bit is written as 0.
- You cannot use MSR 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.

## 2.14.5 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 Table 2-10 shows.

**Table 2-10 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
<b>Signed</b>				
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	$[31:24] - [31:24] \geq 0$	$[23:16] - [23:16] \geq 0$	$[15:8] - [15:8] \geq 0$	$[7:0] - [7:0] \geq 0$
<b>Unsigned</b>				
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  $A \text{ op } B \geq C$ , 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 usual ARM rules determine the GE bits 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.
- 

### 2.14.6 The E bit

ARM and Thumb instructions are provided to set and clear the E bit. The E bit controls load/store endianness. The E bit can be initialized at reset using the **CFGEND0** input. See Chapter 4 *Unaligned Data and Mixed-endian Data Support* for details of the E bit. See *Miscellaneous signals* on page A-10 for details on the **CFGEND0** signal.

### 2.14.7 The A bit

The A bit is set to 1 automatically. It is used to disable imprecise data aborts. It might not be writable in the Nonsecure state if the AW bit in the SCR register is reset.

### 2.14.8 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* on page 2-32
- *Mode bits* on page 2-32.

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 to 1, IRQ interrupts are disabled.
- When the F bit is set to 1, FIQ interrupts are disabled. FIQ can be nonmaskable in the Nonsecure state if the FW bit in SCR register is reset.

---

#### Note

---

You can change the SPSR F bit in the Nonsecure state but this does not update the CPSR if the SCR bit [4] FW does not permit it.

---

## T bit

The T bit reflects the operating state:

- when the T bit is set to 1, the processor is executing in Thumb state or ThumbEE state depending on the J bit
- when the T bit is cleared to 0, the processor is executing in ARM state.

———— **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 processor, this bit is not affected.

## Mode bits

M[4:0] are the mode bits. These bits determine the processor operating mode as Table 2-11 shows.

**Table 2-11 PSR mode bit values**

M[4:0]	Mode	Visible state registers	
		Thumb	ARM
b10000	User	r0–r7, r8–r12 <sup>a</sup> , SP, LR, PC, CPSR	r0–r14, PC, CPSR
b10001	FIQ	r0–r7, r8_fiq–r12_fiq <sup>a</sup> , SP_fiq, LR_fiq PC, CPSR, SPSR_fiq	r0–r7, r8_fiq–r14_fiq, PC, CPSR, SPSR_fiq
b10010	IRQ	r0–r7, r8–r12 <sup>a</sup> , SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0–r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
b10011	Supervisor	r0–r7, r8–r12 <sup>a</sup> , SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0–r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc
b10111	Abort	r0–r7, r8–r12 <sup>a</sup> , SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0–r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
b11011	Undefined	r0–r7, r8–r12 <sup>a</sup> , SP_und, LR_und, PC, CPSR, SPSR_und	r0–r12, r13_und, r14_und, PC, CPSR, SPSR_und
b11111	System	r0–r7, r8–r12 <sup>a</sup> , SP, LR, PC, CPSR	r0–r14, PC, CPSR
b10110	Secure Monitor	r0–r7, r8–r12 <sup>a</sup> , SP_mon, LR_mon, PC, CPSR, SPSR_mon	r0–r12, PC, CPSR, SPSR_mon, r13_mon, r14_mon

- a. In Thumb state, access to these registers is limited.

### 2.14.9 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 ARMv6 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 2-12 on page 2-27 that are in this category are:

- N
- Z
- C
- V
- Q
- GE[3:0]
- 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 processor these bits are not affected.

Bits in Figure 2-12 on page 2-27 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 2-35.

Bits in Figure 2-12 on page 2-27 that are in this category are:

- A
- I
- F
- M[4:0].

Only secure privileged modes can write directly to the CPSR mode bits to enter Monitor mode. If the core is in secure User mode, nonsecure User mode, or nonsecure privileged modes it ignores changes to the CPSR to enter the Secure Monitor. The core does not copy mode bits in the SPSR that are changed in the Nonsecure state, across to the CPSR.

#### **2.14.10 Reserved bits**

The remaining bits in the PSRs are unused and reserved. When changing a PSR flag or control bits, make sure that you do not alter these reserved bits. 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.

## 2.15 Exceptions

Exceptions occur whenever the processor temporarily halts the normal flow of a program, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the processor preserves the current processor state so the original program can resume when the handler routine finishes.

If two or more exceptions occur simultaneously, the processor deals with exceptions in the fixed order given in *Exception priorities* on page 2-42.

This section provides details of the processor exception handling:

- *Exception entry and exit summary*
- *Leaving an exception* on page 2-36.

### 2.15.1 Exception entry and exit summary

Table 2-12 summarizes the PC value preserved in the relevant r14 on exception entry and the recommended instruction for exiting the exception handler.

**Table 2-12 Exception entry and exit**

Exception or entry	Return instruction	Previous state		Notes
		ARM r14_x	Thumb r14_x	
SVC	MOVS PC, R14_svc	PC + 4	PC+2	Where the PC is the address of the SVC, SMC, or Undefined instruction
SMC	MOVS PC, R14_mon	PC + 4	-	
UNDEF	MOVS PC, R14_und	PC + 4	PC+2	
PABT	SUBS PC, R14_abt, #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	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC+4	
DABT	SUBS PC, R14_abt, #8	PC + 8	PC+8	Where the PC is the address of the load or store instruction that generated the data abort
RESET	-	-	-	The value saved in r14_svc on reset is Unpredictable
BKPT	SUBS PC, R14_abt, #4	PC + 4	PC+4	Software breakpoint

### 2.15.2 Leaving an exception

When an exception has completed, the exception handler must move the LR, minus an offset to the PC. The offset varies according to the type of exception, as Table 2-12 on page 2-35 shows.

Typically the return instruction is an arithmetic or logical operation with the S bit set to 1 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.

---

### 2.15.3 Reset

When the reset signals, as described in Chapter 10 *Clock, Reset, and Power Control*, are driven appropriately a reset occurs, and the processor abandons the executing instruction.

When the reset signals are deasserted, the processor:

1. Forces the NS bit in SCR to 0 for secure and CPSR M[4:0] to 5'b10011 for secure Supervisor mode.
2. Sets the A, I, and F bits in the CPSR.
3. Clears the CPSR J bit. The CPSR T bit is set based on the state of the **CFGTE** input. Other bits in the CPSR are indeterminate.
4. Forces the PC to fetch the next instruction from the reset vector address.
5. Resumes execution in ARM or Thumb state based on the state of the **CFGTE** input.

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

### 2.15.4 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. This minimizes 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 processor. It is the output of this register that the processor control logic uses.



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 cleared to 0, the processor checks for a LOW level on the output of the nFIQ register at the end of each instruction.

The FW bit and FIQ bit in the SCR register configure the FIQ as:

- nonmaskable in Nonsecure state (FW bit in SCR)
- branch to either current FIQ mode or Monitor mode (FIQ bit in SCR).

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.

### 2.15.5 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 cleared to 0, the 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.

The IRQ bit in the SCR register configures the IRQ to branch to either the current IRQ mode or to the Monitor mode.

### 2.15.6 Aborts

An abort is an exception that indicates to the operating system that the value associated with a memory access is invalid. Attempting to access invalid instruction or data memory typically causes an abort.

An abort is either:

- an internal abort signaled by the MMU
- an internal abort signaled by an error condition in the L1 or L2 cache
- an external abort signaled by the AXI interface because of an AXI error response.

An internal or external abort is either:

- a prefetch abort
- a data abort.

In addition, aborts can be precise or imprecise. A precise abort occurs on the instruction associated with the access that triggers the abort exception. An imprecise abort can occur on an instruction subsequent to the instruction associated with the access that triggers the abort exception.

———— **Note** —————

All aborts from the TLB are internal except for aborts from translation table walks that are external precise aborts. If the EA bit is 1 for translation aborts, the core branches to Monitor mode in the same way as it does for all other external aborts. See *c1, Secure Configuration Register* on page 3-70.

IRQs are disabled when an abort occurs. When the aborts are configured to branch to Monitor mode, the FIQ is also disabled.

### **Prefetch abort**

A prefetch abort is associated with an instruction fetch as opposed to a data access.

When a prefetch abort occurs, the processor marks the prefetched instruction as invalid, but does not take the exception until it executes the instruction. If the processor does not execute the instruction, 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**

A data abort is associated with a data access as opposed to an instruction fetch.

Data aborts on the processor can be precise or imprecise.

Internal precise data aborts are those generated by data load or store accesses that the MMU checks:

- alignment faults
- translation faults

- access bit faults
- domain faults
- permission faults.

---

**Note**

---

Instruction memory system operations performed with the system control coprocessors can also generate internal precise data aborts.

---

Externally generated data aborts can be precise or imprecise. Two separate FSR encodings indicate if the external abort is precise or imprecise:

- all external aborts to loads or stores to strongly ordered memory are precise
- all external aborts to loads to the Program Counter or the CSPR are precise
- all external aborts on the load part of a SWP are precise
- all other external aborts are imprecise.

External aborts are supported on cacheable locations. The abort is transmitted to the processor only if the processor requests a word that had an external abort.

**Precise data aborts**

The state of the system presented to the abort exception handler for a precise abort is always the state for the instruction that caused the abort. It cannot be the state for a subsequent instruction. As a result, it is straightforward to restart the processor after the exception handler has rectified the cause of the abort.

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

With the base restored Data Abort model, when a data abort exception occurs during the execution of a memory access instruction, the processor hardware always restores the base register 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, that the aborted instruction might have specified. This simplifies the software Data Abort handler. See the *ARM Architecture Reference Manual* for more information.

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***

The state of the system presented to the abort exception handler for an imprecise data abort can be the state for an instruction after the instruction that caused the abort. As a result, it is not often possible to restart the processor from the point at which the exception occurred.

Data aborts that occur because of watchpoints are precise.

## **2.15.7 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. The imprecise data abort 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\_abt and SPSR\_abt) in these cases, that leads the processor to enter an unrecoverable state, the system must hold the existence of a pending imprecise data abort until a time when the Abort mode can safely be entered.

A mask is included in 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 to 0 and the abort is taken. The A bit is set to 1 automatically on entry into Abort Mode, IRQ, and FIQ Modes, and on Reset. See the *ARM Architecture Reference Manual* for more information.

### ———— **Note** —————

You cannot change the CPSR A bit in the Nonsecure state if the SCR bit [5] is reset. You can change the SPSR A bit in the Nonsecure state but this does not update the CPSR if the SCR bit [5] does not permit it.

## **2.15.8 Software interrupt instruction**

You can use the *Supervisor Call* (SVC) instruction to enter Supervisor mode, usually to request a particular supervisor function. The SVC handler reads the opcode to extract the SVC function number. A SVC 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 SVC. IRQs are disabled when a software interrupt occurs.

### 2.15.9 Software Monitor Instruction

When the processor executes the *Secure Monitor Call* (SMC) instruction, the core enters Monitor mode to request a Monitor function.

———— **Note** —————

An attempt by a User process to execute an SMC causes an Undefined Instruction exception.

---

### 2.15.10 Undefined instruction

When the processor encounters an instruction that neither it nor any coprocessor in the system can handle, it takes the Undefined Instruction exception. Software can use this mechanism to extend the ARM instruction set by emulating Undefined coprocessor instructions.

After emulating the failed instruction, the exception 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 exception.

IRQs are disabled when an Undefined Instruction exception occurs. See the *ARM Architecture Reference Manual* for more information about Undefined instructions.

### 2.15.11 Breakpoint instruction

A breakpoint, BKPT, instruction operates as though the instruction causes a prefetch abort. A breakpoint instruction does not cause the processor to take the prefetch abort exception until the instruction reaches the Execute stage of the pipeline. If the processor does not execute the instruction, 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 Halting debug-mode, a breakpoint instruction causes the processor to enter debug state. See *Halting debug-mode debugging* on page 12-4.

### 2.15.12 Exception vectors

The Secure Configuration Register bits [3:1] determine which mode is entered when an IRQ, a FIQ, or an external abort exception occurs. The CP15 c12, Secure or Nonsecure Vector Base Address Register and the Monitor Vector Base Address Register define the base address of the Nonsecure, Secure, and Secure Monitor vector tables. If high vectors are enabled using CP15 c1 bit[13], the base address of the Nonsecure and Secure vector tables is 0xFFFF0000, regardless of the value of these registers. Enabling high vectors has no effect on the Secure Monitor vector addresses.

### 2.15.13 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order that they are handled. Table 2-13 shows the order of exception priorities.

**Table 2-13 Exception priorities**

Priority	Exception
Highest	1 Reset
	2 Precise data abort
	3 FIQ
	4 IRQ
	5 Prefetch abort
	6 Imprecise data abort
Lowest	7 BKPT
	Undefined instruction
	SVC
	SMC

Some exceptions cannot occur together:

- The BKPT, Undefined instruction, SMC, and SVC 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 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.

———— **Note** —————

If the data abort is a precise external abort and bit [3] EA of SCR is set to 1, the processor enters Monitor mode where aborts and FIQs are disabled automatically. Therefore the processor does not proceed to FIQ vector afterwards.

---

## 2.16 Software consideration for Security Extensions

The Monitor mode is responsible for the switch from one state to the other. You must only modify the SCR in Monitor mode.

The recommended way to return to the Nonsecure state is to:

- set the NS bit to 1 in the SCR
- execute a MOV<sub>S</sub> or SUB<sub>S</sub>.

All ARM implementations ensure that the processor cannot execute the prefetched instructions that follow MOV<sub>S</sub>, SUB<sub>S</sub>, or equivalents, with secure access permissions.

It is strongly recommended that you do not use an MSR instruction to switch from the Secure to the Nonsecure state. There is no guarantee enforced in the architecture that, after the NS bit is set to 1 in Monitor mode, an MSR instruction avoids execution of prefetched instructions with secure access permission. This is because the processor prefetches the instructions that follow the MSR with secure privileged permissions. This might form a security hole in the system if the prefetched instructions then execute in the Nonsecure state.

If the prefetched instructions are in nonsecure memory, with the MSR at the boundary between secure and nonsecure memory, they might be corrupted when giving secure information to the Nonsecure state.

To avoid this problem with the MSR instruction, you can use an IMB sequence shortly after the MSR. If you use the IMB sequence you must ensure that the instructions executed after the MSR and before the IMB do not leak any information to the Nonsecure state and do not rely on the secure permission level.

It is strongly recommended that you do not set the NS bit to 1 in privileged modes other than in Monitor mode. If you do so, you face the same problem as a return to the Nonsecure state with the MSR instruction. To avoid leakage after an MSR instruction, use an IMB sequence.

To enter the Secure Monitor, the processor executes the following instruction:

```
SMC {<cond>} <imm4>
```

where:

**<cond>** Is the condition that the processor executes the SMC.

**<imm4>** The processor ignores this 4-bit immediate value, but the Secure Monitor can use it to determine the service to provide.

To return from the Secure Monitor, the processor executes the following instruction:

```
MOVS PC, R14_mon
```



## 2.17 Hardware consideration for Security Extensions

This section describes the following:

- *System boot sequence*
- *Security Extensions write access disable* on page 2-46
- *Secure monitor bus* on page 2-46.

### 2.17.1 System boot sequence

#### Caution

Security Extensions computing enable a secure software environment. The technology does not protect the processor from hardware attacks, and you must make sure that the hardware containing the boot code is appropriately secure.

The processor always boots in the privileged Supervisor mode in the Secure state, that is the NS bit is 0. This means that code not written for Security Extensions always run in the Secure state, but has no way to switch to the Nonsecure state. Because the Secure and Nonsecure states mirror each other, this secure operation does not affect the functionality of code not written for Security Extensions. Peripherals boot in the Secure state.

The secure OS code at the reset vector must:

1. Initialize the secure OS. This includes normal boot actions such as:
  - a. Generate translation tables and switch on the MMU if the design uses caches or memory protection.
  - b. Switch on the stack.
  - c. Set up the run time environment and program stacks for each processor mode.
2. Initialize the Secure Monitor. This includes such actions as:
  - a. Allocate scratch work space.
  - b. Set up the Secure Monitor stack pointer and initialize its state block.
3. Program the partition checker to allocate physical memory available to the nonsecure OS.
4. Yield control to the nonsecure OS with an SMC instruction. The nonsecure OS boots after this.

The overall security of the software relies on the security of the boot code along with the code for the Secure Monitor.

## 2.17.2 Security Extensions write access disable

The processor pin **CP15SDISABLE** disables write access to certain registers in the system control coprocessor. Attempts to write to these registers when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception. Reads from the registers are still permitted. See Chapter 3 *System Control Coprocessor* for more information about the registers affected by this pin.

A change to the **CP15SDISABLE** pin takes effect on the instructions decoded by the processor as quickly as practically possible. Software must perform an ISB instruction, after a change to this pin on the boundary of the macrocell, to ensure that its effect is recognized for following instructions. It is expected that:

- control of the **CP15SDISABLE** pin remains within the SoC that embodies the macrocell
- the **CP15SDISABLE** pin is cleared to logic 0 by the SoC hardware at reset.

You can use the **CP15SDISABLE** pin to disable subsequent access to the system control processor registers after the secure boot code runs and protect the configuration that the secure boot code applies.

———— **Note** ————

The register accesses affected by the **CP15SDISABLE** pin are only accessible in secure privileged modes.

## 2.17.3 Secure monitor bus

The **SECMONOUT** bus exports a set of signals from the processor.

———— **Caution** ————

You must ensure that the **SECMONOUT** signals do not compromise the security of the processor.

The **SECMONOUTEN** input enables the security monitor output **SECMONOUT[86:0]**. The **SECMONOUTEN** signal is sampled at reset. Any change to the state of this pin during functional operation is ignored.

See Appendix A *Signal Descriptions* for a list of signals that appear on the secure monitor bus **SECMONOUT[86:0]**.

### **SECMONOUT protocol**

The following pseudo code shows the protocol of **SECMONOUT[86:0]**.

```

if SECMONOUTEN = 0 at reset, then
    SECMONOUT[86:0] holds its value
else if SECMONOUTEN = 1 at reset, then
    if SECMONOUT[86] = 1, then
        valid L1 data address present on SECMONOUT[59:40]
    else
        invalid L1 data address

    if SECMONOUT[85] = 1, then
        valid exception data present on SECMONOUT[64:60]
    else
        invalid exception data

    if SECMONOUT[82] = 1, then
        valid pipeline 1 instruction address on SECMONOUT[39:20]
        valid pipeline 1 condition code fail on SECMONOUT[84]
    else
        invalid instruction or condition code in pipeline 1

    if SECMONOUT[81] = 1, then
        valid pipeline 0 instruction address on SECMONOUT[19:0]
        valid pipeline 0 condition code fail on SECMONOUT[83]
    else
        invalid instruction or condition code in pipeline 0

any change of state is exported for the following pins:
SECMONOUT[80]    DMB or DWB executed
SECMONOUT[79]    IMB executed
SECMONOUT[78]    instruction caches at all levels enabled if set
                  to 1 or disabled if cleared to 0

```

## 2.18 Control coprocessor

The processor does not have an external coprocessor interface but it does implement two internal coprocessors, CP14 and CP15.

The CP15 coprocessor is also known as the *system control coprocessor* and is used to control and provide status information for the functions implemented in the processor. See Chapter 3 *System Control Coprocessor* for more information on the system control coprocessor.

The CP14 coprocessor is also known as the *debug coprocessor* and is used for various debug functions. See Chapter 12 *Debug* for more information on the debug coprocessor.

# Chapter 3

## System Control Coprocessor

This chapter describes the purpose of the system control coprocessor, its structure, operation, and how to use it. It contains the following sections:

- *About the system control coprocessor* on page 3-2
- *System control coprocessor registers* on page 3-9.

## 3.1 About the system control coprocessor

This section gives an overall view of the system control coprocessor. See *System control coprocessor registers* on page 3-9 for detail of the registers in the system control coprocessor.

The purpose of the system control coprocessor, CP15, is to control and provide status information for the functions implemented in the processor. The main functions of the system control coprocessor are:

- overall system control and configuration
- cache configuration and management
- *Memory Management Unit* (MMU) configuration and management
- preloading engine for L2 cache
- system performance monitoring.

The system control coprocessor does not exist in a distinct physical block of logic.

The following CP15 instructions are valid NOP instructions:

```
MCR p15, 0, <Rd>, c7, c0, 4 ; NOP (wait-for-interrupt, replaced by WFI
                               ; instruction)
```

At reset, the following CP15 instructions are valid NOP instructions. This behavior is strongly recommended for best performance. Following reset, software can configure the instructions to operate in the traditional manner by programming the Auxiliary Control Register. See *c1, Auxiliary Control Register* on page 3-61 for more details.

```
MCR p15, 0, <Rd>, c7, c5, 6 ; NOP (invalidate entire branch predictor array)
MCR p15, 0, <Rd>, c7, c5, 7 ; NOP (invalidate branch predictor array line by
                               ; MVA)
```

### 3.1.1 System control coprocessor functional groups

The system control coprocessor is a set of registers that you can write to and read from. Some of the registers permit more than one type of operation. The functional groups for the registers are:

- *System control and configuration* on page 3-5
- *MMU control and configuration* on page 3-7
- *Cache control and configuration* on page 3-7
- *L2 cache preload engine control and configuration* on page 3-7
- *System performance monitor* on page 3-8
- *Array debug* on page 3-8.

The system control coprocessor controls the Security Extensions operation of the processor:

- some of the registers are only accessible in the Secure state
- some of the registers are banked for Secure and Nonsecure states
- some of the registers are common to Secure and Nonsecure states.

———— **Note** —————

When Monitor mode is active, the core is in the Secure state. The processor treats all accesses as secure and the system control coprocessor behaves as if it operates in the Secure state regardless of the value of the NS bit, see *c1, Secure Configuration Register* on page 3-70. In Monitor mode the NS bit defines which copies of the banked registers in the system control coprocessor the processor can access:

**NS = 0**      Access to Secure state CP15 registers.

**NS = 1**      Access to Nonsecure state CP15 registers.

Registers that are only accessible in the Secure state are always accessible in Monitor mode, regardless of the value of the NS bit.

Table 3-1 shows the overall functionality of the system control coprocessor registers.

**Table 3-1 System control coprocessor register functions**

<b>Function</b>	<b>Register/operation</b>	<b>Reference to description</b>
System control and configuration	Control	<i>c1, Control Register</i> on page 3-58
	Auxiliary Control	<i>c1, Auxiliary Control Register</i> on page 3-61
	Secure Configuration	<i>c1, Secure Configuration Register</i> on page 3-70
	Secure Debug Enable	<i>c1, Secure Debug Enable Register</i> on page 3-72
	Nonsecure Access Control	<i>c1, Nonsecure Access Control Register</i> on page 3-73
	Coprocessor Access Control	<i>c1, Coprocessor Access Control Register</i> on page 3-67
	Secure or Nonsecure Vector Base Address	<i>c12, Secure or Nonsecure Vector Base Address Register</i> on page 3-152
	Monitor Vector Base Address	<i>c12, Monitor Vector Base Address Register</i> on page 3-154
	Main ID Register <sup>a</sup>	<i>c0, Main ID Register</i> on page 3-25
	Silicon ID Register	<i>c0, Silicon ID Register</i> on page 3-53
	Product Features	<i>c0, Memory Model Feature Register 0</i> on page 3-35 - <i>c0, Instruction Set Attributes Registers 5-7</i> on page 3-52

Table 3-1 System control coprocessor register functions (continued)

Function	Register/operation	Reference to description
MMU control and configuration	TLB Type	<i>c0, TLB Type Register on page 3-28</i>
	Translation Table Base 0	<i>c2, Translation Table Base Register 0 on page 3-75</i>
	Translation Table Base 1	<i>c2, Translation Table Base Register 1 on page 3-77</i>
	Translation Table Base Control	<i>c2, Translation Table Base Control Register on page 3-79</i>
	Domain Access Control	<i>c3, Domain Access Control Register on page 3-81</i>
	Data Fault Status	<i>c5, Data Fault Status Register on page 3-83</i>
	Auxiliary Fault Status	<i>c5, Auxiliary Fault Status Registers on page 3-87</i>
	Instruction Fault Status	<i>c6, Instruction Fault Address Register on page 3-88</i>
	Instruction Fault Address	<i>c6, Instruction Fault Address Register on page 3-88</i>
	Data Fault Address	<i>c6, Data Fault Address Register on page 3-87</i>
	TLB Operations	<i>c8, TLB operations on page 3-99</i>
	Memory region remap	<i>c10, Memory Region Remap Registers on page 3-131</i>
	Context ID	<i>c13, Context ID Register on page 3-160</i>
	FCSE PID	<i>c13, FCSE PID Register on page 3-157</i>
Thread and Process ID	<i>c13, Thread and Process ID Registers on page 3-161</i>	
Cache control and configuration	Cache Type	<i>c0, Cache Type Register on page 3-26</i>
	Cache Level Identification	<i>c0, Cache Level ID Register on page 3-52</i>
	Cache Size Identification	<i>c0, Cache Size Identification Registers on page 3-54</i>
	Cache Size Selection	<i>c0, Cache Size Selection Register on page 3-57</i>
	Cache operations	<i>c7, Cache operations on page 3-89</i>
L2 cache PreLoad Engine (PLE) control and configuration	PLE Identification and Status	<i>c11, PLE Identification and Status Registers on page 3-137</i>
	PLE User Accessibility	<i>c11, PLE User Accessibility Register on page 3-139</i>
	PLE Channel Number	<i>c11, PLE Channel Number Register on page 3-141</i>
	PLE Enable	<i>c11, PLE enable commands on page 3-142</i>
	PLE Control	<i>c11, PLE Control Register on page 3-143</i>



**Table 3-1 System control coprocessor register functions (continued)**

Function	Register/operation	Reference to description
L2 cache PLE control and configuration	PLE Internal Start Address	<i>c11</i> , <i>PLE Internal Start Address Register</i> on page 3-146
	PLE Internal End Address	<i>c11</i> , <i>PLE Internal End Address Register</i> on page 3-148
	PLE Channel Status	<i>c11</i> , <i>PLE Channel Status Register</i> on page 3-149
	PLE Context ID	<i>c11</i> , <i>PLE Context ID Register</i> on page 3-151
L1 instruction and data cache, and TLB Debug	L1 instruction and data cache, BTB, GHB, and TLB Debug	<i>c15</i> , <i>L1 system array debug data registers</i> on page 3-162
L2 unified cache	L2 unified cache	<i>c15</i> , <i>L2 system array debug data registers</i> on page 3-177
System performance monitor	Performance monitoring	<i>c9</i> , <i>Performance Monitor Control Register</i> on page 3-101 - <i>c9</i> , <i>Interrupt Enable Clear Register</i> on page 3-119

a. Returns device ID code.

### 3.1.2 System control and configuration

The purpose of the system control and configuration registers is to provide overall management of:

- Security Extensions behavior
- memory functionality
- interrupt behavior
- exception handling
- program flow prediction
- coprocessor access rights for CP0-CP13.

The system control and configuration registers also provide the processor ID. Some of the functionality depends on how you set external signals at reset.

System control and configuration behaves in three ways:

- as a set of flags or enables for specific functionality
- as a set of numbers, values that indicate system functionality
- as a set of addresses for processes in memory.

## Security Extensions write access disable

The processor supports a primary input pin, **CP15SDISABLE**, to disable write access to the CP15 registers.

When the **CP15SDISABLE** input is set to 1, any attempt to write to the secure version of the banked register, NS-bit is 0, or any non-banked register, NS-state is 0 results in an Undefined Instruction exception.

Changes in the pin on an instruction boundary occur as quickly as practically possible after a change to this pin. Software must perform a **IMB** after a change to this pin has occurred on the boundary of the macros to ensure that its effects are recognized on following instructions.

At reset, it is expected that this pin is set to logic 0 by the SoC hardware. Control of this pin is expected to remain within the SoC chip that implements the processor.

Table 3-2 shows the CP15 registers affected by the primary input pin, **CP15SDISABLE**.

**Table 3-2 CP15 registers affected by CP15SDISABLE**

Register	Instruction
Control Register	MCR p15, 0, <Rd>, c1, c0, 0
Translation Table Base 0	MCR p15, 0, <Rd>, c2, c0, 0
Translation Table Control Register	MCR p15, 0, <Rd>, c2, c0, 2
Domain Access Control	MCR p15, 0, <Rd>, c3, c0, 0
Primary Region Remap	MCR p15, 0, <Rd>, c10, c2, 0
Normal Memory Region Remap	MCR p15, 0, <Rd>, c10, c2, 1
Vector Base	MCR p15, 0, <Rd>, c12, c0, 0
Monitor Base	MCR p15, 0, <Rd>, c12, c0, 1
FCSE	MCR p15, 0, <Rd>, c13, c0, 0
Array operations	MCR p15, 0, <Rd>, c15, c0-15, 0-7 MRC p15, 0, <Rd>, c15, c0-15, 0-7

### 3.1.3 MMU control and configuration

The purpose of the MMU control and configuration registers is to:

- allocate physical address locations from the *Virtual Addresses* (VAs) that the processor generates
- control program access to memory
- configure translation table memory type attributes
- detect MMU faults and external aborts
- translate and lock translation table walk entries
- hold thread and process IDs.

### 3.1.4 Cache control and configuration

The purpose of the cache control and configuration registers is to:

- provide information on the size and architecture of the instruction and data caches
- control cache maintenance operations that include clean and invalidate caches, drain and flush buffers, and address translation
- override cache behavior during debug or interruptible cache operations.

### 3.1.5 L2 cache preload engine control and configuration

The purpose of the L2 cache PLE control and configuration control registers is to:

- enable software to control transfers to or from the L2 RAM
- transfer large blocks of data
- determine accessibility
- select the PLE channel.

Code can execute several PLE operations while in User mode if these operations are enabled by the PLE User Accessibility Register.

If the PLE control registers attempt to execute a privileged operation in User mode, the processor takes an Undefined instruction trap.

The PLE control registers operation specifies the block of data for transfer, the location of the transfer, and the direction of the PLE. See *L2 PLE* on page 8-6 for more information on the operation.

### 3.1.6 System performance monitor

The purpose of the performance monitor registers is to:

- control the monitoring operation
- count events.

System performance monitoring counts 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 can generate interrupts when the number of events reaches a given value.

### 3.1.7 Array debug

The purpose of array debug is to enable debug of the Cortex-A8 processor by accessing data, only in a secure state and privilege state, in the following arrays:

- L1:
  - instruction and data cache data RAMs
  - instruction and data cache tag RAMs
  - TLB entries
  - branch predictor arrays.
- L2 cache RAMs
- parity error detection registers.

You can use the registers to observe the contents of the cache without executing a load or store instruction to debug:

- frequency issues
- *Real Time Operating System (RTOS)*.

## 3.2 System control coprocessor registers

This section describes all the registers in the system control coprocessor. The section presents a summary of the registers and descriptions in register order of CRn, Opcode\_1, CRm, Opcode\_2.

See the *ARM Architecture Reference Manual* for more information on using system control coprocessors and the general method on how to access CP15 registers.

### 3.2.1 Register allocation

Table 3-3 shows a summary the register allocation and reset values of the system control coprocessor where:

- CRn is the register number within CP15
- Op1 is the Opcode\_1 value for the register
- CRm is the operational register
- Op2 is the Opcode\_2 value for the register
- Security state can be Secure, S, or Nonsecure, NS, and is:
  - B, registers banked in Secure and Nonsecure states. If the registers are not banked then they are common to Secure or Nonsecure states or only accessible in one state.
  - NA, no access
  - **RO**, read-only access
  - RO, read-only access in privileged modes only
  - **R/W**, read/write access
  - R/W, read/write access in privileged modes only
  - **WO**, write-only access
  - WO, write-only access in privileged modes only
  - X, access depends on another register or external signal.

**Table 3-3 Summary of CP15 registers and operations**

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
c0	0	c0	{0, 4, 6-7}	Main ID	RO	RO	0x413FC081	page 3-25
			1	Cache Type	RO	RO	0x80048004 <sup>a</sup>	page 3-26
			2	TCM Type	RO	RO	0x00000000	page 3-27

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			3	TLB Type	RO	RO	0x00202001	page 3-28
			5	Multiprocessor ID	RO	RO	0x00000000	page 3-29
	c1		0	Processor Feature 0	RO	RO	0x00001031	page 3-30
			1	Processor Feature 1	RO	RO	0x00000011	page 3-31
			2	Debug Feature 0	RO	RO	0x00010400 or 0x00000400	page 3-33
			3	Auxiliary Feature 0	RO	RO	0x00000000	page 3-34
			4	Memory Model Feature 0	RO	RO	0x31100003	page 3-35
			5	Memory Model Feature 1	RO	RO	0x20000000	page 3-37
			6	Memory Model Feature 2	RO	RO	0x01202000	page 3-39
			7	Memory Model Feature 3	RO	RO	0x00000011	page 3-41
	c2		0	Instruction Set Attribute 0	RO	RO	0x00101111	page 3-43
			1	Instruction Set Attribute 1	RO	RO	0x12112111	page 3-44
			2	Instruction Set Attribute 2	RO	RO	0x21232031	page 3-46
			3	Instruction Set Attribute 3	RO	RO	0x11112131	page 3-48
			4	Instruction Set Attribute 4	RO	RO	0x00011142	page 3-50
			5-7	Instruction Set Attribute 5-7	RO	RO	0x00000000	page 3-52

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page	
					NS	S			
		c3-c7	0-7	Reserved for Feature ID Registers	RO	RO	0x00000000	-	
		c8-c15	0-7	Undefined	-	-	-	-	
1	c0	0	0	Cache Size Identification	RO	RO	Unpredictable	page 3-54	
			1	Cache Level ID	RO	RO	0x0A000023 or 0x0A000003	page 3-52	
			2-6	Undefined	-	-	-	-	
			7	Silicon ID	RO	RO	b	page 3-53	
			c1-c15	0-7	Undefined	-	-	-	-
2	c0	0	0	Cache Size Selection	R/W	R/W, B	Unpredictable	page 3-57	
			1-7	Undefined	-	-	-	-	
			c1-c15	0-7	Undefined	-	-	-	-
	3-7	c0-c15	0-7	Undefined	-	-	-	-	
c1	0	c0	0	Control	R/W	R/W, B <sup>c</sup> , X	0x00C50078 <sup>d</sup>	page 3-58	
			1	Auxiliary Control	B	B	0x00000002	page 3-61	
			2	Coprocessor Access Control	R/W	R/W	0x00000000	page 3-67	
			3-7	Undefined	-	-	-	-	
	c1	0	0	0	Secure Configuration	NA	R/W	0x00000000	page 3-70
				1	Secure Debug Enable	NA	R/W	0x00000000	page 3-72
				2	Nonsecure Access Control	RO	R/W	0x00000000	page 3-73

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page	
					NS	S			
			3-7	Undefined	-	-	-	-	
		c2-c15	0-7	Undefined	-	-	-	-	
	1-7	c0-c15	0-7	Undefined	-	-	-	-	
c2	0	c0	0	Translation Table Base 0	R/W	R/W, B, X	Unpredictable	page 3-75	
			1	Translation Table Base 1	R/W	R/W, B	Unpredictable	page 3-77	
			2	Translation Table Base Control	R/W	R/W, B, X	Unpredictable	page 3-79	
			3-7	Undefined	-	-	-	-	
			c1-c15	0-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-	
c3	0	c0	0	Domain Access Control	R/W	R/W, B, X	Unpredictable	page 3-81	
			1-7	Undefined	-	-	-	-	
			c1-c15	0-7	Undefined	-	-	-	-
			1-7	c0-c15	0-7	Undefined	-	-	-
c4	0-7	c0-c15	0-7	Undefined	-	-	-	-	
c5	0	c0	0	Data Fault Status	R/W	R/W, B	Unpredictable	page 3-83	
			1	Instruction Fault Status	R/W	R/W, B	Unpredictable	page 3-85	
			2-7	Undefined	-	-	-	-	
			c1	0	Data Auxiliary Fault Status	R/W	R/W, B	Unpredictable	page 3-87
				1	Instruction Auxiliary Fault Status	R/W	R/W, B	Unpredictable	page 3-87



Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
		c1	2-7	Undefined	-	-	-	-
		c2-c15	0-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-
c6	0	c0	0	Data Fault Address	R/W	R/W, B	Unpredictable	page 3-87
			1	Undefined	-	-	-	-
			2	Instruction Fault Address	R/W	R/W, B	Unpredictable	page 3-88
		3-7	Undefined	-	-	-	-	
		c1-c15	0-7	Undefined	-	-	-	-
		1-7	c0-c15	0-7	Undefined	-	-	-
c7	0	c0	0-3	Undefined	-	-	-	-
			4	NOP (WFI)	WO	WO	-	page 3-2
			5-7	Undefined	-	-	-	-
		c1-c3	0-7	Undefined	-	-	-	-
		c4	0	Physical Address	R/W	R/W, B	0x00000000	page 3-93
	1-7		Undefined	-	-	-	-	
		c5	0	Invalidate all instruction caches to point of unification	WO	WO	-	page 3-89
			1	Invalidate instruction cache line to point of unification	WO	WO	-	page 3-89
			2-3	Undefined	-	-	-	-
			4	Flush Prefetch Buffer	WO	WO	-	page 3-89

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			5	Undefined	-	-	-	-
			6	NOP (Invalidate entire branch predictor array)	WO	WO	-	page 3-2
			7	NOP (Invalidate branch predictor array line by MVA)	WO	WO	-	page 3-2
c6			0	Undefined	-	-	-	-
			1	Invalidate data cache line to point of coherency by MVA	WO	WO	-	page 3-89
			2	Invalidate data cache line by set and way	WO	WO	-	page 3-89
			3-7	Undefined	-	-	-	-
c7			0-7	Undefined	-	-	-	-
c8			0-3	VA to PA translation in the current state	WO	WO	-	page 3-97
			4-7	VA to PA translation in the other state	NA	WO	-	page 3-97
c9			0-7	Undefined	-	-	-	-
c10			0	Undefined	-	-	-	-
			1	Clean data cache line to point of coherency by MVA	WO	WO	-	page 3-89
			2	Clean data cache line by set and way	WO	WO	-	page 3-89

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			3	Undefined	-	-	-	-
			4	Data Synchronization Barrier	WO	WO	-	page 3-98
			5	Data Memory Barrier	WO	WO	-	page 3-99
			6-7	Undefined	-	-	-	-
	c11		0	Undefined	-	-	-	-
			1	Clean data cache line to point of unification by MVA	WO	WO	-	page 3-89
			2-7	Undefined	-	-	-	-
	c12-c13		0-7	Undefined	-	-	-	-
	c14		0	Undefined	-	-	-	-
			1	Clean and invalidate data cache line to point of coherency by MVA	WO	WO	-	page 3-89
			2	Clean and invalidate data cache line by set and way	WO	WO	-	page 3-89
			3-7	Undefined	-	-	-	-
	c15		0-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-
c8	0	c0-c4	0-7	Undefined	-	-	-	-

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
		c5	0	Invalidate Instruction TLB unlocked entries	WO	WO, B	-	page 3-99
			1	Invalidate Instruction TLB entry by MVA	WO	WO, B	-	page 3-99
			2	Invalidate Instruction TLB entry on ASID match	WO	WO, B	-	page 3-99
			3-7	Undefined	-	-	-	-
		c6	0	Invalidate Data TLB unlocked entries	WO	WO, B	-	page 3-99
			1	Invalidate Data TLB entry by MVA	WO	WO, B	-	page 3-99
			2	Invalidate Data TLB entry on ASID match	WO	WO, B	-	page 3-99
			3-7	Undefined	-	-	-	-
		c7	0	Invalidate unified TLB unlocked entries	WO	WO, B	-	page 3-99
			1	Invalidate unified TLB entry by MVA	WO	WO, B	-	page 3-99
			2	Invalidate unified TLB entry on ASID match	WO	WO, B	-	page 3-99
			3-7	Undefined	-	-	-	-
		c8-c15	0-7	Undefined	-	-	-	-

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page	
					NS	S			
	1-7	c0-c15	0-7	Undefined	-	-	-	-	
c9	0	c0-c11	0-7	Undefined	-	-	-	-	
			c12	0	Performance Monitor Control	R/W, X	R/W, X	0x41002000	page 3-101
			1	Count Enable Set	R/W, X	R/W, X	0x00000000	page 3-103	
			2	Count Enable Clear	R/W, X	R/W, X	0x00000000	page 3-104	
			3	Overflow Flag Status	R/W, X	R/W, X	0x00000000	page 3-106	
			4	Software Increment	R/W, X	R/W, X	0x00000000	page 3-107	
			5	Performance Counter Selection	R/W, X	R/W, X	Unpredictable	page 3-108	
			6-7	Undefined	-	-	-	-	
			c13	0	Cycle Count	R/W, X	R/W, X	0x00000000	page 3-110
				1	Event Selection	R/W, X	R/W, X	Unpredictable	page 3-110
				2	Performance Monitor Count	R/W, X	R/W, X	0x00000000	page 3-116
				3-7	Undefined	-	-	-	-
			c14	0	User Enable	R/W	R/W	0x00000000	page 3-117
				1	Interrupt Enable Set	R/W	R/W	0x00000000	page 3-118
				2	Interrupt Enable Clear	R/W	R/W	0x00000000	page 3-119
				3-7	Undefined	-	-	-	-
			c15	0-7	Undefined	-	-	-	-
1	c0	0	L2 Cache Lockdown	R/W	R/W	0x00000000	page 3-121		

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			1	Undefined	-	-	-	-
			2	L2 Cache Auxiliary Control	RO	R/W	0x00000042	page 3-124
			3-7	Undefined	-	-	-	-
		c1-c15	0-7	Undefined	-	-	-	-
	2-7	c0-c15	0-7	Undefined	-	-	-	-
c10	0	c0	0	Data TLB Lockdown Register	R/W	R/W	0x00000000	page 3-128
			1	Instruction TLB Lockdown Register	R/W	R/W	0x00000000	page 3-128
			2-7	Undefined	-	-	-	-
		c1	0	Data TLB Preload	WO	WO	-	page 3-130
			1	Instruction TLB Preload	WO	WO	-	page 3-130
			2-7	Undefined	-	-	-	-
		c2	0	Primary Region Remap Register	R/W	R/W, B, X	0x00098AA4	page 3-131
			1	Normal Memory Remap Register	R/W	R/W, B, X	0x44E048E0	page 3-131
			2-7	Undefined	-	-	-	-
		c3-c15	0-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-
c11	0	c0	0	PLE Identification and Status	RO, X	RO	0x00000003 <sup>c</sup>	page 3-137
			1	Undefined	-	-	-	-
			2-3	PLE Identification and Status	RO, X	RO	0x00000000 <sup>c</sup>	page 3-137

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			4-7	Undefined	-	-	-	-
	c1		0	PLE User Accessibility	R/W, X	R/W	0x00000000	page 3-139
			1-7	Undefined	-	-	-	-
	c2		0	PLE Channel Number	R/W, X	R/W, X	Unpredictable	page 3-141
			1-7	Undefined	-	-	-	-
	c3		0-2	PLE enable	WO, X	WO, X	-	page 3-142
			3-7	Undefined	-	-	-	-
	c4		0	PLE Control	R/W, X	R/W, X	Unpredictable	page 3-143
			1-7	Undefined	-	-	-	-
	c5		0	PLE Internal Start Address	R/W, X	R/W, X	Unpredictable	page 3-146
			1-7	Undefined	-	-	-	-
	c6		0-7	Undefined	-	-	-	-
	c7		0	PLE Internal End Address	R/W, X	R/W, X	Unpredictable	page 3-148
			1-7	Undefined	-	-	-	-
	c8		0	PLE Channel Status	RO, X	RO, X	0x00000000	page 3-149
			1-7	Undefined	-	-	-	-
	c9-14		0-7	Undefined	-	-	-	-
	c15		0	PLE Context ID	R/W, X	R/W	Unpredictable	page 3-151
c11	0	c15	1-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
c12	0	c0	0	Secure or Nonsecure Vector Base Address	R/W	R/W, B, X	0x00000000	page 3-152
			1	Monitor Vector Base Address	NA	R/W, X	0x00000000	page 3-154
			2-7	Undefined	-	-	-	-
	c1	0	Interrupt Status	RO	RO	0x00000000 <sup>f</sup>	page 3-156	
		1-7	Undefined	-	-	-	-	
	c2-15	0-7	Undefined	-	-	-	-	
	1-7	c0-15	0-7	Undefined	-	-	-	-
c13	0	c0	0	FCSE PID	R/W	R/W, B, X	0x00000000	page 3-157
			1	Context ID	R/W	R/W, B	Unpredictable	page 3-160
			2	User read/write Thread and Process ID	<b>R/W</b>	<b>R/W, B</b>	Unpredictable	page 3-161
			3	User read-only Thread and Process ID	R/W, <b>RO</b>	R/W, <b>RO</b> , B <sup>g</sup>	Unpredictable	page 3-161
			4	Privileged only Thread and Process ID	R/W	R/W, B	Unpredictable	page 3-161
			5-7	Undefined	-	-	-	-
			c1-c15	0-7	Undefined	-	-	-
1-7	c0-c15	0-7	Undefined	-	-	-	-	
c14	0-7	c0-c15	0-7	Undefined	-	-	-	-
c15	0	c0	0	D-L1 Data 0 Register	NA	R/W	Unpredictable	page 3-162



Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			1	D-L1 Data 1 Register	NA	R/W	Unpredictable	page 3-162
			2	D-TLB CAM write operation	NA	WO	-	page 3-168
			3	D-TLB ATTR write operation	NA	WO	-	page 3-168
			4	D-TLB PA write operation	NA	WO	-	page 3-168
			5	D-HVAB write operation	NA	WO	-	page 3-171
			6	D-Tag write operation	NA	WO	-	page 3-172
			7	D-Data write operation	NA	WO	-	page 3-173
	c1		0	I-L1 Data 0 Register	NA	R/W	Unpredictable	page 3-162
			1	I-L1 Data 1 Register	NA	R/W	Unpredictable	page 3-162
			2	I-TLB CAM write operation	NA	WO	-	page 3-168
			3	I-TLB ATTR write operation	NA	WO	-	page 3-168
			4	I-TLB PA write operation	NA	WO	-	page 3-168
			5	I-HVAB write operation	NA	WO	-	page 3-171
			6	I-Tag write operation	NA	WO	-	page 3-172
			7	I-Data write operation	NA	WO	-	page 3-173

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
		c2	0-1	Undefined	-	-	-	-
			2	D-TLB CAM read operation	NA	WO	-	page 3-168
			3	D-TLB ATTR read operation	NA	WO	-	page 3-168
			4	D-TLB PA read operation	NA	WO	-	page 3-168
			5	D-HVAB read operation	NA	WO	-	page 3-171
			6	D-Tag read operation	NA	WO	-	page 3-172
			7	D-Data read operation	NA	WO	-	page 3-173
		c3	0-1	Undefined	NA	-	-	-
			2	I-TLB CAM read operation	NA	WO	-	page 3-168
			3	I-TLB ATTR read operation	NA	WO	-	page 3-168
			4	I-TLB PA read operation	NA	WO	-	page 3-168
			5	I-HVAB read operation	NA	WO	-	page 3-171
			6	I-Tag read operation	NA	WO	-	page 3-172
			7	I-Data read operation	NA	WO	-	page 3-173
		c4	0-7	Undefined	-	-	-	-
		c5	0-1	Undefined	-	-	-	-

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			2	GHB write operation	NA	WO	-	page 3-176
			3	BTB write operation	NA	WO	-	page 3-175
			4-7	Undefined	-	-	-	-
c6			0-7	Undefined	-	-	-	-
c7			0-1	Undefined	-	-	-	-
			2	GHB read operation	NA	WO	-	page 3-176
			3	BTB read operation	NA	WO	-	page 3-175
			4-7	Undefined	-	-	-	-
c8			0	L2 Data 0 Register	NA	R/W	Unpredictable	page 3-177
			1	L2 Data 1 Register	NA	R/W	Unpredictable	page 3-177
			2	L2 tag, L2 valid write operation	NA	WO	-	page 3-182
			3	L2 data, L2 dirty write operation	NA	WO	-	page 3-182
			4	L2 parity and ECC write operation	NA	WO	-	page 3-180
			5	L2 Data 2 Register	NA	R/W	Unpredictable	page 3-177
			6-7	Undefined	-	-	-	-
c9			0-1	Undefined	-	-	-	-
			2	L2 tag, L2 valid read operation	NA	WO	-	page 3-182
			3	L2 data, L2 dirty read operation	NA	WO	-	page 3-182

Table 3-3 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	Security state		Reset value	Page
					NS	S		
			4	L2 parity and ECC read operation	NA	WO	-	page 3-180
			5-7	Undefined	-	-	-	-
		c10-c15	0-7	Undefined	-	-	-	-
	1-7	c0-c15	0-7	Undefined	-	-	-	-

- a. Reset value depends on the cache size implemented. The value here is for 16KB instruction and data caches.
- b. Reset value depends on external signals, that is, **SILICONID[31:0]**.
- c. Some bits in this register are banked and some are secure modify only.
- d. Reset value depends on external signals, that is, **VINITHI**, **CFGTE**, and **CFGNMFI**. The value shown in this table assumes these signals are set to zero.
- e. Reset value depends on the number of PLE channels implemented.
- f. Reset value depends on external signals, that is, **nFIQ** and **nIRQ**. The value shown in this table assumes these signals are set to zero.
- g. This register is read/write in privileged modes and read-only in User mode.

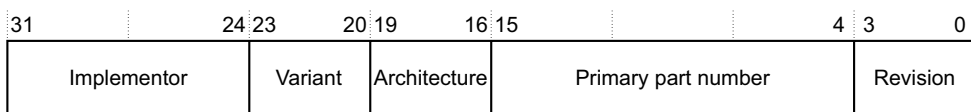
### 3.2.2 c0, Main ID Register

The purpose of the Main ID Register is to return the device ID code that contains information about the processor.

The Main ID Register is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-1 shows the bit arrangement of the Main ID Register.



**Figure 3-1 Main ID Register format**

The contents of the Main ID Register depend on the specific implementation. Table 3-4 shows how the bit values correspond with the Main ID Register functions.

**Table 3-4 Main ID Register bit functions**

Bits	Field	Function
[31:24]	Implementor	Indicates the implementor, ARM: 0x41.
[23:20]	Variant	Indicates the variant number, or major revision, of the processor: 0x3.
[19:16]	Architecture	Indicates that the architecture is given in the feature registers: 0xF.
[15:4]	Primary part number	Indicates the part number, Cortex-A8: 0xC08.
[3:0]	Revision	Indicates the revision number, or minor revision, of the processor: 0x1.

**Note**

If an Opcode\_2 value corresponding to a non-implemented or reserved ID register with CRm equal to c0 and Opcode\_1 = 0 is encountered, the system control coprocessor returns the value of the Main ID Register.

Table 3-5 shows the results of attempted access for each mode.

**Table 3-5 Results of access to the Main ID Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Main ID Register, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c0, 0 ; Read Main ID Register
```

See *c0, Processor Feature Register 0* on page 3-30 - *c0, Instruction Set Attributes Registers 5-7* on page 3-52 for more information on the processor features.

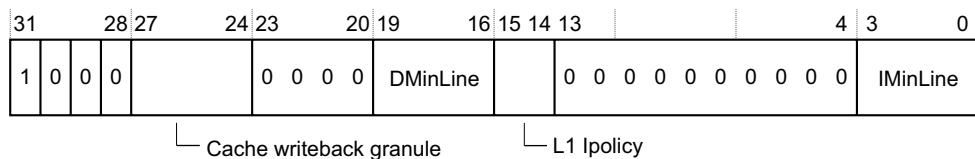
### 3.2.3 c0, Cache Type Register

The purpose of the Cache Type Register is to determine the instruction and data cache minimum line length in bytes to enable a range of addresses to be invalidated.

The Cache Type Register is:

- a read-only register
- accessible in privileged modes only.

The contents of the Cache Type Register depend on the specific implementation. Figure 3-2 shows the bit arrangement of the Cache Type Register.



**Figure 3-2 Cache Type Register format**

Table 3-6 shows how the bit values correspond with the Cache Type Register functions.

**Table 3-6 Cache Type Register bit functions**

Bits	Field	Function
[31:28]	-	Always read as 4'b1000.
[27:24]	Cache writeback granule	Cache writeback granule. $\log_2$ of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location within it modified. 4'b0010 = cache writeback granule size is 4 words.
[23:20]	-	Always read as 4'b0000.
[19:16]	DMinLine	Number of words of smallest line length in L1 or L2 data cache: 4'b0100 = sixteen 32-bit word data line length.
[15:14]	L1 Ipolicy	VIPT instruction cache support: 2'b10 = virtual index, physical tag L1 Ipolicy.
[13: 4]	-	Always read as b0000000000.
[3:0]	IMinLine	Number of words of smallest line length in L1 or L2 instruction cache: 4'b0100 = sixteen 32-bit word data line length.

Table 3-7 shows the results of attempted access for each mode.

**Table 3-7 Results of access to the Cache Type Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Cache Type Register, read CP15 with:

MRC p15, 0, <Rd>, c0, c0, 1 ; Read Cache Type Register

### 3.2.4 c0, TCM Type Register

The processor does not implement *Tightly Coupled Memory* (TCM). The TCM Type Register specifies that the processor does not implement instruction and data TCMs.

The TCM Type Register is:

- a read-only register that is Read-As-Zero
- accessible in privileged modes only.

Table 3-8 shows the results of attempted access for each mode.

**Table 3-8 Results of access to the TCM Type Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the TCM Type Register, read CP15 with:

MRC p15, 0, <Rd>, c0, c0, 2; Read TCM Type Register

### 3.2.5 c0, TLB Type Register

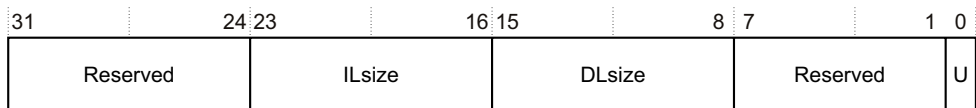
The purpose of the TLB Type Register is to return the number of lockable entries for both the instruction and data TLBs.

Each TLB has 32 entries organized as fully associative and lockable TLB.

The TLB Type Register is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-3 shows the bit arrangement of the TLB Type Register.



**Figure 3-3 TLB Type Register format**



Table 3-9 shows how the bit values correspond with the TLB Type Register functions.

**Table 3-9 TLB Type Register bit functions**

Bits	Field	Function
[31:24]	-	Reserved, <i>Read-As-Zero</i> (RAZ).
[23:16]	ILsize	Instruction lockable size specifies the number of instruction TLB lockable entries: 0x20 = Processor has 32 lockable entries.
[15:8]	DLsize	Data lockable size specifies the number of unified or data TLB lockable entries: 0x20 = Processor has 32 lockable entries.
[7:1]	-	Reserved, RAZ.
[0]	U	Unified specifies if the TLB is unified or if there are separate instruction and data TLBs: 0x1 = Processor has separate instruction and data TLBs.

Table 3-10 shows the results of attempted access for each mode.

**Table 3-10 Results of access to the TLB Type Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the TLB Type Register, read CP15 with:

MRC p15, 0, <Rd>, c0, c0, 3 ; Read TLB Type Register

### 3.2.6 c0, Multiprocessor ID Register

The Multiprocessor ID Register indicates that the processor is a uniprocessor.

The Multiprocessor ID Register is:

- a read-only register that is Read-As-Zero
- accessible in privileged modes only.



Table 3-12 shows how the bit values correspond with the Processor Feature Register 0 functions.

**Table 3-12 Processor Feature Register 0 bit functions**

Bits	Field	Function
[31:16]	-	Reserved, RAZ.
[15:12]	State3	Indicates support for Thumb Execution Environment (ThumbEE): 0x1 = Processor supports ThumbEE.
[11:8]	State2	Indicates support for Jazelle extension interface: 0x1 = Jazelle extension supported.
[7:4]	State1	Indicates the type of Thumb encoding that the processor supports: 0x3 = Processor supports Thumb-2 encoding with all Thumb-2 instructions.
[3:0]	State0	Indicates support for ARM instruction set: 0x1, = Processor supports ARM instructions.

Table 3-13 shows the results of attempted access for each mode.

**Table 3-13 Results of access to the Processor Feature Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Processor Feature Register 0, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 0 ; Read Processor Feature Register 0
```

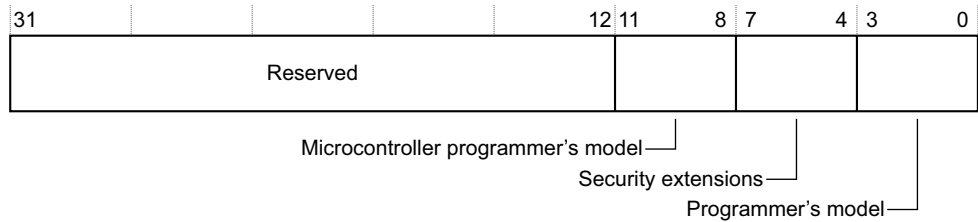
### 3.2.8 c0, Processor Feature Register 1

The purpose of Processor Feature Register 1 is to provide information about the execution state support and programmer's model for the processor.

The Processor Feature Register 1 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-5 shows the bit arrangement of the Processor Feature Register 1.



**Figure 3-5 Processor Feature Register 1 format**

Table 3-14 shows how the bit values correspond with the Processor Feature Register 1 functions.

**Table 3-14 Processor Feature Register 1 bit functions**

Bits	Field	Function
[31:12]	-	Reserved, RAZ.
[11:8]	Microcontroller programmer's model	Indicates support for microcontroller programmer's model: 0x0 = Processor does not support microcontroller programmer's model.
[7:4]	Security extensions	Indicates support for Security Extensions Architecture v1: 0x1 = Processor supports Security Extensions Architecture v1.
[3:0]	Programmer's model	Indicates support for standard ARMv4 programmer's model. All processor operating modes are supported: 0x1 = Processor supports the ARMv4 model.

Table 3-15 shows the results of attempted access for each mode.

**Table 3-15 Results of access to Processor Feature Register 1<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Processor Feature Register 1, read CP15 with:

MRC p15, 0, <Rd>, c0, c1, 1 ; Read Processor Feature Register 1

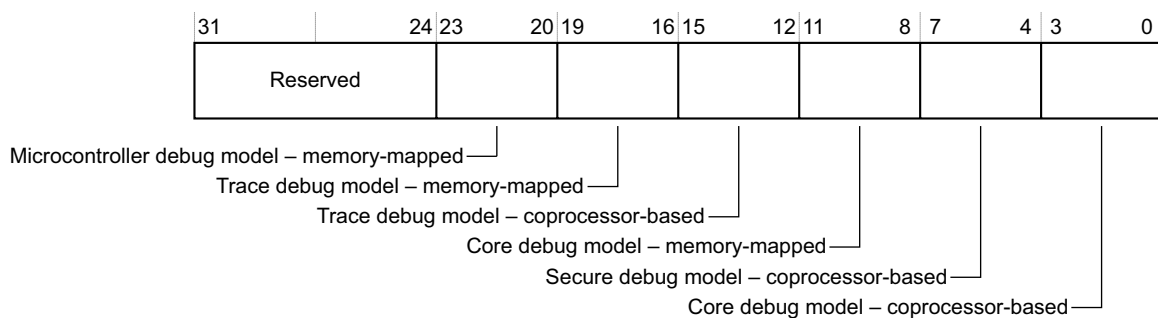
### 3.2.9 c0, Debug Feature Register 0

The purpose of Debug Feature Register 0 is to provide information about the debug system for the processor.

The Debug Feature Register 0 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-6 shows the bit arrangement of the Debug Feature Register 0.



**Figure 3-6 Debug Feature Register 0 format**

Table 3-16 shows how the bit values correspond with the Debug Feature Register 0 functions.

**Table 3-16 Debug Feature Register 0 bit functions**

Bits	Field	Function
[31:24]	-	Reserved, RAZ.
[23:20]	Microcontroller debug model – memory-mapped	Indicates support for the microcontroller debug model: 0x0 = Processor does not support the microcontroller debug model – memory-mapped.
[19:16]	Trace debug model – memory-mapped	Indicates support for the trace debug model – memory-mapped: 0x1 = Processor supports the trace debug model – memory-mapped 0x0 = Processor does not support the trace debug model – memory-mapped. <sup>a</sup>
[15:12]	Trace debug model – coprocessor-based	Indicates support for the coprocessor-based trace debug model: 0x0 = Processor does not support the trace debug model – coprocessor.

**Table 3-16 Debug Feature Register 0 bit functions (continued)**

Bits	Field	Function
[11:8]	Core debug model – memory mapped	Indicates support for the memory-mapped debug model: 0x4 = Processor supports the memory mapped debug model.
[7:4]	Secure debug model – coprocessor-based	Indicates support for the secure debug model – coprocessor: 0x0 = Processor does not support the secure debug model – coprocessor.
[3:0]	Core debug model – coprocessor-based	Indicates support for the coprocessor debug model: 0x0 = Processor does not support the coprocessor debug model.

a. A value of 0x0 indicates that the ETM option is not configured for the processor, see *Configurable options* on page 1-11

Table 3-17 shows the results of attempted access for each mode.

**Table 3-17 Results of access to Debug Feature Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Debug Feature Register 0, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 2 ; Read Debug Feature Register 0
```

### 3.2.10 c0, Auxiliary Feature Register 0

The purpose of Auxiliary Feature Register 0 is to provide additional information about the features of the processor.

The Auxiliary Feature Register 0 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

In the processor, the Auxiliary Feature Register 0 reads as 0x00000000.

Table 3-18 shows the results of attempted access for each mode.

**Table 3-18 Results of access to Auxiliary Feature Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Auxiliary Feature Register 0, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 3 ; Read Auxiliary Feature Register 0
```

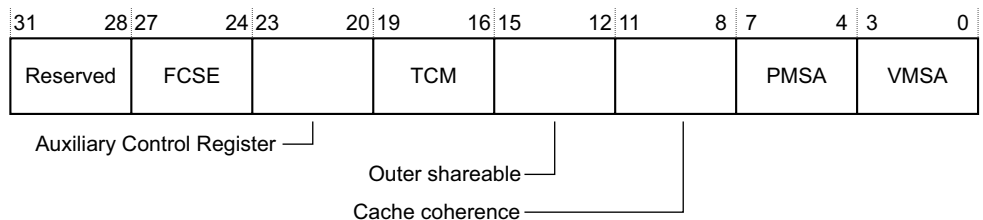
### 3.2.11 c0, Memory Model Feature Register 0

The purpose of the Memory Model Feature Register 0 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 0 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-7 shows the bit arrangement of the Memory Model Feature Register 0.



**Figure 3-7 Memory Model Feature Register 0 format**

Table 3-19 shows how the bit values correspond with the Memory Model Feature Register 0 functions.

**Table 3-19 Memory Model Feature Register 0 bit functions**

Bits	Field	Function
[31:28]	-	Reserved, RAZ.
[27:24]	FCSE	Indicates support for fast context switch memory mappings: 0x1 = Processor supports FCSE.
[23:20]	Auxiliary Control Register	Indicates support for Auxiliary Control Register: 0x1 = Processor supports the Auxiliary Control Register.
[19:16]	TCM	Indicates support for TCM and associated DMA: 0x0 = Processor does not support TCM and DMA.
[15:12]	Outer shareable	Indicates support for outer shareable attribute: 0x0 = Processor does not support this model.
[11:8]	Cache coherence	Indicates support for cache coherency maintenance: 0x0 = Processor does not support this model.
[7:4]	PMSA	Indicates support for <i>Physical Memory System Architecture</i> (PMSA): 0x0 = Processor does not support PMSA.
[3:0]	VMSA	Indicates support for <i>Virtual Memory System Architecture</i> (VMSA). 0x3 = Processor supports: <ul style="list-style-type: none"> <li>• VMSA v7 including cache and TLB type register</li> <li>• Extensions to ARMv6.</li> </ul>

Table 3-20 shows the results of attempted access for each mode.

**Table 3-20 Results of access to Memory Model Feature Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Memory Model Feature Register 0, read CP15 with:

MRC p15, 0, <Rd>, c0, c1, 4 ; Read Memory Model Feature Register 0



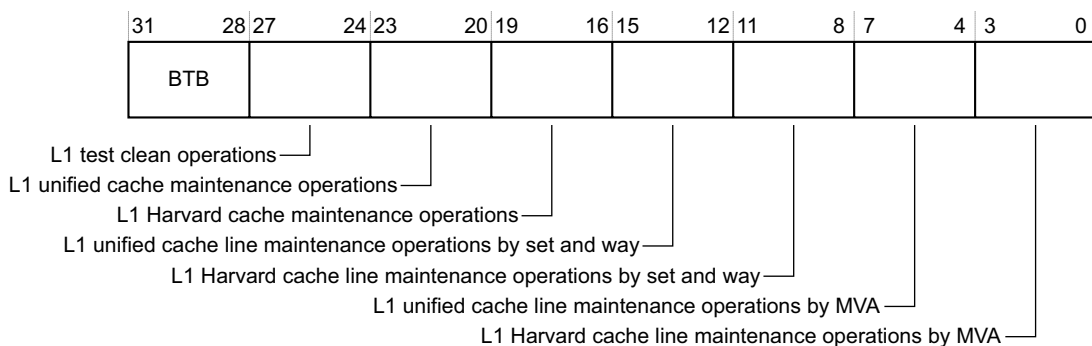
### 3.2.12 c0, Memory Model Feature Register 1

The purpose of the Memory Model Feature Register 1 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 1 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-8 shows the bit arrangement of the Memory Model Feature Register 1.



**Figure 3-8 Memory Model Feature Register 1 format**

Table 3-21 shows how the bit values correspond with the Memory Model Feature Register 1 functions.

**Table 3-21 Memory Model Feature Register 1 bit functions**

Bits	Field	Function
[31:28]	BTB	Indicates support for branch target buffer: 0x2 = Processor does not require flushing of BTB on VA change.
[27:24]	L1 test clean operations	Indicates support for test and clean operations on data cache, Harvard or unified architecture: 0x0 = no support in processor.
[23:20]	L1 unified cache maintenance operations	Indicates support for L1 cache, all maintenance operations, unified architecture: 0x0 = no support in processor.

**Table 3-21 Memory Model Feature Register 1 bit functions (continued)**

Bits	Field	Function
[19:16]	L1 Harvard cache maintenance operations	Indicates support for L1 cache, all maintenance operations, Harvard architecture. $0x0$ = Processor supports: <ul style="list-style-type: none"> <li>invalidate instruction cache including branch target buffer</li> <li>invalidate data cache</li> <li>invalidate instruction and data cache including branch target buffer.</li> </ul> The processor does not support Harvard version.
[15:12]	L1 unified cache line maintenance operations by set and way	Indicates support for L1 cache line maintenance operations by set and way, unified architecture: $0x0$ = no support in processor.
[11:8]	L1 Harvard cache line maintenance operations by set and way	Indicates support for L1 cache line maintenance operations by set and way, Harvard architecture. $0x0$ = Processor supports: <ul style="list-style-type: none"> <li>clean data cache line by set and way</li> <li>clean and invalidate data cache line by set and way</li> <li>invalidate data cache line by set and way</li> <li>invalidate instruction cache line by set and way.</li> </ul>
[7:4]	L1 unified cache line maintenance operations by MVA	Indicates support for L1 cache line maintenance operations by MVA, unified architecture: $0x0$ = no support in processor.
[3:0]	L1 Harvard cache line maintenance operations by MVA	Indicates support for L1 cache line maintenance operations by MVA, Harvard architecture. $0x0$ = Processor supports: <ul style="list-style-type: none"> <li>clean data cache line by MVA</li> <li>invalidate data cache line by MVA</li> <li>invalidate instruction cache line by MVA</li> <li>clean and invalidate data cache line by MVA</li> <li>invalidation of branch target buffer by MVA.</li> </ul>

Table 3-22 shows the results of attempted access for each mode.

**Table 3-22 Results of access to Memory Model Feature Register 1<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Memory Model Feature Register 1, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 5 ; Read Memory Model Feature Register 1
```

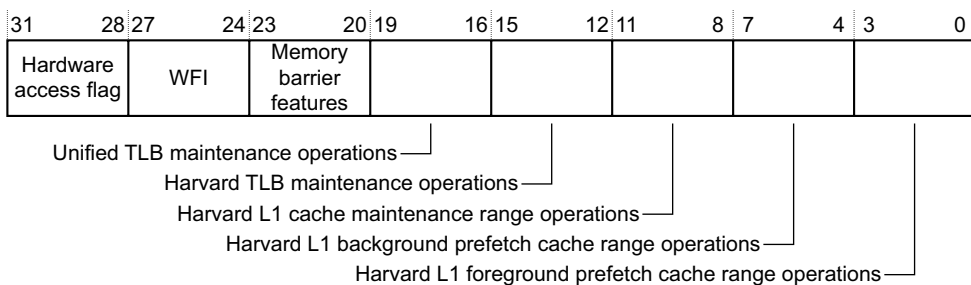
### 3.2.13 c0, Memory Model Feature Register 2

The purpose of the Memory Model Feature Register 2 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 2 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-9 shows the bit arrangement of the Memory Model Feature Register 2.



**Figure 3-9 Memory Model Feature Register 2 format**

Table 3-23 shows how the bit values correspond with the Memory Model Feature Register 2 functions.

**Table 3-23 Memory Model Feature Register 2 bit functions**

Bits	Field	Function
[31:28]	Hardware access flag	Indicates support for hardware access flag: 0x0 = Processor does not support hardware access flag.
[27:24]	WFI	Indicates support for wait-for-interrupt stalling: 0x1 = Processor supports wait-for-interrupt.
[23:20]	Memory barrier features	Indicates support for memory barrier operations. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• data synchronization barrier</li> <li>• instruction synchronization barrier</li> <li>• data memory barrier.</li> </ul>
[19:16]	Unified TLB maintenance operations	Indicates support for TLB maintenance operations, unified architecture. 0x0 = Processor does not support: <ul style="list-style-type: none"> <li>• invalidate all entries</li> <li>• invalidate TLB entry by MVA</li> <li>• invalidate TLB entries by ASID match.</li> </ul>
[15:12]	Harvard TLB maintenance operations	Indicates support for TLB maintenance operations, Harvard architecture. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• invalidate instruction and data TLB, all entries</li> <li>• invalidate instruction TLB, all entries</li> <li>• invalidate data TLB, all entries</li> <li>• invalidate instruction TLB by MVA</li> <li>• invalidate data TLB by MVA</li> <li>• invalidate instruction and data TLB entries by ASID match</li> <li>• invalidate instruction TLB entries by ASID match</li> <li>• invalidate data TLB entries by ASID match.</li> </ul>

**Table 3-23 Memory Model Feature Register 2 bit functions (continued)**

Bits	Field	Function
[11:8]	Harvard L1 cache maintenance range operations	Indicates support for cache maintenance range operations, Harvard architecture: 0x0 = no support in processor.
[7:4]	Harvard L1 background prefetch cache range operations	Indicates support for background prefetch cache range operations, Harvard architecture: 0x0 = no support in processor.
[3:0]	Harvard L1 foreground prefetch cache range operations	Indicates support for foreground prefetch cache range operations, Harvard architecture: 0x0 = no support in processor.

Table 3-24 shows the results of attempted access for each mode.

**Table 3-24 Results of access to Memory Model Feature Register 2<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Memory Model Feature Register 2, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 6 ; Read Memory Model Feature Register 2
```

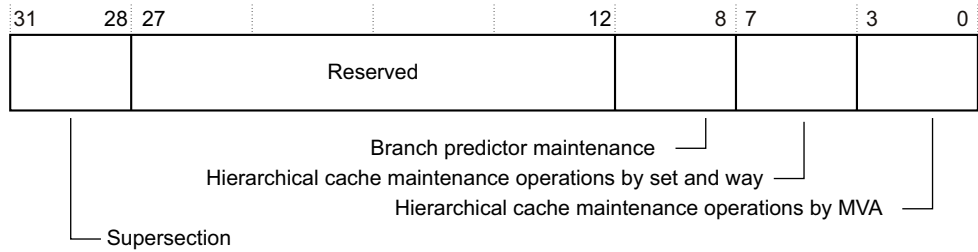
### 3.2.14 c0, Memory Model Feature Register 3

The purpose of the Memory Model Feature Register 3 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 3 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-10 on page 3-42 shows the bit arrangement of the Memory Model Feature Register 3.



**Figure 3-10 Memory Model Feature Register 3 format**

Table 3-25 shows how the bit values correspond with the Memory Model Feature Register 3 functions.

**Table 3-25 Memory Model Feature Register 3 bit functions**

Bits	Field	Function
[31:28]	Supersection	Indicates support for supersections: 0x0 = Processor supports supersections.
[27:12]	-	Reserved, RAZ.
[11:8]	Branch predictor maintenance	Indicates support for branch predictor maintenance operations: 0x2 = Processor supports invalidate entire branch predictor array and invalidate branch predictor by MVA.
[7:4]	Hierarchical cache maintenance operations by set and way	Indicates support for invalidate cache by set and way, clean by set and way, and invalidate and clean by set and way: 0x1 = Processor supports invalidate cache by set and way, clean by set and way, and invalidate and clean by set and way.
[3:0]	Hierarchical cache maintenance operations by MVA	Indicates support for invalidate cache by MVA, clean by MVA, invalidate and clean by MVA, and invalidate all: 0x1 = Processor supports invalidate cache by MVA, clean by MVA, invalidate and clean by MVA, and invalidate all.

Table 3-26 shows the results of attempted access for each mode.

**Table 3-26 Results of access to Memory Model Feature Register 3<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Memory Model Feature Register 3, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c1, 7 ; Read Memory Model Feature Register 3
```

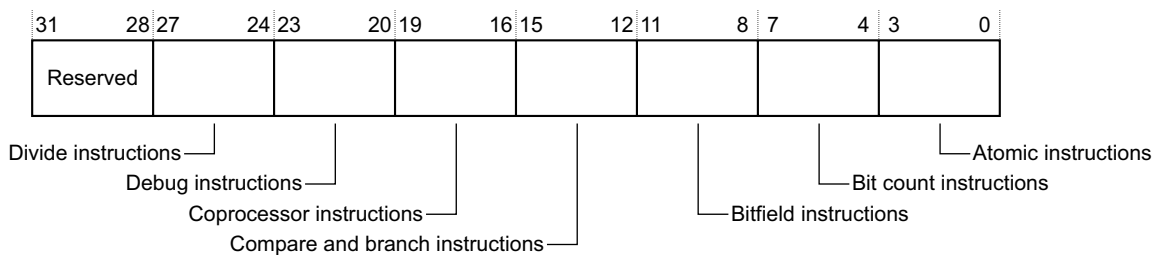
### 3.2.15 c0, Instruction Set Attributes Register 0

The purpose of the Instruction Set Attributes Register 0 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 0 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-11 shows the bit arrangement of the Instruction Set Attributes Register 0.



**Figure 3-11 Instruction Set Attributes Register 0 format**

Table 3-27 shows how the bit values correspond with the Instruction Set Attributes Register 0 functions.

**Table 3-27 Instruction Set Attributes Register 0 bit functions**

Bits	Field	Function
[31:28]	-	Reserved, RAZ.
[27:24]	Divide instructions	Indicates support for divide instructions: 0x0 = Processor does not support divide instructions.
[23:20]	Debug instructions	Indicates support for debug instructions: 0x1 = Processor supports BKPT.
[19:16]	Coprocessor instructions	Indicates support for coprocessor instructions. This field reads as zero (RAZ).

**Table 3-27 Instruction Set Attributes Register 0 bit functions (continued)**

Bits	Field	Function
[15:12]	Compare and branch instructions	Indicates support for combined compare and branch instructions: 0x1 = Processor supports combined compare and branch instructions.
[11:8]	Bitfield instructions	Indicates support for bitfield instructions: 0x1 = Processor supports bitfield instructions.
[7:4]	Bit count instructions	Indicates support for bit counting instructions: 0x1 = Processor supports CLZ.
[3:0]	Atomic instructions	Indicates support for atomic load and store instructions: 0x1 = Processor supports SWP and SWPB.

Table 3-28 shows the results of attempted access for each mode.

**Table 3-28 Results of access to Instruction Set Attributes Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Instruction Set Attributes Register 0, read CP15 with:

MRC p15, 0, <Rd>, c0, c2, 0 ; Read Instruction Set Attributes Register 0

### 3.2.16 c0, Instruction Set Attributes Register 1

The purpose of the Instruction Set Attributes Register 1 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 1 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-12 on page 3-45 shows the bit arrangement of the Instruction Set Attributes Register 1.



31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Jazelle instructions		Inter-working instructions		Immediate instructions		ITE instructions		Extend instructions		Exception 2 instructions		Exception 1 instructions		Endian instructions	

**Figure 3-12 Instruction Set Attributes Register 1 format**

Table 3-29 shows how the bit values correspond with the Instruction Set Attributes Register 1 functions.

**Table 3-29 Instruction Set Attributes Register 1 bit functions**

Bits	Field	Function
[31:28]	Jazelle instructions	Indicates support for Jazelle instructions: 0x1 = Processor supports BXJ and J bit in PSRs.
[27:24]	Interworking instructions	Indicates support for instructions that branch between ARM and Thumb code. 0x3 = Processor supports: <ul style="list-style-type: none"> <li>• BX, and T bit in PSRs</li> <li>• BLX, and PC loads have BX behavior</li> <li>• data-processing instructions in the ARM instruction set with the PC as the destination and the S bit cleared to 0, have the BX behavior.</li> </ul>
[23:20]	Immediate instructions	Indicates support for immediate instructions: 0x1 = Processor supports immediate instructions.
[19:16]	ITE instructions	Indicates support for IfThen instructions: 0x1 = Processor supports IfThen instructions.
[15:12]	Extend instructions	Indicates support for sign or zero extend instructions. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• SXTB, SXTB16, SXTH, UXTB, UXTB16, and UXTH</li> <li>• SXTAB, SXTAB16, SXTAH, UXTAB, UXTAB16, and UXTAH.</li> </ul>
[11:8]	Exception 2 instructions	Indicates support for exception 2 instructions: 0x1 = Processor supports SRS, RFE, and CPS.
[7:4]	Exception 1 instructions	Indicates support for exception 1 instructions: 0x1 = Processor supports LDM(2), LDM(3) and STM(2).
[3:0]	Endian instructions	Indicates support for endianness control instructions: 0x1 = Processor supports SETEND and E bit in PSRs.

Table 3-30 shows the results of attempted access for each mode.

**Table 3-30 Results of access to Instruction Set Attributes Register 1<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Instruction Set Attributes Register 1, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c2, 1 ; Read Instruction Set Attributes Register 1
```

### 3.2.17 c0, Instruction Set Attributes Register 2

The purpose of the Instruction Set Attributes Register 2 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 2 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-13 shows the bit arrangement for the Instruction Set Attributes Register 2.

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Reversal instructions	PSR instructions		Unsigned multiply instructions		Signed multiply instructions		Multiply instructions		Interruptible instructions		Memory hint instructions		Load and store instructions		

**Figure 3-13 Instruction Set Attributes Register 2 format**

Table 3-31 shows how the bit values correspond with the Instruction Set Attributes Register 2 functions.

**Table 3-31 Instruction Set Attributes Register 2 bit functions**

Bits	Field	Function
[31:28]	Reversal instructions	Indicates support for reversal instructions. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• REV</li> <li>• REV16</li> <li>• REVSH</li> <li>• RBIT.</li> </ul>
[27:24]	PSR instructions	Indicates support for PSR instructions: 0x1 = Processor supports MRS and MSR exception return instructions for data processing.
[23:20]	Unsigned multiply instructions	Indicates support for advanced unsigned multiply instructions. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• UMULL and UMLAL</li> <li>• UMAAL.</li> </ul>
[19:16]	Signed multiply instructions	Indicates support for advanced signed multiply instructions. 0x3 = Processor supports: <ul style="list-style-type: none"> <li>• SMULL and SMLAL</li> <li>• SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, SMULWT, and Q flag in PSRs</li> <li>• SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSXD, SMLS LD, SMLS LDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX.</li> </ul>
[15:12]	Multiply instructions	Indicates support for multiply instructions: 0x2 = Processor supports MUL, MLA, and MLS.
[11:8]	Interruptible instructions	Indicates support for multi-access interruptible instructions: 0x0 = Processor does not support restartable LDM and STM.
[7:4]	Memory hint instructions	Indicates support for memory hint instructions: 0x3 = Processor supports PLD, memory hint YIELD (true NOP), and PLI (NOP).
[3:0]	Load and store instructions	Indicates support for load and store instructions: 0x1 = Processor supports LDRD and STRD.

Table 3-32 shows the results of attempted access for each mode.

**Table 3-32 Results of access to Instruction Set Attributes Register 2<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Instruction Set Attributes Register 2, read CP15 with:

```
MRC p15, 0, <Rd>, c0, c2, 2 ; Read Instruction Set Attributes Register 2
```

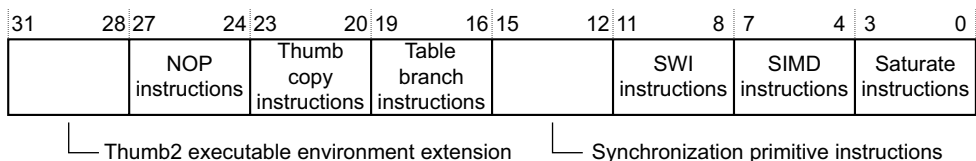
### 3.2.18 c0, Instruction Set Attributes Register 3

The purpose of the Instruction Set Attributes Register 3 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 3 is:

- a read-only registers common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-14 shows the bit arrangement of Instruction Set Attributes Register 3.



**Figure 3-14 Instruction Set Attributes Register 3 format**

Table 3-33 shows how the bit values correspond with the Instruction Set Attributes Register 3 functions.

**Table 3-33 Instruction Set Attributes Register 3 bit functions**

Bits	Field	Function
[31:28]	Thumb2 executable environment extension instructions	Indicates support for Thumb2 Executable Environment Extension instructions: 0x1 = Processor supports ENTERX and LEAVEX instructions and modifies the load behavior to include null checking.
[27:24]	NOP instructions	Indicates support for true NOP instructions: 0x1 = Processor supports true NOP instructions in both the Thumb and ARM instruction sets, and the capability for additional NOP compatible hints.
[23:20]	Thumb copy instructions	Indicates support for Thumb copy instructions: 0x1 = Processor supports Thumb MOV(3) low register ⇒ low register, and the CPY alias for Thumb MOV(3).
[19:16]	Table branch instructions	Indicates support for table branch instructions: 0x1 = Processor supports table branch instructions.
[15:12]	Synchronization primitive instructions	Indicates support for synchronization primitive instructions. 0x2 = Processor supports: <ul style="list-style-type: none"> <li>• LDREX and STREX</li> <li>• LDREXB, LDREXH, LDREXD, STREXB, STREXH, STREXD, and CLREX.</li> </ul>
[11:8]	SVC instructions	Indicates support for SVC instructions: 0x1 = Processor supports SVC.
[7:4]	SIMD instructions	Indicates support for <i>Single Instruction Multiple Data</i> (SIMD) instructions. 0x3 = Processor supports: PKHBT, PKHTB, QADD16, QADD8, QADDSUBX, QSUB16, QSUB8, QSUBADDX, SADD16, SADD8, SADDSUBX, SEL, SHADD16, SHADD8, SHADDSUBX, SHSUB16, SHSUB8, SHSUBADDX, SSAT, SSAT16, SSUB16, SSUB8, SSUBADDX, SXTAB16, SXTB16, UADD16, UADD8, UADDSUBX, UHADD16, UHADD8, UHADDSUBX, UHSUB16, UHSUB8, UHSUBADDX, UQADD16, UQADD8, UQADDSUBX, UQSUB16, UQSUB8, UQSUBADDX, USAD8, USADA8, USAT, USAT16, USUB16, USUB8, USUBADDX, UXTAB16, UXTB16, and the GE[3:0] bits in the PSRs.
[3:0]	Saturate instructions	Indicates support for saturate instructions: 0x1 = Processor supports QADD, QDADD, QDSUB, QSUB and Q flag in PSRs.

Table 3-34 shows the results of attempted access for each mode.

**Table 3-34 Results of access to Instruction Set Attributes Register 3<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Instruction Set Attributes Register 3, read CP15 with:

MRC p15, 0, <Rd>, c0, c2, 3 ; Read Instruction Set Attributes Register 3

### 3.2.19 c0, Instruction Set Attributes Register 4

The purpose of Instruction Set Attributes Register 4 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 4 is:

- a read-only register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-15 shows the bit arrangement of the Instruction Set Attributes Register 4.

31		24	23	20	19	16	15	12	11	8	7	4	3	0
Reserved		Exclusive instructions		Barrier instructions		SMC instructions		Write-back instructions		With-shift instructions		Unprivileged instructions		

**Figure 3-15 Instruction Set Attributes Register 4 format**

Table 3-35 shows how the bit values correspond with the Instruction Set Attributes Register 4 functions.

**Table 3-35 Instruction Set Attributes Register 4 bit functions**

Bits	Field	Function
[31:24]	-	Reserved, RAZ.
[23:20]	Exclusive instructions	Indicates support for exclusive instructions: 0x0 = The processor supports CLREX, LDREX{B H}, and STREX{B H}.
[19:16]	Barrier instructions	Indicates support for barrier instructions: 0x1 = The processor supports DMB, DSB, and ISB.
[15:12]	SMC instructions	Indicates support for SMC instructions: 0x1 = The processor supports SMC.
[11:8]	Write-back instructions	Indicates support for write-back instructions: 0x1 = The processor supports all defined write-back addressing modes.
[7:4]	With-shift instructions	Indicates support for with-shift instructions. 0x4 = The processor supports: <ul style="list-style-type: none"> <li>• shifts of loads and stores over the range LSL 0-3</li> <li>• constant shift options</li> <li>• register-controlled shift options.</li> </ul>
[3:0]	Unprivileged instructions	Indicates support for Unprivileged instructions: 0x2 = The processor supports LDR{SB B SH HT}.

Table 3-36 shows the results of attempted access for each mode.

**Table 3-36 Results of access to Instruction Set Attributes Register 4<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Instruction Set Attributes Register 4, read CP15 with:

MRC p15, 0, <Rd>, c0, c2, 4 ; Read Instruction Set Attributes Register 4

### 3.2.20 c0, Instruction Set Attributes Registers 5-7

The purpose of the Instruction Set Attributes Registers 5-7 are reserved, and they read as 0x00000000.

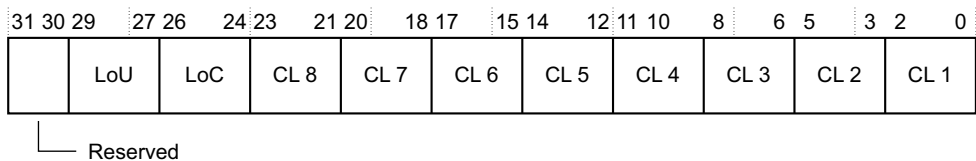
### 3.2.21 c0, Cache Level ID Register

The purpose of the Cache Level ID Register is to indicate the cache levels that are implemented. The register indicates the level of unification, LoU, and the level of coherency, LoC. For example, in the CortexA8 processor, the point where both data and instruction are unified is the Level 2 cache, therefore, the LoU is 3'b001. The point at which both data and instruction are coherent is the AMBA AXI interface, therefore, the LoC is 3'b010.

The Cache Level ID Register is:

- a read-only register common for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-16 shows the bit arrangement of the Cache Level ID Register.



**Figure 3-16 Cache Level ID Register format**

Table 3-37 shows how the bit values correspond with the Cache Level ID Register functions.

**Table 3-37 Cache Level ID Register bit functions**

Bits	Field	Function
[31:30]	-	Reserved, RAZ
[29:27]	LoU	3'b001 = level of unification
[26:24]	LoC	3'b010 = level of coherency
[23:21]	CL 8	3'b000 = no cache at <i>Cache Level</i> (CL) 8
[20:18]	CL 7	3'b000 = no cache at CL 7
[17:15]	CL 6	3'b000 = no cache at CL 6



**Table 3-37 Cache Level ID Register bit functions (continued)**

Bits	Field	Function
[14:12]	CL 5	3'b000 = no cache at CL 5
[11:9]	CL 4	3'b000 = no cache at CL 4
[8:6]	CL 3	3'b000 = no cache at CL 3
[5:3]	CL 2	3'b000 = no cache at CL 2 3'b100 = unified cache at CL 2
[2:0]	CL 1	3'b011 = separate instruction and data cache at CL 1

Table 3-38 shows the results of attempted access for each mode.

**Table 3-38 Results of access to the Cache Level ID Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Cache Level ID Register, read CP15 with:

```
MRC p15, 1, <Rd>, c0, c0, 1 ; Read Cache Level ID Register
```

### 3.2.22 c0, Silicon ID Register

The purpose of the Silicon ID Register is to enable software to identify the silicon manufacturer and revision. The reset value of this register is the **SILICONID[31:0]** input.

The Silicon ID Register is:

- a read-only register common for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-17 on page 3-54 shows the bit arrangement of the Silicon ID Register.

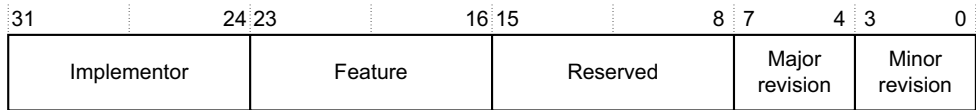


Figure 3-17 Silicon ID Register format

Table 3-39 shows how the bit values correspond with the Silicon ID Register functions.

Table 3-39 Silicon ID Register bit functions

Bits	Field	Function
[31:24]	Implementor	This field contains a code that identifies the silicon manufacturer. ARM assigns this code.
[23:16]	Feature	This field is implementation-defined.
[15:8]	-	Reserved, RAZ.
[7:4]	Major revision	This field is implementation-defined.
[3:0]	Minor revision	This field is implementation-defined.

Table 3-40 shows the results of attempted access for each mode.

Table 3-40 Results of access to the Silicon ID Register<sup>a</sup>

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Silicon ID Register, read CP15 with:

```
MRC p15, 1, <Rd>, c0, c0, 7 ; Read Silicon ID Register
```

### 3.2.23 c0, Cache Size Identification Registers

The purpose of these registers is to provide cache size information for up to eight levels of cache containing instruction, data, or unified caches. The processor contains L1 and L2 cache. The Cache Size Selection Register determines which Cache Size Identification Register to select.

The Cache Size Identification Registers are:

- read-only registers common for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-18 shows the bit arrangement of the Cache Size Identification Register.

31	30	29	28	27							13	12							2	0
W	W	R	W		NumSets								Associativity						Line Size	
T	B	A	A																	

**Figure 3-18 Cache Size Identification Register format**

Table 3-41 shows how the bit values correspond with the Cache Size Identification Register functions. See Table 3-42 on page 3-56 for valid bit field encodings.

**Table 3-41 Cache Size Identification Register bit functions**

Bits	Field	Function
[31]	WT	Indicates support available for write-through: 0 = write-through not supported 1 = write-through supported.
[30]	WB	Indicates support available for write-back: 0 = write-back not supported 1 = write-back supported.
[29]	RA	Indicates support available for read allocation: 0 = read allocation not supported 1 = read allocation supported.
[28]	WA	Indicates support available for write allocation: 0 = write allocation not supported 1 = write allocation supported.
[27:13]	NumSets	Indicates number of sets - 1.
[12:3]	Associativity	Indicates number of ways - 1.
[2:0]	LineSize	Indicates $(\log_2(\text{number of words in cache line})) - 2$ .

Table 3-42 shows the individual bit field and complete register encodings for the Cache Size Identification Register. Use this to match the cache size and level of cache set by the Cache Size Selection Register (CSSR). See *c0, Cache Size Selection Register* on page 3-57.

Table 3-42 Encodings of the Cache Size Identification Register

CSSR	Size	Complete register encoding	Register bit field encoding						
			WT	WB	RA	WA	NumSets	Associativity	LineSize
0x0	16KB	0xE007E01A	1	1	1	0	0x003F	0x3	0x2
	32KB	0xE00FE01A	1	1	1	0	0x007F	0x3	0x2
0x1	16KB	0x2007E01A	0	0	1	0	0x003F	0x3	0x2
	32KB	0x200FE01A	0	0	1	0	0x007F	0x3	0x2
0x2	0KB	0xF0000000	1	1	1	1	0x0000	0x0	0x0
	128KB	0xF01FE03A	1	1	1	1	0x00FF	0x7	0x2
	256KB	0xF03FE03A	1	1	1	1	0x01FF	0x7	0x2
	512KB	0xF07FE03A	1	1	1	1	0x03FF	0x7	0x2
	1024KB	0xF0FFE03A	1	1	1	1	0x07FF	0x7	0x2
0x3-0xF	-	0x0	Reserved						

Table 3-43 shows the results of attempted access for each mode.

Table 3-43 Results of access to the Cache Size Identification Register<sup>a</sup>

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Cache Size Identification Register, read CP15 with:

MRC p15, 1, <Rd>, c0, c0, 0; Cache Size Identification Register

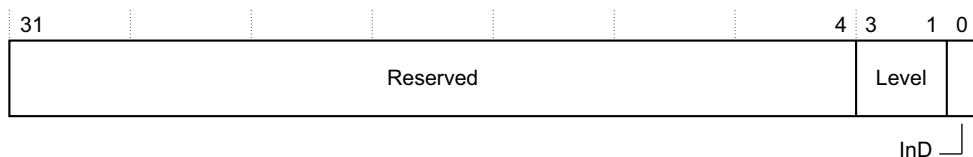
### 3.2.24 c0, Cache Size Selection Register

The purpose of the Cache Size Selection Register is to hold the value that the processor uses to select which Cache Size Identification Register to use.

The Cache Size Selection Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-19 shows the bit arrangement of the Cache Size Selection Register.



**Figure 3-19 Cache Size Selection Register format**

Table 3-44 shows how the bit values correspond with the Cache Size Selection Register functions.

**Table 3-44 Cache Size Selection Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. UNP, SBZ.
[3:1]	Level	Cache level selected 3'b000 = level 1 3'b001 = level 2 3'b010 - 3'b111 = reserved.
[0]	InD	Instruction (1) or Data/Unified (0).

Table 3-45 shows the results of attempted access for each mode.

**Table 3-45 Results of access to the Cache Size Selection Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure Data	Secure Data	Nonsecure Data	Nonsecure Data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Cache Size Selection Register, read CP15 with:

```
MRC p15, 2, <Rd>, c0, c0, 0 ; Cache Size Selection Register
```

### 3.2.25 c1, Control Register

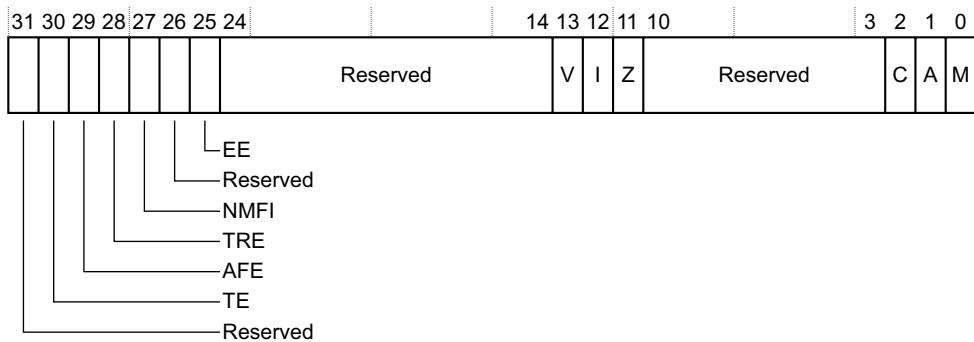
The purpose of the Control Register is to provide control and configuration of:

- memory alignment, endianness, protection, and fault behavior
- MMU, cache enables, and cache replacement strategy
- interrupts and behavior of interrupt latency
- location for exception vectors
- program flow prediction.

The Control Register is:

- a 32-bit read/write register
- accessible in privileged modes only
- partially banked.

Figure 3-20 shows the bit arrangement of the Control Register.



**Figure 3-20 Control Register bit assignments**

Table 3-46 shows how the bit values correspond with the Control Register functions.

**Table 3-46 Control Register bit functions**

Bits	Field	Access	Function
[31]	-	-	Reserved. UNP, SBZP.
[30]	TE	Banked	Thumb exception enable bit: 0 = Enables ARM exception generation. On exception entry, the CPSR T bit is 0 and J bit is 0. 1 = Enables Thumb exception generation. On exception entry, the CPSR T bit is 1 and J bit is 0. The primary input <b>CFGTE</b> defines the reset value.
[29]	AFE	Banked	This is the Access Flag Enable bit. It controls whether VMSAv7 redefines the AP[0] bit as an access flag or whether the software maintains binary compatibility with VMSAv6: 0 = AP[0] behavior defined, reset value 1 = access flag behavior defined. The TLB must be invalidated after changing the AFE bit.
[28]	TRE	Banked	This bit controls the TEX remap functionality in the MMU, see <i>MMU software-accessible registers</i> on page 6-8: 0 = TEX remap disabled. Normal ARMv6 or later behavior, reset value. 1 = TEX remap enabled. TEX[2:1] become translation table bits for OS.
[27]	NMFI	Read-only	This is the Non-Maskable Fast Interrupt enable bit. The reset value is determined by <b>CFGNMFI</b> . The pin cannot be configured by software: 0 = FIQ exceptions can be masked by software 1 = FIQ exceptions cannot be masked by software.
[26]	-	-	Reserved. RAZ, SBZP.
[25]	EE bit	Banked	Determines how the E bit in the CPSR is set on an exception: 0 = CPSR E bit is set to 0 on an exception 1 = CPSR E bit is set to 1 on an exception. The primary input <b>CFGEND0</b> defines the reset value of the EE bit.
[24:14]	-	-	This field returns 11'b01100010100 when read.
[13]	V bit	Banked	Determines the location of exception vectors, see <i>c12, Secure or Nonsecure Vector Base Address Register</i> on page 3-152. The primary input <b>VINITHI</b> defines the reset value of the V bit: 0 = Normal exception vectors selected, reset value. The Vector Base Address Registers determine the address range. 1 = High exception vectors selected, address range = 0xFFFF0000-0xFFFF001C.

Table 3-46 Control Register bit functions (continued)

Bits	Field	Access	Function
[12]	I bit	Banked	Determines if instructions can be cached in any instruction cache at any cache level:0 = instruction caching disabled at all levels, reset value 1 = instruction caching enabled.
[11]	Z bit	Banked	Enables program flow prediction: 0 = program flow prediction disabled, reset value 1 = program flow prediction enabled.
[10:7]	-	-	Reserved. RAZ, SBZP.
[6:3]	-	-	Reserved. <i>Read-As-One (RAO)</i> , <i>Should-Be-One or Preserved (SBOP)</i> .
[2]	C bit	Banked	Determines if data can be cached in a data or unified cache at any cache level:0 = data caching disabled at all levels, reset value 1 = data caching enabled.
[1]	A bit	Banked	Enables strict alignment of data to detect alignment faults in data accesses: 0 = strict alignment fault checking disabled, reset value 1 = strict alignment fault checking enabled.
[0]	M bit	Banked	Enables the MMU: 0 = MMU disabled, reset value 1 = MMU enabled.

Attempts to read or write the Control Register from secure or nonsecure User modes result in an Undefined Instruction exception.

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 2-46.

Table 3-47 shows the actions that result from attempted access for each mode.

Table 3-47 Results of access to the Control Register<sup>a</sup>

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure bit	Secure bit	Nonsecure bit	Nonsecure bit	Undefined	Undefined	Undefined	Undefined



- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Control Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c1, c0, 0 ; Read Control Register
MCR p15, 0, <Rd>, c1, c0, 0 ; Write Control Register
```

Table 3-48 shows the behavior of the processor caching instructions or data for the I bit and C bit of the *c1*, *Control Register* on page 3-58 and the L2EN bit of the *c1*, *Auxiliary Control Register*.

**Table 3-48 Behavior of the processor when enabling caches**

I bit	C bit	L2EN bit	Description
0	0	-	Instruction cache, data cache, L2 cache disabled for all instruction and data requests
0	1	0	Instruction cache disabled, data cache enabled, L2 cache disabled for all instruction and data requests
0	1	1	Instruction cache disabled, data cache enabled, L2 cache enabled for all instruction and data requests
1	0	-	Instruction cache enabled, data cache disabled, L2 cache disabled for all instruction and data requests
1	1	0	Instruction cache enabled, data cache enabled, L2 cache disabled for all instruction and data requests
1	1	1	Instruction cache, data cache, and L2 cache enabled for all requests

### 3.2.26 c1, Auxiliary Control Register

The purpose of the Auxiliary Control Register is to control processor-specific features that are not architecturally described.

The Auxiliary Control Register is:

- partially banked
- accessible in privileged modes only.

Figure 3-21 on page 3-62 shows the bit arrangement of the Auxiliary Control Register.

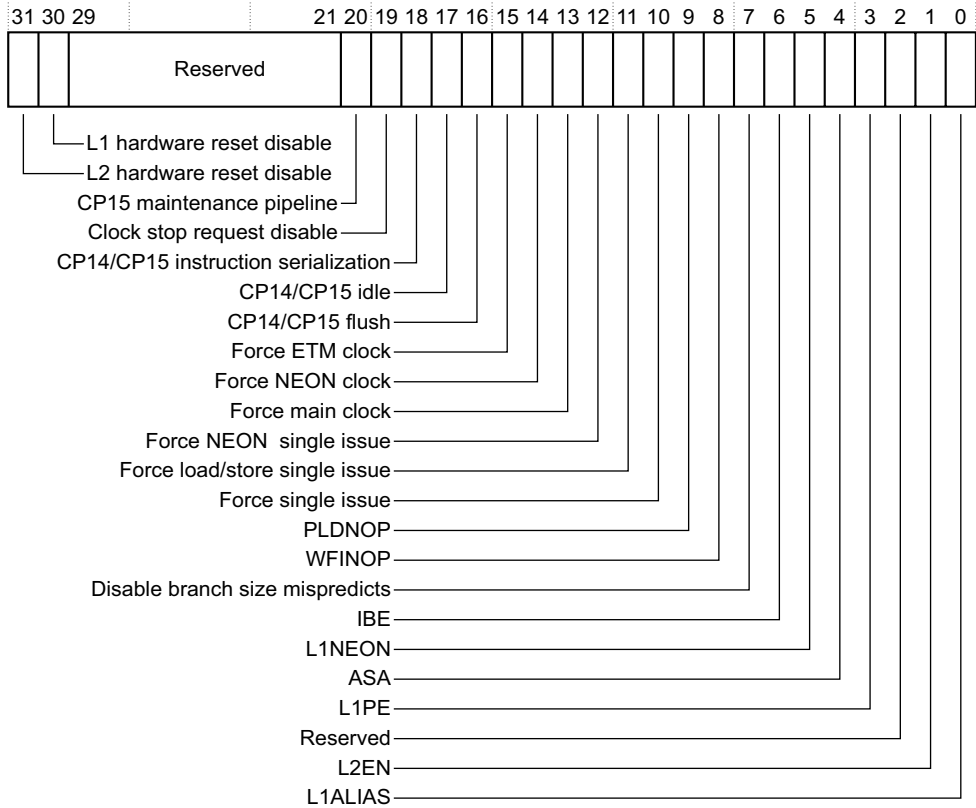


Figure 3-21 Auxiliary Control Register format

Table 3-49 shows how the bit values correspond with the Auxiliary Control Register functions.

**Table 3-49 Auxiliary Control Register bit functions**

Bits	Field	Security State		Function
		NS	S	
[31]	L2 hardware reset disable	RAZ	R	Monitors the L2 hardware reset disable bit, <b>L2RSTDISABLE</b> : 0 = the L2 valid RAM contents are reset by hardware 1 = the L2 valid RAM contents are not reset by hardware.
[30]	L1 hardware reset disable	RAZ	R	Monitors the L1 hardware reset disable bit, <b>L1RSTDISABLE</b> : 0 = the L1 valid RAM contents are reset by hardware 1 = the L1 valid RAM contents are not reset by hardware.
[29:21]	-	R	R/W	Reserved. UNP, SBZP.
[20]	Cache maintenance pipeline	R	R/W	Specify pipelining of CP15 data cache maintenance operations. CP15 data cache clean and invalidate or data cache invalidate operations can be executed by the processor in a pipelined fashion. 0 = pipelined cache maintenance operations, reset value 1 = non-pipelined cache maintenance operations.
[19]	Clock stop request disable	R	R/W	Disables <b>CLKSTOPREQ</b> : 0 = <b>CLKSTOPREQ</b> causes the processor to stop the internal clocks and to assert the <b>CLKSTOPACK</b> output, reset value 1 = disables the <b>CLKSTOPREQ</b> functionality.
[18]	CP14/CP15 instruction serialization	R	R/W	Some CP14 and CP15 instructions execute natively in a serial manner. This control bit imposes serialization on those CP14 and CP15 instructions that are not natively serialized: 0 = does not enforce serialization of CP14 or CP15 instructions, reset value 1 = enforces serialization of CP14 and CP15 instructions.

Table 3-49 Auxiliary Control Register bit functions (continued)

Bits	Field	Security State		Function
		NS	S	
[17]	CP14/CP15 wait on idle	R	R/W	Some CP14 or CP15 instructions that execute in a serial manner require that all outstanding memory accesses complete before execution of the instruction. This control bit imposes wait on idle protocol of CP14 and CP15 serialized instructions that do not natively wait on idle:  0 = does not enforce wait on idle of CP14 and CP15 instructions, reset value  1 = enforces wait on idle for serialized CP14 or CP15 instructions.
[16]	CP14/CP15 pipeline flush	R	R/W	After execution of some CP14 or CP15 instructions, the processor natively performs a pipeline flush before it executes the next instructions. This control bit imposes a pipeline flush on CP14 and CP15 instructions that do not natively include one:  0 = does not impose a pipeline flush on CP14 or CP15 instructions, reset value  1 = imposes a pipeline flush on CP14 and CP15 instructions.
[15]	Force ETM clock	R	R/W	Forces ETM clock enable active:  0 = does not prevent the processor clock generator from stopping the ETM clock, reset value  1 = prevents the processor clock generator from stopping the ETM clock.
[14]	Force NEON clock	R	R/W	Forces NEON clock enable active:  0 = does not prevent the processor clock generator from stopping the NEON clock, reset value  1 = prevents the processor clock generator from stopping the NEON clock.
[13]	Force main clock	R	R/W	Forces the main processor clock enable active:  0 = does not prevent the processor clock generator from stopping the main clock, reset value  1 = prevents the processor clock generator from stopping the main clock.

Table 3-49 Auxiliary Control Register bit functions (continued)

Bits	Field	Security State		Function
		NS	S	
[12]	Force NEON single issue	R	R/W	Forces single issue of Advanced SIMD instructions: 0 = does not force single issue of Advanced SIMD instructions, reset value 1 = forces single issue of Advanced SIMD instructions.
[11]	Force load/store single issue	R	R/W	Forces single issue of load/store instructions: 0 = does not force single issue of load/store instructions, reset value 1 = forces single issue of load/store instructions.
[10]	Force single issue	R	R/W	Forces single issue of all instructions: 0 = does not force single issue of all instructions, reset value 1 = forces single issue of all instructions.
[9]	PLDNOP	R	R/W	Executes PLD instructions as a NOP instruction: 0 = PLD instructions behave as defined in the <i>ARM Architecture Reference Manual</i> , reset value 1 = PLD instructions are executed as NOP instructions.  The PLD instruction acts as a hint to the memory system. If the PLDNOP is set to 0, the processor performs a memory access for the PLD instruction. If the PLDNOP is set to 1, the processor does not perform a memory access. See the <i>ARM Architecture Reference Manual</i> for more information on the PLD instruction.
[8]	WFINOP	R	R/W	Executes WFI instructions as a NOP instruction: 0 = executes WFI instructions as defined in the <i>ARM Architecture Reference Manual</i> , reset value 1 = executes WFI instructions as NOP instruction.  The WFI instruction places the processor in a low-power state and stops it from executing any more until an interrupt or a debug request occurs. If the WFINOP is set to 0, then the WFI instruction places the processor in a low-power state. If the WFINOP is set to 1, the WFI instruction is executed as a NOP and does not place the processor in a low-power state. See the <i>ARM Architecture Reference Manual</i> for more information on the WFI instruction.

Table 3-49 Auxiliary Control Register bit functions (continued)

Bits	Field	Security State		Function
		NS	S	
[7]	Disable branch size mispredicts	R	R/W	Prevents BTB branch size mispredicts: 0 = enables BTB branch size mispredicts, reset value 1 = executes the CP15 Invalidate All and Invalidate by MVA instructions as specified and prevents BTB branch size mispredicts.
[6]	IBE	R	R/W	Invalidates BTB enable: 0 = executes the CP15 Invalidate All and Invalidate by MVA instructions as a NOP instruction, reset value 1 = executes the CP15 Invalidate All and Invalidate by MVA instructions as specified.
[5]	L1NEON	R	R/W	Enables caching NEON data within the L1 data cache: 0 = disables caching NEON data within the L1 data cache, reset value 1 = enables caching NEON data within the L1 data and L2 cache.  <p style="text-align: center;">———— <b>Note</b> ————</p> NEON L1 caching should be enabled for best performance when the L2 cache is off or not present.
[4]	ASA	R	R/W	Enables speculative accesses on AXI: 0 = disables speculative accesses, reset value 1 = enables speculative accesses.
[3]	L1PE	R	R/W	Enables L1 cache parity detection: 0 = L1 cache parity disabled for both instruction and data caches, reset value 1 = L1 cache parity enabled.

**Table 3-49 Auxiliary Control Register bit functions (continued)**

Bits	Field	Security State		Function
		NS	S	
[2]	-	R	R/W	Reserved. UNP, SBZ.
[1]	L2EN	B	B	Enables L2 cache: 0 = L2 cache disabled 1 = L2 cache enabled. See Table 3-48 on page 3-61 for details.
[0]	L1ALIAS	R	R/W	Enables L1 data cache hardware alias checks: 0 = L1 data cache hardware alias support enabled, reset value 1 = L1 data cache hardware alias support disabled.

Table 3-50 shows the results of attempted access for each mode.

**Table 3-50 Results of access to the Auxiliary Control Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Data	Data	Banked Data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Auxiliary Control Register you must use a read modify write technique. To access the Auxiliary Control Register, read or write CP15 with:

MRC p15, 0, <Rd>, c1, c0, 1 ; Read Auxiliary Control Register  
MCR p15, 0, <Rd>, c1, c0, 1 ; Write Auxiliary Control Register

### 3.2.27 c1, Coprocessor Access Control Register

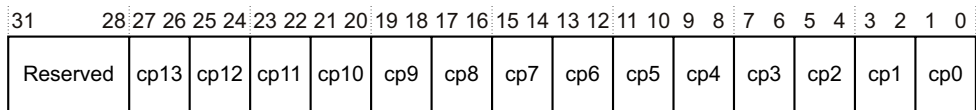
The purpose of the Coprocessor Access Control Register is to set access rights for the coprocessors CP0 through CP13. This register has no effect on access to CP14, the debug control coprocessor, or CP15, the system control coprocessor. This register also provides a means for software to determine if any particular coprocessor, CP0-CP13, exists in the system.

The Coprocessor Access Control Register is:

- a read/write register common to Secure and Nonsecure states

- accessible in privileged modes only.

Figure 3-22 shows the bit arrangement of the Coprocessor Access Control Register.



**Figure 3-22 Coprocessor Access Control Register format**

Table 3-51 shows how the bit values correspond with the Coprocessor Access Control Register functions.

**Table 3-51 Coprocessor Access Control Register bit functions**

Bits	Field	Function
[31:28]	-	Reserved. UNP, SBZP.
-	cp<n> <sup>a</sup>	Defines access permissions for each coprocessor. Access denied is the reset condition and is the behavior for nonexistent coprocessors: b00 = Access denied, reset value. Attempted access generates an Undefined Instruction exception. b01 = Privileged mode access only. b10 = Reserved. b11 = Privileged and User mode access.

a. n is the coprocessor number between 0 and 13.

Access to coprocessors in the Nonsecure state depends on the permissions set in the *c1*, *Nonsecure Access Control Register* on page 3-73.



Attempts to read or write the Coprocessor Access Control Register access bits depend on the corresponding bit for each coprocessor in *c1*, *Nonsecure Access Control Register* on page 3-73. Table 3-52 shows the results of attempted access to coprocessor access bits for each mode.

**Table 3-52 Results of access to the Coprocessor Access Control Register<sup>a</sup>**

Nonsecure Access Control Register bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	b00	Ignored	Undefined		Undefined	
1	Data	Data	Data	Data				

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Coprocessor Access Control Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c1, c0, 2 ; Read Coprocessor Access Control Register
MCR p15, 0, <Rd>, c1, c0, 2 ; Write Coprocessor Access Control Register
```

You must execute an *Instruction Memory Barrier* (IMB) sequence immediately after an update of the Coprocessor Access Control Register, see Memory Barriers in the *ARM Architecture Reference Manual*. You must not attempt to execute any instructions that are affected by the change of access rights between the IMB sequence and the register update.

To determine if any particular coprocessor exists in the system, write the access bits for the coprocessor of interest with a value other than b00. If the coprocessor does not exist in the system the access rights remain set to b00.

#### Note

- For the processor, there is a direct relationship between the **CPEXIST[13:0]** inputs and the Coprocessor Access Control Register bits cp13-cp01.

Each **CPEXIST** input represents the existence of a coprocessor that you use to enable a particular coprocessor. If the appropriate **CPEXIST** input is set to a:

- logical 0, access is denied to that coprocessor or reset state as defined by the register
- logical 1, then you can reprogram that coprocessor.

- You must enable the Coprocessor Access Control Register before accessing any NEON or VFP system register.
- You must set **CPEXIST[11:10]** to b11 to use the NEON or VFP coprocessor. All other **CPEXIST** bits must be set to 0.
- You must set **CPEXIST[11:10]** to b00 if you configure the processor without the NEON coprocessor.

### 3.2.28 c1, Secure Configuration Register

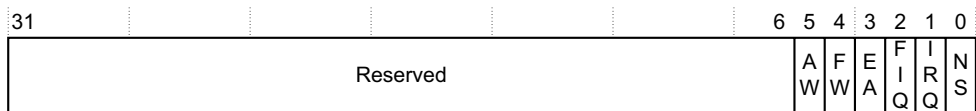
The purpose of the Secure Configuration Register is to define:

- the current state of the processor as Secure or Nonsecure states
- in which state the core executes exceptions
- the ability to modify the A and I bits in the CPSR in the Nonsecure state.

The Secure Configuration Register is:

- a read/write register
- accessible in secure privileged modes only.

Figure 3-23 shows the bit arrangement of the Secure Configuration Register.



**Figure 3-23 Secure Configuration Register format**

Table 3-53 shows how the bit values correspond with the Secure Configuration Register functions.

**Table 3-53 Secure Configuration Register bit functions**

Bits	Field	Function
[31:7]	-	Reserved. UNP, SBZP.
[6]	-	Reserved, RAZ.
[5]	AW	Determines if the A bit in the CPSR can be modified when in the Nonsecure state: 0 = disable modification of the A bit in the CPSR in the Nonsecure state, reset value 1 = enable modification of the A bit in the CPSR in the Nonsecure state.

**Table 3-53 Secure Configuration Register bit functions (continued)**

Bits	Field	Function
[4]	FW	Determines if the F bit in the CPSR can be modified when in the Nonsecure state: 0 = disable modification of the F bit in the CPSR in the Nonsecure state, reset value 1 = enable modification of the F bit in the CPSR in the Nonsecure state.
[3]	EA	Determines External Abort behavior for Secure and Nonsecure states: 0 = branch to abort mode on an External Abort exception, reset value 1 = branch to Monitor mode on an External Abort exception.
[2]	FIQ	Determines FIQ behavior for Secure and Nonsecure states: 0 = branch to FIQ mode on an FIQ exception, reset value 1 = branch to Monitor mode on an FIQ exception.
[1]	IRQ	Determines IRQ behavior for Secure and Nonsecure states: 0 = branch to IRQ mode on an IRQ exception, reset value 1 = branch to Monitor mode on an IRQ exception.
[0]	NS bit	Defines the operation of the processor: 0 = secure, reset value 1 = nonsecure.

**Note**

When the core runs in Monitor mode the state is considered secure regardless of the state of the NS bit.

The permutations of the bits in the Secure Configuration Register have certain security implications. Table 3-54 shows the results for combinations of the FW and FIQ bits.

**Table 3-54 Operation of the FW and FIQ bits**

FW	FIQ	Function
1	0	FIQs handled locally
0	1	FIQs can be configured to give deterministic secure interrupts
1	1	Nonsecure state able to make denial of service attack, avoid use of this function
0	0	For Nonsecure state, avoid because the core might enter an infinite loop for nonsecure FIQ

Table 3-55 shows the results for combinations of the AW and EA bits.

**Table 3-55 Operation of the AW and EA bits**

AW	EA	Function
1	0	Aborts handled locally
0	1	All external aborts trapped to Monitor mode
1	1	All external imprecise Data Aborts trapped to Monitor mode but the Nonsecure state can hide secure aborts from the Monitor, avoid use of this function
0	0	For Nonsecure state, avoid this because the core can unexpectedly enter an abort mode

To access the Secure Configuration Register, read or write CP15 with:

MRC p15, 0, <Rd>, c1, c1, 0 ; Read Secure Configuration Register data

MCR p15, 0, <Rd>, c1, c1, 0 ; Write Secure Configuration Register data

An attempt to access the Secure Configuration Register from any state other than secure privileged results in an Undefined Instruction exception.

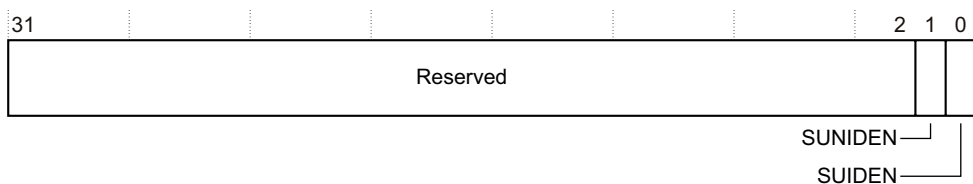
### 3.2.29 c1, Secure Debug Enable Register

The purpose of the Secure Debug Enable Register is to provide control of permissions for debug in secure User mode. See Chapter 12 *Debug* for more details.

The Secure Debug Enable Register is:

- a register in the Secure state only
- accessible in secure privileged modes only.

Figure 3-24 shows the bit arrangement of the Secure Debug Enable Register.



**Figure 3-24 Secure Debug Enable Register format**

Table 3-56 shows how the bit values correspond with the Secure Debug Enable Register functions.

**Table 3-56 Secure Debug Enable Register bit functions**

Bits	Field	Function
[31:2]	-	Reserved. UNP, SBZP.
[1]	SUNIDEN	Enables secure User noninvasive debug: 0 = noninvasive debug is not permitted in secure User mode, reset value 1 = noninvasive debug is permitted in secure User mode.
[0]	SUIDEN	Enables secure User invasive debug: 0 = invasive debug is not permitted in secure User mode, reset value 1 = invasive debug is permitted in secure User mode.

Table 3-57 shows the results of attempted access for each mode.

**Table 3-57 Results of access to the Secure Debug Enable Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Secure Debug Enable Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c1, c1, 1 ; Read Secure Debug Enable Register
MCR p15, 0, <Rd>, c1, c1, 1 ; Write Secure Debug Enable Register
```

### 3.2.30 c1, Nonsecure Access Control Register

The purpose of the Nonsecure Access Control Register is to define the nonsecure access permission for:

- coprocessors
- internal PLE.

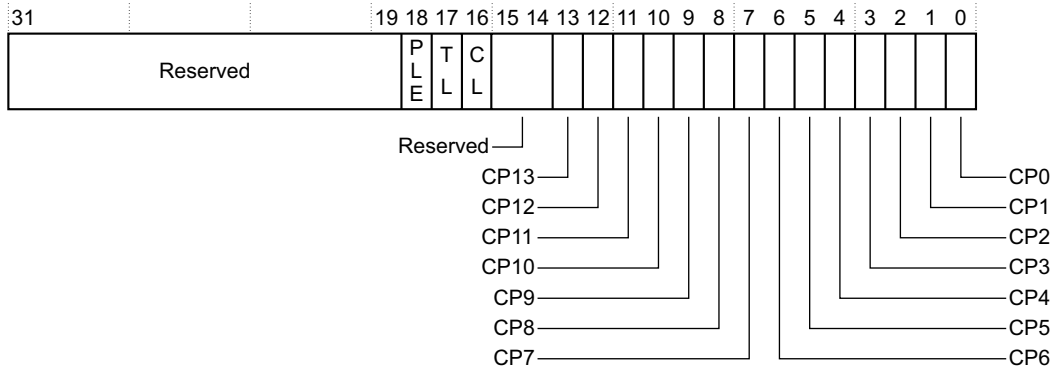
———— **Note** —————

This register has no effect on nonsecure access permissions for the debug control coprocessor, CP14, or the system control coprocessor, CP15.

The Nonsecure Access Control Register is:

- a read/write register in the Secure state
- a read-only register in the Nonsecure state
- only accessible in privileged modes.

Figure 3-25 shows the bit arrangement of the Nonsecure Access Control Register.



**Figure 3-25 Nonsecure Access Control Register format**

Table 3-58 shows how the bit values correspond with the Nonsecure Access Control Register functions.

**Table 3-58 Nonsecure Access Control Register bit functions**

Bits	Field	Function
[31:19]	-	Reserved. UNP, SBZP.
[18]	PLE	Determines if an access to PLE registers is permitted in Nonsecure state: 0 = PLE registers cannot be used in Nonsecure state 1 = PLE registers can be accessed in both Secure and Nonsecure state. Nonsecure translation tables are used for address translation when the PLE bit is set to 1.
[17]	TL	Determines if lockable translation table entries can be allocated in Nonsecure state: 0 = lockable TLB entries cannot be allocated, reset 1 = lockable TLB entries can be allocated.

**Table 3-58 Nonsecure Access Control Register bit functions (continued)**

Bits	Field	Function
[16]	CL	Determines if lockdown entries can be allocated within the L2 cache in Nonsecure state: 0 = entries cannot be allocated, reset value 1 = entries can be allocated. If CL is set to 0, then any L2 cache lockdown operation takes an Undefined Instruction exception.
[15:14]	-	Reserved. UNP, SBZ.
[13:0]	CP<n>	Determines permission to access the given coprocessor in the Nonsecure state, <n> is the number of coprocessor from 0 to 13: 0 = secure access only, reset value 1 = secure or nonsecure access.

To access the Nonsecure Access Control Register, read or write CP15 with:

MRC p15, 0, <Rd>, c1, c1, 2 ; Read Nonsecure Access Control Register data  
MCR p15, 0, <Rd>, c1, c1, 2 ; Write Nonsecure Access Control Register data

Table 3-59 shows the results of attempted access for each mode.

**Table 3-59 Results of access to the Auxiliary Control Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

### 3.2.31 c2, Translation Table Base Register 0

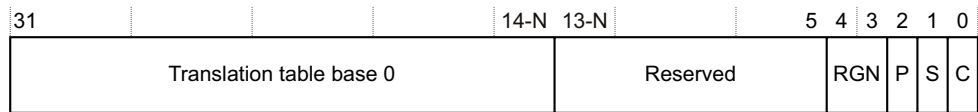
The purpose of the Translation Table Base Register 0 is to hold the physical address of the first level translation table.

You use Translation Table Base Register 0 for process-specific addresses, where each process maintains a separate first level translation table. On a context switch you must modify both Translation Table Base Register 0 and the Translation Table Base Control Register, if appropriate.

The Translation Table Base Register 0 is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-26 shows the bit arrangement of the Translation Table Base Register 0.



**Figure 3-26 Translation Table Base Register 0 format**

Table 3-60 shows how the bit values correspond with the Translation Table Base Register 0 functions.

**Table 3-60 Translation Table Base Register 0 bit functions**

Bits	Field	Function
[31:14-N] <sup>a</sup>	Translation table base 0	Holds the translation table base address, the physical address of the first level translation table.
[13-N:5] <sup>a</sup>	-	Reserved. RAZ, SBZ.
[4:3]	RGN	Indicates the outer cacheable attributes for translation table walking: b00 = outer noncacheable b01 = write-back, write allocate b10 = write-through, no allocate on write b11 = write-back, no allocate on write.
[2]	P	Read-As-Zero and ignore writes. This bit is not implemented on this processor.
[1]	S	Indicates the translation table walk is to nonshared or to shared memory: 0 = nonshared 1 = shared.
[0]	C	Indicates the translation table walk is inner cacheable or inner noncacheable: 0 = inner noncacheable 1 = inner cacheable.

a. For an explanation of N, see *c2, Translation Table Base Control Register* on page 3-79.

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is **HIGH** result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 2-46.



Table 3-61 shows the results of attempted access for each mode.

**Table 3-61 Results of access to the Translation Table Base Register 0<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

A write to the Translation Table Base Register 0 updates the address of the first level translation table from the value in bits [31:7] of the written value, to account for the maximum value of 7 for N. The number of bits of this address that the processor uses, and the required alignment of the first level translation table, depends on the value of N, see *c2, Translation Table Base Control Register* on page 3-79.

A read from the Translation Table Base Register 0 returns the complete address of the first level translation table in bits [31:7] of the read value, regardless of the value of N.

To access the Translation Table Base Register 0, read or write CP15 c2 with:

```
MRC p15, 0, <Rd>, c2, c0, 0 ; Read Translation Table Base Register
MCR p15, 0, <Rd>, c2, c0, 0 ; Write Translation Table Base Register
```

#### ————— Note —————

The processor cannot perform a translation table walk from L1 cache. Therefore, if C is set to 1, to ensure coherency, you must store translation tables in inner write-through memory. If you store the translation tables in an inner write-back memory region, you must clean the appropriate cache entries after modification so that the mechanism for the hardware translation table walks sees them.

### 3.2.32 c2, Translation Table Base Register 1

The purpose of the Translation Table Base Register 1 is to hold the physical address of the first level table. The expected use of the Translation Table Base Register 1 is for OS and I/O addresses.

The Translation Table Base Register 1 is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-27 shows the bit arrangement of the Translation Table Base Register 1.

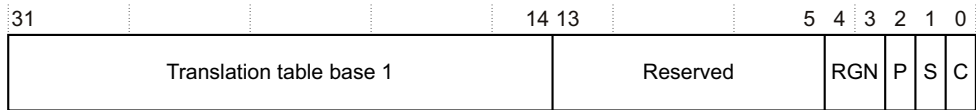
**Figure 3-27 Translation Table Base Register 1 format**

Table 3-62 shows how the bit values correspond with the Translation Table Base Register 1 functions.

**Table 3-62 Translation Table Base Register 1 bit functions**

Bits	Field	Function
[31:14]	Translation table base 1	Holds the translation table base address, the physical address of the first level translation table.
[13:5]	-	Reserved. RAZ, SBZ.
[4:3]	RGN	Indicates the outer cacheable attributes for translation table walking: b00 = outer noncacheable b01 = write-back, write allocate b10 = write-through, no allocate on write b11 = write-back, no allocate on write.
[2]	P	Reserved, RAZ and ignore writes. This bit is not implemented on this processor.
[1]	S	Indicates the translation table walk is to nonshared or to shared memory: 0 = nonshared 1 = shared.
[0]	C	Indicates the translation table walk is inner cacheable or inner noncacheable: 0 = inner noncacheable 1 = inner cacheable.

Table 3-63 shows the results of attempted access for each mode.

**Table 3-63 Results of access to the Translation Table Base Register 1<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

A write to the Translation Table Base Register 1 updates the address of the first level translation table from the value in bits [31:14] of the written value. Bits [13:5] *Should-Be-Zero*. The Translation Table Base Register 1 must reside on a 16KB page boundary.

To access the Translation Table Base Register 1, read or write CP15 with:

```
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
```

#### ————— Note —————

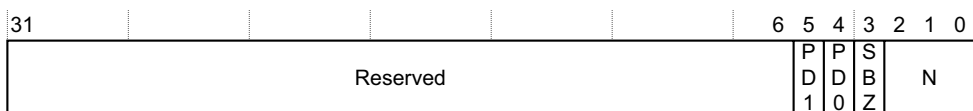
The processor cannot perform a translation table walk from L1 cache. Therefore, if C is set to 1, to ensure coherency, you must store translation tables in inner write-through memory. If you store the translation tables in an inner write-back memory region, you must clean the appropriate cache entries after modification so that the mechanism for the hardware translation table walks sees them.

### 3.2.33 c2, Translation Table Base Control Register

The purpose of the Translation Table Base Control Register is to determine if a translation table miss for a specific VA uses, for its translation table walk, either:

- Translation Table Base Register 0. The recommended use is for task-specific addresses
- Translation Table Base Register 1. The recommended use is for operating system and I/O addresses.

Figure 3-28 on page 3-79 shows the bit arrangement of the Translation Table Base Control Register.



**Figure 3-28 Translation Table Base Control Register format**

Table 3-64 shows how the bit values correspond with the Translation Table Base Control Register functions.

**Table 3-64 Translation Table Base Control Register bit functions**

Bits	Field	Function
[31:6]	-	Reserved. UNP, SBZ.
[5]	PD1	Specifies occurrence of a translation table walk on a TLB miss when using Translation Table Base Register 1. When translation table walk is disabled, a section translation fault occurs instead on a TLB miss: 0 = The processor performs a translation table walk on a TLB miss, with secure or nonsecure privilege appropriate to the current Secure or Nonsecure state. This is the reset value. 1 = The processor does not perform a translation table walk. If a TLB miss occurs with Translation Table Base Register 1 in use, the processor returns a section translation fault.
[4]	PD0	Specifies occurrence of a translation table walk on a TLB miss when using Translation Table Base Register 0. When translation table walk is disabled, a section translation fault occurs instead on a TLB miss: 0 = The processor performs a translation table walk on a TLB miss, with secure or nonsecure privilege appropriate to the current Secure or Nonsecure state. This is the reset value. 1 = The processor does not perform a translation table walk. If a TLB miss occurs with Translation Table Base Register 0 in use, the processor returns a section translation fault.
[3]	-	Reserved. UNP, SBZ.
[2:0]	N	Specifies the boundary size of Translation Table Base Register 0: b000 = 16KB, reset value b001 = 8KB b010 = 4KB b011 = 2KB b100 = 1KB b101 = 512B b110 = 256B b111 = 128B.

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is **HIGH** result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 2-46.

Table 3-65 shows the results of attempted access for each mode.

**Table 3-65 Results of access to the Translation Table Base Control Register**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined <sup>a</sup>	Undefined	Undefined	Undefined

a. The access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Translation Table Base Control Register, read or write CP15 with:

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

A translation table base register is selected in the following fashion:

- If N is set to 0, always use Translation Table Base Register 0. This is the default case at reset. It is backwards compatible with ARMv5 and earlier processors.
- If N is set to a value greater than 0, and bits [31:32-N] of the VA are all zeros, use Translation Table Base Register 0. Otherwise, use Translation Table Base Register 1. N must be in the range 0-7.

#### ———— Note ————

The processor cannot perform a translation table walk from L1 cache. Therefore, if C is set to 1, to ensure coherency, you must store translation tables in inner write-through memory. If you store the translation tables in an inner write-back memory region, you must clean the appropriate cache entries after modification so that the mechanism for the hardware translation table walks sees them.

### 3.2.34 c3, Domain Access Control Register

The purpose of the Domain Access Control Register is to hold the access permissions for a maximum of 16 domains.

The Domain Access Control Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-29 shows the bit arrangement of the Domain Access Control Register.

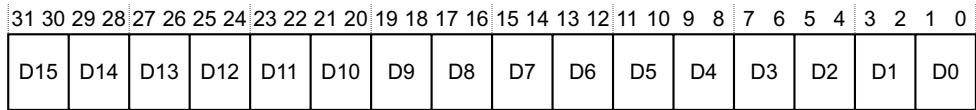


Figure 3-29 Domain Access Control Register format

Table 3-66 shows how the bit values correspond with the Domain Access Control Register functions.

Table 3-66 Domain Access Control Register bit functions

Bits	Field	Function
-	D<n> <sup>a</sup>	<p>The fields D15-D0 in the register define the access permissions for each one of the 16 domains. These domains can be either sections, large pages, or small pages of memory:</p> <p>b00 = No access. Any access generates a domain fault.</p> <p>b01 = Client. Accesses are checked against the access permission bits in the TLB entry.</p> <p>b10 = Reserved. Any access generates a domain fault.</p> <p>b11 = Manager. Accesses are not checked against the access permission bits in the TLB entry, so a permission fault cannot be generated. Attempting to execute code in a page that has the TLB <i>eXecute Never</i> (XN) attribute set does not generate an abort.</p>

a. n is the Domain number in the range between 0 and 15

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 2-46.

Table 3-67 shows the results of attempted access for each mode.

Table 3-67 Results of access to the Domain Access Control Register<sup>a</sup>

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Domain Access Control Register, read or write CP15 with:

MRC p15, 0, <Rd>, c3, c0, 0 ; Read Domain Access Control Register

MCR p15, 0, <Rd>, c3, c0, 0 ; Write Domain Access Control Register

### 3.2.35 c5, Data Fault Status Register

The purpose of the *Data Fault Status Register* (DFSR) is to hold the source of the last data fault.

The Data Fault Status Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-30 shows the bit arrangement of the Data Fault Status Register when the data abort is not imprecise. When the data abort is imprecise, only bits [3:0] are valid.



**Figure 3-30 Data Fault Status Register format**

Table 3-68 shows how the bit values correspond with the Data Fault Status Register functions.

**Table 3-68 Data Fault Status Register bit functions**

Bits	Field	Function
[31:13]	-	Reserved. UNP, SBZ.
[12]	SD	Indicates whether an AXI Decode or Slave error caused an abort. This bit is only valid for external aborts. For all other aborts this bit <i>Should-Be-Zero</i> : 0 = AXI Decode error caused the abort, reset value 1 = AXI Slave error caused the abort.
[11]	RW	Indicates whether a read or write access caused an abort: 0 = read access caused the abort, reset value 1 = write access caused the abort.
[10]	S	Part of the Status field. See bits [3:0] in this table. The reset value is 0.

**Table 3-68 Data Fault Status Register bit functions (continued)**

Bits	Field	Function
[9:8]	-	Reserved, RAZ and ignore writes.
[7:4]	Domain	Indicates which one of the 16 domains, D15-D0, is accessed when a data fault occurs. This field takes values 0-15.
[3:0]	Status	<p>Indicates the type of exception generated. To determine the data fault, bits [12] and [10] must be used in conjunction with bits [3:0]. The following encodings are listed in priority order, highest first:</p> <ul style="list-style-type: none"> <li>• b000001 alignment fault</li> <li>• b000100 instruction cache maintenance fault</li> <li>• bx01100 L1 translation, precise external abort</li> <li>• bx01110 L2 translation, precise external abort</li> <li>• b011100 L1 translation precise parity error</li> <li>• b011110 L2 translation precise parity error</li> <li>• b000101 translation fault, section</li> <li>• b000111 translation fault, page</li> <li>• b000011 access flag fault, section</li> <li>• b000110 access flag fault, page</li> <li>• b001001 domain fault, section</li> <li>• b001011 domain fault, page</li> <li>• b001101 permission fault, section</li> <li>• b001111 permission fault, page</li> <li>• bx01000 precise external abort, nontranslation</li> <li>• bx10110 imprecise external abort</li> <li>• b011000 imprecise error, parity or ECC</li> <li>• b000010 debug event.</li> </ul> <p>Any unused encoding not listed is reserved.            Where <i>x</i> represents bit [12] in the encoding, bit [12] can be either:            0 = AXI Decode error caused the abort, reset value            1 = AXI Slave error caused the abort.</p>

---

**Note**

---

When the SCR EA bit is set to 1, see *c1, Secure Configuration Register* on page 3-70, the processor writes to the Secure Data Fault Status Register on a Monitor entry caused by an external abort.

---

To access the Data Fault Status Register, read or write CP15 with:



MRC p15, 0, <Rd>, c5, c0, 0 ; Read Data Fault Status Register  
MCR p15, 0, <Rd>, c5, c0, 0 ; Write Data Fault Status Register

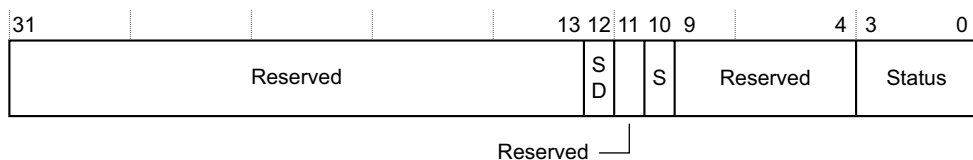
### 3.2.36 c5, Instruction Fault Status Register

The purpose of the *Instruction Fault Status Register* (IFSR) is to hold the source of the last instruction fault.

The Instruction Fault Status Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-31 shows the bit arrangement of the Instruction Fault Status Register.



**Figure 3-31 Instruction Fault Status Register format**

Table 3-69 shows how the bit values correspond with the Instruction Fault Status Register functions.

**Table 3-69 Instruction Fault Status Register bit functions**

Bits	Field	Function
[31:13]	-	Reserved. UNP, SBZ.
[12]	SD	Indicates whether an AXI Decode or Slave error caused an abort. This bit is only valid for external aborts. For all other aborts this bit Should-Be-Zero: 0 = AXI Decode error caused the abort, reset value 1 = AXI Slave error caused the abort.
[11]	-	Reserved. UNP, SBZ.

**Table 3-69 Instruction Fault Status Register bit functions (continued)**

Bits	Field	Function
[10]	S	Part of the Status field. See bits [3:0] in this table.
[9:4]	-	Reserved. UNP, SBZ.
[3:0]	Status	<p>Indicates the type of exception generated. To determine the data fault, bits [12] and [10] must be used in conjunction with bits [3:0]. The following encodings are listed in priority order, highest first:</p> <ul style="list-style-type: none"> <li>• bx01100 L1 translation, precise external abort</li> <li>• bx01110 L2 translation, precise external abort</li> <li>• b011100 L1 translation precise parity error</li> <li>• b011110 L2 translation precise parity error</li> <li>• b000101 translation fault, section</li> <li>• b000111 translation fault, page</li> <li>• b000011 access flag fault, section</li> <li>• b000110 access flag fault, page</li> <li>• b001001 domain fault, section</li> <li>• b001011 domain fault, page</li> <li>• b001101 permission fault, section</li> <li>• b001111 permission fault, page</li> <li>• bx01000 precise external abort, nontranslation</li> <li>• b011001 precise parity error</li> <li>• b000010 debug event.</li> </ul> <p>Any unused encoding not listed is reserved.</p> <p>Where <i>x</i> represents bit [12] in the encoding, bit [12] can be either:</p> <p>0 = AXI Decode error caused the abort, reset value</p> <p>1 = AXI Slave error caused the abort.</p>

———— **Note** ————

When the SCR EA bit is set to 1, see *c1, Secure Configuration Register* on page 3-70, the processor writes to the Secure Instruction Fault Status Register on a Monitor entry caused by an external abort.

To access the Instruction Fault Status Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c5, c0, 1 ; Read Instruction Fault Status Register
MCR p15, 0, <Rd>, c5, c0, 1 ; Write Instruction Fault Status Register
```

### 3.2.37 c5, Auxiliary Fault Status Registers

The Auxiliary Fault Status Register is provided for compatibility with all ARMv7-A designs. This is true for both the instruction and data auxiliary FSR. The processor always reads this as RAZ. All writes are ignored.

The Auxiliary Fault Status Register is:

- a read-only register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Table 3-70 shows the results of attempted access for each mode.

**Table 3-70 Results of access to the Auxiliary Fault Status Registers<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Auxiliary Fault Status Registers, read or write CP15 with:

```
MRC p15, 0, <Rd>, c5, c1, 0; Read Data Auxiliary Fault Status Register
MCR p15, 0, <Rd>, c5, c1, 0; Write Data Auxiliary Fault Status Register
MRC p15, 0, <Rd>, c5, c1, 1; Read Instruction Auxiliary Fault Status Register
MCR p15, 0, <Rd>, c5, c1, 1; Write Instruction Auxiliary Fault Status Register
```

There is no physical register for Auxiliary Data Fault Status Register or Auxiliary Instruction Fault Status Register as the register is always RAZ.

### 3.2.38 c6, Data Fault Address Register

The purpose of the *Data Fault Address Register* (DFAR) is to hold the *Modified Virtual Address* (MVA) of the fault when a precise abort occurs.

The DFAR is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

The Data Fault Address Register bits [31:0] contain the MVA where the precise abort occurred.

Table 3-71 shows the results of attempted access for each mode.

**Table 3-71 Results of access to the Data Fault Address Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the DFAR, read or write CP15 with:

MRC p15, 0, <Rd>, c6, c0, 0 ; Read Data Fault Address Register

MCR p15, 0, <Rd>, c6, c0, 0 ; Write Data Fault Address Register

A write to this register sets the DFAR to the value of the data written. This is useful for a debugger to restore the value of the DFAR.

The processor also updates the DFAR on debug exception entry because of watchpoints. See *Effect of debug exceptions on CP15 registers and WFAR* on page 12-75 for more information.

### 3.2.39 c6, Instruction Fault Address Register

The purpose of the *Instruction Fault Address Register* (IFAR) is to hold the address of instructions that cause a prefetch abort.

The IFAR is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

The Instruction Fault Address Register bits [31:1] contain the instruction fault MVA and bit [0] is RAZ.

Table 3-72 shows the results of attempted access for each mode.

**Table 3-72 Results of access to the Instruction Fault Address Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the IFAR, read or write CP15 with:

```
MRC p15, 0, <Rd>, c6, c0, 2 ; Read Instruction Fault Address Register
MCR p15, 0, <Rd>, c6, c0, 2 ; Write Instruction Fault Address Register
```

A write to this register sets the IFAR to the value of the data written. This is useful for a debugger to restore the value of the IFAR.

### 3.2.40 c7, Cache operations

The purpose of c7 is to manage the associated cache levels. The maintenance operations are formed into two management groups:

- Set and way:
  - clean
  - invalidate
  - clean and invalidate.
- MVA:
  - clean
  - invalidate
  - clean and invalidate.

In addition, the maintenance operations use the following definitions:

#### Point of coherency

The time when the imposition of any more cache becomes transparent for instruction, data, and translation table walk accesses to that address by any processor in the system.

## Point of unification

The time when the instruction and data caches, and the TLB translation table walks have merged for a uniprocessor system.

---

### Note

---

- Reading from *c7*, except for reads from the *Physical Address Register (PAR)*, causes an Undefined Instruction exception.
  - All accesses to *c7* can only be executed in a privileged mode of operation, except Data Synchronization Barrier, Flush Prefetch Buffer, and Data Memory Barrier. These can be executed in User mode. Attempting to execute a privileged instruction in User mode results in an Undefined Instruction exception.
  - For information on the behavior of the invalidate, clean, and prefetch operations in the secure and nonsecure operations, see the *ARM Architecture Reference Manual*.
- 

## Data formats for the cache operations

The possible formats for the data supplied to the cache maintenance and prefetch buffer operations depend on the specific operation:

- *Set and way* on page 3-91
- *MVA* on page 3-92
- *SBZ* on page 3-93.

Table 3-73 shows the data value supplied to each cache maintenance and prefetch buffer operations. See also *Coprocessor instructions* on page 16-12 for the effect on these operations of the setting of bit [20] of the Auxiliary Control Register.

**Table 3-73 Register *c7* cache and prefetch buffer maintenance operations**

CRm	Opcode_2	Function	Data
c5	0	Invalidate all instruction caches to PoU. Also flushes branch target cache. <sup>a</sup>	SBZ
c5	1	Invalidate instruction cache line by MVA to PoC.	MVA
c5	4	Prefetch flush. The prefetch buffer is flushed. <sup>b</sup>	SBZ
c5	6	Invalidate entire branch predictor array.	SBZ
c5	7	Invalidate MVA from branch predictor array	MVA
c6	1	Invalidate Data or Unified cache line by MVA to PoU.	MVA

**Table 3-73 Register c7 cache and prefetch buffer maintenance operations (continued)**

CRm	Opcode_2	Function	Data
c6	2	Invalidate Data or Unified cache line by Set/Way.	Set/Way
c10	1	Clean Data or Unified cache line by MVA to PoC.	MVA
c10	2	Clean Data or Unified cache line by Set/Way.	Set/Way
c11	1	Clean Data or Unified cache line by MVA to PoU.	MVA
c14	1	Clean and Invalidate Data or Unified cache line by MVA to PoC.	MVA
c14	2	Clean and Invalidate Data or Unified cache line by Set/Way.	Set/Way

- a. Only applies to separate instruction caches, does not apply to unified caches.  
b. Available in User mode.

### Set and way

Figure 3-32 shows the set and way format for invalidate and clean operations.

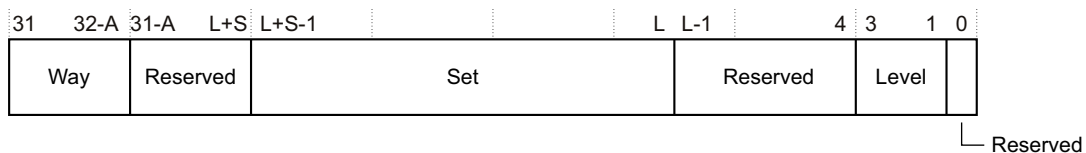
**Figure 3-32 c7 format for set and way**

Table 3-74 shows how the bit values correspond with the Cache Operation functions for set and way format operations.

**Table 3-74 Functional bits of c7 for set and way**

Bits	Field	Function
[31:32-A]	Way	Selects the way for the c7 set and way cache operation.
[31-A:L+S]	-	Reserved, SBZ.
[L+S-1:L]	Set	Selects the set for the c7 set and way cache operation.
[L-1:4]	-	Reserved, SBZ.
[3:1]	Level	Selects the cache level for the c7 set and way operation. 0 indicates cache level 1 is selected.
[0]	-	Reserved, SBZ.

For the processor, the L1 and L2 cache are configurable at implementation time. Therefore, the set and way fields are unique to the configured cache sizes. Table 3-75 shows the values of A, L, and S for L1 cache sizes, and Table 3-76 shows the values of A, L, and S for L2 cache sizes.

**Table 3-75 Values of A, L, and S for L1 cache sizes**

L1	A L S	Way	Set	Level
16KB	2 6 6	[31:30]	[11:6]	[3:1]
32KB	2 6 7	[31:30]	[12:6]	[3:1]

Table 3-76 shows the values of A, L, and S for L2 cache sizes and the resultant bit range for Way, Set, and Level. See Table 3-74 on page 3-91 and Figure 3-32 on page 3-91.

**Table 3-76 Values of A, L, and S for L2 cache sizes**

L2	A L S	Way	Set	Level
0KB	3 6 0	[31:29]	-	[3:1]
128KB	3 6 8	[31:29]	[13:6]	[3:1]
256KB	3 6 9	[31:29]	[14:6]	[3:1]
512KB	3 6 10	[31:29]	[15:6]	[3:1]
1024KB	3 6 11	[31:29]	[16:6]	[3:1]

See *c0*, *Cache Type Register* on page 3-26 for more information on cache sizes.

### **MVA**

Figure 3-33 shows the MVA format for invalidate, clean, and prefetch operations.

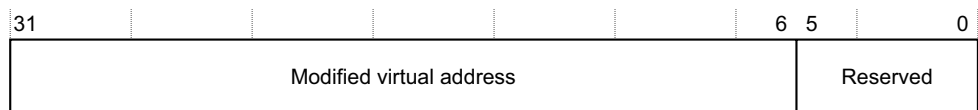
**Figure 3-33 c7 format for MVA**



Table 3-77 shows how the bit values correspond with the Cache Operation functions for MVA format operations.

**Table 3-77 Functional bits of c7 for MVA**

Bits	Field	Function
[31:6]	Modified virtual address	Specifies address to invalidate, clean, or prefetch
[5:0]	-	Reserved, SBZ

### **SBZ**

The value supplied Should-Be-Zero. The value `0x00000000` must be written to the register.

### **VA to PA translation operations**

The purpose of the VA to PA translation operations, nonsecure operations, is to provide a secure means to determine address translation between the Secure and Nonsecure states. VA to PA translations operate through:

- *PA Register*
- *VA to PA translation in the current Secure or Nonsecure state* on page 3-97
- *VA to PA translation in the other Secure or Nonsecure state* on page 3-97.

### **PA Register**

The purpose of the *Physical Address Register* (PAR) is to hold:

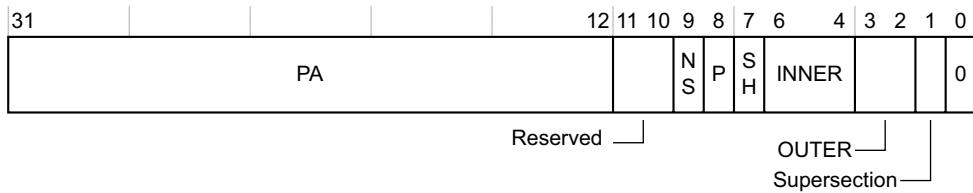
- the *Physical Address* (PA) after a successful translation
- the source of the abort for an unsuccessful translation.

Table 3-78 on page 3-94 shows the purpose of the bits of the PAR for successful translations and Table 3-79 on page 3-96 shows the purpose of the bits of the PAR for unsuccessful translations.

The PAR is:

- a read/write register banked in Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-34 on page 3-94 shows the bit arrangement of the PAR for successful translations.



**Figure 3-34 PA Register format for successful translation**

Figure 3-35 shows the bit arrangement of the PAR for unsuccessful translations.



**Figure 3-35 PA Register format for unsuccessful translation**

Table 3-78 shows how the bit values correspond with the PAR for a successful translation.

**Table 3-78 PA Register for successful translation bit functions**

Bits	Field	Function
[31:12]	PA	Contains the physical address after a successful translation.
[11:10]	-	Reserved. UNP, SBZ.
[9]	NS	Indicates the state of the NS attribute bit in the translation table: 0 = secure memory 1 = nonsecure memory.
[8]	P	Not used in the processor.
[7]	SH	Indicates shareable memory: 0 = nonshared 1 = shared.

**Table 3-78 PA Register for successful translation bit functions (continued)**

Bits	Field	Function
[6:4]	INNER	Indicates the inner attributes from the translation table: b000 = noncacheable b001 = strongly ordered b010 = reserved b011 = device b100 = reserved b101 = inner write-back, allocate on write b110 = inner write-through, no allocate on write b111 = inner write-back, no allocate on write.
[3:2]	OUTER	Indicates the outer attributes from the translation table: b00 = noncacheable b01 = write-back, allocate on write b10 = write-through, no allocate on write b11 = write-back, no allocate on write.
[1]	Supersection	Indicates if the result is a supersection: 0 = page is not a supersection, that is, PAR [31:12] contains PA[31:12], regardless of the page size. 1 = page is part of a supersection: PAR[31:24] contains PA[31:24] PAR[23:16] contains b00000000 PAR[15:12] contains b0000. <p style="text-align: center;">———— <b>Note</b> ————</p> PAR[23:12] is the same as VA[23:12] for supersections.
[0]	-	Indicates that the translation succeeded: 0 = translation successful.

Table 3-79 shows how the bit values correspond with the PAR for an unsuccessful translation.

**Table 3-79 PA Register for unsuccessful translation bit functions**

Bits	Field	Function
[31:7]	-	Reserved. UNP, SBZ.
[6:1]	FSR[12,10,3:0]	Holds the FSR bits for the aborted address. See <i>c5, Data Fault Status Register</i> on page 3-83 and <i>c5, Auxiliary Fault Status Registers</i> on page 3-87.
[0]	-	Indicates that the translation aborted: 1 = translation aborted.

Attempts to access the PAR in User mode results in an Undefined Instruction exception.

———— **Note** —————

The VA to PA translation can only generate an abort to the core if the operation failed because an external abort occurred on the possible translation table request. In this case, the processor does not update the PA Register. The processor updates the Data Fault Status Register and the Fault Address Register:

- if the EA bit in the Secure Configuration Register is set to 1, the secure versions of the two registers are updated and the processor traps the abort into Monitor mode
- if the EA bit in the Secure Configuration Register is not set to 1, the processor updates the secure or nonsecure versions of the two registers, depends whether the core is in Secure or Nonsecure state when the operation was issued.

For all other cases when the VA to PA operation fails, the processor only updates the PA Register, secure or nonsecure version, depends whether the core is in Secure or Nonsecure state when the operation was issued, with the Fault Status Register encoding and bit [0] set to 1. The Data Fault Status Register and Fault Address Register remain unchanged and the processor does not send an abort to the core.

To access the PA Register, read or write CP15 *c7* with:

MRC p15, 0, <Rd>, c7, c4, 0 ; Read PA Register

MCR p15, 0, <Rd>, c7, c4, 0 ; Write PA Register

**VA to PA translation in the current Secure or Nonsecure state**

The purpose of the VA to PA translation in the current Secure or Nonsecure state is to translate the address with the current virtual mapping for either Secure or Nonsecure state.

The VA to PA translation in the current Secure or Nonsecure state use:

- CP15 c7
- four, write-only operations common to the Secure and Nonsecure states
- operations accessible in privileged modes only.

The operations work for privileged or User access permissions and returns information in the PA Register for aborts, when the translation is unsuccessful, or translation table information, when the translation succeeds.

Attempts to access the VA to PA translation operations in the current Secure or Nonsecure state in User mode result in an Undefined Instruction exception.

To access the VA to PA translation in the current Secure or Nonsecure state, write CP15 c7 with:

```
MCR p15, 0, <Rn>, c7, c8, 3 ; get VA = <Rn> and run VA-to-PA translation
                               ; with User write permission.
                               ; if the selected translation table has the
                               ; User write permission, the PA is loaded in the PA
                               ; Register, otherwise abort information is loaded in
                               ; the PA Register.
MRC p15, 0, <Rd>, c7, c4, 0 ; read in <Rd> the PA value
```

**Note**

- The VA that this operation uses is the true VA not the MVA.
- General register <Rn> contains the VA for translation. The result returns in the PA Register.

**VA to PA translation in the other Secure or Nonsecure state**

The purpose of the VA to PA translation in the other Secure or Nonsecure state is to translate the address with the current virtual mapping in the Nonsecure state while the core is in the Secure state.

The VA to PA translation in the other Secure or Nonsecure state use:

- CP15 c7
- four, write-only operations in the Secure state only
- operations accessible in privileged modes only.

The operations work in the Secure state for nonsecure privileged or nonsecure User access permissions and returns information in the PA Register for aborts, when the translation is unsuccessful, or translation table information, when the translation succeeds.

When a VA to PA translation occurs in the other state from the Secure state, the value of the NS bit for a successful translation is Unpredictable.

Attempts to access the VA to PA translation operations in the other Secure or Nonsecure state in any nonsecure or User mode result in an Undefined Instruction exception.

To access the VA to PA translation in the other Secure or Nonsecure state, write CP15 c7 with Opcode\_2 set to:

- 4 for privileged read permission
- 5 for privileged write permission
- 6 for User read permission
- 7 for User write permission.

General register <Rn> contains the VA for translation. The result returns in the PA Register, for example:

```
MCR p15, 0, <Rn>, c7, c8, 4 ; get VA = <Rn> and run nonsecure translation
                               ; with nonsecure privileged read permission.
                               ; if the selected translation table has privileged
                               ; read permission, the PA is loaded in the PA
                               ; Register, otherwise abort information is loaded
                               ; in the PA Register.
MRC p15, 0, <Rd>, c7, c4, 0 ; read in <Rd> the PA value
```

### Data synchronization barrier operation

The purpose of the data synchronization barrier operation is to ensure that all outstanding explicit memory transactions complete before any following instructions begin. This ensures that data in memory is up to date before the processor executes any more instructions.

The data synchronization barrier operation is:

- a write-only operation, common to both Secure and Nonsecure states
- accessible in both User and privileged modes.

Table 3-80 shows the results of attempted access for each mode.

**Table 3-80 Results of access to the data synchronization barrier operation**

Read	Write
Undefined Instruction exception	Data

To perform a data synchronization barrier operation, write CP15 with:

```
MCR p15, 0, <Rd>, c7, c10, 4 ; Data synchronization barrier operation
```

See the *ARM Architecture Reference Manual* for more information on memory barriers.

### Data memory barrier operation

The purpose of the data memory barrier operation is to ensure that all outstanding explicit memory transactions complete before any following explicit memory transactions begin. This ensures that data in memory is up to date for any memory transaction that depends on it.

The data memory barrier operation is:

- a write-only operation, common to the Secure and Nonsecure states
- accessible in User and privileged modes.

Table 3-81 shows the results of attempted access for each mode.

**Table 3-81 Results of access to the data memory barrier operation**

Read	Write
Undefined Instruction exception	Data

To perform a data memory barrier operation, write CP15 with:

```
MCR p15, 0, <Rd>; c7, c10, 5 ; Data memory barrier operation
```

See the *ARM Architecture Reference Manual* for more information on memory barriers.

### 3.2.41 c8, TLB operations

The purpose of the TLB operations is to either:

- invalidate all the unlocked entries in the TLB
- invalidate all TLB entries for an area of memory before the MMU remaps it
- invalidate all TLB entries that match an ASID value.

You can perform these operations on either:

- instruction TLB
- data TLB.

To perform TLB operations, write CP15 with:

```
MCR p15, 0, <Rd>, c8, c5, 0 ; Invalidate Inst-TLB
MCR p15, 0, <Rd>, c8, c5, 1 ; Invalidate Inst-TLB entry (MVA)
MCR p15, 0, <Rd>, c8, c5, 2 ; Invalidate Inst-TLB (ASID)
MCR p15, 0, <Rd>, c8, c6, 0 ; Invalidate Data-TLB
MCR p15, 0, <Rd>, c8, c6, 1 ; Invalidate Data-TLB entry (MVA)
MCR p15, 0, <Rd>, c8, c6, 2 ; Invalidate Data-TLB (ASID)
MCR p15, 0, <Rd>, c8, c7, 0 ; Invalidate Inst-TLB and Data-TLB
MCR p15, 0, <Rd>, c8, c7, 1 ; Invalidate Inst-TLB and Data-TLB entry (MVA)
MCR p15, 0, <Rd>, c8, c7, 2 ; Invalidate Inst-TLB and Data-TLB (ASID)
```

All other ARMv7-A TLB maintenance encodings are Unpredictable.

Functions that update the contents of the TLB occur in program order. Therefore, an explicit data access before 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 unlocked entries

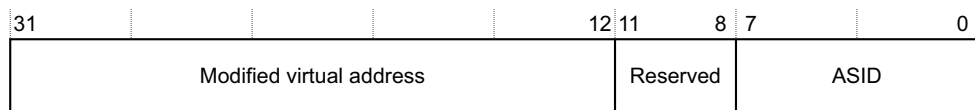
Invalidate TLB unlocked entries invalidates all the unlocked entries in the TLB.

### Invalidate TLB Entry by MVA

For an area of memory to be remapped, you can use the Invalidate TLB Entry by MVA to invalidate any TLB entry, locked or unlocked, by either:

- matching the MVA and ASID
- matching the MVA for a globally marked TLB entry.

The operation uses both the MVA and ASID as arguments. Figure 3-36 shows the format.



**Figure 3-36 TLB Operations MVA and ASID format**



### Invalidate TLB Entry on ASID Match

This operation invalidates all TLB entries that match the provided ASID value. This function invalidates locked entries but does not invalidate entries marked as global.

The Invalidate TLB Entry on ASID Match function requires an ASID as an argument. Figure 3-37 shows the format.



Figure 3-37 TLB Operations ASID format

### 3.2.42 c9, Performance Monitor Control Register

The purpose of the *Performance MoNitor Control* (PMNC) Register is to control the operation of the four Performance Monitor Count Registers, and the Cycle Counter Register:

The PMNC Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Figure 3-38 shows the bit arrangement of the PMNC Register.

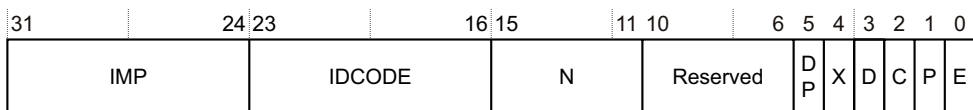


Figure 3-38 Performance Monitor Control Register format

Table 3-82 shows how the bit values correspond with the PMNC Register functions.

**Table 3-82 Performance Monitor Control Register bit functions**

Bits	Field	Function
[31:24]	IMP	Specifies the implementor code: 0x41 = ARM.
[23:16]	IDCODE	Specifies the identification code: 0x0.
[15:11]	N	Specifies the number of counters implemented: 0x4 = 4 counters implemented.
[10:6]	-	Reserved. RAZ, SBZP.
[5]	DP	Disables cycle counter, CCNT, when non-invasive debug is prohibited: 0 = count is enabled in regions where non-invasive debug is prohibited 1 = count is disabled in regions where non-invasive debug is prohibited.
[4]	X	Enables export of the events from the event bus to an external monitoring block, such as the ETM to trace events: 0 = export disabled, reset value 1 = export enabled.
[3]	D	Cycle count divider: 0 = counts every processor clock cycle, reset value 1 = counts every 64th processor clock cycle.
[2]	C	Cycle counter reset: 0 = no action 1 = resets cycle counter, CCNT, to zero. This bit Read-As-Zero.
[1]	P	Performance counter reset: 0 = no action 1 = resets all performance counters to zero. This bit Read-As-Zero.
[0]	E	Enable bit: 0 = disables all counters, including CCNT 1 = enables all counters including CCNT.

The PMNC Register is always accessible in privileged modes. Table 3-83 shows the results of attempted access for each mode.

**Table 3-83 Results of access to the Performance Monitor Control Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the PMNC Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c9, c12, 0 ; Read PMNC Register
MCR p15, 0, <Rd>, c9, c12, 0 ; Write PMNC Register
```

### 3.2.43 c9, Count Enable Set Register

The purpose of the *Count Enable Set* (CNTENS) Register is to enable or disable any of the Performance Monitor Count Registers.

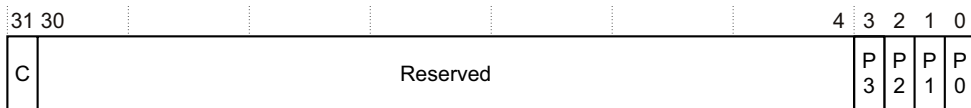
When reading this register, any enable that reads as 0 indicates the counter is disabled. Any enable that reads as 1 indicates the counter is enabled.

When writing this register, any enable written with a value of 0 is ignored, that is, not updated. Any enable written with a value of 1 indicates the counter is enabled.

The CNTENS Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Figure 3-39 shows the bit arrangement of the CNTENS Register.



**Figure 3-39 Count Enable Set Register format**

Table 3-84 shows how the bit values correspond with the CNTENS Register functions.

**Table 3-84 Count Enable Set Register bit functions**

Bits	Field	Function
[31]	C	Enable cycle counter.
[30:4]	-	Reserved. UNP, SBZ.
[3]	P3	Enable Counter 3.
[2]	P2	Enable Counter 2.
[1]	P1	Enable Counter 1.
[0]	P0	Enable Counter 0.

Table 3-85 shows the results of attempted access for each mode.

**Table 3-85 Results of access to the Count Enable Set Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the CNTENS Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c12, 1 ; Read CNTENS Register  
MCR p15, 0, <Rd>, c9, c12, 1 ; Write CNTENS Register

### 3.2.44 c9, Count Enable Clear Register

The purpose of the *Count Enable Clear* (CNTENC) Register is to enable or disable any of the Performance Monitor Count Registers.

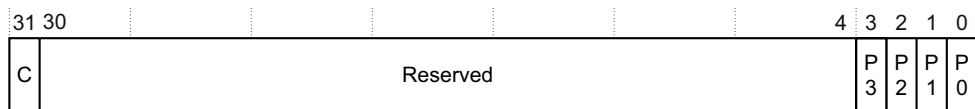
When reading this register, any enable that reads as 0 indicates the counter is disabled. Any enable that reads as 1 indicates the counter is enabled.

When writing this register, any enable written with a value of 0 is ignored, that is, not updated. Any enable written with a value of 1 clears the counter enable to 0.

The CNTENC Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9*, *User Enable Register* on page 3-117.

Figure 3-40 shows the bit arrangement of the CNTENC Register.



**Figure 3-40 Count Enable Clear Register format**

Table 3-86 shows how the bit values correspond with the CNTENC Register functions.

**Table 3-86 Count Enable Clear Register bit functions**

Bits	Field	Function
[31]	C	Disable cycle counter.
[30:4]	-	Reserved. UNP, SBZP.
[3]	P3	Disable Counter 3.
[2]	P2	Disable Counter 2.
[1]	P1	Disable Counter 1.
[0]	P0	Disable Counter 0.

Table 3-87 shows the results of attempted access for each mode.

**Table 3-87 Results of access to the Count Enable Clear Register**

	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
EN = 0 <sup>a</sup>	Data	Data	Data	Data	Undefined <sup>b</sup>	Undefined	Undefined	Undefined
EN = 1	Data	Data	Data	Data	Data	Data	Data	Data

a. The EN bit in *c9*, *User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

b. The access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the CNTENC Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c12, 2 ; Read CNTENC Register

MCR p15, 0, <Rd>, c9, c12, 2 ; Write CNTENC Register

You can use the enable, EN, bit [0] of the PMNC Register to disable all performance counters including CCNT. The CNTENC Register retains its value when the enable bit of the PMNC is set to 0, even though its settings are ignored.

### 3.2.45 c9, Overflow Flag Status Register

The purpose of the *Overflow Flag Status* (FLAG) Register is to enable or disable any of the performance monitor counters producing an overflow flag.

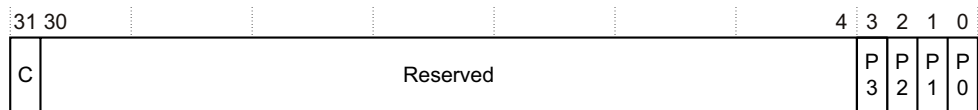
When reading this register, any overflow flag that reads as 0 indicates the counter has not overflowed. Any overflow flag that reads as 1 indicates the counter has overflowed.

When writing this register, any overflow flag written with a value of 0 is ignored, that is, not updated. Any overflow flag written with a value of 1 clears the counter overflow flag to 0.

The FLAG Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Figure 3-41 shows the bit arrangement of the FLAG Register.



**Figure 3-41 Overflow Flag Status Register format**

Table 3-88 shows how the bit values correspond with the FLAG Register functions.

**Table 3-88 Overflow Flag Status Register bit functions**

Bits	Field	Function
[31]	C	Cycle counter overflow flag.
[30:4]	-	Reserved. UNP, SBZP.
[3]	P3	Counter 3 overflow flag.
[2]	P2	Counter 2 overflow flag.
[1]	P1	Counter 1 overflow flag.
[0]	P0	Counter 0 overflow flag.

Table 3-89 shows the results of attempted access for each mode.

**Table 3-89 Results of access to the Overflow Flag Status Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the FLAG Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c9, c12, 3 ; Read FLAG Register
MCR p15, 0, <Rd>, c9, c12, 3 ; Write FLAG Register
```

### 3.2.46 c9, Software Increment Register

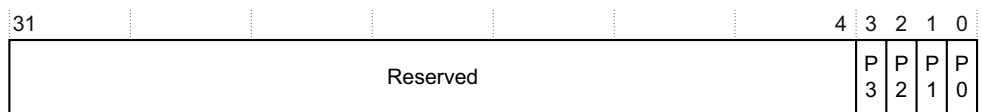
The purpose of the *Software INCRement* (SWINCR) Register is to increment the count of a performance monitor count register.

When writing this register, a value of 1 increments the counter, and a value of 0 does nothing.

The SWINCR Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Figure 3-42 shows the bit arrangement of the SWINCR Register.



**Figure 3-42 Software Increment Register format**

Table 3-90 shows how the bit values correspond with the SWINCR Register functions.

**Table 3-90 Software Increment Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZP
[3]	P3	Increment Counter 3
[2]	P2	Increment Counter 2
[1]	P1	Increment Counter 1
[0]	P0	Increment Counter 0

The SWINCR Register only has effect when counter event is set to  $0x00$ .

Table 3-91 shows the results of attempted access for each mode.

**Table 3-91 Results of access to the Software Increment Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	0	Data	Undefined	Undefined	Undefined	Undefined
1	0	Data	0	Data	0	Data	0	Data

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the SWINCR Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c12, 4 ; Read SWINCR Register  
MCR p15, 0, <Rd>, c9, c12, 4 ; Write SWINCR Register

### 3.2.47 c9, Performance Counter Selection Register

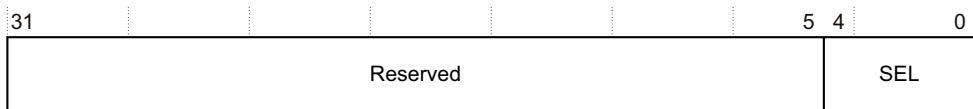
The purpose of the *Performance Counter SElection* (PMNXSEL) Register is to select a Performance Monitor Count Register.

The PMNXSEL Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.



Figure 3-43 shows the bit arrangement of the PMNXSEL Register.



**Figure 3-43 Performance Counter Selection Register format**

Table 3-92 shows how the bit values correspond with the PMNXSEL Register functions.

**Table 3-92 Performance Counter Selection Register bit functions**

Bits	Field	Function
[31:5]	-	RAZ, SBZP.
[4:0]	SEL	Counter select: 5'b00000 = selects counter 0 5'b00001 = selects counter 1 5'b00010 = selects counter 2 5'b00011 = selects counter 3.

Any values programmed in the PMNXSEL Register other than those specified in Table 3-92 are Unpredictable.

Table 3-93 shows the results of attempted access for each mode.

**Table 3-93 Results of access to the Performance Counter Selection Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

b. The EN bit in *c9*, *User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the PMNXSEL Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c12, 5; Read PMNXSEL Register  
MCR p15, 0, <Rd>, c9, c12, 5; Write PMNXSEL Register

### 3.2.48 c9, Cycle Count Register

The purpose of the *Cycle CouNT* (CCNT) Register is to count the number of clock cycles since the register was reset. See bit [3] of the *c9, Performance Monitor Control Register* on page 3-101.

The CCNT Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Table 3-94 shows the results of attempted access for each mode.

**Table 3-94 Results of access to the Cycle Count Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the CCNT Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c13, 0 ; Read CCNT Register  
MCR p15, 0, <Rd>, c9, c13, 0 ; Write CCNT Register

The CCNT Register must be disabled before software can write to it. Any attempt by software to write to this register when enabled is Unpredictable.

### 3.2.49 c9, Event Selection Register

The purpose of the *Event SElection* (EVTSEL) Register is to select the events that you want a Performance Monitor Count Register to count.

The EVTSEL Register is:

- a read/write register common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Figure 3-44 on page 3-111 shows the bit arrangement of the EVTSEL Register.

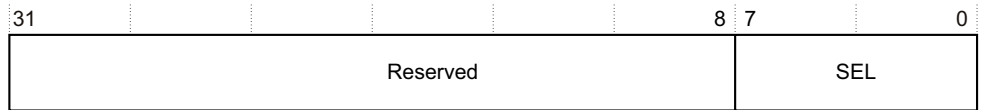
**Figure 3-44 Event Selection Register format**

Table 3-95 shows how the bit values correspond with the EVTSEL Register functions.

**Table 3-95 Event Selection Register bit functions**

Bits	Field	Function
[31:8]	-	Reserved. RAZ, SBZP
[7:0]	SEL	Specifies the event selected as shown in Table 3-97 on page 3-112

Table 3-96 shows the results of attempted access for each mode.

**Table 3-96 Results of access to the Event Selection Register<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- b. The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the EVTSEL Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c13, 1 ; Read EVTSEL Register  
MCR p15, 0, <Rd>, c9, c13, 1 ; Write EVTSEL Register

Table 3-97 shows the range values for predefined events that you can monitor using the EVTSEL Register.

**Table 3-97 Values for predefined events**

<b>Value</b>	<b>Description</b>
0x00	Software increment. The register is incremented only on writes to the Software Increment Register. See <i>c9, Software Increment Register</i> on page 3-107.
0x01	Instruction fetch that causes a refill at the lowest level of instruction or unified cache. Each instruction fetch from normal cacheable memory that causes a refill from outside of the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss are not counted. Where instruction fetches consist of multiple instructions, these accesses count as single events. CP15 cache maintenance operations do not count as events. This counter increments for speculative instruction fetches and for fetches of instructions that reach execution.
0x02	Instruction fetch that causes a TLB refill at the lowest level of TLB. Each instruction fetch that causes a translation table walk or an access to another level of TLB caching is counted. CP15 TLB maintenance operations do not count as events. This counter increments for speculative instruction fetches and for fetches of instructions that reach execution.
0x03	Data read or write operation that causes a refill at the lowest level of data or unified cache. Each data read from or write to normal cacheable memory that causes a refill from outside of the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss are not counted. Each access to a cache line to normal cacheable memory that causes a new linefill is counted, including the multiple transaction of instructions such as LDM or STM, PUSH and POP. Write-through writes that hit in the cache do not cause a linefill and so are not counted. CP15 cache maintenance operations do not count as events. This counter increments for speculative data accesses and for data accesses that are explicitly made by instructions.
0x04	Data read or write operation that causes a cache access at the lowest level of data or unified cache. Each access to a cache line to normal cacheable memory is counted including the multiple transaction of instructions such as LDM or STM. CP15 cache maintenance operations do not count as events. This counter increments for speculative data accesses and for data accesses that are explicitly made by instructions.
0x05	Data read or write operation that causes a TLB refill at the lowest level of TLB. Each data read or write operation that causes a translation table walk or an access to another level of TLB caching is counted. CP15 TLB maintenance operations do not count as events. This counter increments for speculative data accesses and for data accesses that are explicitly made by instructions.
0x06	Data read architecturally executed. This counter increments for every instruction that explicitly read data, including SWP. This counter only increments for instructions that are unconditional or that pass their condition codes.
0x07	Data write architecturally executed. The counter increments for every instruction that explicitly wrote data, including SWP. This counter only increments for instructions that are unconditional or that pass their condition codes.

Table 3-97 Values for predefined events (continued)

Value	Description
0x08	Instruction architecturally executed. This counter counts for all instructions, including conditional instructions that fail their condition codes.
0x09	Exception taken. This counts for each exception taken.
0x0A	Exception return architecturally executed. This includes: <ul style="list-style-type: none"> <li>• RFE &lt;addressing_mode&gt; &lt;Rn&gt;{!}</li> <li>• MOVS PC (and other similar data processing instructions)</li> <li>• LDM &lt;addressing_mode&gt; Rn{!}, &lt;registers_and_pc&gt;</li> </ul> This counter only increments for instructions that are unconditional or that pass their condition codes.
0x0B	Instruction that writes to the Context ID Register architecturally executed. This counter only increments for instructions that are unconditional or that pass their condition codes.
0x0C	Software change of PC, except by an exception, architecturally executed. This counter only increments for instructions that are unconditional or that pass their condition codes.
0x0D	Immediate branch architecturally executed, taken or not taken. This includes B{L}, BLX, CB{N}Z, HB{L}, and HBLP. This counter counts for all immediate branch instructions that are architecturally executed, including conditional instructions that fail their condition codes.
0x0E	Procedure return, other than exception returns, architecturally executed. This includes: <ul style="list-style-type: none"> <li>• BX R14</li> <li>• MOV PC, LR</li> <li>• POP {..., PC}</li> <li>• LDR PC, [R13], #offset</li> <li>• LDMIA R9!, {...,PC}</li> <li>• LDR PC, [R9], #offset</li> </ul> This counter only increments for instructions that are unconditional or that pass their condition codes.
0x0F	Unaligned access architecturally executed. This counts each instruction that is an access to an unaligned address. This counter only increments for instructions that are unconditional or that pass their condition codes.
0x10	Branch mispredicted or not predicted. This counts for every pipeline flush because of a misprediction from the program flow prediction resources.
0x11	Cycle count. This counts for every clock cycle.
0x12	Branches or other change in the program flow that could have been predicted by the branch prediction resources of the processor.
0x13-0x3F	Reserved.

Table 3-97 Values for predefined events (continued)

Value	Description
0x40	Any write buffer full cycle.
0x41	Any store that is merged in the L2 memory system.
0x42	Any bufferable store transaction from load/store to L2 cache, excluding eviction or cast out data.
0x43	Any accesses to the L2 cache.
0x44	Any cacheable miss in the L2 cache.
0x45	The number of AXI read data packets.
0x46	The number of AXI write data packets.
0x47	Any replay event in the memory system.
0x48	Any unaligned memory access that results in a replay.
0x49	Any L1 data memory access that misses in the cache as a result of the hashing algorithm. The cases covered are: <ul style="list-style-type: none"> <li>• hash hit and physical address miss</li> <li>• hash hit and physical address hit in another way</li> <li>• hash miss and physical address hit.</li> </ul>
0x4a	Any L1 instruction memory access that misses in the cache as a result of the hashing algorithm. The cases covered are: <ul style="list-style-type: none"> <li>• hash hit and physical address miss</li> <li>• hash hit and physical address hit in another way</li> <li>• hash miss and physical address hit.</li> </ul>
0x4b	Any L1 data memory access in which a page coloring alias occurs. alias = virtual address [12] != physical address [12] This behavior results in a data memory eviction or cast out.
0x4c	Any NEON access that hits in the L1 data cache.
0x4d	Any NEON cacheable data accesses for L1 data cache.
0x4e	Any L2 cache accesses as a result of a NEON memory access.
0x4f	Any NEON hit in the L2 cache.
0x50	Any L1 instruction cache access, excluding CP15 cache accesses.
0x51	Any return stack misprediction because of incorrect target address for a taken return stack pop.

Table 3-97 Values for predefined events (continued)

Value	Description
0x52	Two forms of branch direction misprediction: <ul style="list-style-type: none"> <li>• branch predicted taken, but was not taken</li> <li>• branch predicted not taken, but was taken.</li> </ul>
0x53	Any predictable branch that is predicted to be taken.
0x54	Any predictable branch that is executed and taken.
0x55	Number of operations issued, where an operation is either: <ul style="list-style-type: none"> <li>• an instruction</li> <li>• one operation in a sequence of operations that make up a multi-cycle instruction.</li> </ul>
0x56	Increment for every cycle that no instructions are available for issue.
0x57	For every cycle, this event counts the number of instructions issued in that cycle. Multi-cycle instructions are only counted once.
0x58	Number of cycles the processor stalls waiting on MRC data from NEON.
0x59	Number of cycles that the processor stalls as a result of a full NEON instruction queue or NEON load queue.
0x5a	Number of cycles that NEON and integer processors are both not idle.
0x60-0x6F	Reserved.
0x70	Counts any event from external input source <b>PMUEXTIN[0]</b> .
0x71	Counts any event from external input source <b>PMUEXTIN[1]</b> .
0x72	Counts any event from both external input sources <b>PMUEXTIN[0]</b> and <b>PMUEXTIN[1]</b> .
0x73-0xFF	Reserved.

If this unit generates an interrupt, the processor asserts the pin **nPMUIRQ**. You can route this pin to an external interrupt controller for prioritization and masking. This is the only mechanism that signals this interrupt to the core.

The absolute counts recorded might vary because of pipeline effects. This has negligible effect except in cases where the counters are enabled for a very short time.

In addition to the counters within the processor, most of the events that Table 3-97 on page 3-112 shows are available to the ETM unit or other external trace hardware to enable the events to be monitored. See Chapter 14 *Embedded Trace Macrocell* and Chapter 15 *Cross Trigger Interface* for more information.

### 3.2.50 c9, Performance Monitor Count Registers

There are four *Performance Monitor CouNT* (PMCNT0-PMCNT3) Registers in the processor. The purpose of each PMCNT Register, as selected by the PMNXSEL Register, is to count instances of an event selected by the EVTSEL Register. Bits [31:0] of each PMCNT Register contain an event count.

The PMCNT0-PMCNT3 Registers are:

- read/write registers common to Secure and Nonsecure states
- accessible as determined by *c9, User Enable Register* on page 3-117.

Table 3-98 shows the results of attempted access for each mode.

**Table 3-98 Results of access to the Performance Monitor Count Registers<sup>a</sup>**

EN <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.  
 b. The EN bit in *c9, User Enable Register* on page 3-117 enables User mode access of the Performance Monitor Registers.

To access the PMCNT Registers, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c13, 2; Read PMCNT0-PMCNT3 Registers  
 MCR p15, 0, <Rd>, c9, c13, 2; Write PMCNT0-PMCNT3 Registers

Table 3-99 shows what signal settings are required and the Secure or Nonsecure state and mode that you can enable the counters.

**Table 3-99 Signal settings for the Performance Monitor Count Registers**

DBGEN    NIDEN	SPIDEN    SPNIDEN	SUNIDEN	Secure state	User mode	PMNC[5]	Performance counters enabled	CCNT enabled
0	-	-	-	-	b0	No	Yes
0	-	-	-	-	b1	No	No
1	-	-	No	-	-	Yes	Yes
1	-	-	Yes	-	-	Yes	Yes
1	0	-	Yes	No	b0	No	Yes



Table 3-99 Signal settings for the Performance Monitor Count Registers (continued)

DBGEN    NIDEN	SPIDEN    SPNIDEN	SUNIDEN	Secure state	User mode	PMNC[5]	Performance counters enabled	CCNT enabled
1	0	-	Yes	No	b1	No	No
1	0	0	Yes	Yes	b0	No	Yes
1	0	0	Yes	Yes	b1	No	No
1	0	1	Yes	Yes	X	Yes	Yes

### 3.2.51 c9, User Enable Register

The purpose of the *USER ENable* (USEREN) Register is to enable User mode to have access to the Performance Monitor Registers.

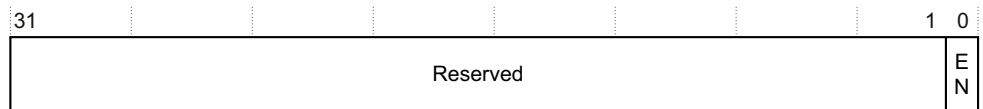
———— **Note** —————

USEREN Register does not provide access to the registers that control interrupt generation.

The USEREN Register is:

- a read/write register common to Secure and Nonsecure states
- writable only in privileged mode and readable in any processor mode.

Figure 3-45 shows the bit arrangement of the USEREN Register.



**Figure 3-45 User Enable Register format**

Table 3-100 shows how the bit values correspond with the USEREN Register functions.

**Table 3-100 User Enable Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved. RAZ, SBZP
[0]	EN	User mode enable

Table 3-101 shows the results of attempted access for each mode.

**Table 3-101 Results of access to the User Enable Register**

EN	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Data	Undefined exception	Data	Undefined exception
1	Data	Data	Data	Data	Data	Undefined exception	Data	Undefined exception

To access the USEREN Register, read or write CP15 with:

MRC p15, 0, <Rd>, c9, c14, 0 ; Read USEREN Register

MCR p15, 0, <Rd>, c9, c14, 0 ; Write USEREN Register

### 3.2.52 c9, Interrupt Enable Set Register

The purpose of the *INTerrupt ENable Set* (INTENS) Register is to determine if any of the Performance Monitor Count Registers, PMCNT0-PMCNT3 and CCNT, generate an interrupt on overflow.

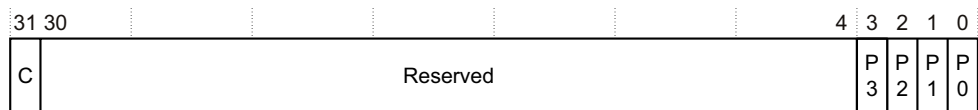
The INTENS Register is:

- a read/write register common to Secure and Nonsecure states
- accessible in privileged mode only.

When reading this register, any interrupt overflow enable bit that reads as 0 indicates the interrupt overflow flag is disabled. Any interrupt overflow enable bit that reads as 1 indicates the interrupt overflow flag is enabled.

When writing this register, any interrupt overflow enable bit written with a value of 0 is ignored, that is, not updated. Any interrupt overflow enable bit written with a value of 1 sets the interrupt overflow enable bit.

Figure 3-46 shows the bit arrangement of the INTENS Register.



**Figure 3-46 Interrupt Enable Set Register format**

Table 3-102 shows how the bit values correspond with the INTENS Register functions.

**Table 3-102 Interrupt Enable Set Register bit functions**

Bits	Field	Function
[31]	C	CCNT overflow interrupt enable.
[30:4]	-	Reserved. UNP, SBZP.
[3]	P3	PMCNT3 overflow interrupt enable.
[2]	P2	PMCNT2 overflow interrupt enable.
[1]	P1	PMCNT1 overflow interrupt enable.
[0]	P0	PMCNT0 overflow interrupt enable.

Table 3-103 shows the results of attempted access for each mode.

**Table 3-103 Results of access to the Interrupt Enable Set Register<sup>a</sup>**

EN	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the INTENS Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c9, c14, 1 ; Read INTENS Register
MCR p15, 0, <Rd>, c9, c14, 1 ; Write INTENS Register
```

### 3.2.53 c9, Interrupt Enable Clear Register

The purpose of the *INTerrupt ENable Clear* (INTENC) Register is to determine if any of the Performance Monitor Count Registers, PMCNT0-PMCNT3 and CCNT, generate an interrupt on overflow.

The INTENC Register is:

- a read/write register common to Secure and Nonsecure states
- accessible in privileged mode only.



To access the INTENC Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c9, c14, 2 ; Read INTENC Register  
MCR p15, 0, <Rd>, c9, c14, 2 ; Write INTENC Register
```

### 3.2.54 c9, L2 Cache Lockdown Register

The L2 Cache Lockdown Register controls the L2 cache lockdown. The Lockdown Format C provides a method to restrict the replacement algorithm on cache linefills to only use selected cache ways within a set. Using this method, you can fetch or load code into the L2 cache and protect data from being evicted, or you can use the method to reduce cache pollution.

The L2 Cache Lockdown Register is:

- a read-only or read/write register, depending on the security selected for the register access using CL bit [16] in the *c1, Nonsecure Access Control Register* on page 3-73.
- accessible in privileged modes only.

Figure 3-48 shows the bit arrangement of the L2 Cache Lockdown Register.

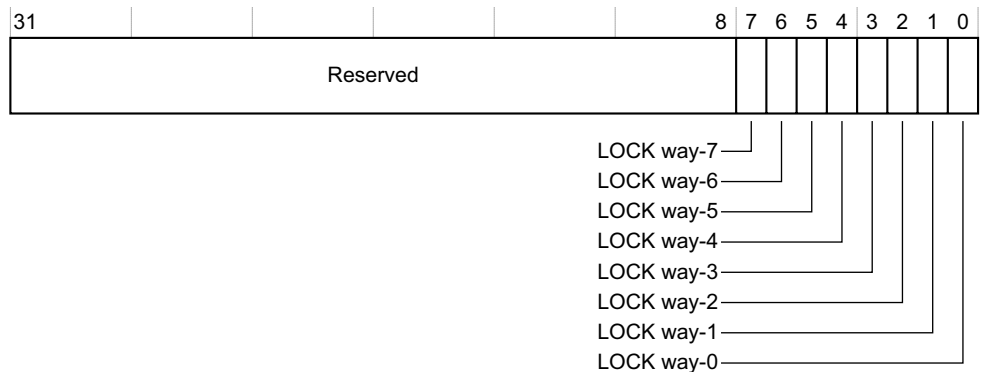


Figure 3-48 L2 Cache Lockdown Register format

Table 3-106 shows how the bit values correspond with the L2 Cache Lockdown Register functions.

**Table 3-106 L2 Cache Lockdown Register bit functions**

Bits	Field	Function
[31:8]	-	Reserved. UNP, SBZP.
[7]	LOCK way-7	Lockdown bit for way 7 of the L2 cache: 0 = way 7 is not locked and allocation is determined by standard replacement algorithm 1 = way 7 is locked and no allocation is performed to this cache way.
[6]	LOCK way-6	Lockdown bit for way 6 of the L2 cache: 0 = way 6 is not locked and allocation is determined by standard replacement algorithm 1 = way 6 is locked and no allocation is performed to this cache way.
[5]	LOCK way-5	Lockdown bit for way 5 of the L2 cache: 0 = way 5 is not locked and allocation is determined by standard replacement algorithm 1 = way 5 is locked and no allocation is performed to this cache way.
[4]	LOCK way-4	Lockdown bit for way 4 of the L2 cache: 0 = way 4 is not locked and allocation is determined by standard replacement algorithm 1 = way 4 is locked and no allocation is performed to this cache way.
[3]	LOCK way-3	Lockdown bit for way 3 of the L2 cache: 0 = way 3 is not locked and allocation is determined by standard replacement algorithm 1 = way 3 is locked and no allocation is performed to this cache way.
[2]	LOCK way-2	Lockdown bit for way 2 of the L2 cache: 0 = way 2 is not locked and allocation is determined by standard replacement algorithm 1 = way 2 is locked and no allocation is performed to this cache way.
[1]	LOCK way-1	Lockdown bit for way 1 of the L2 cache: 0 = way 1 is not locked and allocation is determined by standard replacement algorithm 1 = way 1 is locked and no allocation is performed to this cache way.
[0]	LOCK way-0	Lockdown bit for way 0 of the L2 cache: 0 = way 0 is not locked and allocation is determined by standard replacement algorithm 1 = way 0 is locked and no allocation is performed to this cache way.

Table 3-107 shows the results of attempted access for each mode.

**Table 3-107 Results of access to the L2 Cache Lockdown Register<sup>a</sup>**

CL bit value	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the L2 Cache Lockdown Register, read or write CP15 with:

```
MRC p15, 1, <Rd>, c9, c0, 0 ; Read L2 Cache Lockdown Register
MCR p15, 1, <Rd>, c9, c0, 0 ; Write L2 Cache Lockdown Register
```

### Specific loading of addresses into cache way

The following procedure for lock down into a data or an 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. Disable interrupts to ensure that no processor exceptions can occur during the execution of this procedure. If this is not possible, all code and data that any exception handlers can call must meet the conditions specified in step 2 and step 3.
2. Ensure that all data that the following code uses, apart from the data that is to be locked down, is either:
  - in a noncacheable area of memory
  - in an already locked cache way.
3. Ensure that the data to be locked down is in a cacheable area of memory.
4. Ensure that the data to be locked down is not already in the cache, using either:
  - cache clean
  - invalidate
  - cache clean and invalidate.

See *c7, Cache operations* on page 3-89.

5. Enable allocation to the target cache way by writing to the Instruction or Data Cache Lockdown Register, with the CRm field set to 0, setting L to 0 for bit *i*, and L 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 the Instruction or Data Cache Lockdown Register, setting L to 1 for bit *i* and restore all the other bits to the previous values before this routine was started.

### Cache unlock procedure

To unlock the lock down portion of the cache, write to register c9, setting L to 0 for each bit.

### 3.2.55 c9, L2 Cache Auxiliary Control Register

The purpose of the L2 Cache Auxiliary Control Register is to enable you to configure the L2 cache behavior.

The L2 Cache Auxiliary Control Register is:

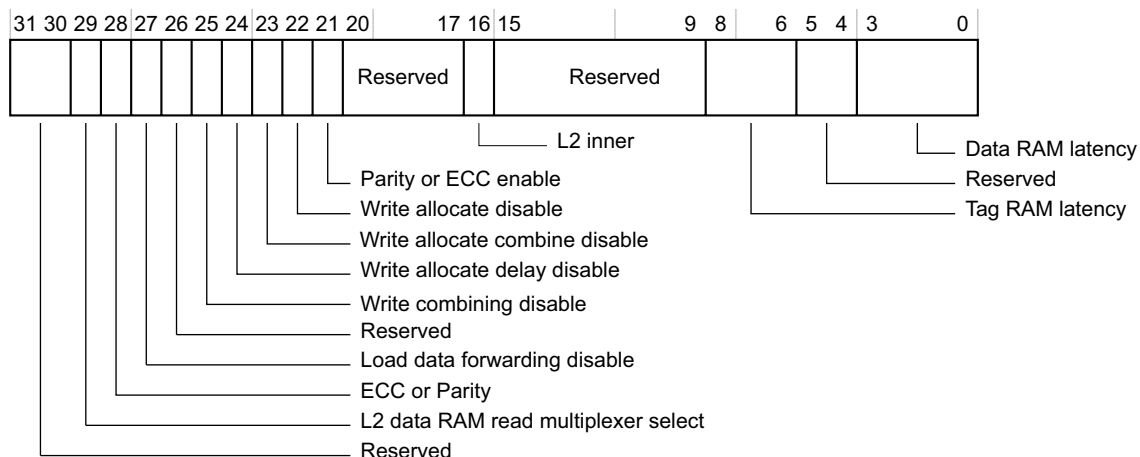
- a read register, common to Secure and Nonsecure states
- a write register in Secure state only
- accessible in privileged modes only.

#### ———— Note —————

If bit [24] of the L2 Cache Auxiliary Control Register is not set to 1, the L2 cache does not perform a security check for the data that is placed in the L2 cache. If software requires a higher level of security within the processor, then you must set bit [24] to 1 of this register. By setting bit [24] to 1, the L2 cache performs an external linefill, and the AXI slave performs the security check on that linefill.

Figure 3-49 on page 3-125 shows the bit arrangement of the L2 Cache Auxiliary Control Register.





**Figure 3-49 L2 Cache Auxiliary Control Register format**

Table 3-108 shows how the bit values correspond with the L2 Cache Auxiliary Control Register functions.

**Table 3-108 L2 Cache Auxiliary Control Register bit functions**

Bits	Field	Function
[31:30]	-	Reserved. UNP, SBZP.
[29]	L2 data RAM read multiplexer select	Configures the timing of the read data multiplexer select between one or two cycles for all L2 data RAM read operations: 0 = two cycles, default 1 = one cycle.
[28]	ECC or Parity	Selects ECC or parity: 0 = parity 1 = ECC.
[27]	Load data forwarding disable	Enables or disables load data forwarding to any LS or NEON request: 0 = enables load data forwarding, default 1 = disables load data forwarding.
[26]	-	Reserved. UNP, SBZP.
[25]	Write combining disable	Enables or disables write combining: 0 = enables write combine, default 1 = disables write combine.

**Table 3-108 L2 Cache Auxiliary Control Register bit functions (continued)**

<b>Bits</b>	<b>Field</b>	<b>Function</b>
[24]	Write allocate delay disable	Enables or disables external linefill when storing an entire line with write allocate permission: 0 = enables write allocate delay, default 1 = disables write allocate delay.
[23]	Write allocate combine disable	Enables or disables combining of data in the L2 write combining buffers: 0 = enables write allocate combine, default 1 = disables write allocate combine.
[22]	Write allocate disable	Enables or disables allocate on write miss in L2: 0 = enables write allocate, default 1 = disables write allocate.
[21]	Parity or ECC enable	Parity or ECC enable: 0 = disables parity or ECC, default 1 = enables parity or ECC.
[20:17]	-	Reserved. UNP, SBZP.
[16]	L2 inner	Defines whether the L2 observes the inner or outer cacheability attributes: 0 = L2 observes outer cacheability 1 = L2 observes inner cacheability.
[15:9]	-	Reserved. UNP, SBZP.

**Table 3-108 L2 Cache Auxiliary Control Register bit functions (continued)**

<b>Bits</b>	<b>Field</b>	<b>Function</b>
[8:6]	Tag RAM latency	Program tag RAM latency: b000 = 2 cycles b001 = 2 cycles b010 = 3 cycles b011 = 4 cycles b100 = 4 cycles b101 = 4 cycles b110 = 4 cycles b111 = 4 cycles.
[5:4]	-	Reserved. UNP, SBZP.
[3:0]	Data RAM latency	Program data RAM latency: b0000 = 2 cycles b0001 = 2 cycles b0010 = 3 cycles b0011 = 4 cycles b0100 = 5 cycles b0101 = 6 cycles b0110 = 7 cycles b0111 = 8 cycles b1000 = 9 cycles b1001 = 10 cycles b1010 = 11 cycles b1011 = 12 cycles b1100 = 13 cycles b1101 = 13 cycles b1110 = 13 cycles b1111 = 13 cycles.

Table 3-109 shows the results of attempted access for each mode.

**Table 3-109 Results of access to the L2 Cache Auxiliary Control Register<sup>a</sup>**

<b>Secure privileged</b>		<b>Nonsecure privileged</b>		<b>Secure User</b>		<b>Nonsecure User</b>	
<b>Read</b>	<b>Write</b>	<b>Read</b>	<b>Write</b>	<b>Read</b>	<b>Write</b>	<b>Read</b>	<b>Write</b>
Data	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the L2 Cache Auxiliary Control Register, read or write CP15 with:

```
MRC p15, 1, <Rd>, c9, c0, 2 ; Read L2 Cache Auxiliary Control Register
MCR p15, 1, <Rd>, c9, c0, 2 ; Write L2 Cache Auxiliary Control Register
```

If you have not configured the processor to include parity and ECC RAM, then software cannot set bit [21] to 1, parity or ECC enable bit. The following code sequence shows how to determine if the processor was configured to include parity and ECC RAM.

```
MRC p15, 1, <Rd>, c9, c0, 2 ; Read L2 Cache Auxiliary Control Register
ORR <Rd>, <Rd>, #0x0020_0000; Set parity/ECC enable
MCR p15, 1, <Rd>, c9, c0, 2 ; Write L2 Cache Auxiliary Control Register
MRC p15, 1, <Rd>, c9, c0, 2 ; Read L2 Cache Auxiliary Control Register
TST <Rd>, #0x0020_0000 ; Test for parity/ECC enable
BEQ no_parity_ram_setup
```

```
parity_ram_setup:
    ;do parity RAM setup
    B done_parity_RAM_setup

no_parity_ram_setup:
    ;do no parity/ECC RAM setup

done_parity_RAM_setup:
    ;<continue>
```

### 3.2.56 c10, TLB Lockdown Registers

The purpose of the TLB Lockdown Registers is to control which of the fully-associative TLB entries to allocate on the next table walk. The TLB is normally allocated on a rotating basis. The oldest entry is always the next allocated.

You can configure the TLB Lockdown Registers to exclude a range of entries from the round-robin allocation scheme. You must use the TLB Lockdown Registers with the TLB preload operation. See *c10, TLB preload operation* on page 3-130 for more information.

The TLB Lockdown Registers are:

- read/write registers common to Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-50 on page 3-129 shows the bit arrangement of the Data and Instruction TLB Lockdown Registers.

**Figure 3-50 TLB Lockdown Register format**

Table 3-110 shows how the bit values correspond with the Data and Instruction TLB Lockdown Register functions.

**Table 3-110 TLB Lockdown Register bit functions**

Bits	Field	Function
[31:27]	Base	Defines the offset from TLB entry 0 for which entries 0 to base - 1 are locked assuming P equals 1 during a hardware translation table walk.
[26:22]	Victim	Specifies the entry where the next hardware translation table walk can place a TLB entry. The reset value is 0. Each hardware translation table walk increments the value of the Victim field.
[21:1]	-	Reserved. UNP, SBZP.
[0]	P	Determines if TLB entries allocated by subsequent translation table walks are not invalidated by the Invalidate TLB unlocked entries operation: 0 = allocated TLB entries are invalidated, reset value 1 = allocated TLB entries are not invalidated.

For the Data or Instruction TLB Lockdown Register:

- The TLB lockdown behavior depends on the TL bit, see *c1, Nonsecure Access Control Register* on page 3-73. If the TL bit is not set to 1, the lockdown entries are reserved for the Secure state.
- The TLB Lockdown Register must be used with the TLB preload operation. See *c10, TLB preload operation* on page 3-130.

———— **Note** ————

Setting the P bit before a hardware translation table walk does not guarantee the locking down of an entry. The Base field must be set to the first unlocked entry. The Victim field must always be set to a value greater than or equal to the value of the Base field.

Table 3-111 shows the results of attempted access for each mode.

**Table 3-111 Results of access to the TLB Lockdown Register<sup>a</sup>**

TL bit value	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Data TLB Lockdown Register, read or write CP15 with:

MRC p15, 0, <Rd>, c10, c0, 0 ; Read Data TLB Lockdown Register  
MCR p15, 0, <Rd>, c10, c0, 0 ; Write Data TLB Lockdown Register

To access the Instruction TLB Lockdown Register, read or write CP15 with:

MRC p15, 0, <Rd>, c10, c0, 1 ; Read Instruction TLB Lockdown Register  
MCR p15, 0, <Rd>, c10, c0, 1 ; Write Instruction TLB Lockdown Register

### 3.2.57 c10, TLB preload operation

The TLB preload operations are used to load entries into either the instruction or data TLB as specified by the virtual address. The operation performs a TLB lookup to determine if the virtual address has been cached in the TLB array. If the TLB lookup misses in the TLB array, a hardware translation table walk is performed. There are two possible results of the hardware translation table walk:

- the descriptor is cached in the TLB array at the entry specified by the Victim field in the TLB Lockdown Register
- the descriptor faults.

If the operation is a preload D-TLB instruction and the descriptor faults, a data abort is indicated. The DFSR and DFAR indicate the fault type and the fault address, respectively.

If the operation is a preload I-TLB instruction and the descriptor faults, a data abort is indicated. The DFSR indicates the instruction cache maintenance fault value. The DFAR contains the faulty virtual address, and the IFSR contains the fault type encoding.

The TLB preload operations are:

- accessible in privileged modes only, User mode causes Undefined Instruction exception
- supported in both Secure and Nonsecure states
- when CP15 c1 M-bit [0] is LOW, the instruction executes as a NOP
- when CP15 c1 A-bit [1] is HIGH, no alignment faults are generated as VA[1:0] are ignored.

The data and instruction TLB preload operations are as follows:

```
MCR p15, 0, <Rd>, c10, c1, 0 ; Data TLB preload operation
MCR p15, 0, <Rd>, c10, c1, 1 ; Instruction TLB preload operation
```

The TLB preload operation and the TLB lockdown operation can be used to lock entries into the TLB array. Example 3-1 is a code sequence that locks an entry into the TLB array.

---

**Note**

---

This example assumes that the FCSE PID Register is set to 0. If the FCSE PID is not 0, then the MVA of the TLB entry must be invalidated.

---

### Example 3-1 Lock an entry to the instruction TLB array

---

```
LDR r1,=VA           ; Address of entry to lock
MCR p15,0,r1,c8,c5,1 ; Invalidate TLB entry corresponding to VA
LDR r0,=0x00000001   ; base=victim=0 (protect bit=1 [lock])
LDR r2,=0x08400000   ; base=victim=1 (protect bit=0 [unlock])
MCR p15,0,r0,c10,c0,1 ; Write I-TLB Lockdown Register
MCR p15,0,r1,c10,c1,1 ; Prefetch I-TLB
MCR p15,0,r2,c10,c0,1 ; Write I-TLB Lockdown Register
```

---

## 3.2.58 c10, Memory Region Remap Registers

The purpose of the Memory Region Remap Registers is to remap memory region attributes encoded by the TEX[2:0], C, and B bits in the translation tables that the data side, instruction side, and PLE use. See *MMU software-accessible registers* on page 6-8 for information on memory remap.

The Memory Region Remap Registers are:

- two read/write registers banked for the Secure and Nonsecure states:
  - the Primary Region Remap Register
  - the Normal Memory Remap Register.
- accessible in privileged modes only.

These registers apply to all memory accesses and this includes accesses from the data side, instruction side, and PLE. Table 3-113 on page 3-133 shows the purposes of the individual bits in the Primary Region Remap Register. Table 3-115 on page 3-134 shows the purposes of the individual bits in the Normal Memory Remap Register.

**Note**

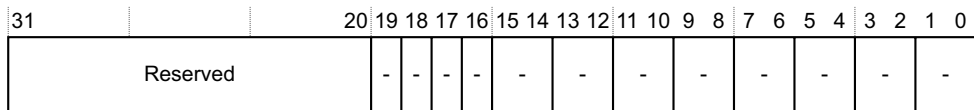
The behavior of the Memory Region Remap Registers depends on the TEX Remap bit, see *c1, Control Register* on page 3-58.

Table 3-112 describes the behavior of memory accesses when the region remapped registers, PRRR and NMRR, are applied.

**Table 3-112 Application of remapped registers on memory access**

CP15 M bit	CP15 TRE bit	Expected behavior
0	0	Memory accesses are controlled by the remapped default memory attributes as defined in the <i>ARM Architecture Reference Manual</i>
0	1	Memory accesses are not remapped but used the default memory attributes
1	0	Memory accesses are not remapped and are controlled by the MMU translation table descriptors
1	1	Memory accesses are controlled by the remapped MMU translation table descriptors

Figure 3-51 shows the bit arrangement of the Primary Region Remap Register.



**Figure 3-51 Primary Region Remap Register format**



Table 3-113 shows how the bit values correspond with the Primary Region Remap Register functions.

**Table 3-113 Primary Region Remap Register bit functions**

Bits	Field	Function <sup>a</sup>
[31:20]	-	Reserved. UNP, SBZ
[19]	-	Remaps shareable attribute when S=1 for Normal regions <sup>b</sup> 1 = reset value
[18]	-	Remaps shareable attribute when S=0 for Normal regions <sup>b</sup> 0 = reset value
[17]	-	Remaps shareable attribute when S=1 for Device regions <sup>b</sup> 0 = reset value
[16]	-	Remaps shareable attribute when S= 0 for Device regions <sup>b</sup> 1= reset value
[15:14]	-	Remaps {TEX[0],C,B} = b111 b10 = reset value
[13:12]	-	Remaps {TEX[0],C,B} = b110 b00 = reset value
[11:10]	-	Remaps {TEX[0],C,B} = b101 b10 = reset value
[9:8]	-	Remaps {TEX[0],C,B} = b100 b10 = reset value
[7:6]	-	Remaps {TEX[0],C,B} = b011 b10 = reset value
[5:4]	-	Remaps {TEX[0],C,B} = b010 b10 = reset value
[3:2]	-	Remaps {TEX[0],C,B} = b001 b01 = reset value
[1:0]	-	Remaps {TEX[0],C,B} = b000 b00 = reset value

a. The reset values ensure that no remapping occurs at reset.

- b. Shareable attributes can map for both shared and nonshared memory. If the shared bit read from the TLB or translation tables is 0, then the bit remaps to the nonshared attributes in this register. If the shared bit read from the TLB or translation tables is 1, then the bit remaps to the shared attributes of this register.

Table 3-114 shows the encoding of the remapping for the primary memory type.

**Table 3-114 Encoding for the remapping of the primary memory type**

Encoding	Memory type
b00	Strongly ordered
b01	Device
b10	Normal
b11	UNP (Normal)

Figure 3-52 shows the bit arrangement of the Normal Memory Remap Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

**Figure 3-52 Normal Memory Remap Register format**

Table 3-115 shows how the bit values correspond with the Normal Memory Remap Register functions.

**Table 3-115 Normal Memory Remap Register bit functions**

Bits	Field	Function <sup>a</sup>
[31:30]	-	Remaps outer attribute for {TEX[0],C,B} = b111 b01 = reset value
[29:28]	-	Remaps outer attribute for {TEX[0],C,B} = b110 b00 = reset value
[27:26]	-	Remaps outer attribute for {TEX[0],C,B} = b101 b01 = reset value
[25:24]	-	Remaps outer attribute for {TEX[0],C,B} = b100 b00 = reset value

**Table 3-115 Normal Memory Remap Register bit functions (continued)**

Bits	Field	Function <sup>a</sup>
[23:22]	-	Remaps outer attribute for {TEX[0],C,B} = b011 b11 = reset value
[21:20]	-	Remaps outer attribute for {TEX[0],C,B} = b010 b10 = reset value
[19:18]	-	Remaps outer attribute for {TEX[0],C,B} = b001 b00 = reset value
[17:16]	-	Remaps outer attribute for {TEX[0],C,B} = b000 b00 = reset value
[15:14]	-	Remaps inner attribute for {TEX[0],C,B} = b111 b01 = reset value
[13:12]	-	Remaps inner attribute for {TEX[0],C,B} = b110 b00 = reset value
[11:10]	-	Remaps inner attribute for {TEX[0],C,B} = b101 b10 = reset value
[9:8]	-	Remaps inner attribute for {TEX[0],C,B} = b100 b00 = reset value
[7:6]	-	Remaps inner attribute for {TEX[0],C,B} = b011 b11 = reset value
[5:4]	-	Remaps inner attribute for {TEX[0],C,B} = b010 b10 = reset value
[3:2]	-	Remaps inner attribute for {TEX[0],C,B} = b001 b00 = reset value
[1:0]	-	Remaps inner attribute for {TEX[0],C,B} = b000 b00 = reset value

a. The reset values ensure that no remapping occurs at reset.

Table 3-116 shows the encoding for the inner or outer cacheable attribute bit fields I0 to I7 and O0 to O7.

**Table 3-116 Remap encoding for inner or outer cacheable attributes**

Encoding	Cacheable attribute
b00	Noncacheable
b01	Write-back, allocate on write
b10	Write-through, no allocate on write
b11	Write-back, no allocate on write

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 3-6.

Table 3-117 shows the results of attempted access for each mode.

**Table 3-117 Results of access to the memory region remap registers<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Memory Region Remap Registers read or write CP15 with:

```
MRC p15, 0, <Rd>, c10, c2, 0 ; Read Primary Region Remap Register
MCR p15, 0, <Rd>, c10, c2, 0 ; Write Primary Region Remap Register
MRC p15, 0, <Rd>, c10, c2, 1 ; Read Normal Memory Remap Register
MCR p15, 0, <Rd>, c10, c2, 1 ; Write Normal Memory Remap Register
```

Memory remap occurs in two stages:

1. The processor uses the Primary Region Remap Register to remap the primary memory type, normal, device, or strongly ordered, and the shareable attribute.
2. For memory regions that the Primary Region Remap Register defines as Normal memory, the processor uses the Normal Memory Remap Register to remap the inner and outer cacheable attributes.

The behavior of the Memory Region Remap Registers depends on the TEX Remap bit, see *c1, Control Register* on page 3-58. If the TEX Remap bit is set to 1, the entries in the Memory Region Remap Registers remap each possible value of the TEX[0], C and B bits in the translation tables. You can therefore set your own definitions for these values. If the TEX Remap bit is cleared to 0, the Memory Region Remap Registers are not used and no memory remapping takes place. See *MMU software-accessible registers* on page 6-8 for more information.

The Memory Region Remap Registers are expected to remain static during normal operation. When you write to the Memory Region Remap Registers, you must invalidate the TLB and perform an IMB operation before you can rely on the new written values. You must also stop the PLE if it is running.

———— **Note** —————

For security reasons, you cannot remap the NS bit.

### 3.2.59 c11, PLE Identification and Status Registers

The purpose of the PLE Identification and Status Registers is to define:

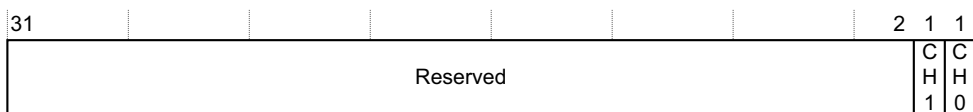
- the PLE channels that are physically implemented on the particular device
- the current status of the PLE channels.

Processes that handle PLE can read this register to determine the physical resources implemented and their availability.

The PLE Identification and Status Register is:

- four read-only registers common to Secure and Nonsecure states
- accessible only in privileged modes.

Figure 3-53 shows the bit arrangement of the PLE Identification and Status Registers 0-3.



**Figure 3-53 PLE identification and Status Registers format**

Table 3-118 shows how the bit values correspond with the PLE Identification and Status Registers functions.

**Table 3-118 PLE Identification and Status Register bit functions**

Bits	Field	Function
[31:2]	-	Reserved. UNP, SBZ.
[1]	CH1	Provides information on PLE Channel 1 functions: 0 = PLE Channel 1 function disabled 1 = PLE Channel 1 function enabled. This is the reset value.
[0]	CH0	Provides information on PLE Channel 0 functions: 0 = PLE Channel 0 function disabled 1 = PLE Channel 0 function enabled. This is the reset value.

Table 3-119 shows the Opcode\_2 values for PLE channel function selection.

**Table 3-119 Opcode\_2 values for PLE Identification and Status Register functions**

Opcode_2	Function
0	Indicates channel present: 0 = channel is not present 1 = channel is present.
1	Reserved. Does not result in an Undefined Instruction exception.
2	Indicates channel running: 0 = channel is not running 1 = channel is running.
3	Indicates channel interrupting: 0 = channel is not interrupting 1 = channel is interrupting, through completion or an error.
4-7	Reserved. Results in an Undefined Instruction exception.

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can only access these registers in privileged modes. Table 3-120 shows the results of attempted access for each mode.

**Table 3-120 Results of access to the PLE Identification and Status Registers<sup>a</sup>**

PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
1	Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Identification and Status Registers in a privileged mode, read CP15 with:

```
MRC p15, 0, <Rd>, c11, c0, 0 ; Read PLE Identification and Status Register present
MRC p15, 0, <Rd>, c11, c0, 2 ; Read PLE Identification and Status Register running
MRC p15, 0, <Rd>, c11, c0, 3 ; Read PLE Identification and Status Register interrupting
```

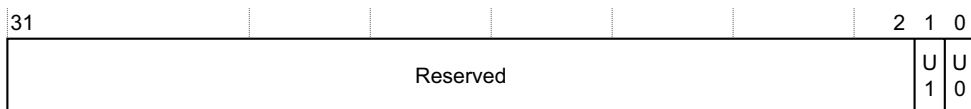
### 3.2.60 c11, PLE User Accessibility Register

The purpose of the PLE User Accessibility Register is to determine if a User mode process can access the registers for each channel. This register contains a bit for each channel, referred to as the U bit for that channel.

The PLE User Accessibility Register is:

- a read/write register common to the Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-54 shows the bit arrangement of the PLE User Accessibility Register.



**Figure 3-54 PLE User Accessibility Register format**

Table 3-121 shows how the bit values correspond with the PLE User Accessibility Register functions.

**Table 3-121 PLE User Accessibility Register bit functions**

Bits	Field	Function
[31:2]	-	Reserved. UNP, SBZP.
[1]	U1	Indicates if a User mode process can access the registers for channel 1: 0 = User mode cannot access channel 1, reset value. User mode accesses cause an Undefined Instruction exception. 1 = User mode can access channel 1.
[0]	U0	Indicates if a User mode process can access the registers for channel 0: 0 = User mode cannot access channel 0, reset value. User mode accesses cause an Undefined Instruction exception. 1 = User mode can access channel 0.

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can only access this register in privileged modes. Table 3-122 shows the results of attempted access for each mode.

**Table 3-122 Results of access to the PLE User Accessibility Register<sup>a</sup>**

PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE User Accessibility Register, read or write CP15 with:

MRC p15, 0, <Rd>, c11, c1, 0 ; Read PLE User Accessibility Register  
MCR p15, 0, <Rd>, c11, c1, 0 ; Write PLE User Accessibility Register

The registers that you can access in User mode when the U1 or U0 bit = 1 for the current channel are:

- *c11, PLE enable commands* on page 3-142
- *c11, PLE Control Register* on page 3-143



- *c11*, PLE Internal Start Address Register on page 3-146
- *c11*, PLE Internal End Address Register on page 3-148
- *c11*, PLE Channel Status Register on page 3-149.

You can access the PLE Channel Number Register, see *c11*, PLE Channel Number Register, in User mode when the U1 or U0 bit for any channel is 1.

The contents of these registers must be preserved on a task switch if the registers are user accessible.

If the U bit for the currently selected channel is set to 0, and a User mode process attempts to access any of these registers, the processor takes an Undefined instruction trap.

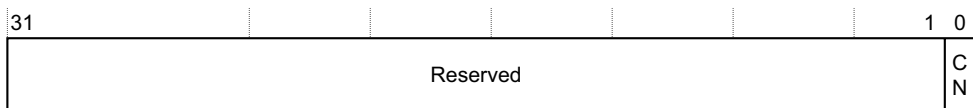
### 3.2.61 c11, PLE Channel Number Register

The purpose of the PLE Channel Number Register is to select a PLE channel.

The PLE Channel Number Register is:

- a read/write register common to Secure and Nonsecure states
- accessible in User and privileged modes.

Figure 3-55 shows the bit arrangement of the PLE Channel Number Register.



**Figure 3-55 PLE Channel Number Register format**

Table 3-123 shows how the bit values correspond with the PLE Channel Number Register functions.

**Table 3-123 PLE Channel Number Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved. UNP, SBZ.
[0]	CN	Indicates PLE channel selected: 0 = PLE channel 0 selected, reset value 1 = PLE channel 1 selected.

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can access this register in User mode if the U bit for any channel is set to 1, see *c11, PLE User Accessibility Register* on page 3-139.

Table 3-124 shows the results of attempted access for each mode.

**Table 3-124 Results of access to the PLE User Accessibility Register<sup>a</sup>**

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	0	Data	Data	Undefined	Undefined	Data	Data	Undefined	Undefined
	1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Channel Number Register, read or write CP15 with:

MRC p15, 0, <Rd>, c11, c2, 0 ; Read PLE Channel Number Register  
MCR p15, 0, <Rd>, c11, c2, 0 ; Write PLE Channel Number Register

### 3.2.62 c11, PLE enable commands

The purpose of the PLE enable commands is to start, stop, or clear PLE transfers for each channel implemented.

The PLE enable commands are:

- three commands for each PLE channel common to Secure and Nonsecure states
- accessible in User and privileged modes.

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can access these commands in User mode if the U bit, see *c11, PLE User Accessibility Register* on page 3-139, for the currently selected channel is set to 1.

Table 3-125 shows the results of attempted access for each mode.

**Table 3-125 Results of access to the PLE enable commands<sup>a</sup>**

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Undefined	Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined
1	0	Undefined	Data	Undefined	Undefined	Undefined	Data	Undefined	Undefined
	1	Undefined	Data	Undefined	Data	Undefined	Data	Undefined	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To issue a PLE enable command, set the PLE Channel Number Register to the appropriate PLE channel and execute one of the following CP15 command:

```
MCR p15, 0, <Rd>, c11, c3, 0 ; Stop PLE enable command
MCR p15, 0, <Rd>, c11, c3, 1 ; Start PLE enable command
MCR p15, 0, <Rd>, c11, c3, 2 ; Clear PLE enable command
```

### 3.2.63 c11, PLE Control Register

The purpose of the PLE Control Register for each channel is to control the operations of that PLE channel.

Table 3-126 on page 3-144 shows the purposes of the individual bits in the PLE Control Register.

The PLE Control Register is:

- one read/write register for each PLE channel common to Secure and Nonsecure states
- accessible in User and privileged modes.

Figure 3-56 on page 3-144 shows the bit arrangement of the PLE Control Register.

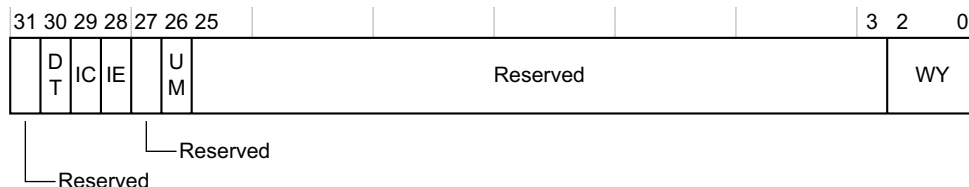


Figure 3-56 PLE Control Register format

Table 3-126 shows how the bit values correspond with the PLE Control Register functions.

Table 3-126 PLE Control Register bit functions

Bits	Field	Function
[31]	-	Reserved. UNP, SBZP.
[30]	DT	Indicates direction of transfer: 0 = transfer from AXI SoC memory to the L2 cache, performing cache linefill. 1 = transfer from L2 cache to AXI SoC memory, performing a cache clean-and-invalidate operation.
[29]	IC	Indicates whether the PLE channel must assert an interrupt on completion of the PLE transfer, or if the Stop command stops the PLE, see <i>c11</i> , <i>PLE enable commands</i> on page 3-142. The interrupt is deasserted from this source, if the processor performs a clear operation on the channel that caused the interrupt. See <i>c11</i> , <i>PLE enable commands</i> on page 3-142 for more information.  <p style="text-align: center;">———— <b>Note</b> ————</p> <p>The U bit has no affect on whether an interrupt is generated on completion.</p> <p>0 = no interrupt on completion 1 = interrupt on completion.</p>
[28]	IE	Indicates that the PLE 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>c11</i> , <i>PLE enable commands</i> on page 3-142 for more information.  <p style="text-align: center;">———— <b>Note</b> ————</p> <p>If the U bit is set to 1, then an interrupt on error occurs regardless of the state of the IE bit. See <i>c11</i>, <i>PLE User Accessibility Register</i> on page 3-139 for information on the U bit.</p> <p>0 = no interrupt on error 1 = interrupt on error.</p>
[27]	-	Reserved. UNP, SBZP.

**Table 3-126 PLE Control Register bit functions (continued)**

Bits	Field	Function
[26]	UM	Indicates that the permission checks are based on the PLE in User or privileged mode. The UM bit is provided so that the privileged mode process can emulate a User mode. See Table 3-127 for more details on the UM bit: 0 = transfer is a privileged transfer 1 = transfer is a User mode transfer.
[25:3]	-	Reserved. UNP, SBZP.
[2:0]	WY	Indicates the selected L2 cache way for filling data. This is used in conjunction with the L2 Cache Lockdown Register: b000 = way 0 b001 = way 1 b010 = way 2 b011 = way 3 b100 = way 4 b101 = way 5 b110 = way 6 b111 = way 7.

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can access this register in User mode if the U bit for the currently selected channel is set to 1, see *c11, PLE User Accessibility Register* on page 3-139.

Table 3-127 shows the behavior of the processor when writing the UM bit [26] for various processor modes and U bit settings.

**Table 3-127 Writing to UM bit [26]**

Data to be written to UM bit [26]	Mode of CP15 instruction	User accessibility (U bit)	Value written to UM bit [26]
b1	User mode	b0	Undefined Instruction exception taken
b0	User mode	b0	Undefined Instruction exception taken
b1	Privileged mode	b0	b1

Table 3-127 Writing to UM bit [26] (continued)

Data to be written to UM bit [26]	Mode of CP15 instruction	User accessibility (U bit)	Value written to UM bit [26]
b0	Privileged mode	b0	b0
b1	User mode	b1	b1
b0	User mode	b1	b1
b1	Privileged mode	b1	b1
b0	Privileged mode	b1	b1

Table 3-128 shows the results of attempted access for each mode.

Table 3-128 Results of access to the PLE Control Registers<sup>a</sup>

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	0	Data	Data	Undefined	Undefined	Data	Data	Undefined	Undefined
	1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Control Register, set the PLE Channel Number Register to the appropriate PLE channel and read or write CP15 with:

MRC p15, 0, <Rd>, c11, c4, 0 ; Read PLE Control Register

MCR p15, 0, <Rd>, c11, c4, 0 ; Write PLE Control Register

While the channel has the status of Running, any attempt to write to the PLE Control Register results in architecturally Unpredictable behavior. For the processor, writes to the PLE Control Register have no effect when the PLE channel is running.

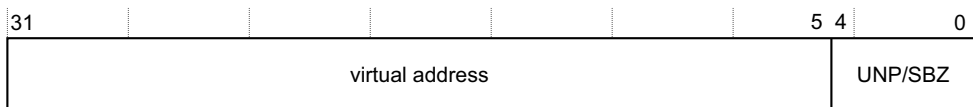
### 3.2.64 c11, PLE Internal Start Address Register

The purpose of the PLE Internal Start Address Register for each channel is to define the start address, that is, the first address that data transfers go to or from.

The PLE Internal Start Address Register is:

- a 32-bit read/write register with one register for each PLE channel common to Secure and Nonsecure states
- accessible in User and privileged modes.

The PLE Internal Start Address Register bits [31:0] contain the Internal Start *Virtual Address* (VA). Figure 3-57 shows this format.



**Figure 3-57 PLE Internal Start Address Register bit format**

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. The processor can access this register in User mode if the U bit for the currently selected channel is set to 1, see *c11, PLE User Accessibility Register* on page 3-139.

Table 3-129 shows the results of attempted access for each mode.

**Table 3-129 Results of access to the PLE Internal Start Address Register<sup>a</sup>**

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	0	Data	Data	Undefined	Undefined	Data	Data	Undefined	Undefined
	1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Internal Start Address Register, set the PLE Channel Number Register to the appropriate PLE channel and read or write CP15 c11 with:

```
MRC p15, 0, <Rd>, c11, c5, 0 ; Read PLE Internal Start Address Register
MCR p15, 0, <Rd>, c11, c5, 0 ; Write PLE Internal Start Address Register
```





Table 3-131 shows the results of attempted access for each mode.

**Table 3-131 Results of access to the PLE Internal End Address Register<sup>a</sup>**

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined
1	0	Data	Data	Undefined	Undefined	Data	Data	Undefined	Undefined
	1	Data	Data	Data	Data	Data	Data	Data	Data

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Internal End Address Register, set the PLE Channel Number Register to the appropriate PLE channel and read or write CP15 with:

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

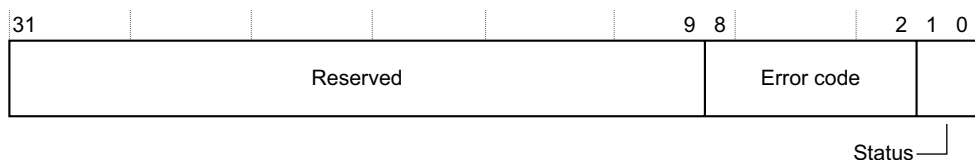
### 3.2.66 c11, PLE Channel Status Register

The purpose of the PLE Channel Status Register for each channel is to define the status of the most recently started PLE operation on that channel.

The PLE Channel Status Register is:

- one read-only register for each PLE channel common to Secure and Nonsecure states
- accessible in User and privileged modes.

Figure 3-59 shows the bit arrangement of the PLE Channel Status Register.



**Figure 3-59 PLE Channel Status Register format**

Table 3-132 shows how the bit values correspond with the PLE Channel Status Register functions.

**Table 3-132 PLE Channel Status Register bit functions**

Bits	Field	Function
[31:9]	-	Reserved. UNP, SBZ.
[8:2]	Error code	<p>Indicates the status of the external address error. All other encodings are reserved:</p> <p>b0xxxxxx = no error</p> <p>b1x01100 = precise external abort, L1 translation</p> <p>b1x01110 = precise external abort, L2 translation</p> <p>b1011100 = parity/ECC error on L1 translation</p> <p>b1011110 = parity/ECC error on L2 translation</p> <p>b1000101 = translation fault, section</p> <p>b1000111 = translation fault, page</p> <p>b1000011 = access flag fault, section</p> <p>b1000110 = access flag fault, page</p> <p>b1001001 = domain fault, section</p> <p>b1001011 = domain fault, page</p> <p>b1001101 = permission fault, section</p> <p>b1001111 = permission fault, page</p> <p>b1x10110 = imprecise external abort</p> <p>b1011000 = imprecise parity or ECC error, nontranslation.</p> <p>Any unused encoding not listed is reserved.</p> <p>Where <i>x</i> represents bit [7] in the encoding, bit [7] can be either:</p> <p>0 = AXI Decode error caused the abort, reset value</p> <p>1 = AXI Slave error caused the abort.</p>
[1:0]	Status	<p>Indicates the status of the PLE channel:</p> <p>b00 = idle, reset value</p> <p>b01 = reserved</p> <p>b10 = running</p> <p>b11 = complete or error.</p>

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. You can access these registers in User mode if the U bit for the currently selected channel is set to 1, see *c11, PLE User Accessibility Register* on page 3-139.

Table 3-133 shows the results of attempted access for each mode.

**Table 3-133 Results of access to the PLE Channel Status Register<sup>a</sup>**

U bit	PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
		Read	Write	Read	Write	Read	Write	Read	Write
0	0	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
	1	Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined
1	0	Data	Undefined	Undefined	Undefined	Data	Undefined	Undefined	Undefined
	1	Data	Undefined	Data	Undefined	Data	Undefined	Data	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Channel Status Register, set PLE Channel Number Register to the appropriate PLE channel and read CP15 with:

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

For more details on the operation of the L2 *PreLoad Engine* (PLE), see *L2 PLE* on page 8-6.

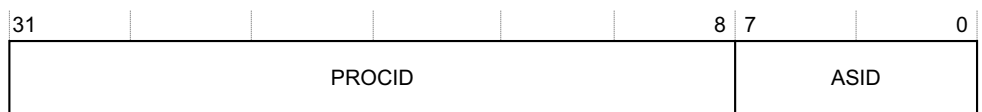
### 3.2.67 c11, PLE Context ID Register

The PLE Context ID Register for each channel contains the processor context ID of the process that uses that channel.

The PLE Context ID Register is:

- a read/write register for each PLE channel common to Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-60 shows the bit arrangement of the PLE Context ID Register.



**Figure 3-60 PLE Context ID Register format**

Table 3-134 shows how the bit values correspond with the PLE Context ID Register functions.

**Table 3-134 PLE Context ID Register bit functions**

Bits	Field	Function
[31:8]	PROCID	Extends the ASID to form the process ID and identifies the current process
[7:0]	ASID	Holds the ASID of the current process and identifies the current ASID

Access in the Nonsecure state depends on the PLE bit, see *c1, Nonsecure Access Control Register* on page 3-73. Table 3-135 shows the results of attempted access for each mode.

**Table 3-135 Results of access to the PLE Context ID Register<sup>a</sup>**

PLE bit	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
0	Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
1	Data	Data	Data	Data	Undefined	Undefined	Undefined	Undefined

a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the PLE Context ID Register in a privileged mode, set the PLE Channel Number Register to the appropriate PLE channel and read or write CP15 with:

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

### 3.2.68 c12, Secure or Nonsecure Vector Base Address Register

The purpose of the Secure or Nonsecure Vector Base Address Register is to hold the base address for exception vectors in the Secure and Nonsecure states. See *Exceptions* on page 2-35 for more information.

The Secure or Nonsecure Vector Base Address Register is:

- a read/write register banked in Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-61 on page 3-153 shows the bit arrangement of the Secure or Nonsecure Vector Base Address Register.



Table 3-137 shows the results of attempted access for each mode.

**Table 3-137 Results of access to the Secure or Nonsecure Vector Base Address Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Secure or Nonsecure Vector Base Address Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c12, c0, 0 ; Read Secure or Nonsecure Vector Base
                               ; Address Register
```

```
MCR p15, 0, <Rd>, c12, c0, 0 ; Write Secure or Nonsecure Vector Base
                               ; Address Register
```

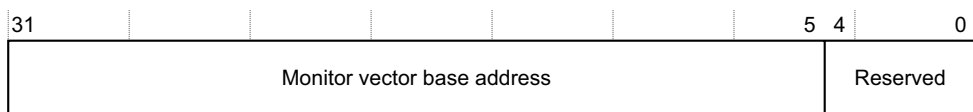
### 3.2.69 c12, Monitor Vector Base Address Register

The purpose of the Monitor Vector Base Address Register is to hold the base address for the Monitor mode exception vector. See *Exceptions* on page 2-35 for more information.

The Monitor Vector Base Address Register is:

- a read/write register in the Secure state only
- accessible in secure privileged modes only.

Figure 3-62 shows the bit arrangement of the Monitor Vector Base Address Register.



**Figure 3-62 Monitor Vector Base Address Register format**

Table 3-138 shows how the bit values correspond with the Monitor Vector Base Address Register functions.

**Table 3-138 Monitor Vector Base Address Register bit functions**

Bits	Field	Function
[31:5]	Monitor vector base address	Holds the base address. Determines the location that the core branches to, on a Monitor mode exception. The reset value is 0.
[4:0]	-	Reserved. UNP, SBZ.

When an exception branches to the Monitor mode, the core branches to address:

$\text{Monitor\_Base\_Address} + \text{Exception\_Vector\_Address}$ .

The Software Monitor Exception caused by an SMC instruction branches to Monitor mode. You can configure IRQ, FIQ, and External abort exceptions to branch to Monitor mode, see *c1, Secure Configuration Register* on page 3-70. These are the only exceptions that can branch to Monitor mode and that use the Monitor Vector Base Address Register to calculate the branch address. See *Exceptions* on page 2-35 for more information.

———— **Note** ————

The Monitor Vector Base Address Register is  $0x00000000$  at reset. The secure boot code must program the register with an appropriate value for the Monitor.

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 3-6.

Table 3-139 shows the results of attempted access for each mode.

**Table 3-139 Results of access to the Monitor Vector Base Address Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Data	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the Monitor Vector Base Address Register, read or write CP15 with:

MRC p15, 0, <Rd>, c12, c0, 1 ; Read Monitor Vector Base Address Register  
 MCR p15, 0, <Rd>, c12, c0, 1 ; Write Monitor Vector Base Address Register

### 3.2.70 c12, Interrupt Status Register

The purpose of the Interrupt Status Register is to:

- reflect the state of the **nFIQ** and **nIRQ** pins on the processor
- reflect the state of external aborts.

The Interrupt Status Register is:

- a read-only register common to Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-63 shows the bit arrangement of the Interrupt Status Register.



**Figure 3-63 Interrupt Status Register format**

Table 3-140 shows how the bit values correspond with the Interrupt Status Register functions.

**Table 3-140 Interrupt Status Register bit functions**

Bits	Field	Function <sup>a</sup>
[31:9]	-	Reserved. UNP, SBZ.
[8]	A	Indicates when an external abort is pending: 0 = no abort, reset value 1 = abort pending.
[7]	I	Indicates when an IRQ is pending: 0 = no IRQ, reset value 1 = IRQ pending.
[6]	F	Indicates when an FIQ is pending: 0 = no FIQ, reset value 1 = FIQ pending.
[5:0]	-	Reserved. UNP, SBZ.

a. The reset values depend on external signals.



**Note**

- The F and I bits directly reflect the state of the **nFIQ** and **nIRQ** pins respectively, but are the inverse state.
- The A bit is set to 1 when an external abort occurs and automatically clears to 0 when the abort is taken.

Table 3-141 shows the results of attempted access for each mode.

**Table 3-141 Results of access to the Interrupt Status Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Data	Undefined	Data	Undefined	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

The A, I, and F bits map to the same format as the CPSR so that you can use the same mask for these bits.

The Monitor can poll these bits to detect the exceptions before it completes context switches. This can reduce interrupt latency.

To access the Interrupt Status Register, read CP15 with:

```
MRC p15, 0, <Rd>, c12, c1, 0 ; Read Interrupt Status Register
```

### 3.2.71 c13, FCSE PID Register

The *c13, Context ID Register* on page 3-160 replaces the FCSE PID Register. Use of the FCSE PID Register is deprecated.

The FCSE PID Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-64 on page 3-158 shows the bit arrangement of the FCSE PID Register.

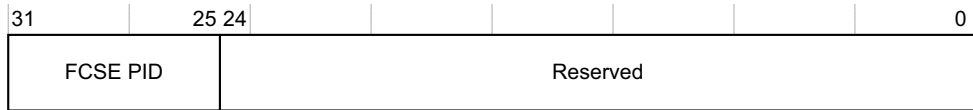


Figure 3-64 FCSE PID Register format

Table 3-142 shows how the bit values correspond with the FCSE PID Register functions.

Table 3-142 FCSE PID Register bit functions

Bits	Field	Function
[31:25]	FCSE PID	Holds the ProcID. Identifies a specific process for fast context switch. The reset value is 0. The purpose of the FCSE PID Register is to provide the ProcID for fast context switch memory mappings. The MMU uses the contents of this register to map memory addresses in the range 0-32MB.
[24:0]	-	Reserved. UNP, SBZP.

Attempts to write to this register in secure privileged mode when **CP15SDISABLE** is HIGH result in an Undefined Instruction exception, see *Security Extensions write access disable* on page 3-6.

Table 3-143 shows the results of attempted access for each mode.

Table 3-143 Results of access to the FCSE PID Register<sup>a</sup>

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

To access the FCSE PID Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c13, c0, 0 ; Read FCSE PID Register
MCR p15, 0, <Rd>, c13, c0, 0 ; Write FCSE PID Register
```

To change the ProcID and perform a fast context switch, write to the FCSE PID Register. You are not required to flush the contents of the TLB after the switch because the TLB still holds the valid address tags.

Because a write to the FCSE PID Register causes a pipeline flush, the effect is immediate. The next executed instruction is fetched with the new PID.

———— **Note** —————

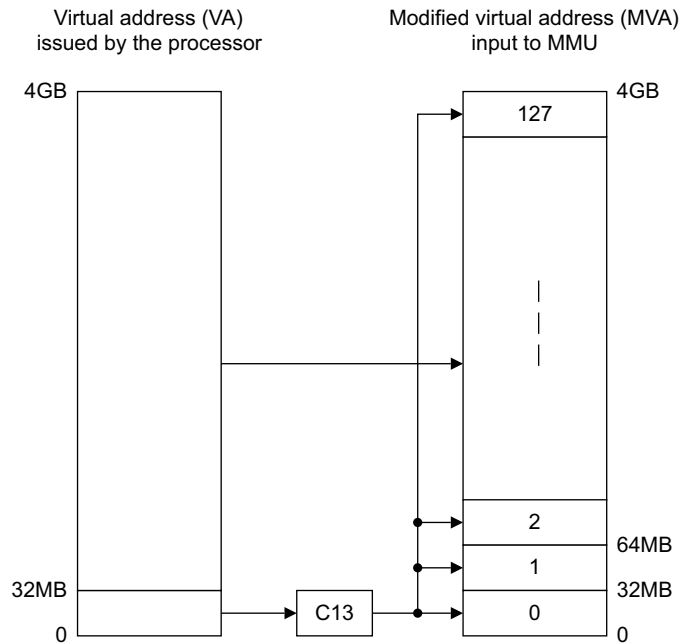
You must not rely on this behavior for future compatibility. An IMB must be executed between changing the ProcID and fetching from locations that are translated by the ProcID.

Addresses issued by the processor in the range 0-32MB are translated by the ProcID. Address A becomes  $A + (\text{ProcID} \times 32\text{MB})$ . The MMU uses this translated address, the MVA. Addresses above 32MB are not translated. The ProcID is a 7-bit field, enabling 128 x 32MB processes to be mapped.

———— **Note** —————

If ProcID is 0, as it is on Reset, then there is a flat mapping between the processor and the MMU.

Figure 3-65 shows how addresses are mapped using the FCSE PID Register.



**Figure 3-65 Address mapping with the FCSE PID Register**

### 3.2.72 c13, Context ID Register

The purpose of the Context ID Register is to provide information on the current ASID and process ID, for example for the ETM and debug logic.

Debug logic uses the ASID information to enable process-dependent breakpoints and watchpoints. The ASID field of the Context ID Register and FCSE PID Register cannot be used simultaneously. The FCSE PID Register remapping of VA to MVA has priority over setting the ASID field to designate non-global pages. Therefore, non-global pages cannot be used if the FCSE PID Register is set to a non-zero value.

The Context ID Register is:

- a read/write register banked for Secure and Nonsecure states
- accessible in privileged modes only.

Figure 3-66 shows the bit arrangement of the Context ID Register.



**Figure 3-66 Context ID Register format**

Table 3-144 shows how the bit values correspond with the Context ID Register functions.

**Table 3-144 Context ID Register bit functions**

Bits	Field	Function
[31:8]	PROCID	Extends the ASID to form the process ID and identifies the current process. The reset value is 0.
[7:0]	ASID	Holds the ASID of the current process to identify the current ASID. The reset value is 0.

Table 3-145 shows the results of attempted access for each mode.

**Table 3-145 Results of access to the Context ID Register<sup>a</sup>**

Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
Read	Write	Read	Write	Read	Write	Read	Write
Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- a. An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.

The current ASID value in the Context ID Register is exported to the MMU.

To access the Context ID Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c13, c0, 1 ; Read Context ID Register
MCR p15, 0, <Rd>, c13, c0, 1 ; Write Context ID Register
```

You must ensure that software executes a Data Synchronization Barrier operation before changes to this register. This ensures that all accesses are related to the correct context ID.

You must execute an IMB instruction immediately after changes to the Context ID Register. You must not attempt to execute any instructions that are from an ASID-dependent memory region between the change to the register and the IMB instruction. Code that updates the ASID must execute from a global memory region.

You must program each process with a unique number to ensure that the ETM and debug logic can correctly distinguish between processes.

### 3.2.73 c13, Thread and Process ID Registers

The purpose of the Thread and Process ID Registers is to provide locations to store the IDs of software threads and processes for OS management purposes.

The Thread and Process ID Registers are:

- three read/write registers banked for Secure and Nonsecure states:
  - user read/write Thread and Process ID Register
  - user read-only Thread and Process ID Register
  - privileged only Thread and Process ID Register.
- accessible in different modes:
  - the user read/write Thread and Process ID Register is read/write in User and privileged modes
  - the user read-only Thread and Process ID Register is read-only in User mode, and read/write in privileged modes
  - the privileged only Thread and Process ID Register is only accessible in privileged modes, and is read/write.

Table 3-146 shows the results of attempted access to each register for each mode.

**Table 3-146 Results of access to the Thread and Process ID Registers<sup>a</sup>**

Register <sup>b</sup>	Secure privileged		Nonsecure privileged		Secure User		Nonsecure User	
	Read	Write	Read	Write	Read	Write	Read	Write
User read/write	Secure data	Secure data	Nonsecure data	Nonsecure data	Secure data	Secure data	Nonsecure data	Nonsecure data
User read-only	Secure data	Secure data	Nonsecure data	Nonsecure data	Secure data	Undefined	Nonsecure data	Undefined
Privileged only	Secure data	Secure data	Nonsecure data	Nonsecure data	Undefined	Undefined	Undefined	Undefined

- An entry of Undefined in the table means that the access gives an Undefined Instruction exception when the coprocessor instruction is executed.
- Each row refers to a Thread and Process ID Register. For example, the User read/write row refers to the User read/write Thread and Process ID Register.

To access the Thread and Process ID Registers, read or write CP15 with:

```
MRC p15, 0, <Rd>, c13, c0, 2 ; Read User read/write Thread and Process ID Register
MCR p15, 0, <Rd>, c13, c0, 2 ; Write User read/write Thread and Process ID Register
MRC p15, 0, <Rd>, c13, c0, 3 ; Read User read-only Thread and Process ID Register
MCR p15, 0, <Rd>, c13, c0, 3 ; Write User read-only Thread and Process ID Register
MRC p15, 0, <Rd>, c13, c0, 4 ; Read Privileged only Thread and Process ID Register
MCR p15, 0, <Rd>, c13, c0, 4 ; Write Privileged only Thread and Process ID Register
```

Reading or writing the Thread and Process ID Registers has no effect on the processor state or operation. These registers provide OS support and must be managed by the OS.

You must clear the contents of all Thread and Process ID Registers on process switches to prevent data leaking from one process to another. This is important to ensure the security of secure data.

### 3.2.74 c15, L1 system array debug data registers

The purpose of the L1 system array debug data registers is to hold the data:

- that is returned on instruction side or data side TLB CAM, TLB ATTR, TLB PA, HVAB, tag, data, GHB, and BTB instruction or data side read operations
- for TLB CAM, TLB ATTR, TLB PA, HVAB, tag, data, GHB, and BTB instruction side or data side write operations.

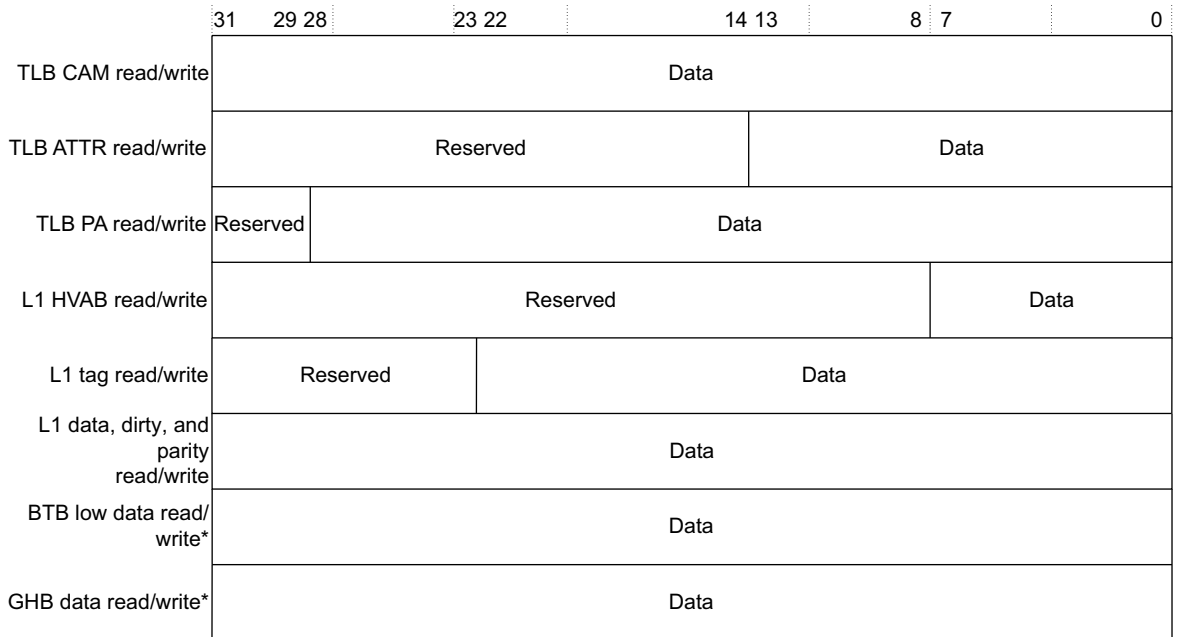
Because BTB, TLB, and data arrays are greater than 32-bits wide, the processor contains two registers, data low register and data high register, to hold data when retrieving or registering data as a result of read/write operations. If the data is greater than 32-bit wide, both the low and high registers are used to transfer data. Otherwise, only the low register is used to transfer data.

The Data 0 and Data 1 read/write registers are accessible in secure privileged modes only.

To access the L1 system debug registers, read or write CP15 with:

```
MCR p15, 0, <Rd>, c15, c0, 0 ; Write data L1 Data 0 Register
MRC p15, 0, <Rd>, c15, c0, 0 ; Read data L1 Data 0 Register
MCR p15, 0, <Rd>, c15, c0, 1 ; Write data L1 Data 1 Register
MRC p15, 0, <Rd>, c15, c0, 1 ; Read data L1 Data 1 Register
MCR p15, 0, <Rd>, c15, c1, 0 ; Write instruction L1 Data 0 Register
MRC p15, 0, <Rd>, c15, c1, 0 ; Read instruction L1 Data 0 Register
MCR p15, 0, <Rd>, c15, c1, 1 ; Write instruction L1 Data 1 Register
MRC p15, 0, <Rd>, c15, c1, 1 ; Read instruction L1 Data 1 Register
```

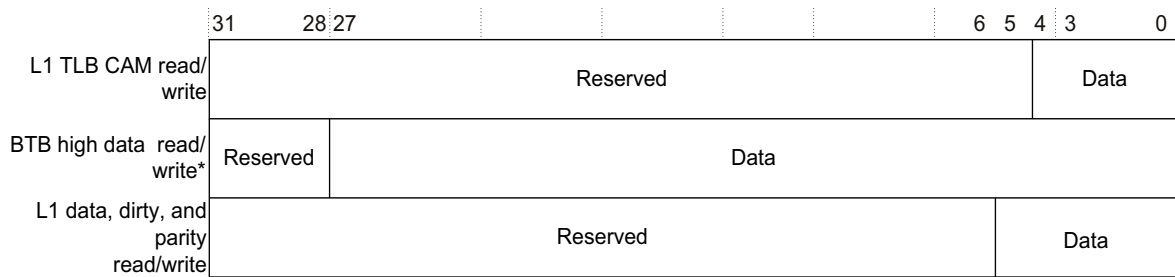
Figure 3-67 shows the bit arrangement of the L1 Data 0 Register when retrieving or registering data as a result of the read/write operations.



\*Level one instruction side read/write data only

**Figure 3-67 Instruction and Data side Data 0 Registers format**

Figure 3-68 shows the bit arrangement of the L1 Data 1 Register when retrieving or registering data as a result of the read/write operations.



\*Level one instruction side read/write data only

**Figure 3-68 Instruction and Data side Data 1 Registers format**

Table 3-147 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register functions as a result of a TLB CAM read/write operation.

**Table 3-147 Functional bits of I-L1 or D-L1 Data 0 Register for a TLB CAM operation**

Bits	Field	Function
[31:0]	Data	Holds TLB CAM information.

Table 3-148 shows how the bit values correspond with the I-L1 or D-L1 Data 1 Register functions as a result of a TLB CAM read/write operation.

**Table 3-148 Functional bits of I-L1 or D-L1 Data 1 Register for a TLB CAM operation**

Bits	Field	Function
[31:5]	-	Reserved. UNP, SBZ.
[4:0]	Data	Holds TLB CAM information.

To perform a TLB CAM operation, read or write CP15 with:

```
MCR p15, 0, <Rd>, c15, c0, 2 ; D-L1 CAM write
MCR p15, 0, <Rd>, c15, c2, 2 ; D-L1 CAM read
MCR p15, 0, <Rd>, c15, c1, 2 ; I-L1 CAM write
MCR p15, 0, <Rd>, c15, c3, 2 ; I-L1 CAM read
```



Table 3-149 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register functions as a result of a TLB ATTR read/write operation.

**Table 3-149 Functional bits of I-L1 or D-L1 Data 0 Register for a TLB ATTR operation**

Bits	Field	Function
[31:12]	-	Reserved. UNP, SBZ.
[11:0]	Data	Holds TLB ATTR information.

To perform a TLB ATTR operation, read or write CP15 with:

MCR p15 0, <Rd>, c15, c0, 3 ; D-L1 TLB ATTR write  
MCR p15 0, <Rd>, c15, c2, 3 ; D-L1 TLB ATTR read  
MCR p15 0, <Rd>, c15, c1, 3 ; I-L1 TLB ATTR write  
MCR p15 0, <Rd>, c15, c3, 3 ; I-L1 TLB ATTR read

Table 3-150 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register as a result of a TLB PA array read/write operation.

**Table 3-150 Functional bits of I-L1 or D-L1 Data 0 Register for a TLB PA array operation**

Bits	Field	Function
[31:29]	-	Reserved. UNP, SBZ.
[28:0]	Data	Holds TLB PA information.

To perform a TLB PA operation, read or write CP15 with:

MCR p15 0, <Rd>, c15, c0, 4 ; D-L1 TLB PA write  
MCR p15 0, <Rd>, c15, c2, 4 ; D-L1 TLB PA read  
MCR p15 0, <Rd>, c15, c1, 4 ; I-L1 TLB PA write  
MCR p15 0, <Rd>, c15, c3, 4 ; I-L1 TLB PA read

Table 3-151 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register as a result of a HVAB array read/write operation.

**Table 3-151 Functional bits of I-L1 or D-L1 Data 0 Register for an HVAB array operation**

Bits	Field	Function
[31:8]	-	Reserved. UNP, SBZ.
[7:0]	Data	Holds HVAB information.

To perform a HVAB operation, read or write CP15 with:

MCR p15 0, <Rd>, c15, c0, 5 ; D-L1 HVAB write  
MCR p15 0, <Rd>, c15, c2, 5 ; D-L1 HVAB read  
MCR p15 0, <Rd>, c15, c1, 5 ; I-L1 HVAB write  
MCR p15 0, <Rd>, c15, c3, 5 ; I-L1 HVAB read

Table 3-152 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register as a result of an L1 tag array read/write operation.

**Table 3-152 Functional bits of I-L1 or D-L1 Data 0 Register for an L1 tag array operation**

Bits	Field	Function
[31:23]	-	Reserved. UNP, SBZ.
[22:0]	Data	Holds L1 tag array information.

To perform a tag operation, read or write CP15 with:

MCR p15 0, <Rd>, c15, c0, 6 ; D-L1 tag write  
MCR p15 0, <Rd>, c15, c2, 6 ; D-L1 tag read  
MCR p15 0, <Rd>, c15, c1, 6 ; I-L1 tag write  
MCR p15 0, <Rd>, c15, c3, 6 ; I-L1 tag read

Table 3-153 shows how the bit values correspond with the I-L1 or D-L1 Data 0 Register as a result of an L1 data array read/write operation.

**Table 3-153 Functional bits of I-L1 or D-L1 Data 0 Register for L1 data array operation**

Bits	Field	Function
[31:0]	Data	Holds L1 data array information.

Table 3-154 shows how the bit values correspond with the I-L1 or D-L1 Data 1 Register as a result of an L1 data array read/write operation.

**Table 3-154 Functional bits of I-L1 or D-L1 Data 1 Register for L1 data array operation**

Bits	Field	Function
[30:6]	-	Reserved. UNP, SBZ.
[5:0]	Data	Holds L1 data array information.

To perform a data operation on the Data 0 or Data 1 Register, read or write CP15 with:

MCR p15 0, <Rd>, c15, c0, 7 ; D-L1 data write  
MCR p15 0, <Rd>, c15, c2, 7 ; D-L1 data read  
MCR p15 0, <Rd>, c15, c1, 7 ; I-L1 data write  
MCR p15 0, <Rd>, c15, c3, 7 ; I-L1 data read

Table 3-155 shows how the bit values correspond with the I-L1 Data 0 Register as a result of a BTB array read/write operation.

**Table 3-155 Functional bits of I-L1 Data 0 Register for a BTB array operation**

Bits	Field	Function
[31:0]	Data	Holds L1 Data 0 Register BTB information

Table 3-156 shows how the bit values correspond with the I-L1 Data 1 Register as a result of a BTB array read/write operation.

**Table 3-156 Functional bits of I-L1 Data 1 Register for a BTB array operation**

Bits	Field	Function
[31:28]	-	Reserved. UNP, SBZ.
[27:0]	Data	Holds L1 Data 1 Register BTB information.

To perform a BTB operation on the Data 0 or Data 1 Register, read or write CP15 with:

MCR p15 0, <Rd>, c15, c5, 3 ; I-L1 BTB write  
MCR p15 0, <Rd>, c15, c7, 3 ; I-L1 BTB read

Table 3-157 shows how the bit values correspond with the I-L1 Data 0 Register as a result of a GHB array read/write operation.

**Table 3-157 Functional bits of I-L1 Data 0 Register for a GHB array operation**

Bits	Field	Function
[31:0]	Data	Holds L1 Data 0 Register GHB information

To perform a GHB operation on the Data 0 Register, read or write CP15 with:

MCR p15 0, <Rd>, c15, c5, 2 ; I-L1 GHB write  
MCR p15 0, <Rd>, c15, c7, 2 ; I-L1 GHB read

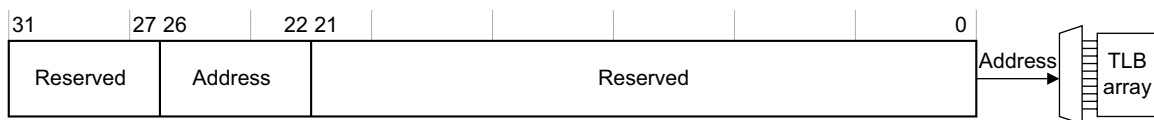
### 3.2.75 c15, L1 TLB operations

The purpose of the L1 TLB operations is to:

- read the instruction or data side L1 TLB array contents and write into the system debug data registers
- write into the system debug data registers and instruction or data side L1 TLB array.

The L1 TLB operations are accessible in secure privileged modes only.

Figure 3-69 shows the bit arrangement of the L1 TLB CAM read operations.



**Figure 3-69 L1 TLB CAM read operation format**

Figure 3-70 on page 3-169 shows the bit arrangement of the L1 TLB CAM write operations.

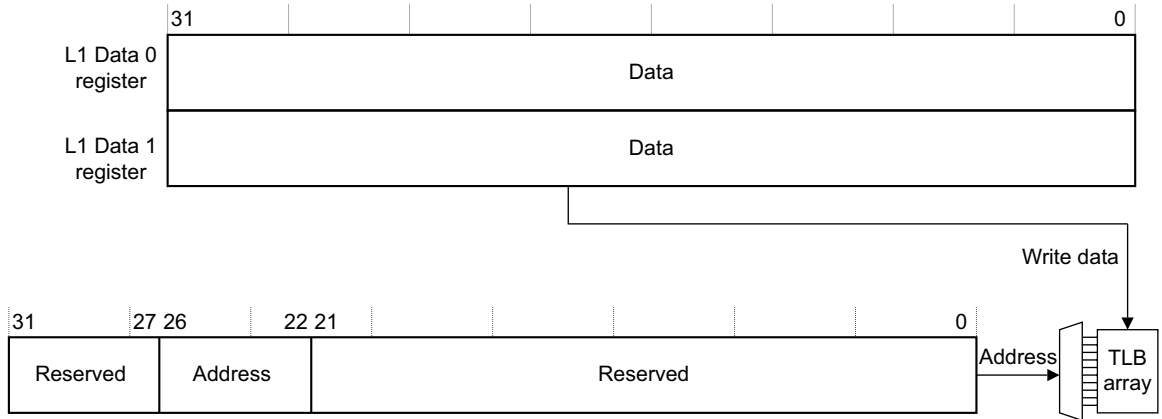


Figure 3-70 L1 TLB CAM write operation format

### TLB CAM array examples

To write one entry in data side TLB CAM array, for example:

```
LDR R0, =0x03000323;
MCR p15, 0, R0, c15, c0, 0;    Move R0 to D-L1 Data 0 Register
LDR R2, =0x01;
MCR p15, 0, R2, c15, c0, 1;    Move R0 to D-L1 Data 1 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c0, 2;    Write D-L1 Data 0 or 1 Register to D-TLB CAM
```

To read one entry in data side TLB CAM array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c2, 2;    Read D-TLB CAM into data L1 Data 0/1 Register
MRC p15, 0, R0, c15, c0, 0;    Move D-L1 Data 0 Register to R0
MRC p15, 0, R2, c15, c0, 1;    Move D-L1 Data 1 Register to R2
```

To write one entry in instruction side TLB CAM array, for example:

```
LDR R0, =0x03000323;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R2, =0x01;
MCR p15, 0, R2, c15, c1, 1;    Move R0 to I-L1 Data 1 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c1, 2;    Write I-L1 Data 0 or 1 Register to D-TLB CAM
```

To read one entry in instruction side TLB CAM array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c3, 2;    Read I-TLB CAM into data L1 Data 0/1 Register
MRC p15, 0, R0, c15, c1, 0;    Move I-L1 Data 0 Register to R0
```

MRC p15, 0, R2, c15, c1, 1;      Move I-L1 Data 1 Register to R2

### TLB ATTR array examples

To write one entry in data side TLB ATTR array, for example:

```
LDR R0, =0x252E;
MCR p15, 0, R0, c15, c0, 0;      Move R0 to D-L1 Data 0 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c0, 3;      Write D-L1 Data 0 Register to D-TLB ATTR
```

To read one entry in data side TLB ATTR array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c2, 3;      Read D-TLB ATTR into data L1 Data 0 Register
MRC p15, 0, R0, c15, c0, 0;      Move D-L1 Data 0 Register to R0
```

To write one entry in instruction side TLB ATTR array, for example:

```
LDR R0, =0x252E;
MCR p15, 0, R0, c15, c1, 0;      Move R0 to I-L1 Data 0 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c1, 3;      Write I-L1 Data 0 Register to I-TLB ATTR
```

To read one entry in instruction side TLB ATTR array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c3, 3;      Read I-TLB ATTR into data L1 Data 0 Register
MRC p15, 0, R0, c15, c0, 0;      Move I-L1 Data 0 Register to R0
```

### TLB PA array examples

To write one entry in data side TLB PA array, for example:

```
LDR R0, =0x05730000;
MCR p15, 0, R0, c15, c0, 0;      Move R0 to D-L1 Data 0 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c0, 4;      Write D-L1 Data 0 Register to D-TLB PA
```

To read one entry in data side TLB PA array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c2, 4;      Read D-TLB PA into data L1 Data 0 Register
MRC p15, 0, R0, c15, c0, 0;      Move D-L1 Data 0 Register to R0
```

To write one entry in instruction side TLB PA array, for example:

```
LDR R0, =0x05730000;
MCR p15, 0, R0, c15, c1, 0;      Move R0 to I-L1 Data 0 Register
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c1, 4;      Write I-L1 Data 0 Register to I-TLB PA
```

To read one entry in instruction side TLB PA array, for example:

```
LDR R1, =0x00C00000;
MCR p15, 0, R1, c15, c3, 4;    Read I-TLB PA into data L1 Data 0 Register
MRC p15, 0, R0, c15, c1, 0;    Move I-L1 Data 0 Register to R0
```

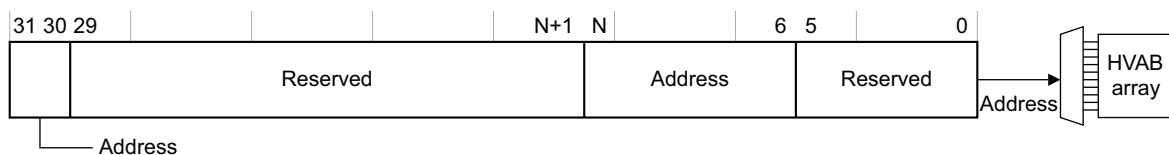
### 3.2.76 c15, L1 HVAB array operations

The purpose of the L1 HVAB array operations is to:

- read the L1 HVAB array contents and write to the system debug data registers
- write into the system debug data registers and into the L1 HVAB array.

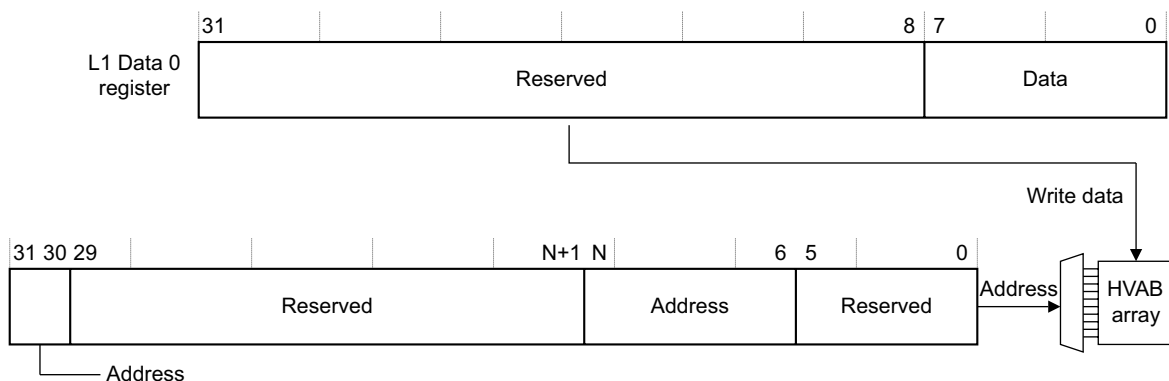
The L1 HVAB array operation is accessible in secure privileged modes only. You can calculate the value of N in Figure 3-71 and Figure 3-72 using the NumSets and LineSize fields as defined in Table 3-42 on page 3-56.

Figure 3-71 shows the bit arrangement of the L1 HVAB array read operation.



**Figure 3-71 L1 HVAB array read operation format**

Figure 3-72 shows the bit arrangement of the L1 HVAB array write operation.



**Figure 3-72 L1 HVAB array write operation format**

To write one entry in data side HVAB array, for example:

```

LDR R0, =0x47;
MCR p15, 0, R0, c15, c0, 0;    Move R0 to D-L1 Data 0 Register
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c0, 5;    Write D-L1 Data 0 Register to D-HVAB

```

To read one entry from data side HVAB array, for example:

```

LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c2, 5;    Read D-HVAB into data L1 Data 0 Register
MRC p15, 0, R2, c15, c0, 0;    Move D-L1 Data 0 Register to R2

```

To write one entry in instruction side HVAB array, for example:

```

LDR R0, =0x47;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c1, 5;    Write I-L1 Data 0 Register to I-HVAB

```

To read one entry from instruction side HVAB array, for example:

```

LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c3, 5;    Read I-HVAB into I-L1 Data 0 Register
MRC p15, 0, R2, c15, c1, 0;    Move I-L1 Data 0 Register to R2

```

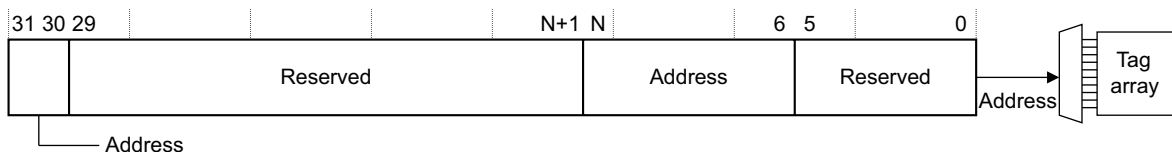
### 3.2.77 c15, L1 tag array operations

The purpose of the L1 tag array operations is to:

- read the L1 tag array contents and write into the system debug data registers
- write into the system debug data registers and copy into the L1 tag array.

The L1 tag array operation is accessible in secure privileged modes only. You can calculate the value of N in Figure 3-73 and Figure 3-74 on page 3-173 using the NumSets and LineSize fields as defined in Table 3-42 on page 3-56.

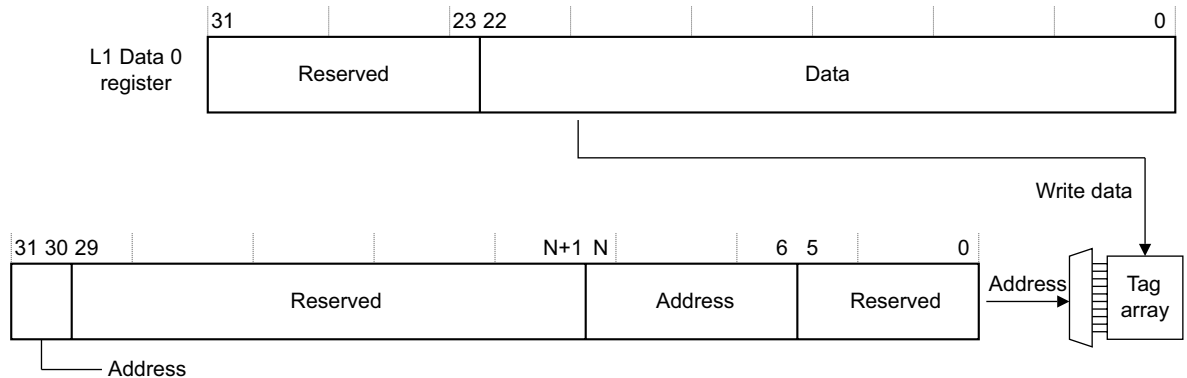
Figure 3-73 shows the bit arrangement of the L1 tag array read operation.



**Figure 3-73 L1 tag array read operation format**

Figure 3-74 on page 3-173 shows the bit arrangement of the L1 tag array write operation.





**Figure 3-74 L1 tag array write operation format**

To write one entry to the data side L1 tag array, for example:

```
LDR R0, =0x00500007;
MCR p15, 0, R0, c15, c0, 0;    Move R0 to D-L1 Data 0 Register
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c0, 6;    Write D-L1 Data 0 Register to D-tag
```

To read one entry from the data side L1 tag array, for example:

```
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c2, 6;    Read D-tag into data L1 Data 0 Register
MRC p15, 0, R2, c15, c0, 0;    Move D-L1 Data 0 Register to R2
```

To write one entry to the instruction side L1 tag array, for example:

```
LDR R0, =0x00500007;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c1, 6;    Write I-L1 Data 0 Register to I-tag
```

To read one entry from the instruction side L1 tag array, for example:

```
LDR R1, =0x800000C0;
MCR p15, 0, R1, c15, c3, 6;    Read I-tag into I-L1 Data 0 Register
MRC p15, 0, R2, c15, c1, 0;    Move I-L1 Data 0 Register to R2
```

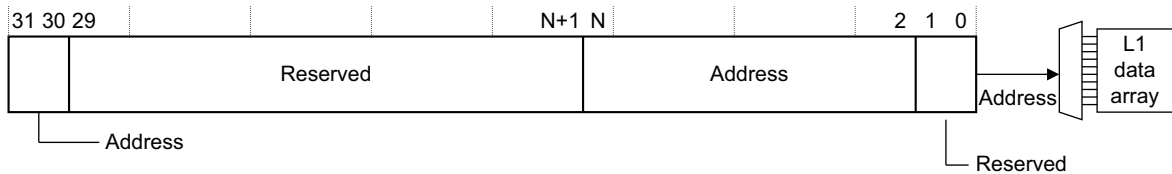
### 3.2.78 c15, L1 data array operations

The purpose of the L1 data array operations is to:

- read the L1 data array contents and write into the system debug data registers
- write into the system debug data registers and copy into the L1 data array.

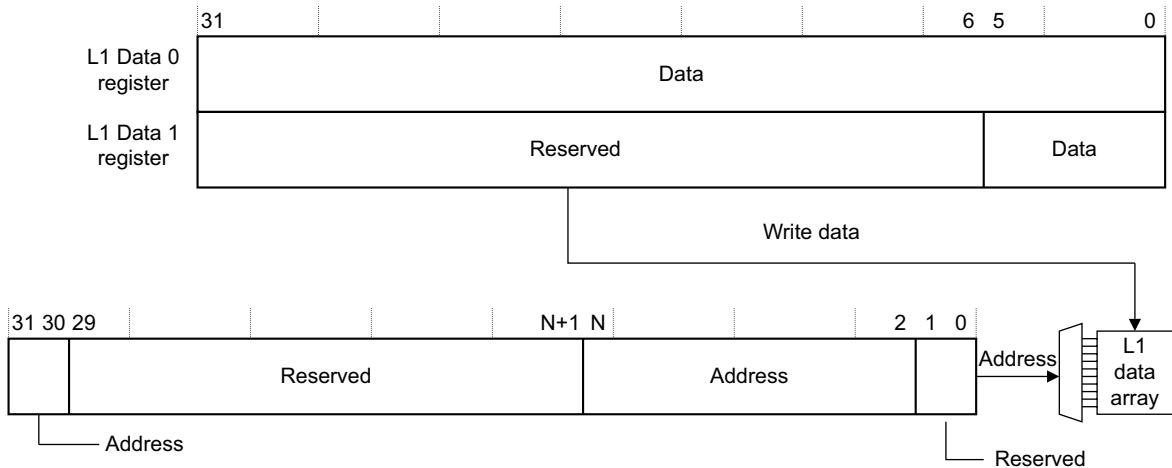
The L1 data array operation is accessible in secure privileged modes only. You can calculate the value of N in Figure 3-75 and Figure 3-76 using the NumSets and LineSize fields as defined in Table 3-42 on page 3-56.

Figure 3-75 shows the bit arrangement of the L1 data array read operation.



**Figure 3-75 L1 data array read operation format**

Figure 3-76 shows the bit arrangement of the L1 data array write operation.



**Figure 3-76 L1 data array write operation format**

To write one entry in data side L1 data array, for example:

```
LDR R0, =0x01234567;
MCR p15, 0, R0, c15, c0, 0;   Move R0 to D-L1 Data 0 Register
LDR R2, =0x1B;
MCR p15, 0, R2, c15, c0, 1;   Move R0 to D-L1 Data 1 Register
LDR R1, =0x800000D8;
MCR p15, 0, R1, c15, c0, 7;   Write D-L1 Data 0 or 1 Register to D-L1 data
```

To read one entry in data side L1 data array, for example:

```
LDR R1, =0x800000D8;
```

```

MCR p15, 0, R1, c15, c2, 7;    Read D-L1 data into data L1 Data 0 or 1 Register
MRC p15, 0, R0, c15, c0, 0;    Move D-L1 Data 0 Register to R0
MRC p15, 0, R2, c15, c0, 1;    Move D-L1 Data 1 Register to R2

```

To write one entry in instruction side L1 data array, for example:

```

LDR R0, =0x01234567;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R2, =0x1B;
MCR p15, 0, R2, c15, c1, 1;    Move R0 to I-L1 Data 1 Register
LDR R1, =0x800000D8;
MCR p15, 0, R1, c15, c1, 7;    Write I-L1 Data 0 or 1 Register to I-L1 data

```

To read one entry in instruction side L1 data array, for example:

```

LDR R1, =0x800000D8;
MCR p15, 0, R1, c15, c3, 7;    Read I-L1 data into I-L1 Data 0 or 1 Register
MRC p15, 0, R0, c15, c1, 0;    Move I-L1 Data 0 Register to R0
MRC p15, 0, R2, c15, c1, 1;    Move I-L1 Data 1 Register to R2

```

#### ———— Note ————

The granularity of the dirty bits is so that two sets of dirty bits are updated for each CP15 L1 DATA array write operation. A doubleword set of dirty bits are updated so that if word 0 or word 1 is being updated, then the D-bit and D-bit parity bits are updated for both word 0 and word 1.

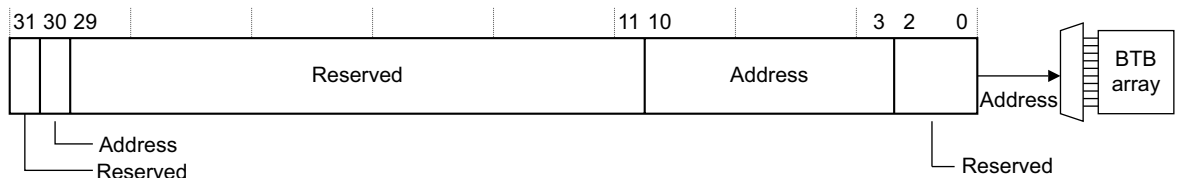
### 3.2.79 c15, BTB array operations

The purpose of the *Branch Target Buffer* (BTB) array operations is to:

- read the BTB array contents and write into the system debug data registers
- write into the system debug data registers and into the BTB array.

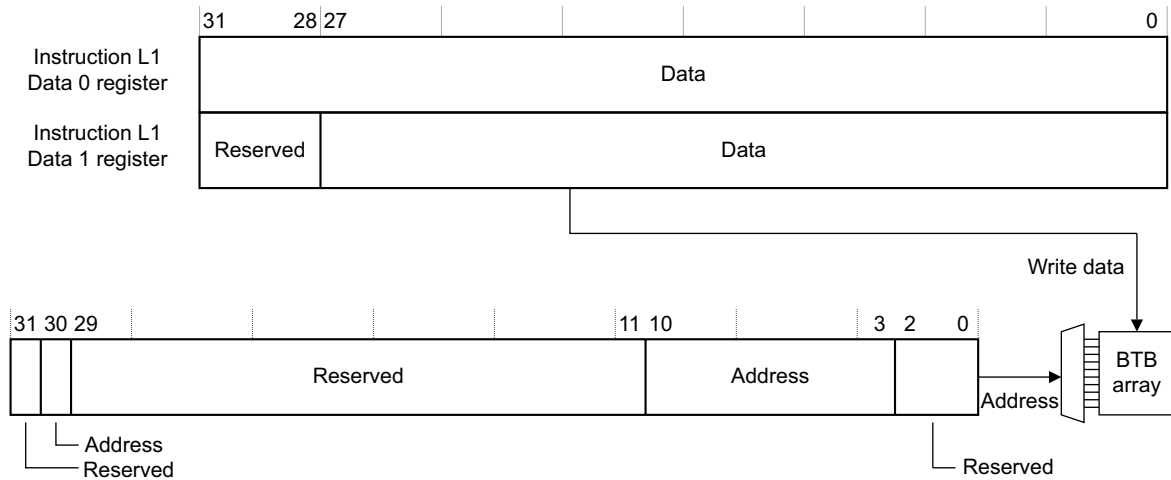
The BTB array operation is accessible in secure privileged modes only.

Figure 3-77 shows the bit arrangement of the BTB array read operation.



**Figure 3-77 BTB array read operation format**

Figure 3-78 on page 3-176 shows the bit arrangement of the BTB array write operation.



**Figure 3-78 BTB array write operation format**

To write one entry in the instruction side BTB array, for example:

```
LDR R0, =0x01234567;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R2, =0x0DDFFFFFF;
MCR p15, 0, R2, c15, c1, 1;    Move R0 to I-L1 Data 1 Register
LDR R1, =0x40000408;
MCR p15, 0, R1, c15, c5, 3;    Write I-L1 Data 0 or 1 Register to BTB
```

To read one entry in instruction side BTB array, for example:

```
LDR R1, =0x40000408;
MCR p15, 0, R1, c15, c7, 3;    Read BTB into I-L1 Data 0 or 1 Register
MRC p15, 0, R0, c15, c1, 0;    Move I-L1 Data 0 Register to R0
MRC p15, 0, R2, c15, c1, 1;    Move I-L1 Data 1 Register to R2
```

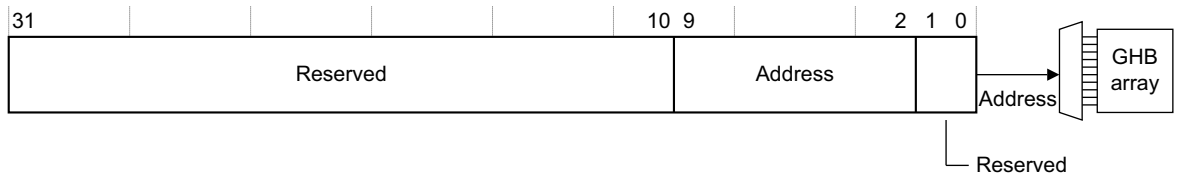
### 3.2.80 c15, GHB array operations

The purpose of the *Global History Buffer* (GHB) array operation is to:

- read the GHB array contents and write into the system debug data registers
- write into the system debug data registers and into the GHB array.

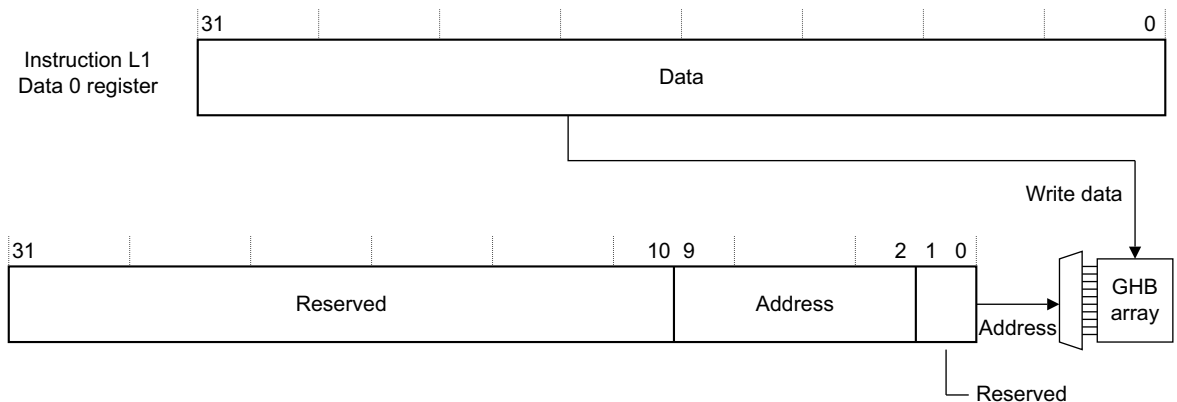
The GHB array operation is accessible in secure privileged modes only.

Figure 3-79 on page 3-177 shows the bit arrangement of the GHB array read operation.



**Figure 3-79 GHB array read operation format**

Figure 3-80 shows the bit arrangement of the GHB array write operation.



**Figure 3-80 GHB array write operation format**

To write one entry in the instruction side GHB array, for example:

```
LDR R0, =0x3333AAAA;
MCR p15, 0, R0, c15, c1, 0;    Move R0 to I-L1 Data 0 Register
LDR R1, =0x0000020C;
MCR p15, 0, R1, c15, c5, 2;    Write I-L1 Data 0 Register to GHB
```

To read one entry in the instruction side GHB array, for example:

```
LDR R1, =0x0000020C;
MCR p15, 0, R1, c15, c7, 2;    Read GHB into I-L1 Data 0 Register
MRC p15, 0, R0, c15, c1, 0;    Move I-L1 Data 0 Register to R0
```

### 3.2.81 c15, L2 system array debug data registers

The purpose of the L2 system array debug data registers is to hold the data:

- that is returned from the L2 tag, data, parity/ECC read operations
- for L2 tag, data, parity/ECC write operations.

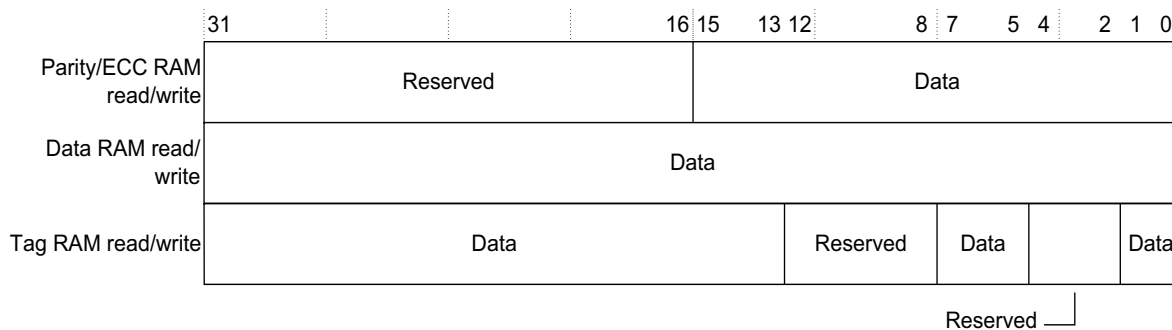
Because the L2 data arrays are greater than 32-bits wide, the processor contains three registers, Data 0, Data 1, and Data 2 registers, to hold data when retrieving or registering data as a result of read/write operations. If the data is greater than 32-bit wide, all of the registers are used to transfer data.

The Data 0, Data 1, and Data 2 read/write registers are accessible in secure privileged modes only.

To access the L2 system debug registers, read or write CP15 with:

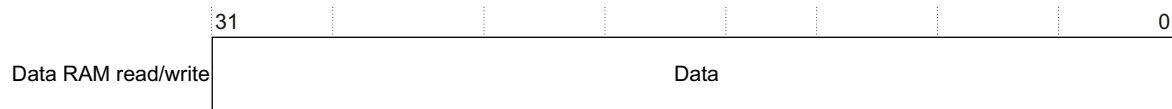
```
MCR p15, 0, <Rd>, c15, c8, 0 ; Write L2 Data 0 Register
MRC p15, 0, <Rd>, c15, c8, 0 ; Read L2 Data 0 Register
MCR p15, 0, <Rd>, c15, c8, 1 ; Write L2 Data 1 Register
MRC p15, 0, <Rd>, c15, c8, 1 ; Read L2 Data 1 Register
MCR p15, 0, <Rd>, c15, c8, 5 ; Write L2 Data 2 Register
MRC p15, 0, <Rd>, c15, c8, 5 ; Read L2 Data 2 Register
```

Figure 3-81 shows the bit arrangement of the L2 Data 0 Register when retrieving or registering data as a result of the read/write operations.



**Figure 3-81 L2 Data 0 Register format**

Figure 3-82 shows the bit arrangement of the L2 Data 1 Register when retrieving or registering data as a result of the read/write operations.



**Figure 3-82 L2 Data 1 Register format**

Figure 3-83 on page 3-179 shows the bit arrangement of the L2 Data 2 Register when retrieving or registering data as a result of the read/write operations.

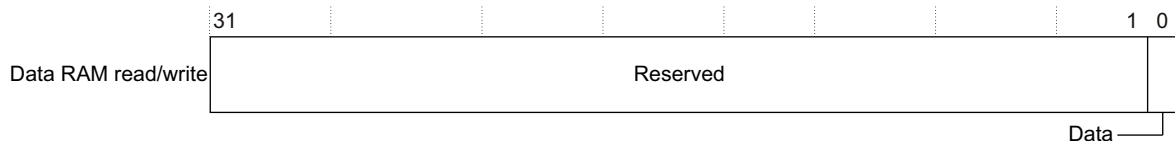
**Figure 3-83 L2 Data 2 Register format**

Table 3-158 shows how the bit values correspond with the L2 Data 0 Register functions as a result of an L2 parity/ECC read/write operation.

**Table 3-158 Functional bits of L2 Data 0 Register for an L2 parity/ECC operation**

Bits	Field	Function
[31:16]	-	Reserved. UNP, SBZ.
[15:0]	Data	Holds L2 parity/ECC information.

To perform an L2 parity/ECC operation, read or write CP15 with:

MCR p15, 0, c15, c8, 4 ; L2 parity and ECC write  
MCR p15, 0, c15, c9, 4 ; L2 parity and ECC read

Table 3-159 shows how the bit values correspond with the L2 Data 0 Register functions as a result of an L2 tag RAM read/write operation.

**Table 3-159 Functional bits of L2 Data 0 Register for a tag RAM operation**

Bits	Field	Function
[31:13]	Data	Holds bits [31:13] of the physical address tag read from or written to the L2 tag RAM.
[12:8]	-	Reserved. UNP, SBZ.
[7:5]	Data	Bit [7] holds the L2 tag RAM parity. Bits [6:5] hold the L2 tag RAM outer attributes.
[4:2]	-	Reserved. UNP, SBZ.
[1:0]	Data	Bit [1] holds the L2 tag RAM secure valid bit and bit [0] holds the nonsecure valid bit.

To perform an L2 tag RAM operation, read or write CP15 with:

MCR p15, 0, c15, c8, 2 ; L2 tag write  
MCR p15, 0, c15, c9, 2 ; L2 tag read

Table 3-160 shows how the bit values correspond with the L2 Data 0 Register functions as a result of an L2 data RAM read/write operation.

**Table 3-160 Functional bits of L2 Data 0 Register for a data RAM operation**

Bits	Field	Function
[31:0]	Data	Holds L2 data RAM information

Table 3-161 shows how the bit values correspond with the L2 Data 1 Register as a result of a data RAM read/write operation.

**Table 3-161 Functional bits of L2 Data 1 Register for a data RAM operation**

Bits	Field	Function
[31:0]	Data	Holds L2 data RAM information

Table 3-162 shows how the bit values correspond with the L2 Data 2 Register as a result of a data RAM read/write operation.

**Table 3-162 Functional bits of L2 Data 2 Register for a data RAM operation**

Bits	Field	Function
[31:1]	-	Reserved. UNP, SBZ.
[0]	Data	Holds a duplicate copy of the dirty bit that the L2 tag RAM stores.

To perform an L2 data 0, data 1, or data 2 operation, read or write CP15 with:

```
MCR p15, 0, c15, c8, 0 ; L2 data 0 write
MRC p15, 0, c15, c8, 0 ; L2 data 0 read
MCR p15, 0, c15, c8, 1 ; L2 data 1 write
MRC p15, 0, c15, c8, 1 ; L2 data 1 read
MCR p15, 0, c15, c8, 5 ; L2 data 2 write
MRC p15, 0, c15, c8, 5 ; L2 data 2 read
```

### 3.2.82 c15, L2 parity/ECC array operations

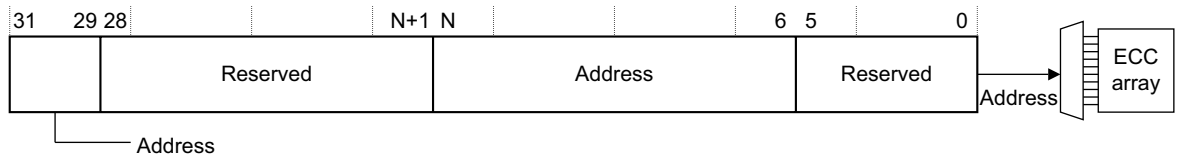
The purpose of the L2 parity/ECC array operations is to:

- read the L2 parity/ECC array contents and write into the system debug data registers
- write into the system debug data registers and copy into the L2 parity/ECC array.



The L2 parity/ECC array operation is accessible in secure privileged mode only. You can determine the value of N in Figure 3-84 and Figure 3-85 on page 3-182 from Table 3-163.

Figure 3-84 shows the bit arrangement of the L2 parity/ECC array read operation.



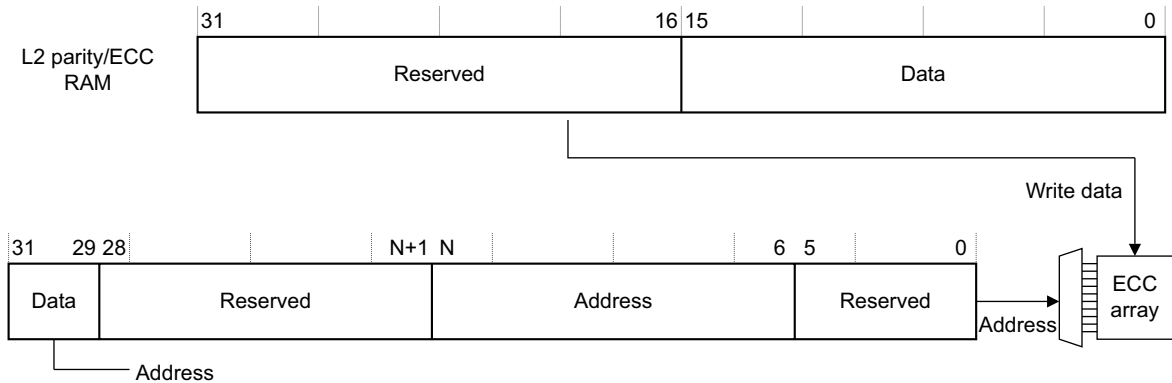
**Figure 3-84 L2 parity/ECC array read operation format**

The value of N in the Address field is:

**Table 3-163 Address field values**

L2 cache size	N
0KB	-
128KB	13
256KB	14
512KB	15
1024KB	16

Figure 3-85 on page 3-182 shows the bit arrangement of the L2 parity/ECC array write operation.



**Figure 3-85 L2 parity/ECC array write operation format**

To write one entry to the L2 parity/ECC array, for example:

```
LDR R0, =0x0000ABCD;
MCR p15, 0, R0, c15, c8, 0;    Move R0 to L2 Data 0 Register
LDR R1, =0x400000C0;
MCR p15, 0, R1, c15, c8, 4;    Write L2 Data 0 Register to L2 parity/ECC RAM
```

To read one entry from the L2 parity/ECC array, for example:

```
LDR R1, =0x400000C0 ;
MCR p15, 0, R1, c15, c9, 4;    Read L2 parity/ECC RAM into L2 Data 0 Register
MRC p15, 0, R2, c15, c8, 0;    Move L2 Data 0 Register to R2
```

### 3.2.83 c15, L2 tag array operations

The purpose of the L2 tag array operations is to:

- read the L2 tag array contents and write into the system debug data registers
- write into the system debug data registers and copy into the L2 tag array.

The L2 tag array operation is accessible in secure privileged modes only. You can determine the value of N in Figure 3-86 on page 3-183 and Figure 3-87 on page 3-183 from Table 3-163 on page 3-181.

Figure 3-86 on page 3-183 shows the bit arrangement of the L2 tag array read operation.

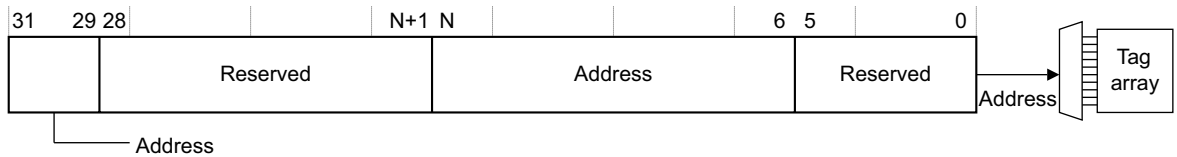


Figure 3-86 L2 tag array read operation format

Figure 3-87 shows the bit arrangement of the L2 tag array write operation.

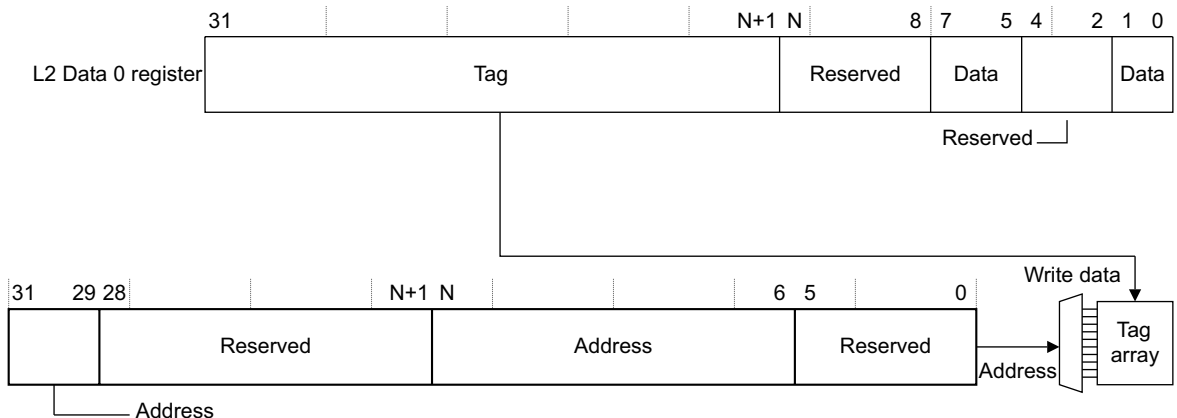


Figure 3-87 L2 tag array write operation format

To write one entry to the L2 tag array, for example:

```
LDR R0, =0x000020D1;
MCR p15, 0, R0, c15, c8, 0;    Move R0 to L2 Data 0 Register
LDR R1, =0x400000C0;
MCR p15, 0, R1, c15, c8, 2;    Write L2 Data 0 Register to L2 tag RAM
```

To read one entry from the L2 tag array, for example:

```
LDR R1, =0x400000C0;
MCR p15, 0, R1, c15, c9, 2;    Read L2 tag RAM into L2 Data 0 Register
MRC p15, 0, R2, c15, c8, 0;    Move L2 Data 0 Register to R2
```

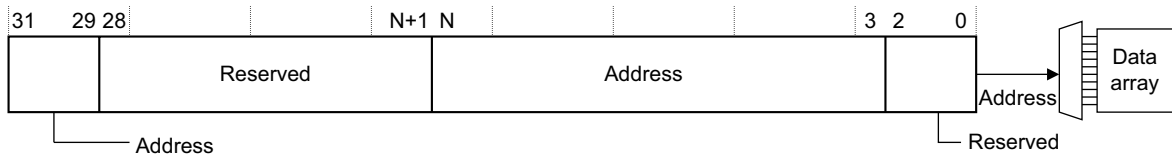
### 3.2.84 c15, L2 data array operations

The purpose of the L2 data array operations is to:

- read the L2 data array contents and write into the system debug data registers
- write into the system debug data registers and copy into the L2 data array.

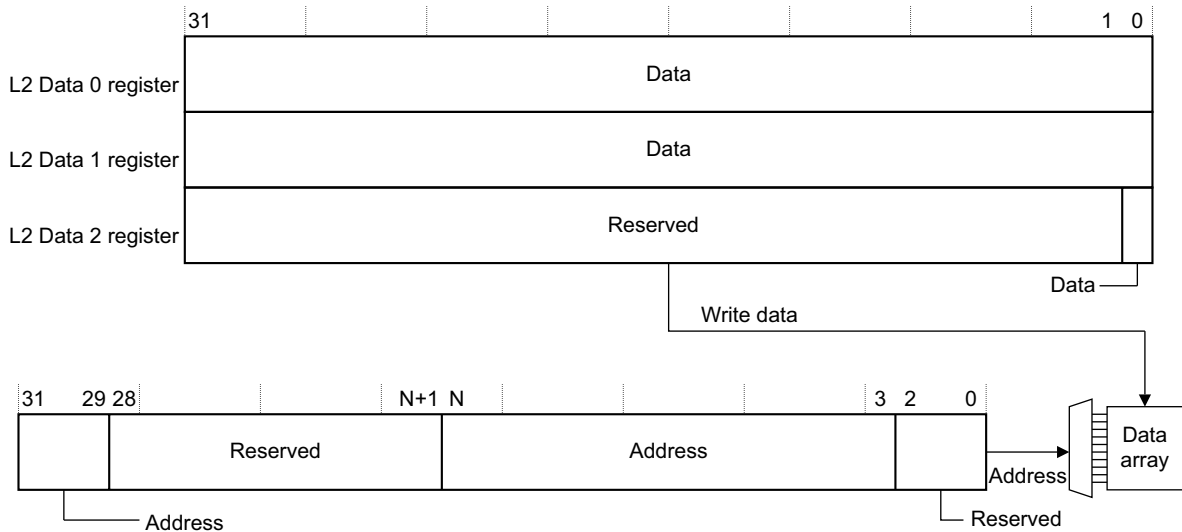
The L2 data array operation is accessible in secure privileged modes only. You can determine the value of N in Figure 3-88 and Figure 3-89 from Table 3-163 on page 3-181.

Figure 3-88 shows the bit arrangement of the L2 data RAM array read operation.



**Figure 3-88 L2 data RAM array read operation format**

Figure 3-89 shows the bit arrangement of the L2 data RAM array write operation.



**Figure 3-89 L2 data RAM array write operation format**

To write one entry to the L2 data RAM array, for example:

```
LDR R0, =0x01234567;
MCR p15, 0, R0, c15, c8, 0;    Move R0 to L2 Data 0 Register
LDR R0, =0x89ABCDEF;
MCR p15, 0, R0, c15, c8, 1;    Move R0 to L2 Data 1 Register
LDR R0, =0x00000001;
MCR p15, 0, R0, c15, c8, 5;    Move R0 to L2 Data 2 Register
LDR R1, =0x400000C8;
```

MCR p15, 0, R1, c15, c8, 3;      Write L2 Data 0-2 Registers to L2 data RAM

To read one entry from the L2 data RAM array, for example:

```
LDR R1, =0x400000C8;
MCR p15, 0, R1, c15, c9, 3;      Read L2 data RAM into L2 Data 0-2 Registers
MRC p15, 0, R2, c15, c8, 0;      Move L2 Data 0 Register to R2
MRC p15, 0, R3, c15, c8, 1;      Move L2 Data 1 Register to R3
MRC p15, 0, R4, c15, c8, 5;      Move L2 Data 2 Register to R4
```



# Chapter 4

## Unaligned Data and Mixed-endian Data Support

This chapter describes the unaligned and mixed-endianness data access support for the processor. It contains the following sections:

- *About unaligned and mixed-endian data* on page 4-2
- *Unaligned data access support* on page 4-3
- *Mixed-endian access support* on page 4-5.

## 4.1 About unaligned and mixed-endian data

The processor has the following features to support unaligned and mixed-endian data access:

- permanently enabled support for unaligned data access
- architecturally defined unaligned word, unaligned halfword, and word-aligned doubleword access
- byte-reverse instructions that operate on general-purpose register contents to support signed and unsigned halfword data values
- separate endianness control for data with instructions fixed as little-endian format, naturally aligned, but with legacy support for 32-bit word-invariant binary images and ROM
- EE bit in CP15 c1 Control Register 1 that controls load and store endianness during exceptions
- ARM and Thumb instructions to change the endianness and the E flag in the *Program Status Registers* (PSRs)
- byte-invariant addressing to support fine-grain big-endian and little-endian shared data structures, to conform to a shared memory standard.

———— **Note** —————

Instructions are always little-endian and must be aligned according to the size of the instruction:

- 32-bit ARM instructions must be word-aligned with address bits [1:0] equal to b00.
  - 16-bit or 32-bit Thumb instructions must be halfword-aligned with address bit [0] equal to 0.
-



## 4.2 Unaligned data access support

The processor supports loads and stores of unaligned words and halfwords. The processor makes the required number of memory accesses and transfers adjacent bytes transparently.

———— **Note** —————

Data accesses that cross a word boundary can add to the access time.

Setting the A bit in the CP15 c1 Control Register enables alignment checking. When the A bit is set to 1, two types of memory access generate a Data Abort signal and an Alignment fault status code:

- a 16-bit access that is not halfword-aligned
- a 32-bit load or store that is not word-aligned.

Alignment fault detection is a mandatory address-generation function rather than an optionally supported function of external memory management hardware.

See the *ARM Architecture Reference Manual* for more information on unaligned data access support.

### 4.2.1 NEON data alignment

This section describes NEON data access, alignment and the alignment qualifiers.

#### Alignment specifiers

In vector load and vector store operations, you can specify alignment requirements in the instruction.

Alignment faults are generated based on both memory attributes and alignment qualifiers.

When executing NEON vector accesses, the number of memory accesses (N) is determined based on the internal interface (128-bit) and the number of bytes being accessed. If no alignment qualifier is specified, the number of memory accesses is equal to  $N + 1$ . Adding alignment qualifiers improves performance by reducing extra cycles required to access memory.

## Normal memory

Normal memory conforms to the following rules concerning NEON alignment qualifiers:

- If an alignment qualifier is specified, a check is made for strict alignment based on the qualifier, independent of the A-bit setting. Table 4-1 shows the NEON alignment qualifiers:

**Table 4-1 NEON normal memory alignment qualifiers**

Alignment specifier	Low-order address bits
@16	Address[0] = b0
@32	Address[1:0] = b00
@64	Address[2:0] = b000
@128	Address[3:0] = b0000

- If an alignment qualifier is not specified, and A=1, the alignment fault is taken if it is not aligned to element size.
- If an alignment qualifier is not specified, and A=0, it is treated as *unaligned* access.

## Device memory and strongly ordered memory

Strongly ordered or device memory conforms to the following rules concerning NEON alignment qualifiers:

- if an alignment qualifier is specified, a check is made for strict alignment based on the qualifier, independent of the A-bit setting as Table 4-1 shows
- independent of the A bit, the alignment fault is taken if it is not aligned to element size
- access to the external memory is handled by creating a burst sequence of element sized transfers, up to the bus width.

## 4.3 Mixed-endian access support

In the processor, instruction endianness and data endianness are separated:

**Instructions** Instructions are fixed little-endian.

**Data** Data accesses can be either little-endian or big-endian as controlled by the E bit in the Program Status Register.

On any exception entry, including reset, the EE bit in the CP15 c1 Control Register determines the state of the E bit in the CPSR. See *c1, Control Register* on page 3-58 for details.

See the *ARM Architecture Reference Manual* for more information on mixed-endian access support.



# Chapter 5

## Program Flow Prediction

This chapter describes how the processor performs branch prediction. It contains the following sections:

- *About program flow prediction* on page 5-2
- *Predicted instructions* on page 5-3
- *Nonpredicted instructions* on page 5-6
- *Guidelines for optimal performance* on page 5-7
- *Enabling program flow prediction* on page 5-8
- *Operating system and predictor context* on page 5-9.

## 5.1 About program flow prediction

The processor contains program flow prediction hardware, also known as *branch prediction*. With program flow prediction disabled, all taken branches incur a 13-cycle penalty. With program flow prediction enabled, all mispredicted branches incur a 13-cycle penalty.

To avoid this penalty, the branch prediction hardware operates at the front of the instruction pipeline. The branch prediction hardware consists of:

- a 512-entry 2-way set associative *Branch Target Buffer* (BTB)
- a 4096-entry *Global History Buffer* (GHB)
- an 8-entry return stack.

An unpredicted branch executes in the same way as a branch that is predicted as not taken. Incorrect or invalid prediction of the branch prediction or target address causes the pipeline to flush, invalidating all of the following instructions.

## 5.2 Predicted instructions

This section shows the instructions that the processor predicts. Unless otherwise specified, the list applies to ARM, Thumb-2, and ThumbEE instructions. See the *ARM Architecture Reference Manual* for more information about instructions or addressing modes.

The flow prediction hardware predicts the following instructions:

- B conditional
- B unconditional
- BL
- BLX(1) immediate
 

The BL and BLX(1) instructions act as function calls and push the return address and ARM or Thumb state onto the return stack.
- BLX(2) register
 

The BLX(2) instruction acts as a function call and pushes the return address and ARM or Thumb state onto the return stack.
- BX
 

The BX r14 instruction acts as a function return and pops the return address and ARM or Thumb state from the return stack.
- LDM(1) with PC in the register list in ARM state
 

The LDM instruction with r13 specified as the base register acts as a function return and pops the return address and ARM or Thumb state from the return stack.
- POP with PC in register list in Thumb state
 

The POP instruction acts as a function return and pops the return address and ARM or Thumb state from the return stack.
- LDM with PC in register list in Thumb or ThumbEE state
 

The LDM instruction with r13 specified as the base register, or r9 specified as the base register in ThumbEE state acts as a function return and pops the return address and ARM or Thumb state from the return stack.
- LDR with PC destination
 

The LDR instruction with r13 specified as the base register, or r9 specified as the base register in ThumbEE state, acts as function return and pops the return address and ARM or Thumb state from the return stack.

- PC-destination data-processing operations in ARM state

In ARM state, the second operand of a data-processing instruction can be a 32-bit immediate value, an immediate shift value, or a register shift value. An instruction with an immediate shift value or a register shift value is predicted. An instruction with a 32-bit immediate value is not predicted. For example:

- `MOV pc, r10, LSL r3` is predicted
- `ADD pc, r0, r1, LSL #2` is predicted
- `ADD pc, r4, #4` is not predicted.

There is no restriction on the opcode predicted, but a majority of opcodes do not make sense for branch-type instructions. Usually only `MOV`, `ADD`, and `SUB` are useful.

———— **Note** ————

Instructions with the *S* suffix are not predicted. They are typically used to return from exceptions and have side effects that can change privilege mode and security state.

- `ADD(4)` with PC destination in Thumb state
- `MOV(3)` with PC destination in Thumb state
- `CPY` with PC destination in Thumb state
- `CZB` in Thumb state
- `TBB/TBH` in Thumb state

———— **Note** ————

In Thumb state, a branch that is normally encoded as unconditional can be conditioned by inclusion in an *If-Then-Else* (ITE) block. Then it is treated as a normal conditional branch.

- `HB` (ThumbEE state only)
- `HBP` (ThumbEE state only)
- `HBL` (ThumbEE state only)
- `HBLP` (ThumbEE state only).

The `HBL` and `HBLP` instructions act as function calls and push the return address onto the return stack.



## 5.2.1 Return stack predictions

The return stack stores the address and the ARM or Thumb state of the instruction after a function-call type branch instruction. This address is equal to the link register value stored in r14.

The following instructions cause a return stack push if predicted:

- BL immediate
- BLX(1) immediate
- BLX(2) register
- HBL (ThumbEE state)
- HBLP (ThumbEE state).

The following instructions cause a return stack pop if predicted:

- BX r14
- MOV pc, r14
- LDM r13, {...pc}
- LDR pc, [r13]
- LDM r9, {...pc} (ThumbEE state only)
- LDR pc, [r9] (ThumbEE state only).

The LDR instruction can use any of the addressing modes, as long as r13 is the base register. Additionally, in ThumbEE state you can also use r9 as a stack pointer so the LDR and LDM instructions with pc as a destination and r9 as a base register are also treated as a return stack pop.

Because return-from-exception instructions can change processor privilege mode and security state, they are not predicted. This includes the LDM(3) instruction, and the MOVS pc, r14 instruction.

## 5.3 Nonpredicted instructions

The following instructions are not predicted:

- Instructions that can be used to return from an exception  
These instructions change the PC. They potentially change processor state, privilege mode, and security state. To fetch the target instructions in the new privilege mode, the pipeline must be flushed.
- Instructions that restore the CPSR from memory or from the SPSR  
These instructions change the PC. They potentially change processor state, privilege mode, and security state. To fetch the target instructions in the new privilege mode, the processor must flush the pipeline.
- PC-destination data-processing instructions with immediate values in ARM state.  
As described, PC-destination data-processing instructions where the second operand is an immediate, are not predicted
- BXJ  
The processor implements the trivial Jazelle extension, so BXJ becomes BX. This can be used as an unpredictable indirect branch instruction to force a pipeline flush on execution.
- ENTERX/LEAVEX  
Transitions between Thumb and ThumbEE state are not predicted.

## 5.4 Guidelines for optimal performance

You can avoid certain code constructs to maximize branch prediction performance. For example:

- Using conditional Undefined instructions in normal code to enter the undefined handler as a means of doing emulation.
- Coding more than two likely taken branches per fetch. This can only happen in Thumb state. Unless used as a jump table where each branch is its own basic block, use NOPs for padding.
- Coding more than three branches per fetch that are likely to be executed in sequence.

In Thumb state, it is possible to pack four branches in a single fetch, for example, in a multiway branch:

```
BVS overflow  
BGT greater_than  
BLT less_than  
B equal
```

This is a sequence of more than three branches with three conditional branches, and the fourth branch is likely to be reached. Avoid this kind of sequence, or use NOPs to break up the branch sequence.

## 5.5 Enabling program flow prediction

You can enable program flow prediction by setting the Z bit in the CP15 c1 Control Register to 1. See *c1, Control Register* on page 3-58 for details. Reset disables program flow prediction, invalidates the BTB, and resets the GHB to a known state. No software intervention is required to prepare the prediction logic before enabling program flow prediction.

## 5.6 Operating system and predictor context

The BTB does not have to be invalidated on a context switch, self-modifying code, or any other change in the VA-to-PA mapping.

ARMv7-A specifies two branch prediction invalidation operations:

- MCR p15, 0, Rx, c7, c5, 6 ; invalidate entire branch predictor array
- MCR p15, 0, Rx, c7, c5, 7 ; invalidate VA from branch predictor array

These operations are not required to perform a context switch in the processor and are implemented as NOPs. ARMv7-A generic context-switching or self-modifying code can contain these operations without cycle penalty. These instructions can be enabled by setting the IBE bit in the Auxiliary Control Register to 1. See *c1, Auxiliary Control Register* on page 3-61 for details.

### 5.6.1 Instruction memory barriers

ARMv7-A requires *Instruction Memory Barriers* (IMBs) after updates to certain CP15 registers or CP15 operations. The processor flushes the pipeline to ensure that the instructions following the given CP15 instruction are fetched in the new context. In addition, self-modifying code sequences must be preceded by an IMB. The recommended means of implementing an IMB is the ISB instruction.

The following prefetch flush instruction is from earlier versions of the ARM architecture. The processor supports this instruction, but its use is deprecated in ARMv7-A.

MCR p15, 0, Rx, c7, c5, 4



# Chapter 6

## Memory Management Unit

This chapter describes the *Memory Management Unit (MMU)*. It contains the following sections:

- *About the MMU* on page 6-2
- *Memory access sequence* on page 6-3
- *16MB supersection support* on page 6-4
- *MMU interaction with memory system* on page 6-5
- *External aborts* on page 6-6
- *TLB lockdown* on page 6-7
- *MMU software-accessible registers* on page 6-8.

## 6.1 About the MMU

The MMU works with the L1 and L2 memory system to translate virtual addresses to physical addresses. It also controls accesses to and from external memory. See the *ARM Architecture Reference Manual* for a full architectural description of the MMU.

The processor implements the ARMv7-A MMU enhanced with Security Extensions features to provide address translation and access permission checks. The MMU controls table walk hardware that accesses translation tables in main memory. The MMU enables fine-grained memory system control through a set of virtual-to-physical address mappings and memory attributes held in instruction and data TLBs.

The MMU features include the following:

- full support for *Virtual Memory System Architecture version 7* (VMSAv7)
- separate, fully-associative, 32-entry data and instruction TLBs
- support for 32 lockable entries using the lock-by-entry model
- TLB entries that support 4KB, 64KB, 1MB, and 16MB pages
- 16 domains
- global and application-specific identifiers to prevent context switch TLB flushes
- extended permissions check capability
- round-robin replacement policy
- CP15 TLB preloading instructions to enable locking of TLB entries.



## 6.2 Memory access sequence

When the processor generates a memory access, the MMU:

1. Performs a lookup for the requested virtual address and current ASID and security state in the relevant instruction or data TLB.
2. Performs a hardware translation table walk if the lookup in step 1 misses.

The MMU might not find global mapping, mapping for the currently selected ASID, or a matching NSTID for the virtual address in the TLB. The hardware does a translation table walk if the translation table walk is enabled by the PD0 or PD1 bit in the TTB Control Register. If translation table walks are disabled, the processor returns a Section Translation fault.

If the MMU finds a matching TLB entry, it uses the information in the entry as follows:

1. The access permission bits and the domain determine if the access is enabled. If the matching entry does not pass the permission checks, the MMU signals a memory abort. See the *ARM Architecture Reference Manual* for a description of abort types and priorities, and for a description of the *Instruction Fault Status Register (IFSR)* and *Data Fault Status Register (DFSR)*.
2. The memory region attributes specified in the CP15 c10 registers control the cache and write buffer, and determine if the access is secure or nonsecure, cached or noncached, and device or shared.
3. The MMU translates the virtual address to a physical address for the memory access.

If the MMU does not find a matching entry, a hardware table walk occurs.

### 6.2.1 TLB match process

Each TLB entry contains a virtual address, a page size, a physical address, and a set of memory attributes.

A TLB entry matches when these conditions are true:

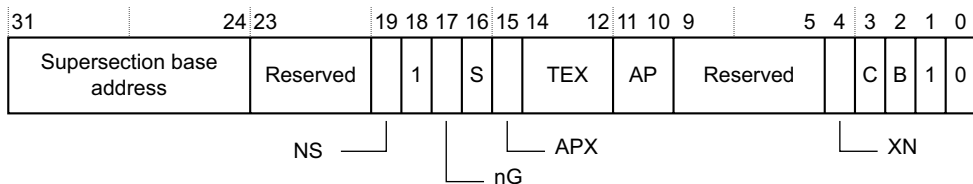
- its virtual address matches that of the requested address
- its NSTID matches the secure or nonsecure state of the MMU request
- its ASID matches the current ASID or is global.

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 only one TLB entry matches at any time. Entries with different NSTIDs can never be hit simultaneously.

## 6.3 16MB supersection support

The processor supports supersections that consist of 16MB blocks of memory. The processor does not support the optional extension of physical address bits [39:32].

Figure 6-1 shows the descriptor format for supersections.



**Figure 6-1 16MB supersection descriptor format**

———— **Note** ————

Each translation table entry for a supersection must be repeated 16 times in consecutive memory locations in the level 1 translation tables, and each of the 16 repeated entries must have identical translation and permission information. See the *ARM Architecture Reference Manual* for more information.

## 6.4 MMU interaction with memory system

You can enable or disable the MMU as described in the *ARM Architecture Reference Manual*.

After a CP15 c1 instruction enables the MMU, the processor flushes all following instructions in the pipeline. The processor then begins refetching instructions, and the MMU performs virtual-to-physical address mapping according to the translation table descriptors in main memory.

After a CP15 c1 instruction disables the MMU, the processor flushes all following instructions in the pipeline. The processor then begins refetching instructions and uses flat address mapping. In flat address mapping, PA = VA.

The following is an example of enabling the MMU:

```
MRC p15, 0, r1, c1, c0, 0    ; read CP15 Register 1
ORR r1, r1, #0x1
MCR p15, 0, r1, c1, c0, 0    ; enable MMUs
Fetch translated
Fetch translated
Fetch translated
Fetch translated
```

The following is an example of disabling the MMU:

```
MRC p15, 0, r1, c1, c0, 0    ; read CP15 Register 1
BIC r1, r1, #0x1
MCR p15, 0, r1, c1, c0, 0    ; disabled
Fetch flat
Fetch flat
Fetch flat
Fetch flat
```

## 6.5 External aborts

External memory errors are defined as those that occur in the memory system but are not detected by the MMU. External memory errors are expected to be extremely rare and are likely to be fatal to the running processor. External aborts are caused by errors flagged by AXI when the request goes external to the processor. External aborts can be configured to trap to the monitor by setting the EA bit in the Secure Configuration Register to 1. See *c1, Secure Configuration Register* on page 3-70 for more information.

### 6.5.1 External aborts on data read or write

Externally generated errors during a data read or write can be imprecise. This means that the r14\_abt on entry into the abort handler on such an abort might not hold the address of the instruction that caused the exception.

The DFAR is Unpredictable when an imprecise abort occurs.

In the case of a load multiple or store multiple operation, the address captured in the DFAR is that of the address that generated the precise external abort.

### 6.5.2 Precise and imprecise aborts

Chapter 3 *System Control Coprocessor* describes precise and imprecise aborts, their priorities, and the IFSR and DFSR. To determine a fault type, read the DFSR for a data abort or the IFSR for an instruction abort.

The processor supports an Auxiliary Fault Status Register for software compatibility reasons only. The processor does not modify this register because of any generated abort.

## 6.6 TLB lockdown

The TLB supports the TLB lock-by-entry model as described in the *ARM Architecture Reference Manual*. CP15 preload TLB instructions support loading entries into the TLB to be locked. Any preload operation first looks in the TLB to determine if the entry hits within the TLB array. If the entry misses, a hardware translation table walk loads that entry into the TLB array. See *c10, TLB Lockdown Registers* on page 3-128 and *c10, TLB preload operation* on page 3-130 for more information.

## 6.7 MMU software-accessible registers

Table 6-1 shows the CP15 registers that control the MMU. See Chapter 3 *System Control Coprocessor* for more information on CP15.

**Table 6-1 CP15 register functions**

<b>Register</b>	<b>Cross reference</b>
TLB Type Register	<i>c0, TLB Type Register on page 3-28</i>
Control Register	<i>c1, Control Register on page 3-58</i>
Nonsecure Access Control Register	<i>c1, Nonsecure Access Control Register on page 3-73</i>
Translation Table Base Register 0	<i>c2, Translation Table Base Register 0 on page 3-75</i>
Translation Table Base Register 1	<i>c2, Translation Table Base Register 1 on page 3-77</i>
Translation Table Base Control Register	<i>c2, Translation Table Base Control Register on page 3-79</i>
Domain Access Control Register	<i>c3, Domain Access Control Register on page 3-81</i>
Data Fault Status Register (DFSR)	<i>c5, Data Fault Status Register on page 3-83</i>
Instruction Fault Status Register (IFSR)	<i>c5, Instruction Fault Status Register on page 3-85</i>
Data Fault Address Register (DFAR)	<i>c6, Data Fault Address Register on page 3-87</i>
Instruction Fault Address Register (IFAR)	<i>c6, Instruction Fault Address Register on page 3-88</i>
TLB operations	<i>c8, TLB operations on page 3-99</i>
TLB Lockdown Registers	<i>c10, TLB Lockdown Registers on page 3-128</i>
Primary Region Remap Register	<i>c10, Memory Region Remap Registers on page 3-131</i>
Normal Memory Remap Register	<i>c10, Memory Region Remap Registers on page 3-131</i>
FCSE PID Register	<i>c13, FCSE PID Register on page 3-157</i>
Context ID Register	<i>c13, Context ID Register on page 3-160</i>

# Chapter 7

## Level 1 Memory System

This chapter describes the L1 memory system. It contains the following sections:

- *About the L1 memory system* on page 7-2
- *Cache organization* on page 7-3
- *Memory attributes* on page 7-6
- *Cache debug* on page 7-9
- *Data cache features* on page 7-10
- *Instruction cache features* on page 7-11
- *Hardware support for virtual aliasing conditions* on page 7-13
- *Parity detection* on page 7-14.

## 7.1 About the L1 memory system

The L1 memory system consists of separate instruction and data caches in a Harvard arrangement. The L1 memory system provides the core with:

- fixed line length of 64 bytes
- support for 16KB or 32KB caches
- two 32-entry fully associative ARMv7-A MMU
- data array with parity for error detection
- an instruction cache that is virtually indexed, IVIPT
- a data cache that is physically indexed, PIPT
- 4-way set associative cache structure
- random replacement policy
- nonblocking cache behavior for Advanced SIMD code
- blocking for integer code
- MBIST
- support for hardware reset of the L1 data cache valid RAM, see *Hardware RAM array reset* on page 10-8.



## 7.2 Cache organization

Each cache is 4-way set associative of configurable size. They are physically tagged, and virtually indexed for instruction and physically indexed for data. The cache sizes are configurable with sizes of 16KB or 32KB. Both the instruction cache and the data cache are capable of providing two words per cycle for all requesting sources. Data cache can provide four words per cycle for NEON or VFP memory accesses.

The system control coprocessor, CP15, handles the control of the L1 memory system and the associated functionality, together with other system wide control attributes. See Chapter 3 *System Control Coprocessor* for more information on CP15 registers.

### 7.2.1 Cache control operations

The cache control operations that are supported by the processor are described in Chapter 3 *System Control Coprocessor*.

### 7.2.2 Cache miss handling

A cache miss results when a read access is not present in the cache. The caches perform critical word-first cache refilling.

### 7.2.3 Cache disabled behavior

If you disable the cache then the cache is not accessed for reads or writes. This ensures that you can achieve maximum power savings. It is therefore important that before you disable the cache, 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 completely cleaned and invalidated before being disabled. Unlike normal reads and writes to the cache, cache maintenance operations are performed even if the cache is disabled.

### 7.2.4 Unexpected hit behavior

An unexpected hit is where the cache reports a hit on a memory location that is marked as noncacheable or shared. The unexpected hit is ignored.

For writes, an unexpected cache hit does not result in the cache being updated.

### 7.2.5 Cache parity error detection

The purpose of cache parity error detection is to increase the tolerance to memory faults.

### Instruction cache data RAM parity error detection

The instruction cache RAM is written on cache linefills. Parity error detection is done on a fetch-wide basis, that is, a parity error on any byte in a 64-bit fetch region causes a parity error on the first instruction within that fetch. The detection of a parity error instruction cache RAM causes the processor to return a Prefetch Abort.

When the processor executes the instruction:

- the address of the fetch containing the parity error is stored in the Instruction Fault Address Register
- the Instruction Fault Status Register is set to indicate the presence of a parity error.

### Data cache data RAM parity error detection

The detection of a parity error in the data cache RAM causes the processor to return a Data Abort. The Data Fault Status Register is set to indicate the presence of a parity error. The parity error is always imprecise on the data cache.

## 7.2.6 Instruction cache maintenance

The Cortex-A8 processor is implemented with an optional extension, the *IVIPT extension* (Instruction cache Virtually Indexed Physically Tagged extension). The effect of this extension is to reduce the instruction cache maintenance requirement to a single condition:

- writing new data to an instruction address.

#### ————— **Note** —————

This condition is consistent with the maintenance required for a Virtually Indexed Physically Tagged (VIPT) instruction cache.

Software can read the Cache Type Register to determine whether the IVIPT extension is implemented, see *c0*, *Cache Type Register* on page 3-26.

Software written to rely on a VIPT instruction cache must only be used with processors that implement the IVIPT. For maximum compatibility across processors, ARM recommends that operating systems target the ARMv7 base architecture that uses ASID-tagged VIVT instruction caches, and do not assume the presence of the IVIPT extension. Software that relies on the IVIPT extension might fail in an unpredictable way on an ARMv7 implementation that does not include the IVIPT extension.

With an instruction cache, the distinction between a VIPT cache and a PIPT cache is much less visible to the programmer than it is for a data cache, because normally the contents of an instruction cache are not changed by writing to the cached memory. However, there are situations where a program must distinguish between the different cache tagging strategies. Example 7-1 describes such a situation.

**Example 7-1 A situation where software must be aware of the  
Instruction cache tagging strategy**

---

Two processes, P<sub>1</sub> and P<sub>2</sub>, share some code and have separate virtual mappings to the same region of instruction memory. P<sub>1</sub> changes this region, for example as a result of a JIT, or some other self-modifying code operation. P<sub>2</sub> needs to see the modified code.

As part of its self-modifying code operation, P<sub>1</sub> must invalidate the changed locations from the instruction cache. If this invalidation is performed by MVA, and the instruction cache is a VIPT cache, then P<sub>2</sub> might continue to see the old code. For more information, see the *ARM Architecture Reference Manual*.

In this situation, if the instruction cache is a VIPT cache, after the code modification the entire instruction cache must be invalidated to ensure P<sub>2</sub> observes the new version of the code.

---

## 7.3 Memory attributes

ARMv7-A defines three memory regions using the TEX, C and B bits. They are:

- *Strongly ordered*
- *Device*
- *Normal.*

An access can be marked as shared. If it is marked as shared, the access is treated as noncacheable.

### 7.3.1 Strongly ordered

Strongly ordered memory type is noncacheable, nonbufferable, and serialized. This type of memory flushes all buffers and waits for acknowledge from the bus before executing the next instruction. You must execute strongly ordered memory nonspeculatively. Unaligned accesses to strongly ordered memory result in an alignment fault.

### 7.3.2 Device

Device memory type is noncacheable and must be executed nonspeculatively. Ordering requirement for device accesses are as follows:

- device loads cannot bypass device stores
- device stores can be buffered, but must be executed with respect to other device stores
- unaligned accesses to strongly ordered memory result in an alignment fault.

### 7.3.3 Normal

Normal memory type can have the following attributes:

- noncacheable, bufferable
- write-back cached, write-allocate
- write-through cached, no allocate on write, buffered
- write-back cached, no allocate on write, buffered.

Table 7-1 on page 7-7 shows how L1 and L2 memory systems handle these memory types.

Table 7-1 Memory types affecting L1 and L2 cache flows

L1 inner policy <sup>a</sup>	L2 outer policy	Buffers flushed	Description
Strongly ordered	Strongly ordered	Yes	Noncacheable and nonbufferable.
Device shared	Device shared	No	Device accesses are noncacheable and must be executed nonspeculative.
Device nonshared	Device nonshared	No	Device accesses are noncacheable and must be executed nonspeculative.
Noncacheable, nonbufferable	Cacheable, write-back, no write-allocate	No	Loads and stores are not cached at L1. Loads are not filled into the fill buffer but are allocated to the L2 cache. Stores bypass integer store buffer and are directly sent to L2.
Noncacheable, nonbufferable	Cacheable, write-back, write-allocate	No	Loads and stores are not cached at L1. Loads are not filled into the fill buffer but are allocated to the L2 cache. Stores bypass integer store buffer and are directly sent to L2. L2 store misses allocate the line into L2 cache.
Noncacheable, nonbufferable	Cacheable, write-through, no write-allocate	No	Loads and stores are not cached at L1. Loads are not filled into the fill buffer. Stores bypass integer store buffer and are directly sent to L2. L2 store misses do not allocate the line into L2 cache. Store hits are sent externally in addition to updating L2.
Cacheable, write-back, no write-allocate	Noncacheable, nonbufferable	No	Load misses are filled into L1. Store misses are sent to L2. L2 does not allocate the line into L2 cache on an L2 miss but is sent externally.
Cacheable, write-back, write-allocate	Noncacheable, nonbufferable	No	This is not supported. L1 is always in the no write-allocate mode.
Cacheable, write-through, no write-allocate	Noncacheable, nonbufferable	No	Load misses are filled into L1. Store hits and store misses are sent to L2. L2 does not allocate the line into L2 cache on an L2 miss because it is sent externally.
Cacheable, write-back, no write-allocate	Cacheable, write-back, no write-allocate	No	Load misses are allocated into L1. Store misses bypass integer store buffer and are sent to L2. Store hits update the cache. L2 does not allocate the line into L2 cache on store misses.
Cacheable, write-back, no write-allocate	Cacheable, write-back, write-allocate	No	Load misses are allocated into L1. Store misses bypass integer store buffer and are sent to L2. Store hits update the cache. L2 allocates the line into L2 cache for store misses.

Table 7-1 Memory types affecting L1 and L2 cache flows (continued)

L1 inner policy <sup>a</sup>	L2 outer policy	Buffers flushed	Description
Cacheable, write-back, no write-allocate	Cacheable, write-through, no write-allocate	No	Load misses are allocated into L1. L2 makes store hits write-through at L2 cache and does not allocate the line into L2 for store misses.
Cacheable, write-through, no write-allocate	Cacheable, write-back, no write-allocate	No	Loads are allocated into L1. Store hits are made write-through at L1. Store hits update the cache and are sent to L2. L2 does not allocate store misses but they are sent externally.
Cacheable, write-through, no write-allocate	Cacheable, write-back, write-allocate	No	Loads are allocated into L1. Store hits are made write-through at L1. Store hits update the cache and are sent to L2. Store misses are allocated into L2.
Cacheable, write-through, no write-allocate	Cacheable, write-through, no write-allocate	No	Loads are allocated into L1. Store hits are made write-through at L1. Store hits update the cache and are sent to L2. Store misses are not allocated into L2.
Noncacheable, bufferable	Noncacheable, bufferable	No	Loads are replayed and access is sent externally. Stores bypass integer store buffer and are placed into L2 write buffer. Stores are sent externally.

a. You can configure the L2 cache to use the inner policy attributes.

## 7.4 Cache debug

See Chapter 12 *Debug* for information on cache debug.

## 7.5 Data cache features

This section describes the unique features of the data cache. It contains the following:

- *Data cache preload instruction*
- *Data cache behavior with C-bit disabled.*

### 7.5.1 Data cache preload instruction

ARMv7-A specifies the PLD instruction as a preload hint instruction. The processor uses the PLD instruction to preload cache lines to the L2 cache. If the PLD instruction results in a L1 cache hit, L2 cache hit, or TLB miss no more action is taken. If a cache miss and TLB hit result, the line is retrieved from external memory and is loaded into the L2 memory cache.

### 7.5.2 Data cache behavior with C-bit disabled

The C bit in CP15 Control Register c1 enables or disables the L1 data cache. See *c1, Control Register* on page 3-58 for more information on caching data when enabling the data cache. If the C bit is disabled, then memory requests do not access any of the data cache arrays.

An exception to this rule is the CP15 data cache operations. If the data cache is disabled, all data cache maintenance operations can still execute normally.



## 7.6 Instruction cache features

This section describes the unique features of the instruction cache. It contains the following:

- *Instruction cache preload instruction*
- *Instruction cache speculative memory accesses*
- *Instruction cache disabled behavior.*

### 7.6.1 Instruction cache preload instruction

ARMv7-A specifies the PLI instruction as a preload hint instruction. Because the processor implements a blocking L1 cache and to avoid the penalty associated with a cache miss, the processor handles the PLI instruction as a NOP.

### 7.6.2 Instruction cache speculative memory accesses

An instruction can remain in the pipeline between being fetched and being executed. Because there can be several unresolved branches in the pipeline, instruction fetches are speculative, meaning there is no guarantee that they are executed. A branch or exceptional instruction in the code stream can cause a pipeline flush, discarding the currently fetched instructions.

Fetches or instruction table walks that begin without an empty pipeline are marked speculative. If the pipeline contains any instruction up to the point of branch and exception resolution, then the pipeline is considered not empty. If a fetch is marked speculative and misses the L1 instruction cache and the L2 cache, it is not forwarded to the external interface. Fetching is suspended until all outstanding instructions are resolved or the pipeline is flushed.

This behavior is controlled by the ASA bit in the CP15 Auxiliary Control Register c1. See *c1, Auxiliary Control Register* on page 3-61 for information on the ASA bit. By default, this bit is 0, indicating that speculative fetches or instruction table walks are not forwarded to the external interface. If this bit is set to 1, then neither fetches nor instruction table walks are marked speculative, and are forwarded to the external interface.

Given the aggressive prefetching behavior, you must not place read-sensitive devices in the same page as code. Pages containing read-sensitive devices must be marked with the TLB XN (execute never) attribute bit.

### 7.6.3 Instruction cache disabled behavior

The I bit in CP15 Control Register c1 enables or disables the L1 instruction cache. If the I bit is disabled, then fetches do not access any of the instruction cache arrays.

An exception to this rule is the CP15 instruction cache operations. If the instruction cache is disabled, the instruction cache maintenance operations can still execute normally.

## 7.7 Hardware support for virtual aliasing conditions

Previously, restrictions were placed on software to ensure that no virtual aliasing conditions arise. This restriction, referred to as page coloring, is no longer required.

## 7.8 Parity detection

The L1 memory system instruction and data caches support parity detection on data arrays. There is one parity bit for each data byte. For data cache, because the dirty bit is also held in the data array, there is a corresponding parity bit to cover the dirty bit. Parity errors reported by instruction cache accesses result in precise prefetch aborts. Parity errors reported by data cache accesses result in imprecise data aborts.

Parity errors reported by instruction cache accesses are reported on a fetch-granularity basis, that is, if any byte within a fetch region contains a parity error, the parity error is reported on the first instruction in the fetch, although this instruction might not contain the parity error.

The Auxiliary Control Register bit [3], L1PE, controls parity errors reported by the L1 caches. Parity errors are enabled if the L1PE bit is set to 1.

If a cache access result is a parity error in the L1 data cache, then the L1 data cache and the L2 cache are unpredictable. No recovery is possible. The abort handler must:

- disable the caches
- communicate the fail directly with the external system
- request a reboot.

# Chapter 8

## Level 2 Memory System

This chapter describes the L2 memory system. It contains the following sections:

- *About the L2 memory system* on page 8-2
- *Cache organization* on page 8-3
- *Enabling and disabling the L2 cache controller* on page 8-5
- *L2 PLE* on page 8-6
- *Synchronization primitives* on page 8-11
- *Locked access* on page 8-13
- *Parity and error correction code* on page 8-14.

## 8.1 About the L2 memory system

The processor contains an on-chip L2 memory system that consists of the following components:

- L2 *PreLoad Engine* (PLE)
- AXI interface
- configurable L2 RAM.

The L2 memory system is tightly coupled to the L1 data cache and L1 instruction cache. The L2 memory system does not support hardware cache coherency, therefore software intervention is required to maintain coherency in the system.

The key features of the L2 memory system include:

- configurable cache size of 0KB, 128KB, 256KB, 512KB, and 1MB
- fixed line length of 64 bytes
- physically indexed and tagged
- 8-way set associative cache structure
- support for lockdown format C
- configurable 64-bit or 128-bit wide AXI system bus interface with support for multiple outstanding requests
- random replacement policy
- optional ECC or parity protection on the data RAM
- optional parity protection on the tag RAM
- MBIST
- support for hardware reset of the L2 unified cache valid RAM, see *Hardware RAM array reset* on page 10-8.

## 8.2 Cache organization

The L2 cache is 8-way set associative of configurable size. The cache is physically addressed. The cache sizes are configurable with sizes in the range of 0KB, 128KB, 256KB, 512KB, and 1MB.

You can reduce the effective cache size using lockdown format C. This feature enables you to lock cache ways to prevent allocation to locked entries.

You can configure the L2 memory pipeline to insert wait states to take into account the latencies of the compiled memories for the implemented RAMs.

To enable streaming of NEON read accesses from the L1 data cache, the L2 memory system supports up to twelve NEON read accesses. The write buffer handles integer writes, NEON writes, and eviction accesses from the L1 data cache. This enables streaming of write requests from the L1 data cache.

The L2 cache incorporates a dirty bit per quadword to reduce AXI traffic. This eliminates unnecessary transfer of clean data on the AXI interface.

### 8.2.1 L2 cache bank structure

The L2 cache is partitioned into multiple banks to enable parallel operations. There are two levels of banking:

- the tag array is partitioned into multiple banks to enable up to two requests to access different tag banks of the L2 cache simultaneously
- each tag bank is partitioned into multiple data banks to enable streaming accesses to the data banks.

Figure 8-1 on page 8-4 shows the logical representation of the L2 cache bank structure. The diagram shows a configuration with all possible tag and data bank combinations.

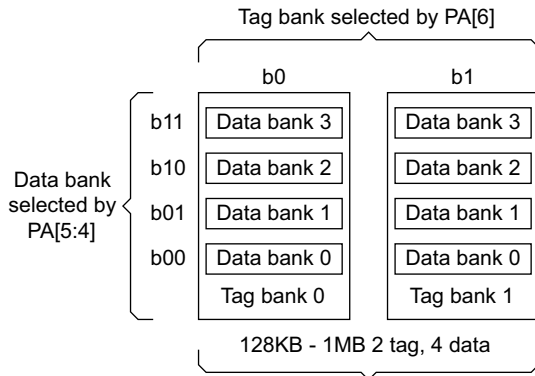


Figure 8-1 L2 cache bank structure

### 8.2.2 L2 cache transfer policy

Table 8-1 describes instruction and data transfers to and from the L2 cache.

Table 8-1 L2 cache transfer policy

Request type	L2 hit	L2 miss
Instruction miss (read)	L2 → L1	AXI → L1 AXI → L2
Data miss (read)	L2 → L1	AXI → L1 AXI → L2
NEON (read)	L2 → NEON	AXI → NEON AXI → L2
Data or NEON (write)	Write data → L2 Read, modify, and write to recalculate error correction code if necessary	Initiates write allocate fill AXI (merged with write data) → L2
TLB table walk (instruction or data)	L2 → TLB	AXI → TLB AXI → L2



### 8.3 Enabling and disabling the L2 cache controller

The L2 cache is enabled when both the C bit of the CP15 Control Register *c1* and the L2EN bit of the CP15 Auxiliary Control Register, *c1*, are both active. If either of these bits is disabled, then the L2 cache is disabled.

To enable the L2 cache following a reset or to change the settings of the L2 Cache Auxiliary Control Register, you must use the following sequence:

1. Complete the processor reset sequence or disable the L2 cache.
2. Program the L2 Cache Auxiliary Control Register. See *c9, L2 Cache Auxiliary Control Register* on page 3-124 for details.

———— **Note** —————

If you have configured the processor to support parity or ECC memory, you must enable those features before you can program the C bit.

3. Program the Auxiliary Control Register to set the L2EN bit to 1. See *c1, Auxiliary Control Register* on page 3-61 for details.
4. Program the C bit in the CP15 Control Register *c1*. See *c1, Control Register* on page 3-58 for details.

To disable the L2 cache, but leave the L1 data cache enabled, use the following sequence:

1. Disable the C bit.
2. Clean and invalidate the L1 and L2 caches.
3. Disable the L2 cache by clearing the L2EN bit to 0.
4. Enable the C bit.

———— **Note** —————

To keep memory coherent when using cache maintenance operations, you must follow the L2 cache disabling sequence. Cache maintenance operations have an effect on the L1 and L2 caches when they are disabled. A cache maintenance operation can evict a cache line from the L1 data cache. If the L2EN bit is set to 1, the evicted cache line can be allocated to the L2 cache. If the L2EN bit is not set to 1, then evictions from the L1 data cache are sent directly to external memory using the AXI interface.

## 8.4 L2 PLE

The L2 cache controller supports transactions from a programmable preloading engine. This PLE is not the same *Dynamic Memory Allocation* (DMA) engine used in previous ARM family of processors but has a similar programming interface.

The L2 PLE has two channels to permit two blocks of data movement to or from the L2 cache RAM.

The L2 PLE shares the translation table base TTBR0, TTBR1 and control, TTBCR, registers with the main translation table walk hardware.

The L2 PLE also supports the ability to lock data to a specific L2 cache way. If software requires the data to always remain resident in the L2 cache way, software can lock the specific cache way per channel when the PLE transfers data to or from the L2 cache RAM. Locking of a specified way only guarantees that the PLE is within the L2 cache RAM after completion. If the way is not locked, it is possible that the software might have evicted or replaced data with the way that the PLE is transferring data. To lock a cache way, you must program the L2 Cache Lockdown Register c9. See *c9, L2 Cache Lockdown Register* on page 3-121 for more information.

The programming of other registers within the PLE is possible only within the secure privileged state with specific extensions as described in this section. You can reprogram this capability using the Nonsecure Access Control Register and setting the PLE bit [18] to 1. If you program any register in nonsecure privileged state when the PLE bit [18] is 0, an Undefined Instruction exception occurs. Additionally, you can use the software to program the L2 Preload Engine Control Register UM bit [26] to 1 to enable more accessibility to the PLE registers.

### 8.4.1 Configuring the preload engine

To start the PLE, the software must program the following registers:

- L2 PLE User Accessibility
- L2 PLE Channel Number
- L2 PLE Control
- L2 PLE Internal Start Address
- L2 PLE Internal End Address
- L2 PLE Context ID.

After the software has programmed the registers, it enables the PLE by programming the L2 PLE Enable Register with a start command. The start command triggers data to be transferred to or from the L2 cache RAM as defined by DT bit [30] of the L2 PLE

Control Register. The internal start address defines the block of data transfer beginning at the 64-byte aligned address and ending when the number of cache lines is transferred to or from the L2 cache as defined by the internal end address.

---

**Note**

---

The number of lines is limited to the size of the L2 cache RAM way.

---

If the direction bit indicates that data is being transferred into the L2 RAM, then the L2 RAM cache way is loaded. However, if the software programmed the direction bit to indicate the transferring of data from the L2 RAM, then each address performs an L2 RAM lookup. Any cache line found to be dirty is evicted from the L2 cache RAM.

---

**Note**

---

It is entirely possible that the L1 data cache contains the same line that is transferred by the PLE engine to the external memory. Therefore it is possible for the line to become valid in the L2 cache as a result of an L1 eviction.

---

During data transfers into the L2 cache RAM, any L2 cache RAM data present in a different L2 cache RAM way, other than the way specified by the L2 PLE Control Register bits [2:0], remain in the different way. The preload engine continues with the next cache line to be loaded and the line is not relocated to the specified way.

During transfers to or from the L2 cache RAM, if the PLE crosses a page boundary, a hardware translation table walk is performed to obtain a new physical address for that new page. All standard fault checks are also performed. If a fault occurs, the PLE signals an interrupt on error. The PLE updates the L2 PLE Channel Status Register to capture the fault status. The address where the fault occurred is captured in the L2 PLE Internal Start Address Register.

When a PLE channel completes the transfer of the data block to or from the L2 cache RAM, it signals an interrupt. This interrupt can be either secure, **nDMASIRQ**, or nonsecure, **nDMAIRQ**, if IC bit [29] in the L2 PLE Control Register is enabled. In addition, there might be an interrupt-on-error, **nDMAEXTERRIRQ**, indicated if the PLE aborts for any reason and if the interrupt-on-error bit is enabled.

If you program the PLE to load data into the L2 cache RAM, the PLE transfers data to the L2 cache RAM if the memory region type is cacheable. To determine the memory region type, the PLE performs a hardware translation table walk at the start of the sequence and for any 4KB page boundary. The PLE channel does not save any state for the table walk. The translation procedure is for exception checking purposes and for determination of the memory attributes of the page. Any unexpected L2 cache RAM hits found when using the PLE are ignored for any type of data transfer.

---

**Note**

---

Both channels can run concurrently and be programmed to transfer data from external memory to the same L2 cache RAM way. At the completion of both PLE transactions, the data from either channel 0 or 1 might be present in the L2 cache.

---

**8.4.2 Preload engine commands and status interaction**

When the preloading engine channel has been configured, the channel begins to transfer data after it executes the start command. If at any time during the transfer, a preloading engine channel command of stop or clear is executed, the following rules apply for that command:

**START** Channel status transitions from idle to running. It has no effect on a channel status of running. The start and end address registers are updated as preloading engine transfers complete.

**STOP** Channel status transitions from running to idle. The start and end address reflect the next transfer to occur. When the channel is stopped, the address plus stride of the last transfer is stored in the PLE Internal Start Address Register. In addition, the remaining number of cache lines to be transferred is stored in the PLE Internal End Address Register. Therefore, by executing a start command, the preloading engine continues from the point when it was stopped.

**CLEAR** Channel status transitions from error or complete to idle and the interrupt or error flag is cleared to 0. It has no effect on a channel status of running. The start and end address registers are unchanged.

---

**Note**

---

While the PLE channel is running, the contents of the PLE Internal Start and End Address registers are Unpredictable for that channel.

---

**8.4.3 Interaction of the preload engine with WFI**

If one or more channels of the preload engine are active when the processor executes a WFI instruction, the preload engine controller suspends the PLE channels to enable the processor to enter WFI. When the processor wakes up from WFI, the PLE controller restarts all suspended PLE channels.

If it is important for the PLE channels to complete the data transfer, software must poll each PLE Channel Status Register for a status of completion or error. When each channel has completed, software can then execute the WFI instruction.

#### 8.4.4 Memory region interaction with the preload engine

If you programmed the preload engine to load data into the L2 cache RAM, the preload engine transfers data to the L2 cache RAM if the memory region type is cacheable. Table 8-2 shows the memory region types that the L2 memory system considers cacheable or noncacheable.

**Table 8-2 Cacheable and noncacheable memory region types**

Memory region type	Cacheable
Strongly ordered	No
Shared device	No
Nonshared device	No
Write-through, nonshared	Yes
Write-through, shared	No
Write-back, no write-allocate, nonshared	Yes
Write-back, no write-allocate, shared	No
Write-back, write-allocate, nonshared	yes
Write-back, write-allocate, shared	No
Noncached	No

When the preload engine encounters a noncached memory region, including at the start of the transfer, the preload engine stops the transfer and marks the transfer as complete.

#### 8.4.5 Processor configuration and the impact on the preload engine

During a preloading engine transfer, if the processor configuration of the memory system that affects the preloading engine is altered, the behavior of the preloading engine is Unpredictable. Some processor configurations that cause Unpredictable behavior are of the following:

- altering the MMU enable
- modifying the memory region remap registers
- changing the Context ID
- disabling the L2 cache controller.

———— **Note** ————

You must enable the MMU for the PLE to operate. If you disabled the MMU during preloading engine configurations, the PLE treats all memory as noncacheable regardless of the state of the Memory Region Remap Registers.

**8.4.6 Effects of cache maintenance operations during preloading engine transfers**

When a CP15 operation is performed during a preloading engine transfer, the preload engine pauses the transfer of data and waits for all outstanding AXI transactions to complete. Following completion of the CP15 operation, the preload engine restarts.

## 8.5 Synchronization primitives

On previous versions of the ARM architectures, support for shared memory synchronization was 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. See the *ARM Architecture Reference Manual* for details of the swap instructions.

ARMv7-A describes support for more comprehensive shared-memory synchronization primitives that scale for multiple-processor system designs. Two sets of instructions are introduced to support multiple-processor and shared-memory inter-process communication:

- load-exclusive, LDR{B,H,D}EX
- store-exclusive, STR{B,H,D}EX.

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 nonshared memory regions, the LDR{B,H,D}EX and STR{B,H,D}EX instructions are presented to the ports as normal LDR or STR. If a processor does an STR on a memory region that it has already marked as exclusive, this does not clear the tag. However, if the region has been marked by another processor, an STR clears the tag.

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 processor internal monitor in its exclusive state and might affect your software. If it does, you must execute a store-exclusive to an unused location in your abort handler or use the CLREX instruction to clear the processor internal monitor to an open state.

---

### 8.5.1 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.

## 8.5.2 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 on BRESP, indicating whether the write was successful. If BRESP is EXOKAY, then the destination register is written with a value of 0. Otherwise, it is written with a value of 1. The exclusive monitor is cleared after completion.

A store-exclusive that fails because of the local monitor does not cause a translation table walk, MMU fault, or watchpoint.

## 8.5.3 Example of LDREX and STREX usage

The following 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 LDREX  R0, [LockAddr]   ; load the lock value
    CMP R0, #0              ; is the lock free?
    STREXEQ R1, R0, [LockAddr]; try and claim the lock
    CMPEQ R0, #0            ; did this succeed?
    BNE try                 ; no - try again. . .
; yes - we have the lock

```



## 8.6 Locked access

The AXI protocol specifies that, when a locked transaction occurs, the master must follow the locked transaction with an unlocked transaction to remove the lock of the interconnect. The locked sequence is not complete until the end of the locking transaction. The SWP{B,H} instructions include separate read and write transactions on the AXI. The read transaction is marked as a locked transaction while the write transaction is not marked as a locked transaction. Therefore, the write transaction serves as the unlocking transaction and the AXI interconnect is unlocked when the write response is generated.

The SWP{B,H} instructions can access cacheable or noncacheable memory. If it is to cacheable memory, the bus transaction is not marked as a locked transaction. If it is to noncacheable memory, both the read and write transactions are treated as strongly ordered memory type, and the bus transaction is marked as a locked transaction.

If an abort occurs, the swapping of data between the register and memory is unsuccessful. To clear the lock, the processor issues a write transaction on the AXI interface without any byte strobes active.

———— **Note** —————

All transactions related to the swap instructions are issued with the lock indicator on its respective port, **ARLOCK** or **AWLOCK**.

---

## 8.7 Parity and error correction code

The L2 memory supports parity detection on the tag arrays. The data arrays can support parity or *Error Correction Code* (ECC). If ECC support is implemented, two extra cycles are added to the L2 pipeline to perform the checking and correction functionality. In addition, ECC introduces extra cycles to support read-modified-write conditions when a subset of the data covered by the ECC logic is updated. The ECC supports single-bit correction and double-bit detection.

The L2 Cache Auxiliary Control Register bits [28] and [21] control the parity and ECC support.

If a cache access result is a parity error or double bit ECC error in the L2 Cache, then both the L1 data cache and the L2 cache are unpredictable. No recovery is possible. The abort handler must:

- disable the caches
- communicate the fail directly with the external system
- request a reboot.

# Chapter 9

## External Memory Interface

This chapter describes the features of the AXI interconnect used by the processor. It contains the following sections:

- *About the external memory interface* on page 9-2
- *AXI control signals in the processor* on page 9-4
- *AXI instruction transactions* on page 9-7
- *AXI data read/write transactions* on page 9-8.

## 9.1 About the external memory interface

The external memory interface enables the processor to interface with third level caches, peripherals, and external memory. You can configure the processor to connect to either a 64-bit or 128-bit AXI interconnect that provides flexibility to system designs. The external memory interface supports the following interfaces:

- read address channel
- read data/response channel
- write address channel
- write data channel
- write response channel.

All internal requests that require access to an external interface must use the appropriate external interface. You can generate requests with the following:

- instruction fetch unit
- load/store unit
- table walk
- preload engine
- internal L2 cache controller.

By using the features of the AXI interconnect that enable split address and data transactions, in addition to multiple outstanding requests, the processor can reduce the external pin interface without reducing performance. The processor has a single AXI master interface. It does not contain an AXI slave interface.

### 9.1.1 External interface servicing instruction fetch transactions

The L2 memory system handles all instruction-side cache misses, including those for noncacheable memory. All instruction fetch requests are read-only and are routed to the external read address and data channels. For cacheable memory accesses, a wrapping burst transaction is generated to fetch an entire cache line from external memory. A nonwrapping burst transaction is generated by the L2 memory system for noncacheable, strongly ordered, or device memory instruction fetch accesses. See Table 9-5 on page 9-7 for information on AXI instruction transactions.

### 9.1.2 External interface servicing data transactions

The L2 memory system handles all data-side cache misses, including those for noncacheable memory, and those generated by the preload engine. Read data accesses are routed to the read address and data channels, whereas write data accesses are routed

to the write address and data channels. Swap and semaphore instruction support is also built into the L2 memory system and external interface that are unique to data-side accesses.

Cacheable accesses generate a wrapping burst transaction on the external interface. Strongly ordered, device, and noncacheable accesses typically result in single transaction requests to external interface. See Table 9-7 on page 9-10 for information on data transactions.

## 9.2 AXI control signals in the processor

For additional information about AXI control signals, see the *AMBA AXI Protocol Specification*.

### 9.2.1 AXI identifiers

The AXI interconnect uses identifiers with each transaction that enables requests to be serviced out-of-order under certain circumstances. The processor supports multiple outstanding transactions and assigns unique IDs to each specific transaction. There are two sets of identifiers, one for the read address channel, **ARRID[3:0]**, and one for the write address channel, **AWRID[3:0]**. Table 9-1 shows the AXI ID assignment for read address channel.

**Table 9-1 Read address channel AXI ID**

Read address channel request type		ID tag value
Instruction fetch	L2 cacheable	b1000-b1011
	L1 cacheable (L2 non-cacheable)	b1111
	Noncacheable or Strongly Ordered	b0100
	Shared or nonshared device	b0101
Integer data and CP14 loads	L2 cacheable	b1000-b1011
	L1 cacheable (L2 non-cacheable)	b1110
	Noncacheable or Strongly Ordered	b0000
	Shared device	b0001
	Nonshared device	b0011
NEON and VFP loads	L2 cacheable	b1000-b1011
	Noncacheable or Strongly Ordered	b0000
	Shared device	b0001
	Nonshared device	b0011
Table Walks	L2 cacheable	b1000-b1011
	Noncacheable	b0110
PLD and PLE	L2 cacheable	b1000-b1011

Table 9-2 shows the AXI ID assignment for write address channel.

**Table 9-2 Write address channel AXI ID**

<b>Write address channel request type</b>		<b>ID tag value</b>
Evictions from L2 cache	Each ID corresponds to one eviction in the L2 cache	b1000 - b1011
Integer data and CP14 writes	Noncacheable/cacheable	b0000
	Write-through/strongly ordered	
	Cacheable non-burst writes	b0010
	Shared device	b0001
	Nonshared device	b0011
NEON and VFP writes	Noncacheable/cacheable	b0000
	Write-through/strongly ordered	
	Cacheable non-burst writes	b0010
	Shared device	b0001
	Nonshared device	b0011
PLE	L2 cacheable	b1000-b1011
CP15 (cache maintenance)	L2 cacheable	b1000-b1011

The processor supports multiple read and write channel transactions as Table 9-3 shows.

**Table 9-3 AXI master interface attributes**

<b>Attribute</b>	<b>Outstanding transactions</b>
Write Issuing Capability	12
Read Issuing Capability	18
Combined Issuing Capability	26 <sup>a</sup>

- a. The combined issuing capability is limited to a total of four outstanding linefills or evictions. Therefore the sum of the read and write issuing capability does not equal the combined issuing capability.

## 9.2.2 Read/write data bus width configuration pin

The primary input pin **A64n128** of the processor determines the width of the AXI interface read/write data buses. You must ensure that this pin is driven appropriately for your system configuration.

Table 9-4 shows the supported values for **A64n128**.

**Table 9-4 A64n128 encoding**

<b>Value</b>	<b>Description</b>
0	128-bit interface
1	64-bit interface



## 9.3 AXI instruction transactions

This section describes the AXI master interface behavior for instruction side transactions to either Cacheable or Noncacheable regions of memory.

See the *AMBA AXI Protocol Specification* for details of the other AXI signals.

### 9.3.1 AXI instruction address transactions

Table 9-5 shows the values of **ARADDR[31:0]**, **ARLEN[3:0]**, **ARSIZE[2:0]**, **ARBURST[1:0]**, and **ARLOCK[1:0]** for instruction transactions.

**Table 9-5 AXI address channel for instruction transactions**

Transfer	Bus width	ARADDR [31:0] <sup>a</sup>	ARLEN [3:0]	ARSIZE [2:0]	ARBURST[ 1:0]	ARLOCK [1:0]
MMU translation table translation table walk <sup>b</sup>	64	[31:6]bbbb00	0	32-bit	Incr	Normal
	128	[31:6]bbbb00	0	32-bit	Incr	Normal
Noncacheable	64	[31:6]bbb000	0-7	64-bit	Incr	Normal
	128	[31:6]bb0000	0-3	128-bit	Incr	Normal
Cacheable linefill	64	[31:6]bbb000	7	64-bit	Wrap	Normal
	128	[31:6]bb0000	3	128-bit	Wrap	Normal

- a. **ARADDR[31:0]** is a 32-bit signal with bits [5:3] set to any value and bits [2:0] set to 0, unless otherwise indicated. This determines the **ARLEN[3:0]** value depending on the transfer type and bus width. For example, a noncacheable instruction fetch with **ARADDR[5:0] = b101000** for a 64-bit bus width, results in an **ARLEN[3:0] = b0010**. In this example, doublewords 5, 6, and 7 of the cache line are transferred.
- b. This is for noncacheable or strongly ordered table walk only. For cacheable table walk, the bus transaction is a cacheable linefill.

## 9.4 AXI data read/write transactions

This section describes the AXI master interface behavior for data read/write transactions.

### 9.4.1 Linefills

An AXI wrapping burst transaction transfers a cache line from external memory to the internal caches. The critical doubleword or quadword, depending on the 64-bit or 128-bit bus configuration, is requested first. In the event of an external abort, the cache line is not written into the caches and the line is never marked as valid.

### 9.4.2 Evictions

To reduce the number of burst transfers on the AXI interface, a subset of the cache line is written only if it is partially dirty. The burst size depends on the bus configuration and which quadwords of the cache line contain dirty data.

The CortexA8 processor contains four dirty bits per 64 byte cache line, representing four 128-bit packets of data, or four quadwords of data. The encoding of the dirty bits defines the number, or length, of transfers on the AWLEN[3:0] signals. Table 9-6 shows the different possible evictions that can take place for all combinations of dirty bits.

**Table 9-6 Number of transfers on AXI write channel for an eviction**

<b>Quadword Dirty Bit[3:0]</b>	<b>128-bit AXI AWLEN[3:0]</b>	<b>64-bit AXI AWLEN[3:0]</b>
b0000	No eviction	No eviction
b0001	b0000	b0001
b0010	b0000	b0001
b0011	b0001	b0011
b0100	b0000	b0001
b0101	b0010	b0101
b0110	b0001	b0011
b0111	b0010	b0101
b1000	b0000	b0001
b1001	b0011	b0111

Table 9-6 Number of transfers on AXI write channel for an eviction (continued)

Quadword Dirty Bit[3:0]	128-bit AXI AWLEN[3:0]	64-bit AXI AWLEN[3:0]
b1010	b0010	b0101
b1011	b0011	b0111
b1100	b0001	b0011
b1101	b0011	b0111
b1110	b0010	b0101
b1111	b0011	b0111

### 9.4.3 NEON accesses to strongly ordered and device memory

NEON vector type transfers are based on an element size and can require multiple AXI transfers. Each transfer consists of incrementing burst transactions of up to 128-bit bus width boundary. For example, if the Advanced SIMD instruction `VLD1.16 {D0}, [r1]` is executed to address offset `0xE`, then the following two burst transactions are generated on the AXI interface. The first transaction consists of the following:

- **ARBURST[1:0]** = `0x1`
- **ARLEN[3:0]** = `0x0` for single data transfer
- **ARSIZE[2:0]** = `0x1`.

The second transaction consists of the following:

- **ARBURST[1:0]** = `0x1`
- **ARLEN[3:0]** = `0x2` for three data transfers
- **ARSIZE[2:0]** = `0x1`.

### 9.4.4 AXI data address transactions

Table 9-7 on page 9-10 shows the values of **AxADDR[31:0]**, **AxLEN[3:0]**, **AxSIZE[2:0]**, **AxBURST[1:0]**, and **AxLOCK[1:0]** for data transactions excluding load/store multiples.

In this table:

<b>NA</b>	Naturally Aligned
<b>BW</b>	Bus Width
<b>BC</b>	Boundary Cross

<b>NoT</b>	Number of Transactions
<b>TS</b>	Transaction sequence number if multiple transactions are required
<b>SAO</b>	Starting Address Offset
<b>AxA</b>	<b>AxADDR</b> , either <b>ARADDR</b> or <b>AWADDR</b>
<b>AxLN</b>	<b>AxLEN</b> , either <b>ARLEN</b> or <b>AWLEN</b>
<b>AxS</b>	<b>AxSIZE</b> , either <b>ARSIZE</b> or <b>AWSIZE</b>
<b>AxB</b>	<b>AxBURST</b> , either <b>ARBURST</b> or <b>AWBURST</b>
<b>AxLK</b>	<b>AxLOCK</b> , either <b>ARLOCK</b> or <b>AWLOCK</b>

Table 9-7 AXI address channel for data transactions - excluding load/store multiples

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]
MMU translation table walk <sup>b</sup>	Yes	64	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal
		128	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal
Noncacheable, or strongly ordered, or device load byte	Yes	64	N/A	1	-	-	[31:0]	0	8-bit	Incr	Normal
		128	N/A	1	-	-	[31:0]	0	8-bit	Incr	Normal
Noncacheable, or strongly ordered, or device load halfword	Yes	64	N/A	1	-	-	[31:1]0	0	16-bit	Incr	Normal
		128	N/A	1	-	-	[31:1]0	0	16-bit	Incr	Normal

Table 9-7 AXI address channel for data transactions - excluding load/store multiples (continued)

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]	
Noncacheable load halfword	No	64	QW	2	1st	-	[31:1]0	0	16-bit	Incr	Normal	
			QW	2	2nd	-	[31:4]0000	0	16-bit	Incr	Normal	
			DW	1	-	-	[31:1]0	1	16-bit	Incr	Normal	
			W	1	-	-	[31:3]000	0	64-bit	Incr	Normal	
			HW	1	-	-	[31:2]00	0	32-bit	Incr	Normal	
	128	QW	2	1st	-	[31:1]0	0	16-bit	Incr	Normal		
					-	[31:4]0000	0	16-bit	Incr	Normal		
				DW	1	-	-	[31:4]0000	0	128-bit	Incr	Normal
				W	1	-	-	[31:3]000	0	64-bit	Incr	Normal
				HW	1	-	-	[31:2]00	0	32-bit	Incr	Normal
Noncacheable, or strongly ordered, or device load word	Yes	64	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal	
		128	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal	
Noncacheable load word	No	64	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal	
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal	
			DW	1	-	-	[31:2]00	1	32-bit	Incr	Normal	
			W	1	-	-	[31:3]000	0	64-bit	Incr	Normal	
	128	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal		
					-	[31:4]0000	0	32-bit	Incr	Normal		
				DW	1	-	-	[31:4]0000	0	128-bit	Incr	Normal
				W	1	-	-	[31:3]000	0	64-bit	Incr	Normal
Noncacheable, or strongly ordered, or device load doubleword	Yes	64	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Normal	
		128	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Normal	

Table 9-7 AXI address channel for data transactions - excluding load/store multiples (continued)

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]
Noncacheable, or strongly ordered, or device load doubleword	No	64	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal
		128	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:3]000	0	32-bit	Incr	Normal
	Yes	64	N/A	1	-	-	[31:0]	0	8-bit	Incr	Normal
					128	N/A	1	-	-	[31:0]	0
		128	N/A	1	-	-	[31:1]0	0	16-bit	Incr	Normal
					128	N/A	1	-	-	[31:1]0	0
Noncacheable store halfword	No	64	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal
		128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal
		128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal
		128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal
		128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal
128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal		
			2nd	-	[31:4]0000	0	8-bit	Incr	Normal		
128	QW	2	1st	-	[31:0]	0	8-bit	Incr	Normal		
			2nd	-	[31:4]0000	0	8-bit	Incr	Normal		

Table 9-7 AXI address channel for data transactions - excluding load/store multiples (continued)

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]				
Noncacheable, or strongly ordered, or device store word	Yes	64	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal				
		128	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Normal				
Noncacheable store word	No	64	QW	2	1st	0xD	[31:3]000	0	32-bit	Incr	Normal				
					2nd	-	[31:4]0000	0	8-bit	Incr	Normal				
					1st	0xE	[31:4]1110	0	16-bit	Incr	Normal				
					2nd	-	[31:4]0000	0	16-bit	Incr	Normal				
					1st	0xF	[31:4]1111	0	8-bit	Incr	Normal				
					2nd	-	[31:4]0000	0	64-bit	Incr	Normal				
		DW	1	-	0x5	[31:3]000	1	64-bit	Incr	Normal					
						0x6	[31:3]110	1	16-bit	Incr	Normal				
						0x7	[31:3]000	1	64-bit	Incr	Normal				
		W	1	-	-	[31:3]000	0	64-bit	Incr	Normal					
						128	QW	2	1st	0xD	[31:3]000	0	64-bit	Incr	Normal
									2nd	-	[31:4]0000	0	8-bit	Incr	Normal
		1st	0xE	[31:4]1110	0				16-bit	Incr	Normal				
		2nd	-	[31:4]0000	0				16-bit	Incr	Normal				
		1st	0xF	[31:4]1111	0				8-bit	Incr	Normal				
		2nd	-	[31:4]0000	0				64-bit	Incr	Normal				
		DW	1	-	0x5	[31:4]0000	0	128-bit	Incr	Normal					
						0x6	[31:4]0000	0	128-bit	Incr	Normal				
0x7	[31:4]0000					0	128-bit	Incr	Normal						
W	1	-	-	[31:3]000	0	64-bit	Incr	Normal							

Table 9-7 AXI address channel for data transactions - excluding load/store multiples (continued)

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]
Noncacheable, or strongly ordered, or device store doubleword	Yes	64	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Normal
		128	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Normal
Noncacheable store doubleword	No	64	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal
		DW	1 <sup>c</sup>	-	-	[31:2]00	1	32-bit	Incr	Normal	
				2 <sup>d</sup>	1st	-	[31:2]00	0	32-bit	Incr	Normal
		128	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal
DW	1	-	-	[31:4]0000	0	128-bit	Incr	Normal			
Strongly ordered, or device store doubleword	No	64	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal
			DW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:3]000	0	32-bit	Incr	Normal
		128	QW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:4]0000	0	32-bit	Incr	Normal
			DW	2	1st	-	[31:2]00	0	32-bit	Incr	Normal
					2nd	-	[31:3]000	0	32-bit	Incr	Normal
Cacheable linefill	Yes	64	N/A	8	-	-	[31:3]000	7	64-bit	Wrap	Normal
		128	N/A	4	-	-	[31:4]0000	3	128-bit	Wrap	Normal
Eviction/castout	Yes	64	N/A	8	-	-	[31:3]000	7	64-bit	Incr	Normal
		128	N/A	4	-	-	[31:4]0000	3	128-bit	Incr	Normal
Swap byte (load/store)	Yes	64	N/A	1	-	-	[31:0]	0	8-bit	Incr	Locked
		128	N/A	1	-	-	[31:0]	0	8-bit	Incr	Locked



Table 9-7 AXI address channel for data transactions - excluding load/store multiples (continued)

Transfer	NA	BW	BC <sup>a</sup>	NoT	TS	SAO [3:0]	AxA [31:0]	AxLN [3:0]	AxS [2:0]	AxB [1:0]	AxLK [1:0]
Swap word (load/store)	Yes	64	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Locked
		128	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Locked
Exclusive byte (load/store)	Yes	64	N/A	1	-	-	[31:0]	0	8-bit	Incr	Exclusive
		128	N/A	1	-	-	[31:0]	0	8-bit	Incr	Exclusive
Exclusive half word (load/store)	Yes	64	N/A	1	-	-	[31:1]0	0	16-bit	Incr	Exclusive
		128	N/A	1	-	-	[31:1]0	0	16-bit	Incr	Exclusive
Exclusive word (load/store)	Yes	64	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Exclusive
		128	N/A	1	-	-	[31:2]00	0	32-bit	Incr	Exclusive
Exclusive doubleword (load/store)	Yes	64	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Exclusive
		128	N/A	1	-	-	[31:3]000	0	64-bit	Incr	Exclusive

a. In the Boundary cross column, HW = 16 bits, W = 32 bits, DW = 64 bits, and QW = 128 bits.

b. This is for noncacheable or strongly ordered table walk only. For cacheable table walk, the bus transaction is a cacheable linefill.

c. This is for write combining enabled.

d. This is for write combining disabled.

Table 9-8 on page 9-16 shows the values of **ARADDR[31:0]**, **ARLEN[2:0]**, **ARSIZE[2:0]**, **ARBURST[1:0]**, **ARLOCK[1:0]**, and **ARPROT[2:0]** for data transactions for load/store multiples.

In this table:

**ENR** Even Number Registers

**FA** First Access

**LA** Last Access

**Table 9-8 AXI address channel for data transactions for load/store multiples**

Transfer	Alignment	ENR	FA	LA	ARADDR [31:0]	ARLEN [2:0]	ARSIZE [2:0]	ARBURST [1:0]	ARLOCK [1:0]
Noncacheable, or strongly ordered, or device LDMs	Even word	Yes	1	0	[31:3]000	0	64-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	64-bit	Incr	Normal
			1	1	[31:3]000	0	64-bit	Incr	Normal
		No	1	0	[31:3]000	0	64-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	64-bit	Incr	Normal
			1	1	[31:3]000	0	32-bit	Incr	Normal
	Odd word	Yes	1	0	[31:2]00	0	32-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	32-bit	Incr	Normal
			No	1	0	[31:2]00	0	32-bit	Incr
0		0		[31:3]000	0	64-bit	Incr	Normal	
0		1		[31:3]000	0	64-bit	Incr	Normal	
1		1		[31:3]000	0	32-bit	Incr	Normal	
Noncacheable, or strongly ordered, or device STMs		Even word	Yes	1	0	[31:3]000	0	64-bit	Incr
	0			0	[31:3]000	0	64-bit	Incr	Normal
	0			1	[31:3]000	0	64-bit	Incr	Normal
	1			1	[31:3]000	0	64-bit	Incr	Normal
	No		1	0	[31:3]000	0	64-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	64-bit	Incr	Normal
			1	1	[31:3]000	0	32-bit	Incr	Normal

Table 9-8 AXI address channel for data transactions for load/store multiples (continued)

Transfer	Alignment	ENR	FA	LA	ARADDR [31:0]	ARLEN [2:0]	ARSIZE [2:0]	ARBURST [1:0]	ARLOCK [1:0]
	Odd word	Yes	1	0	[31:3]100	0	32-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	32-bit	Incr	Normal
	No	No	1	0	[31:3]100	0	32-bit	Incr	Normal
			0	0	[31:3]000	0	64-bit	Incr	Normal
			0	1	[31:3]000	0	64-bit	Incr	Normal



# Chapter 10

## **Clock, Reset, and Power Control**

This chapter describes the clock domains and reset inputs of the processor. It also describes dynamic and static power control techniques. It contains the following sections:

- *Clock domains* on page 10-2
- *Reset domains* on page 10-5
- *Power control* on page 10-10.

## 10.1 Clock domains

The processor has three major clock domains:

**CLK** High speed core clock used to clock all major processor interfaces. The L1 memory system uses both the rising and falling edges of **CLK**. If the implemented design uses logic requiring the negative edge of the **CLK** signal, the duty cycle of **CLK** must be 50%. Figure 10-1 shows this.

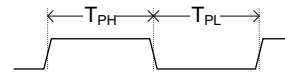


Figure 10-1 CLK duty cycle

CLK controls the following units within the processor:

- instruction fetch unit
- instruction decode unit
- instruction execute unit
- load/store unit
- L2 cache unit, including AXI interface
- NEON unit
- ETM unit, not including the ATB interface
- debug logic, not including the APB interface.

———— **Note** —————

The instruction fetch, instruction decode, instruction execute, load/store, and L2 cache are called the core or integer core.

**PCLK** APB clock that controls the debug interface for the processor. **PCLK** is asynchronous to **CLK** and **ATCLK**. **PCLK** controls the debug interface and logic in the **PCLK** domain.

**ATCLK** ATB clock that controls the ATB interface for the processor. **ATCLK** is asynchronous to **CLK** and **PCLK**. **ATCLK** controls the ATB interface.

———— **Note** —————

You can implement **PCLK** and **ATCLK** to be synchronous to **CLK**. You can also implement **PCLK** and **ATCLK** to run synchronously to each other.

### 10.1.1 AXI clocking using ACLKEN

The processor contains a single synchronous AXI interface. The AXI interface is clocked using a gated **CLK** that is gated using **ACLKEN**. The AXI interface can operate at any integer multiple slower than the processor clock, **CLK**. In previous ARM

family of processors, sampling **ACLKEN** on the rising edge of **CLK** indicated that the rising edge of the AXI bus clock, **ACLK**, had occurred. However, for the processor, the cycle timing of **ACLKEN** has changed.

Figure 10-2 shows the timing behavior of **ACLKEN**.

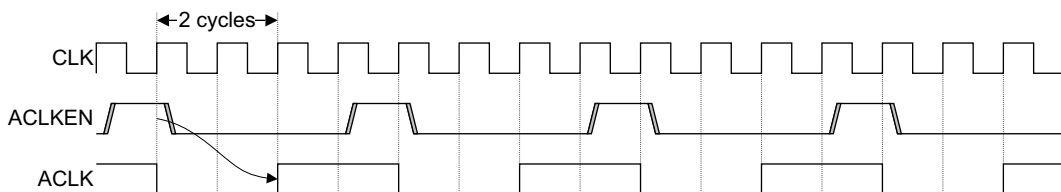


Figure 10-2 CLK-to-ACLK ratio of 4:1

————— **Note** —————

Figure 10-2 shows the timing relationship between the AXI bus clock, **ACLK**, and **ACLKEN**, where **ACLKEN** asserts two **CLK** cycles prior to the rising edge of **ACLK**. It is critical that the relationship between **ACLK** and **ACLKEN** is maintained.

Figure 10-3 shows a change to a 1:1 clock ratio. In this figure, **ACLKEN** remains asserted, changing the **CLK:ACLK** frequency ratio from 4:1 to 1:1.

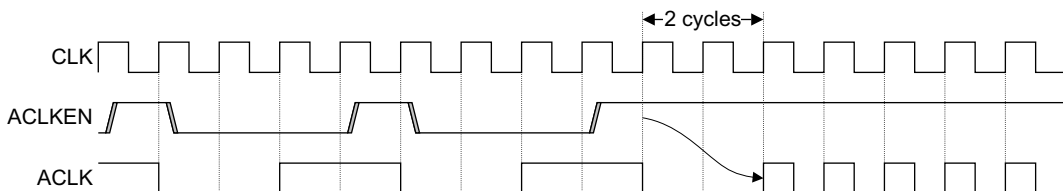
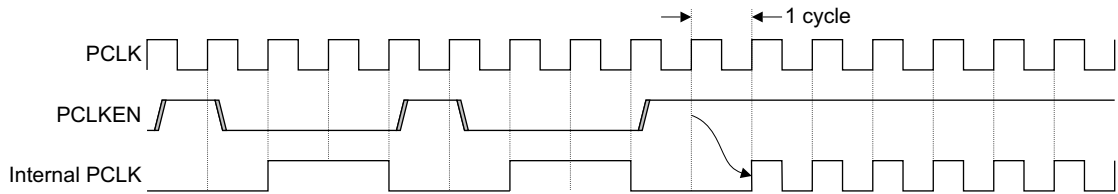


Figure 10-3 Changing the CLK-to-ACLK ratio from 4:1 to 1:1

### 10.1.2 Debug clocking using **PCLKEN**

All debug logic within the processor operates at an integer multiple of **PCLK** that is the same frequency as or slower than the APB clock, **PCLK**, using **PCLKEN**. Figure 10-4 on page 10-4 shows the behavior of **PCLKEN**. In this figure, **PCLKEN** remains asserted, changing the **PCLK:internal PCLK** frequency ratio from 4:1 to 1:1.



**Figure 10-4 Changing the PCLK-to-internal-PCLK ratio from 4:1 to 1:1**

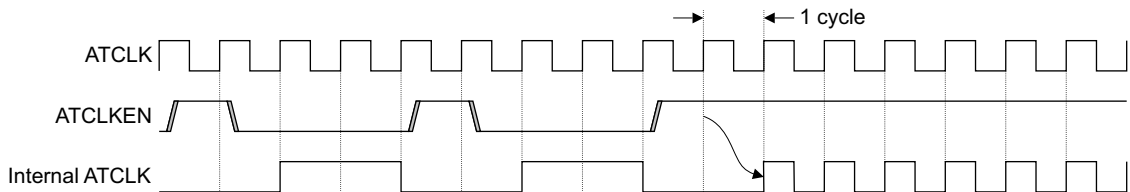
**Note**

If **PCLKEN** is not used, then it must be tied HIGH. This results in all state in the debug logic being clocked by **PCLK** directly.

### 10.1.3 ATB clocking using ATCLKEN

All ATB logic within the processor operates at an integer multiple of **ATCLK** that is the same frequency as or slower than the ATB clock, **ATCLK**, using **ATCLKEN**.

Figure 10-5 shows the behavior of **ATCLKEN**. In this figure, **ATCLKEN** remains asserted, changing the **ATCLK**:internal **ATCLK** frequency ratio from 4:1 to 1:1.



**Figure 10-5 Changing the ATCLK-to-internal-ATCLK ratio from 4:1 to 1:1**

**Note**

If **ATCLKEN** is not used, then it must be tied HIGH. This results in all state in the ATB logic being clocked by **ATCLK** directly.



## 10.2 Reset domains

Similar to the multiple clock domains within the processor, there are multiple reset domains:

- *Power-on reset*
- *Soft reset* on page 10-7
- *APB and ATB reset* on page 10-8
- *Hardware RAM array reset* on page 10-8
- *Reset of memory arrays* on page 10-9.

All resets are active-LOW inputs, and each reset can affect one or more clock domains. Table 10-1 shows the different resets and what areas of the processor are controlled by those resets.

**Table 10-1 Reset inputs**

Signal	Core (CLK)	NEON (CLK)	ETM (CLK)	Debug (CLK)	APB (PCLK)	ATB (ATCLK)
<b>nPORESET</b>	Reset	Reset	Reset	Reset	-	-
<b>ARESETn</b>	Reset	Reset	-	-	-	-
<b>PRESETn</b>	-	-	Reset	Reset	Reset	-
<b>ARESETNEONn</b>	-	Reset	-	-	-	-
<b>ATRESETn</b>	-	-	-	-	-	Reset

### Note

- There are specific requirements that must be met to reset each clock domain within the processor. Not adhering to these requirements can lead to a clock domain that is not functional.
- The documented reset sequences are the only reset sequences validated. Any deviation from the documented reset sequences might cause an improper reset of the clock domain.

### 10.2.1 Power-on reset

The power-on reset sequence is the most critical to the device because logic in all clock domains must be placed in a benign state following the deassertion of the reset sequence. Figure 10-6 on page 10-6 shows the power-on reset sequence.

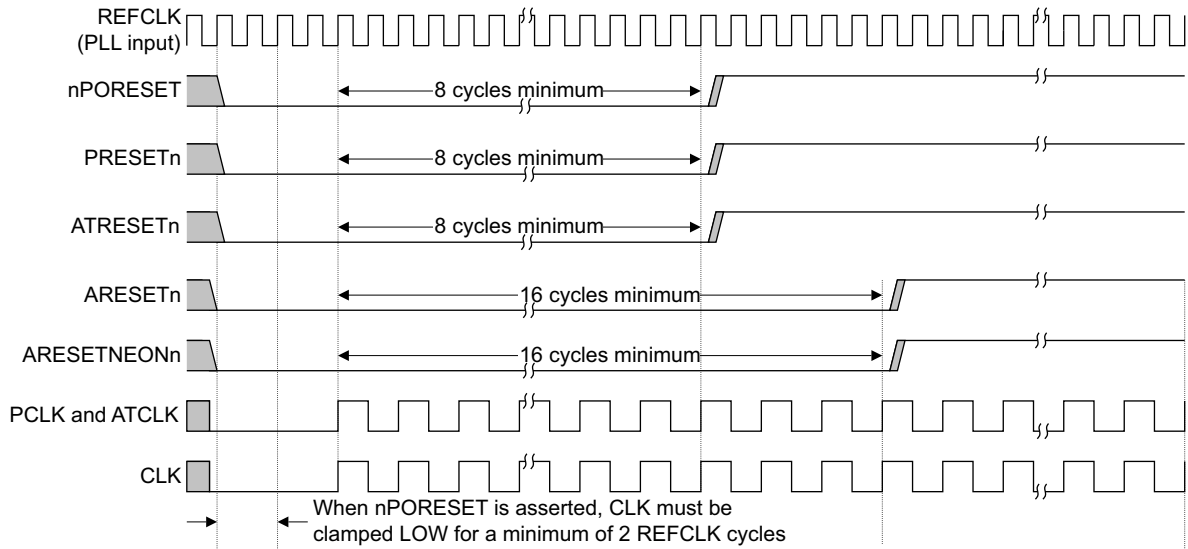


Figure 10-6 Power-on reset timing

Figure 10-6 shows three critical aspects:

1. At the beginning of power-on reset, **CLK** must be held LOW for a minimum of the equivalent of two **REFCLK** clock cycles to place components within the processor in a safe state.
2. The **nPORESET**, **PRESETn**, and **ATRESETn** resets must be held for eight **CLK** cycles. This ensures that reset has propagated to all locations within the processor.
3. The **ARESETn** and **ARESETNEONn** resets must be held for an additional eight **CLK** cycles following the release of **nPORESET** and **PRESETn** to enable those domains to exit reset safely.

#### Note

- The **PCLK** and **ATCLK** domains must also be reset during a power-on reset sequence to ensure that the interfaces between those domains and the **CLK** domain are reset properly.

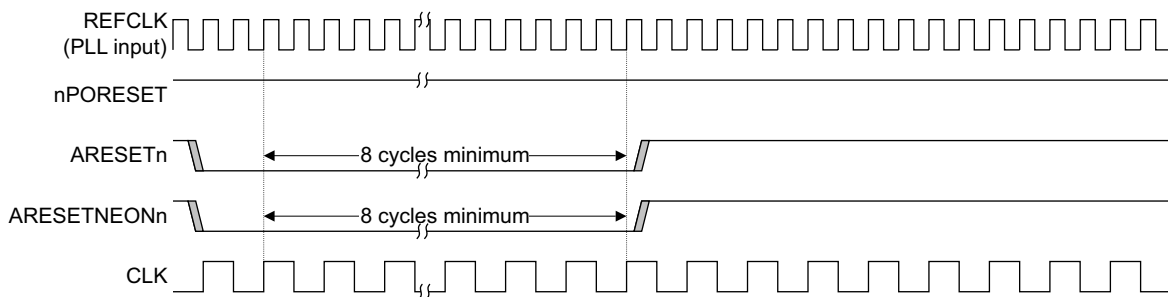
**PRESETn** and **ATRESETn** must be deasserted simultaneously with or after the deassertion of **nPORESET**.

- Figure 10-6 on page 10-6 shows that **PRESETn** must be asserted for a minimum of eight cycles. Because **PCLK** is an asynchronous clock domain that can operate faster or slower than **CLK**, **PRESETn** must be asserted for the slowest of eight **CLK** or eight **PCLK** cycles.

The power-on reset also controls entry and exit from a power-down state for various power domains within the processor. See *Power control* on page 10-10 for more information.

## 10.2.2 Soft reset

The soft reset sequence is used to trace with ETM or debug across a reset event. By asserting only the **ARESETn** and **ARESETNEONn** signals, the reset domains controlled by **nPORESET**, ETM, and debug in particular, are not reset. Therefore, breakpoints and watchpoints are retained during a soft reset sequence. Figure 10-7 shows a soft reset sequence.



**Figure 10-7** Soft reset timing

An additional reset is provided to control the NEON unit independently of the processor reset. This reset can be used to hold the NEON unit in a reset state so that the power to the NEON unit can be safely removed without placing any logic within the NEON unit in a different state. The reset cycle timing requirements for **ARESETNEONn** are identical to those for **ARESETn**. **ARESETNEONn** must be held for a minimum of eight **CLK** cycles when asserted to guarantee that the NEON unit has entered a reset state.

In addition, both **ARESETn** and **ARESETNEONn** are used to manage various power domains within the processor. See *Power control* on page 10-10 for information on the management of these resets and power domains.

### 10.2.3 APB and ATB reset

**PRESETn** is used to reset the debug hardware within the processor in addition to the ETM **CLK** domain. **ATRESETn** is used to reset the ATB interface and *Cross Trigger Interface* (CTI). To safely reset the debug hardware, ATB, and CTI domains, **PRESETn** and **ATRESETn** must be asserted for a minimum of eight clock cycles of the slowest of **CLK**, **PCLK**, or **ATCLK**. Figure 10-8 shows the assertion of **PRESETn** and **ATRESETn**.

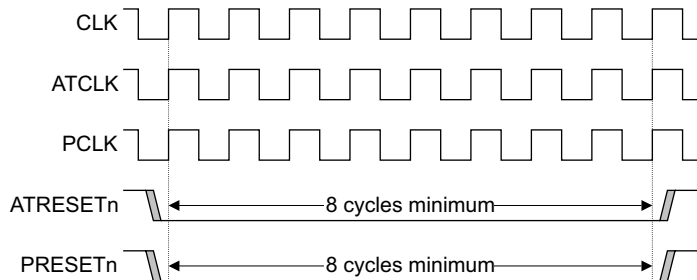


Figure 10-8 PRESETn and ATRESETn assertion

———— Note ————

**PRESETn** and **ATRESETn** must always be asserted simultaneously.

—————

### 10.2.4 Hardware RAM array reset

During a power-on reset or soft reset, by default the processor clears the valid bits of both the L1 data cache and the L2 unified cache. Depending on the size of the L2 cache, this can take up to 1024 cycles after the deasserting edge of the reset signals. The L1 data cache reset can take up to 512 cycles, and occurs coincident with the L1 instruction cache reset. The processor does not begin execution until the L1 caches are reset. The L2 hardware reset occurs in the background and does not interfere with reset code. Any attempt to enable the L2 unified cache or perform any L2 cache maintenance operations stalls the processor until the hardware reset is complete.

The processor has two pins, **L1RSTDISABLE** and **L2RSTDISABLE**, to control the hardware reset process. The usage models of the hardware reset pins are as follows:

1. For applications that *do not* retain the L1 data cache and L2 unified cache RAM contents throughout a core power-down sequence, the hardware resets both the L1 data cache and L2 unified cache at every reset, using **ARESETn** or **nPORESET**. Both **L1RSTDISABLE** and **L2RSTDISABLE** must be tied LOW. This is the recommended usage model.

2. For applications that *do* retain the L1 data cache or L2 unified cache RAM contents throughout a core power-down sequence, hardware must control both the **L1RSTDISABLE** and **L2RSTDISABLE** signals during reset. When the system is powering up for the first time, the hardware reset signals, **L1RSTDISABLE** and **L2RSTDISABLE**, must be tied LOW to invalidate both the L1 data cache and L2 unified cache RAM contents using the hardware reset mechanism. If either the L1 data cache or L2 unified cache must retain its data during a reset sequence, then the corresponding hardware reset disable must be tied HIGH.
3. If the hardware array reset mechanism is not used, then both the **L1RSTDISABLE** and **L2RSTDISABLE** pins must be tied HIGH.

Both the **L1RSTDISABLE** and **L2RSTDISABLE** pins must be valid at least 16 CLK cycles before and after the deasserting edge of **ARESETn** and **nPORESET**.

### 10.2.5 Reset of memory arrays

During reset of the processor, the following memory arrays are invalidated at reset:

- branch prediction arrays (BTB and GHB)
- L1 instruction and data TLBs
- L1 data cache valid RAM, if **L1RSTDISABLE** is tied LOW
- L2 unified cache valid RAM, if **L2RSTDISABLE** is tied LOW.

## 10.3 Power control

Both the clocks and resets in the processor play key roles in the power management of the processor, enabling islands to be powered down or powered up in a controlled manner. They also provide many key control mechanisms to manage dynamic power.

This section describes:

- *Dynamic power management*
- *Static or leakage power management* on page 10-14
- *Debugging the processor while powered down* on page 10-23
- *L1 data and L2 cache power domains* on page 10-25
- *Special note on reset during power transition* on page 10-29.

### 10.3.1 Dynamic power management

The processor has many different dynamic power management facilities. The most common form of dynamic power management is control of the clock network within the processor.

The processor has three levels of clock gating to manage dynamic power. The levels correspond to the following functions:

- Level 1** This is architectural gating, also known as *Wait-For-Interrupt* (WFI), or the **CLKSTOPREQ** and **CLKSTOPACK** signals on the Cortex-A8 processor.
- Level 2** This is major function gating, such as NEON, ETM, or integer core gating.
- Level 3** This is state element gating, such as local clock gating.

The processor contains all hardware necessary for architecture, unit, and local clock gating. No external hardware is required to clock gate the processor.

#### Wait-For-Interrupt architecture

Executing a Wait-For-Interrupt instruction puts the processor into a low-power state until one of the following occurs:

- an IRQ or FIQ interrupt
- a halting debug event when the **DBGNOCLKSTOP** signal is HIGH.

See *Halting debug event* on page 12-71 for information on halting debug events.

---

**Note**


---

- If you are debugging software running on the Cortex-A8 processor, **DBGNOCLKSTOP** must be HIGH. Otherwise, halting debug events do not work as architected and the APB interface does not return a response when accessing the ETM, CTI, or core domain debug registers. See Table 12-3 on page 12-9 for information on which debug registers are in the core.
  - If **DBGNOCLKSTOP** is HIGH and you execute the Wait-For-Interrupt instruction, the processor goes into an idle state but not into a low-power state.
  - The **STANDBYWFI** pin remains HIGH even when **DBGNOCLKSTOP** is HIGH.
- 

When executing the WFI instruction, the processor waits for the following events to complete before entering the idle or low-power state:

- L1 data memory system loads and stores are complete
- all L1 instruction memory system fetches are complete
- all L2 memory system transactions are complete
- all AXI interface transactions are complete
- all Advanced SIMD instructions are complete
- all ETM data transfers from core clock domain to ATB clock domain are complete
- preloading engine, PLE, activity is interrupted.

On entry into the low-power state, the processor asserts the **STANDBYWFI** signal. Assertion of **STANDBYWFI** guarantees that the processor and the AXI interface are in the idle state. The APB **PCLK** clock domain and the ATB **ATCLK** clock domain can remain active.

Figure 10-9 on page 10-12 shows the upper bound for the **STANDBYWFI** deassertion timing after assertion of **nIRQ** or **nFIQ**.

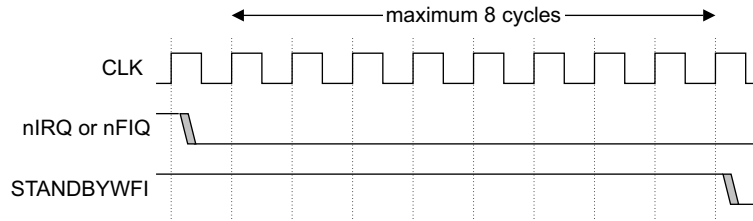


Figure 10-9 STANDBYWFI deassertion

### Hardware clock stopping

Another form of architectural clock gating is controlled by the processor **CLKSTOPREQ** input. Asserting **CLKSTOPREQ** puts the processor into a low-power state until **CLKSTOPREQ** is deasserted.

Figure 10-10 shows the relationship between **CLKSTOPREQ** and **CLKSTOPACK**.

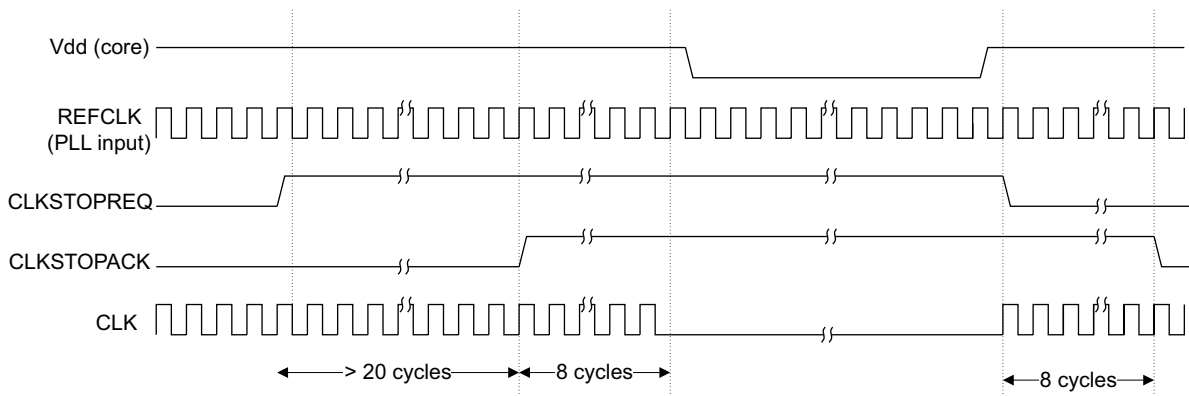


Figure 10-10 CLKSTOPREQ and CLKSTOPACK

When the system asserts **CLKSTOPREQ**, the processor waits for completion of the same events as in the Wait-For-Interrupt case before entering the low-power state. See *Wait-For-Interrupt architecture* on page 10-10 for more information.

On entry into the low-power state, the processor asserts the **CLKSTOPACK** output. Assertion of **CLKSTOPACK** guarantees that the processor and the AXI interface are in idle state. The APB **PCLK** domain and the ATB **ATCLK** clock domain can remain active.



The number of cycles between **CLKSTOPREQ** and **CLKSTOPACK** assertion has a lower bound of 20 cycles but no upper bound. The upper bound is a function of the latency to access the slowest device mapped on the processor AXI bus and, therefore, is system-dependent. After the processor asserts **CLKSTOPACK**, it closes the architectural clock gate. However, eight **CLK** cycles must pass before you can rely on the architectural clock gate being completely closed.

Figure 10-10 on page 10-12 shows the system stopping **CLK** after the architectural clock gate is closed. This enables additional energy savings, but it is optional. In addition, the supply voltage,  $V_{dd}$  (core) can also be lowered as shown in Figure 10-10 on page 10-12 to improve energy savings. However, **CLK** must not stop before the architectural clock gate is closed, that is, it must continue to run for at least eight cycles after **CLKSTOPACK** is asserted.

After the architectural clock gate closes, the system can keep the processor in this low-power state for as long as required, by holding **CLKSTOPREQ** HIGH. When the system deasserts **CLKSTOPREQ**, this causes the architectural clock gate to open. The processor then responds by deasserting **CLKSTOPACK** and resuming instruction execution. The upper bound for the number of **CLK** cycles between **CLKSTOPREQ** and **CLKSTOPACK** deassertion is 8.

When driving **CLKSTOPREQ**, the system must comply with a set of protocol rules, otherwise the processor behavior is Unpredictable. The rules are as follows:

- **CLKSTOPREQ** must not transition from LOW to HIGH if **CLKSTOPACK** is already HIGH.
- When **CLKSTOPREQ** is HIGH, it must remain HIGH until **CLKSTOPACK** goes HIGH. Only when **CLKSTOPACK** goes HIGH can **CLKSTOPREQ** go LOW.

---

#### Note

---

- If you are debugging software running on the Cortex-A8 processor, **DBGNOCLKSTOP** must be HIGH. Otherwise, halting debug events do not work as architected and the APB interface does not return a response when accessing the ETM, CTI, or core domain debug registers. See Table 12-3 on page 12-9 for information on which debug registers are in the core.
  - If **DBGNOCLKSTOP** is HIGH and the system asserts **CLKSTOPREQ**, the processor goes into an idle state but not into a low-power state.
  - The **CLKSTOPACK** output pin remains HIGH even when **DBGNOCLKSTOP** is HIGH.
-

## NEON or ETM unit level gating

In addition to the architectural gating mechanism, the processor supports gating of major components within the processor such as the NEON unit, VFP coprocessor, and ETM unit.

The cp10 and cp11 fields in the CP15 c1 Coprocessor Access Control Register control access to the NEON and VFP coprocessor. See *c1, Coprocessor Access Control Register* on page 3-67. Reset clears the cp10 and cp11 fields and disables the NEON and VFP clocks.

The ETM Control Register enables the ETM. See the *Embedded Trace Macrocell Architecture Specification* for more information. The global enable bit in the CTI Control Register enables the ETM clocks, excluding the ATB clock, **ATCLK**, which can only be gated external to the processor. See *CTI Control Register; CTICONTROL* on page 15-13.

## DFF gating

The finest level of dynamic power control is at the *Delay Flip-Flop* (DFF) level. This is implicit to the design and requires no external support.

### 10.3.2 Static or leakage power management

The processor can accommodate many different levels of static, or leakage power management. All of these techniques are specific to a given implementation of the processor. Some possibilities that the processor can accommodate are:

- full retention
- power domains or islands such as integer core, ETM and debug, L2 RAM, and NEON
- usage of multi-Vt such as high-Vt, standard-Vt, or low-Vt.

———— **Note** —————

This technical reference manual does not document retention or the usage of multi-Vt. However, this manual describes the power domains, or islands that are supported and the methods that are required to manage those domains in a manner that has been validated within the processor.

To completely eliminate leakage power consumption in the processor, you must remove the power supplied to the processor. Before powering down, all architectural state must be saved to memory and the L1 data cache or L2 unified cache must be cleaned to the

point of coherency. When powering up the processor, you must apply a complete reset sequence with software that restores the architectural state. The sequence takes significant time and energy to perform a full power-down of the processor.

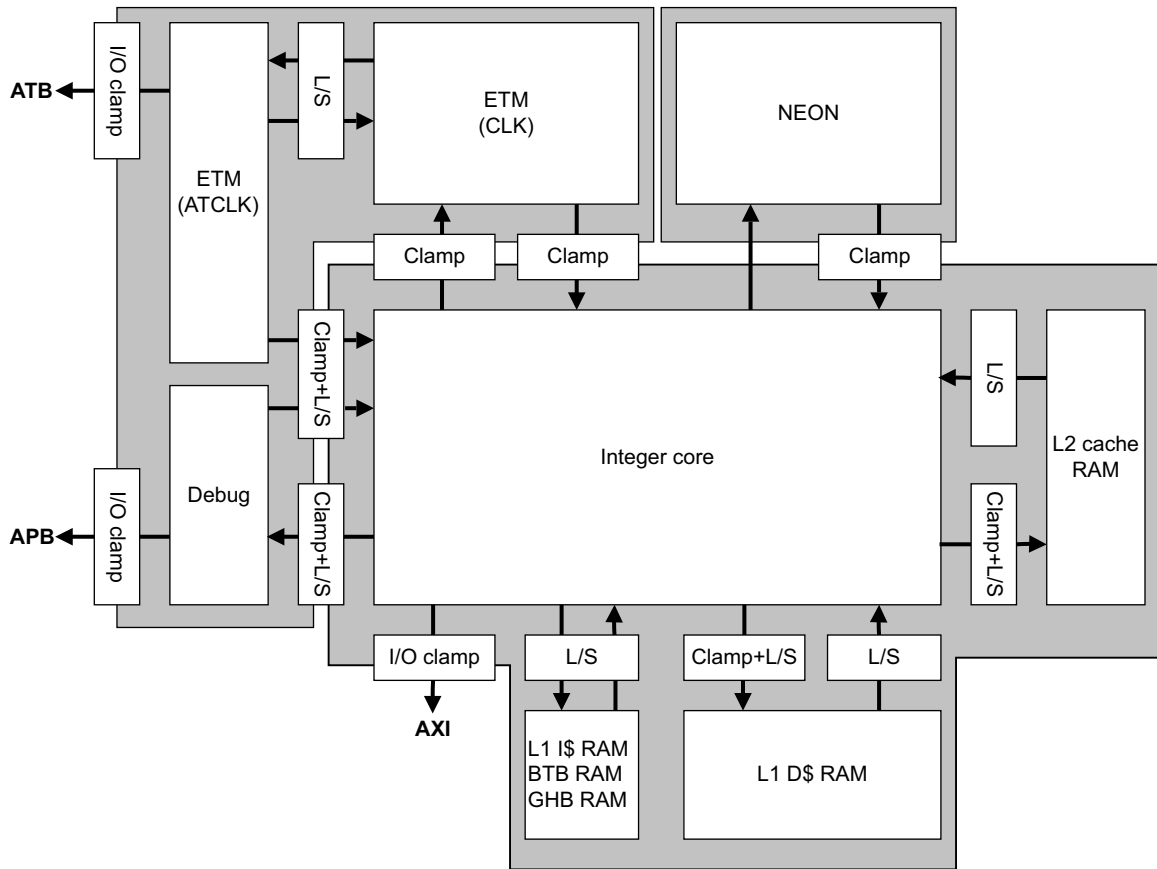
To improve the response time of a power-down sequence, the processor supports several key features to minimize the response time and to reduce the leakage power consumption:

- The processor enables the debug, ETM, and NEON units to be powered down while the rest of the processor is active.
- The processor is designed so that the L1 data cache or L2 unified cache can retain state while the rest of the processor is powered down. This avoids the time and energy consuming process of cleaning the caches before powering down.
- The processor enables the debug logic to remain powered up while the rest of the processor is powered down. This enables system debug to continue while the processor is powered down. All powered-down processor resources are not available to the debugger. As a result, the debug logic indicates an error to the debugger that the processor is in a powered-down state.

The processor supports many different power islands combinations, including a single monolithic power grid, resulting in a single power domain. The supported power domains are:

- the NEON unit
- all debug **PCLK** logic, ETM **CLK** logic, and ETM **ATCLK** logic
- the L2 cache arrays
- the L1 data cache arrays
- all remaining logic within the processor, excluding the previous power domain, also known as the integer core.

Figure 10-11 on page 10-16 shows the supported power domains.



L/S = Level Shift

**Figure 10-11 Power domains**

When implementing the different power domains, the following modes of operation apply:

- integer core in running mode:
  - All logic are powered and operational.
  - NEON are powered down and all other logic powered and operational. This mode minimizes the NEON leakage when NEON is not required.

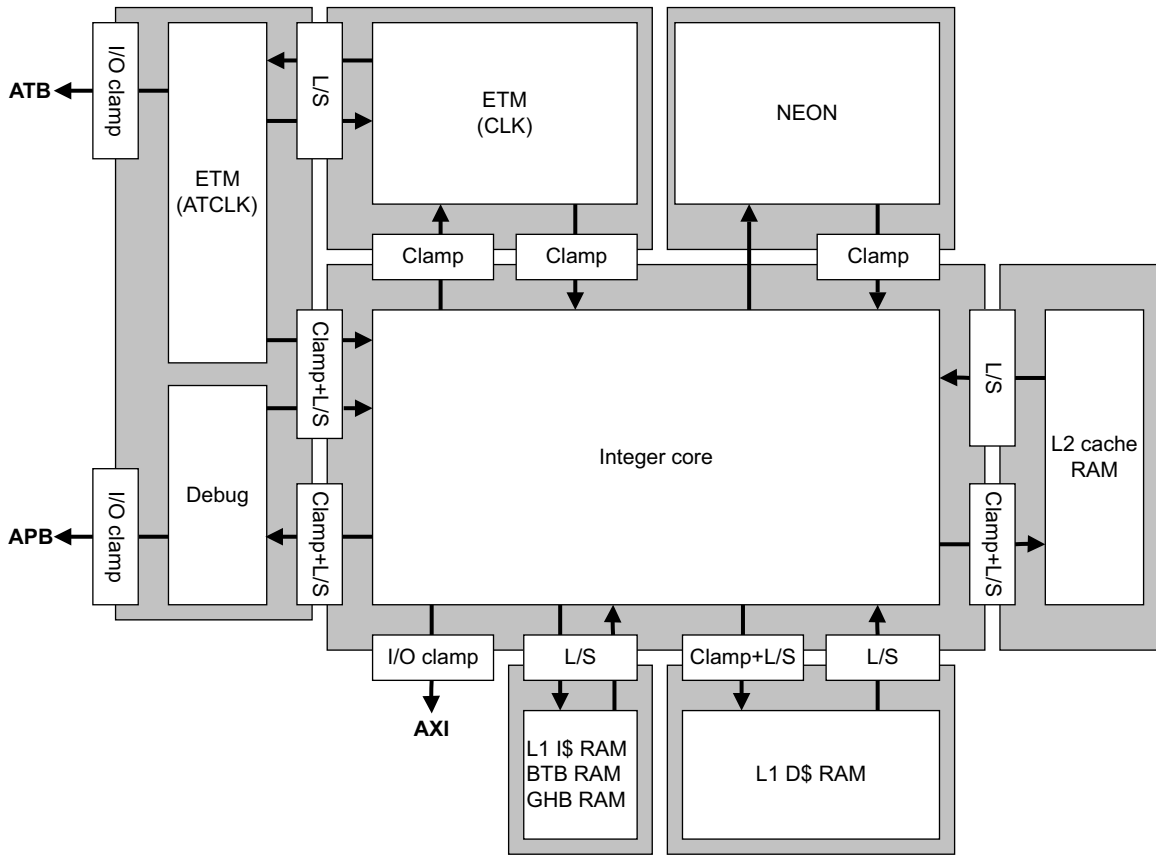
- Debug **PCLK**, **ETM CLK**, and **ETM ATCLK** are powered down and all other logic powered and operational. This mode minimizes the leakage of the debug and trace facilities when they are not required.
- NEON, debug **PCLK**, **ETM ATCLK**, and **ETM CLK** are powered down, with all other logic powered and operational.
- Integer core and NEON in powered down mode:
  - L1 data cache or L2 cache are powered up. This mode enables data to be retained in the L1 data cache or the L2 cache. This mode can greatly minimize the time and energy required to power down the processor.
  - Debug **PCLK**, **ETM CLK**, and **ETM ATCLK** are powered up. This mode enables the debug and trace external interfaces to remain active, enabling the debugger to detect that the processor is powered down.
  - L1 data cache or L2 cache, debug **PCLK**, **ETM CLK**, and **ETM ATCLK** are powered down.

If all power domains are implemented, the power domains can be independently controlled to give eight combinations of power-up and power-down domains. However, only some power-up and power-down domain combinations are valid. These are shown in Table 10-2.

**Table 10-2 Valid power domains**

<b>Integer core</b>	<b>Debug and ETM</b>	<b>NEON</b>
Powered down	Powered down	Powered down
Powered down	Powered up	Powered down
Powered up	Powered down	Powered down
Powered up	Powered up	Powered down
Powered up	Powered down	Powered up
Powered up	Powered up	Powered up

From the power domains shown in Figure 10-11 on page 10-16, the following voltage domains can be derived. Figure 10-12 on page 10-18 shows this.



LS = Level Shift

Figure 10-12 Voltage domains

The voltage domains represent the power supply distributions that might be required in the Cortex-A8 processor. These include:

### **Debug PCLK and ETM ATCLK**

Connects to SoC debug power domain.

**ETM CLK** Operates at the same voltage as the processor but exists in same power domain as debug.

**NEON** Operates at the same voltage as the processor and can be powered down while the processor is running.

**L2 RAMs** Supports retention in the L2 cache, and supports SRAM voltage.

### **L1 data cache RAMs**

Supports retention in the L1 data cache, and supports SRAM voltage.

### **Other L1 RAMs**

Supports SRAM voltage.

### **Integer core**

All logic within the integer core, not including SRAMs.

Any or all of these voltage domains can be removed from the processor. However, the removal of those domains must comply with the supported power domain configurations listed in Table 10-2 on page 10-17.

### **NEON power domain**

If NEON is not required, you can reduce leakage by turning off the power to the NEON unit. While the NEON unit is powered down, any Advanced SIMD instructions executed take the Undefined Instruction exception. The OS uses the Undefined Instruction exception on an Advanced SIMD instruction as a signal to apply power to the NEON unit, if powered down, or to activate NEON, if disabled.

To enable NEON to be powered down, the implementation must place NEON on a separately controlled power supply. In addition, the outputs of NEON must be clamped to benign values while NEON is powered down, to indicate that NEON is idle.

### **Powering down the NEON power domain while the processor is in reset**

To power down the NEON power domain while the processor is in reset, apply the following sequence:

1. Assert both **ARESETn** and **ARESETNEONn** to place the processor in reset. You must assert **ARESETn** and **ARESETNEONn** for at least eight **CLK** cycles before activating the NEON clamps.
2. Activate the NEON output clamps by asserting the **CLAMPNEONOUT** input HIGH.
3. Remove power from the NEON power domain.
4. Deassert **ARESETn**, but continue to assert **ARESETNEONn**.

If the processor is executing a power-on reset sequence or is first powering up:

1. Assert both **ARESETn** and **ARESETNEONn**. You must assert **ARESETn** and **ARESETNEONn** for at least eight **CLK** cycles before activating the NEON clamps.
2. Activate the NEON output clamps by asserting the **CLAMPNEONOUT** input HIGH.
3. While keeping the NEON power domain off, supply power to the other active power domains.
4. Deassert **ARESETn**, but continue to assert **ARESETNEONn**.

While **ARESETNEONn** remains asserted, all Advanced SIMD instructions cause an Undefined Instruction exception.

———— **Note** —————

If **ARESETNEONn** is deasserted or the NEON output clamps are released without following one of the specified NEON power-up sequences, the results are Unpredictable and might cause the processor to deadlock.

---



**Powering down the NEON power domain while the processor is not in reset**

To power down the NEON power domain while the processor is not in reset, the NEON power domain must be placed into an idle state. Apply the following sequence to place the NEON power domain into an idle state:

1. Software must disable access to the NEON unit using the Coprocessor Access Control Register, see *c1, Coprocessor Access Control Register* on page 3-67. All outstanding Advanced SIMD instructions retire and all subsequent Advanced SIMD instruction cause an Undefined Instruction exception.
 

MRC p15, 0, <Rd>, c1, c0, 2;	Read Coprocessor Access Control Register
BIC <Rd>, <Rd>, #0xF00000;	Disable access to CP10 and CP11
MCR p15, 0, <Rd>, c1, c0, 2;	Write Coprocessor Access Control Register
2. Software must signal to the external system that the NEON unit is disabled.
3. Assert **ARESETNEONn** to place NEON in reset. You must assert **ARESETNEONn** for at least eight **CLK** cycles before activating the NEON clamps.
4. Activate the NEON output clamps by asserting the **CLAMPNEONOUT** input HIGH.
5. Remove power from the NEON power domain.

————— **Note** —————

If **ARESETNEONn** is deasserted or the NEON output clamps are released without following one of the specified NEON power-up sequences, the results are Unpredictable and might cause the processor to deadlock.

**Powering up the NEON power domain while the processor is in reset**

To apply power to the NEON power domain while the processor is in reset, use the following sequence:

1. Assert **ARESETn** and keep **ARESETNEONn** asserted.
2. Apply power to the NEON power domain.
3. Release the NEON output clamps by deasserting **CLAMPNEONOUT**.
4. Deassert **ARESETn** and **ARESETNEONn**.

After the completion of the reset sequence, you can enable the NEON unit using the Coprocessor Access Control Register. See *c1, Coprocessor Access Control Register* on page 3-67.

**Powering up the NEON power domain while the processor is not in reset**

To apply power to the NEON power domain while the processor is not in reset, use the sequence that follows. With the NEON power domain currently powered down, it is assumed that **ARESETNEONn** is asserted.

1. Software must disable access to the NEON unit using the Coprocessor Access Control Register, see *c1, Coprocessor Access Control Register* on page 3-67.
 

MRC p15, 0, <Rd>, c1, c0, 2;	Read Coprocessor Access Control Register
BIC <Rd>, <Rd>, #0xF00000;	Disable access to CP10 and CP11
MCR p15, 0, <Rd>, c1, c0, 2;	Write Coprocessor Access Control Register
2. Software must signal to the external system that it is safe to power up the NEON unit.
3. Apply power to the NEON power domain.
4. Deassert **ARESETNEONn**. NEON requires a minimum of 20 **CLK** cycles to complete its reset sequence. Therefore, the system must wait until NEON has completed its reset sequence before releasing the NEON clamps.
5. Release the NEON output clamps by deasserting **CLAMPNEONOUT**.
6. Software must poll the external system to determine that it is safe to enable the NEON unit.

After the completion of the reset sequence, you can enable the NEON unit using the Coprocessor Access Control Register. See *c1, Coprocessor Access Control Register* on page 3-67.

**Debug and ETM power domains**

If the core is running in an environment where debug facilities are not required, you can reduce leakage power by powering down the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains. Debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains must be built using a common power supply.

**Powering down the debug and ETM power domains**

To power down the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains, the implementation must place debug **PCLK**, **ETM CLK**, and **ETM ATCLK** on a separately controlled and shared power supply. In addition, the outputs of debug **PCLK**, **ETM CLK**, and **ETM ATCLK** must be clamped to benign values while powered down to indicate that the interface is idle.

To power down the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains, apply the following sequence:

1. Assert both **PRESETn** and **ATRESETn**. You must assert **PRESETn** for at least eight **PCLK** cycles and **ATRESETn** for at least eight **ATCLK** cycles before asserting **CLAMPDBGOUT**.
2. Activate the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** output clamps by asserting the **CLAMPDBGOUT** input HIGH.
3. Remove power from the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains. **PRESETn** and **ATRESETn** must remain asserted while the domain is powered down.

### Powering up the debug and ETM domains

To power up the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains, use the sequence that follows. It is assumed that both **PRESETn** and **ATRESETn** are asserted during the sequence.

1. Apply power to the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains.
2. Release the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** output clamps by deasserting **CLAMPDBGOUT**.
3. If the system uses the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** hardware, it is safe to deassert either **PRESETn**, **ATRESETn**, or both.

### 10.3.3 Debugging the processor while powered down

If the processor is powered down, the SoC can still be functional and used for debug across the power domains. If the debugger accesses the processor, the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** domains must be powered up. See Chapter 12 *Debug* for more information on debugging during power down.

If the integer core power domain is powered down while the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains are still powered up, all inputs from the integer core power domain to the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains must be clamped to benign values.

## Powering down the integer core power domain

Apply the following sequence to power down the integer core power domain:

1. Assert **DBGPWRDWNREQ** to indicate that processor debug and ETM resources are not available for APB accesses. Wait for **DBGPWRDWNACK** to be asserted.

———— **Note** —————

The **ETMPWRDWNREQ** and **ETMPWRDWNACK** signals are not required because debug and the ETM use the same power domain. **ETMPWRDWNREQ** must be tied to 0.

---

2. Assert **ARESETn**, **ARESETNEONn**, and **nPORESET**. You must assert **ARESETn**, **ARESETNEONn**, and **nPORESET** for at least eight **CLK** cycles before activating the integer core and NEON clamps.
3. Activate the NEON output clamps and the clamps to the debug **PCLK**, ETM **CLK**, and ETM **ATCLK** power domains from the core by asserting the **CLAMPCOREOUT** and **CLAMPNEONOUT** inputs HIGH.
4. Remove power from the integer core and NEON power domains while retaining power to the debug **PCLK**, ETM **CLK**, and ETM **ATCLK** power domains.

## Powering up the integer core and NEON power domains

Apply the following sequence to power up the integer core and NEON power domains:

1. Apply power to the integer core and NEON power domains while keeping **ARESETn**, **ARESETNEONn** and **nPORESET** asserted.
2. Release the NEON output clamps and the clamps to the debug **PCLK**, ETM **CLK**, and ETM **ATCLK** power domains from the core by deasserting **CLAMPCOREOUT** and **CLAMPNEONOUT**.
3. Deassert **DBGPWRDWNREQ** to indicate that processor debug and ETM resources are available. There is no requirement for hardware to wait for **DBGPWRDWNACK** to be deasserted.

———— **Note** —————

The **ETMPWRDWNREQ** and **ETMPWRDWNACK** signals are not required because debug and the ETM use the same power domain. **ETMPWRDWNREQ** must be tied to 0.

---

4. Continue a normal power-on reset sequence.

### Powering up the integer core power domain while keeping NEON powered down

Apply the following sequence to power up the integer core while keeping NEON powered down:

1. Apply power to the integer core power domain while keeping **ARESETn**, **ARESETNEONn** and **nPORESET** asserted. Be sure to keep the NEON power domain off.
2. Release the clamps to the debug **PCLK**, **ETM CLK**, and **ETM ATCLK** power domains from the core by deasserting **CLAMPCOREOUT** and keeping **CLAMPNEONOUT** asserted.
3. Deassert **DBGPWRDWNREQ** to indicate that processor debug and ETM resources are available. There is no requirement for hardware to wait for **DBGPWRDWNACK** to be deasserted.

————— **Note** —————

The **ETMPWRDWNREQ** and **ETMPWRDWNACK** signals are not required because debug and the ETM use the same power domain. **ETMPWRDWNREQ** must be tied to 0.

4. Continue a normal power-on reset sequence while **ARESETNEONn** and **CLAMPNEONOUT** remain asserted. To power up the NEON power domain, see *Powering up the NEON power domain while the processor is not in reset* on page 10-22.

#### 10.3.4 L1 data and L2 cache power domains

During periods when the entire core is not required, you can stop the processor clocks by executing a Wait For Interrupt instruction. However, leakage continues to occur. To remove the leakage component, you must remove the power supplied to the power domains within the processor. However, the time required to remove and restore the power limits the advantage of a full power-down of the processor. A full power-down sequence for the processor might include:

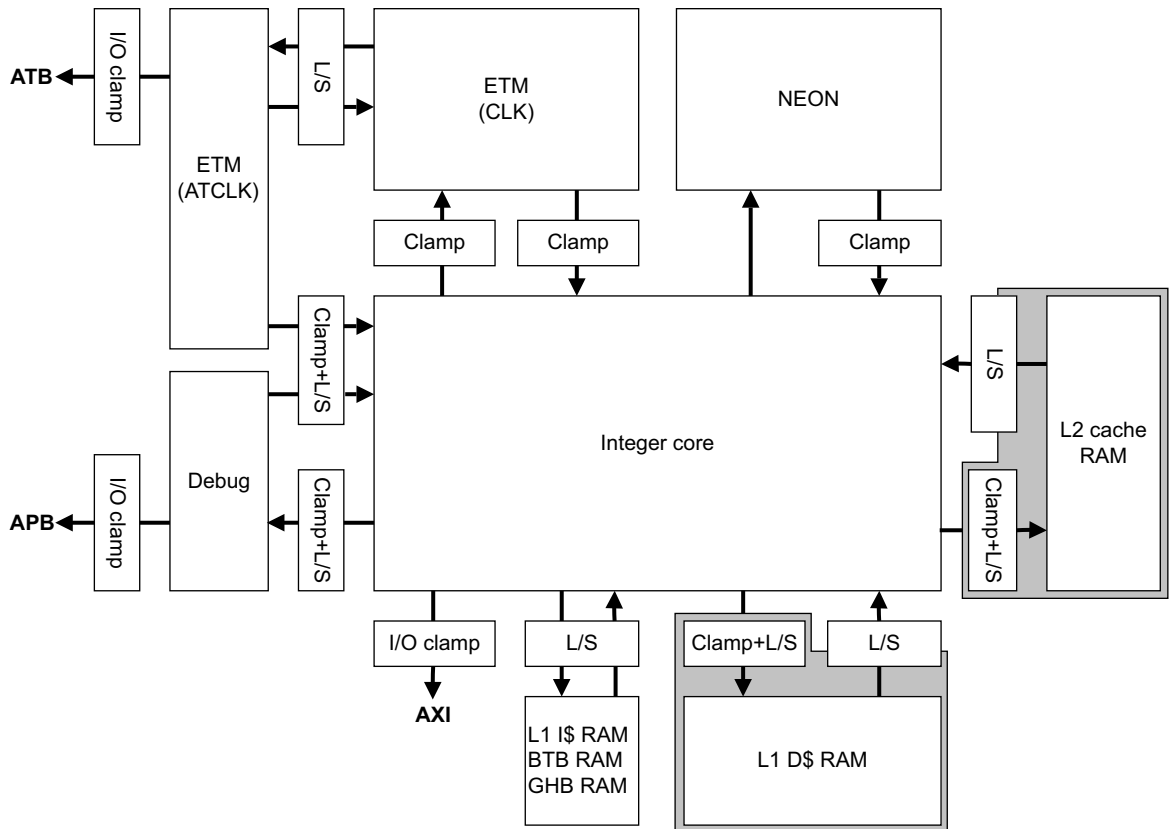
1. Clean and invalidate the caches, L1 data and L2 caches, to the point of coherency.
2. Disable the L1 data and L2 cache.
3. Save off any TLB state such as locked entries, if required.
4. Save off architectural state.

5. Reset and power down the processor. See *Powering down the integer core power domain* on page 10-24.
6. Power up the processor. See *Powering up the integer core and NEON power domains* on page 10-24.
7. Perform a normal software reset sequence.

The largest potential time and energy required in the sequence is the clean and invalidate of the caches. This operation is bounded by the time required to transfer the data into an external memory. To reduce or remove this Clean and Invalidate operation, the processor supports a separate power domain for the L1 data cache in addition to a separate power domain for the L2 cache RAMs. The L1 data cache and L2 unified cache contain hardware reset assistance that is controlled with the input pins **L1RSTDISABLE** and **L2RSTDISABLE**, respectively. The **L1RSTDISABLE** and **L2RSTDISABLE** pins must be tied LOW to enable hardware reset if the L1 data cache and L2 cache contents are not retained during core power down. Conversely, if the L1 data cache contents and the L2 cache contents are retained by a separate powered-up domain, the **L1RSTDISABLE** and **L2RSTDISABLE** pins must be enabled to ensure updated data contained in the caches is not invalidated by the power-up reset sequence. See *Hardware RAM array reset* on page 10-8 for more information about the timing requirements of these pins.

If an implementation places the L1 or L2 cache on separate power domains as shown in Figure 10-13 on page 10-27, the rest of the processor can be powered down while the L1 or L2 cache retains their data. This requires that all inputs to the L1 or L2 RAMs such as tag, parity, valid, and data RAMs are clamped to safe values to avoid corrupting the data when entering or exiting a power-down state.

If the L1 data cache contents are placed on a separate power domain, then the L2 cache must also be placed on a separate power domain. The L1 data cache contents cannot be retained without retaining the L2 cache contents. An exception to this rule is the OKB L2 cache configuration.



L/S = Level Shift

**Figure 10-13 Retention power domains**

Similarly, the L1 data cache can be placed on a separate power domain from the rest of the processor. This L1 data cache power domain can be shared with the L2. However, sharing of the two cache power domains is not required. In addition, all inputs into the L1 data cache RAMs such as tag, HVAB, and data RAMs must be clamped to safe values to avoid corrupting the data when entering or exiting a power-down state.

**Note**

Data retention within the L1 instruction cache is not supported.

### Power cycle the core with L2 cache retaining state

A power down and reset sequence of the processor with the L2 cache retained is as follows:

1. Clean to the point of unification the L1 data cache.
2. Save off any TLB state such as locked entries, if required.
3. Save off architectural state, if required.
4. Assert **L2RSTDISABLE** to disable L2 hardware reset.
5. Reset and power down the processor. See *Powering down the integer core power domain* on page 10-24.
6. Power up the processor. See *Powering up the integer core and NEON power domains* on page 10-24.
7. Perform a normal software initialization of the L1 instruction and data caches.
8. Perform a software read of a memory location to determine that the L2 has valid data and to skip the L2 software invalidation.
9. Before enabling the L2 cache or using any CP15 cache-related operations, software must signal the system to release the L2 cache input clamps and receive confirmation that the clamps have been released.

### Power cycle the core with L1 data cache and L2 cache retaining state

A power down and reset sequence of the processor with the L1 data cache and L2 cache is as follows:

1. Save off any TLB state such as locked entries, if required.
2. Save off architectural state, if required.
3. Assert **L1RSTDISABLE** and **L2RSTDISABLE** to inhibit hardware reset of the L1 data cache and L2 cache.
4. Reset and power down the processor. See *Powering down the integer core power domain* on page 10-24.
5. Power up the processor. See *Powering up the integer core and NEON power domains* on page 10-24.
6. Perform a normal software initialization of the L1 instruction cache.



7. Perform a software read of a memory location to determine that the L1 data cache and L2 cache have valid data and to skip the software initialization sequence.
8. Before enabling the L1 data cache or L2 cache, or using any CP15 cache-related operations, software must signal the system to release the L1 data cache and L2 cache input clamps and receive confirmation that the clamps have been released.

———— **Note** —————

The details of how to clamp the inputs to various arrays are implementation-specific and are not described in this document. Care must be taken that **nPORESET** does not affect the state in the RAM arrays.

---

### 10.3.5 Special note on reset during power transition

During any transition of the power supply to a component of the processor, the asynchronous reset to that component must be asserted. This is a safety mechanism for implementation to ensure that hardware can be protected against supply transition, DC paths, such as precharge or discharge circuits, or bus contention. The primary inputs to the processor that act as asynchronous resets are:

- **ATRESETn**
- **PRESETn**
- **nPORESET**.

If an implementation retains state in the L1 data cache or L2 cache as described, care must be taken that reset, particularly **nPORESET**, does not corrupt the state of the RAM arrays when lowering or raising the power supply to the rest of the processor. You can achieve this by clamping the primary I/O to the RAM arrays or designing the RAM arrays in such a way that they do not require a reset. If a reset is required, hardware must ensure that reset is inactive to those RAMs while clamped.



# Chapter 11

## Design for Test

This chapter describes the DFT features that are included in the *Register Transfer Language* (RTL). It contains the following sections:

- *MBIST* on page 11-2
- *ATPG test features* on page 11-37.

## 11.1 MBIST

This section describes the array architecture and operation of the MBIST:

- *About MBIST*
- *MBIST registers* on page 11-3
- *MBIST operation* on page 11-18
- *Pattern selection* on page 11-24.

### 11.1.1 About MBIST

The processor has three separate MBIST controllers:

#### L1 and L2 MBIST controllers

The L1 and L2 MBIST controllers communicate with RAM arrays distributed around the chip. Their controls are directly ported to the interface for use with external testbench or *Automated Test Equipment* (ATE) drivers.

#### CAMBIST controller

The CAMBIST controller is a slave of the L1 MBIST controller. It targets the comparator logic of the *Content-Addressable Memory* (CAM). The L1 MBIST controller tests the contents of the I-CAM and D-CAM arrays.

The following arrays require MBIST support:

- *Instruction cache* (I-cache)
- *Data cache* (D-cache)
- *Global History Buffer* (GHB)
- *Branch Target Buffer* (BTB)
- *Translation Look-aside Buffer* (TLB)

———— **Note** —————

The TLB has separate instruction and data arrays, each containing an attribute array, a CAM array, and a *Physical Address* (PA) array.

- *Hash Virtual Address Buffer* (HVAB)
- L1 tag RAM
- all L2 cache RAM such as data, parity, tag, and valid RAMs.

## 11.1.2 MBIST registers

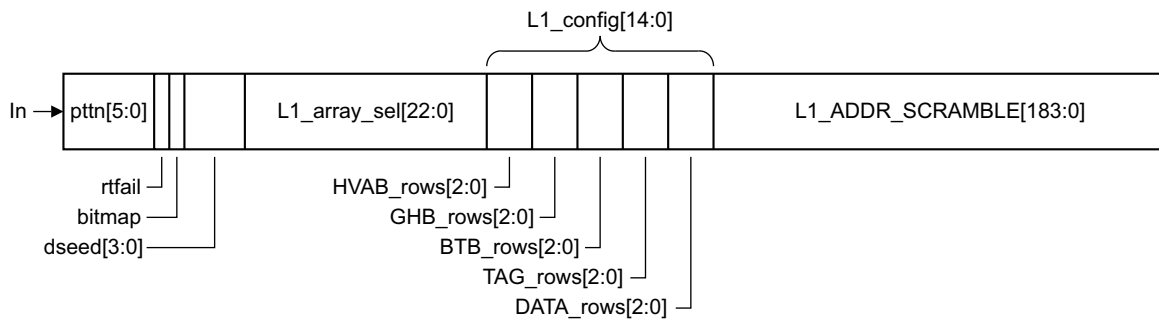
Table 11-1 shows the MBIST registers. See Figure 11-7 on page 11-20 for information about the timing of an MBIST instruction load.

**Table 11-1 MBIST register summary**

Register	Access	Reference
L1 MBIST Instruction Register	W	See <i>L1 MBIST Instruction Register</i>
L2 MBIST Instruction Register	W	See <i>L2 MBIST Instruction Register</i> on page 11-7
L1 and L2 MBIST GO-NOGO Instruction Registers	W	See <i>L1 and L2 MBIST GO-NOGO Instruction Registers</i> on page 11-13
L1 MBIST Datalog Register	R	See <i>L1 MBIST Datalog Register</i> on page 11-14
L2 MBIST Datalog Register	R	See <i>L2 MBIST Datalog Register</i> on page 11-15

### L1 MBIST Instruction Register

Figure 11-1 shows the fields of the L1 MBIST Instruction Register.



**Figure 11-1 L1 MBIST Instruction Register bit assignments**

***pttn[5:0]***

Use the `pttn[5:0]` field to select test patterns as Table 11-2 shows.

**Table 11-2 Selecting a test pattern with `pttn[5:0]`**

<b>Field</b>	<b>Selected test pattern</b>																										
<code>pttn[5:0]</code>	Pattern select field: <sup>a</sup>																										
	<table> <tbody> <tr> <td><code>b010010 = PTTN_WRITE_SOLID</code></td> <td><code>b101010 = PTTN_READBANG</code></td> </tr> <tr> <td><code>b010011 = PTTN_READ_SOLID</code></td> <td><code>b001011 = PTTN_YMARCH_C</code></td> </tr> <tr> <td><code>b100001 = PTTN_SOLIDS</code></td> <td><code>b001100 = PTTN_WRITE_ROWBAR</code></td> </tr> <tr> <td><code>b000010 = PTTN_WRITE_CKBD</code></td> <td><code>b001101 = PTTN_READ_ROWBAR</code></td> </tr> <tr> <td><code>b000011 = PTTN_READ_CKBD</code></td> <td><code>b101101 = PTTN_ROWBAR</code></td> </tr> <tr> <td><code>b100011 = PTTN_CKBD</code></td> <td><code>b001110 = PTTN_WRITE_COLBAR</code></td> </tr> <tr> <td><code>b000100 = PTTN_XMARCH_C</code></td> <td><code>b001111 = PTTN_READ_COLBAR</code></td> </tr> <tr> <td><code>b000101 = PTTN_PTTN_FAIL</code></td> <td><code>b101111 = PTTN_COLBAR</code></td> </tr> <tr> <td><code>b000110 = PTTN_RW_XMARCH</code></td> <td><code>b010000 = PTTN_RW_XADDRBAR</code></td> </tr> <tr> <td><code>b000111 = PTTN_RW_YMARCH</code></td> <td><code>b010001 = PTTN_RW_YADDRBAR</code></td> </tr> <tr> <td><code>b001000 = PTTN_RWR_XMARCH</code></td> <td><code>b010100 = PTTN_ADDR_DEC</code></td> </tr> <tr> <td><code>b001001 = PTTN_RWR_YMARCH</code></td> <td><code>b000000 = PTTN_GONOGO<sup>b</sup></code></td> </tr> <tr> <td><code>b001010 = PTTN_WRITEBANG</code></td> <td><code>b111111 = PTTN_SLAVE</code></td> </tr> </tbody> </table>	<code>b010010 = PTTN_WRITE_SOLID</code>	<code>b101010 = PTTN_READBANG</code>	<code>b010011 = PTTN_READ_SOLID</code>	<code>b001011 = PTTN_YMARCH_C</code>	<code>b100001 = PTTN_SOLIDS</code>	<code>b001100 = PTTN_WRITE_ROWBAR</code>	<code>b000010 = PTTN_WRITE_CKBD</code>	<code>b001101 = PTTN_READ_ROWBAR</code>	<code>b000011 = PTTN_READ_CKBD</code>	<code>b101101 = PTTN_ROWBAR</code>	<code>b100011 = PTTN_CKBD</code>	<code>b001110 = PTTN_WRITE_COLBAR</code>	<code>b000100 = PTTN_XMARCH_C</code>	<code>b001111 = PTTN_READ_COLBAR</code>	<code>b000101 = PTTN_PTTN_FAIL</code>	<code>b101111 = PTTN_COLBAR</code>	<code>b000110 = PTTN_RW_XMARCH</code>	<code>b010000 = PTTN_RW_XADDRBAR</code>	<code>b000111 = PTTN_RW_YMARCH</code>	<code>b010001 = PTTN_RW_YADDRBAR</code>	<code>b001000 = PTTN_RWR_XMARCH</code>	<code>b010100 = PTTN_ADDR_DEC</code>	<code>b001001 = PTTN_RWR_YMARCH</code>	<code>b000000 = PTTN_GONOGO<sup>b</sup></code>	<code>b001010 = PTTN_WRITEBANG</code>	<code>b111111 = PTTN_SLAVE</code>
<code>b010010 = PTTN_WRITE_SOLID</code>	<code>b101010 = PTTN_READBANG</code>																										
<code>b010011 = PTTN_READ_SOLID</code>	<code>b001011 = PTTN_YMARCH_C</code>																										
<code>b100001 = PTTN_SOLIDS</code>	<code>b001100 = PTTN_WRITE_ROWBAR</code>																										
<code>b000010 = PTTN_WRITE_CKBD</code>	<code>b001101 = PTTN_READ_ROWBAR</code>																										
<code>b000011 = PTTN_READ_CKBD</code>	<code>b101101 = PTTN_ROWBAR</code>																										
<code>b100011 = PTTN_CKBD</code>	<code>b001110 = PTTN_WRITE_COLBAR</code>																										
<code>b000100 = PTTN_XMARCH_C</code>	<code>b001111 = PTTN_READ_COLBAR</code>																										
<code>b000101 = PTTN_PTTN_FAIL</code>	<code>b101111 = PTTN_COLBAR</code>																										
<code>b000110 = PTTN_RW_XMARCH</code>	<code>b010000 = PTTN_RW_XADDRBAR</code>																										
<code>b000111 = PTTN_RW_YMARCH</code>	<code>b010001 = PTTN_RW_YADDRBAR</code>																										
<code>b001000 = PTTN_RWR_XMARCH</code>	<code>b010100 = PTTN_ADDR_DEC</code>																										
<code>b001001 = PTTN_RWR_YMARCH</code>	<code>b000000 = PTTN_GONOGO<sup>b</sup></code>																										
<code>b001010 = PTTN_WRITEBANG</code>	<code>b111111 = PTTN_SLAVE</code>																										

- a. See *Pattern selection* on page 11-24.  
 b. Default value of `pttn[5:0]`.

**Note**

The `PTTN_SLAVE` pattern (`b111111`) is for testing only the I-CAMBIST and D-CAMBIST.

***rtfail***

Setting the `rtfail` bit to 1 enables the fail signal to assert on every cycle that a failure occurs. Clearing the `rtfail` bit to 0 causes a sticky failure reporting, and the fail signal remains asserted after the first failure that occurs. Reset clears the `rtfail` bit to 0.

***bitmap***

Setting the `bitmap` bit to 1 enables bitmap test mode. Reset clears the instruction register `bitmap` bit to 0. See *Bitmap test mode* on page 11-19.

***dseed[3:0]***

Write the data seed in the *dseed* field. The MBIST controller repeats the *dseed* data to the full array bus width. The reset value of *dseed[3:0]* is *b0000*.

***L1\_array\_sel[22:0]***

Set bits in the *L1\_array\_sel[22:0]* field to select the L1 arrays for test. The MBIST executes the selected arrays serially beginning with the array indicated by the LSB. Table 11-3 shows how each bit selects one of the L1 arrays. The reset value of the *L1\_array\_sel* is *23'h1FFFFFF*.

**Table 11-3 Selecting the L1 arrays to test with *L1\_array\_sel[22:0]***

Bit	Array selected
[0]	I-RAM word0 [31:0] parity and dirty included. <sup>a</sup>
[1]	I-RAM word1 [63:32] parity and dirty included. <sup>a</sup>
[2]	I-RAM word2 [95:64] parity and dirty included. <sup>a</sup>
[3]	I-RAM word3 [127:96] parity and dirty included. <sup>a</sup>
[4]	I-CAM array.
[5]	I-PA.
[6]	I-tag.
[7]	I-attributes of TLB.
[8]	I-HVAB.
[9]	BTBI. <sup>a</sup>
[10]	BTBH. <sup>a</sup>
[11]	GHB.
[12]	D-RAM word0 [31:0] parity and dirty included. <sup>a</sup>
[13]	D-RAM word1 [63:32] parity and dirty included. <sup>a</sup>
[14]	D-RAM word2 [95:64] parity and dirty included. <sup>a</sup>
[15]	D-RAM word3 [127:96] parity and dirty included. <sup>a</sup>
[16]	D-CAM array.
[17]	D-PA.

**Table 11-3 Selecting the L1 arrays to test with L1\_array\_sel[22:0] (continued)**

Bit	Array selected
[18]	D-tag.
[19]	D-attributes of TLB.
[20]	D-HVAB.
[21]	I-CAMBIST. Tests CAM compare logic.
[22]	D-CAMBIST. Tests CAM compare logic.

- a. You can test the RAM and BTB arrays by accessing the entire array width during writes. The selected words are compared according to the L1\_array\_sel bit currently under test. For this reason, exercise care when creating iddq or data retention patterns because individual word slices cannot be initialized and maintained with different data seeds.

**Note**

Do not test the CAMBIST arrays in the same run as other arrays.

**L1\_config[14:0]**

The L1\_config[14:0] field contains five 3-bit fields for defining the number of physical rows in each L1 array as Table 11-4 shows.

**Table 11-4 L1\_config[14:0]**

L1_config bit field	Field name
L1_config[14:12]	HVAB_rows[2:0]
L1_config[11:9]	GHB_rows[2:0]
L1_config[8:6]	BTB_rows[2:0]
L1_config[5:3]	TAG_rows[2:0]
L1_config[2:0]	DATA_rows[2:0]

The value in each field is implementation-defined and programmable to ensure that physically targeted RAM tests perform correctly.



**Note**

Only arrays with variable row sizes are programmable. The CAM, PA, and attributes arrays have an architecturally fixed depth of 32. Because of timing limits, physical rows beyond 512 are not supported.

Table 11-5 shows the possible values for each of the 4-bit fields of L1\_config[14:0].

**Table 11-5 Configuring the number of L1 array rows with L1\_config[14:0]**

Field value	Number of rows sharing a bitline pair
b000	16
b001	32
b010	64
b011	128
b100	256
b101	512
b110-b111	Reserved

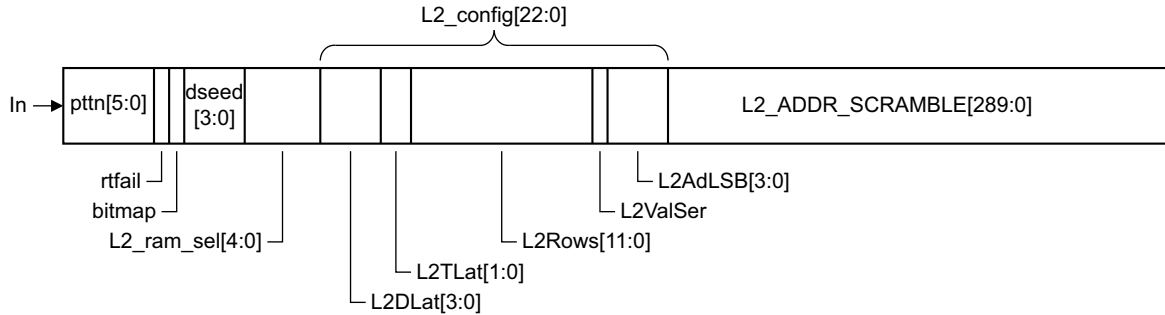
Reset clears the HVAB\_rows[2:0], GHB\_rows[2:0], and TAG\_rows[2:0] fields to b000. Reset initializes the BTB\_rows[2:0] and DATA\_rows[2:0] fields to b010.

**L1\_ADDR\_SCRAMBLE[183:0]**

Proper physical mapping prevents unintended pattern sequences that result in loss of test quality. This field defines the physical-to-logical address scramble settings for your implementation. See the *Cortex-A8 Release Notes* for information on how to program this for your design.

**L2 MBIST Instruction Register**

Figure 11-2 on page 11-8 shows the fields of the L2 MBIST Instruction Register.



**Figure 11-2 L2 MBIST Instruction Register bit assignments**

The pttm[5:0], rtfail, bitmap, and dseed[3:0] fields function the same as in the L1 MBIST Instruction Register.

### ***L2\_ram\_sel[4:0]***

Set bits in the L2\_ram\_sel[4:0] field to select the L2 RAMs for test as Table 11-6 shows.

**Table 11-6 Selecting L2 RAMs for test with L2\_ram\_sel[4:0]**

Bit	Selected RAM
[0]	L2 data RAM low order bits [64:0]
[1]	L2 data RAM high order bits [129:65]
[2]	L2 parity RAM
[3]	L2 tag RAM
[4]	L2 valid RAM

Setting an L2\_ram\_sel bit selects the corresponding RAM for test.

The MBIST accesses the RAMs serially in the order shown in Table 11-6, except that the L2 tag RAM and L2 valid RAM are tested in parallel. You can set the L2ValSer bit to 1 to test these two RAMs serially. See *L2ValSer* on page 11-11.

The reset value of the L2\_ram\_sel[4:0] field is b11111.

### ***L2\_config[22:0]***

The L2\_config[22:0] field contains fields for selecting:

- read and write latency of the L2 data array
- read and write latency of the L2 tag array

- number of rows of the L2 data, parity, tag and valid physical RAM
- testing of valid RAM separately or in parallel with tag RAM testing
- column address LSB sequencing of 00, 01, 10, 11 or 00, 01, 11, 10.

Table 11-7 shows the bit fields of L2\_config[22:0].

**Table 11-7 L2\_config[22:0]**

<b>L2_config bit field</b>	<b>Field name</b>
L2_config[22:19]	L2DLat[3:0]
L2_config[18:17]	L2TLat[1:0]
L2_config[16:5]	L2Rows[11:0]
L2_config[4]	L2ValSer
L2_config[3:0]	L2AdLSB[3:0]

### **L2DLat[3:0]**

Use the L2DLat[3:0] field to select the read and write latency of the L2 data array as Table 11-8 shows. The reset value of the L2DLat[3:0] field is b1111.

**Table 11-8 Selecting L2 data array latency with L2DLat[3:0]**

<b>L2DLat[3:0]</b>	<b>Wait states</b>
b0000	2
b0001	2
b0010	3
b0011	4
b0100	5
b0101	6
b0110	7
b0111	8
b1000	9
b1001	10
b1010	11

**Table 11-8 Selecting L2 data array latency with L2DLat[3:0] (continued)**

<b>L2DLat[3:0]</b>	<b>Wait states</b>
b1011	12
b1100	13
b1101	14
b1110	15
b1111	16

**L2TLat[1:0]**

Use the L2TLat[1:0] field to select the read and write latency of the L2 tag array as Table 11-9 shows. Reset sets the L2TLat[1:0] field, selecting four wait states.

**Table 11-9 Selecting L2 tag array latency with L2TLat[1:0]**

<b>L2TLat[1:0]</b>	<b>Wait states</b>
b00	2
b01	2
b10	3
b11	4

**L2Rows[11:0]**

The four 3-bit fields in the L2Rows[11:0] field control the number of rows in the data, parity, tag, and valid RAMs. Table 11-10 shows the fields that control each of the four RAMs.

**Table 11-10 Selecting the L2 RAMs with L2Rows[11:0]**

<b>Bit range</b>	<b>Reset value</b>	<b>Function</b>
[11:9]	b100	Selects number of data RAM rows
[8:6]	b100	Selects number of parity RAM rows
[5:3]	b000	Selects number of tag RAM rows
[2:0]	b000	Selects number of valid RAM rows

Table 11-11 shows how to configure array depth with the L2Rows fields.

**Table 11-11 Configuring the number of L2 RAM rows with L2Rows[11:0]**

Field value	Number of rows sharing a bitline pair
b000	16
b001	32
b010	64
b011	128
b100	256
b101	512
b110-b111	Reserved

Not all row settings are valid for all RAMs in all L2 cache size configurations. Table 11-12 shows the range of values from Table 11-11, that is possible for each RAM type, and for each cache size.

**Table 11-12 Valid L2 array row numbers**

Cache size (KB)	Data/parity RAM (rows)	Tag/valid RAM (rows)
	Valid range	
128	32-512	16-128
256	32-512	16-128
512	32-512	16-128
1024	64-512	16-256

### **L2ValSer**

By default, the MBIST tests the L2 tag RAM and L2 valid RAM at the same time. Table 11-13 on page 11-12 shows that you can select serial testing of the tag and valid RAMs by setting the L2ValSer bit to 1. The reset value of the L2ValSer bit is 0.

When L2ValSer is 0, that is, parallel testing is selected, the address scramble configuration for the valid RAM is the same as that of the tag RAM. This means that the valid RAM uses the tag RAM address scramble configuration, even if the tag RAM is not selected for test. The L2ValSer bit is provided to enable you to serially test the tag RAM with different address scramble configurations.

**Table 11-13 Selecting the L2ValSer test type**

L2ValSer	Testing of L2 tag RAM and L2 valid RAM
1	Serial testing
0	Parallel testing

**L2AdLSB[3:0]**

Use the L2AdLSB[3:0] field to select how to increment or decrement the two LSBs of the column address of L2 valid, tag, parity and data RAM accesses. This field is provided as a way to configure non-linear address sequences found in some compiled RAMs. Table 11-14 shows the L2 array controlled by each L2AdLSB[3:0] bit.

**Table 11-14 Selecting L2 RAMs for LSB control**

L2AdLSB[3:0] bit	Selected RAM
[0]	L2 valid RAM
[1]	L2 tag RAM
[2]	L2 parity RAM
[3]	L2 data RAM

Table 11-15 shows how each L2AdLSB[3:0] bit controls the increment and decrement sequence of the two column address LSBs.

**Table 11-15 Selecting counting sequence of L2 RAM column address LSBs**

L2AdLSB[n]	LSB increment sequence	LSB decrement sequence
[0]	00, 01, 10, 11	11, 10, 01, 00
[1]	00, 01, 11, 10	10, 11, 01, 00

The reset value of the L2AdLSB[3:0] field is b0000.

**L2\_ADDR\_SCRAMBLE[289:0]**

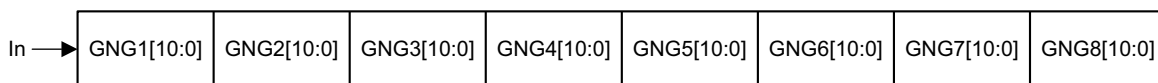
Proper physical mapping prevents unintended pattern sequences that result in loss of test quality. Use the ADDR\_SCRAMBLE[289:0] field to define the physical-to-logical address scramble setting for your implementation. See the *Cortex-A8 Release Notes* for information on how to program this for your design.

**L1 and L2 MBIST GO-NOGO Instruction Registers**

You can use the L1 and L2 MBIST GO-NOGO Instruction Registers to program a custom sequence of up to eight patterns for either L1 or L2 memory. Figure 11-3 shows the fields of the L1 and L2 MBIST GO-NOGO Instruction Registers.

**Note**

*GO-NOGO* on page 11-36 describes the default GO-NOGO sequence available at power-up.

**Figure 11-3 L1 and L2 MBIST GO-NOGO Instruction Registers bit assignments**

Each GNG[10:0] field is a concatenation of three fields as Table 11-16 shows.

**Table 11-16 GNG[10:0] field**

GNG[10:0] bit field	Field name
GNG[10]	Valid
GNG[9:6]	data seed[3:0]
GNG[5:0]	pattern selection[5:0]

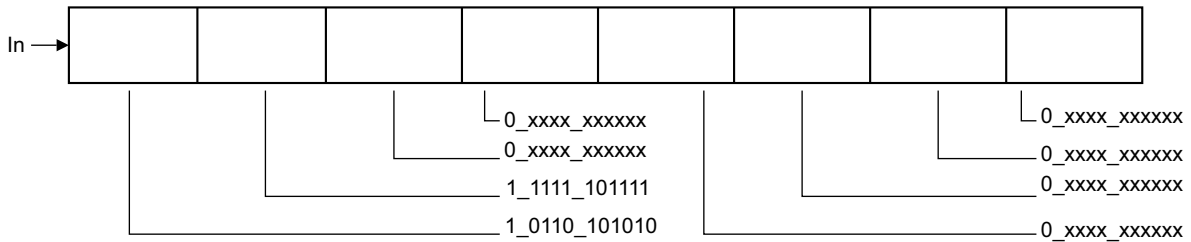
The patterns execute in order, starting with GNG1. It is not necessary to load the entire register when fewer than eight patterns are required. If you load fewer than eight patterns, the unloaded fields cannot execute because their valid bits are cleared to 0 at reset.

For example, to execute a READBANG with a data seed of 0x6 followed by a COLBAR with a data seed of 0xF, you only have to load two fields:

1\_0110\_101010 → 1\_1111\_10111 = GNG1[10:0] → GNG2[10:0]

See *READBANG* on page 11-35 and *COLBAR* on page 11-27 for more details.

Figure 11-4 shows the L1 MBIST GO-NOGO Instruction Register contents after loading a COLBAR with a data seed of 0xF and a READBANG with a data seed of 0x6.

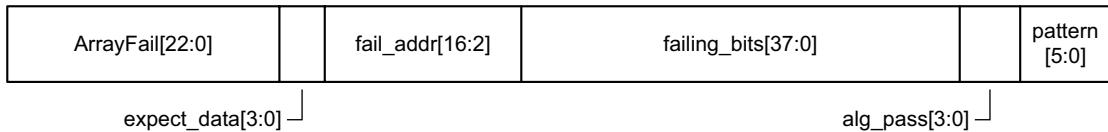


**Figure 11-4 L1 MBIST GO-NOGO Instruction Register example with two patterns**

The **MBISTSHIFTL1** signal must toggle one cycle before initiation and one cycle before completion of the **MBISTDATAINL1** stream as Figure 11-7 on page 11-20 shows. During GO-NOGO instruction load, **MBISTDSHIFTL1** must toggle at the same time as **MBISTSHIFTL1**. See Figure 11-8 on page 11-21.

### L1 MBIST Datalog Register

The L1 MBIST Datalog Register records information about failing arrays. Figure 11-5 shows the fields of the L1 MBIST Datalog Register.



**Figure 11-5 L1 MBIST Datalog Register bit assignments**

#### ***ArrayFail[22:0]***

Read the ArrayFail[22:0] field to identify arrays that produce failures. The bits in this field correspond to the bits in the L1\_array\_sel[22:0] field in the L1 MBIST Instruction Register. Table 11-5 on page 11-7 shows how each bit corresponds to one of the L1 arrays. Testing more than one array while not in bitmap test mode can set more than one ArrayFail[22:0] bit to 1. The least-significant 1 in the ArrayFail[22:0] field indicates the first failing array.

#### ***expect\_data[3:0]***

Read the expect\_data[3:0] field for the expected data seed for the first failing array. Because data seed toggling occurs throughout pattern execution, the value in this field does not always correspond to the programmed data seed.



**fail\_addr[16:2]**

Read the fail\_addr[16:2] field for the physical address of the first failing array. See the address scramble information contained within the Design for Test implementation documentation for details on shows how this address is constructed.

**failing\_bits[37:0]**

Read the failing\_bits[37:0] field to identify failing bits in the first array that fails. This field contains the EXCLUSIVE-OR of read data and expect data.

**alg\_pass[3:0]**

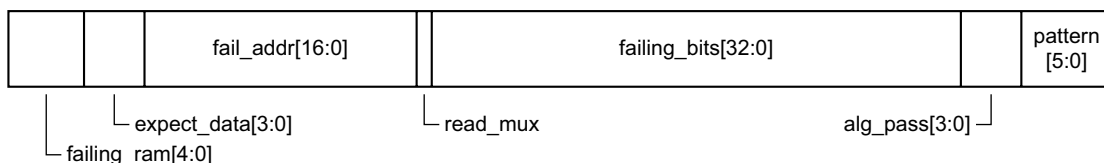
For the first failing array, read the alg\_pass[3:0] field to identify which pass of the algorithm produced a failure. For example, the CKBD algorithm has four passes, wscan, rscan, wscan, and rscan, numbered 1, 2, 3, and 4. Because failures only occur on reads, a CKBD failure results in an alg\_pass[3:0] value of b0010 or b0100.

**pattern[5:0]**

Read the pattern[5:0] field to identify the pattern running at the time of the first failure. Table 11-2 on page 11-4 shows the pattern codes. This field is useful when running more than one pattern during a GO-NOGO test.

**L2 MBIST Datalog Register**

Figure 11-6 shows the fields of the L2 MBIST Datalog Register.



**Figure 11-6 L2 MBIST Datalog Register bit assignments**

**failing\_ram[4:0]**

Read the failing\_ram[4:0] field to identify the RAMs that produce failures. The bits in this field correspond to the bits in the L2\_ram\_sel[4:0] field in the L2 MBIST Instruction Register. Table 11-6 on page 11-8 shows how each bit corresponds to one of the L2 RAMs. Testing more than one RAM while not in bitmap test mode can set more than one failing\_ram[4:0] bit to 1. The least-significant bit that is set to 1 in the failing\_ram[4:0] field indicates the first failing RAM.

---

**Note**

---

When the L2ValSer bit is 0, the tag RAM and valid RAM are tested in parallel. When testing both these RAM in parallel, a failure in either RAM sets both bit [3] and bit [4] in the failing\_ram[4:0] field to 1. To determine if the tag RAM, valid RAM, or both failed, process the failing\_bits[32:0] field, see Table 11-18 on page 11-17.

---

**expect\_data[3:0]**

Read the expect\_data[3:0] field for the expected data seed for the first failing RAM. Because data seed toggling occurs throughout algorithm execution, the value in this field does not always correspond to the programmed data seed.

**fail\_addr[16:0]**

Read the fail\_addr[16:0] field for the physical address of the first RAM failure. This is the address sent to the RAM through the L2 MBIST interface. See the *Cortex-A8 Release Notes* for information on how you can construct this address.

When testing the data array, there are no cache way select bits, but the index value is still right-justified with fail\_addr[0]. You can ignore the upper bits of this field that might be unused for smaller cache sizes (except for bit [16], which is always zero). The values shifted out of unused address bits reflect the values assigned to those bits in the address scramble configuration.

When testing the tag array, bits [16:15] of this field contain the cache way select bits, and the tag array index value is the least-significant bits of fail\_addr. Because the fail\_addr[14:11] bits are not used for the tag array, they are always zero. Similar to the data array, the upper bits of the array index value are not used for lower cache sizes and can be ignored. Values to these upper bits are supplied by the address scramble configuration.

Table 11-17 shows how the cache ways are grouped into two ways of read data sent back from the tag RAMs.

**Table 11-17 L2 cache way grouping**

Test sequence number <sup>a</sup>	Cache way grouping in read data
0	way 1, way 0
1	way 3, way 2
2	way 5, way 4
3	way 7, way 6

- a. Test sequence number is the order that the MBIST controller accesses the cache ways.

The lower-numbered cache ways are always assigned to bits [22:0] of the read data bus for the current test group. The valid RAM contains two data bits for each of the eight cache ways for a total of 16 bits. To achieve a high test quality, all 16 bits are tested in parallel when testing the first group of cache ways. Because the valid bits are typically implemented as a single 16-bit RAM, testing all cache ways in parallel enables the full 16 bits to be accessed each time instead of testing it in slices. This provides greater flexibility with data backgrounds and can reduce test time if the valid RAM is tested serially after the tag RAM.

When testing tag RAMs and valid RAMs in parallel, the valid RAM chip select is disabled to prevent the valid RAM from being accessed during testing of subsequent groups of cache ways within the tag array.

#### ***read\_mux***

The read\_mux bit indicates which half of the 65-bit read produced the first failure:

- 0**            Indicates failure in bits [31:0].
- 1**            Indicates failure in bits [64:32].

#### ***failing\_bits[32:0]***

Read the failing\_bits[32:0] field to identify failing bits in the first RAM that fails. This field contains the EXCLUSIVE-OR of read data and expect data. Table 11-18 shows how to identify failing L2 bits.

**Table 11-18 Identifying failing L2 bits with failing\_bits[32:0]**

<b>failing_ram[4:0]</b>	<b>read_mux</b>	<b>failing_bits[32:0]</b>	
		<b>Valid bits<sup>a</sup></b>	<b>Value</b>
Data, low order	0	[31:0]	Data RAM bits [31:0]
Data, low order	1	[32:0]	Data RAM bits [64:32]
Data, high order	0	[31:0]	Data RAM bits [96:65]
Data, high order	1	[32:0]	Data RAM bits [129:97]

Table 11-18 Identifying failing L2 bits with failing\_bits[32:0] (continued)

failing_ram[4:0]	read_mux	failing_bits[32:0]	
		Valid bits <sup>a</sup>	Value
Parity	0 <sup>b</sup>	[15:0]	Parity RAM bits [15:0]
Tag/valid RAMs	0	[31:0]	Tag RAM read bits [15:0] <sup>c</sup> , valid RAM read bits [15:0]
Tag/valid RAMs	1	[29:0]	Tag RAM read bits [45:16] <sup>b</sup>

- Unused bits are RAZ.
- The read\_mux value for the parity RAM is always 0 because it is only 16 bit-wide and is always stored in the lower half of the 65-bit read bus.
- Not all tag RAM read bits are active. The MBIST controller masks any unused bits and does not generate a failure.

### ***alg\_pass[3:0]***

Read the alg\_pass[3:0] field to identify which pass of the algorithm produced a failure in the first failing RAM. A pass is defined as one complete pass through the entire address space of the RAM under test. The numbering starts with b0001, indicating the first pass.

### ***pattern[5:0]***

Read the pattern[5:0] field to identify the pattern running at the time of the first failure. Table 11-2 on page 11-4 shows the pattern codes. This field is useful in GO-NOGO testing when more than one pattern is run during the test.

## 11.1.3 MBIST operation

There are two MBIST modes:

- *Manufacturing test mode*
- *Bitmap test mode* on page 11-19.

### **Manufacturing test mode**

Manufacturing test mode determines the pass or fail status of the arrays. If the failure flag is set to 1 when the complete flag is set to 1, you can retrieve the datalog to identify the failing arrays. After analyzing the datalog, you can use bitmap test mode to identify the failing bits.

In manufacturing test mode, an MBIST test consists of the following steps:

1. Assert **MBISTMODE** for the entire test.
2. MBIST pipeline flush. Assert the system reset signal for at least 15 cycles.

3. Instruction load. Write to the MBIST Instruction Register.
4. Test execute. Wait for complete flag or fail flag.
5. Datalog retrieval. If the fail pin, **MBISTRESULT[1]**, is HIGH, read the MBIST Datalog Register.

### Bitmap test mode

In bitmap test mode, the MBIST controller stops when it detects a failure. It asserts **MBISTRESULT[1]** until the tester begins datalog retrieval. After datalog retrieval, the MBIST controller resumes the test from the point where it stopped. This handshake continues until test completion. The collected datalogs are useful for offline bitmap and redundancy analysis.

In bitmap test mode, an MBIST test consists of the following steps:

1. MBIST pipeline flush. Assert the system reset signal for at least 15 cycles.
2. Instruction load. Write to the MBIST Instruction Register.
3. Test execute.
  - a. If failures are detected, go to step 4.
  - b. If no failure is detected, go to step 5.
4. Datalog retrieval. Go to step 3 to continue.
5. End of test.

### MBIST Instruction load

Figure 11-7 on page 11-20 shows the timing of an MBIST instruction load. The **MBISTMODE** signal must remain asserted while the core is under reset. See Figure 10-6 on page 10-6 for more information on reset timing. **MBISTSHIFT** is asserted and instruction load data is serially loaded into the Instruction Register through the **MBISTDATAIN** pin. **MBISTSHIFT** is deasserted on completion of the instruction load. **MBISTDATAIN** has one cycle of latency in relation to **MBISTSHIFT**.

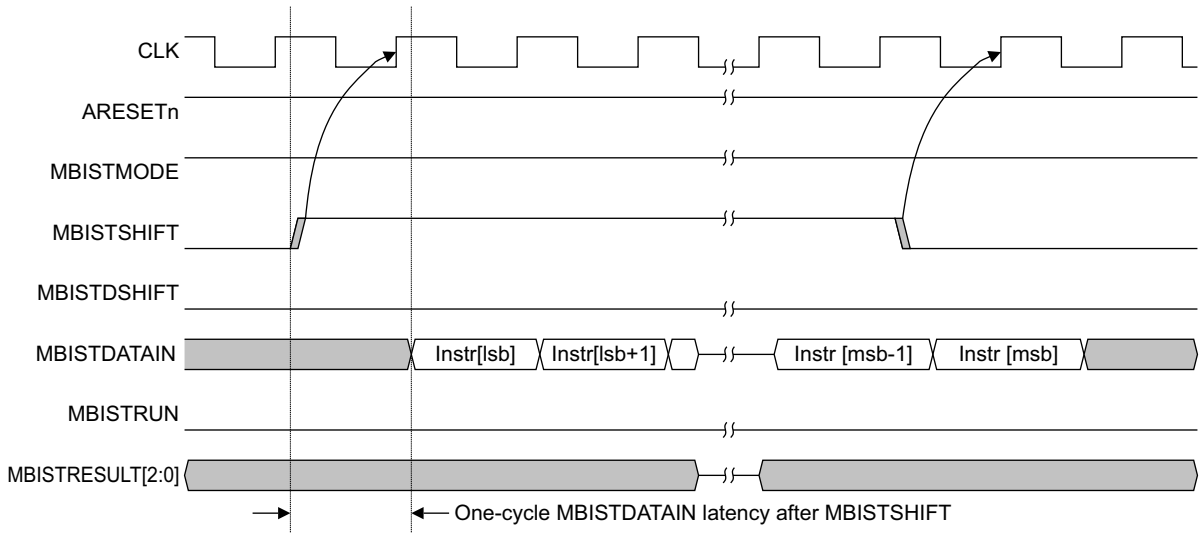
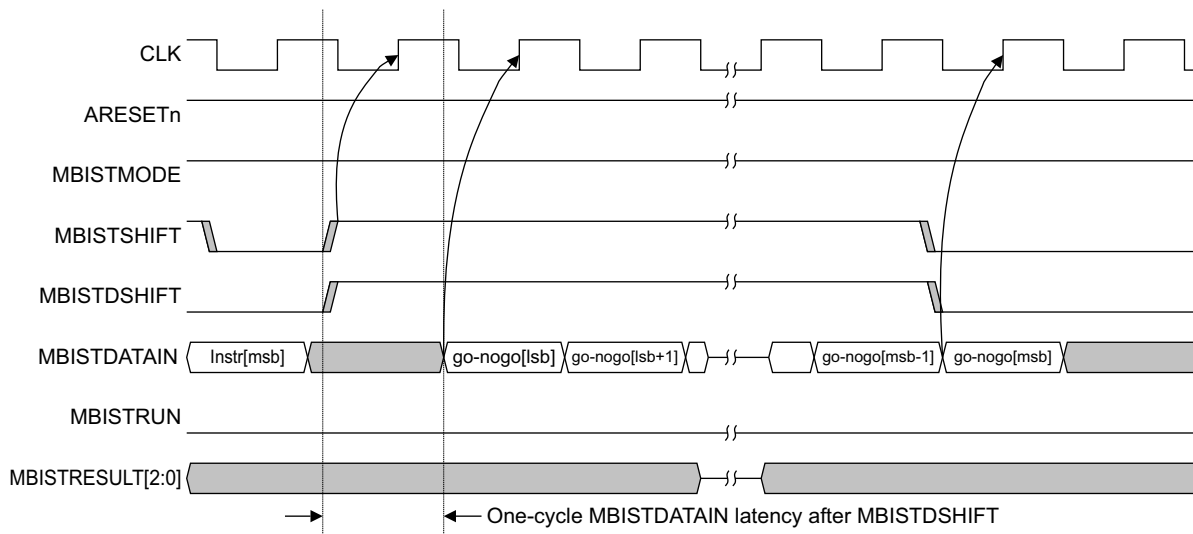


Figure 11-7 Timing of MBIST instruction load

### MBIST custom GO-NOGO instruction load

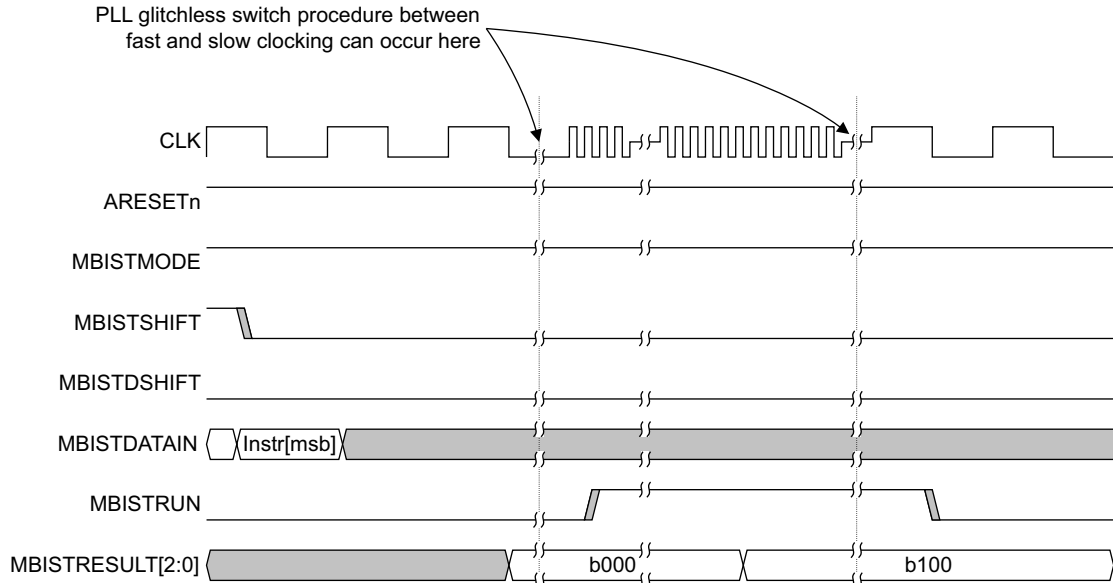
Figure 11-8 on page 11-21 shows an example of an MBIST instruction load followed immediately by a GO-NOGO instruction load. During the GO-NOGO portion of the load, **MBISTDSHIFT** and **MBISTSHIFT** both equal 1.



**Figure 11-8 Timing of MBIST custom GO-NOGO instruction load**

### Test execution

Figure 11-9 on page 11-22 shows an example of normal MBIST test execution. The complete flag, **MBISTRESULT[2]**, is asserted at the end of the test. This indicates a pass result in the absence of **MBISTRESULT[1]**, the fail flag. While bitmap mode is enabled, the test execution is interrupted when a failure occurs to shift out the fail data. Figure 11-9 on page 11-22 shows the timing wave forms for at speed test of the core. Slow clocking is implemented during the shifting of the Instruction Register and fast clocking is for the actual MBIST execution. It is assumed that slow clocking is required because of packaging pin timing restrictions.



**Figure 11-9 Timing of MBIST at-speed execution**

**Note**

During at-speed clocking with real-time fail mode active, you can ignore **MBISTRESULT[0]**.

**End-of-test datalog retrieval**

Figure 11-10 on page 11-23 shows an example of retrieval of the first failure datalog and the pass/fail status for every array. This is typically run at the end of testing. You can use bitmap test mode on the failing arrays.

Be careful not to miss a subsequent failure that might occur near the end of the testing sequence. For example, if a failure occurs on the last RAM access of the test sequence, then the complete flag asserts only three at-speed cycles after the fail flag asserts. If the fail flag signal goes through more external delay than the complete flag, the complete flag might be visible externally before the fail flag. Before classifying a test as passing, give adequate time after recognizing the complete flag to ensure that the fail flag does not assert.



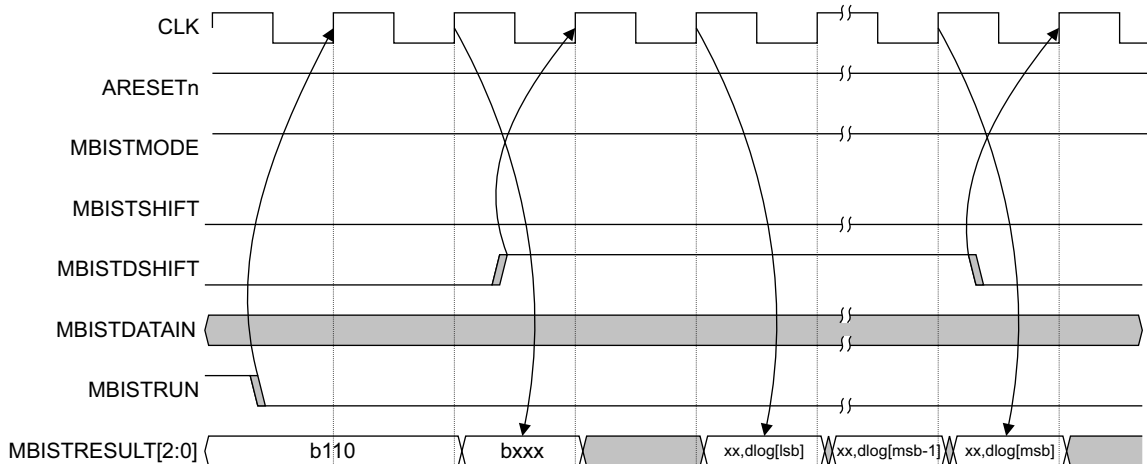


Figure 11-10 Timing of MBIST end-of-test datalog retrieval

### Bitmap datalog retrieval

Figure 11-11 shows an example of the start of a failure datalog retrieval during bitmap mode. The fail flag remains asserted and no more MBIST testing occurs until the **MBISTDSHIFT** signal is asserted, which initiates the serial shift-out of the bitmap datalog. This provides time to switch from fast to slow clocking required for shifting.

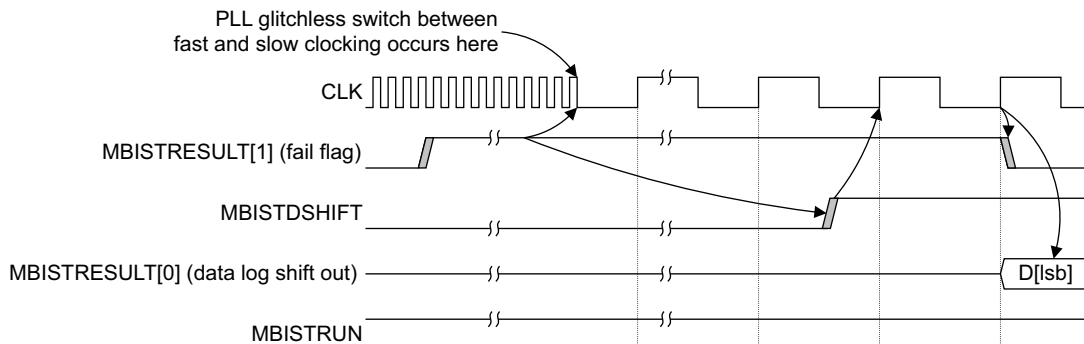
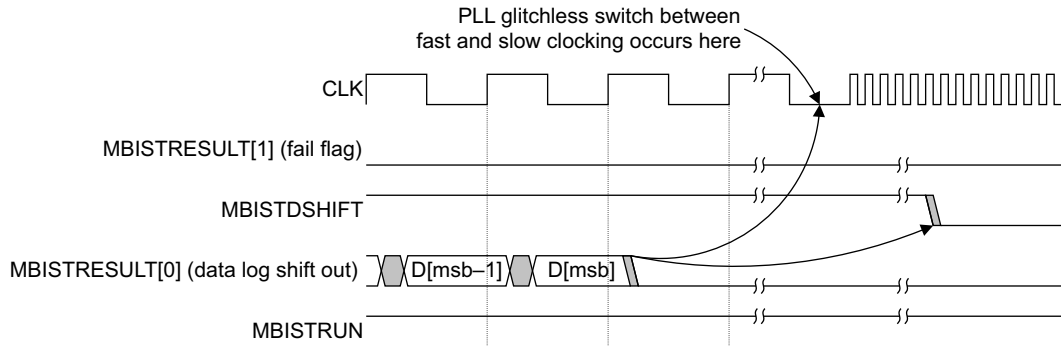


Figure 11-11 Timing of MBIST start of bitmap datalog retrieval

Figure 11-12 on page 11-24 shows an example of the end of a failure datalog retrieval during the execution of a failure bitmap. When all of the bits are shifted out, the PLL switches back to fast clocking and negates the **MBISTDSHIFT** signal. This causes the MBIST controller to resume testing.



**Figure 11-12** Timing of MBIST end of bitmap datalog retrieval

#### 11.1.4 Pattern selection

The processor implementation includes a toolbox of patterns for testing the arrays. When creating test vectors, you can select the group of algorithms that is most effective for your fabrication process.

Some of the pattern sequence descriptions use the following terms:

<b>R</b>	Read instruction data seed.
<b>W</b>	Write instruction data seed.
<b>R_</b>	Read inverse of instruction data seed.
<b>W_</b>	Write inverse of instruction data seed.
<b>incr</b>	Increment address starting with 0 until address = addrmax.
<b>decr</b>	Decrement address starting with addrmax until address = 0.
<b>wscan</b>	Write entire array.
<b>rscan</b>	Read entire array.
<b>N</b>	Total number of accesses per address location.

Table 11-19 shows the patterns that are selected using the pttm[5:0] field of the MBIST Instruction Register.

**Table 11-19 Summary of MBIST patterns**

<b>Pattern</b>	<b>N</b>	<b>Address updating</b>	<b>Description</b>
CAMBIST	-	-	Tests CAM compare logic, see <i>CAMBIST</i> on page 11-26
CKBD	4N	Row-fast	Checkerboard-checkerboard_bar wscan-rscan pattern, see <i>CKBD</i> on page 11-27
COLBAR	4N	Column-fast	Column bar-stripe wscan-rscan pattern, see <i>COLBAR</i> on page 11-27
ROWBAR	4N	Row-fast	Row bar-stripe wscan-rscan pattern, see <i>ROWBAR</i> on page 11-28
SOLIDS	4N	Row-fast	Solid wscan-rscan pattern, see <i>SOLIDS</i> on page 11-29
RWXMARCH	6N	Row-fast	Standard R W_ increment-decrement march, see <i>RWXMARCH</i> on page 11-29
RWYMARCH	6N	Column-fast	Standard R W_ increment-decrement march, see <i>RWYMARCH</i> on page 11-30
RWRXMARCH	8N	Row-fast	Standard R W_ R_ increment-decrement march, see <i>RWRXMARCH</i> on page 11-30
RWRYMARCH	8N	Column-fast	Standard R W_ R_ increment-decrement march, see <i>RWRYMARCH</i> on page 11-31
XMARCHC	14N	Row-fast	Standard marchC, see <i>XMARCHC</i> on page 11-32 R W_ R_ incr, R_ W R_ incr, R W_ R_ decr R_ W R_ decr
YMARCHC	14N	Column-fast	Standard marchC, see <i>YMARCHC</i> on page 11-32 R W_ R_ incr, R_ W R_ incr, R W_ R_ decr R_ W R_ decr
XADDRBAR	4N	Row-fast	Wscan/rscan through opposite addresses, see <i>XADDRBAR</i> on page 11-33
YADDRBAR	4N	Column-fast	Wscan/rscan through opposite addresses, see <i>YADDRBAR</i> on page 11-34
WRITEBANG	20N	Row-fast	Custom bitline stress test, see <i>WRITEBANG</i> on page 11-34 W_ R_ (wsac 5) R_ W
READBANG	17N	Row-fast	Custom bitcell read stress test, see <i>READBANG</i> on page 11-35

Table 11-19 Summary of MBIST patterns (continued)

Pattern	N	Address updating	Description
FAIL	6N	Row-fast	R W march with built-in failures, see <i>FAIL</i> on page 11-35
ADDRDECODER	$N(1 + 2\log_2 N)$	NA	Detection of open decoder faults on address lines, see <i>ADDRESS DECODER</i> on page 11-36
Default GO-NOGO	32N	Mix	CKBD-RWRYMARCH-WRITEBANG, see <i>GO-NOGO</i> on page 11-36
WCKBD WCOLBAR WROWBAR WSOLIDS	1N	-	Single pass wscan for I <sub>DDQ</sub> and data retention style tests
RCKBD RCOLBAR RROWBAR RSOLIDS	1N	-	Single pass rscan for I <sub>DDQ</sub> and data retention style tests

## CAMBIST

The CAMBIST performs a simultaneous match check across all 32 entries by comparing each entry against an incoming compare value. This function is executed by performing a bitwise XOR of the inverted compare value and each individual CAM entry. If the XOR is true, a hit is determined. CAMBIST tests this compare function by testing that each CAM bit is capable of generating a hit and a miss for both 1 and 0.

CAMBIST performs the following sequence:

1. Write all entries with 0xA.
2. Write 0s to entry 0.
3. Compare 0s and check for hit with 0s.
4. Compare 0x00000001 and check for miss.
5. Write 0x00000001 into CAM entry.
6. Check for hit and miss with compare = 0x00000001 and 0s.
7. Left shift compare bit until every bit is checked for hit and miss, repeating steps 4-6 until all bits are tested.
8. Write 0xAs to tested entry.
9. Repeat steps 2-8 for all 32 CAM entries.

———— **Note** ————

Normal MBIST tests CAM array entries. The CAMBIST routine checks the compare and hit functions only.

### CKBD

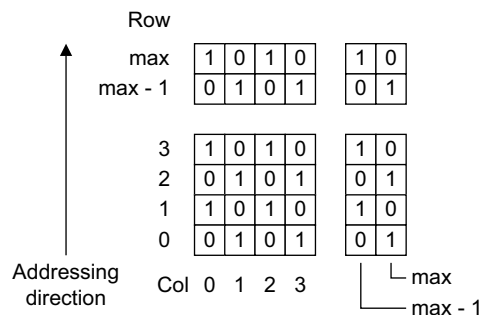
CKBD is a row-fast checkerboard scan pattern in which the following condition determines data seed inversion:

$$\text{invert} = \text{row\_index}[0] \wedge \text{col\_index}[0]$$

CKBD performs the following sequence:

1. wscan array, data\_seed = true.
2. rscan array, data\_seed = true.
3. wscan array, data\_seed = invert.
4. rscan array, data\_seed = invert.

Figure 11-13 shows the physical array after the first CKBD pass.



**Figure 11-13 Physical array after pass 1 of CKBD**

### COLBAR

COLBAR is a column-fast stripe scan pattern in which the following condition determines data seed inversion:

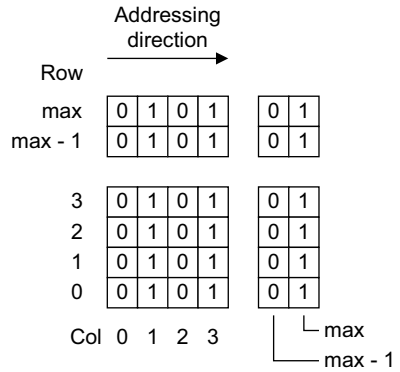
$$\text{invert} = \text{col\_index}[0]$$

COLBAR performs the following sequence:

1. wscan array, data\_seed = true.
2. rscan array, data\_seed = true.
3. wscan array, data\_seed = invert.

4. rscan array, data\_seed = invert.

Figure 11-14 shows the physical array after the first COLBAR pass.



**Figure 11-14 Physical array after pass 1 of COLBAR**

### ROWBAR

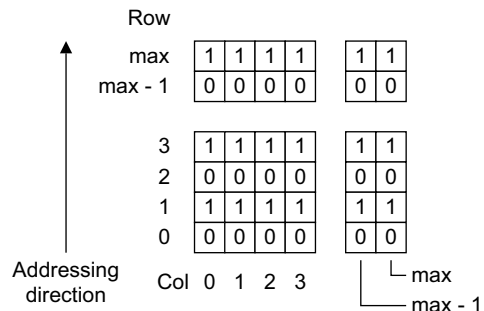
ROWBAR is a row-fast bar-stripe scan pattern in which the following condition determines data seed inversion:

$$\text{invert} = \text{row\_index}[0]$$

ROWBAR performs the following sequence:

1. wscan array, data\_seed = true.
2. rscan array, data\_seed = true.
3. wscan array, data\_seed = invert.
4. rscan array, data\_seed = invert.

Figure 11-15 on page 11-29 shows the physical array after the first ROWBAR pass.



**Figure 11-15 Physical array after pass 1 of ROWBAR**

## SOLIDS

SOLIDS is a row-fast scan pattern in which data seed inversion is not a function of address.

SOLIDS performs the following sequence:

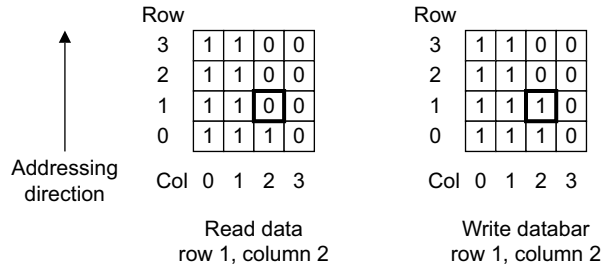
1. wscan array, data\_seed = true.
2. rscan array, data\_seed = true.
3. wscan array, data\_seed = invert.
4. rscan array, data\_seed = invert.

## RWXMARCH

RWXMARCH is a row-fast RW increment/decrement march. It performs the following sequence:

1. wscan data to entire array.
2. R, W\_, incr.
3. R\_, W, decr.
4. rscan data from entire array.

Figure 11-16 on page 11-30 shows the state of row 1, column 2 in a 4 4 array during pass 2.



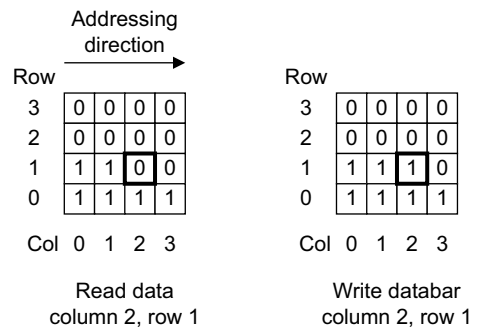
**Figure 11-16 Row 1 column 2 state during pass 2 of RWXMARCH**

### RWYMARCH

RWYMARCH is a column-fast RW increment/decrement march. It performs the following sequence:

1. wscan data to entire array.
2. R, W\_, incr.
3. R\_, W, decr.
4. rscan data from entire array.

Figure 11-17 shows the state of row 1, column 2 in a 4 4 array during pass 2.



**Figure 11-17 Row 1 column 2 state during pass 2 of RWYMARCH**

### RWRXMARCH

RWRXMARCH is a row-fast RWR increment/decrement march. It differs from the RW march in that it requires consecutive reads of opposite data from consecutive addresses.

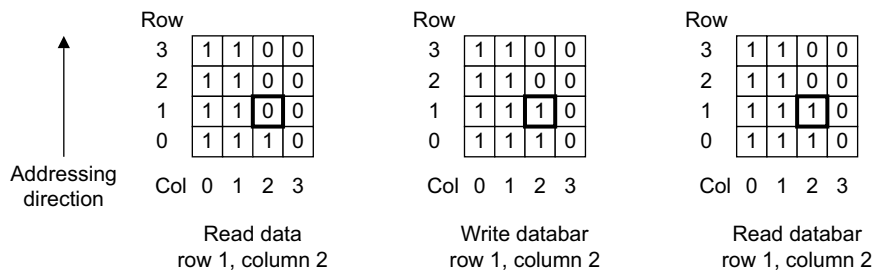
RWRXMARCH performs the following sequence:

1. wscan data to entire array.



2. R, W\_, R\_, incr.
3. R\_, W, R, decr.
4. rscan data from entire array.

Figure 11-18 shows the state of row 1, column 2 in a 4 4 array during pass 2.



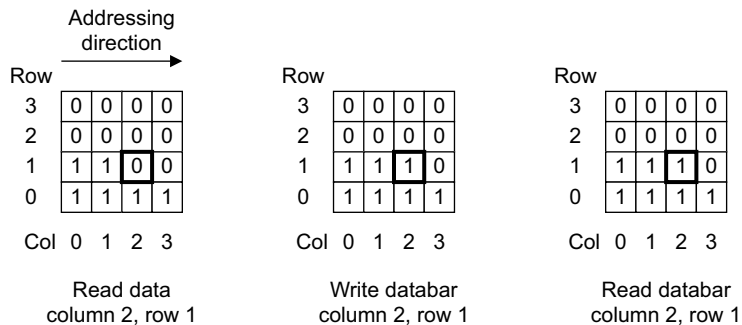
**Figure 11-18 Row 1 column 2 state during pass 2 of RWRXMARCH**

## RWRYMARCH

RWRYMARCH is a column-fast RWR increment/decrement march. It performs the following sequence:

1. wscan data to entire array.
2. R, W\_, R\_, incr.
3. R\_, W, R, decr.
4. rscan data from entire array.

Figure 11-19 shows the state of row 1, column 2 in a 4 4 array during pass 2.



**Figure 11-19 Row 1 column 2 state during pass 2 of RWRYMARCH**

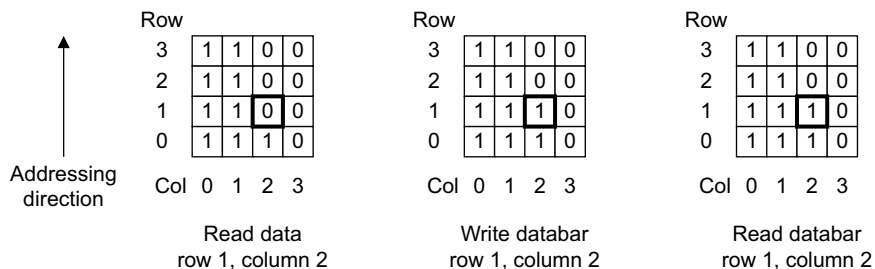
## XMARCHC

XMARCHC is a row-fast RWR increment/decrement march for embedded memory test. It differs from the MARCH patterns in that it repeats the increment/decrement passes with the opposite start data.

XMARCHC performs the following sequence:

1. wscan data.
2. R, W\_, R\_, incr.
3. R\_, W, R, incr.
4. R, W\_, R\_, decr.
5. R\_, W, R, decr.
6. rscan data, decr.

Figure 11-20 shows the state of row 1, column 2 in a 4 4 array during pass 2.



**Figure 11-20 Row 1 column 2 state during pass 2 of XMARCHC**

## YMARCHC

YMARCHC is a column-fast MARCHC. It performs the following sequence:

1. R, W\_, R\_, incr.
2. R, W\_, R\_, incr.
3. R\_, W, R, incr.
4. R, W\_, R\_, decr.
5. R\_, W, R, decr.
6. rscan data, decr.

Figure 11-21 on page 11-33 shows the state of row 1, column 2 in a 4 4 array during pass 2.

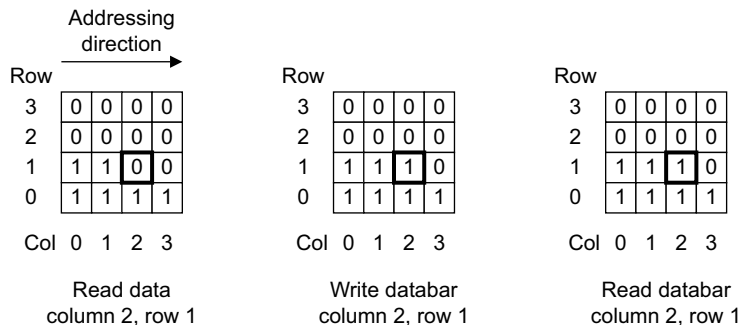


Figure 11-21 Row 1 column 2 state during pass 2 of YMARCHC

## XADDRBAR

XADDRBAR is a write/read row-fast scan pattern with two exceptions. This algorithm uses only half of the MBIST address space. For each address, XADDRBAR also makes an access to the inverted address with inverted data. Unlike the standard scan pattern that moves through the entire address space linearly, it alternates between opposite addresses until it addresses the entire array.

XADDRBAR performs the following sequence:

1. W to Addr, W\_ to inverse of Addr, incr until AddrMax/2.
2. R from Addr, R\_ from inverse of Addr, increment until AddrMax/2.
3. W\_ to Addr, W to inverse of Addr, decrement from AddrMax/2 to zero.
4. R\_ from Addr, R from inverse of Addr, decrement from AddrMax/2 to zero.

In a 4 4 array, Figure 11-22 shows:

- the order of array accesses during XADDRBAR execution
- the data after pass 1 of XADDRBAR using a data seed of 0.

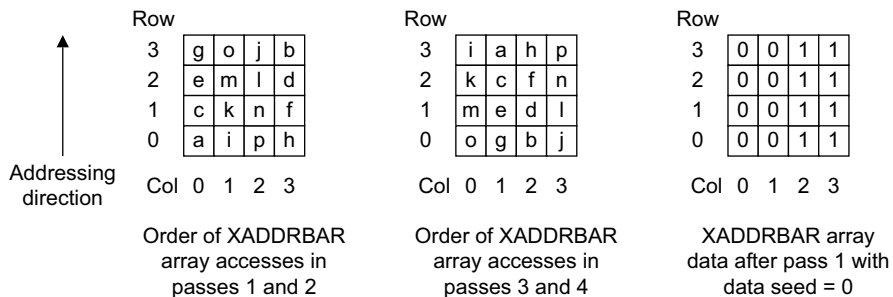


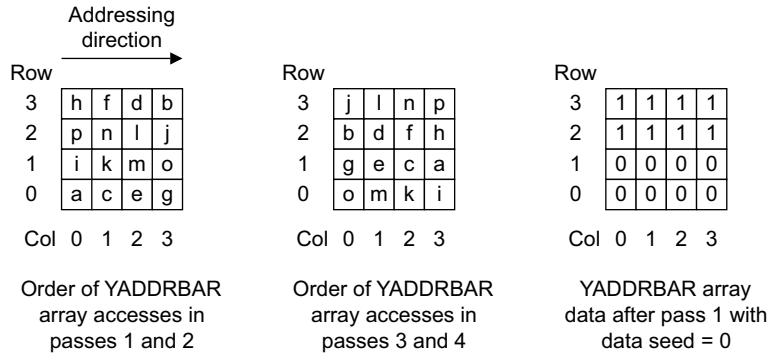
Figure 11-22 XADDRBAR array accessing and data

## YADDRBAR

The YADDRBAR pattern is similar to the XADDRBAR pattern with the exception of incrementing and decrementing the array column-fast.

In a 4 4 array, Figure 11-23 shows:

- the order of array accesses during YADDRBAR execution
- the data after pass 1 of YADDRBAR using a data seed of 0.



**Figure 11-23 YADDRBAR array accessing and data**

## WRITEBANG

WRITEBANG is a row-fast bitline stress pattern. It operates on a bitline pair, that is, a column. It tries to create slow reads from target data cells in the column that can cause hard faults in self-timed and high-speed RAMs. It writes the bitline multiple times to the opposite data state of the target read, trying to create an imbalance in the bitline pair that the cell must correct. The pattern reveals insufficient bitline precharge or equalization. The target cell has opposite data from all other cells on the bitline pair. This is a worst-case bitline condition for a cell to drive because any leakage from other cells in the column oppose the targeted read. In the following description, wsac indicates a write to row 0, a sacrificial (untested) row used during test. WRITEBANG performs the following sequence:

1. Wscan data to entire array.
2. W\_, R\_, wsac, wsac, wsac, wsac, R\_, W, incr.
3. Wscan databar.
4. W, R, wsac\_, wsac\_, wsac\_, wsac\_, wsac\_, R, W\_, incr.

Figure 11-24 on page 11-35 shows the state of row 1, column 2 in a 4 4 array during pass 2.

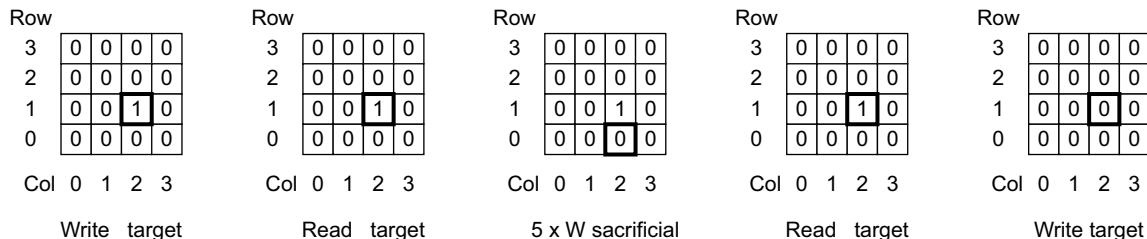


Figure 11-24 WRITEBANG

## READBANG

READBANG is a row-fast bitcell stress test pattern. The pattern operates on a bitcell, reading it multiple times in an attempt to weaken its latched margin. It writes opposite data to the sacrificial row, and makes a final read of the target cell. In the following description, wsac indicates a write to row 0, a sacrificial row used during test.

READBANG performs the following sequence:

1. Wscan data to entire array.
2. R, R, R, R, R, wsac\_, R, W\_, incr.
3. R\_, R\_, R\_, R\_, R\_, wsac, R\_, W, incr.

Figure 11-25 shows the state of row 1, column 2 in a 4x4 array during pass 2.

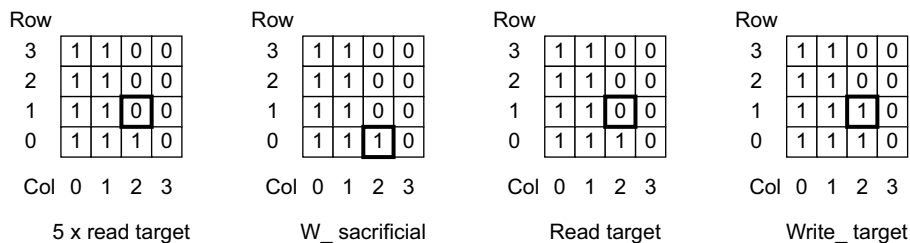


Figure 11-25 READBANG

## FAIL

FAIL is a row-fast algorithm similar to the RWXMARCH pattern but contains injected failures of opposite data written during the wscan portion of the algorithm in one of every 16 accesses. Running FAIL with the production test suite ensures that the MBIST error detection and reporting occurs properly. You can use FAIL to check bitmap mode function in simulation.

## ADDRESS DECODER

ADDRESS DECODER targets the address decoders in memory instead of the bitcells. It performs the following sequence:

1. W addr.
2.  $W\_ (Addr \wedge \text{shift\_reg})$ .
3. R Addr,  $\text{shift\_reg} \ll 1$ .
4. Repeat steps 2 and 3 until  $\text{shift\_reg} = \text{Addr MSB}$ .
5. Incr addr, repeat steps 1-4 until addr expire.

The shift register is a one-hot register with a width equal to the number of address bits.

The test time for the maximum L2 indexing (17 bits) is:

$$(1 + (2^{17})) 2^{17} = 4,587,520 \text{ cycles, not including latency.}$$

The test time for the maximum L1 indexing (11 bits) is:

$$(1 + (2^{11})) 2^{11} = 47,104 \text{ cycles, not including latency.}$$

## GO-NOGO

GO-NOGO is a concatenation of other basic patterns. You can choose between a standard default GO-NOGO sequence and a programmable sequence. The default GO-NOGO performs the following sequence:

1. CKBD, data seed = 0x5.
2. RWRYMARCH, data seed = 0xF.
3. WRITEBANG, data seed = 0xC.

You can select up to eight different pattern combinations and select the data seed of your choice.

### ———— Note —————

Be sure to properly order the sequence of tests. For example, a write CKBD followed by a read solid always fails because the data read was different from what was written.

## 11.2 ATPG test features

This section describes test features that are included in the RTL to ensure that the DFT implementation meets minimum requirements:

- *Wrapper*
- *Enabling sections of the core* on page 11-39
- *Reset handling* on page 11-40.
- *Safe shift RAM signals* on page 11-40.

### 11.2.1 Wrapper

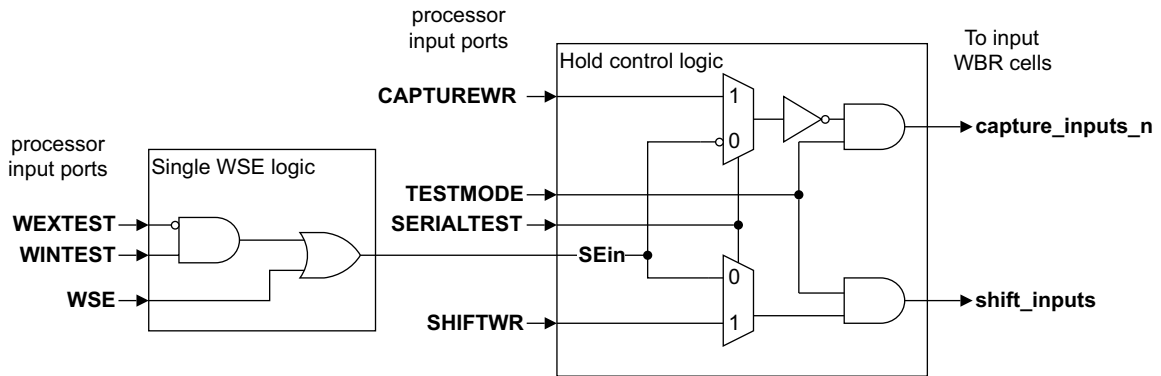
There are seven input signals that control the logic of the core to support the *Wrapper Boundary Register (WBR)* and the IEEE 1500 standard:

- **WEXTEST**
- **WINTEST**
- **WSE**
- **CAPTUREWR**
- **TESTMODE**
- **SERIALTEST**
- **SHIFTR.**

This logic:

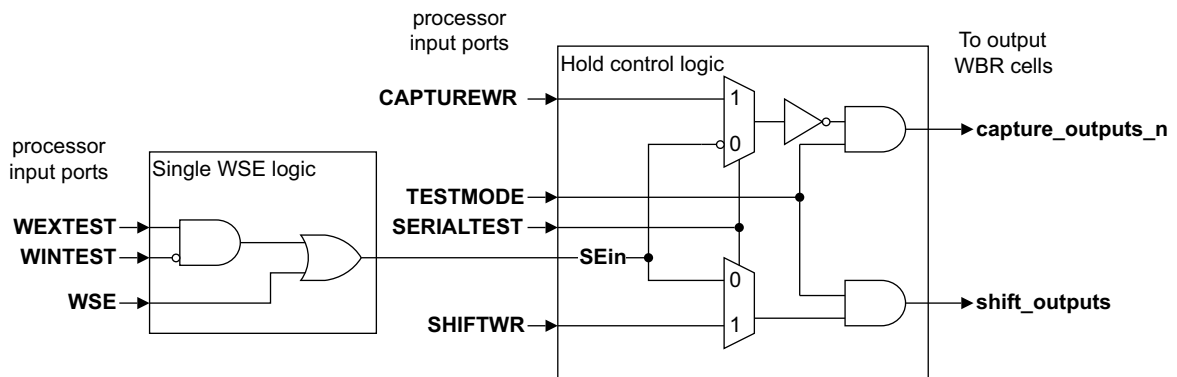
- separates the shift and capture for IEEE 1500 compliance so that the shared wrapper cell can hold state when neither shifting nor capturing
- requires only one external wrapper scan enable and prevents unknown states in wrapper cells with multiple capture cycles, which is preferable for delay testing and for testing through the memories.

Figure 11-26 on page 11-38 shows the RTL logic for a set of input WBR cells.



**Figure 11-26 Input wrapper boundary register cell control logic**

Figure 11-27 shows the RTL logic for a set of output WBR cells.



**Figure 11-27 Output wrapper boundary register cell control logic**

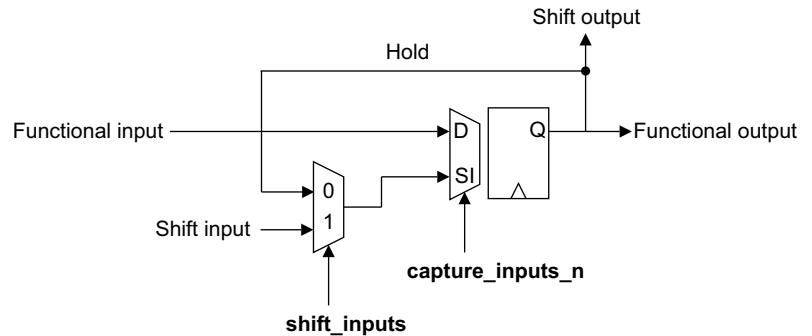
The hold control logic in Figure 11-26 and Figure 11-27 has capture and shift signals that enable the WBR cell to hold data during test mode while both these signals are deasserted. The only difference between the input wrapper and output wrapper cells is that the **WINTEST** and **WEXTEST** connections switch polarity. The type of IEEE 1500 compliant-wrapper cell used with this logic is shown in Figure 11-28 on page 11-39.

This utilization provides the benefit of requiring only one external wrapper scan enable and preventing unknown states from being output from the WBR cells during patterns with multiple capture cycles. If you use a standard multiplexed-scan flip-flop in the



WBR in place of the WBR cell as shown in Figure 11-28, you can use the **shift\_outputs** and **shift\_inputs** signals for the scan enable to the output and input WBR cells, respectively.

Figure 11-28 shows the type of WBR cell required to meet IEEE 1500 compliance.



**Figure 11-28 IEEE 1500-compliant input wrapper boundary register cell**

————— **Note** —————

The IEEE 1500-compliant output wrapper boundary register cell uses the **shift\_outputs** and **capture\_outputs\_n** signals.

## 11.2.2 Enabling sections of the core

Three Cortex-A8 signals control whether or not sections of the processor can update when **MBISTMODE** is asserted or when **TESTMODE** is asserted. These signals are:

- **TESTEGATE**
- **TESTNGATE**
- **TESTCGATE**.

When either of these modes is asserted:

- if the **TESTEGATE** signal is LOW, the flops within the ETM unit cannot be updated
- if the **TESTNGATE** signal is LOW, the flops within the NEON unit cannot be updated
- the rest of the flops within the Cortex-A8 core are not updated when the **TESTCGATE** signal is LOW.

If these signals are HIGH, then the flip-flops that they control are allowed to update. When both **MBISTMODE** and **TESTMODE** are negated, the values of the **TESTEGATE**, **TESTNGATE**, and **TESTCGATE** inputs are not used.

### 11.2.3 Reset handling

The internal asynchronous reset signals are driven from a flip-flop. **SE** prevents the resettable registers from being corrupted during shift using the logic shown in Figure 11-29.

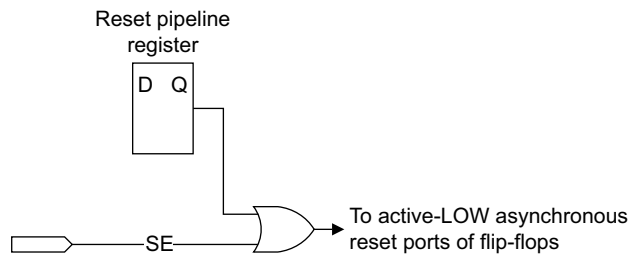


Figure 11-29 Reset handling

### 11.2.4 Safe shift RAM signals

The Cortex-A8 core has separate safe shift RAM signal for each logical unit that uses it, they are:

- **SAFESHIFTRAMIF**
- **SAFESHIFTRAMLS**
- **SAFESHIFTRAML2.**

These safe shift RAM signals are top-level signals with scan enable functionality. They are asserted during scan shifting to gate off the chip selects and write enables of the L1 cache RAMs. These signals are also used to gate off the clock signal to the L2 cache RAMs, as Figure 11-30 shows.

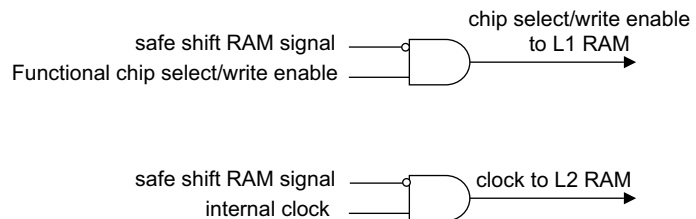


Figure 11-30 Safe shift RAM signal

One methodology for testing the shadow logic of the RAMs is to test through the RAMs. The ATPG tool uses this gate for easier testability of this logic for this methodology. However, if there is a scan chain or bypass wrapper within the RAM, this gate prevents the clock from toggling during shift and causes the chain or wrapper to be ignored during test. If you do not require this gate, you can optimize it out during synthesis by setting **SAFESHIFTRAMIF**, **SAFESHIFTRAMLS**, or **SAFESHIFTRAML2** LOW.

———— **Note** —————

When removing the safe shift RAM gate from a logical unit, all RAMs in that logical unit are affected.

---



# Chapter 12

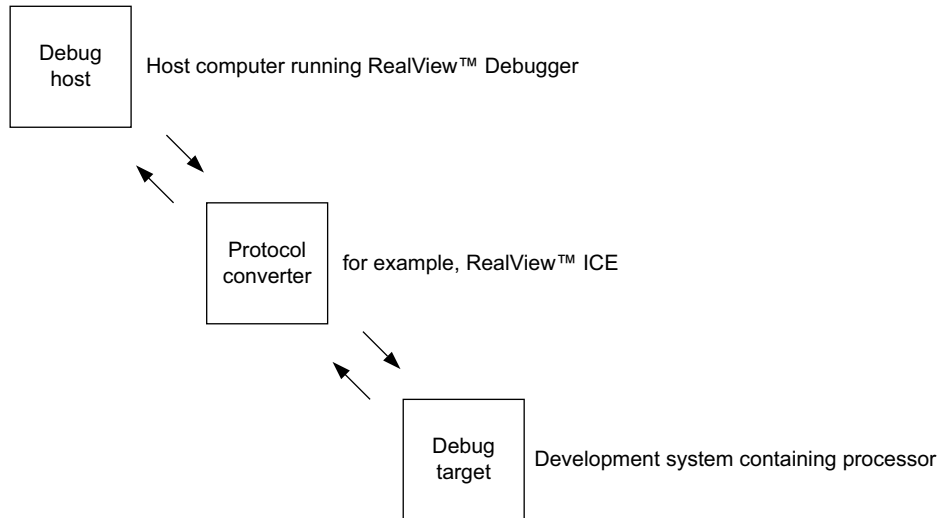
## Debug

This chapter describes the processor debug unit. This feature assists the development of application software, operating systems, and hardware. This chapter contains the following sections:

- *Debug systems* on page 12-2
- *About the debug unit* on page 12-4
- *Debug register interface* on page 12-7
- *Debug register descriptions* on page 12-17
- *Management registers* on page 12-54
- *Debug events* on page 12-70
- *Debug exception* on page 12-74
- *Debug state* on page 12-78
- *Cache debug* on page 12-87
- *External debug interface* on page 12-89
- *Using the debug functionality* on page 12-94
- *Debugging systems with energy management capabilities* on page 12-116.

## 12.1 Debug systems

The processor forms one component of a debug system. Figure 12-1 shows a typical system.



**Figure 12-1 Typical debug system**

This typical system has three parts:

- *Debug host*
- *Protocol converter*
- *Debug target* on page 12-3.

### 12.1.1 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 setting a breakpoint at a certain location, or examining the contents of a memory address.

### 12.1.2 Protocol converter

The debug host sends messages to the debug target using an interface such as Ethernet. However, the debug target typically implements a different interface protocol. A device such as RealView ICE is required to convert between the two protocols.

### 12.1.3 Debug target

The debug target is the lowest level of the system. An example of a debug target is a development system with a test chip or a silicon part with a processor.

The debug target implements system support for the protocol converter to access the debug unit using the *Advanced Peripheral Bus* (APB) slave interface.

The debug unit enables you to:

- stall program execution
- examine the internal state of the processor and the state of the memory system
- resume program execution.

## 12.2 About the debug unit

The processor debug unit assists in debugging software running on the processor. You can use the processor debug unit, in combination with a software debugger program, to debug:

- application software
- operating systems
- hardware systems based on an ARM processor.

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 software running on the processor in the following ways:

- *Halting debug-mode debugging*
- *Monitor debug-mode debugging*
- trace debugging, see Chapter 14 *Embedded Trace Macrocell*.

The processor external debug interface is compliant with the *AMBA 3 APB Protocol Specification*.

### 12.2.1 Halting debug-mode debugging

When the processor debug unit is in Halting debug-mode, the processor halts when a debug event, such as a breakpoint, occurs. When the processor is halted, an external debugger can examine and modify the processor state using the APB interface. This debug mode is invasive to program execution.

### 12.2.2 Monitor debug-mode debugging

When the processor debug unit is in Monitor debug-mode and a debug event occurs, 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 debug-mode is essential in real-time systems where the processor cannot be halted to collect debug information. Examples of these systems are engine controllers and servo mechanisms in hard drive controllers that cannot stop the code without physically damaging the components.



When execution of a monitor target starts, the state of the processor is preserved in the same manner as all ARM exceptions. The monitor target then communicates with the debugger to access processor and coprocessor state, and to access memory contents and input/output peripherals. Monitor debug-mode requires a debug monitor program to interface between the debug hardware and the software debugger.

———— **Note** —————

Monitor debug-mode, used for debugging, is not the same as Secure Monitor mode, which is a CPSR[4:0] processor mode.

See *CP14 c1, Debug Status and Control Register* on page 12-21 for information on how to select between Halting debug-mode or Monitor debug-mode.

### 12.2.3 Security extensions and debug

To prevent access to secure system software or data while still permitting Nonsecure state and optionally secure User mode to be debugged, you can set debug to one of three levels:

- Nonsecure state only
- Nonsecure state and Secure User mode only
- any Secure or Nonsecure state.

The **SPIDEN** and **SPNIDEN** signals, and the two bits, **SUIDEN** and **SUNIDEN**, in the Secure Debug Enable Register in CP15 coprocessor control the debug permissions. See *External debug interface* on page 12-89 and *c1, Secure Debug Enable Register* on page 3-72 for details.

The processor implements two types of debug support:

#### **Invasive debug**

Invasive debug is defined as a debug process where you can control and observe the processor. Most debug features in this chapter are considered invasive debug because they enable you to halt the processor and modify its state.

**SPIDEN** and **SUIDEN** control invasive debug permissions.

#### **Noninvasive debug**

Noninvasive debug is defined as a debug process where you can observe the processor but not control it. The ETM interface and the performance monitor registers are features of noninvasive debug. See Chapter 14

*Embedded Trace Macrocell* for information on ETM. See Chapter 3 *System Control Coprocessor* for information on performance monitor registers.

**SPNIDEN** and **SUNIDEN** control noninvasive debug permissions.

#### 12.2.4 Programming the debug unit

The processor debug unit is programmed using the APB interface. See Table 12-3 on page 12-9 for a complete list of memory-mapped debug registers accessible using the APB interface. Some features of the debug unit that you can access using the memory-mapped registers are:

- instruction address comparators for triggering breakpoints, see *Breakpoint Value Registers* on page 12-37 and *Breakpoint Control Registers* on page 12-38
- data address comparators for triggering watchpoints, see *Watchpoint Value Registers* on page 12-42 and *Watchpoint Control Registers* on page 12-43
- a bidirectional *Debug Communication Channel (DCC)*, see *Debug communications channel* on page 12-95
- all other state information associated with the debug unit.

## 12.3 Debug register interface

You can access the debug register map using the APB interface. This is the only way to get full access to the processor debug capability. ARM recommends that if your system requires the processor to access its own debug registers, you choose a system interconnect structure that enables the processor to access the APB interface by executing load and stores to a certain area of physical memory.

### 12.3.1 Coprocessor registers

Although most of the debug registers are accessible through the memory-mapped interface, there are several registers that you can access through a coprocessor interface. This is important for boot-strap access to the ARM register file. It enables software running on the processor to identify the debug architecture version implemented by the device.

### 12.3.2 CP14 access permissions

By default, you can access all CP14 debug registers from a nonprivileged mode. However, you can program the processor to disable user-mode access to all coprocessor registers using bit [12] of the DSCR, see *CP14 c1, Debug Status and Control Register* on page 12-21 for more information. CP14 debug registers accesses are always permitted while the processor is in debug state regardless of the processor mode.

Table 12-1 shows access to the CP14 debug registers.

**Table 12-1 Access to CP14 debug registers**

Debug state	Processor mode	DSCR[12]	CP14 debug access
Yes	X <sup>a</sup>	X <sup>a</sup>	Permitted
No	User	b0	Permitted
No	User	b1	Undefined
No	Privileged	X <sup>a</sup>	Permitted

a. X indicates a *Don't care* condition. The outcome does not depend on this condition.

### 12.3.3 Coprocessor registers summary

Table 12-2 on page 12-8 shows the valid CP14 debug instructions for accessing the debug registers. All CP14 debug instructions not listed are Undefined.

---

**Note**


---

The CP14 debug instructions are defined as having Opcode\_1 set to 0.

**Table 12-2 CP14 debug registers summary**

<b>Instruction</b>	<b>Mnemonic</b>	<b>Description</b>
MRC p14, 0, <Rd>, c0, c0, 0	DIDR	Debug Identification Register. See <i>CP14 c0, Debug ID Register</i> on page 12-18.
MRC p14, 0, <Rd>, c1, c0, 0	DRAR	Debug ROM Address Register. See <i>CP14 c0, Debug ROM Address Register</i> on page 12-19.
MRC p14, 0, <Rd>, c2, c0, 0	DSAR	Debug Self Address Register. See <i>CP14 c0, Debug Self Address Offset Register</i> on page 12-20.
MRC p14, 0, <Rd>, c0, c5, 0 STC p14, c5, <addressing mode>	DTRRX	Data Transfer Register - Receive. See <i>Data Transfer Register</i> on page 12-29.
MCR p14, 0, <Rd>, c0, c5, 0 LDC p14, c5, <addressing mode>	DTRTX	Data Transfer Register - Transmit. See <i>Data Transfer Register</i> on page 12-29.
MRC p14, 0, <Rd>, c0, c1, 0 MRC p14, 0, PC, c0, c1, 0	DSCR	Debug Status and Control Register. See <i>CP14 c1, Debug Status and Control Register</i> on page 12-21.

### 12.3.4 Memory-mapped registers

Table 12-3 on page 12-9 shows the complete list of memory-mapped registers accessible using the APB interface.

---

**Note**


---

You must ensure that the base address of this 4KB register region is aligned to a 4KB boundary in physical memory.

---

Table 12-3 Debug memory-mapped registers

Offset	Register number	Access	Mnemonic	Power domain	Description
0x000	c0	R	DIDR	Debug	CP14 c0, Debug ID Register on page 12-18
0x004-0x014	c1-c5	R	-	-	RAZ
0x18	c6	RW	WFAR	Core	Watchpoint Fault Address Register on page 12-30
0x01C	c7	RW	VCR	Core	Vector Catch Register on page 12-31
0x020	c8	R	-	-	RAZ
0x024	c9	RW	ECR	Debug	Event Catch Register on page 12-33
0x028	c10	RW	DSCCR	Core	Debug State Cache Control Register on page 12-34
0x02C	c11	R	-	-	RAZ
0x030-0x07C	c12-c31	R	-	-	RAZ
0x080	c32	RW	DTRRX	Core	Data Transfer Register on page 12-29
0x084	c33	W	ITR	Core	Instruction Transfer Register on page 12-35
0x088	c34	RW	DSCR	Core	CP14 c1, Debug Status and Control Register on page 12-21
0x08C	c35	RW	DTRTX	Core	Data Transfer Register on page 12-29
0x090	c36	W	DRCR	Debug	Debug Run Control Register on page 12-36
0x094-0x0FC	c37-c63	R	-	-	RAZ
0x100-0x114	c64-c69	RW	BVR	Core	Breakpoint Value Registers on page 12-37
0x118-0x13C	c70-c79	R	-	-	RAZ
0x140-0x154	c80-c85	RW	BCR	Core	Breakpoint Control Registers on page 12-38
0x158-0x17C	c86-c95	R	-	-	RAZ
0x180-0x184	c96-c97	RW	WVR	Core	Watchpoint Value Registers on page 12-42
0x188-0x1BC	c97-c111	R	-	-	RAZ

Table 12-3 Debug memory-mapped registers (continued)

Offset	Register number	Access	Mnemonic	Power domain	Description
0x1C0-0x1C4	c112-c113	RW	WCR	Core	<i>Watchpoint Control Registers</i> on page 12-43
0x1C8-0x1FC	c114-c127	R	-	-	RAZ
0x200-0x2FC	c128-c191	R	-	-	RAZ
0x300	c192	W	OSLAR	Debug	<i>Operating System Lock Access Register</i> on page 12-46
0x304	c193	R	OSLSR	Debug	<i>Operating System Lock Status Register</i> on page 12-47
0x308	c194	RW	OSSRR	-	<i>Operating System Save and Restore Register</i> on page 12-48
0x30C	c195	R	-	-	RAZ
0x310	c196	RW	PRCR	Debug	<i>Device Power Down and Reset Control Register</i> on page 12-50
0x314	c197	R	PRSR	Debug	<i>Device Power Down and Reset Status Register</i> on page 12-51
0x318-0x7FC	c198-c511	R	-	-	RAZ
0x800-0x8FC	c512-575	R	-	-	RAZ
0x900-0xFCF	c576-c831	R	-	-	RAZ
0xD00-0xFFC	c832-c1023	-	-	-	<i>Management registers</i> on page 12-54

### 12.3.5 Memory addresses for breakpoints and watchpoints

The breakpoint and watchpoint comparisons are done on a *Virtual Address (VA)*. Therefore you must program *Breakpoint Value Registers (BVRs)* and *Watchpoint Value Registers (WVRs)* with a VA, not a *Modified Virtual Address (MVA)*.

The *Vector Catch Register (VCR)* sets breakpoints on exception vectors as virtual addresses.

The *Watchpoint Fault Address Register (WFAR)* reads a VA plus a processor state dependent offset, +8 for ARM state and +4 for Thumb and ThumbEE states.

### 12.3.6 Power domains and debug

This section describes how the debug registers are split in different power domains. Table 12-3 on page 12-9 describes which debug registers are included in which power domain. Generally, debug registers are in the core power domain unless they must be accessible or hold their values while the core is powered down, in which case they are in the debug power domain. In addition, the APB interface itself is also in the debug power domain. Debug registers that are in debug power domain are:

- ID registers and most of the registers in the Management Registers space
- Registers that implement the functionality for debugging through power down such as:
  - Event Catch Register
  - Debug Run Control Register
  - OS Lock Access Register
  - Device Power Down and Reset Control Register
  - Device Power Down and Status Control Register.

### 12.3.7 Effects of resets on debug registers

The processor has three reset signals that affect the debug registers in the following ways:

**nPORESET** The system asserts this signal when powering up the core domain. It sets all of the core power domain logic to the reset value, including all debug registers in the core power domain.

**ARESETn** The system asserts this signal for a warm or soft reset. It sets all the processor logic except debug or ETM, to the reset value. Therefore, the state of a debug or trace session is not affected by this reset signal.

**PRESETn** The system asserts this signal to set all of the debug and ETM logic to the reset value.

Table 12-4 shows the processor reset effect on debug and ETM logic.

**Table 12-4 Processor reset effect on debug and ETM logic**

Signal	Debug power domain		Core power domain
	Debug and ETM logic	Debug and ETM logic	Non-debug and non-ETM logic
<b>nPORESET</b>	Not reset	Reset	Reset
<b>ARESETn</b>	Not reset	Not reset	Reset
<b>PRESETn</b>	Reset	Reset	Not reset

### 12.3.8 APB interface access permissions

The restrictions on accesses to the APB interface are described as follows:

#### Privilege of memory access

The system disables accesses to the memory-mapped registers based on the privilege of the memory access.

#### Locks

The debugger or software running on the system might lock out different parts of the register map so they cannot be accessed while the debug session is in certain states.

#### Power down

The APB interface does not permit accesses to registers inside the core power domain when the core powers down.

#### Privilege of memory access permission

When nonprivileged software tries to access the APB interface, the system ignores or generates an abort response on the access. You must implement this restriction at the system level because the APB protocol does not have a control signal for privileged or user access. You can choose to have the system either ignore or abort the access. Although you can place additional restrictions on the memory transactions that are permitted to access the APB interface, ARM does not recommend this.



## Locks permission

You can lock the APB interface so access to some debug registers is restricted. There are two locks:

### Software lock

The debug monitor can set this lock to prevent erratic software from modifying debug registers settings. See the *ARM Architecture Reference Manual* for more information. A debug monitor can also set this lock prior to returning control to the application, to reduce the chance of erratic code changing the debug settings. When this lock is set, writes to all debug registers are ignored, except those writes generated by the external debugger. See *Lock Access Register* on page 12-63 for more information.

**OS Lock** An OS can set this lock on the debug register map so access to some debug registers is not permitted while the OS is performing a save or restore sequence. When this lock is set, the APB interface aborts accesses to registers in the core power domain. See *Operating System Lock Access Register* on page 12-46 for more information.

---

### Note

---

- The state of these locks is held on debug power domain and, therefore, is not lost when the core powers down.
  - These locks are set to their reset values only on reset of the debug power domain (**PRESETn** reset).
  - Be sure to set the **PADDR31** input signal to 1 for accesses originated from the external debugger for the Software Lock override feature to work. See Table 12-5 on page 12-14 for more details.
  - If you access a reserved or unused register while any lock is set to 1, it is Unpredictable whether or not the APB interface generates an error response.
-

Table 12-5 shows the APB interface access permissions with relation to software lock.

**Table 12-5 APB interface access with relation to software lock**

Conditions		Registers	
PADDR31	Lock	LAR	Other registers
1 <sup>a</sup>	X <sup>b</sup>	OK <sup>c</sup>	OK
0	1 <sup>d</sup>	OK	WI <sup>e</sup>
0	0	OK	OK

- a. The **PADDR31** signal is HIGH, indicating the external debugger generated the access.
- b. X indicates a *Don't care* condition. The outcome does not depend on this condition.
- c. OK indicates that the access succeeds.
- d. LSR[1] bit is set to 1.
- e. WI indicates that writes are ignored, and that reads do not change the processor state.

### Power down permission

Access to registers inside the core power domain is not possible when the core powers down. The APB interface ignores accesses to powered-down registers and returns an error response, that is, **PSLVERR** is set to 1.

When the core powers down, the PRSR[1] sticky power down bit is set to 1. While PRSR[1] is set to 1, the APB interface also ignores accesses to registers inside the core power domain and returns an error response, that is, **PSLVERR** is set to 1. This bit remains set until the debugger reads the PRSR. See *Device Power Down and Reset Status Register* on page 12-51 for more details.

Table 12-6 shows the behavior of APB interface accesses to debug registers with relation to power-down event.

**Table 12-6 Debug registers access with relation to power-down event**

Conditions		Registers			
DBGPWRDWNREQ	Sticky power down	OS Lock	DIDR, ECR, DRCR	Other debug <sup>a</sup>	Management <sup>b</sup>
1	X <sup>c</sup>	X	OK <sup>d</sup>	ERR <sup>e</sup>	OK
0 <sup>f</sup>	0	0	OK	OK	OK
0	0	1 <sup>g</sup>	OK	ERR	OK
0	1 <sup>h</sup>	X	OK	ERR	OK

- This column indicates registers in the address range of 0x000 through 0xF00 except for DIDR, ECR, DRCR, and the power management registers specified in Table 12-7.
- This column indicates registers in the address range of 0xF04 through 0xFFC.
- X indicates a *Don't care* condition. The outcome does not depend on this condition.
- OK indicates that the access succeeds.
- ERR indicates a **PSLVERR** error response; written value is ignored and reads return an Unpredictable value.
- The **DBGPWRDWNREQ** signal is LOW, indicating the processor is powered up.
- 1 indicates that OSLSR[1] is set to 1.
- 1 indicates that PRSR[1] is set to 1.

Table 12-7 shows the behavior of APB interface accesses to power management registers with relation to power-down event.

**Table 12-7 Power management registers access with relation to power-down event**

Conditions		Registers			
DBGPWRDWNREQ	Sticky power down	OS Lock	OSLSR, PRCR, PRSR	OSLAR	OSSRR
1	X <sup>a</sup>	X	OK <sup>b</sup>	UNP <sup>c</sup>	UNP
0 <sup>d</sup>	0	0	OK	OK	UNP
0	0	1 <sup>e</sup>	OK	OK	OK
0	1 <sup>f</sup>	X	OK	OK	UNP

- X indicates a *Don't care* condition. The outcome does not depend on this condition.
- OK indicates that the access succeeds.
- UNP indicates that the access has Unpredictable results; reads return an Unpredictable value.
- The **DBGPWRDWNREQ** signal is LOW, indicating the processor is powered up.
- 1 indicates that OSLSR[1] is set to 1.

- f. 1 indicates that PRSR[1] is set to 1.

### Accesses to ETM and CTI registers

Similarly to the restrictions on accesses to debug registers as described in *Power down permission* on page 12-14, the APB interface can restrict accesses to the ETM and CTI registers based on the occurrence of power-down events.

Table 12-8 shows the behavior of APB interface accesses to ETM and CTI registers with relation to power-down event.

**Table 12-8 ETM and CTI registers access with relation to power-down event**

Conditions	Registers					
	DBGPWRDWNREQ	OS Lock	OSLSR	OSLAR	OSSRR	Other <sup>a</sup>
1	X <sup>c</sup>	OK <sup>d</sup>	UNP <sup>e</sup>	UNP	ERR <sup>f</sup>	OK
0 <sup>g</sup>	0	OK	OK	UNP	OK	OK
0	1 <sup>h</sup>	OK	OK	OK	ERR	OK

- a. This column indicates registers in the address range of 0x000 through 0xF00 except for OSLSR, OSLAR, OSSRR, and PRSR registers.
- b. This column indicates registers in the address range of 0xF04 through 0xFFC.
- c. X indicates a *Don't care* condition. The outcome does not depend on this condition.
- d. OK indicates that the access succeeds.
- e. UNP indicates that the access has Unpredictable results; reads return an Unpredictable value.
- f. ERR indicates a **PSLVERR** error response; written value is ignored and reads return an Unpredictable value.
- g. The **DBGPWRDWNREQ** signal is LOW, indicating the processor is powered up.
- h. 1 indicates that OSLSR[1] is set to 1.

#### Note

The OS Lock, OSLSR, OSLAR, OSSRR, and PRSR registers described in this section are all part of the ETM programmer's model. Do not confuse these registers with the debug registers of the same name.

## 12.4 Debug register descriptions

Table 12-9 shows definitions of terms used in the register descriptions.

**Table 12-9 Terms used in register descriptions**

Term	Description
R	Read-only. Written values are ignored.
W	Write-only. This bit cannot be read. Reads return an Unpredictable value.
RW	Read or write.
RAZ	<i>Read-As-Zero</i> (RAZ). Return a value of zero when read.
SBZ	<i>Should-Be-Zero</i> (SBZ). Should be written as zero (or all 0s for bit fields) by software. Non-zero values might produce Unpredictable results.
SBZP	<i>Should-Be-Zero or Preserved</i> (SBZP). Should be written as zero (or all 0s for bit fields) or preserved by writing the same value that has been previously read from the same fields on the same processor.
UNP	A read of this bit returns an Unpredictable value.

### 12.4.1 Accessing debug registers

To access the CP14 debug registers you set Opcode\_1 to 0. The rest of the fields of the coprocessor instruction determine the debug register being accessed.

Table 12-10 shows the CP14 debug register map.

**Table 12-10 CP14 debug registers**

CRn	Op1	CRm	Op2	CP14 debug register name	Abbreviation	Reference
c0	0	c0	0	Debug ID Register	DIDR	<i>CP14 c0, Debug ID Register</i> on page 12-18
c1	0	c0	0	Debug ROM Address Register	DRAR	<i>CP14 c0, Debug ROM Address Register</i> on page 12-19
c2	0	c0	0	Debug Self Address Offset Register	DSAR	<i>CP14 c0, Debug Self Address Offset Register</i> on page 12-20
c3-c15	0	c0	0	Reserved	-	-

Table 12-10 CP14 debug registers (continued)

CRn	Op1	CRm	Op2	CP14 debug register name	Abbreviation	Reference
c0	0	c1	0	Debug Status and Control Register	DSCR	<i>CP14 c1, Debug Status and Control Register on page 12-21</i>
c1-c15	0	c1	0	Reserved	-	-
c0-c15	0	c2-c4	0	Reserved	-	-
c0	0	c5	0	Data Transfer Register	DTR	<i>Data Transfer Register on page 12-29</i>
c0-c15	0	c6-c15	0	Reserved	-	-
c0-c15	0	c0-c15	1-7	Reserved	-	-

### 12.4.2 CP14 c0, Debug ID Register

The DIDR is a read-only register that identifies the debug architecture version and specifies the number of debug resources that the processor implements.

The Debug ID Register is:

- in CP14 c0
- a read-only register
- accessible in User and privileged modes.

Figure 12-2 shows the bit arrangement of the DIDR.

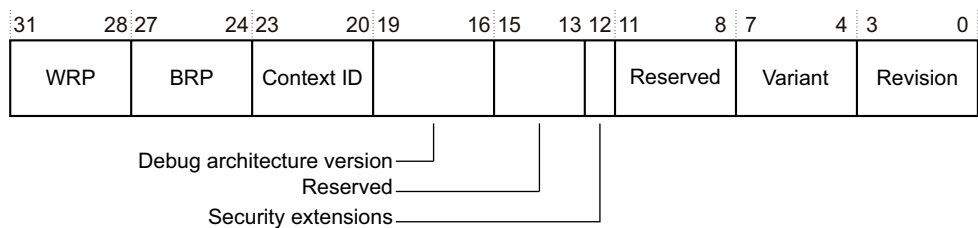


Figure 12-2 Debug ID Register format

Table 12-11 shows how the bit values correspond with the Debug ID Register functions.

**Table 12-11 Debug ID Register bit functions**

Bits	Field	Function
[31:28]	WRP	Number of Watchpoint Register Pairs. For the processor, this field reads b0001 to indicate 2 WRPs are implemented.
[27:24]	BRP	Number of Breakpoint Register Pairs. For the processor, this field reads b0101 to indicate 6 BRPs are implemented.
[23:20]	Context	Number of Breakpoint Register Pairs with context ID comparison capability. For the processor, this field reads b0001 to indicate 2 BRPs have context ID capability.
[19:16]	Debug architecture version	Debug architecture version: b0100 = ARMv7 Debug.
[15:13]	-	RAZ.
[12]	Security extensions	Security extensions bit: 0 = security extensions are not implemented 1 = security extensions implemented. For the processor, this field reads b1 to indicate that the debug security extensions are implemented.
[11:8]	-	RAZ.
[7:4]	Variant	Implementation-defined variant number. This number is incremented on functional changes. The value matches bits [23:20] of the Main ID Register in CP15 c0. See <i>c0, Main ID Register</i> on page 3-25 for more information.
[3:0]	Revision	Implementation-defined revision number. This number is incremented on bug fixes. The value matches bits [3:0] of the Main ID Register in CP15 c0. See <i>c0, Main ID Register</i> on page 3-25 for more information.

To access the Debug ID Register, read CP14 c0 with:

```
MRC p14, 0, <Rd>, c0, c0, 0 ; Read Debug ID Register
```

### 12.4.3 CP14 c0, Debug ROM Address Register

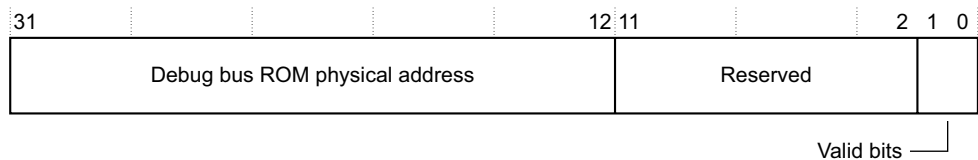
The Debug ROM Address Register is a read-only register that returns a 32-bit Debug ROM Address Register value. This is the physical address that indicates where in memory a debug monitor can locate the debug bus ROM specified by the CoreSight™ multiprocessor trace and debug architecture. This ROM holds information about all the

components in the debug bus. You can configure the address read in this register using **DBGROMADDR[31:12]** and **DBGROMADDRV** inputs. **DBGROMADDRV** must be tied off to 1 if **DBGROMADDR[31:12]** is tied off to a valid value.

The Debug ROM Address Register is:

- in CP14 c0
- a read-only register
- accessible in User and privileged modes.

Figure 12-3 shows the bit arrangement of the Debug ROM Address Register.



**Figure 12-3 Debug ROM Address Register format**

Table 12-12 shows how the bit values correspond with the Debug ROM Address Register functions.

**Table 12-12 Debug ROM Address Register bit functions**

Bits	Field	Function
[31:12]	Debug bus ROM physical address	Indicates bits [31:12] of the debug bus ROM physical address.
[11:2]	-	Reserved. UNP, SBZP.
[1:0]	Valid bits	Reads b11 if <b>DBGROMADDRV</b> is set to 1, reads b00 otherwise. <b>DBGROMADDRV</b> must be set to 1 if <b>DBGROMADDR[31:12]</b> is set to a valid value.

To access the Debug ROM Address Register, read CP14 c0 with:

```
MRC p14, 0, <Rd>, c1, c0, 0 ; Read Debug ROM Address Register
```

#### 12.4.4 CP14 c0, Debug Self Address Offset Register

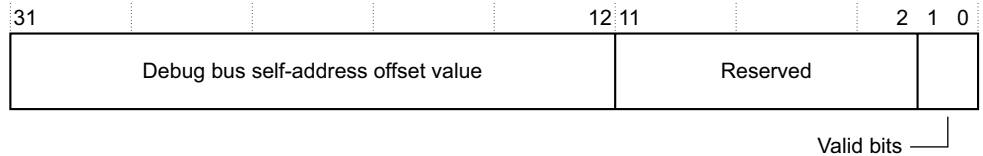
The Debug Self Address Offset Register is a read-only register that returns a 20-bit offset value from the Debug ROM Address Register to the physical address of the processor debug registers. The address read from this register depends on the **DBGSELFADDR[31:12]** and **DBGSELFADDRV** inputs. **DBGSELFADDRV** must be tied off to 1 if **DBGSELFADDR[31:12]** is tied off to a valid value.



The Debug Self Address Offset Register is:

- in CP14 c0
- a read-only register
- accessible in User and privileged modes.

Figure 12-4 shows the bit arrangement of the Debug Self Address Offset Register.



**Figure 12-4 Debug Self Address Offset Register format**

Table 12-13 shows how the bit values correspond with the Debug Self Address Offset Register functions.

**Table 12-13 Debug Self Address Offset Register bit functions**

Bits	Field	Function
[31:12]	Debug bus self-address offset value	Indicates bits [31:12] of the 2's complement offset from the debug ROM physical address to the physical address of the start of the region where the debug registers are mapped. The value read by this field corresponds to the value of <b>DBGSELFADDR[31:12]</b> .
[11:2]	-	Reserved. RAZ, SBZP.
[1:0]	Valid bits	Reads b11 if <b>DBGSELFADDRV</b> is set to 1, reads b00 otherwise. <b>DBGSELFADDRV</b> must be set to 1 if <b>DBGSELFADDR[31:12]</b> is set to a valid value.

To access the Debug Self Address Offset Register, read CP14 c0 with:

MRC p14, 0, <Rd>, c2, c0, 0 ; Read Debug Self Address Offset Register

#### 12.4.5 CP14 c1, Debug Status and Control Register

The DSCR is a read-only register that contains status and control information about the debug unit. Figure 12-5 on page 12-22 shows the bit arrangement of the DSCR.

———— **Note** —————

For the APB interface, the DSCR is a read/write register.

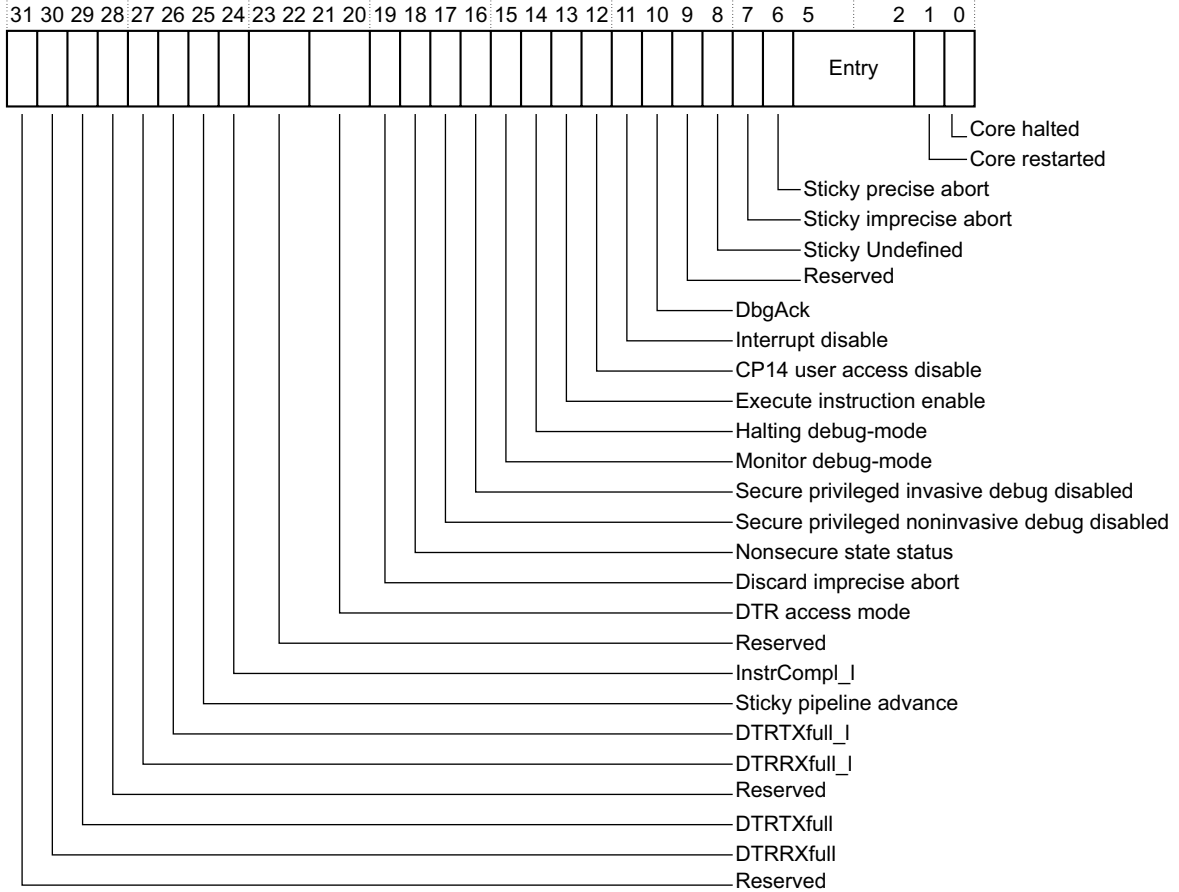


Figure 12-5 Debug Status and Control Register format

Table 12-14 shows how the bit values correspond with the Debug Status and Control Register functions.

**Table 12-14 Debug Status and Control Register bit functions**

Bits	Field	Function
[31]	-	Reserved. RAZ, SBZP.
[30]	DTRRXfull	<p>The DTRRXfull flag:</p> <p>0 = DTRRX empty, reset value</p> <p>1 = DTRRX full.</p> <p>When set to 1, this flag indicates that there is data available in the Receive Data Transfer Register, DTRRX. It is automatically set to 1 on writes to the DTRRX by the debugger, and is cleared to 0 when the processor reads the CP14 DTR. If the flag is not set to 1, the DTRRX returns an Unpredictable value.</p>
[29]	DTRTXfull	<p>The DTRTXfull flag:</p> <p>0 = DTRTX empty, reset value</p> <p>1 = DTRTX full.</p> <p>When set to 0, this flag indicates that the Transmit Data Transfer Register, DTRTX, is ready for data write. It is automatically set to 0 on reads of the DTRTX by the debugger, and is set to 1 when the processor writes to the CP14 DTR. If this bit is set to 1 and the core attempts to write to the DTRTX, the register contents are overwritten and the DTRTXfull flag remains set.</p>
[28]	-	Reserved. RAZ, SBZP.
[27]	DTRRXfull_1	<p>The latched DTRRXfull flag. This flag is read in one of the following ways:</p> <ul style="list-style-type: none"> <li>• using CP14 instruction</li> <li>• using the DSCR memory address</li> <li>• using the OSSRR memory address.</li> </ul> <p>CP14 instruction returns an Unpredictable value for this bit.</p> <p>DSCR memory address returns the same value as DTRRXfull.</p> <p>OSSRR memory address returns the latched DTRRXfull value, that is, the value of DTRRXfull that the processor captured on the last DSCR memory address read.</p> <p>If a write to the DTRRX APB address succeeds, DTRRXfull_1 is set to 1.</p>

Table 12-14 Debug Status and Control Register bit functions (continued)

Bits	Field	Function
[26]	DTRTXfull_1	<p>The latched DTRTXfull flag. This flag is read in one of the following ways:</p> <ul style="list-style-type: none"> <li>• using CP14 instruction</li> <li>• using the DSCR memory address</li> <li>• using the OSSRR memory address.</li> </ul> <p>CP14 instruction returns an Unpredictable value for this bit.  DSCR memory address returns the same value as DTRTXfull.  OSSRR memory address returns the latched DTRTXfull value, that is, the value of DTRTXfull that the processor captured on the last DSCR memory address read.  If a read to the DTRTX APB address succeeds, DTRTXfull_1 is cleared to 0.</p>
[25]	Sticky pipeline advance	<p>Sticky pipeline advance bit. This bit enables the debugger to detect whether the processor is idle. In some situations, this might mean that the system bus port is deadlock. This bit is set to 1 every time the processor pipeline retires one instruction. A write to DRCR[3] clears this bit to 0. See <i>Debug Run Control Register</i> on page 12-36.</p> <p>0 = no instruction has completed execution since the last time this bit was cleared, reset value  1 = an instruction has completed execution since the last time this bit was cleared.</p>
[24]	InstrCompl_1	<p>The latched InstrCompl flag. This flag is read in one of the following ways:</p> <ul style="list-style-type: none"> <li>• using CP14 instruction</li> <li>• using the DSCR memory address</li> <li>• using the OSSRR memory address.</li> </ul> <p>CP14 instruction returns an Unpredictable value for this bit.  DSCR memory address returns the same value as InstrCompl.  OSSRR memory address returns the latched InstrCompl value, that is, the value of InstrCompl that the processor captured on the last DSCR memory address read.  If a write to the ITR APB address succeeds while in Stall or Nonblocking mode, InstrCompl_1 and InstrCompl are cleared to 0.  If a write to the DTRRX APB address or a read to the DTRTX APB address succeeds while in Fast mode, InstrCompl_1 and InstrCompl are cleared to 0.  InstrCompl is the instruction complete bit. This internal flag determines whether the processor has completed execution of an instruction issued through the APB interface.  0 = the processor is currently executing an instruction fetched from the ITR Register, reset value  1 = the processor is not currently executing an instruction fetched from the ITR Register.</p>
[23:22]	-	Reserved. UNP, SBZP.

Table 12-14 Debug Status and Control Register bit functions (continued)

Bits	Field	Function
[21:20]	DTR access mode	<p>DTR access mode. This is a read/write field. You can use this field to optimize DTR traffic between a debugger and the processor:</p> <p>b00 = Nonblocking mode, reset value  b01 = Stall mode  b10 = Fast mode  b11 = reserved.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <ul style="list-style-type: none"> <li>This field only affects the behavior of DSCR, DTR, and ITR accesses through the APB interface, and not through CP14 debug instructions.</li> <li>Nonblocking mode is the default setting. Improper use of the other modes might result in the debug access bus becoming jammed.</li> </ul> <hr/> <p>See <i>DTR access mode</i> on page 12-28 for more information.</p>
[19]	Discard imprecise abort	<p>Discard imprecise abort. This read-only bit is set to 1 while the processor is in debug state and is cleared to 0 on exit from debug state. While this bit is set to 1, the processor does not record imprecise Data Aborts. However, the sticky imprecise Data Abort bit is set to 1.</p> <p>0 = imprecise Data Aborts not discarded, reset value  1 = imprecise Data Aborts discarded.</p>
[18] <sup>a</sup>	Nonsecure state status	<p>Nonsecure state status bit:</p> <p>0 = the processor is in Secure state or the processor is in Monitor mode  1 = the processor is in Nonsecure state and is not in Monitor mode.</p>
[17] <sup>a</sup>	Secure privileged noninvasive debug disabled	<p>Secure privileged noninvasive debug disabled:</p> <p>0 = ((<b>NIDEN</b>    <b>DBGEN</b>) &amp;&amp; (<b>SPNIDEN</b>    <b>SPIDEN</b>)) is HIGH  1 = ((<b>NIDEN</b>    <b>DBGEN</b>) &amp;&amp; (<b>SPNIDEN</b>    <b>SPIDEN</b>)) is LOW.</p> <p>This value is the inverse of bit [6] of the Authentication Status Register. See <i>Authentication Status Register</i> on page 12-64.</p>
[16] <sup>a</sup>	Secure privileged invasive debug disabled	<p>Secure privileged invasive debug disabled:</p> <p>0 = (<b>DBGEN</b> &amp;&amp; <b>SPIDEN</b>) is HIGH  1 = (<b>DBGEN</b> &amp;&amp; <b>SPIDEN</b>) is LOW.</p> <p>This value is the inverse of bit [4] of the Authentication Status Register. See <i>Authentication Status Register</i> on page 12-64.</p>

Table 12-14 Debug Status and Control Register bit functions (continued)

Bits	Field	Function
[15]	Monitor debug-mode	<p>The Monitor debug-mode enable bit. This is a read/write bit.</p> <p>0 = Monitor debug-mode disabled, reset value</p> <p>1 = Monitor debug-mode enabled.</p> <p>If Halting debug-mode is enabled, bit [14] is set to 1, then the processor is in Halting debug-mode regardless of the value of bit [15]. If the external interface input <b>DBGEN</b> is LOW, DSCR[15] reads as 0. If <b>DBGEN</b> is HIGH, then the read value reverts to the programmed value.</p>
[14]	Halting debug-mode	<p>The Halting debug-mode enable bit. This is a read/write bit.</p> <p>0 = Halting debug-mode disabled, reset value</p> <p>1 = Halting debug-mode enabled.</p> <p>If the external interface input <b>DBGEN</b> is LOW, DSCR[14] reads as 0. If <b>DBGEN</b> is HIGH, then the read value reverts to the programmed value.</p>
[13]	Execute instruction enable	<p>Execute ARM instruction enable bit. This is a read/write bit.</p> <p>0 = disabled, reset value</p> <p>1 = enabled.</p> <p>If this bit is set to 1 and an ITR write succeeds, the processor fetches an instruction from the ITR for execution. If this bit is set to 1 when the processor is not in debug state, the behavior of the processor is Unpredictable.</p>
[12]	CP14 user access disable	<p>CP14 debug user access disable control bit. This is a read/write bit.</p> <p>0 = CP14 debug user access enable, reset value</p> <p>1 = CP14 debug user access disable.</p> <p>If this bit is set to 1 and a User mode process tries to access any CP14 debug registers, the Undefined Instruction exception is taken.</p>
[11]	Interrupt disable	<p>Interrupts disable bit. This is a read/write bit.</p> <p>0 = interrupts enabled, reset value</p> <p>1 = interrupts disabled.</p> <p>If this bit is set to 1, the <b>IRQ</b> and <b>FIQ</b> input signals are disabled. The external debugger can set this bit to 1 before it executes code in normal state as part of the debugging process. If this bit is set to 1, an interrupt does not take control of the program flow. For example, the debugger might use this bit to execute an OS service routine to bring a page from disk into memory. It might be undesirable to service any interrupt during the routine execution.</p>
[10]	DbgAck	<p>Debug Acknowledge bit. This is a read/write bit. If this bit is set to 1, both the <b>DBGACK</b> and <b>DBGTRIGGER</b> output signals are forced HIGH, regardless of the processor state. The external debugger can use this bit if it wants the system to behave as if the processor is in debug state. Some systems rely on <b>DBGACK</b> to determine whether the application or debugger generates the data accesses. The reset value is 0.</p>

**Table 12-14 Debug Status and Control Register bit functions (continued)**

Bits	Field	Function
[9]	-	Reserved. UNP, SBZ.
[8]	Sticky Undefined	<p>Sticky Undefined bit:</p> <p>0 = No Undefined Instruction exception occurred in debug state since the last time this bit was cleared. This is the reset value.</p> <p>1 = An Undefined Instruction exception has occurred while in debug state since the last time this bit was cleared.</p> <p>This flag detects Undefined instruction exceptions generated by instructions issued to the processor through the ITR. This bit is set to 1 when an Undefined Instruction exception occurs while the processor is in debug state. Writing a 1 to DRCCR[2] clears this bit to 0. See <i>Debug Run Control Register</i> on page 12-36.</p>
[7]	Sticky imprecise abort	<p>Sticky imprecise Data Abort bit:</p> <p>0 = no imprecise Data Aborts occurred since the last time this bit was cleared, reset value</p> <p>1 = an imprecise Data Abort occurred since the last time this bit was cleared.</p> <p>This flag detects imprecise Data Aborts triggered by instructions issued to the processor through the ITR. This bit is set to 1 when an imprecise Data Abort occurs while the processor is in debug state. Writing a 1 to DRCCR[2] clears this bit to 0. See <i>Debug Run Control Register</i> on page 12-36.</p>
[6]	Sticky precise abort	<p>Sticky precise Data Abort bit:</p> <p>0 = no precise Data Abort occurred since the last time this bit was cleared, reset value</p> <p>1 = a precise Data Abort occurred since the last time this bit was cleared.</p> <p>This flag detects precise Data Aborts generated by instructions issued to the processor through the ITR. This bit is set to 1 when a precise Data Abort occurs while the processor is in debug state. Writing a 1 to DRCCR[2] clears this bit to 0. See <i>Debug Run Control Register</i> on page 12-36.</p>

Table 12-14 Debug Status and Control Register bit functions (continued)

Bits	Field	Function
[5:2]	Entry	<p>Method of entry bits. This is a read/write field.</p> <p>b0000 = a DRCCR[0] halting debug event occurred, reset value</p> <p>b0001 = a breakpoint occurred</p> <p>b0100 = an <b>EDBGRQ</b> halting debug event occurred</p> <p>b0011 = a BKPT instruction occurred</p> <p>b0101 = a vector catch occurred</p> <p>b1000 = an OS unlock catch occurred</p> <p>b1010 = a precise watchpoint occurred</p> <p>other = reserved.</p> <p>These bits are set to indicate any of:</p> <ul style="list-style-type: none"> <li>the cause of a debug exception</li> <li>the cause for entering debug state.</li> </ul> <p>A Prefetch Abort or Data Abort handler must check the value of the CP15 Fault Status Register to determine whether a debug exception occurred and then use these bits to determine the specific debug event.</p>
[1] <sup>a</sup>	Core restarted	<p>Core restarted bit:</p> <p>0 = The processor is exiting debug state.</p> <p>1 = The processor has exited debug state. This is the reset value.</p> <p>The debugger can poll this bit to determine when the processor responds to a request to leave debug state.</p>
[0] <sup>a</sup>	Core halted	<p>Core halted bit:</p> <p>0 = The processor is in normal state. This is the reset value.</p> <p>1 = The processor is in debug state.</p> <p>The debugger can poll this bit to determine when the processor has entered debug state.</p>

- a. These bits always reflect the status of the processor and, therefore they return to their reset values if the particular reset event affects the processor. For example, a **PRESETn** event leaves these bits unchanged whereas a core reset event such as **nPORESET** or **ARESETn** sets DSCR[18] to a 0 and DSCR[1:0] to b10.

To access the Debug Status and Control Register, read CP14 c1 with:

```
MRC p14, 0, <Rd>, c0, c1, 0 ; Read Debug Status and Control Register
```

### DTR access mode

You can use the DTR access mode field to optimize data transfer between a debugger and the processor.



The DTR access mode can be one of the following:

- Nonblocking, this is the default mode
- Stall
- Fast.

In Nonblocking mode, the APB reads from the DTRTX and writes to the DTRRX and ITR are ignored if the appropriate READY flag is not set. In particular:

- writes to DTRRX are ignored if DTRRXfull\_1 is set to 1
- writes to ITR are ignored if InstrCompl\_1 is not set to 1
- reads from DTRTX are ignored and return an Unpredictable value if DTRTXfull\_1 is not set to 1.

The debugger accessing these registers must first read the DSCR, and perform any of the following:

- write to the DTRRX if the DTRRXfull\_1 flag was cleared to 0
- write to the ITR if the InstrCompl\_1 flag was set to 1
- read from the DTRTX if the DTRTXfull\_1 flag was set to 1.

Failure to read the DSCR before one of these operations leads to Unpredictable behavior.

In Stall mode, the APB accesses to DTRRX, DTRTX, and ITR stall under the following conditions:

- writes to DTRRX are stalled until DTRRXfull is cleared to 0
- writes to ITR are stalled until InstrCompl is set to 1
- reads from DTRTX are stalled until DTRTXfull is set to 1.

Fast mode is similar to Stall mode except that in Fast mode, the processor fetches an instruction from the ITR when a DTRRX write or DTRTX read succeeds. In Stall mode and Nonblocking mode, the processor fetches an instruction from the ITR when an ITR write succeeds.

### 12.4.6 Data Transfer Register

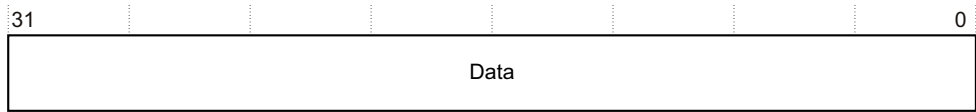
The DTR consists of two separate physical registers:

- the DTRRX (Data Transfer Register - Receive)
- the DTRTX (Data Transfer Register - Transmit).

The register accessed is dependent on the instruction used:

- writes, MCR and LDC instructions, access the DTRTX
- reads, MRC and STC instructions, access the DTRRX.

See *Debug communications channel* on page 12-95 for details on the use of these registers with the DTRRXfull and DTRTXfull flags. Figure 12-6 shows the bit arrangement of both the DTRRX and DTRTX.



**Figure 12-6 DTR Register format**

Table 12-15 shows how the bit values correspond with the DTRRX and DTRTX functions.

**Table 12-15 Data Transfer Register bit functions**

Bits	Field	Function
[31:0]	-	Data Transfer Register - receive (read-only for the CP14 interface).  <div style="text-align: center;"> <b>Note</b> </div> Reads of the DTRRX through the coprocessor interface cause the DTRRXfull flag to be cleared to 0. However, reads of the DTRRX through the APB interface do not affect this flag.
[31:0]	-	Data Transfer Register - transmit (write-only for the CP14 interface).  <div style="text-align: center;"> <b>Note</b> </div> Writes to the DTRTX through the coprocessor interface cause the DTRTXfull flag to be set to 1. However, writes to the DTRTX through the APB interface do not affect this flag.

## 12.4.7 Watchpoint Fault Address Register

The WFAR is a read/write register that holds the virtual address of the instruction that triggers the watchpoint.

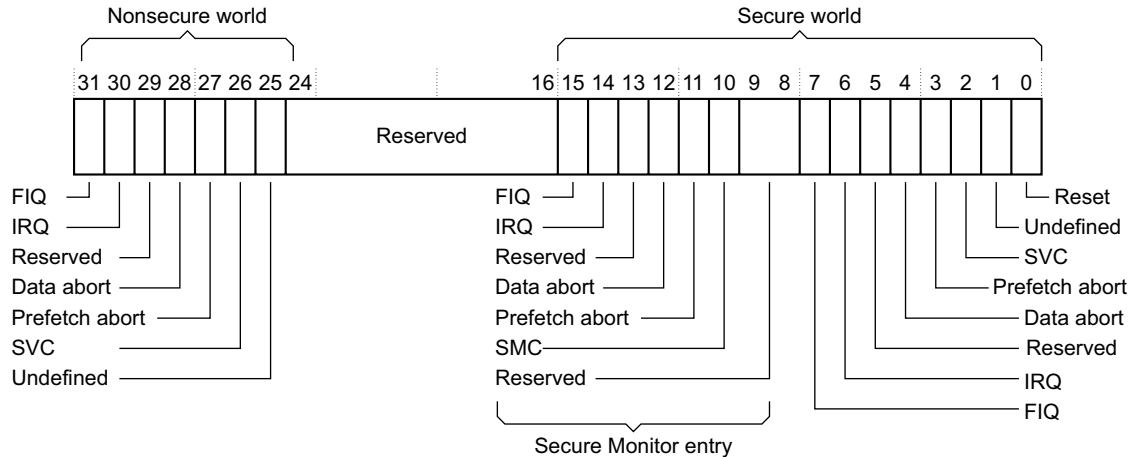
Table 12-16 shows how the bit values correspond with the WFAR functions.

**Table 12-16 Watchpoint Fault Address Register bit functions**

Bits	Field	Function
[31:1]	-	Virtual address of the watchpointed instruction. When a watchpoint occurs in ARM state, the WFAR contains the address of the instruction causing it plus 0x8. When a watchpoint occurs in Thumb state, the address is plus 0x4. The reset value is Unpredictable.
[0]	-	Reserved. UNP, SBZ.

## 12.4.8 Vector Catch Register

The processor supports efficient exception vector catching. This is controlled by the read/write Vector Catch Register as Figure 12-7 shows.



**Figure 12-7 Vector Catch Register format**

If one of the bits in this register is set to 1 and the corresponding vector is committed for execution, then the processor either enters debug state or takes a debug exception.

### Note

- Under this model, any kind of prefetch of an exception vector can trigger a vector catch, not only the ones caused by exception entries. An explicit branch to an exception vector might generate a vector catch debug event.
- Catches because of bits [15:0] are only triggered when the processor is in secure state or in Monitor mode. Catches because of bits [31:25] are only triggered when the processor is in nonsecure state and not in Monitor mode.
- If bit [28], [27], [12], [11], [4], or [3] is set to 1 while the processor is in Monitor debug mode, then the processor ignores the setting and does not generate a vector catch debug event. This prevents the processor to enter an unrecoverable state. The debugger must program these bits to zero when Monitor debug mode is selected and enabled to ensure forward-compatibility.

Table 12-17 shows the bit field definitions of the Vector Catch Register. In this table, VBAR is the CP15 Vector Base Address Register for secure, VBAR<sub>NS</sub> is CP15 Vector Base Address Register for nonsecure, and MVBAR is CP15 Monitor Vector Base Address Register.

**Table 12-17 Vector Catch Register bit functions**

Bits	Access	Normal address	High vectors address	Function
[31]	RW	VBAR <sub>NS</sub> +0x0000001C	0xFFFF001C	Vector catch enable, FIQ in Nonsecure state. The reset value is 0.
[30]	RW	VBAR <sub>NS</sub> +0x00000018	0xFFFF0018	Vector catch enable, IRQ in Nonsecure state. The reset value is 0.
[29]	R	-	-	Reserved. RAZ, SBZP.
[28]	RW	VBAR <sub>NS</sub> +0x00000010	0xFFFF0010	Vector catch enable, Data Abort in Nonsecure state. The reset value is 0.
[27]	RW	VBAR <sub>NS</sub> +0x0000000C	0xFFFF000C	Vector catch enable, Prefetch Abort in Nonsecure state. The reset value is 0.
[26]	RW	VBAR <sub>NS</sub> +0x00000008	0xFFFF0008	Vector catch enable, SVC in Nonsecure state. The reset value is 0.
[25]	RW	VBAR <sub>NS</sub> +0x00000004	0xFFFF0004	Vector catch enable, Undefined instruction in Nonsecure state. The reset value is 0.
[24:16]	R	-	-	Reserved. RAZ, SBZP.
[15]	RW	MVBAR+0x0000001C	MVBAR+0x0000001C	Vector catch enable, FIQ in Secure state. The reset value is 0.
[14]	RW	MVBAR+0x00000018	MVBAR+0x00000018	Vector catch enable, IRQ in Secure state. The reset value is 0.
[13]	R	-	-	Reserved. RAZ, SBZP.
[12]	RW	MVBAR+0x00000010	MVBAR+0x00000010	Vector catch enable, Data Abort in Secure state. The reset value is 0.
[11]	RW	MVBAR+0x0000000C	MVBAR+0x0000000C	Vector catch enable, Prefetch Abort in Secure state. The reset value is 0.
[10]	RW	MVBAR+0x00000008	MVBAR+0x00000008	Vector catch enable, SMC in Secure state. The reset value is 0.
[9:8]	R	-	-	Reserved. RAZ, SBZP.

Table 12-17 Vector Catch Register bit functions (continued)

Bits	Access	Normal address	High vectors address	Function
[7]	RW	VBAR+0x0000001C	0xFFFF001C	Vector catch enable, FIQ in Secure state. The reset value is 0.
[6]	RW	VBAR+0x00000018	0xFFFF0018	Vector catch enable, IRQ in Secure state. The reset value is 0.
[5]	R	-	-	Reserved. RAZ, SBZP.
[4]	RW	VBAR+0x00000010	0xFFFF0010	Vector catch enable, Data Abort in Secure state. The reset value is 0.
[3]	RW	VBAR+0x0000000C	0xFFFF000C	Vector catch enable, Prefetch Abort in Secure state. The reset value is 0.
[2]	RW	VBAR+0x00000008	0xFFFF0008	Vector catch enable, SVC in Secure state. The reset value is 0.
[1]	RW	VBAR+0x00000004	0xFFFF0004	Vector catch enable, Undefined instruction in Secure state. The reset value is 0.
[0]	RW	0x00000000	0xFFFF0000	Vector catch enable, Reset. The reset value is 0.

### 12.4.9 Event Catch Register

The ECR enables the external debugger or debug monitor to configure the debug logic to trigger a debug state or debug exception entry on certain events.

Figure 12-8 shows the bit arrangement of the ECR.

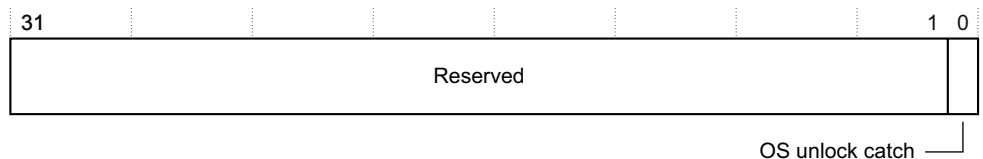


Figure 12-8 Event Catch Register format

Table 12-18 shows how the bit values correspond with the Event Catch Register functions.

**Table 12-18 Event Catch Register bit functions**

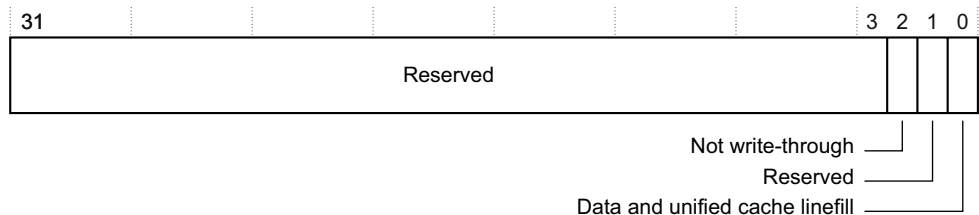
Bits	Field	Function
[31:1]	-	Reserved. RAZ, SBZP.
[0]	OS unlock catch	<p>OS unlock catch:</p> <p>0 = catch disabled, reset value</p> <p>1 = catch enabled.</p> <p>When this bit is set to 1, the debug logic generates a debug event when the OS lock state transitions from 1 to 0. This debug event might trigger a debug state entry, or might be ignored, depending on the invasive debug security configuration. The OS unlock catch debug event is a halting debug event and, therefore it cannot cause a debug exception.</p> <p>———— <b>Note</b> ————</p> <p>If you are debugging an application running on top of an OS that preserves the state of the debug unit when powering down the core, this event indicates when the debug session can continue.</p>

### 12.4.10 Debug State Cache Control Register

The DSCCR controls both L1 and L2 cache behavior while the processor is in debug state.

Figure 12-9 shows the bit arrangement of the DSCCR.

See *Cache debug* on page 12-87 for information on the usage model of the DSCCR register.



**Figure 12-9 Debug State Cache Control Register format**

Table 12-19 shows how the bit values correspond with the Debug State Cache Control Register functions.

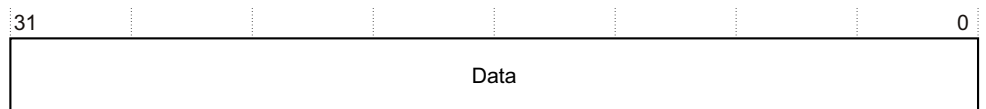
**Table 12-19 Debug State Cache Control Register bit functions**

Bits	Field	Function
[31:3]	-	Reserved. RAZ, SBZP.
[2]	Not write-through	Not write-through: 0 = force write-through behavior for regions marked as write-back in debug state, reset value 1 = normal operation of regions marked as write-back in debug state.
[1]	-	Reserved. RAZ, SBZP.
[0]	Data and unified cache linefill	Data and unified cache linefill: 0 = L1 data cache and L2 cache linefills disabled in debug state, reset value 1 = normal operation of L1 data cache and L2 cache in debug state.

### 12.4.11 Instruction Transfer Register

The ITR enables the external debugger to feed instructions into the core for execution while in debug state. The ITR is a write-only register. Reads from the ITR return an Unpredictable value.

Figure 12-10 shows the bit arrangement of the ITR.



**Figure 12-10 ITR format**

Table 12-20 shows how the bit values correspond with the Instruction Transfer Register functions.

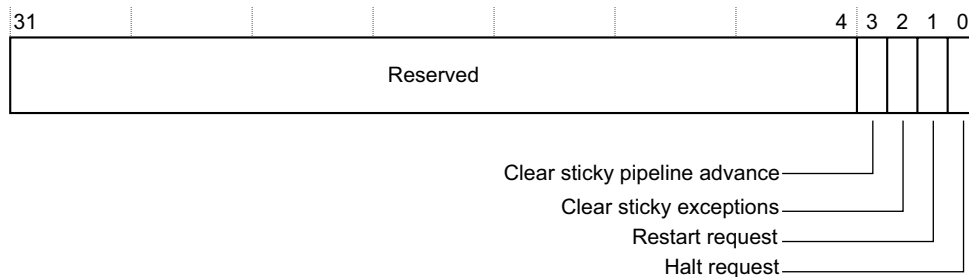
**Table 12-20 Instruction Transfer Register bit functions**

Bits	Field	Function
[31:0]	-	Indicates an ARM instruction for the processor to execute while in debug state. The reset value is Unpredictable.
<p>———— <b>Note</b> —————</p> <p>Writes to the ITR when the processor is not in debug state or the DSCR[13] execute instruction enable bit is 0 are Unpredictable.</p>		

### 12.4.12 Debug Run Control Register

The DRCCR requests the processor to enter or leave debug state. It also clears the sticky exception bits present in the DSCR to 0.

Figure 12-11 shows the bit arrangement of the DRCCR.



**Figure 12-11 Debug Run Control Register format**



Table 12-21 shows how the bit values correspond with the Debug Run Control Register functions.

**Table 12-21 Debug Run Control Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZP.
[3]	Clear sticky pipeline advance	Clear sticky pipeline advance. Writing a 1 to this bit clears DSCR[25] to 0.
[2]	Clear sticky exceptions	Clear sticky exceptions. Writing a 1 to this bit clears DSCR[8:6] to b000.
[1]	Restart request	Restart request. Writing a 1 to this bit requests that the processor leaves debug state. This request is held until the processor exits debug state. The debugger must poll DSCR[1] to determine when this request succeeds. This bit always reads as zero. Writes are ignored when the processor is not in debug state.
[0]	Halt request	Halt request. Writing a 1 to this bit triggers a halting debug event, that is, a request that the processor enters debug state. This request is held until the debug state entry occurs. The debugger must poll DSCR[0] to determine when this request succeeds. This bit always reads as zero. Writes are ignored when the processor is already in debug state.

### 12.4.13 Breakpoint Value Registers

The BVRs are registers 64-79, at offsets 0x100-0x13C. Each BVR is associated with a *Breakpoint Control Register* (BCR), for example:

- BVR0 with BCR0
- BVR1 with BCR1.

This pattern continues up to BVR15 with BCR15.

A pair of breakpoint registers, BVRn and BCRn, is called a *Breakpoint Register Pair* (BRPn).

The breakpoint value contained in this register corresponds to either an *Instruction Virtual Address* (IVA) or a context ID. Breakpoints can be set on:

- an IVA
- a context ID value
- an IVA and context ID pair.

For an IVA and context ID pair, two BRPs must be linked. A debug event is generated when both the IVA and the context ID pair match at the same time.

Table 12-22 shows how the bit values correspond with the Breakpoint Value Registers functions.

**Table 12-22 Breakpoint Value Registers bit functions**

Bits	Field	Description
[31:0]	-	Breakpoint value. The reset value is 0.

**Note**

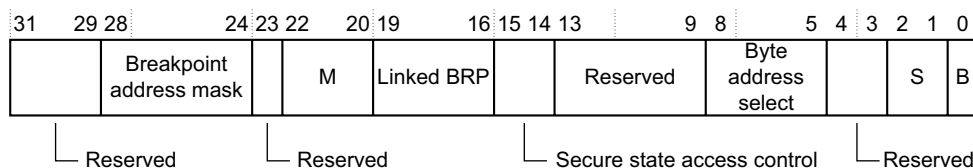
- Only BRP4 and BRP5 support context ID comparison.
- BVR0[1:0], BVR1[1:0], BVR2[1:0], and BVR3[1:0] are Should-Be-Zero or Preserved on writes and Read-As-Zero on reads because these registers do not support context ID comparisons.
- The context ID value for a BVR to match with is given by the contents of the CP15 Context ID Register. See Chapter 3 *System Control Coprocessor* for information on the Context ID Register.

#### 12.4.14 Breakpoint Control Registers

The BCR is a read/write register that contains the necessary control bits for setting:

- breakpoints
- linked breakpoints.

Figure 12-12 shows the bit arrangement of the BCRs.



**Figure 12-12 Breakpoint Control Registers format**

Table 12-23 shows how the bit values correspond with the Breakpoint Control Registers functions.

**Table 12-23 Breakpoint Control Registers bit functions**

Bits	Field	Function
[31:29]	-	Reserved. RAZ, SBZP.
[28:24]	Breakpoint address mask	Breakpoint address mask. This field is used to set a breakpoint on a range of addresses by masking lower order address bits out of the breakpoint comparison. <sup>a</sup> b00000 = no mask b00001 = reserved b00010 = reserved b00011 = 0x00000007 mask for instruction address b00100 = 0x0000000F mask for instruction address b00101 = 0x0000001F mask for instruction address . . . b11111 = 0x7FFFFFFF mask for instruction address.
[23]	-	Reserved. RAZ, SBZP.
[22:20]	M	Meaning of BVR: b000 = instruction virtual address match b001 = linked instruction virtual address match b010 = unlinked context ID b011 = linked context ID b100 = instruction virtual address mismatch b101 = linked instruction virtual address mismatch b11x = reserved.  <p style="text-align: center;">———— <b>Note</b> —————</p> BCR0[21], BCR1[21], BCR2[21], and BCR3[21] are RAZ because these registers do not have context ID comparison capability.

Table 12-23 Breakpoint Control Registers bit functions (continued)

Bits	Field	Function
[19:16]	Linked BRP	<p>Linked BRP number. The value of this field indicates another BRP to link this one with.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <ul style="list-style-type: none"> <li>• if a BRP is linked with itself, it is Unpredictable whether a breakpoint debug event is generated</li> <li>• if this BRP is linked to another BRP that is not configured for linked context ID matching, it is Unpredictable whether a breakpoint debug event is generated.</li> </ul> <hr/>
[15:14]	Secure state access control	<p>Secure state access control. This field enables the breakpoint to be conditional on the security state of the processor.</p> <p>b00 = breakpoint matches in both Secure and Nonsecure state</p> <p>b01 = breakpoint only matches in Nonsecure state</p> <p>b10 = breakpoint only matches in Secure state</p> <p>b11 = reserved.</p> <hr/>
[13:9]	-	Reserved. RAZ, SBZP.
[8:5]	Byte address select	<p>Byte address select. For breakpoints programmed to match an IVA, you must write a word-aligned address to the BVR. You can then use this field to program the breakpoint so it hits only if you access certain byte addresses.</p> <p>If you program the BRP for IVA match:</p> <p>b0000 = the breakpoint never hits</p> <p>b0011 = the breakpoint hits if any of the two bytes starting at address BVR &amp; 0xFFFFF0 is accessed</p> <p>b1100 = the breakpoint hits if any of the two bytes starting at address BVR &amp; 0xFFFFF2 is accessed</p> <p>b1111 = the breakpoint hits if any of the four bytes starting at address BVR &amp; 0xFFFF0 is accessed.</p> <p>If you program the BRP for IVA mismatch, the breakpoint hits where the corresponding IVA breakpoint does not hit, that is, the range of addresses covered by an IVA mismatch breakpoint is the negative image of the corresponding IVA breakpoint.</p> <p>If you program the BRP for context ID comparison, this field must be set to b1111. Otherwise, breakpoint and watchpoint debug events might not be generated as expected.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <p>Writing a value to BCR[8:5] where BCR[8] is not equal to BCR[7], or BCR[6] is not equal to BCR[5], has Unpredictable results.</p> <hr/>

**Table 12-23 Breakpoint Control Registers bit functions (continued)**

Bits	Field	Function
[4:3]	-	Reserved. RAZ, SBZP.
[2:1]	S	Supervisor access control. The breakpoint can be conditioned on the mode of the processor: b00 = User, System, or Supervisor b01 = privileged b10 = User b11 = any.
[0]	B	Breakpoint enable: 0 = breakpoint disabled, reset value 1 = breakpoint enabled.

- a. If BCR[28:24] is not set to b00000, then BCR[8:5] must be set to b1111. Otherwise, the behavior is Unpredictable. In addition, if BCR[28:24] is not set to b00000, then the corresponding BVR bits that are not being included in the comparison Should-Be-Zero. Otherwise, the behavior is Unpredictable. If you program this BRP for context ID comparison, you must set this field to b00000. Otherwise, the behavior is Unpredictable. There is no encoding for a full 32-bit mask but you can achieve the same effect of a *break anywhere* breakpoint by setting BCR[22] to 1 and BCR[8:5] to b0000.

**Table 12-24 Meaning of BVR bits [22:20]**

BVR[22:20]	Meaning
b000	The corresponding BVR[31:2] is compared against the IVA bus and the state of the processor against this BCR. It generates a breakpoint debug event on a joint IVA and state match.
b001	The corresponding BVR[31:2] is compared against the IVA bus and the state of the processor against this BCR. 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, context ID, and state match.
b010	The corresponding BVR[31:0] is compared against CP15 Context ID Register, c13 and the state of the processor against this BCR. This BRP is not linked with any other one. It generates a breakpoint debug event on a joint context ID and state match. For this BRP, BCR[8:5] must be set to b1111. Otherwise, it is Unpredictable whether a breakpoint debug event is generated.
b011	The corresponding BVR[31:0] is compared against CP15 Context ID Register, c13. This BRP links another BRP (of the BCR[21:20]=b01 type), or WRP (with WCR[20]=b1). They generate a breakpoint or watchpoint debug event on a joint IVA or DVA and context ID match. For this BRP, BCR[8:5] must be set to b1111, BCR[15:14] must be set to b00, and BCR[2:1] must be set to b11. Otherwise, it is Unpredictable whether a breakpoint debug event is generated.

Table 12-24 Meaning of BVR bits [22:20] (continued)

BVR[22:20]	Meaning
b100	The corresponding BVR[31:2] and BCR[8:5] are compared against the IVA bus and the state of the processor against this BCR. It generates a breakpoint debug event on a joint IVA mismatch and state match.
b101	The corresponding BVR[31:2] and BCR[8:5] are compared against the IVA bus and the state of the processor against this BCR. This BRP is linked with the one indicated by BCR[19:16] linked BRP field. It generates a breakpoint debug event on a joint IVA mismatch, state and context ID match.
b11x	Reserved. The behavior is Unpredictable.

### 12.4.15 Watchpoint Value Registers

The WVRs are registers 96-111, at offsets 0x180-0x1BC. Each WVR is associated with a *Watchpoint Control Register (WCR)*, for example:

- WVR0 with WCR0
- WVR1 with WCR1.

This pattern continues up to WVR15 with WCR15.

A pair of watchpoint registers, WVRn and WCRn, is called a *Watchpoint Register Pair (WRPn)*.

The watchpoint value contained in the WVR always corresponds to a *Data Virtual Address (DVA)* and can be set either on:

- a DVA
- a DVA and context ID pair.

For a DVA and context ID pair, a WRP and a BRP with context ID comparison capability must be linked. A debug event is generated when both the DVA and the context ID pair match simultaneously. Table 12-25 shows how the bit values correspond with the Watchpoint Value Registers functions.

Table 12-25 Watchpoint Value Registers bit functions

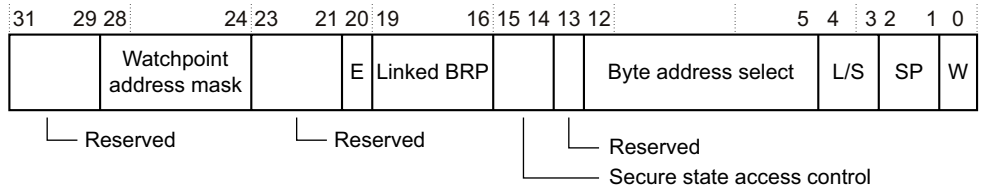
Bits	Field	Function
[31:2]	-	Watchpoint address
[1:0]	-	Reserved. RAZ, SBZP

### 12.4.16 Watchpoint Control Registers

The WCRs contain the necessary control bits for setting:

- watchpoints
- linked watchpoints.

Figure 12-13 shows the bit arrangement of the Watchpoint Control Registers.



**Figure 12-13 Watchpoint Control Registers format**

Table 12-26 shows how the bit values correspond with the Watchpoint Control Registers functions.

**Table 12-26 Watchpoint Control Registers bit functions**

Bits	Field	Function
[31:29]	-	Reserved. RAZ, SBZP.
[28:24]	Watchpoint address mask	<p>Watchpoint address mask. This field is used to watch a range of addresses by masking lower order address bits out of the watchpoint comparison:</p> <p>b00000 = no mask  b00001 = reserved  b00010 = reserved  b00011 = 0x00000007 mask for data address  b00100 = 0x0000000F mask for data address  b00101 = 0x0000001F mask for data address  .  .  .  b11111 = 0x7FFFFFFF mask for data address.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <ul style="list-style-type: none"> <li>• If WCR[28:24] is not set to b00000, then WCR[12:5] must be set to b11111111. Otherwise, the behavior is Unpredictable.</li> <li>• If WCR[28:24] is not set to b00000, then the corresponding WVR bits that are not being included in the comparison Should-Be-Zero. Otherwise, the behavior is Unpredictable.</li> <li>• To watch for a write to any byte in an 8-byte aligned object of size 8 bytes, ARM recommends that a debugger sets WCR[28:24] to b00111, and WCR[12:5] to b11111111. This is compatible with both ARMv7 debug compliant implementations that have an eight-bit WCR[12:5] and with those that have a four-bit WCR[8:5] byte address select field.</li> </ul> <hr/>
[23:21]	-	Reserved. RAZ, SBZP.
[20]	E	<p>Enable linking bit:  0 = linking disabled  1 = linking enabled.</p> <p>When this bit is set to 1, this watchpoint is linked with the context ID holding BRP selected by the linked BRP field.</p>
[19:16]	Linked BRP	<p>Linked BRP number. The binary number encoded here indicates a context ID holding BRP to link this WRP with. If this WRP is linked to a BRP that is not configured for linked context ID matching, it is Unpredictable whether a watchpoint debug event is generated.</p>



Table 12-26 Watchpoint Control Registers bit functions (continued)

Bits	Field	Function
[15:14]	Secure state access control	Secure state access control. This field enables the watchpoint to be conditioned on the security state of the processor: b00 = watchpoint matches in both Secure and Nonsecure state b01 = watchpoint only matches in Nonsecure state b10 = watchpoint only matches in Secure state b11 = reserved.
[13]	-	Reserved. RAZ, SBZP.
[12:5]	Byte address select	Byte address select. The WVR is programmed with word-aligned address. You can use this field to program the watchpoint so it only hits if certain byte addresses are accessed. For word-aligned addresses, WVRn[2]=1 indicates a 32-bit aligned address: b00000000 = the watchpoint never hits b0000xxx1 = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8)+0 is accessed b0000xx1x = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8)+1 is accessed b0000x1xx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8)+2 is accessed b00001xxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8)+3 is accessed bxxx1xxxx = UNPREDICTABLE bxx1xxxx = UNPREDICTABLE bx1xxxxx = UNPREDICTABLE b1xxxxxx = UNPREDICTABLE For double word-aligned addresses, WVRn[2]=0 indicates a 64-bit aligned address: b00000000 = the watchpoint never hits bxxxxxxx1 = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +0 is accessed bxxxxxx1x = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +1 is accessed bxxxxx1xx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +2 is accessed bxxxx1xxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +3 is accessed bxxx1xxxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +4 is accessed bxx1xxxxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +5 is accessed bx1xxxxxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +6 is accessed b1xxxxxxx = the watchpoint hits if the byte at address (WVR & 0xFFFFFFF8) +7 is accessed.

Table 12-26 Watchpoint Control Registers bit functions (continued)

Bits	Field	Function
[4:3]	L/S	<p>Load/store access. The watchpoint can be conditioned to the type of access being done:</p> <p>b00 = reserved</p> <p>b01 = load, load exclusive, or swap</p> <p>b10 = store, store exclusive or swap</p> <p>b11 = either.</p> <p>SWP and SWPB trigger a watchpoint on b01, b10, or b11. A load exclusive instruction triggers a watchpoint on b01 or b11. A store exclusive instruction triggers a watchpoint on b10 or b11 only if it passes the local monitor within the processor.<sup>a</sup></p>
[2:1]	S	<p>Privileged access control. The watchpoint can be conditioned to the privilege of the access being done:</p> <p>b00 = reserved</p> <p>b01 = privileged, match if the processor does a privileged access to memory</p> <p>b10 = User, match only on nonprivileged accesses</p> <p>b11 = either, match all accesses.</p> <p style="text-align: center;"><b>Note</b></p> <p>For all cases, the match refers to the privilege of the access, not the mode of the processor.</p>
[0]	W	<p>Watchpoint enable:</p> <p>0 = watchpoint disabled, reset value</p> <p>1 = watchpoint enabled.</p>

- a. A store exclusive that fails the local monitor does not cause a translation table walk, MMU fault, or watchpoint, see *Store-exclusive instruction* on page 8-12.

### 12.4.17 Operating System Lock Access Register

The OSLAR is a write-only register that locks the debug registers so that the APB interface returns a slave-generated error response for accesses to locked registers. This is useful for the OS to interrupt the debug session cleanly when it wants to save the state of the debug registers.

Figure 12-14 shows the bit arrangement of the OS Lock Access Register.

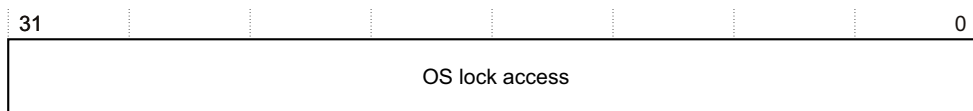


Figure 12-14 OS Lock Access Register format

Table 12-27 shows how the bit values correspond with the OS Lock Access Register functions.

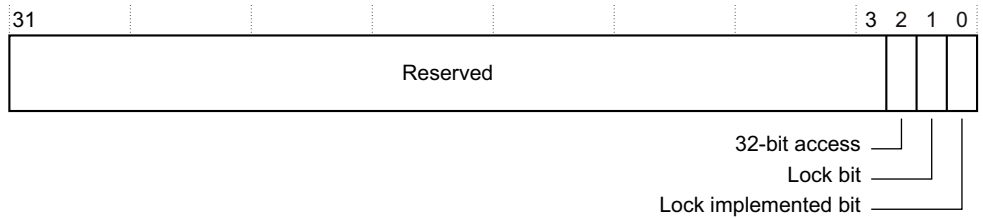
**Table 12-27 OS Lock Access Register bit functions**

Bits	Field	Function
[31:0]	OS lock access	OS lock access. Writing a 0xC5ACCE55 key locks the debug registers. Access to locked registers returns a slave-generated error response. To unlock the registers, write any other value.  <div style="text-align: center;"> <p>———— <b>Note</b> ————</p> <p>Writing the key also resets the <i>Operating System Save and Restore Register (OSSRR)</i> sequence to the beginning.</p> </div>

### 12.4.18 Operating System Lock Status Register

The OSLSR contains status information about the locked debug registers.

Figure 12-15 shows the bit arrangement of the OSLSR.



**Figure 12-15 OS Lock Status Register format**

Table 12-28 shows how the bit values correspond with the OS Lock Status Register functions.

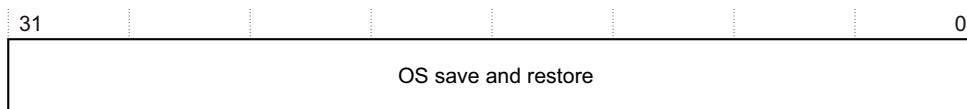
**Table 12-28 OS Lock Status Register bit functions**

Bits	Field	Function
[31:3]	-	RAZ.
[2]	32-bit access	Indicates that a 32-bit access is required to write the key to the OS Lock Access Register. This bit always reads 0.
[1]	Lock bit	Locked bit: 0 = lock is not set 1 = lock is set. Writes are ignored. On <b>Present</b> , this bit initializes to the value of <b>DBGOSLOCKINIT</b> , that is, the OS lock is set if <b>DBGOSLOCKINIT</b> is HIGH.
[0]	Lock implemented bit	Lock implemented bit. It indicates that the OS lock functionality is implemented. This bit always reads 1.

#### 12.4.19 Operating System Save and Restore Register

The OSSRR is a 32-bit read/write register that enables an operating system to save (prior to power-up) or restore (after power-down) those debug registers that reside on the core power domain while the OS lock is set.

Figure 12-16 shows the bit arrangement of the OSSRR.



**Figure 12-16 OS Save and Restore Register format**

Table 12-29 shows how the bit values correspond with the OS Save and Restore Register functions.

**Table 12-29 OS Save and Restore Register bit functions**

Bits	Field	Function
[31:0]	OS save and restore	<p>OS save and restore. A sequence of reads from this register returns the contents of all the registers that can be saved. A sequence of writes restores the saved values. The OS must initiate the sequence by writing a 0xC5ACCE55 key to the OSLAR to set the internal pointer to the starting value. This is followed by a read from the OSSRR, and then followed by a series of reads or writes. The first OSSRR read returns the length of the rest of the sequence, that is, the number of registers to be saved or restored.</p> <p>These registers are saved and restored in the following order:</p> <ol style="list-style-type: none"> <li>1. WCR1</li> <li>2. WCR0</li> <li>3. WVR1</li> <li>4. WVR0</li> <li>5. BCR5</li> <li>6. BCR4</li> <li>7. BCR3</li> <li>8. BCR2</li> <li>9. BCR1</li> <li>10. BCR0</li> <li>11. BVR5</li> <li>12. BVR4</li> <li>13. BVR3</li> <li>14. BVR2</li> <li>15. BVR1</li> <li>16. BVR0</li> <li>17. DTRTX</li> <li>18. DSCR</li> <li>19. DTRRX</li> <li>20. DSCCR</li> <li>21. VCR</li> <li>22. WFER.</li> </ol>

**Note**

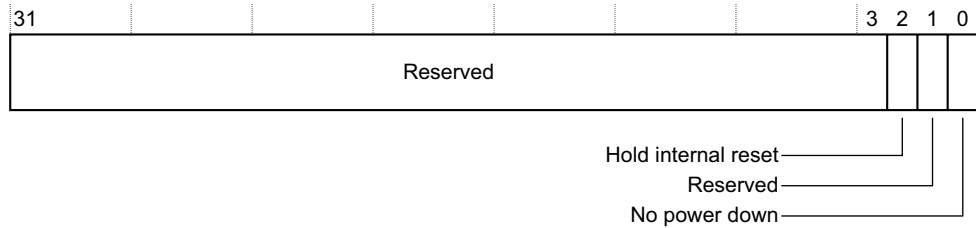
- If the OS issues a write to the OSSRR after the sequence has been initialized by writing the key to the OSLAR, the behavior is Unpredictable.

- Subsequent accesses after reading the length of the sequence must be either all reads or all writes. If the OS mixes reads and writes, the result is Unpredictable. Additionally, if the OS performs more accesses than registers are in the sequence, the result is also Unpredictable.
- This process restores only writable bits. Readable bits such as flags that reflect the processor state, are not updated. This means that, after the restore sequence, the readable bits indicate the current state of the processor rather than the state of the processor at the time the OS saved them. The only exceptions to this rule are the DSCR[30:29] and DSCR[27:26] bits, these can be restored.
- DTRRX writes and DTRTX reads through the OSSRR do not cause the APB interface to stall regardless of the value of the DSCR[22:21] field.
- The sequence can be abandoned and restarted from the beginning by writing the key again to the OSLAR. However, the results of accesses issued before it was abandoned are committed.
- If this register is read or written while the core is powered-down or the OS lock is not set, the results are Unpredictable.

### 12.4.20 Device Power Down and Reset Control Register

The PRCR is a read/write register that controls reset and power-down related functionality.

Figure 12-17 shows the bit arrangement of the PRCR.



**Figure 12-17 PRCR format**

Table 12-30 shows how the bit values correspond with the Device Power Down and Reset Control Register functions.

**Table 12-30 PRCR bit functions**

Bits	Field	Function
[31:3]	-	Reserved. RAZ, SBZP.
[2]	Hold internal reset	<p>Hold internal reset bit. This bit prevents the processor from running again before the debugger detects a power-down event and restores the state of the debug registers in the core power domain. This bit is also used to detect a reset (<b>ARESETn</b>) event. By examining PRSR[1], the debugger can determine whether a power-down or a reset event occurred. The effect of this bit is that if it is set to 1 and a processor reset occurs, <b>ARESETn</b> or <b>nPORESET</b>, then the processor behaves as if <b>ARESETn</b> is still asserted, until the debugger clears PRCR[2] to 0. This bit does not have any effect on initial system power up as <b>PRESETn</b> clears it to 0:</p> <p>0 = does not hold internal reset on power up or reset, reset value            1 = holds the processor nondebug logic in reset on power up or reset until this bit is cleared to 0.</p>
[1]	-	Reserved. RAZ, SBZP.
[0]	No power down	<p>No power down. When set to 1, the <b>DBGNOPWRDWN</b> output signal is HIGH. This output is connected to the system power controller and is interpreted as a request to operate in emulate mode. In this mode, the core and ETM are not actually powered down when requested by software or hardware handshakes. This mode is useful when debugging applications on top of working operating systems:</p> <p>0 = <b>DBGNOPWRDWN</b> is LOW, reset value            1 = <b>DBGNOPWRDWN</b> is HIGH.</p>

#### 12.4.21 Device Power Down and Reset Status Register

The PRSR is a read-only register that provides information about the reset and power-down state of the processor.

Figure 12-18 on page 12-52 shows the bit arrangement of the PRSR.

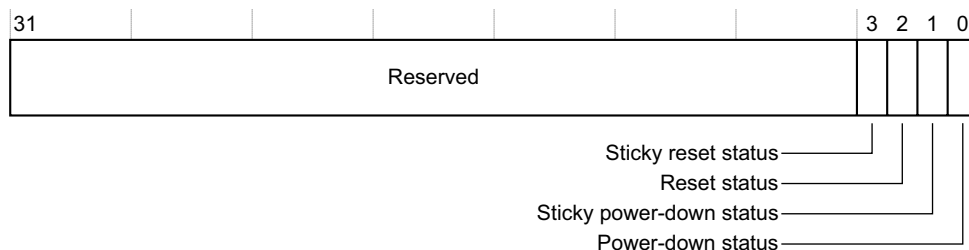


Figure 12-18 PRSR format

Table 12-31 shows how the bit values correspond with the PRSR functions.

Table 12-31 PRSR bit functions

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZP.
[3]	Sticky reset status	Sticky reset status bit. This bit is cleared to 0 on read: 0 = the processor has not been reset since the last time this register was read 1 = the processor has been reset since the last time this register was read. This sticky bit is set to 1 when either <b>ARESETn</b> or <b>nPORESET</b> is asserted. This sticky bit is set to 0 when <b>PRESETn</b> is asserted. If both <b>PRESETn</b> and <b>ARESETn</b> or <b>nPORESET</b> are asserted at the same time, this bit is set to an Unpredictable value.
[2]	Reset status	Reset status bit: 0 = the processor is not currently held in reset 1 = the processor is currently held in reset. This bit reads 1 when either <b>ARESETn</b> or <b>nPORESET</b> is asserted.
[1]	Sticky power-down status	Sticky power-down status bit. This bit is cleared to 0 on read: 0 = the processor has not powered down since the last time this register was read 1 = the processor has powered down since the last time this register was read. This is the reset value.
[0]	Power-down status	Power-down status bit. This status bit reflects the invert value of the <b>DBGPWRDWNREQ</b> input: 0 = the core is not powered up 1 = the core is powered up.



---

**Note**

---

On system reset, PRSR[1] resets to 1. Table 12-6 on page 12-15 specified that if PRSR[1] is set to 1, then accessing any register in the core power domain results in an error response. For these reasons, the debugger cannot access any register in the core power domain unless the debugger clears PRSR[1] to 0.

---

## 12.5 Management registers

The Management registers define the standardized set of registers that is implemented by all CoreSight components. These registers are described in this section.

Table 12-32 shows the contents of the Management registers for the debug unit.

**Table 12-32 Management registers**

Offset	Register number	Access	Mnemonic	Power domain	Description
0xD00-0xDFC	832-895	R	-	Debug	Processor Identifier Registers. See <i>Processor ID Registers</i> on page 12-55.
0xE00-0xEF0	854-956	R	-	-	RAZ.
0xEF4	957	RW	ITCTRL-IOC	Core	Integration Internal Output Control Register. See <i>Integration Internal Output Control Register</i> on page 12-56.
0xEF8	958	RW	ITCTRL-EOC	Core	Integration External Output Control Register. See <i>Integration External Output Control Register</i> on page 12-58.
0xEFC	959	R	ITCTRL-IS	Core	Integration Input Status Register. See <i>Integration Input Status Register</i> on page 12-59.
0xF00	960	RW	ITCTRL	Core	Integration Mode Control Register. See <i>Integration Mode Control Register</i> on page 12-61.
0xF04-0xF9C	961-999	R	-	Debug	RAZ, reserved for Management Register expansion.
0xFA0	1000	RW	CLAIMSET	Debug	Claim Tag Set Register. See <i>Claim Tag Set Register</i> on page 12-61.
0xFA4	1001	RW	CLAIMCLR	Debug	Claim Tag Clear Register. See <i>Claim Tag Clear Register</i> on page 12-62.
0xFA8-0xFBC	1002-1003	R	-	-	RAZ.
0xFB0	1004	W	LOCKACCESS	Debug	Lock Access Register. See <i>Lock Access Register</i> on page 12-63.
0xFB4	1005	R	LOCKSTATUS	Debug	Lock Status Register. See <i>Lock Status Register</i> on page 12-63.

Table 12-32 Management registers (continued)

Offset	Register number	Access	Mnemonic	Power domain	Description
0xFB8	1006	R	AUTHSTATUS	Debug	Authentication Status Register. See <i>Authentication Status Register</i> on page 12-64.
0xFBC-0xFC4	1007-1009	R	-	-	RAZ.
0xFC8	1010	R	DEVID	Debug	RAZ, reserved for Device Identifier.
0xFCC	1011	R	DEVTYPE	Debug	Device Type Register. See <i>Device Type Register</i> on page 12-65.
0xFD0-0xFFC	1012-1023	R	-	Debug	Identification Registers. See <i>Identification Registers</i> on page 12-66.

### 12.5.1 Processor ID Registers

The Processor ID Registers are read-only registers that return the same values as the corresponding CP15 registers.

Table 12-33 shows the offset value, register number, mnemonic, and description that are associated with each Process ID Register.

Table 12-33 Processor Identifier Registers

Offset	Register number	Mnemonic	Function
0xD00	832	CPUID	Main ID Register, see <i>c0, Main ID Register</i> on page 3-25
0xD04	833	CTYPR	Cache Type Register, see <i>c0, Cache Type Register</i> on page 3-26
0xD08	834	-	Reserved, RAZ
0xD0C	835	TTYPR	TLB Type Register, see <i>c0, TLB Type Register</i> on page 3-28
0xD10	836	-	Main ID Register, see <i>c0, Main ID Register</i> on page 3-25
0xD14	837	-	Reserved, RAZ
0xD18	838	-	Main ID Register, see <i>c0, Main ID Register</i> on page 3-25
0xD1C	839	-	Main ID Register, see <i>c0, Main ID Register</i> on page 3-25
0xD20	840	ID_PFR0	Processor Feature Register 0, see <i>c0, Processor Feature Register 0</i> on page 3-30

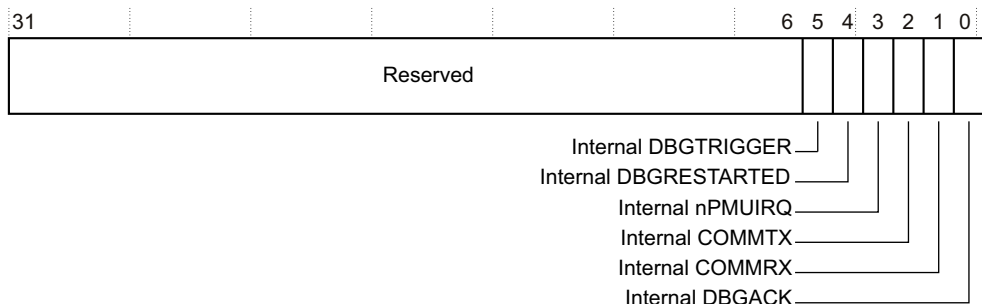
Table 12-33 Processor Identifier Registers (continued)

Offset	Register number	Mnemonic	Function
0xD24	841	ID_PFR1	Processor Feature Register 1, see <i>c0, Processor Feature Register 1</i> on page 3-31
0xD28	842	ID_DFR0	Debug Feature Register 0, see <i>c0, Debug Feature Register 0</i> on page 3-33
0xD2C	843	ID_AFR0	Auxiliary Feature Register 0, see <i>c0, Auxiliary Feature Register 0</i> on page 3-34
0xD30	844	ID_MMFR0	Memory Model Feature Register 0, see <i>c0, Memory Model Feature Register 0</i> on page 3-35
0xD34	845	ID_MMFR1	Memory Model Feature Register 1, see <i>c0, Memory Model Feature Register 1</i> on page 3-37
0xD38	846	ID_MMFR2	Memory Model Feature Register 2, see <i>c0, Memory Model Feature Register 2</i> on page 3-39
0xD3C	847	ID_MMFR3	Memory Model Feature Register 3, see <i>c0, Memory Model Feature Register 3</i> on page 3-41
0xD40	848	ID_ISAR0	Instruction Set Attributes Register 0, see <i>c0, Instruction Set Attributes Register 0</i> on page 3-43
0xD44	849	ID_ISAR1	Instruction Set Attributes Register 1, see <i>c0, Instruction Set Attributes Register 1</i> on page 3-44
0xD48	850	ID_ISAR2	Instruction Set Attributes Register 2, see <i>c0, Instruction Set Attributes Register 2</i> on page 3-46
0xD4C	851	ID_ISAR3	Instruction Set Attributes Register 3, see <i>c0, Instruction Set Attributes Register 3</i> on page 3-48
0xD50	852	ID_ISAR4	Instruction Set Attributes Register 4, see <i>c0, Instruction Set Attributes Register 4</i> on page 3-50
0xD54	853	ID_ISAR5	Instruction Set Attributes Register 5, see <i>c0, Instruction Set Attributes Registers 5-7</i> on page 3-52

## 12.5.2 Integration Internal Output Control Register

When the processor is in integration mode, you can use the read/write Integration Internal Output Control Register to drive certain debug unit outputs to determine how they are connected to the *Cross Triggered Interface (CTI)*.

Figure 12-19 shows the bit arrangement of the Integration Internal Output Control Register.



**Figure 12-19 Integration Internal Output Control Register format**

Table 12-34 shows how the bit values correspond with the Integration Internal Output Control Register functions.

**Table 12-34 Integration Internal Output Control Register bit functions**

Bits	Field	Function
[31:6]	-	RAZ for reads, SBZP for writes.
[5]	Internal DBGTRIGGER	Internal <b>DBGTRIGGER</b> . This bit drives the internal signal that goes from the debug unit to the CTI to indicate early entry to the debug state. The reset value is 0.
[4]	Internal DBGRESTARTED	Internal <b>DBGRESTARTED</b> . This bit drives the internal signal that goes from the debug unit to the CTI to acknowledge success of a debug restart command. The reset value is 0.
[3]	Internal nPMUIRQ	Internal <b>nPMUIRQ</b> . This bit drives the internal signal equivalent to <b>nPMUIRQ</b> that goes from the debug unit to the CTI. If this bit is set to 1, the corresponding internal <b>nPMUIRQ</b> signal is asserted, that is, cleared to 0. The reset value is 0.
[2]	Internal COMMTX	Internal <b>COMMTX</b> . This bit drives the internal signal equivalent to <b>COMMTX</b> that goes from the debug unit to the CTI. The reset value is 0.
[1]	Internal COMMRX	Internal <b>COMMRX</b> . This bit drives the internal signal equivalent to <b>COMMRX</b> that goes from the debug unit to the CTI. The reset value is 0.
[0]	Internal DBGACK	Internal <b>DBGACK</b> . This bit drives the internal signal equivalent to <b>DBGACK</b> that goes from the debug unit to the CTI. The reset value is 0.

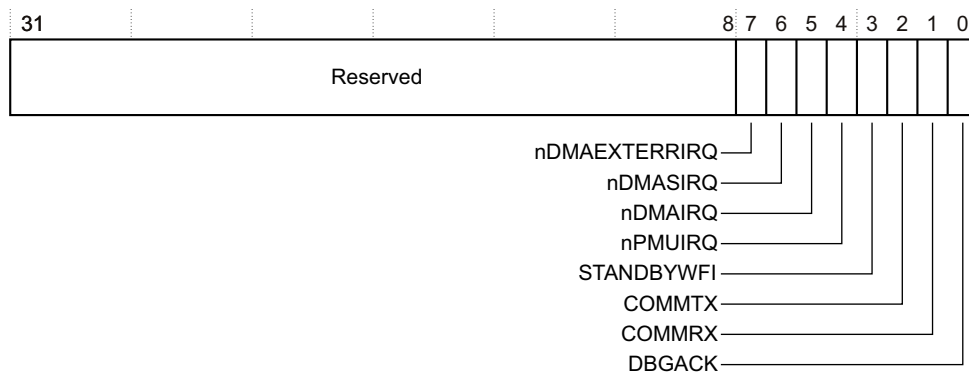
———— **Note** ————

Both the **DBGTRIGGER** and **DBGACK** signals are asserted on entry to debug state. The only difference is that **DBGTRIGGER** is asserted before the implicit *Data Synchronization Barrier* (DSB) associated with the debug state entry, while **DBGACK** is asserted after the DSB.

### 12.5.3 Integration External Output Control Register

When the processor is in integration mode, you can use the read/write Integration External Output Control Register to drive certain debug unit outputs to determine how they are connected to other parts of the system.

Figure 12-20 shows the bit arrangement of the Integration External Output Control Register.



**Figure 12-20 Integration External Output Control Register format**

Table 12-35 shows how the bit values correspond with the Integration External Output Control Register functions.

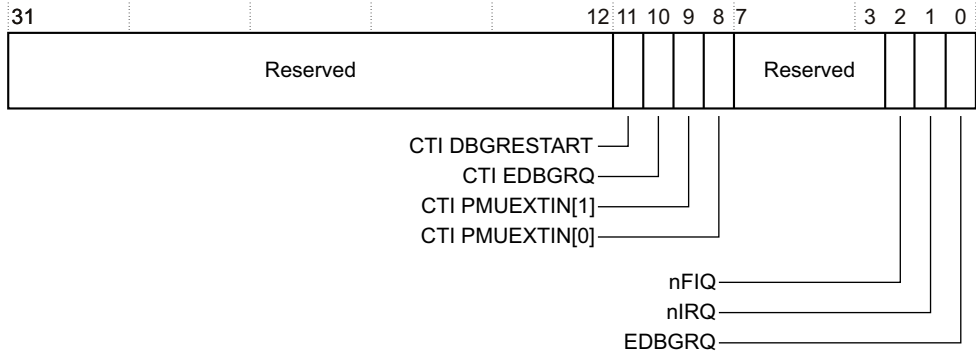
**Table 12-35 Integration External Output Control Register bit functions**

Bits	Field	Function
[31:8]	-	Reserved. RAZ, SBZP.
[7]	nDMAEXTERRIQ	<b>nDMAEXTERRIQ</b> . This signal drives the <b>nDMAEXTERRIQ</b> output. If this bit is set to 1, the corresponding internal <b>nDMAEXTERRIQ</b> signal is asserted, that is, cleared to 0. The reset value is 0.
[6]	nDMASIRQ	<b>nDMASIRQ</b> . This signal drives the <b>nDMASIRQ</b> output. If this bit is set to 1, the corresponding internal <b>nDMASIRQ</b> signal is asserted, that is, cleared to 0. The reset value is 0.
[5]	nDMAIRQ	<b>nDMAIRQ</b> . This signal drives the <b>nDMAIRQ</b> output. If this bit is set to 1, the corresponding internal <b>nDMAIRQ</b> signal is asserted, that is, cleared to 0. The reset value is 0.
[4]	nPMUIRQ	<b>nPMUIRQ</b> . This signal drives the <b>nPMUIRQ</b> output. If this bit is set to 1, the corresponding internal <b>nPMUIRQ</b> signal is asserted, that is, cleared to 0. The reset value is 0.
[3]	STANDBYWFI	<b>STANDBYWFI</b> . This signal drives the <b>STANDBYWFI</b> output. The reset value is 0.
[2]	COMMTX	<b>COMMTX</b> . This signal drives the <b>COMMTX</b> output. The reset value is 0.
[1]	COMMRX	<b>COMMRX</b> . This signal drives the <b>COMMRX</b> output. The reset value is 0.
[0]	DBGACK	<b>DBGACK</b> . This signal drives the <b>DBGACK</b> output. The reset value is 0.

#### 12.5.4 Integration Input Status Register

When the processor is in integration mode, you can use the read-only Integration Input Status Register to read the state of the debug unit inputs to determine how they are connected to the CTI and to other parts of the system.

Figure 12-21 on page 12-60 shows the bit arrangement of the Integration Input Status Register.



**Figure 12-21 Integration Input Status Register format**

Table 12-36 shows how the bit values correspond with the Integration Input Status Register functions.

**Table 12-36 Integration Input Status Register bit functions**

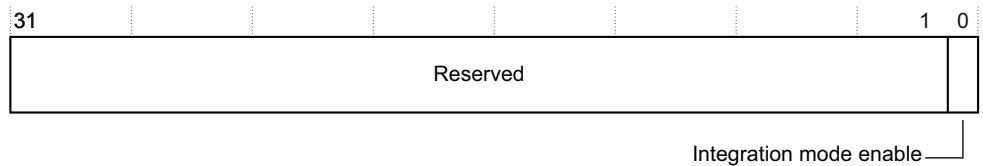
Bits	Field	Function
[31:12]	-	Reserved. RAZ, SBZP.
[11]	CTI DBGRESTART	CTI debug restart bit. This field reads the state of the debug restart input coming from the CTI into the Performance Monitoring Unit.
[10]	CTI EDBGQR	CTI debug request bit. This field reads the state of the debug request input coming from the CTI into the Performance Monitoring Unit.
[9]	CTI PMUEXTIN[1]	CTI <b>PMUEXTIN[1]</b> signal. This field reads the state of the <b>PMUEXTIN[1]</b> input coming from the CTI into the Performance Monitoring Unit.
[8]	CTI PMUEXTIN[0]	CTI <b>PMUEXTIN[0]</b> signal. This field reads the state of the <b>PMUEXTIN[0]</b> input coming from the CTI into the Performance Monitoring Unit.
[7:3]	-	Reserved. RAZ, SBZP.
[2]	nFIQ	<b>nFIQ</b> . This field reads 1 when the <b>nFIQ</b> input is asserted, that is, cleared to 0.
[1]	nIRQ	<b>nIRQ</b> . This field reads 1 when the <b>nIRQ</b> input is asserted, that is, cleared to 0.
[0]	EDBGRQ	<b>EDBGRQ</b> . This field reads the state of the <b>EDBGRQ</b> input.



## 12.5.5 Integration Mode Control Register

The read/write Integration Mode Control Register enables the processor to switch from a functional mode which is the default, into integration mode, where the inputs and outputs of the device can be directly controlled for integration testing or topology detection. When the processor is in this mode, you can use the Integration Internal Output Control Register or the Integration External Output Control Register to drive output values. You can use the Integration Input Status Register to read input values.

Figure 12-22 shows the bit arrangement of the Integration Mode Control Register.



**Figure 12-22 Integration Mode Control Register format**

Table 12-37 shows how the bit values correspond with the Integration Mode Control Register functions.

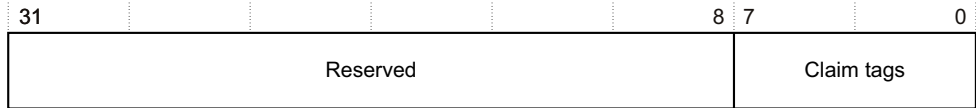
**Table 12-37 Integration Mode Control Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved. RAZ, SBZP.
[0]	Integration mode enable	Integration mode enable bit: 0 = normal operation, reset value 1 = integration mode enabled. When this bit is set to 1, the processor reverts into integration mode to enable integration testing or topology detection.

## 12.5.6 Claim Tag Set Register

Bits in the Claim Tag Set Register do not have any specific functionality. The external debugger and debug monitor set these bits to lay claims on debug resources.

Figure 12-23 on page 12-62 shows the bit arrangement of the Claim Tag Set Register.



**Figure 12-23 Claim Tag Set Register format**

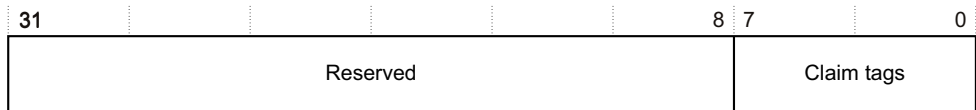
Table 12-38 shows how the bit values correspond with the Claim Tag Set Register functions.

**Table 12-38 Claim Tag Set Register bit functions**

Bits	Field	Function
[31:8]	-	Reserved. RAZ, SBZP.
[7:0]	Claim tags	Indicates the claim tags. Writing 1 to a bit in this register sets that particular claim. You can read the claim status at the Claim Tag Clear Register. For example, if you write 1 to bit [3] of this register, bit [3] of the Claim Tag Clear Register is read as 1. Writing 0 to a specific claim tag bit has no effect. This register always reads 0xFF, indicating that up to eight claims can be set.

### 12.5.7 Claim Tag Clear Register

The read/write Claim Tag Clear Register is used to read the claim status on debug resources. Figure 12-24 shows the bit arrangement of the Claim Tag Clear Register.



**Figure 12-24 Claim Tag Clear Register format**

Table 12-39 shows how the bit values correspond with the Claim Tag Clear Register functions.

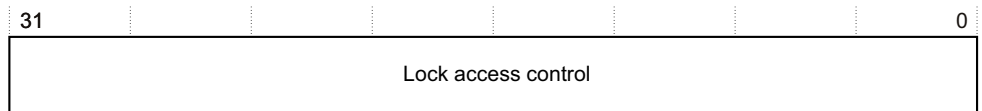
**Table 12-39 Claim Tag Clear Register bit functions**

Bits	Field	Function
[31:8]	-	Reserved. RAZ, SBZP.
[7:0]	Claim tags	Indicates the claim tag status. Writing 1 to a specific bit clears the corresponding claim tag to 0. Reading this register returns the current claim tag value. For example, if you write 1 to bit [3] of this register, it is read as 0. The reset value is 0.

## 12.5.8 Lock Access Register

The Lock Access Register is a write-only register that controls writes to the debug registers. The purpose of the Lock Access Register is to reduce the risk of accidental corruption to the contents of the debug registers. It does not prevent all accidental or malicious damage. Because the state of the Lock Access Register is in the debug power domain, it is not lost when the core powers down.

Figure 12-25 shows the bit arrangement of the Lock Access Register.



**Figure 12-25 Lock Access Register format**

Table 12-40 shows how the bit values correspond with the Lock Access Register functions.

**Table 12-40 Lock Access Register bit functions**

Bits	Field	Function
[31:0]	Lock access control	Lock access control. To unlock the debug registers, write a 0xC5ACCE55 key to this register. To lock the debug registers, write any other value. Accesses to locked debug registers are ignored.

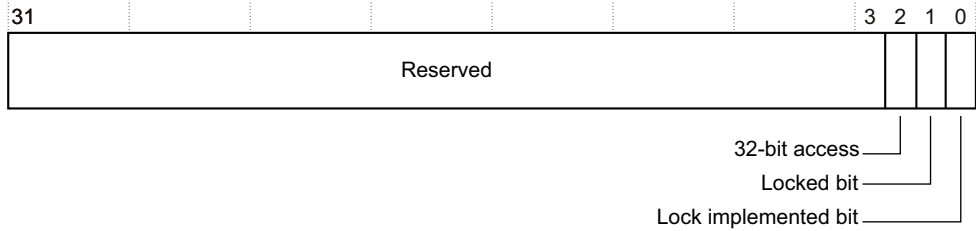
———— **Note** ————

You can only access the Lock Access Register when the **PADDR31** input is LOW. Writes are ignored when **PADDR31** is HIGH.

## 12.5.9 Lock Status Register

The Lock Status Register is a read-only register that returns the current lock status of the debug registers.

Figure 12-26 on page 12-64 shows the bit arrangement of the Lock Status Register.



**Figure 12-26 Lock Status Register format**

Table 12-41 shows how the bit values correspond with the Lock Status Register functions.

**Table 12-41 Lock Status Register bit functions**

Bits	Field	Function
[31:3]	-	RAZ.
[2]	32-bit access	32-bit access. Indicates that a 32-bit access is required to write the key to the Lock Access Register. This bit always reads 0.
[1]	Locked bit	Locked bit: 0 = writes are permitted 1 = writes are ignored, reset value.
[0]	Lock implemented bit	Lock implemented bit. Indicates that the lock functionality is implemented. This bit always reads 1.

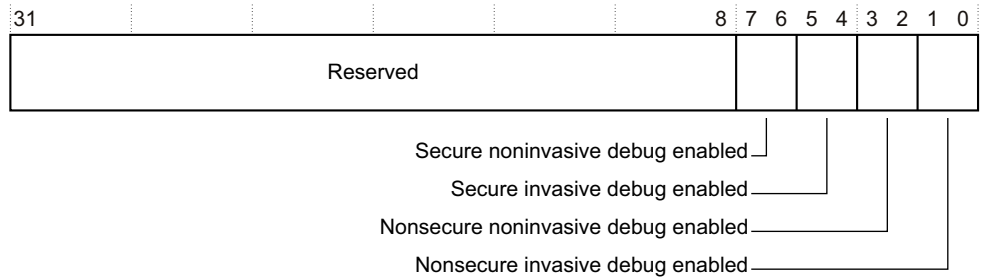
**Note**

This lock has no effect on accesses initiated by the debugger. Therefore, if **PADDR31** is HIGH, all the bits in this register read 0.

## 12.5.10 Authentication Status Register

The Authentication Status Register is a read-only register that reads the current values of the configuration inputs that determine the debug permission level.

Figure 12-27 on page 12-65 shows the bit arrangement of the Authentication Status Register.



**Figure 12-27 Authentication Status Register format**

Table 12-42 shows how the bit values correspond with the Authentication Status Register functions.

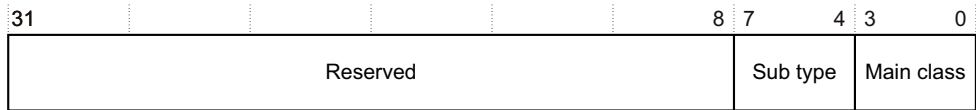
**Table 12-42 Authentication Status Register bit functions**

Bits	Field	Value	Function
[31:8]	-	-	RAZ
[7]	Secure noninvasive debug enabled	b1	Secure noninvasive debug enable field
[6]		( <b>DBGEN</b>    <b>NIDEN</b> ) && ( <b>SPIDEN</b>    <b>SPNIDEN</b> )	
[5]	Secure invasive debug enabled	b1	Secure invasive debug enable field
[4]		<b>DBGEN</b> && <b>SPIDEN</b>	
[3]	Nonsecure noninvasive debug enabled	b1	Nonsecure noninvasive debug enable field
[2]		<b>DBGEN</b>    <b>NIDEN</b>	
[1]	Nonsecure invasive debug enabled	b1	Nonsecure invasive debug enable field
[0]		<b>DBGEN</b>	

### 12.5.11 Device Type Register

The Device Type Register is a read-only register that indicates the type of debug component.

Figure 12-28 on page 12-66 shows the bit arrangement of the Device Type Register.



**Figure 12-28 Device Type Register format**

Table 12-43 shows how the bit values correspond with the Device Type Register functions.

**Table 12-43 Device Type Register bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:4]	Sub type	Indicates that the sub-type of the processor is <i>core</i> . This value is 0x1.
[3:0]	Main class	Indicates that the main class of the processor is <i>debug logic</i> . This value is 0x5.

## 12.5.12 Identification Registers

The Identification Registers are read-only registers that consist of the Peripheral Identification Registers and the Component Identification Registers. The Peripheral Identification Registers provide standard information required by all CoreSight components. Only bits [7:0] of each register are used, the remaining bits Read-As-Zero.

The Component Identification Registers identify the processor as a CoreSight component. Only bits [7:0] of each register are used, the remaining bits Read-As-Zero. The values in these registers are fixed.

Table 12-44 shows the offset value, register number, and description that are associated with each Peripheral Identification Register.

**Table 12-44 Peripheral Identification Registers**

Offset	Register number	Function
0xFD0	1012	Peripheral Identification Register 4
0xFD4	1013	Reserved
0xFD8	1014	Reserved
0xFDC	1015	Reserved
0xFE0	1016	Peripheral Identification Register 0

**Table 12-44 Peripheral Identification Registers (continued)**

Offset	Register number	Function
0xFE4	1017	Peripheral Identification Register 1
0xFE8	1018	Peripheral Identification Register 2
0xFEC	1019	Peripheral Identification Register 3

Table 12-45 shows fields that are in the Peripheral Identification Registers.

**Table 12-45 Fields in the Peripheral Identification Registers**

Field	Size	Function
4KB Count	4 bits	Indicates the $\text{Log}_2$ of the number of 4KB blocks that the processor occupies. The debug registers occupy a single 4KB block, therefore this field is always 0x0.
JEP106	4+7 bits	Identifies the designer of the processor. This field consists of a 4-bit continuation code and a 7-bit identity code. Because the processor is designed by ARM, the continuation code is 0x4 and the identity code is 0x3B.
Part number	12 bits	Indicates the part number of the processor. The part number for the processor is 0xC08.
Revision	4 bits	Indicates the major and minor revision of the product. The major revision contains functionality changes and the minor revision contains bug fixes for the product. The revision number starts at 0x0 and increments by 1 at both major and minor revisions.
RevAnd	4 bits	Indicates the manufacturer revision number. This number starts at 0x0 and increments by the integrated circuit manufacturer on metal fixes. For the processor, the initial value is 0x0 but can be changed by the manufacturer.
Customer modified	4 bits	For the processor, this value is 0x0.

Table 12-46 shows how the bit values correspond with the Peripheral ID Register 0 functions.

**Table 12-46 Peripheral ID Register 0 bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:0]	-	Indicates bits [7:0] of the part number for the processor. This value is 0x08.

Table 12-47 shows how the bit values correspond with the Peripheral ID Register 1 functions.

**Table 12-47 Peripheral ID Register 1 bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:4]	-	Indicates bits [3:0] of the JEDEC JEP106 Identity Code. This value is 0xB.
[3:0]	-	Indicates bits [11:8] of the part number for the processor. This value is 0xC.

Table 12-48 shows how the bit values correspond with the Peripheral ID Register 2 functions.

**Table 12-48 Peripheral ID Register 2 bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:4]	-	Indicates the revision number for the processor. This value changes based on the product major and minor revision. This value is set to b0110.
[3]	-	This field is always set to 1.
[2:0]	-	Indicates bits [6:4] of the JEDEC JEP106 Identity Code. This value is set to b011.

Table 12-49 shows how the bit values correspond with the Peripheral ID Register 3 functions.

**Table 12-49 Peripheral ID Register 3 bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:4]	-	Indicates the manufacturer revision number. This value changes based on the manufacturer metal fixes. This value is set to 0x1.
[3:0]	-	For the processor, this value is set to 0x0.



Table 12-50 shows how the bit values correspond with the Peripheral ID Register 4 functions.

**Table 12-50 Peripheral ID Register 4 bit functions**

Bits	Field	Function
[31:8]	-	RAZ.
[7:4]	-	Indicates the number of blocks occupied by the processor. This field is always set to 0x0.
[3:0]	-	Indicates the JEDEC JEP106 Continuation Code. For the processor, this value is 0x4.

Table 12-51 shows the offset value, register number, and value that are associated with each Component Identification Register.

**Table 12-51 Component Identification Registers**

Offset	Register number	Value	Function
0xFF0	1020	0x0000000D	Component Identification Register 0
0xFF4	1021	0x00000090	Component Identification Register 1
0xFF8	1022	0x00000005	Component Identification Register 2
0xFFC	1023	0x000000B1	Component Identification Register 3

## 12.6 Debug events

A debug event is any of the following:

- *Software debug event*
- *Halting debug event* on page 12-71.

A processor responds to a debug event in one of the following ways:

- ignores the debug event
- takes a debug exception
- enters debug state.

### 12.6.1 Software debug event

A software debug event is any of the following:

- A watchpoint debug event. This occurs when:
  - The *D-side Virtual Address (DVA)* for a load or store matches the watchpoint value. Memory hints and cache operations do not trigger watchpoints.
  - All the conditions of the WCR match.
  - The watchpoint is enabled.
  - The linked context ID-holding BRP, if any, is enabled and its value matches the context ID in CP15 c13. See Chapter 3 *System Control Coprocessor*.
  - The instruction that initiated the memory access is committed for execution. Watchpoint debug events are only generated if the instruction passes its condition code.
- A breakpoint debug event. This occurs when:
  - An instruction is fetched and the *I-side Virtual Address (IVA)* present in the instruction bus matched the breakpoint value.
  - At the same time the instruction is fetched, all the conditions of the BCR for linked or unlinked IVA-based breakpoint generation matched the I-side control signals.
  - The breakpoint is enabled.
  - At the same time the instruction is fetched, the linked context ID-holding BRP, if any, is enabled and its value matched the context ID in CP15 c13.
  - The instruction is committed for execution. These debug events are generated whether the instruction passes or fails its condition code.

- A breakpoint debug event also occurs when:
  - An instruction is fetched and the CP15 Context ID Register c13 matched the breakpoint value.
  - At the same time the instruction is fetched, all the conditions of the BCR for unlinked context ID breakpoint generation matched the I-side control signals.
  - The breakpoint is enabled.
  - The instruction is committed for execution. These debug events are generated whether the instruction passes or fails its condition code.
- A BKPT debug event. This occurs when a BKPT instruction is committed for execution. BKPT is an unconditional instruction.
- A vector catch debug event. This occurs when:
  - An instruction is prefetched and the IVA matched a vector location address. This includes any kind of prefetches, not only the ones because of exception entry.
  - At the same time the instruction is fetched, the corresponding bit of the VCR is set to 1, that is, vector catch enabled.
  - Either the vector is not one of the Prefetch Abort or Data Abort vectors, or Halting debug mode is enabled.
  - The instruction is committed for execution. These debug events are generated whether the instruction passes or fails its condition code.

## 12.6.2 Halting debug event

The debugger or the system can cause the core to enter into debug state by triggering any of the following halting debug events:

- assertion of the external debug request signal, **EDBGRQ**
- write to the **DRCR[0]** Halt Request control bit
- detection of the OS unlock catch event
- assertion of the Cross Trigger Interface debug request signal.

If **EDBGRQ** or CTI debug request is asserted while **DBGEN** is HIGH but invasive debug is not permitted, the devices that assert these signals must hold them until the processor enters debug state, that is, until **DBGACK** is asserted. Otherwise, the behavior of the processor is Unpredictable. For **DRCR[0]** and OS unlock catch halting debug events, the processor records them internally until it is in a state and mode where they can be acknowledged.

### 12.6.3 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 12-78 for information on how the processor behaves while in debug state. When the processor is in Monitor debug-mode, Prefetch Abort and Data Abort vector catch debug events are ignored. All other software debug events generate a debug exception such as Data Abort for watchpoints, and Prefetch Abort for anything else.

When debug is disabled, the BKPT instruction generates a debug exception, Prefetch Abort. All other software debug events are ignored.

When **DBGEN** is LOW, debug is disabled regardless of the value of DSCR[15:14].

Table 12-52 shows the behavior of the processor on debug events.

**Table 12-52 Processor behavior on debug events**

DBGEN	DSCR[15:14]	Debug mode	Action on software debug event	Action on halting debug event
0	bxx	Debug disabled	Ignore or Prefetch Abort <sup>a</sup>	Ignore
1	b00	None	Ignore or Prefetch Abort <sup>a</sup>	Debug state entry
1	bx1	Halting	Debug state entry	Debug state entry
1	b10	Monitor	Debug exception or Ignore <sup>b</sup>	Debug state entry

a. The BKPT instruction generates a Prefetch Abort. All other software debug events are ignored.

b. Prefetch Abort and Data Abort vector catch debug events are ignored in Monitor debug-mode. All other software debug events generate a debug exception.

### 12.6.4 Debug event priority

Breakpoint, IVA or CID match, vector catch, and halting debug events have the same priority. If more than one of these events occurs on the same instruction, it is Unpredictable which event is taken.

If an instruction triggers a watchpoint debug event any breakpoint, vector catch, or halting debug event scheduled to cancel that instruction, has higher priority.

BKPT debug events have a lower priority than all other debug events.

### 12.6.5 Watchpoint debug events

A precise watchpoint exception has similar behavior as a precise data abort exception with the following differences:

- If the processor is in Halting debug-mode R14\_abt and SPSR\_abt are not updated.
- If the processor is in Monitor debug-mode the DFSR is updated with the encoding for a debug event, DFSR[10,3:0] = b00010. If the processor is in Halting debug-mode the DFSR is unchanged.
- If the processor is in Monitor debug-mode the DFAR is Unpredictable.
- The DSCR[5:2] bits are set to Precise Watchpoint Occurred.

If the watchpointed access is subject to a precise data abort, then the precise abort takes priority over the watchpoint because it is a higher priority exception. If the watchpointed access is subject to an imprecise data abort, then the watchpoint takes priority.

## 12.7 Debug exception

The processor takes a debug exception when a software debug event occurs while in Monitor debug-mode. Prefetch Abort and Data Abort Vector catch debug events are ignored though.

If the processor takes a debug exception because of a breakpoint, BKPT, or vector catch debug event, the processor performs the following actions:

- Sets the DSCR[5:2] method of entry bits to indicate that a watchpoint occurred.
- Sets the CP15 IFSR and IFAR registers as described in *Effect of debug exceptions on CP15 registers and WFAR* on page 12-75.
- Performs the same sequence of actions as in a Prefetch Abort exception by:
  - updating the SPSR\_abt with the saved CPSR
  - changing the CPSR to abort mode and ARM state with normal interrupts and imprecise aborts disabled
  - setting R14\_abt as a regular Prefetch Abort exception, that is, this register gets the address of the cancelled instruction plus 0x04
  - setting the PC to the appropriate Prefetch Abort vector.

---

### Note

---

The Prefetch Abort handler checks the IFSR bit to determine if a debug exception or other kind of Prefetch Abort exception causes the exception entry. If the cause is a debug exception, the Prefetch Abort handler must branch to the debug monitor. You can find the address of the instruction to restart in the R14\_abt register.

---

If the processor takes a debug exception because of a watchpoint debug event, the processor performs the following actions:

- sets the DSCR[5:2] method of debug entry bits to the Precise Watchpoint Occurred encoding
- sets the CP15 DFSR, FAR, and WFAR registers as described in *Effect of debug exceptions on CP15 registers and WFAR* on page 12-75
- performs the same sequence of actions as in a Data Abort exception by:
  - updating the SPSR\_abt with the saved CPSR
  - changing the CPSR to abort mode and ARM state with normal interrupts and imprecise aborts disabled
  - setting R14\_abt as a regular Data Abort exception, that is, this register gets the address of the cancelled instruction plus 0x08
  - setting the PC to the appropriate Data Abort vector.

**Note**

The Data Abort handler checks the DFSR bits to determine if the exception entry was caused by a Debug exception or other kind of Data Abort exception. If the cause is a Debug exception, the Data Abort handler must branch to the debug monitor. The address of the instruction to restart can be found in the R14\_abt register.

Table 12-53 shows the values in the Link Register after exceptions. The ARM and Thumb columns in this table represent the processor state in which the exception occurred.

**Table 12-53 Values in Link Register after exceptions**

Cause of fault	ARM	Thumb	Return address (RA) meaning
Breakpoint	RA+4	RA+4	Breakpointed instruction address
Watchpoint	RA+8	RA+8	Address of the instruction that triggered the watchpoint event
BKPT instruction	RA+4	RA+4	BKPT instruction address
Vector catch	RA+4	RA+4	Vector address
Prefetch Abort	RA+4	RA+4	Address of the instruction that the prefetch abort event canceled
Data Abort	RA+8	RA+8	Address of the instruction that the data abort event canceled

**12.7.1 Effect of debug exceptions on CP15 registers and WFAR**

The four CP15 registers that record abort information are:

- *Data Fault Address Register (DFAR)*
- *Instruction Fault Address Register (IFAR)*
- *Instruction Fault Status Register (IFSR)*
- *Data Fault Status Register (DFSR).*

See Chapter 3 *System Control Coprocessor* for more information on these registers.

If the processor takes a debug exception because of a watchpoint debug event, the processor performs the following actions on these registers:

- it does not change the IFSR or IFAR
- it updates the DFSR with the debug event encoding
- it writes an Unpredictable value to the DFAR

- it updates the WFAR with the address of the instruction that accessed the watchpointed address, plus a processor state dependent offset:
  - + 8 for ARM state
  - + 4 for Thumb and ThumbEE states.

If the processor takes a debug exception because of a breakpoint, BKPT, or vector catch debug event, the processor performs the following actions on these registers:

- it updates the IFSR with the debug event encoding
- it writes an Unpredictable value to the IFAR
- it does not change the DFSR, DFAR, or WFAR.

## 12.7.2 Avoiding unrecoverable states

The processor ignores vector catch debug events on the Prefetch or Data Abort vectors while in Monitor debug-mode because these events put the processor in an unrecoverable state.

The debuggers must avoid other similar cases by following these rules, that apply only if the processor is in Monitor debug-mode:

- if BCR[22:20] is set to b010, an unlinked context ID breakpoint is selected, then the debugger must program BCR[2:1] for the same breakpoint as stated in this section
- if BCR[22:20] is set to b100 or b101, an IVA mismatch breakpoint is selected, then the debugger must program BCR[2:1] for the same breakpoint as stated in this section.

The debugger must write BCR[2:1] for the same breakpoint as either b00 or b10, that selects either match in only USR, SYS, SVC modes or match in only USR mode, respectively. The debugger must not program either b01 (match in any privileged mode) or b11 (match in any mode).

You must only request the debugger to write b00 to BCR[2:1] if you know that the abort handler does not switch to one of the USR, SYS, or SVC mode before saving the context that might be corrupted by a later debug event. You must also be careful about requesting the debugger to set a breakpoint or BKPT debug event inside a Prefetch Abort or Data Abort handler, or a watchpoint debug event on a data address that any of these handlers might access.

In general, you must only set breakpoint or BKPT debug events inside an abort handler after it saves the context of the abort. You can avoid breakpoint debug events in abort handlers by setting BCR[2:1] as previously described.



If the debugged code is not running in a privileged mode, you can prevent watchpoint debug events in abort handlers by setting WCR[2:1] to b10 for matching only nonprivileged accesses.

Failure to follow these guidelines can lead to debug events occurring before the handler is able to save the context of the abort, causing the corresponding registers to be overwritten, and resulting in Unpredictable software behavior.

## 12.8 Debug state

The debug state enables an external agent, usually a debugger, to control the processor following a debug event. While in debug state, the processor behaves as follows:

- Sets the DSCR[0] core halted bit.
- Asserts the **DBGACK** signal, see *DBGACK* on page 12-89.
- Sets the DSCR[5:2] method of entry bits appropriately.
- Flushes the pipeline and does not prefetch any instructions.
- Does not change the execution mode and the CPSR.
- Continues to run the DMA engine. The debugger can stop and restart it using CP15 operations if it has permission.
- Treats exceptions as described in *Exceptions in debug state* on page 12-84.
- Ignores interrupts.
- Ignores new debug events.

### 12.8.1 Entering debug state

When a debug event occurs while the processor is in Halting debug-mode, it switches to a special state called *debug state* so the debugger can take control. You can configure Halting debug-mode by setting DSCR[14] to 1.

If a halting debug event occurs, the processor enters debug state even when Halting debug-mode is not configured.

While the processor is in debug state, the PC does not increment on instruction execution. If the PC is read at any point after the processor has entered debug state, but before an explicit PC write, it returns a value as described in Table 12-54 on page 12-79, depending on the previous state and the type of debug event.

Table 12-54 shows the read PC value after debug state entry for different debug events. The ARM and the Thumb and ThumbEE columns in this table represent the processor state in which the exception occurred.

**Table 12-54 Read PC value after debug state entry**

Debug event	ARM	Thumb and ThumbEE	Return address (RA) meaning
Breakpoint	RA+8	RA+4	Breakpointed instruction address
Watchpoint	RA+8	RA+4	Address of the instruction that triggered the watchpoint debug event
BKPT instruction	RA+8	RA+4	BKPT instruction address
Vector catch	RA+8	RA+4	Vector address
External debug request signal activation	RA+8	RA+4	Address of the instruction that the external debug request signal activation canceled
Debug state entry request command	RA+8	RA+4	Address of the instruction that the debug state entry request command canceled
OS unlock catch event	RA+8	RA+4	Address of the instruction that the OS unlock catch event canceled
CTI debug request signal activation	RA+8	RA+4	Address of the instruction that the CTI debug request signal activation canceled

### 12.8.2 Behavior of the PC and CPSR in debug state

The behavior of the PC and CPSR registers while the processor is in debug state is as follows:

- The PC is frozen on entry to debug state. That is, it does not increment on the execution of ARM instructions. However, the processor still updates the PC as a response to instructions that explicitly modify the PC.
- If the PC is read after the processor has entered debug state, it returns a value as described in Table 12-54, depending on the previous state and the type of debug event.
- If the debugger executes a sequence for writing a certain value to the PC and subsequently it forces the processor to restart without any additional write to the PC or CPSR, the execution starts at the address corresponding to the written value.

- If the debugger forces the processor to restart without performing a write to the PC, the restart address is Unpredictable.
- If the debugger writes to the CPSR, subsequent reads to the PC return an Unpredictable value. If it forces the processor to restart without performing a write to the PC, the restart address is Unpredictable. However, CPSR reads after a CPSR write return the written value.
- If the debugger writes to the PC, subsequent reads to the PC return an Unpredictable value.
- The processor behavior is Unpredictable when executing a conditional PC-updating instruction while in debug state.
- While the processor is in debug state, the CPSR does not change unless an instruction writes to it. In particular, the CPSR IT execution state bits do not change on instruction execution. The CPSR IT execution state bits do not have any effects on instruction execution.
- If the processor executes a data processing instruction with  $Rd == r15$  and  $S == 0$ , then  $alu\_out[0]$  must equal the current value of the CPSR T bit. Otherwise, the processor behavior is Unpredictable.

### 12.8.3 Executing instructions in debug state

In debug state, the processor executes instructions issued through the *Instruction Transfer Register (ITR)*. Before the debugger can force the processor to execute any instruction, it must enable this feature through  $DSCR[13]$ .

While the processor is in debug state, it always decodes ITR instructions as per the ARM instruction set, regardless of the value of the T and J bits of the CPSR.

The following restrictions apply to instructions executed through the ITR while in debug state:

- with the exception of branch instructions and instructions that modify the CPSR, the processor executes any ARM instruction in the same manner as if it was not in debug state
- the branch instructions B, BL, BLX(1), and BLX(2) are Unpredictable
- certain instructions that normally update the CPSR are Unpredictable
- instructions that load a value into the PC from memory are Unpredictable.

## 12.8.4 Writing to the CPSR in debug state

The only instruction that can update the CPSR while in debug state is the MSR instruction. All other ARMv7 instructions that write to the CPSR are Unpredictable, that is, the BX, BXJ, SETEND, CPS, RFE, LDM(3), and data processing instructions with Rd == r15 and S == 1.

The behavior of the CPSR forms of the MSR and MRS instructions in debug state is different to their behavior in normal state:

- When not in debug state, an MSR instruction that modifies the execution state bits in the CPSR is Unpredictable. However, in debug state an MSR instruction can update the execution state bits in the CPSR. A direct modification of the execution state bits in the CPSR by an MSR instruction must be followed by an instruction memory barrier sequence.
- When not in debug state, an MRS instruction reads the CPSR execution state bits as zeros. However, in debug state an MRS instruction returns the actual values of the execution state.

The debugger must execute an instruction memory barrier sequence after it writes to the CPSR execution state bits using an MSR instruction. If the debugger reads the CPSR using an MRS instruction after a write to any of these bits, but before an instruction memory barrier sequence, the value that MRS returns is Unpredictable. Similarly, if the debugger forces the processor to leave debug state after an MSR writes to the execution state bits, but before any instruction memory barrier sequence, the behavior of the processor is Unpredictable.

## 12.8.5 Privilege

While the processor is in debug state, ARM instructions issued through the ITR are subject to different rules whether they can change the processor state. As a general rule, instructions in debug state are always permitted to change the processor state, unless the processor is in a state, mode, and configuration where there are security restrictions.

If the debugger uses the ITR to execute an instruction that is not permitted, the processor ignores the instruction and sets the sticky undefined bit, DSCR[8], to 1.

### Accessing registers and memory

The processor accesses register bank and memory as indicated by the CPSR mode bits. For example, if the CPSR mode bits indicate the processor is in User mode, ARM register reads and returns the User mode banked registers, and memory accesses are presented to the MMU as not privileged.

## Updating CPSR bits

If the debugger writes to the CPSR a value so that it sets the CPSR[4:0] bits to a processor mode where invasive debug is not permitted, this update of the CPSR[4:0] bits is ignored. Similarly, if invasive debug is not permitted for privilege modes in the current security state, writes to the CPSR privileged bits are ignored.

Table 12-55 shows which updates are permitted in debug state:

**Table 12-55 Permitted updates to the CPSR in debug state**

Mode	Secure state <sup>a</sup> or Monitor mode	DBGEN & SPIDEN	Modify CPSR[4:0] to Monitor mode	Update privileged CPSR bits <sup>b</sup>
User	Yes	0	Update ignored	Update ignored
Privileged	Yes	0	Permitted	Permitted
Any	No	0	Update ignored	Permitted
Any	X	1	Permitted	Permitted

- a. The processor is in secure state when CP15 SCR[0] nonsecure bit is set to 0.  
 b. This column excludes the case where the debugger attempts to change CPSR[4:0] to Monitor mode, that is, it only includes updates of the A, I, or F bits, or the CPSR[4:0] bits to a mode other than Monitor.

## Writing to the CPSR SCR

While in debug state, if the debugger forces the processor to execute a CP15 MCR instruction to write to the CP15 *Secure Configuration Register* (SCR), it is only permitted to execute if either of these conditions is true:

- the processor is in a secure privileged mode including Monitor mode
- the processor is in secure User mode, and both **DBGEN** and **SPIDEN** are asserted.

### Note

- Writes to the SCR while in nonsecure state are not permitted even if both **DBGEN** and **SPIDEN** are asserted, except if the processor is in Monitor mode because it is considered to be a secure privileged mode regardless of the value of the SCR[0] NS bit.
- The processor treats attempts to write to the SCR when they are not permitted as Undefined instruction exceptions. See *Exceptions in debug state* on page 12-84 for details of how the processor behaves when Undefined instruction exceptions occur while in debug state.

## Coprocessor instructions

The rules for executing coprocessor instructions other than CP14 and CP15 while in debug state are the same as in normal state. CP14 debug instructions are always permitted while in debug state regardless of the debug permissions, processor mode, and security state.

### ———— Note —————

Nondebug CP14 instructions behave as CP15 instructions while in debug state.

For CP15 instructions, the processor behaves as follows:

- If the debugger is permitted to execute an MSR instruction to change the CPSR[4:0] bits to a privileged mode, then the debugger is also permitted to access privileged CP15 registers. In this situation, the debugger is not required to switch the processor into a privileged mode before executing a privileged CP15 instruction.
- If the processor is in nonsecure state and is permitted to change to Monitor mode, it must do so before issuing CP15 instructions that must be executed in a secure privileged mode. In this case, the processor is not automatically granted secure privileged permissions.
- If the debugger tries to execute a CP15 instruction that is not permitted, the processor generates an Undefined Instruction exception. See *Exceptions in debug state* on page 12-84 for information on how the processor behaves when Undefined instruction exceptions occur while in debug state.

Table 12-56 shows the CP14 and CP15 instruction execution rules.

**Table 12-56 Accesses to CP15 and CP14 registers in debug state**

Mode	SCR[0]	DBGEN & SPIDEN	Access to CP14 registers	Access to banked CP15 registers	Access to restricted access CP15 registers	Access to configurable access CP15 registers
User	0	0	Permitted	Undefined	Undefined	Undefined
User	0	1	Permitted	Secure	Permitted	Permitted
Monitor	0	X	Permitted	Secure	Permitted	Permitted
Monitor	1	X	Permitted	Nonsecure	Permitted	Permitted
PxM <sup>a</sup>	0	X	Permitted	Secure	Permitted	Permitted
User or PxM	1	X	Permitted	Nonsecure	Undefined	As configured

- a. Any privileged mode excluding Secure Monitor mode.

### 12.8.6 Effect of debug state on noninvasive debug

The noninvasive debug features of the processor are the ETM and *Performance Monitoring Unit* (PMU). All of these noninvasive debug features are disabled when the processor is in debug state. See *System performance monitor* on page 3-8 and Chapter 14 *Embedded Trace Macrocell* for more information.

When the processor is in debug state:

- the ETM ignores all instructions and data transfers
- PMU events are not counted
- events are not visible to the ETM
- the PMU *Cycle Count Register* (CCNT) is stopped.

### 12.8.7 Effects of debug events on registers

On entry to debug state, the processor does not update any general-purpose or program status register, this includes the SPSR\_abt or R14\_abt register. Additionally, the processor does not update any coprocessor register, including the CP15 IFSR, DFSR, FAR, or IFAR register, except for CP14 DSCR[5:2] method of debug entry bits. These bits indicate which type of debug event caused the entry into debug state.

———— **Note** —————

On entry to debug state, the processor updates the WFAR register with the virtual address of the instruction accessing the watchpointed address plus:

- + 8 in ARM state
- + 4 in Thumb or ThumbEE state.

### 12.8.8 Exceptions in debug state

While in debug state, exceptions are handled as follows:

**Reset** This exception is taken as in normal processor state. This means the processor leaves debug state as a result of the system reset.

**Prefetch Abort**

This exception cannot occur because the processor does not fetch any instructions while in debug state.

**Debug** The processor ignores debug events, including BKPT instruction.



**SVC** The processor ignores SVC exceptions.

**SMC** The processor ignores SMC exceptions.

### Undefined

When an Undefined Instruction exception occurs in debug state, the behavior of the core is as follows:

- PC, CPSR, SPSR\_und, and R14\_und are unchanged
- the processor remains in debug state
- DSCR[8], sticky undefined bit, is set to 1.

### Precise Data abort

When a precise Data Abort occurs in debug state, the behavior of the core is as follows:

- PC, CPSR, SPSR\_abt, and R14\_abt are unchanged
- the processor remains in debug state
- DSCR[6], sticky precise data abort bit, is set to 1
- DFSR and FAR are set to the same values as if the abort had occurred in normal state.

### Imprecise Data Abort

When an imprecise Data Abort occurs in debug state, the behavior of the core is as follows, regardless of the setting of the CPSR A bit:

- PC, CPSR, SPSR\_abt, and R14\_abt are unchanged
- the processor remains in debug state
- DSCR[7], sticky imprecise data abort bit, is set to 1
- the imprecise Data Abort does not cause the processor to perform an exception entry sequence so DFSR remains unchanged
- the processor does not act on this imprecise Data Abort on exit from the debug state, that is, the imprecise abort is discarded.

### Imprecise Data Aborts on entry and exit from debug state

The processor performs an implicit *Data Synchronization Barrier* (DSB) operation as part of the debug state entry sequence. If this operation detects an imprecise Data Abort, the processor records this event and its type as if the CPSR A bit was set to 1. The purpose of latching this event is to ensure that it can be taken on exit from debug state.

If the processor detects an imprecise Data Abort while already in debug state, for example a debugger-generated imprecise abort, the processor sets the sticky imprecise Data Abort bit, DSCR[7], to 1 but otherwise it discards it. The act of discarding these debugger-generated imprecise Data Aborts does not affect recorded application-generated imprecise Data Aborts.

Before forcing the processor to leave debug state, the debugger must execute a DSB sequence to ensure that all debugger-generated imprecise Data Aborts are detected, and therefore discarded, while still in debug state. After exiting debug state, the processor acts on any recorded imprecise Data Aborts as indicated by the CPSR A bit.

### Imprecise Data Aborts and watchpoints

The watchpoint exception has a higher priority than an imprecise Data Abort. If a data access causes both a watchpoint and an imprecise Data Abort, the processor enters debug state before taking the imprecise Data Abort. The imprecise Data Abort is recorded. This priority order ensures correct behavior where invasive debug is not permitted in privileged modes.

#### 12.8.9 Leaving debug state

The debugger can force the processor to leave debug state by setting the restart request bit, DRCCR[1], to 1. Another way of forcing the processor to leave debug state is through the CTI external restart request mechanism. When one of those restart requests occurs, the processor:

1. Clears the DSCR[1] core restarted flag to 0.
2. Leaves debug state.
3. Clears the DSCR[0] core halted flag to 0.
4. Drives the **DBGACK** signal LOW, unless the DSCR[11] DbgAck bit is set to 1.
5. Starts executing instructions from the address last written to the PC in the processor mode and state indicated by the current value of the CPSR.
6. Sets the DSCR[1] core restarted flag to 1.

## 12.9 Cache debug

There are several memory system requirements for a debugger to work optimally on the cached processor:

- if the debugger performs a memory access while in debug state, caches must not change their state unless the access is a write that hits in the cache
- if the debugger performs a memory access while in debug state so that the cache state becomes incoherent with memory while in normal state, then one of the following conditions must be true:
  - the memory system detects this situation and performs some implicit operations that keep the cache coherent
  - the processor guarantees that the debugger can restore coherency after the memory access.
- the means to keep cache coherency must be sufficient so that it does not significantly slow down the debugging process
- a way to profile cache usage must be available.

### 12.9.1 Cache pollution in debug state

If bit [0] of the *Debug State Cache Control Register* (DSCCR) is set to 0 while the processor is in debug state, then neither the L1 data cache or L2 cache performs any eviction or linefill. However, evictions still occur in any of the following cases:

- If identical virtual addresses, except for bit [12], are mapped to the same physical address and the line that corresponds to the first virtual address is in the L1 data cache, then an access using the second virtual address causes an eviction of the cache line to the L2 cache.
- The L1 data cache controller uses a hash algorithm to determine hits. If two different virtual addresses have the same hash and the line that corresponds to the first VA is in the L1 data cache, then an access using the second VA evicts the line to the L2 cache.

———— **Note** —————

No special feature is required to prevent L1 instruction cache pollution because I-side fetches cannot occur while in debug state.

### 12.9.2 Cache coherency in debug state

The debugger can update memory while in debug state for the following reasons:

- to replace an instruction with a BKPT, or to restore the original instruction
- to download code for the processor to execute on leaving debug state.

The debugger can maintain cache coherency in both these situations with the following features:

- If bit [2] of the DSCCR is set to 0 while the processor is in debug state, it treats any memory access that hits in either L1 data cache or L2 cache as write-through, regardless of the memory region attributes. This guarantees that the L1 instruction cache can see the changes to the code region without the debugger executing a time-consuming and device-specific sequence of cache clean operations.
- After the code is written to memory, the debugger can execute either a CP15 I-cache Invalidate All or a CP15 I-cache Invalidate Line by MVA operation.

---

**Note**

---

- The processor can execute CP15 I-cache Invalidate All or CP15 I-cache Invalidate Line by MVA operation only in privileged mode. However, in debug state the processor can execute these instructions even when invasive debug is not permitted in privileged mode. This exception to the CP15 permission rules described in *Coprocessor instructions* on page 12-83 enables the debugger to maintain coherency in a secure user debug scenario.
  - The CP15 Flush Branch Target Buffer instruction is also valid in debug state regardless of the processor mode. Although the processor implements this instruction as a NOP, making it available in debug state ensures software compatibility with other ARMv7 compliant processors.
  - Execution of the CP15 I-cache Invalidate All operation while in nonsecure state flushes the secure and nonsecure lines from the I-cache.
  - If bit [2] of the DSCCR is set to 0 while the processor is in debug state, then memory writes go through all levels of cache up to the point of coherency, that is, to external memory.
- 

### 12.9.3 Cache usage profiling

There are two ways to obtain cache usage profiling information:

- Statistic profiling using the *Performance Monitoring Unit* (PMU). The processor can count cache accesses and misses over a period of time.
- CP15 operations for accessing L1 and L2 cache tag and data arrays. These instructions provide greater visibility into the cache state at the cost of interrupting the program flow to execute them.

## 12.10 External debug interface

The system can access memory-mapped debug registers through the APB interface. The system can also access ETM and CTI registers through this port.

The APB interface is compliant with the AMBA 3 *Advanced Peripheral Bus* (APB) interface. This APB slave interface supports 32-bits wide data, stalls, slave-generated aborts, and ten address bits [11:2] mapping 4KB of memory. An extra **PADDR31** signal indicates to the processor the source of access. See Appendix A *Signal Descriptions* for a complete list of the APB signals.

### 12.10.1 Miscellaneous debug signals

This section describes some of the miscellaneous debug input and output signals.

#### **EDBGRQ**

This signal generates a halting debug event, that is, it requests the processor to enter debug state. When this occurs, the DSCR[5:2] method of debug entry bits are set to b0100. When **EDBGRQ** is asserted, it must be held until **DBGACK** is asserted. Failure to do so leads to Unpredictable behavior of the processor.

#### **DBGACK**

The processor asserts **DBGACK** to indicate that the system has entered debug state. It serves as a handshake for the **EDBGRQ** signal. The processor also drives the **DBGACK** signal HIGH when the debugger sets the DSCR[10] DbgAck bit to 1.

#### **COMMRX and COMMTX**

The **COMMRX** and **COMMTX** output signals enable interrupt-driven communications over the DTR. By connecting these signals to an interrupt controller, software using the debug communications channel can be interrupted whenever there is new data on the channel or when the channel is clear for transmission.

**COMMRX** is asserted when the CP14 DTR has data for the processor to read, and it is deasserted when the processor reads the data. Its value is equal to DSCR[30] DTRRXfull flag.

**COMMTX** is asserted when the CP14 is ready for write data, and it is deasserted when the processor writes the data. Its value equals the inverse of DSCR[29] DTRTXfull flag.

## DBGNOPWRDWN

The processor asserts **DBGNOPWRDWN** when bit [0] of the Device Power Down and Reset Control Register is 1. The processor power controller works in emulate mode when this signal is HIGH.

## DBGPWRDWNREQ

You must set the **DBGPWRDWNREQ** signal HIGH before removing power from the core domain. Bit [0] of the Device Power Down and Reset Status Register reflects the value of this **DBGPWRDWNREQ** signal.

———— **Note** —————

**DBGPWRDWNREQ** must be tied LOW if the particular implementation does not support separate core and debug power domains.

## DBGPWRDWNACK

This signal indicates to the system that it is safe to bring the core voltage down.

Figure 12-29 shows the relationship of the **DBGPWRDWNREQ** and **DBGPWRDWNACK** signals with the core domain power-down and power-up sequences.

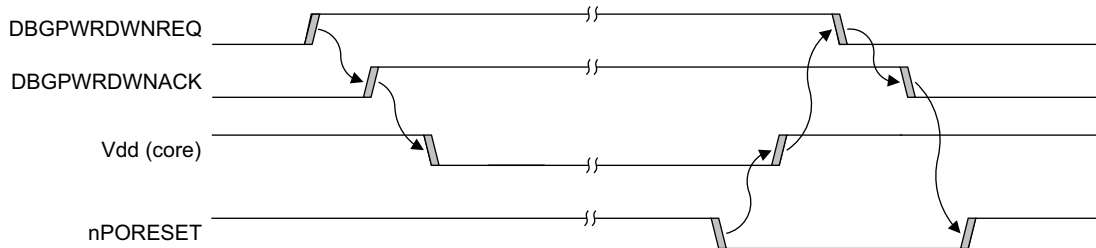


Figure 12-29 Timing of core power-down and power-up sequences

## DBGOSLOCKINIT

When the **DBGOSLOCKINIT** signal is asserted on **PRESETn** reset, the OS lock is set. Otherwise, the OS lock is clear on **PRESETn** reset. ARM recommends that this signal is tied LOW.

## DBGROMADDR

The **DBGROMADDR** signal specifies bits [31:12] of the debug ROM physical address. This is a configuration input and must be tied off or changed while the processor is in reset. In a system with multiple debug ROMs, this address must be tied off to point to the top-level ROM address.

**DBGROMADDRV** is the valid signal for **DBGROMADDR**. If the address cannot be determined, **DBGROMADDR** must be tied off to zero and **DBGROMADDRV** must be tied LOW.

## DBGSELFADDR

The **DBGSELFADDR** signal specifies bits [31:12] of the offset of the debug ROM physical address to the physical address where the APB interface is mapped to the base of the 4KB debug register map. This is a configuration input and must be tied off or changed while the processor is in reset.

**DBGSELFADDRV** is the valid signal for **DBGSELFADDR**. If the offset cannot be determined, **DBGSELFADDR** must be tied off to zero and **DBGSELFADDRV** must be tied LOW.

### 12.10.2 Authentication signals

Table 12-57 shows a list of the valid combination of authentication signals along with its associated debug permissions. Authentication signals are used to configure the processor so its activity can only be debugged or traced in a certain subset of processor modes and security states.

**Table 12-57 Authentication signal restrictions**

SPIDEN	DBGEN <sup>a</sup>	SPNIDEN	NIDEN	Secure invasive debug permitted	Nonsecure invasive debug permitted	Secure noninvasive debug permitted	Nonsecure noninvasive debug permitted
0	0	0	0	No	No	No	No
0	0	0	1	No	No	No	Yes
0	0	1	0	No	No	No	No
0	0	1	1	No	No	Yes	Yes
0	1	0	0	No	Yes	No	Yes
0	1	0	1	No	Yes	No	Yes

Table 12-57 Authentication signal restrictions (continued)

SPIDEN	DBGEN <sup>a</sup>	SPNIDEN	NIDEN	Secure invasive debug permitted	Nonsecure invasive debug permitted	Secure noninvasive debug permitted	Nonsecure noninvasive debug permitted
0	1	1	0	No	Yes	Yes	Yes
0	1	1	1	No	Yes	Yes	Yes
1	0	0	0	No	No	No	No
1	0	0	1	No	No	Yes	Yes
1	0	1	0	No	No	No	No
1	0	1	1	No	No	Yes	Yes
1	1	0	0	Yes	Yes	Yes	Yes
1	1	0	1	Yes	Yes	Yes	Yes
1	1	1	0	Yes	Yes	Yes	Yes
1	1	1	1	Yes	Yes	Yes	Yes

- a. When **DBGEN** is LOW, the processor behaves as if DSCR[15:14] equals b00 with the exception that halting debug events are ignored when this signal is LOW.

### Changing the authentication signals

The **NIDEN**, **DBGEN**, **SPIDEN**, and **SPNIDEN** input signals are either tied off to some fixed value or controlled by some external device.

If software running on the processor has control over an external device that drives the authentication signals, it must make the change using a safe sequence:

1. Execute an implementation-specific sequence of instructions to change the signal value. For example, this might be a single STR instruction that writes certain value to a control register in a system peripheral.
2. If step 1 involves any memory operation, issue a *Data Synchronization Barrier* (DSB).
3. Poll the DSCR or Authentication Status Register to check whether the processor has already detected the changed value of these signals. This is required because the processor might not see the signal change until several cycles after the DSB completes.



4. Issue an *Instruction Synchronization Barrier* (ISB) exception entry or exception return.

The software cannot perform debug or analysis operations that depend on the new value of the authentication signals until this procedure is complete. The same rules apply when the debugger has control of the processor through the ITR while in debug state.

The relevant combinations of the **DBGEN**, **NIDEN**, **SPIDEN**, and **SPNIDEN** values can be determined by polling DSCR[17:16], DSCR[15:14], or the Authentication Status Register.

## 12.11 Using the debug functionality

This section provides some examples of using the debug functionality, both from the point of view of a software engineer writing code to run on an ARM processor and of a developer creating debug tools for the processor. In the former case, examples are given in ARM assembly language. In the latter case, the examples are in C pseudo-language, intended to convey the algorithms to be used. These examples are not intended as source code for a debugger.

The debugger examples use a pair of pseudo-functions such as the following:

```
uint32 ReadDebugRegister(int reg_num)
{
    // read the value of the debug register reg_num at address reg_num << 2
}

WriteDebugRegister(int reg_num, uint32 val)
{
    // write the value val to the debug register reg_num at address reg_num >> 2
}
```

A basic function for using the debug state is executing an instruction through the ITR. Example 12-1 shows the sequence for executing an ARM instruction through the ITR.

### Example 12-1 Executing an ARM instruction through the ITR

---

```
ExecuteARMInstruction(uint32 instr)
{
    // Step 1. Poll DSCR until InstrComp1 is set to 1.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (dscr & (1<<24));
    // Step 2. Write the opcode to the ITR.
    WriteDebugRegister(33, instr);
    // Step 3. Poll DSCR until InstrComp1 is set to 1.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (dscr & (1<<24));
}
```

---

### 12.11.1 Debug communications channel

There are two ways that an external 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 a set of resources used for communicating between the external 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. See *Executing instructions in debug state* on page 12-80 for details.

#### Rules for accessing the DCC

At the core side, the debug communications channel resources are:

- CP14 Debug Register c5 (DTR)
- CP14 Debug Register c1 (DSCR).

Implementations of the ARMv7 debug for the processor are so that:

- If a read of the CP14 DSCR returns 1 for the DTRRXfull flag, then a following read of the CP14 DTR returns valid data and DTRRXfull is cleared to 0. No prefetch flush is required between these two CP14 instructions.
- If a read of the CP14 DSCR returns 0 for the DTRRXfull flag, then a following read of the CP14 DTR returns an Unpredictable value.
- If a read of the CP14 DSCR returns 0 for the DTRTXfull flag, then a following write to the CP14 DTR writes the intended 32-bit word, and sets DTRTXfull to 1. No prefetch flush is required between these two CP14 instructions.
- If a read of the CP14 DSCR returns 1 for the DTRTXfull flag, then a following write to the CP14 DTR is Unpredictable.

When nonblocking mode is selected for DTR accesses, the following conditions are true for memory-mapped DSCR, memory-mapped DTRRX, and DTRTX registers:

- If a read of the memory-mapped DSCR returns 0 for the DTRRXfull flag, then a following write of the memory-mapped DTRRX passes valid data to the processor and sets DTRRXfull to 1.
- If a read of the memory-mapped DSCR returns 1 for the DTRRXfull flag, then a following write of the memory-mapped DTRRX is ignored, that is, both DTRRXfull and DTRRX contents are unchanged.

- If a read of the memory-mapped DSCR returns 1 for the DTRTXfull flag, then a following read of the memory-mapped DTRTX returns valid data and clears DTRTXfull to 0.
- If a read of the memory-mapped DSCR returns 0 for the DTRTXfull flag, then a following read of the memory-mapped DTRTX is ignored, for example, the content of DTRTXfull is unchanged and the read returns an Unpredictable value.

Other uses of the DCC resources are not supported by the ARMv7 debug architecture. In particular, ARMv7 debug does not support the following:

- polling CP14 DSCR[30:29] flags to access the memory-mapped DTRRX and DTRTX registers
- polling memory-mapped DSCR[30:29] flags to access CP14 DTR.

---

**Note**

---

Using the DCC in any of the nonsupported ways can be subject to race conditions.

---

### Software access to the DCC

Software running on the processor that sends data to the debugger through the transmit channel can use the following sequence of instructions as shown in Example 12-2.

#### Example 12-2 Transmit data transfer (target end)

---

```

WriteDCC    ; r0 -> word to send to the debugger
            MRC    p14, 0, PC, c0, c1, 0
            BCS    WriteDCC
            MCR    p14, 0, Rd, c0, c5, 0
            BX     lr

```

---

Example 12-3 shows the sequence of instructions for sending data to the debugger through the receive channel.

#### Example 12-3 Receive data transfer (target end)

---

```

ReadDCC    ; r0 -> word sent by the debugger
            MRC    p14, 0, PC, c0, c1, 0
            BNE    ReadDCC
            MRC    p14, 0, Rd, c0, c5, 0
            BX     lr

```

---

## Debugger access to the DCC

A debugger accessing the DCC through the external interface when not in debug state can use the pseudo-code operations as shown in this section.

Example 12-4 shows the code for transmit data transfer.

### Example 12-4 Transmit data transfer (host end)

---

```
uint32 ReadDCC()
{
    // Step 1. Poll DSCR until DTRRXfull is set to 1.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (dscr & (1<<29));
    // Step 2. Read the value from DTRRX.
    dtr_val := ReadDebugRegister(35);

    return dtr_val;
}
```

---

Example 12-5 shows the code for receive data transfer.

### Example 12-5 Receive data transfer (host end)

---

```
WriteDCC(uint32 dtr_val)
{
    // Step 1. Poll DSCR until DTRTXfull is cleared to 0.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (!(dscr & (1<<30)));
    // Step 2. Write the value to DTRTX.
    WriteDebugRegister(32, dtr_val);
}
```

---

While the processor is running, if the DCC is being used as a data channel, it might be appropriate to poll the DCC regularly.

Example 12-6 on page 12-98 shows the code for polling the DCC.

**Example 12-6 Polling the DCC (host end)**


---

```

PollDCC
{
    dscr := ReadDebugRegister(34);
    if (dscr & (1<<29))
    {
        // DTRTX (data transfer register - transmit) full
        dtr := ReadDebugRegister(35)
        ProcessTransmitWord(dtr);
    }
    elseif (!(dscr & (1<<30)))
    {
        // DTRRX (data transfer register - receive) empty
        dtr := GetNextReceiveWord();
    }
}

```

---

**12.11.2 Programming breakpoints and watchpoints**

The following operations are described in this section:

- *Programming simple breakpoints and the byte address select*
- *Setting a simple aligned watchpoint* on page 12-100
- *Setting a simple unaligned watchpoint* on page 12-101.

**Programming simple breakpoints and the byte address select**

When programming a simple breakpoint, you must set the byte address select bits in the control register appropriately. For a breakpoint in ARM state, this is simple. For Thumb or ThumbEE, you must calculate the value based on the address.

For a simple breakpoint, you can program the settings for the other control bits as Table 12-58 shows:

**Table 12-58 Values to write to BCR for a simple breakpoint**

Bits	Value to write	Description
[31:29]	b000	Reserved
[28:24]	b00000	Breakpoint address mask
[23]	b0	Reserved
[22:20]	b000	Meaning of BVR

**Table 12-58 Values to write to BCR for a simple breakpoint (continued)**

Bits	Value to write	Description
[19:16]	b0000	Linked BRP number
[15:14]	b00	Secure state access control
[13:9]	b00000	Reserved
[8:5]	Derived from address	Byte address select
[4:3]	b00	Reserved
[2:1]	b11	Supervisor access control
[0]	b1	Breakpoint enable

Example 12-7 shows the sequence of instructions for setting a simple breakpoint.

#### Example 12-7 Setting a simple breakpoint

```

SetSimpleBreakpoint(int break_num, uint32 address, iset_t isa)
{
    // Step 1. Disable the breakpoint being set.
    WriteDebugRegister(80 + break_num, 0x0);
    // Step 2. Write address to the BVR, leaving the bottom 2 bits zero.
    WriteDebugRegister(64 + break_num, address & 0xFFFFF0);
    // Step 3. Determine the byte address select value to use.
    case (isa) of
    {
        // Note: The processor does not support Jazelle state,
        // but the ARMv7 Debug architecture does
    when JAZELLE:
        byte_address_select := (1'b1 << (address & 3));
    when THUMB, THUMBEE:
        byte_address_select := (2'b11 << (address & 2));
    when ARM:
        byte_address_select := 4'b1111;
    }
    // Step 4. Write the mask and control register to enable the breakpoint.
    WriteDebugRegister(80 + break_num, 3'b111 | (byte_address_select << 5));
}

```

## Setting a simple aligned watchpoint

The simplest and most common type of watchpoint watches for a write to a given address in memory. In practice, a data object spans a range of addresses but is aligned to a boundary corresponding to its size, so you must set the byte address select bits in the same way as for a breakpoint.

For a simple watchpoint, you can program the settings for the other control bits as Table 12-59 shows:

**Table 12-59 Values to write to WCR for a simple watchpoint**

Bits	Value to write	Description
[31:29]	b000	Reserved
[28:24]	b00000	Watchpoint address mask
[23:21]	b000	Reserved
[20]	b0	Enable linking
[19:16]	b0000	Linked BRP number
[15:14]	b00	Secure state access control
[13]	b0	Reserved
[12:5]	Derived from address	Byte address select
[4:3]	b10	Load/Store access control
[2:1]	b11	Privileged access control
[0]	b1	Watchpoint enable

Example 12-8 shows the code for setting a simple aligned watchpoint.

### Example 12-8 Setting a simple aligned watchpoint

```
SetSimpleAlignedWatchpoint(int watch_num, uint32 address, int size)
{
    // Step 1. Disable the watchpoint being set.
    WriteDebugRegister(112 + watch_num, 0);
    // (Step 2. Write address to the WCR, leaving the bottom 3 bits zero.
    WriteDebugRegister(96 + watch_num, address & 0xFFFFF8);
    // Step 3. Determine the byte address select value to use.
    case (size) of
    {
```



```

when 1:
    byte_address_select := (1'b1 << (address & 7));
when 2:
    byte_address_select := (2'b11 << (address & 6));
when 4:
    byte_address_select := (4'b1111 << (address & 4));
when 8:
    byte_address_select := 8'b11111111;
}
// Step 4. Write the mask and control register to enable the watchpoint.
WriteDebugRegister(112 + watch_num, 5'b10111 | (byte_address_select << 5));
}

```

### Setting a simple unaligned watchpoint

Using the byte address select bits, certain unaligned objects up to a double-word (64 bits) can be watched in a single watchpoint. However, not all cases can be covered and in many cases, a second watchpoint might be required.

Table 12-60 shows some examples.

**Table 12-60 Example byte address masks for watchpointed objects**

Address of object	Object size in bytes	First address value	First byte address mask	Second address value	Second byte address mask
0x00008000	1	0x00008000	b0000001	Not required	-
0x00008007	1	0x00008000	b1000000	Not required	-
0x00009000	2	0x00009000	b0000011	Not required	-
0x0000900c	2	0x00009000	b1100000	Not required	-
0x0000900d	2	0x00009000	b1000000	0x00009008	b0000001
0x0000A000	4	0x0000A000	b0000111	Not required	-
0x0000A003	4	0x0000A000	b0111000	Not required	-
0x0000A005	4	0x0000A000	b1110000	0x0000A008	b0000001
0x0000B000	8	0x0000B000	b1111111	Not required	-
0x0000B001	8	0x0000B000	b1111110	0x0000B008	b0000001

Example 12-9 on page 12-102 shows the code for setting a simple unaligned watchpoint.

**Example 12-9 Setting a simple unaligned watchpoint**


---

```

bool SetSimpleWatchpoint(int watch_num, uint32 address, int size)
{
    // Step 1. Disable the watchpoint being set.
    WriteDebugRegister(112 + watch_num, 0x0);
    // Step 2. Write addresses to the WVRs, leaving the bottom 3 bits zero.
    WriteDebugRegister(96 + watch_num, (address & 0xFFFFF8));
    // Step 3. Determine the byte address select value to use.
    byte_address_select := (1'b1 << size) - 1;
    byte_address_select := (byte_address_select) << (address & 3'b111);
    // Step 4. Write the mask and control register to enable the breakpoint.
    WriteDebugRegister (112 + watch_num, 5'b10111 | ((byte_address_select & 0xFF) << 5));
    // Step 5. Set second watchpoint if required. This is the case if the byte
    // address mask is more than 8 bits.
    if (byte_address_select >= 9'b10000000)
    {
        WriteDebugRegister(112 + watch_num + 1, 0);
        WriteDebugRegister(96 + watch_num + 1, (address & 0xFFFFF8) + 8);
        WriteDebugRegister(112 + watch_num + 1 5'b10111 | ((byte_address_select & 0xFF0) >> 3));
    }
    // Step 6. Return flag to caller indicating if second watchpoint was used.
    return (byte_address_select >= 9'b10000000)
}

```

---

**12.11.3 Single-stepping**

You can use the breakpoint mismatch bit to implement single-stepping on the processor. Unlike high-level stepping, single-stepping implements a low-level step that executes a single instruction at a time. With high-level stepping, the instruction is decoded to determine the address of the next instruction and a breakpoint is set at that address.

Example 12-10 shows the code for single-stepping off an instruction.

**Example 12-10 Single-stepping off an instruction**


---

```

SingleStepOff(uint32 address)
{
    bkpt := FindUnusedBreakpointWithMismatchCapability();
    SetComplexBreakpoint(address, 2'b100 << 20);
}

```

---

---

**Note**

---

In Example 12-10 on page 12-102, the second parameter of `SetComplexBreakpoint()` indicates the value to set `BCR[22:20]`.

This method of single-stepping steps off the instruction that might not necessarily be the same as stepping to the next instruction executed. In certain circumstances, the next instruction executed might be the same instruction being stepped off.

The simplest example of this is a branch to a self instruction such as (`B .`). In this case, the required behavior is most likely to step off the branch to self because this is often used as a means of waiting for an interrupt.

A more complex example is a return from function that returns to the same point. For example, a simple recursive function might terminate with:

```
BL   ThisFunction
POP  {saved_registers, pc}
```

In this case, the `POP` instruction loads a link register that is saved at the start of the function, and if that is the link register created by the `BL` instruction as shown, it points back at the `POP` instruction. Therefore, this single step code unwinds the entire call stack to the point of the original caller, rather than stepping out a level at a time.

---

**Note**

---

It is not possible to single step this piece of code using either the high-level or low-level stepping method.

#### 12.11.4 Debug state entry

On entry to debug state, the debugger must first flush the load/store unit of pending memory transactions so that it can flag imprecise Data Aborts. The debugger can then read the processor state, including all registers and the PC, and determine the cause of the exception from the DSCR Method of Entry bits.

Example 12-11 shows the code for entry to debug state.

#### Example 12-11 Entering debug state

---

```
OnEntryToDebugState(PROCESSOR_STATE *state)
{
    // Step 1. Read the DSCR to determine the cause of debug entry.
    state->dscr := ReadDebugRegister(34);
    // Step 2. Issue a Data Synchronization Barrier instruction if required;
```

## Debug

```
// this is not required by the processor but is required for ARMv7
// debug.
if ((state->dscr & (1<<19)) == 0)
{
    ExecuteARMInstruction(0xEE070F9A)
    // Step 3. Poll the DSCR for DSCR[19] to be set to 1.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (dscr & (1<<19));
}
// Step 4. Read the entire processor state. The function ReadAllRegisters
//reads all general-purpose registers for all processor mode, and saves
//the data in "state".
ReadAllRegisters(state);
// Step 5. Based on the CPSR (processor state), determine the actual restart
//address
if (state->cpsr & (1<<5);
{
    // set the T bit to Thumb or ThumbEE state
    state->pc := state->pc - 4;
}
elseif (state->cpsr & (1<<24))
{
    // Set the J bit to Jazelle state. Note: This code is not relevant for processor
    // because it does not implement Jazelle state.
    state->pc := state->pc - IMPLEMENTATION_DEFINED
value;
}
else
{
    // ARM state
    state->pc := state->pc - 8;
}
// Step 6. If the method of entry was Watchpoint Occurred, read the WFAR
// register
method_of_debug_entry := ((state->dscr >> 10) & 0xF;
    if (method_of_debug_entry == 2'b0010 || method_of_debug_entry == 2'b1010)
    {
        state->wfar := ReadDebugRegister(6);
    }
}
```

---

### 12.11.5 Debug state exit

When exiting debug state, the program counter must always be written. If the execution state or CPSR must be changed, this must be done before writing to the PC because writing to the CPSR can affect the PC.

Having restored the program state, the debugger can restart by writing to bit [1] of the Debug Run Control Register. It must then poll bit [1] of the Debug Status and Control Register to determine if the core has restarted.

Example 12-12 shows the code for exit from debug state.

#### Example 12-12 Leaving debug state

---

```

ExitDebugState(PROCESSOR_STATE *state)
{
    // Step 1. Update the CPSR value
    WriteCPSR(state->cpsr);
    // Step 2. Restore any registers corrupted by debug state. The function
    // WriteAllRegisters restores all general-purpose registers for all
    // processor modes apart from R0.
    WriteAllRegisters(state);
    // Step 3. Write the return address.
    WritePC(state->pc);
    // Step 4. Writing the PC corrupts R0 therefore, restore R0 now.
    WriteRegister(0, state->r0);
    // Step 5. Write the restart request bit in the DRCR.
    WriteDebugRegister(36, 1<<1);
    // Step 6. Poll the RESTARTED flag in the DSCR.
    repeat
    {
        dscr := ReadDebugRegister(34);
    }
    until (dscr & (1<<1));
}

```

---

### 12.11.6 Accessing registers and memory in debug state

This section describes the following:

- *Reading and writing registers through the DCC* on page 12-106
- *Reading the PC in debug state* on page 12-106
- *Reading the CPSR in debug state* on page 12-107
- *Writing the CPSR in debug state* on page 12-107
- *Reading memory* on page 12-108

- *Fast register read/write* on page 12-110
- *Fast memory read/write* on page 12-112
- *Accessing secure and nonsecure coprocessor registers* on page 12-114.

## Reading and writing registers through the DCC

To read a single register, the debugger can use the sequence shown in Example 12-13. This sequence depends on two other sequences, *Executing an ARM instruction through the ITR* on page 12-94 and *Transmit data transfer (host end)* on page 12-97.

### Example 12-13 Reading an ARM register

---

```
uint32 ReadARMRegister(int Rd)
{
    // Step 1. Execute instruction MCR p14, 0, Rd, c0, c5, 0 through the ITR.
    ExecuteARMInstruction(0xEE000E15 + (Rd<<12));
    // Step 2. Read the register value through DTRTX.
    reg_val := ReadDCC();
    return reg_val;
}
```

---

Example 12-14 shows a similar sequence for writing an ARM register.

### Example 12-14 Writing an ARM register

---

```
WriteRegister(int Rd, uint32 reg_val)
{
    // Step 1. Write the register value to DTRRX.
    WriteDCC(reg_val);
    // Step 2. Execute instruction MRC p14, 0, Rd, c0, c5, 0 to the ITR.
    ExecuteARMInstruction(0xEE100E15 + (Rd<<12));
}
```

---

## Reading the PC in debug state

Example 12-15 on page 12-107 shows the code to read the PC.

### ———— Note ————

You can use a similar sequence to write to the PC to set the return address when leaving debug state.

---

**Example 12-15 Reading the PC**


---

```

ReadPC()
{
    // Step 1. Save R0
    saved_r0 := ReadRegister(0);
    // Step 2. Execute the instruction MOV r0, pc through the ITR.
    ExecuteARMInstruction(0xE1A0000F);
    // Step 3. Read the value of r0 that now contains the PC.
    pc := ReadRegister(0);
    // Step 4. Restore the value of R0.
    WriteRegister(0, saved_r0);
    return pc;
}

```

---

**Reading the CPSR in debug state**

Example 12-16 shows the code for reading the CPSR.

**Example 12-16 Reading the CPSR**


---

```

ReadCPSR()
{
    // Step 1. Save R0.
    saved_r0 := ReadRegister(0);
    // Step 2. Execute instruction MRS r0, CPSR through the ITR.
    ExecuteARMInstruction(0xE10F0000);
    // Step 3. Read the value of r0 that now contains the CPSR
    cpsr_val := ReadRegister(0);
    // Step 4. Restore the value of R0.
    WriteRegister(0, saved_r0);
    return cpsr_val;
}

```

---

**Note**

You can use similar sequences to read the SPSR in privileged modes.

---

**Writing the CPSR in debug state**

Example 12-17 on page 12-108 shows the code for writing the CPSR.

**Example 12-17 Writing the CPSR**


---

```

WriteCPSR(uint32 cpsr_val)
{
    // Step 1. Save R0.
    saved_r0 := ReadRegister(0);
    // Step 2. Write the new CPSR value to r0.
    WriteRegister(0, cpsr_val);
    // Step 3. Execute instruction MSR r0, CPSR through the ITR.
    ExecuteARMInstruction(0xE12FF000);
    // Step 4. Execute a PrefetchFlush instruction through the ITR.
    ExecuteARMInstruction(0x9EE070F95);
    // Step 5. Restore the value of r0.
    WriteRegister(0, saved_r0);
}

```

---

**Reading memory**

Example 12-18 shows the code for reading a byte of memory.

**Example 12-18 Reading a byte of memory**


---

```

uint8 ReadByte(uint32 address, bool &aborted)
{
    // Step 1. Save the values of r0 and r1.
    saved_r0 := ReadRegister(0);
    saved_r1 := ReadRegister(1);
    // Step 2. Write the address to r0.
    WriteRegister(0, address);
    // Step 3. Execute the instruction LDRB r1,[r0] through the ITR.
    ExecuteARMInstruction(0xE5D01000);
    // Step 4. Read the value of r1 that contains the data at the address.
    datum := ReadRegister(1);
    // Step 5. Restore the corrupted registers r0 and r1.
    WriteRegister(0, saved_r0);
    WriteRegister(1, saved_r1);
    // Step 6. Check the DSCR for a sticky abort.
    aborted := CheckForAborts();
    return datum;
}

```

---

Example 12-19 on page 12-109 shows the code for checking for aborts after a memory access.



**Example 12-19 Checking for an abort after memory access**


---

```

bool CheckForAborts()
{
    // Step 1. Check the DSCR for a sticky abort.
    dscr := ReadDebugRegister(34);
    if (dscr & ((1<<6) + (1<<7)))
    {
        // Step 2. Clear the sticky flag by writing DRCCR[2].
        WriteDebugRegister(36, 1<<2);
        return true;
    }
    else
    {
        return false;
    }
}

```

---

**Note**

You can use similar sequence to read half-word of memory and to write to memory.

---

To read or write blocks of memory, substitute the data instruction with one that uses post-indexed addressing. For example:

```
LDRB r1, [r0],#1
```

This is done to prevent reloading the address value for each sequential word.

Example 12-20 shows the code for reading a block of bytes of memory.

**Example 12-20 Reading a block of bytes of memory**


---

```

ReadBytes(uint32 address, bool &aborted, uint8 *data, int nbytes)
{
    // Step 1. Save the value of r0 and r1.
    saved_r0 := ReadRegister(0);
    saved_r1 := ReadRegister(1);
    // Step 2. Write the address to r0
    WriteRegister(0, address);
    while (nbytes > 0)
    {
        // Step 3. Execute instruction LDRB r1,[r0],1 through the ITR.
        ExecuteARMInstruction(0xE4D01001);
        // Step 4. Read the value of r1 that contains the data at the
        // address.
    }
}

```

---

```

        *data++ := ReadRegister(1);
        --nbytes;
    }
    // Step 5. Restore the corrupted registers r0 and r1.
    WriteRegister(0, saved_r0);
    WriteRegister(1, saved_r1);
    // Step 6. Check the DSCR for a sticky abort.
    aborted := CheckForAborts();
    return datum;
}

```

---

Example 12-21 shows the sequence for reading a word of memory.

---

**Note**

A faster method is available for reading and writing words using the direct memory access function of the DCC. See *Fast memory read/write* on page 12-112.

---

### Example 12-21 Reading a word of memory

---

```

uint32 ReadWord(uint32 address, bool &aborted)
{
    // Step 1. Save the value of r0.
    saved_r0 := ReadRegister(0);
    // Step 2. Write the address to r0.
    WriteRegister(0, address);
    // Step 3. Execute instruction STC p14, c5, [r0] through the ITR.
    ExecuteARMInstruction(0xED905E00);
    // Step 4. Read the value from the DTR directly.
    datum := ReadDCC();
    // Step 5. Restore the corrupted register r0.
    WriteRegister(0, saved_r0);
    // Step 6. Check the DSCR for a sticky abort.
    aborted := CheckForAborts();
    return datum;
}

```

---

### Fast register read/write

When multiple registers must be read in succession, you can optimize the process by placing the DCC into stall mode and by writing the value 1 to the DCC access mode bits. For more information, see *CP14 c1, Debug Status and Control Register* on page 12-21.

Example 12-22 shows the sequence to change the DTR access mode.

---

#### Example 12-22 Changing the DTR access mode

---

```
SetDTRAccessMode(int mode)
{
    // Step 1. Write the mode value to DSCR[21:20].
    dscr := ReadDebugRegister(34);
    dscr := (dscr & ~(0x3<<20)) | (mode<<20);
    WriteDebugRegister(34, dscr);
}
```

---

Example 12-23 shows the sequence to read registers in stall mode.

---

#### Example 12-23 Reading registers in stall mode

---

```
ReadRegisterStallMode(int Rd)
{
    // Step 1. Write the opcode for MCR p14, 0, Rd, c5, c0 to the ITR.
    // Write stalls until the ITR is ready.
    WriteDebugRegister(33, 0xEE00E15 + (Rd<<12));
    // Step 2. Read the register value through the DCC. Read stalls until
    // DTRTX is ready
    reg_val := ReadDebugRegister(32);
    return reg_val;
}
```

---

Example 12-24 shows the sequence to write registers in stall mode.

---

#### Example 12-24 Writing registers in stall mode

---

```
WriteRegisterInStallMode(int Rd, uint32 value)
{
    // Step 1. Write the value to the DTRRX.
    // Write stalls until the DTRRX is ready.
    WriteDebugRegister(35, value);
    // Step 2. Write the opcode for MRC p14, 0, Rd, c5, c0 to the ITR.
    // Write stalls until the ITR is ready.
    WriteDebugRegister(33, 0xEE10E15 + (Rd<<12));
}
```

---

---

**Note**

---

To transfer a register to the processor when in stall mode, you are not required to poll the DSCR each time an instruction is written to the ITR and a value read from or written to the DTR. The processor stalls using the signal **PREADY** until the previous instruction has completed or the DTR register is ready for the operation.

---

**Fast memory read/write**

This section provides example code that enable faster reads from memory by making use of the DTR access mode.

Example 12-25 shows the sequence for reading a block of words of memory.

**Example 12-25 Reading a block of words of memory**

---

```

ReadWords(uint32 address, bool &aborted, uint32 *data, int nwords)
{
    // Step 1. Write the value b01 to DSCR[21:20] for stall mode.
    SetDTRAccessMode(2'b01);
    // Step 2. Save the value of r0.
    saved_r0 := ReadRegisterInStallMode(0);
    // Step 3. Write the address to read from to the DTRRX.
    // Write stalls until the DTRRX is ready.
    WriteRegisterInStallMode(0, address);
    // Step 4. Write the opcode for LDC p14, c5, [r0], 4 to the ITR.
    // Write stalls until the ITR is ready.
    WriteDebugRegister(33, 0xED903E00);
    // Step 5. Write the value 2'b10 to DSCR[21:20] for fast mode.
    SetDCCAccessMode(2);
    // Step 6. Loop reading out the data.
    // Each time a word is read, the instruction is reissued.
    while (nwords > 1)
    {
        *data++ = ReadDebugRegister(32);
        --nwords;
    }
    // Step 7. Write the value 2'b00 to DSCR[21:20] for normal mode.
    SetDTRAccessMode(2'b00);
    // Step 8. Read the last word from the DTR.
    // The DSCR[29] check may timeout if a data abort occurred while reading
    // memory.
    dscr := ReadDebugRegister(34);
    timeout = 0;
    until ((dscr & (1<<29)) || (timeout == TIMEOUT))
    {

```

```

        dscr := ReadDebugRegister(34);
        ++timeout;
    }
    // Step 9. Restore the corrupted register r0.
    WriteRegisterInStallMode(0, saved_r0);
    // Step 10. Check the DSCR for a sticky abort.
    aborted := CheckForAborts();
}

```

---

Example 12-26 shows the sequence for writing a block of words to memory.

### Example 12-26 Writing a block of words to memory (fast download)

---

```

WriteWords(uint32 address, bool &aborted, uint32 *data, int nwords)
{
    // Step 1. Save the value of r0.
    saved_r0 := ReadRegister(0);
    // Step 2. Write the value 2'b10 to DSCR[21:20] for fast mode.
    SetDTRAccessMode(2'b10);
    // Step 3. Write the opcode for MCR p14, 0, r0, c5, c0 to the ITR.
    // Write stalls until the ITR is ready but the instruction is not issued.
    WriteDebugRegister(33, 0xEE00E15);
    // Step 4. Write the address to read from to the DTRRX
    // Write stalls until the ITR is ready, but the instruction is not reissued.
    WriteDebugRegister(33, address);
    // Step 5. Write the opcode for STC p14, c5, [r0], 4 to the ITR.
    // Write stalls until the ITR is ready but the instruction is not issued.
    WriteDebugRegister(33, 0xED803E00);
    // Step 6. Loop writing the data.
    // Each time a word is written to the DTR, the instruction is reissued.
    while (nwords > 0)
    {
        WriteDebugRegister(33, *data++);
        --nwords;
    }
    // Step 7. Write the value 2'b00 to DSCR[21:20] for normal mode.
    SetDTRAccessMode(2'b00);
    // Step 8. Restore the corrupted register r0.
    WriteRegister(0, saved_r0);
    // Step 9. Check the DSCR for a sticky abort.
    aborted := CheckForAborts();
}

```

---

---

**Note**

---

As the amount of data transferred increases, these functions reach an optimum performance of one debug register access per data word transferred.

---

**Accessing secure and nonsecure coprocessor registers**

The sequence for accessing coprocessor registers is the same as for the PC and CPSR. That is, you must first execute an instruction to transfer the register to an ARM register, then read the value back through the DTR.

Example 12-27 shows the sequence for reading a coprocessor register.

**Example 12-27 Reading a coprocessor register**

---

```
uint32 ReadCPReg(int CPnum, int opc1, int CRn, int CRm, int opc2)
{
    // Step 1. Save R0.
    saved_r0 := ReadRegister(0);
    // Step 2. Execute instruction MCR p15, 0, r0, c0, c1, 0 through the ITR.
    ExecuteARMInstruction(0xEE000010 + (CPnum<<8) + (opc1<<21) + (CRn<<16) + CRm + (opc2<<5));
    // Step 3. Read the value of r0 that now contains the CP register.
    CP15c1 := ReadRegister(0);
    // Step 4. Restore the value of R0.
    WriteRegister(0, saved_r0);
    return CP15c1;
}
```

---



---

**Note**

---

For banked CP15 registers, it might be necessary to switch security state to read the required coprocessor register. Switching from secure to nonsecure state is simple because you can write to the *Secure Configuration Register* (SCR) from any secure privileged mode.

---

Example 12-28 shows the sequence for changing from secure to nonsecure state.

**Example 12-28 Changing from secure to nonsecure state**

---

```
SecureToNonSecure()
{
    // Step 1. Set the NS bit.
    scr := ReadCPReg(15, 0, 1, 1, 0);
}
```

---

```

scr := (scr | 1);
WriteCPRreg(15, 0, 1, 1, 0, scr);
}

```

---

Example 12-29 shows the sequence to change from nonsecure to secure state, however the processor must first be in Monitor mode.

#### Example 12-29 Changing from nonsecure to secure state

---

```

NonSecureToSecure()
{
    // Step 1. Change the processor to Monitor mode.
    saved_cpsr := ReadCPSR();
    new_cpsr := (saved_cpsr & ~0x1F) | 0x16;
    WriteCPSR(new_cpsr);
    // Step 2. Clear the NS bit.
    scr := ReadCPRreg(15, 0, 1, 1, 0);
    scr := (scr & ~1);
    WriteCPRreg(15, 0, 1, 1, 0, scr);
    // Step 3. Restore the processor mode.
    WriteCPSR(saved_cpsr);
}

```

---

## 12.12 Debugging systems with energy management capabilities

The processor offers functionality for debugging systems with energy management capabilities. This section describes scenarios where the OS takes energy-saving measures when in an idle state.

The different measures that the OS can take to save energy during an idle state are divided into two groups:

**Standby** The OS takes measures that reduce energy consumption but maintain the processor state.

**Power down** The OS takes measures that reduce energy consumption but do not maintain the processor state. Recovery involves a reset of the core after the power level has been restored, and reinstallation of the processor state.

### 12.12.1 Standby

Standby is the least invasive OS energy saving state because it only implies that the core is unavailable. It does not clear any of the debug settings. For this case, if **DBGNOCLKSTOP** is HIGH, the processor guarantees the following:

- If the processor is in standby and a halting debug event occurs, the processor:
  - leaves standby
  - retires the *Wait-For-Interrupt* (WFI) instruction
  - enters debug state.
- The processor responds to APB accesses as if it was not in standby.

———— **Note** —————

If you implement the CoreSight *Debug Access Port* (DAP) in your system, ARM recommends that the DAP **CDBGPWRUPREQ** output is connected to the **DBGNOCLKSTOP** processor input.

### 12.12.2 Emulating power down

By writing to bit [0] of the **PRCR**, the debugger asserts the **DBGNOPWRDWN** output. The expected usage model of this signal is that it is connected to the system power controller and that, when HIGH, it indicates that this controller can work in emulate mode.



If on a power-down request from the processor, the power controller is in emulate mode. It does not remove core or ETM power but, otherwise, it behaves exactly the same as in normal mode.

Emulating power down is ideal for debugging applications running on top of operating systems that are free of errors because the debug register settings are not lost on a power-down event. However, there are a number of disadvantages such as:

- **nIRQ** and **nFIQ** interrupts to the processor must be externally masked as part of the emulation to prevent them from retiring the WFI instruction from the pipeline.
- The reset controller must also be aware of this emulate mode to assert **ARESETn** on power up, rather than **nPORESET**. Asserting **nPORESET** on power up clears the debug registers inside the core power domain.
- The timing effects of power down and voltage stabilization are not factored in the power-down emulation. This is the case for systems with voltage recovery controlled by a closed loop system that monitors the core supply voltage, rather than a fixed timed for voltage recovery.
- State lost during power down is not modeled by the emulation, making it possible to miss errors in the state storage and recovery routines.
- Attaching the debugger for a post-mortem debug session is not possible because setting the **DBGNOPWRDWN** signal to 1 might not cause the processor to power up. The effect of setting **DBGNOPWRDWN** to 1 when the processor is already powered down is implementation-defined, and is up to the system designer.

### 12.12.3 Detecting power down

The processor enables the debugger to detect a power-down event occurrence so it can attempt to restore the debug session. Power-down events are detected by the following features:

- While the core is powered down, accesses to debug registers return a slave-generated error response. See *APB interface* on page A-8 and *Power down permission* on page 12-14 for more information.
- If the processor powers back up again before the debugger had a chance to access the APB interface, the debugger can still detect the occurrence of a power-down event. This is because the sticky power down status bit forces the processor to generate a slave-generated error response. See *Device Power Down and Reset Status Register* on page 12-51 for more details on the sticky power down bit.

## 12.12.4 Operating system support

The OS Save and Restore Registers enable an operating system to save the debug registers before power down and to recover them after power up. The debugger and the debug monitor are prevented from accessing the debug registers from the time the OS starts saving the registers through power down, until they are restored after power up. This behavior minimizes the possibility of race conditions and therefore, increases the chances that the debug agent is able to resynchronize successfully after the OS completes the restore.

Example 12-30 shows the sequence on power down of an operating system.

### Example 12-30 OS debug register save sequence

---

```

; On entry, r0 points to a block of memory to save the debug registers.
SaveDebugRegisters
    PUSH    {r4, lr}
    MOV     r4, r0    ; save pointer
; Step 1. Set the OS Lock Access Register (OSLAR).
    BL     GetDebugRegisterBase    ; returns base in R0
    LDR    r1, =0xC5ACCE55
    STR    r1, [r0, 0x300]    ; write OSLAR
; Step 2. Get the number of words to save.
    LDR    r1, [r0, 0x308]    ; OSSRR returns size on first read
; Step 3. Loop reading words from the OSSRR.
1  LDR    r2, [r0, 0x308]    ; load a word of data
    STR    r2, [r4], 4    ; push onto the save stack
    SUBS  r1, r1, 1
    BNE%B1    ; loop
; Step 4. Return the pointer to the first word not written and
; leave the OSLAR set because no further changes are required.
    MOV    r0, r4
    POP    {r4, pc}

```

---

Example 12-31 shows the sequence on power up of an operating system.

### Example 12-31 OS debug register restore sequence

---

```

; On entry, r0 points to a block of saved debug registers.
RestoreDebugRegisters
    PUSH    {r4, lr}
    MOV     r4, r0    ; save pointer
; Step 1. Get the number of words saved.
    BL     GetDebugRegisterBase    ; returns base in R0
    LDR    r1, [r0, 0x308]    ; OSSRR returns size on first read

```

---

```

; Step 2. Loop writing words from the OSSRR.
1  LDR  r2, [r4], 4           ; load a word from the save stack
   STR  r2, [r0, 0x308]      ; store a word of data
   SUBS r1, r1, 1
   BNE%B1                    ; loop
; Step 3. Clear the OS Lock Access Register (OSLAR).
   LDR  r1, =0xC5ACCE55
   STR  r1, [r0, 0x300]      ; write OSLAR
; Step 4. Return the pointer to the first word not written.
   MOV  r0, r4
   POP  {r4, pc}

```

---

———— **Note** ————

When the OSLAR is cleared, a debug event is triggered if the OS unlock catch bit is set to 1. This can be useful for a debugger to restart a debugging session.

---

## 12.12.5 Registers available during power down

Some debug registers reside in the debug power domain so they can be available while the core is powered down. This register set is chosen so the debugger can identify the part at any time and debug the OS power-up sequence.

## 12.12.6 Scenarios and usage models

This section describes the different debugging scenarios for systems with energy management capabilities along with a description of how the debug features help with those.

### Application debug on a stable OS

For this system, set the **DBGNOPWRDWN** signal to 1 to emulate power down.

### Application debug on an OS with save and restore capability

For this system, application debug is possible without power-down emulation if the OS supports storing and recovering of the debug registers. This is useful in systems where either:

- the save and restore capability is already implemented in the OS
- power-down emulation is undesirable because it makes it difficult to reproduce the error
- the system design does not support the power-down emulation.

---

**Note**

---

The debugger or debug monitor can program the debug logic to trigger a debug event on clearing of the OS lock.

---

**Debugging of the power-up sequence**

When debugging the OS power-up sequence:

- The processor can be identified while the core is powered down.
- The internal signal **ARESETn** can be held on power up. If bit [2] of the **PRCR** is set to 1, the nondebug logic of the processor is held in reset on power up. When this bit is set to 1, it enables the debugger to wait for the power-up event to occur, reprogram the debug registers, and start execution by clearing this bit to 0.
- The **EDBGRQ** or **DRCR[0]** halting debug events can be set to 1 at any point in time, even if the core is powered down.
- The debugger can set **PRCR[2]** to 1, wait for the power-up event to occur, assert **EDBGRQ** or **DRCR[0]**, and clear the **PRCR[2]** bit to 0 for the processor to enter debug state on executing the first instruction. This enables single-stepping of the power-up sequence.

When the debugger detects a slave-generated error response, it indicates that one of the following is true:

- the debug registers are not available because the core is powered down
- the debug registers are not available because the OS locks the APB interface
- the debug registers are available but the error response warns that a previous power-down event cleared them, that is, the sticky power down bit is set to 1.

# Chapter 13

## NEON and VFP Programmers Model

This chapter describes the NEON and VFP programmers model. It contains the following sections:

- *About the NEON and VFP programmers model* on page 13-2
- *General-purpose registers* on page 13-4
- *Short vectors* on page 13-6
- *System registers* on page 13-12
- *Modes of operation* on page 13-21
- *Compliance with the IEEE 754 standard* on page 13-23.

## 13.1 About the NEON and VFP programmers model

The processor implements both the ARM Advanced SIMD and VFPv3 architectures. See the *ARM Architecture Reference Manual* for information on the Advanced SIMD and VFPv3 instruction sets.

---

### Note

---

This chapter uses the older assembler language instruction mnemonics. See Appendix B *Instruction Mnemonics* for information about the *Unified Assembler Language* (UAL) equivalents of the Advanced SIMD and VFP data-processing instruction mnemonics. See the *ARM Architecture Reference Manual* for more information on the UAL syntax.

---

### 13.1.1 NEON media coprocessor

The NEON coprocessor implements the Advanced SIMD media processing architecture. Advanced SIMD is an optional part of the ARMv7-A architecture. The components of the NEON coprocessor are:

- NEON register file with 32x64-bit general-purpose registers
- NEON integer execute pipeline (ALU, Shift, MAC)
- NEON dual, single-precision floating-point execute pipeline (FADD, FMUL)
- NEON load/store and permute pipeline
- nonpipelined VFP coprocessor that implements VFPv3 data-processing floating-point operations.

The NEON coprocessor can receive up to two valid Advanced SIMD instructions per cycle from the ARM integer instruction execute unit. In addition, it can receive 32-bit MCR data from or send 32-bit MRC data to the ARM integer instruction execute unit.

The NEON coprocessor can load data from either the L1 data cache or the L2 memory system. Enable L1 data caching for best performance of the NEON coprocessor when the L2 memory system is off or not present. See *c1, Auxiliary Control Register* on page 3-61.

### 13.1.2 VFP coprocessor

The VFP coprocessor implements the VFPv3 architecture. The VFP coprocessor provides a floating-point computation coprocessor that is fully compliant with the *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*,

referred to in this document as the IEEE 754 standard. The VFP coprocessor supports all data-processing instructions and data types in the VFPv3 architecture and is described in the *ARM Architecture Reference Manual*.

Designed for the processor, the VFP coprocessor fully supports single-precision and double-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. Conversions between fixed-point and floating-point data formats, and floating-point constant instruction are provided.

## 13.2 General-purpose registers

The NEON and VFP coprocessor shares the same register bank. This is distinct from the ARM register bank.

You can reference the NEON and VFP register bank using three explicitly aliased views, as described in the following sections.

Figure 13-1 on page 13-5 shows the three views of the register bank and the way the word, doubleword, and quadword registers overlap.

### 13.2.1 NEON views of the register bank

NEON views the register bank as:

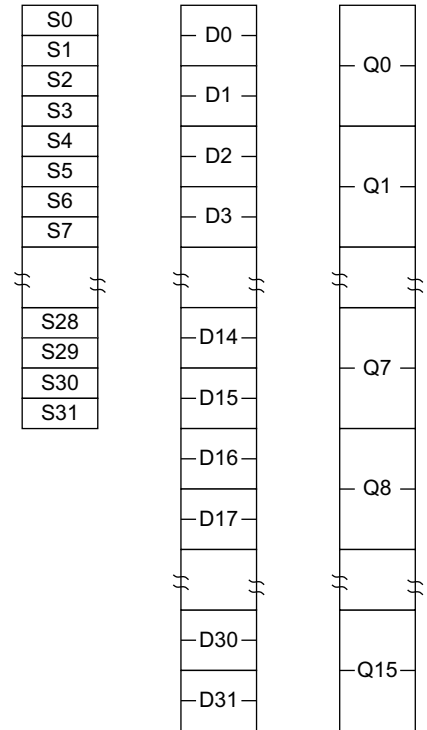
- Sixteen 128-bit quadword registers, Q0-Q15.
- Thirty-two 64-bit doubleword registers, D0-D31. This view is also available in VFP.
- A combination of these 128-bit and 64-bit registers, Q0-Q15 and D0-D31.

### 13.2.2 VFP views of the register bank

In VFP, you can view the register bank as:

- Thirty-two 64-bit doubleword registers, D0-D31. This view is also available in NEON.
- Thirty-two 32-bit single word registers, S0-S31. Only half of the register bank is accessible in this view.
- A combination of these 128-bit and 64-bit registers, D0-D31 and S0-S31.





**Figure 13-1 NEON and VFP register bank**

The mapping between the registers is as follows:

- $S\langle 2n \rangle$  maps to the least significant half of  $D\langle n \rangle$
- $S\langle 2n+1 \rangle$  maps to the most significant half of  $D\langle n \rangle$
- $D\langle 2n \rangle$  maps to the least significant half of  $Q\langle n \rangle$
- $D\langle 2n+1 \rangle$  maps to the most significant half of  $Q\langle n \rangle$ .

For example, you can access the least significant half of the elements of a vector in  $Q6$  by referring to  $D12$ , and the most significant half of the elements by referring to  $D13$ .

## 13.3 Short vectors

The VFPv3 architecture supports execution of short vector instructions of up to eight operations on single-precision data and up to four operations on double-precision data.

The register file is especially suited for short vector operations. The four single-precision and eight double-precision register banks function as four hardware circular queues.

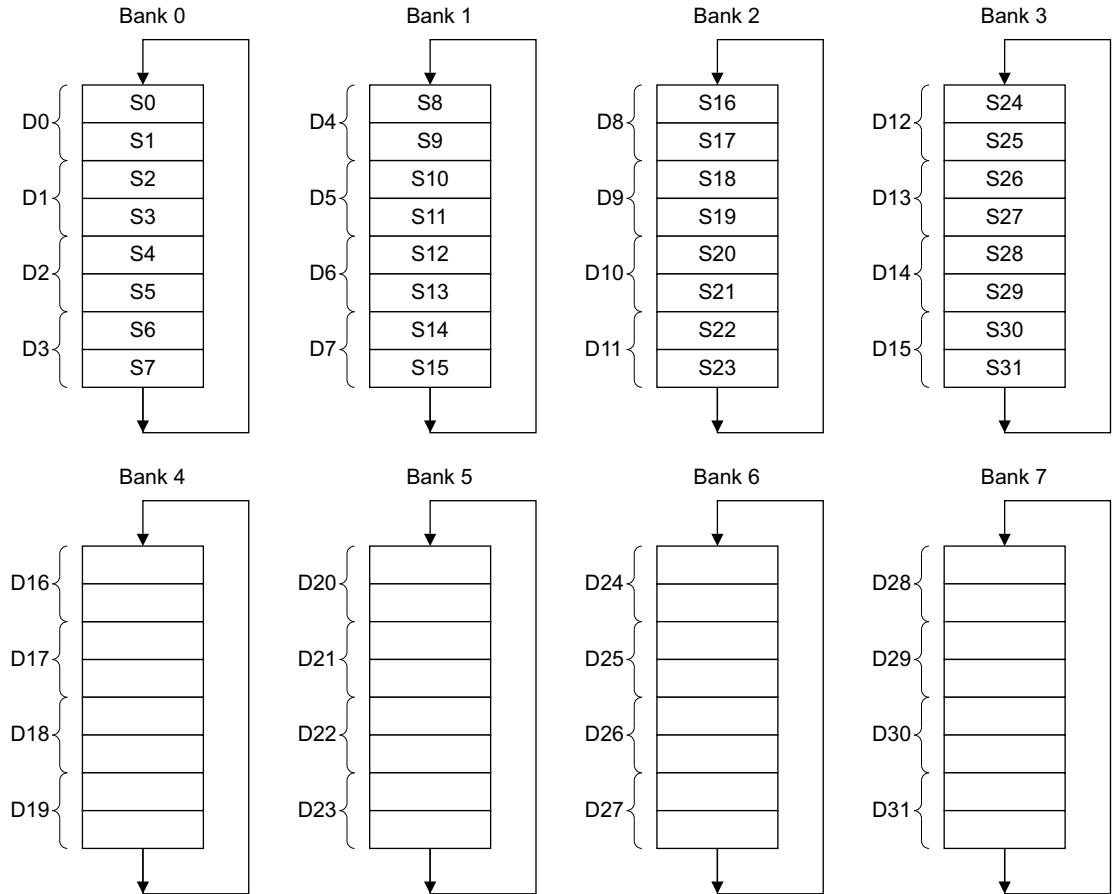
### 13.3.1 About register banks

As Figure 13-2 on page 13-7 shows, the register file is divided into four banks with eight registers in each bank for single-precision instructions and eight banks with four registers per bank for double-precision instructions. CDP instructions access the banks in a circular manner. Load and store multiple instructions do not access the registers in a circular manner but treat the register file as a linearly ordered structure.

The VFPv3 architecture adds 16 double-precision registers, making use of the additional register addressing bits currently used to specify single-precision registers. The first 16 registers, D0 through D15, in the NEON register file provides the same functionality as the register file defined in the VFPv2 architecture. VFPv3 adds 16 new double-precision registers, D16 through D31, which provides a second set of 16 double-precision registers. These registers behave in vector mode in an identical manner to the lower 16 registers, with bank 4 specified as registers D16-D19, bank 5 specified as registers D20-D23, bank 6 specified as registers D24-D27, and bank 7 specified as D28-D31. Bank 4 of the second set of registers has the same characteristics when used in short vector instructions as bank 0 of the first set of registers.

Short vector operations on double-precision data support vector lengths of two through four iterations. The additional registers provides the capability to double-buffer double-precision operations in a similar way as is available for single-precision operations.

See the *ARM Architecture Reference Manual* for more information on VFP addressing modes.



**Figure 13-2 Register banks**

A short vector CDP operation that has a source or destination vector crossing a bank boundary wraps around and accesses the first register in the bank.

Example 13-1 on page 13-8 shows the iterations of the following short vector add instruction:

```
FADDS S11, S22, S31
```

In this instruction, the `LEN` field contains `b101`, selecting a vector length of six iterations, and the `STRIDE` field contains `b00`, selecting a vector stride of one.

See *Floating-Point Status and Control Register, FPSCR* on page 13-14 for details of the `LEN` and `STRIDE` fields and the FPSCR Register.

**Example 13-1 Register bank wrapping**


---

```

FADDS S11, S22, S31      ; 1st iteration
FADDS S12, S23, S24      ; 2nd iteration. The 2nd source vector wraps around
                          ; and accesses the 1st register in the 4th bank
FADDS S13, S16, S25      ; 3rd iteration. The 1st source vector wraps around
                          ; and accesses the 1st register in the 3rd bank
FADDS S14, S17, S26      ; 4th iteration
FADDS S15, S18, S27      ; 5th iteration
FADDS S8, S19, S28       ; 6th and last iteration. The destination vector
                          ; wraps around and writes to the 1st register in the
                          ; 2nd bank

```

---

**13.3.2 Operations using register banks**

The register file organization supports four types of operations described in the following sections:

- *Scalar-only instructions*
- *Short vector-only instructions* on page 13-9
- *Short vector instructions with scalar source* on page 13-9
- *Scalar instructions in short vector mode* on page 13-10.

**Scalar-only instructions**

An instruction is a scalar-only operation if the operands are treated as scalars and the result is a scalar.

Clearing the LEN field in the FPSCR Register selects a vector length of one iteration. For example, if the LEN field contains b000, then the following operation writes the sum of the single-precision values in S21 and S22 to S12:

```
FADDS S12, S21, S22
```

Some instructions can operate only on scalar data regardless of the value in the LEN field. These instructions are:

**Compare operations**

FCMPS/D, FCOMPZS/D, FCMPEP/D, and FCMPEZS/D.

**Integer conversions**

FTOUIS/D, FTUOIZS/D, FTOSIS/D, FTOSIZS/D, FUITOS/D, and FSITOS/D.

**Precision conversions**

FCVTDS and FCVTSD.

### Fixed-point instructions

FSHTOS/D, FSCTOS/D, FUHTOS/D, FULTOS/D, FTOSHS/D, FTOSLS/D, FTOUHS/D, and FTOULS/D.

### Short vector-only instructions

Vector-only instructions require that the value in the LEN field is nonzero, and that the destination and Fm registers are not in bank 0.

Example 13-2 shows the iterations of the following short vector instruction:

```
FMACS S16, S0, S8
```

In the example, the LEN field contains b011, selecting a vector length of four iterations, and the STRIDE field contains b00, selecting a vector stride of one.

#### Example 13-2 Short vector instruction

---

```
FMACS S16, S0, S8 ; 1st iteration
FMACS S17, S1, S9 ; 2nd iteration
FMACS S18, S2, S10 ; 3rd iteration
FMACS S19, S3, S11 ; 4th and last iteration
```

---

### Short vector instructions with scalar source

The VFPv3 architecture enables a vector to be operated on by a scalar operand. The destination must be a vector, that is, not in bank 0, and the Fm operand must be in bank 0.

Example 13-3 shows the iterations of the following short vector instruction with a scalar source:

```
FMULD D12, D8, D2
```

In the example, the LEN field contains b001, selecting a vector length of two iterations, and the STRIDE field contains b00, selecting a vector stride of one.

#### Example 13-3 Short vector instruction with scalar source

---

```
FMULD D12, D8, D2 ; 1st iteration
FMULD D13, D9, D2 ; 2nd and last iteration
```

---

This scales the two source registers, D8 and D9, by the value in D2 and writes the new values to D12 and D13.

### Scalar instructions in short vector mode

You can mix scalar and short vector operations by carefully selecting the source and destination registers. If the destination is in bank 0 or bank 4, the operation is scalar-only regardless of the value in the LEN field. You do not have to change the LEN field from a nonzero value to b000 to perform scalar operations.

Example 13-4 shows the sequence of operations for the following instructions:

```
FABSD D4, D8
FADDS S0, S0, S31
FMULS S24, S26, S1
```

In the example, the LEN field contains b001, selecting a vector length of two iterations, and the STRIDE field contains b00, selecting a vector stride of one.

#### Example 13-4 Scalar operation in short vector mode

---

```
FABSD D4, D8      ; vector DP ABS operation on regs (D8, D9) to (D4, D5)
FABSD D5, D9
FADDS S0, S0, S31 ; scalar increment of S0 by S31
FMULS S24, S26, S1 ; vector (S26, S27) scaled by S1 and written to (S24, S25)
FMULS S25, S27, S1
```

---

The tables that follow show the four types of operations possible in the VFPv3 architecture. In the tables, *Any* refers to the availability of all registers in the precision for the specified operand. *S* refers to a scalar operand with only a single register. *V* refers to a vector operand with multiple registers. Table 13-1 describes single-precision three-operand register usage.

**Table 13-1 Single-precision three-operand register usage**

LEN field	Fd	Fn	Fm	Operation type
b000	Any	Any	Any	S = S op S OR S = S S S
Nonzero	0-7	Any	Any	S = S op S OR S = S S S
Nonzero	8-31	Any	0-7	V = V op S OR V = V V S
Nonzero	8-31	Any	8-31	V = V op V OR V = V V V

Table 13-2 describes single-precision two-operand register usage.

**Table 13-2 Single-precision two-operand register usage**

LEN field	Fd	Fm	Operation type
b000	Any	Any	S = op S
Nonzero	0-7	Any	S = op S
Nonzero	8-31	0-7	V = op S
Nonzero	8-31	8-31	V = op V

Table 13-3 describes double-precision three-operand register usage.

**Table 13-3 Double-precision three-operand register usage**

LEN field	Fd	Fn	Fm	Operation type
b000	Any	Any	Any	S = S op S OR S = S S S
Nonzero	0-3, 16-19	Any	Any	S = S op S OR S = S S S
Nonzero	4-15	Any	0-3	V = V op S OR V = V V S
Nonzero	4-15	Any	4-15	V = V op V OR V = V V V

Table 13-4 describes double-precision two-operand register usage.

**Table 13-4 Double-precision two-operand register usage**

LEN field	Fd	Fm	Operation type
b000	Any	Any	S = op S
Nonzero	0-3, 16-19	Any	S = op S
Nonzero	4-15, 20-31	0-3	V = op S
Nonzero	4-15, 20-31	4-15	V = op V

## 13.4 System registers

The VFPv3 architecture describes the following system registers:

- *Floating-Point System ID Register, FPSID* on page 13-13
- *Floating-Point Status and Control Register, FPSCR* on page 13-14
- *Floating-point exception Register, FPEXC* on page 13-17
- *Media and VFP Feature Registers, MVFR0 and MVFR1* on page 13-18.

Table 13-5 shows the NEON and VFP system registers.

**Table 13-5 NEON and VFP system registers**

Register	FMXR/FMRX <reg> field	Access type	Reset state
Floating-Point System ID Register, FPSID	b0000	Read-only	0x410330c3
Floating-Point Status and Control Register, FPSCR	b0001	Read/write	0x00000000
Floating-Point Exception Register, FPEXC	b1000	Read/write	0x00000000
Media and VFP Feature Register 0, MVFR0	b0111	Read-only	0x11110222
Media and VFP Feature Register 1, MVFR1	b0110	Read-only	0x00011111

———— **Note** ————

The FPSID, MVFR0, and MVFR1 Registers are read-only. Attempts to write these registers are ignored.

Table 13-6 shows the processor modes for accessing the NEON and VFP system registers.

**Table 13-6 Accessing NEON and VFP system registers**

Register	Privileged access		User access	
	FPEXC.EN=0	FPEXC.EN=1	FPEXC.EN=0	FPEXC.EN=1
FPSID	Permitted	Permitted	Not permitted	Not permitted
FPSCR	Not permitted	Permitted	Not permitted	Permitted
MVFR0, MVFR1	Permitted	Permitted	Not permitted	Not permitted
FPEXC	Permitted	Permitted	Not permitted	Not permitted



Table 13-6 on page 13-12 shows that a privileged mode is sometimes required to access a NEON and VFP system register. When a privileged mode is required, an instruction that tries to access a register in a nonprivileged mode takes the Undefined Instruction trap.

For a NEON or VFP system register to be accessible, it must follow the rules in Table 13-6 on page 13-12 and it must also be accessible by the Coprocessor Access Control Register and the Nonsecure Access Control Register. See *c1, Coprocessor Access Control Register* on page 3-67 and *c1, Nonsecure Access Control Register* on page 3-73 for more information.

———— **Note** —————

All hardware ID information is now privileged access only.

**FPSID is privileged access only**

This is a change in VFPv3. In VFPv2 implementation, the FPSID register can be accessed in all modes.

**MVFR registers are privileged access only**

User code must issue a system call to determine what features are supported.

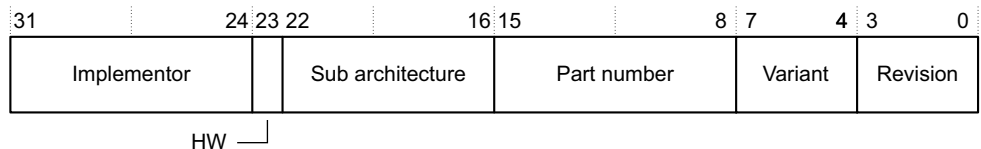
The following sections describe the NEON and VFP system registers:

- *Floating-Point System ID Register, FPSID*
- *Floating-Point Status and Control Register, FPSCR* on page 13-14
- *Floating-point exception Register, FPEXC* on page 13-17
- *Media and VFP Feature Registers, MVFR0 and MVFR1* on page 13-18.

### 13.4.1 Floating-Point System ID Register, FPSID

The FPSID Register is a read-only register that must be accessed in privileged mode only. It indicates which NEON and VFP implementation is being used.

Figure 13-3 shows the bit arrangement of the FPSID Register.



**Figure 13-3 Floating-Point System ID Register format**

Table 13-7 shows how the bit values correspond with the FPSID Register functions.

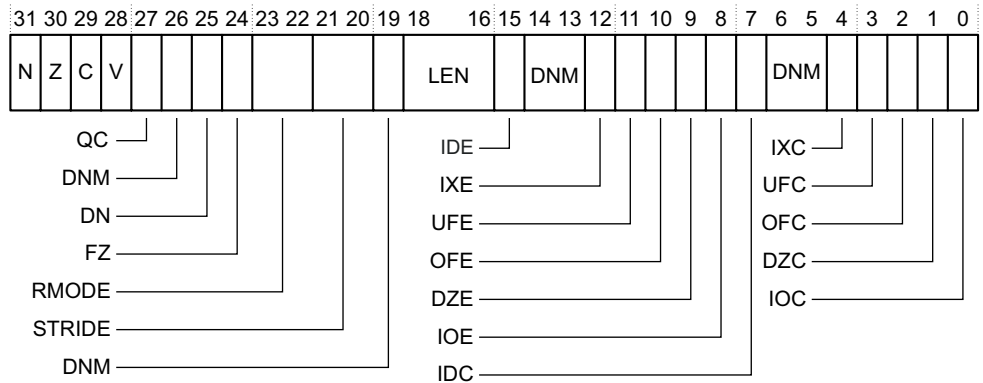
**Table 13-7 FPSID Register bit functions**

Bits	Field	Function
[31:24]	Implementor	ARM: 0x41 = A
[23]	Hardware or software	0 = hardware implementation
[22:16]	Sub architecture version	The Null VFP sub-architecture: 0x03
[15:8]	Part number	VFP: 0x30
[7:4]	Variant	VFP interface: 0xC
[3:0]	Revision	Version: 0x3

### 13.4.2 Floating-Point Status and Control Register, FPSCR

FPSCR is a read/write register that can be accessed in both privileged and unprivileged modes. All bits described as DNM in Figure 13-4 on page 13-15 are reserved for future expansion. They must be initialized to zeros. To ensure that these bits are not modified, code other than initialization code must use read/modify/write techniques when writing to FPSCR. Failure to observe this rule can cause Unpredictable results in future systems.

Figure 13-4 on page 13-15 shows the bit arrangement of the FPSCR Register.



**Figure 13-4 Floating-Point Status and Control Register format**

Table 13-8 shows how the bit values correspond with the FPSCR Register functions.

**Table 13-8 FPSCR Register bit functions**

Bits	Field	Function
[31]	N	Set if comparison produces a <i>less than</i> result
[30]	Z	Set if comparison produces an <i>equal</i> result
[29]	C	Set if comparison produces an <i>equal, greater than, or unordered</i> result
[28]	V	Set if comparison produces an <i>unordered</i> result
[27]	QC	Saturation cumulative flag
[26]	DNM	Do Not Modify
[25]	DN	Default NaN mode enable bit: 0 = default NaN mode disabled 1 = default NaN mode enabled.
[24]	FZ	Flush-to-zero mode enable bit: 0 = flush-to-zero mode disabled 1 = flush-to-zero mode enabled.
[23:22]	RMODE	Rounding mode control field: b00 = round to nearest (RN) mode b01 = round towards plus infinity (RP) mode b10 = round towards minus infinity (RM) mode b11 = round towards zero (RZ) mode.
[21:20]	STRIDE	See <i>Vector length and stride control</i> on page 13-16
[19]	DNM	Do Not Modify
[18:16]	LEN	See <i>Vector length and stride control</i> on page 13-16

**Table 13-8 FPSCR Register bit functions (continued)**

<b>Bits</b>	<b>Field</b>	<b>Function</b>
[15]	IDE	Input Subnormal exception enable bit
[14:13]	DNM	Do Not Modify
[12]	IXE	Inexact exception enable bit
[11]	UFE	Underflow exception enable bit
[10]	OFE	Overflow exception enable bit
[9]	DZE	Division by Zero exception enable bit
[8]	IOE	Invalid Operation exception enable bit
[7]	IDC	Input Subnormal cumulative flag
[6:5]	DNM	Do Not Modify
[4]	IXC	Inexact cumulative flag
[3]	UFC	Underflow cumulative flag
[2]	OFC	Overflow cumulative flag
[1]	DZC	Division by Zero cumulative flag
[0]	IOC	Invalid Operation cumulative flag

### Vector length and stride control

FPSCR[18:16] is the LEN field and controls the vector length for VFP instructions that operate on short vectors. The vector length is the number of iterations in a short vector instruction.

FPSCR[21:20] is the STRIDE field and controls the vector stride. The vector stride is the increment value used to select the registers involved in the next iteration of the short vector instruction.

The rules for vector operation do not allow a vector to use the same register more than once. LEN and STRIDE combinations that use a register more than once produce Unpredictable results, as Table 13-9 shows. Some combinations that work normally in single-precision short vector instructions cause Unpredictable results in double-precision instructions.

**Table 13-9 Vector length and stride combinations**

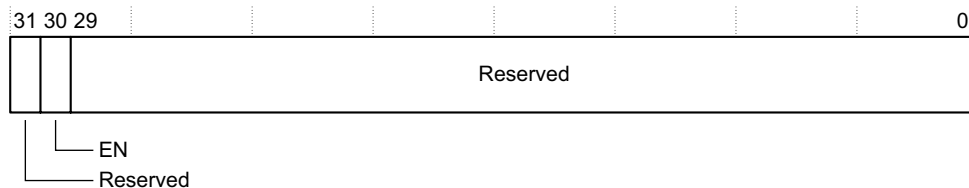
LEN	Vector length	STRIDE	Vector stride	Single-precision vector instructions	Double-precision vector instructions
b000	1	b00	-	All instructions are scalar	All instructions are scalar
b000	1	b11	-	Unpredictable	Unpredictable
b001	2	b00	1	Work normally	Work normally
b001	2	b11	2	Work normally	Work normally
b010	3	b00	1	Work normally	Work normally
b010	3	b11	2	Work normally	Unpredictable
b011	4	b00	1	Work normally	Work normally
b011	4	b11	2	Work normally	Unpredictable
b100	5	b00	1	Work normally	Unpredictable
b100	5	b11	2	Unpredictable	Unpredictable
b101	6	b00	1	Work normally	Unpredictable
b101	6	b11	2	Unpredictable	Unpredictable
b110	7	b00	1	Work normally	Unpredictable
b110	7	b11	2	Unpredictable	Unpredictable
b111	8	b00	1	Work normally	Unpredictable
b111	8	b11	2	Unpredictable	Unpredictable

### 13.4.3 Floating-point exception Register, FPEXC

The FPEXC Register is accessible in privileged modes only.

The EN bit, FPEXC[30], is the NEON and VFP enable bit. Clearing EN disables the NEON and VFP coprocessor. The EN bit is cleared to 0 on reset.

Figure 13-5 on page 13-18 shows the bit arrangement of the FPEXC Register.



**Figure 13-5 Floating-Point Exception Register format**

Table 13-10 shows how the bit values correspond with the FPExc Register functions.

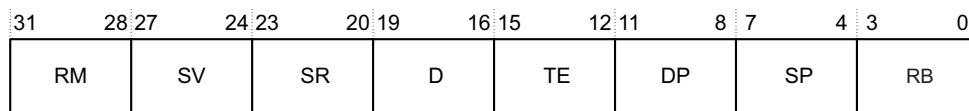
**Table 13-10 Floating-Point Exception Register bit functions**

Bits	Field	Function
[31]	-	Reserved.
[30]	EN	NEON and VFP enable bit. Setting the EN bit to 1 enables the NEON and VFP coprocessor. Reset clears EN to 0.
[29:0]	-	Reserved.

#### 13.4.4 Media and VFP Feature Registers, MVFR0 and MVFR1

The Media and VFP Feature Registers, MVFR0 and MVFR1, describe the features supported by the NEON and VFP coprocessor. These registers are accessible in privileged modes only.

Figure 13-6 shows the bit arrangement of the MVFR0 Register.



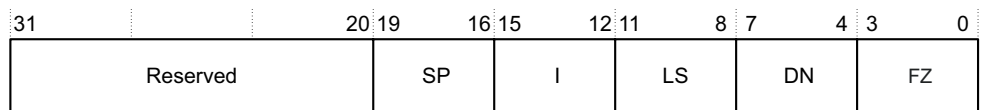
**Figure 13-6 MVFR0 Register format**

Table 13-11 shows how the bit values correspond with the MVFR0 Register functions.

**Table 13-11 MVFR0 Register bit functions**

Bits	Field	Function
[31:28]	RM	All VFP rounding modes supported: 0x1
[27:24]	SV	VFP short vector supported: 0x1
[23:20]	SR	VFP hardware square root supported: 0x1
[19:16]	D	VFP hardware divide supported: 0x1
[15:12]	TE	Only untrapped exception handling can be selected: 0x0
[11:8]	DP	Double precision supported in VFPv3: 0x2
[7:4]	SP	Single precision supported in VFPv3: 0x2
[3:0]	RB	32x64-bit media register bank supported: 0x2

Figure 13-7 shows the bit arrangement of the MVFR1 Register.



**Figure 13-7 MVFR1 Register format**

Table 13-12 shows how the bit values correspond with the MVFR1 Register.

**Table 13-12 MVFR1 Register bit functions**

<b>Bits</b>	<b>Field</b>	<b>Function</b>
[31:20]	-	Reserved
[19:16]	SP	Single precision floating-point instructions supported for NEON: 0x1
[15:12]	I	Integer instructions supported for NEON: 0x1
[11:8]	LS	Load/store instructions supported for NEON: 0x1
[7:4]	DN	Propagation of NaN values supported for VFP: 0x1
[3:0]	FZ	Full denormal arithmetic supported for VFP: 0x1



## 13.5 Modes of operation

To accommodate a variety of applications, the VFP coprocessor provides four modes of operation:

- *Full-compliance mode*
- *Flush-to-zero mode*
- *Default NaN mode*
- *RunFast mode.*

### 13.5.1 Full-compliance mode

When the VFP coprocessor is in full-compliance mode, all operations are processed according to the IEEE 754 standard in hardware.

### 13.5.2 Flush-to-zero mode

Setting the FZ bit, FPSCR[24], enables flush-to-zero mode and increases performance on very small inputs and results. In flush-to-zero mode, the VFP coprocessor treats all subnormal input operands of arithmetic CDP operations as zeros in the operation. Exceptions that result from a zero operand are signaled appropriately. FABS, FNEG, and FCPY are not considered arithmetic CDP operations and are not affected by flush-to-zero mode. A result that is *tiny*, as described in the IEEE 754 standard, for the destination precision is smaller in magnitude than the minimum normal value *before rounding* and is replaced with a zero. The IDC flag, FPSCR[7], indicates when an input flush occurs. The UFC flag, FPSCR[3], indicates when a result flush occurs.

### 13.5.3 Default NaN mode

Setting the DN bit, FPSCR[25], enables default NaN mode. In default NaN mode, the result of any operation that involves an input NaN or generated a NaN result returns the default NaN. Propagation of the fraction bits is maintained only by FABS, FNEG, and FCPY operations, all other CDP operations ignore any information in the fraction bits of an input NaN.

### 13.5.4 RunFast mode

RunFast mode is the combination of the following conditions:

- the VFP coprocessor is in flush-to-zero mode
- the VFP coprocessor is in default NaN mode
- all exception enable bits are cleared to 0.

In RunFast mode the VFP coprocessor:

- processes subnormal input operands as zeros
- processes results that are tiny before rounding, that is, between the positive and negative minimum normal values for the destination precision, as zeros
- processes input NaNs as default NaNs
- returns the default result specified by the IEEE 754 standard for overflow, division by zero, invalid operation, or inexact operation conditions fully in hardware and without additional latency.

## 13.6 Compliance with the IEEE 754 standard

The VFP coprocessor is fully compliant with the IEEE 754 standard in hardware, no support code is required to achieve this compliance.

See the *ARM Architecture Reference Manual* for information about VFP architectural compliance with the IEEE 754 standard.

### 13.6.1 Complete implementation of the IEEE 754 standard

The following operations from the IEEE 754 standard are not supplied by the VFP instruction set:

- remainder
- round floating-point number to integer-valued floating-point number
- binary-to-decimal conversions
- decimal-to-binary conversions
- direct comparison of single-precision and double-precision values.

For complete implementation of the IEEE 754 standard, the VFP coprocessor must be augmented with library functions that implement these operations. See *Application Note 98, VFP Support Code* for details of the available library functions.

### 13.6.2 IEEE 754 standard implementation choices

Some of the implementation choices permitted by the IEEE 754 standard and used in the VFPv3 architecture are described in the *ARM Architecture Reference Manual*.

#### NaN handling

Any single-precision or double-precision values with the maximum exponent field value and a nonzero fraction field are valid NaNs. A most significant fraction bit of zero indicates a *Signaling NaN* (SNaN). A one indicates a *Quiet NaN* (QNaN). Two NaN values are treated as different NaNs if they differ in any bit. Table 13-13 shows the default NaN values in both single and double precision.

**Table 13-13 Default NaN values**

	Single-precision	Double-precision
Sign	0	0
Exponent	0xFF	0x7FF
Fraction	bit [22] = 1 bits [21:0] are all zeros	bit [51] = 1 bits [50:0] are all zeros

Any SNaN passed as input to an operation causes an Invalid Operation exception and sets the IOC flag, FPSCR[0], to 1. A default QNaN is written to the destination register. The rules for cases involving multiple NaN operands are in the *ARM Architecture Reference Manual*.

Processing of input NaNs for ARM floating-point coprocessors and libraries is defined as follows:

- In full-compliance mode, NaNs are handled according to the *ARM Architecture Reference Manual*. The hardware processes the NaNs directly for arithmetic CDP instructions. For data transfer operations, NaNs are transferred without raising the Invalid Operation exception. For the non-arithmetic CDP instructions, FABS, FNEG, and FCPY, NaNs are copied, with a change of sign if specified in the instructions, without causing the Invalid Operation exception.
- In default NaN mode, NaNs are handled completely within the hardware. SNaNs in an arithmetic CDP operation set the IOC flag, FPSCR[0], to 1. NaN handling by data transfer and non-arithmetic CDP instructions is the same as in full-compliance mode. Arithmetic CDP instructions involving NaN operands return the default NaN regardless of the fractions of any NaN operands.

Table 13-14 summarizes the effects of NaN operands on instruction execution.

**Table 13-14 QNaN and SNaN handling**

Instruction type	Default NaN mode	With QNaN operand	With SNaN operand
Arithmetic CDP	Off	The QNaN or one of the QNaN operands, if there is more than one, is returned according to the rules given in the <i>ARM Architecture Reference Manual</i> .	IOC <sup>a</sup> set to 1. The SNaN is quieted and the result NaN is determined by the rules given in the <i>ARM Architecture Reference Manual</i> .
	On	Default NaN returns.	IOC <sup>b</sup> set to 1. Default NaN returns.
Non-arithmetic CDP	Off	NaN passes to destination with sign changed as appropriate.	
	On		
FCMP(Z)	-	Unordered compare.	IOC set to 1. Unordered compare.
FCMPE(Z)	-	IOC set to 1. Unordered compare.	IOC set to 1. Unordered compare.
Load/store	Off	All NaNs transferred.	
	On		

a. IOC is the Invalid Operation exception flag, FPSCR[0].

## Comparisons

Comparison results modify condition code flags in the FPSCR Register. The FMSTAT instruction transfers the current condition code flags in the FPSCR Register to the CPSR Register. See the *ARM Architecture Reference Manual* for more information. The condition code flags used are chosen so that subsequent conditional execution of ARM instructions can test the predicates defined in the IEEE 754 standard.

The VFP coprocessor handles all comparisons of numeric and reserved values in hardware, generating the appropriate condition code depending on whether the result is less than, equal to, or greater than.

The VFP coprocessor supports:

### Compare operations

The compare operations are FCMPs, FCMPZs, FCMPD, and FCMPZD.

A compare instruction involving a QNaN produces an unordered result. An SNaN produces an unordered result and generates an Invalid Operation exception.

### Compare with exception operations

The compare with exception operations are FCMPEs, FCMPEZs, FCMPEd, and FCMPEZD.

A compare with exception operation involving either an SNaN or a QNaN produces an unordered result and generates an Invalid Operation exception.

## Underflow

In the generation of Underflow exceptions, the *before rounding* form of *tininess* and the *inexact result* form of *loss of accuracy* as described in the IEEE 754 standard, are used.

In flush-to-zero mode, results that are tiny before rounding, as described in the IEEE 754 standard, are flushed to a zero, and the UFC flag, FPSCR[3], is set to 1. See the *ARM Architecture Reference Manual* for information on flush-to-zero mode.

When the VFP coprocessor is not in flush-to-zero mode, operations are performed on subnormal operands. If the operation does not produce a tiny result, it returns the computed result, and the UFC flag, FPSCR[3], is not set to 1. The IXC flag, FPSCR[4], is set to 1 if the operation is inexact. If the operation produces a tiny result, the result is a subnormal or zero value, and the UFC flag, FPSCR[3], is set to 1.

## Exceptions

The VFP coprocessor implements the VFPv3 architecture and sets all exception status bits in the FPSCR register as required for each instruction. The VFP coprocessor does not support user-mode traps. The VFP coprocessor ignores exception enable bits in the FPSCR register. FPSCR bits [15, 12:8] are read-only and read-as-zero. Writes to these enable bits are ignored.

# Chapter 14

## Embedded Trace Macrocell

This chapter describes the ETM. It contains the following sections:

- *About the ETM* on page 14-2
- *ETM configuration* on page 14-6
- *ETM register summary* on page 14-8
- *ETM register descriptions* on page 14-10
- *Precision of TraceEnable and ViewData* on page 14-23
- *Exact match bit* on page 14-26
- *Context ID tracing* on page 14-28
- *Instrumentation instructions* on page 14-29
- *Idle state control* on page 14-30
- *Interaction with the Performance Monitoring Unit* on page 14-32.

## 14.1 About the ETM

The ETM is a CoreSight™ component designed for use with the CoreSight Design Kit. CoreSight is the ARM extensible, system-wide debug and trace architecture. The Cortex-A8 processor implements the ETM architecture v3.3.

For more information about CoreSight and ETM functionality, see the *Embedded Trace Macrocell Architecture Specification* and the CoreSight documentation listed in *Additional reading* on page xxxii.

### 14.1.1 ETM features

The ETM has the following main features:

#### Core interface module

The core interface module monitors the behavior of the processor.

#### Trace generation

The ETM generates a real-time trace that can be configured to include:

- instruction tracing containing:
  - the addresses of executed instructions
  - passed or failed condition codes of the instructions
  - information about exceptions
  - context IDs.
- data address tracing containing the addresses of data transfers as viewed by the ARM architecture.

———— **Note** —————

The Cortex-A8 ETM does not support tracing of data values.

---

#### Filtering and triggering resources

You can filter the ETM trace such as configuring it to trace only instructions or data transfers in certain address ranges. You can also configure the ETM to filter based on the values of data transfers even though these cannot be traced. More complicated logic analyzer style filtering options are also available.

The ETM can also generate a trigger that is a signal to the trace capture device to stop capturing trace.



**Main FIFO** The trace generated by ETM is in a highly compressed form. The main FIFO enables bursts caused by the trace compression to be flattened out. When the FIFO becomes full, the FIFO signals an overflow. The trace generation logic does not generate any new trace until the FIFO has emptied. This causes a gap in the trace when viewed in the debugger.

You can also configure the ETM to suppress data address tracing when the FIFO is close to being full. This can prevent overflows from occurring.

### AMBA 3 ATB interface

The ETM outputs trace using the AMBA 3 *Advanced Trace Bus* (ATB) interface. See the *CoreSight Architecture Specification* for more information on AMBA 3 ATB.

You can output trace asynchronously to the core clock.

### AMBA 3 APB interface

The AMBA 3 *Advanced Peripheral Bus* (APB) interface enables access to the ETM, CTI, and the debug registers. The APB interface is compatible with the CoreSight architecture which is the ARM architecture for multi-processor trace and debug. See the *CoreSight Architecture Specification* for more information.

## 14.1.2 The debug environment

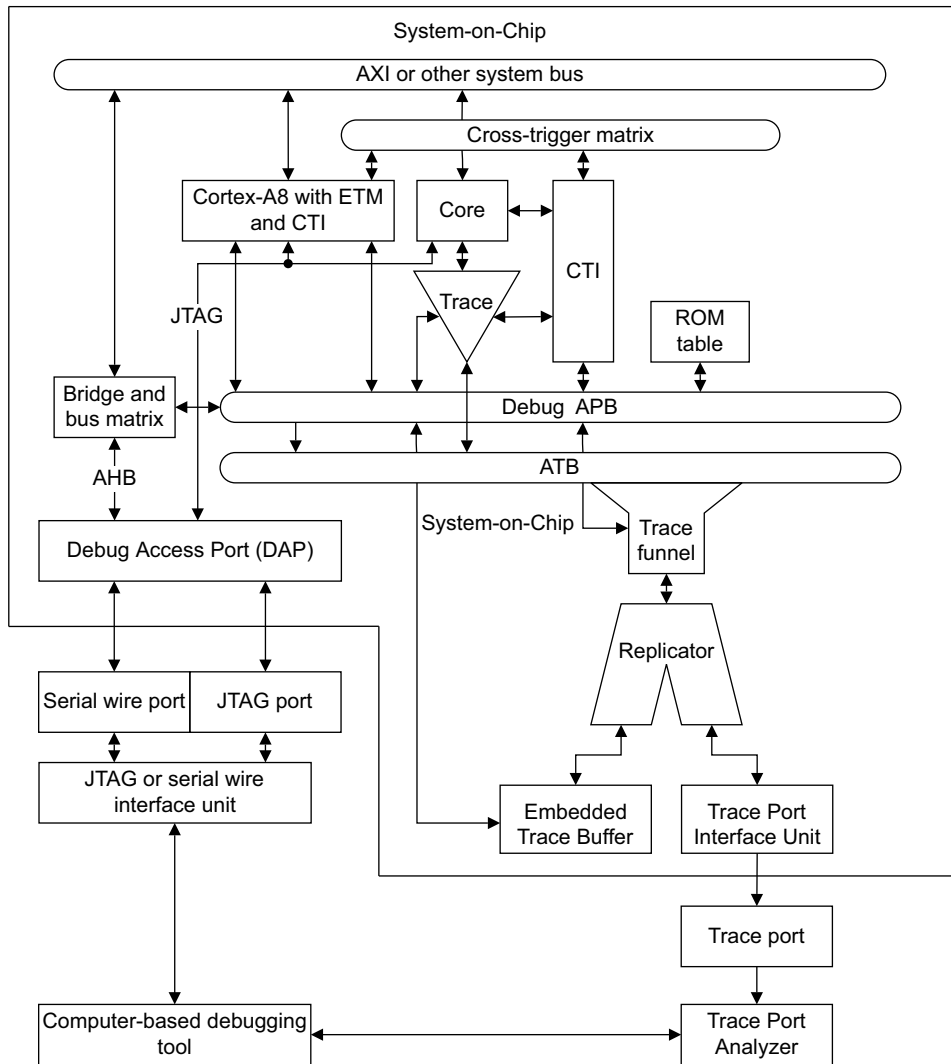
A software debugger provides the user interface to the ETM. The debugger enables all the ETM facilities such as the trace port to be configured. The debugger also displays the trace information that has been captured.

The ETM compresses the trace information and outputs it to the AMBA 3 ATB interface. The ETM can then either:

- Export the trace information through a narrow trace port. An external *Trace Port Analyzer* (TPA) captures the trace information as Figure 14-1 on page 14-4 shows.
- Write the trace information directly to an on-chip *Embedded Trace Buffer* (ETB). The trace is read out at low speed using the JTAG or Serial Wire interface when the trace capture is complete as Figure 14-1 on page 14-4 shows.

When the trace is captured, the debugger extracts the information from the TPA and decompresses it to provide a full disassembly, with symbols, of the code that was executed. The debugger can also link this back to the original high-level source code, to provide you with a visualization of how the code was executed on the target system.

Figure 14-1 shows how the ETM fits into the CoreSight debug environment. See the *CoreSight Architecture Specification* for more information.



**Figure 14-1 Example CoreSight debug environment**

In Figure 14-1 on page 14-4, the ETM and the Cross Trigger Interface, are part of a CoreSight system consisting of other cores with their own ETMs, and various other trace sources. The CoreSight components are programmed using the *Debug Access Port*

(DAP) through the APB programming bus, and trace is output over the ATB trace bus. This is then either exported through the *Trace Port Interface Unit* (TPIU), or stored in the ETB.

See the *Embedded Trace Macrocell Architecture Specification* for information about the trace protocol, and about controlling tracing using triggering and filtering resources.

### 14.1.3 NEON

The ETM ignores data transfers to and from the NEON register file. The addresses and data values of these transfers are not traced, and have no effect on the address comparators.

## 14.2 ETM configuration

ETMv3.3 permits a number of configurations. Table 14-1 shows the options implemented in the Cortex-A8 ETM.

**Table 14-1 ETM implementation**

<b>Resource description</b>	<b>Configuration</b>
Instruction trace	Yes
Data address trace	Yes
Data value trace	No
Jazelle trace	-
Address comparator pairs	4
Data comparators	2
Context ID comparators	1
Sequencer	Yes
Start/stop block	Yes
EmbeddedICE comparators	0
External inputs	4
External outputs	2
Extended external inputs	49
Extended external input selectors	2
Instrumentation resources	4
FIFOFULL	No
FIFOFULL level setting	N/A
Branch broadcasting	Yes
ASIC Control Register (bits)	8
Data suppression	Yes
Software access to registers	Memory
Readable registers	Yes

**Table 14-1 ETM implementation (continued)**

<b>Resource description</b>	<b>Configuration</b>
FIFO size	128 bytes
Minimum port size	32
Maximum port size	32
Port modes	Dynamic
Asynchronous ATB interface	Yes
Load pc first	No
Fetch comparisons	No

## 14.3 ETM register summary

The ETM registers are defined in the *ETM Architecture Specification*.

Table 14-2 shows the values of the Identification registers and the Integration registers that are implementation-defined and are not described in the *ETM Architecture Specification*.

**Table 14-2 ETM register summary**

Register name	Base offset <sup>a</sup>	Type	Reset value	Description
Configuration Code	0x004	R	0x8D294024	<i>Configuration Code Register</i> on page 14-11
ID	0x1E4	R	0x410CF236	<i>ID Register</i> on page 14-10
Configuration Code Extension	0x1E8	R	0x000008A2	<i>Configuration Code Extension Register</i> on page 14-13
ITMISCOUT	0xEDC	W	-	<i>ITMISCOUT Register</i> on page 14-18
ITMISCIN	0xEE0	R	.. <sup>b</sup>	<i>ITMISCIN Register</i> on page 14-18
ITTRIGGER	0xEE8	W	-	<i>ITTRIGGER Register</i> on page 14-19
ITATBDATA0	0xEEC	W	-	<i>ITATBDATA0 Register</i> on page 14-19
ITATBCTR2	0xEF0	R	.. <sup>b</sup>	<i>ITATBCTR2 Register</i> on page 14-20
ITATBCTR1	0xEF4	W	-	<i>ITATBCTR1 Register</i> on page 14-21
ITATBCTR0	0xEF8	W	-	<i>ITATBCTR0 Register</i> on page 14-21
PeripheralID4	0xFD0	R	0x00000004	<i>Peripheral Identification Registers</i> on page 14-14
PeripheralID5	0xFD4	R	0x00000000	
PeripheralID6	0xFD8	R	0x00000000	
PeripheralID7	0xFDC	R	0x00000000	
PeripheralID0	0xFE0	R	0x00000021	
PeripheralID1	0xFE4	R	0x000000B9	
PeripheralID2	0xFE8	R	0x0000006B	
PeripheralID3	0xFEC	R	0x00000010	

Table 14-2 ETM register summary (continued)

Register name	Base offset <sup>a</sup>	Type	Reset value	Description
ComponentID0	0xFF0	R	0x00000000	<i>Component Identification Registers</i> on page 14-15
ComponentID1	0xFF4	R	0x00000090	
ComponentID2	0xFF8	R	0x00000005	
ComponentID3	0xFFC	R	0x000000B1	

- a. The value given in the Base offset column is the address offset for memory-mapped access. To get the register number used in the *ETM Architecture Specification*, divide this offset by four.
- b. The values of these read-only registers depend on the signals on external pins of the ETM. Therefore it is not possible to define the register reset values.

## 14.4 ETM register descriptions

This section describes the following registers:

- *ID Register*
- *Configuration Code Register* on page 14-11
- *Configuration Code Extension Register* on page 14-13
- *Peripheral Identification Registers* on page 14-14
- *Component Identification Registers* on page 14-15
- *Integration Test Registers* on page 14-16.

For more details about these registers and the other registers implemented by the ETM, see the *Embedded Trace Macrocell Architecture Specification*.

### 14.4.1 ID Register

The ID Register, at offset 0x1E4, is a 32-bit read-only register that provides information about the ETM architecture version and options supported. Figure 14-2 shows the bit arrangement of the ID Register.

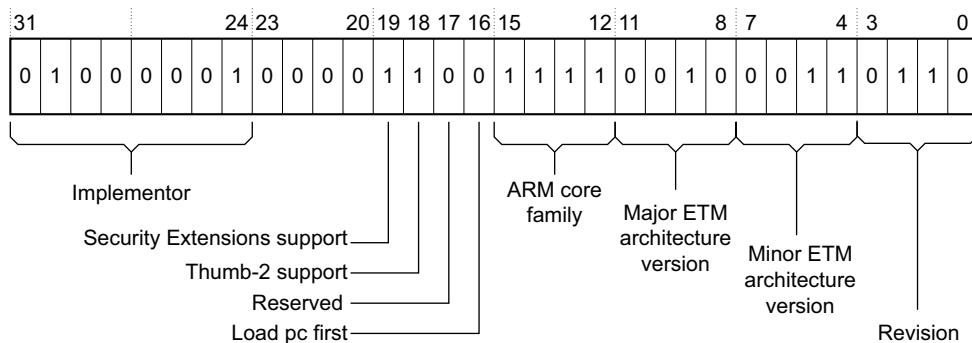


Figure 14-2 ID Register format



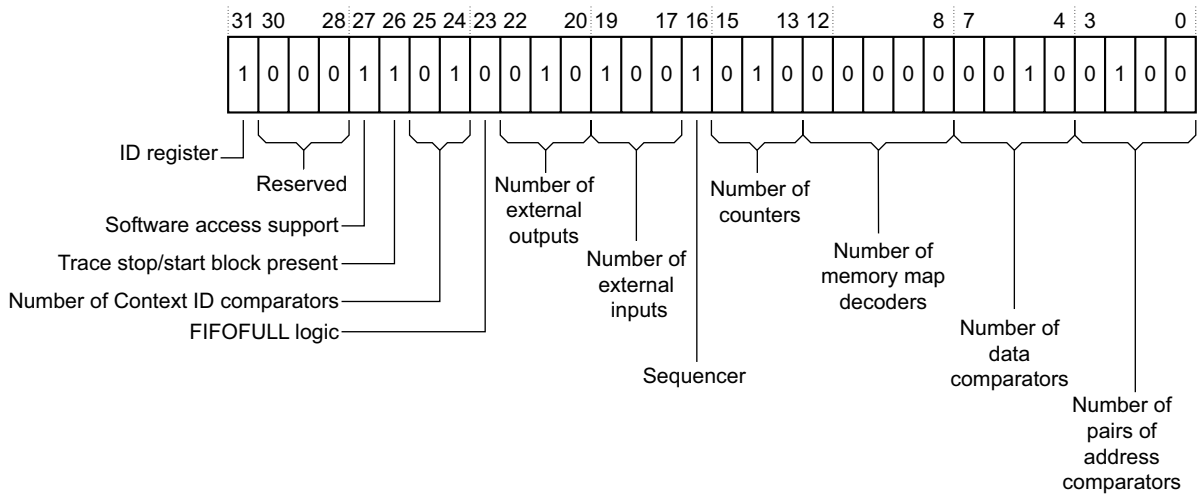
Table 14-3 shows how the bit values correspond with the ID Register functions.

**Table 14-3 ID Register bit functions**

Bits	Field	Function
[31:24]	Implementor	Indicates implementor, ARM: 0x41.
[23:20]	-	Reserved, RAZ.
[19]	Security Extensions support	Indicates Security Extensions support. The processor supports Security Extensions architecture. If this bit is not set to 1, then the ETM behaves as if the processor is in secure state at all times.
[18]	Thumb-2 support	All 32-bit Thumb instructions are traced as a single instruction, including BL and BLX immediate.
[17]	-	Reserved, RAZ.
[16]	Load pc first	All data transfers are traced in the same order that they appear in the <i>ARM Architecture Reference Manual</i> .
[15:12]	ARM core family	Indicates the Cortex-A8 processor.
[11:8]	Major ETM architecture version	Indicates the major ETM architecture version number, ETMv3.
[7:4]	Minor ETM architecture version	Indicates the minor ETM architecture version number, ETMv3.3.
[3:0]	Revision	Indicates the implementation revision.

#### 14.4.2 Configuration Code Register

The Configuration Code Register, at offset 0x004, is a 32-bit read-only register that provides information about the configuration of the ETM. Figure 14-3 on page 14-12 shows the bit arrangement for the Configuration Code Register.



**Figure 14-3 Configuration Code Register format**

Table 14-4 shows how the bit values correspond with the Configuration Code Register functions. The Configuration Code Register has the value 0x8D294024.

**Table 14-4 Configuration Code Register bit functions**

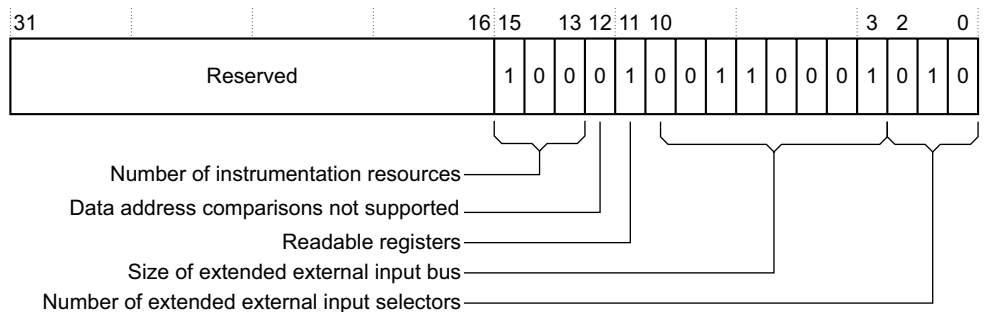
Bits	Field	Function
[31]	ID Register	Indicates that the ETM ID Register is present
[30:28]	-	Reserved, RAZ
[27]	Software access	Indicates that software access is supported
[26]	Trace stop/start block	Indicates that the trace start/stop block is present
[25:24]	Number of Context ID comparators	Specifies the number of Context ID comparators
[23]	FIFOFULL logic	Indicates that it is not possible to stall the processor to prevent FIFO overflow, uses data suppression instead
[22:20]	Number of external outputs	Specifies the number of external outputs
[19:17]	Number of external inputs	Specifies the number of external inputs
[16]	Sequencer	Indicates that the sequencer is present
[15:13]	Number of counters	Specifies the number of counters

**Table 14-4 Configuration Code Register bit functions (continued)**

Bits	Field	Function
[12:8]	Number of memory map decoders	Specifies the number of memory map decoders
[7:4]	Number of data comparators	Specifies the number of data comparators
[3:0]	Number of pairs of address comparators	Specifies the number of pairs of address comparators

### 14.4.3 Configuration Code Extension Register

The Configuration Code Extension Register, at offset 0x1E8, is a read-only register that provides additional information about the configuration of the ETM. Figure 14-4 shows the bit arrangement of the Configuration Code Extension Register.



**Figure 14-4 Configuration Code Extension Register format**

Table 14-5 shows how the bit values correspond with the Configuration Code Extension Register functions. The Configuration Code Register has the value 0x0000898A.

**Table 14-5 Configuration Code Extension Register bit functions**

Bits	Field	Function
[31:16]	-	Reserved, RAZ.
[15:13]	Number of instrumentation resources	Specifies the number of instrumentation resources.
[12]	Data address comparisons not supported	Indicates that data address comparisons are supported by ETM.

**Table 14-5 Configuration Code Extension Register bit functions (continued)**

Bits	Field	Function
[11]	Readable registers	Indicates that all registers, except some Integration Test Registers, are readable. See Table 14-2 on page 14-8 for details of the access permission to the Integration Test Registers. Registers with names that start with IT are the Integration Test Registers, for example ITATBCTR1.
[10:3]	Size of extended external input bus	Specifies the size of the extended external input bus.
[2:0]	Number of extended external input selectors	Specifies the number of extended external input selectors.

#### 14.4.4 Peripheral Identification Registers

The ETM Peripheral Identification Registers are a set of eight read-only registers, PeripheralID7 to PeripheralID0. These registers are defined in the *ETM Architecture Specification*. Only bits [7:0] of each register are used.

Table 14-6 shows the bit field definitions of the Peripheral Identification Registers. The *ETM Architecture Specification* describes many of these fields in more detail.

**Table 14-6 Peripheral Identification Registers bit functions**

Register name	Register offset	Bit range	Value	Function
PeripheralID7	0xFDC	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID6	0xFD8	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID5	0xFD4	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID4	0xFD0	[31:8]	-	Unused, RAZ
		[7:4]	0x0	Indicates that the ETM uses one 4KB block of memory
		[3:0]	0x4	JEP106 continuation code [3:0]
PeripheralID3	0xFEC	[31:8]	-	Unused, RAZ
		[7:4]	0x1	RevAnd (at top level)

Table 14-6 Peripheral Identification Registers bit functions (continued)

Register name	Register offset	Bit range	Value	Function
		[3:0]	0x0	Customer Modified 0x00 indicates from ARM
PeripheralID2	0xFE8	[31:8]	-	Unused, RAZ
		[7:4]	0x6	Revision number of Peripheral
		[3]	0x1	Indicates that a JEDEC assigned value is used
		[2:0]	0x3	JEP106 identity code [6:4]
PeripheralID1	0xFE4	[31:8]	-	Unused, RAZ
		[7:4]	0xB	JEP106 identity code [3:0]
		[3:0]	0x9	Part number 1 upper <i>Binary Coded Decimal</i> (BCD) value of Device number
PeripheralID0	0xFE0	[31:8]	-	Unused, RAZ
		[7:0]	0x21	Part number 0 middle and lower BCD value of Device number

———— **Note** ————

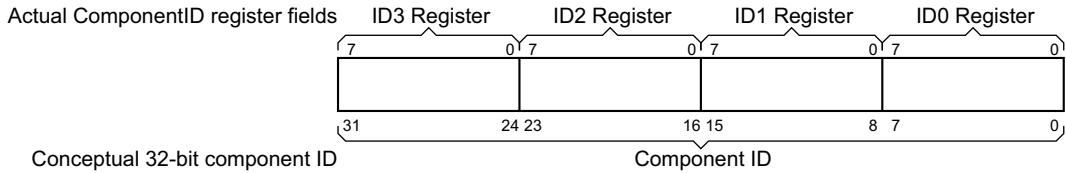
In Table 14-6 on page 14-14, the Peripheral Identification Registers are listed in order of register name, from most significant (ID7) to least significant (ID0). This does not match the order of the register offsets. Similarly, in Table 14-7 on page 14-16, the Component Identification Registers are listed in order of register name, from most significant (ID3) to least significant (ID0).

#### 14.4.5 Component Identification Registers

There are four read-only Component Identification Registers, ComponentID3 to ComponentID0. Although these are implemented as standard 32-bit registers:

- the most significant 24 bits of each register are not used and Read-As-Zero
- the least significant eight bits of each register together make up the component ID.

Figure 14-5 on page 14-16 shows this concept of a single 32-bit component ID, obtained from the four Component Identification Registers.



**Figure 14-5 Mapping between the Component ID Registers and the component ID value**

Table 14-7 shows the bit field definitions of the Component Identification Registers. This register structure is defined in the *ETM Architecture Specification*.

**Table 14-7 Component Identification Registers bit functions**

Register	Register offset	Bit range	Value	Function
ComponentID3	0xFFC	[31:8]	-	Unused, RAZ
		[7:0]	0xB1	Component identifier, bits [31:24]
ComponentID2	0xFF8	[31:8]	-	Unused, RAZ
		[7:0]	0x05	Component identifier, bits [23:16]
ComponentID1	0xFF4	[31:8]	-	Unused, RAZ
		[7:4]	0x9	Component class; component identifier, bits [15:12]
		[3:0]	0x0	Component identifier, bits [11:8]
ComponentID0	0xFF0	[31:8]	-	Unused, RAZ
		[7:0]	0x0D	Component identifier, bits [7:0]

### 14.4.6 Integration Test Registers

The following sections describe the Integration Test Registers. To access these registers you must first set bit [0] of the Integration Mode Control Register to 1.

- You can use the write-only Integration Test Registers to set the outputs of some of the ETM signals. Table 14-8 on page 14-17 lists the signals that can be controlled in this way.
- You can use the read-only Integration Test Registers to read the state of some of the ETM input signals. Table 14-9 on page 14-17 lists the signals that can be read in this way.

See the *ETM Architecture Specification* for more information.

**Table 14-8 Output signals that can be controlled by the Integration Test Registers**

Signal	Register	Bit	Description
<b>AFREADYM</b>	ITATBCTR0	[1]	See <i>ITATBCTR0 Register</i> on page 14-21
<b>ATBYTESM[1:0]</b>	ITATBCTR0	[9:8]	See <i>ITATBCTR0 Register</i> on page 14-21
<b>ATDATAM[31, 23, 15, 7, 0]</b>	ITATBDATA0	[4:0]	See <i>ITATBDATA0 Register</i> on page 14-19
<b>ATIDM[6:0]</b>	ITATBCTR1	[6:0]	See <i>ITATBCTR1 Register</i> on page 14-21
<b>ATVALIDM</b>	ITATBCTR0	[0]	See <i>ITATBCTR0 Register</i> on page 14-21
<b>EXTOUT[1:0]</b>	ITMISCOUT	[9:8]	See <i>ITMISCOUT Register</i> on page 14-18
<b>TRIGGER</b>	ITTRIGGER	[0]	See <i>ITTRIGGER Register</i> on page 14-19

**Table 14-9 Input signals that can be read by the Integration Test Registers**

Signal	Register	Bit	Description
<b>AFVALIDM</b>	ITATBCTR2	[1]	<i>ITATBCTR2 Register</i> on page 14-20
<b>ATREADYM</b>	ITATBCTR2	[0]	<i>ITATBCTR2 Register</i> on page 14-20
<b>DBGACK</b>	ITMISCIN	[4]	<i>ITMISCIN Register</i> on page 14-18
<b>EXTIN[3:0]</b>	ITMISCIN	[3:0]	<i>ITMISCIN Register</i> on page 14-18

## Using the Integration Test Registers

The *CoreSight Design Kit Technical Reference Manual* gives a full description of the use of the Integration Test Registers to check integration. In brief, when bit [0] of the Integration Mode Control Register is set to 1:

- Values written to the write-only Integration Test Registers map onto the specified outputs of ETM. For example, writing 0x3 to **ITMISCOUT[1:0]** causes **EXTOUT[1:0]** to take the value 0x3.
- Values read from the read-only integration test registers correspond to the values of the specified inputs of ETM. For example, if you read **ITMISCIN[1:0]** you obtain the value of **EXTIN**.

### ITMISCOUT Register

The ITMISCOUT Register, miscellaneous outputs, at offset 0xEDC, is write-only. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-6 shows the bit arrangement of the ITMISCOUT Register.

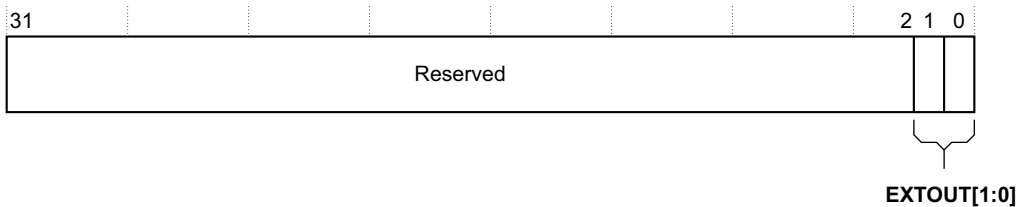


Figure 14-6 ITMISCOUT Register format

Table 14-10 shows how the bit values correspond with the ITMISCOUT Register functions.

Table 14-10 ITMISCOUT Register bit functions

Bits	Field	Function
[31:2]	-	Reserved, SBZ
[1:0]	EXTOUT	Drives the <b>EXTOUT[1:0]</b> outputs

### ITMISCIN Register

The ITMISCIN Register, miscellaneous inputs, at offset 0xEE0, is read-only. This register enables the values of signal inputs to be read when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-7 shows the bit arrangement of the ITMISCIN Register.

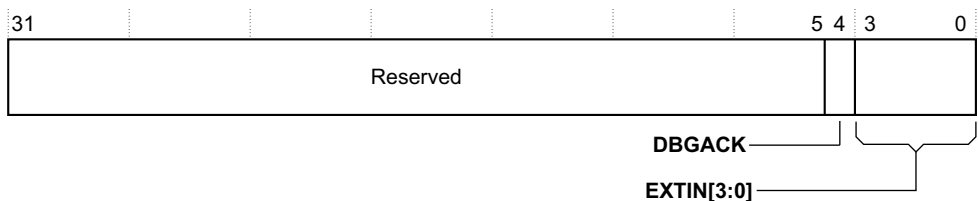


Figure 14-7 ITMISCIN Register format



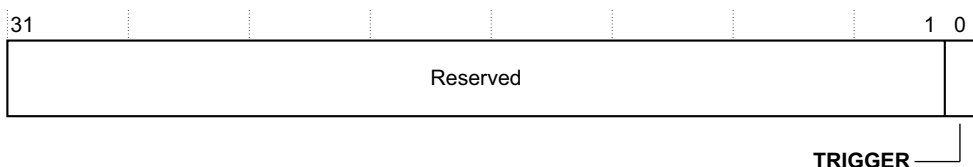
Table 14-11 shows how the bit values correspond with the ITMISCIN Register functions. The value of these fields depend on the signals on the input pins when the register is read.

**Table 14-11 ITMISCIN Register bit functions**

Bits	Field	Function
[31:5]	-	Reserved, RAZ
[4]	DBGACK	Returns the value of the <b>DBGACK</b> input
[3:0]	EXTIN	Returns the value of the <b>EXTIN[3:0]</b> inputs

### ITTRIGGER Register

The ITTRIGGER Register, trigger request, at offset 0xEE8, is write-only. This register controls the signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-8 shows the bit arrangement of the ITTRIGGER Register.



**Figure 14-8 ITTRIGGER Register format**

Table 14-12 shows how the bit values correspond with the ITTRIGGER Register functions.

**Table 14-12 ITTRIGGER Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved, SBZ
[0]	TRIGGER	Drives the <b>TRIGGER</b> output

### ITATBDATA0 Register

The ITATBDATA0 Register, ATB data 0, at offset 0xEEC, is write-only. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-9 on page 14-20 shows the bit assignment of the ITATBDATA0 Register.

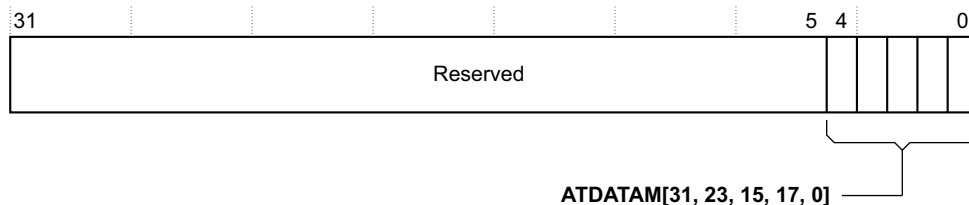


Figure 14-9 ITATBDATA0 Register format

Table 14-13 shows how the bit values correspond with the ITATBDATA0 Register functions.

Table 14-13 ITATBDATA0 Register bit functions

Bits	Field	Function
[31:5]	-	Reserved, SBZ
[4:0]	ATDATAM	Drives the <b>ATDATAM[31, 23, 15, 17, 0]</b> outputs

### ITATBCTR2 Register

The ITATBCTR2 Register, ATB control 2, at offset 0xEF0, is read-only. This register enables the values of signal inputs to be read when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-10 shows the bit assignment of the ITATBCTR2 Register.

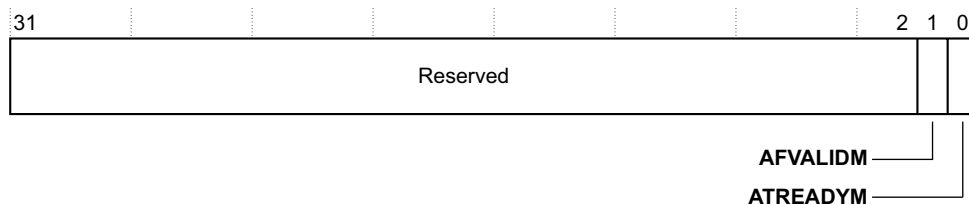


Figure 14-10 ITATBCTR2 Register format

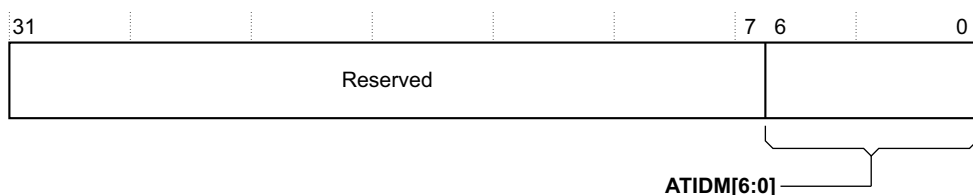
Table 14-14 shows how the bit values correspond with the ITATBCTR2 Register functions. The value of these fields depend on the signals on the input pins when the register is read.

**Table 14-14 ITATBCTR2 Register bit functions**

Bits	Field	Function
[31:2]	-	Reserved, RAZ
[1]	AFVALIDM	Returns the value of the <b>AFVALIDM</b> input
[0]	ATREADYM	Returns the value of the <b>ATREADYM</b> input

### ITATBCTR1 Register

The ITATBCTR1 Register, ATB control 1, at offset 0xEF4, is write-only. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-11 shows the bit assignment of the ITATBCTR1 Register.



**Figure 14-11 ITATBCTR1 Register format**

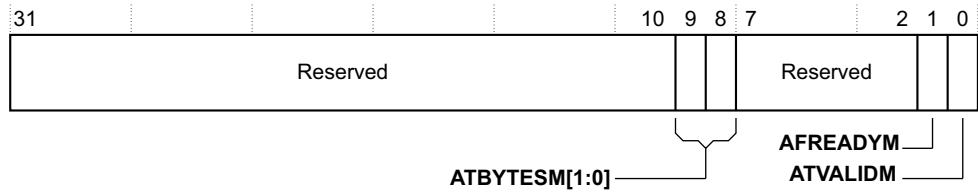
Table 14-15 shows how the bit values correspond with the ITATBCTR1 Register functions.

**Table 14-15 ITATBCTR1 Register bit functions**

Bits	Field	Function
[31:7]	-	Reserved, SBZ
[6:0]	ATIDM	Drives the <b>ATIDM[6:0]</b> outputs

### ITATBCTR0 Register

The ITATBCTR0 Register, ATB control 0, at offset 0xEF8, is write-only. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 14-12 on page 14-22 shows the bit assignment of the ITATBCTR0 Register.



**Figure 14-12 ITATBCTR0 Register format**

Table 14-16 shows how the bit values correspond with the ITATBCTR0 Register functions.

**Table 14-16 ITATBCTR0 Register bit functions**

Bits	Field	Function
[31:10]	-	Reserved, SBZ
[9:8]	ATBYTESM	Drives the ATBYTESM[1:0] outputs
[7:2]	-	Reserved, SBZ
[1]	AFREADYM	Drives the AFREADYM output
[0]	ATVALIDM	Drives the ATVALIDM output

## 14.5 Precision of TraceEnable and ViewData

The *ETM Architecture Specification* states that **TraceEnable** or **ViewData** is Imprecise under certain conditions. This section describes when **TraceEnable** and **ViewData** are Precise.

### 14.5.1 TraceEnable

**TraceEnable** is Precise if all of the following are true:

- The **TraceEnable** enabling event is Precise.
- The single address comparators selected by the start/stop resource are Precise, or **TraceEnable** is not configured to use the start/stop resource.
- **TraceEnable** is configured to include regions, the selected single address comparators and address range comparators are Precise.
- **TraceEnable** is configured to exclude regions, the selected single address comparators and address range comparators are Precise and are configured for instruction addresses. It is not possible to exclude instruction trace based on the addresses of data transfers.

The processor can execute two instructions in a cycle. The **TraceEnable** enabling event is calculated once per cycle. The other parts of **TraceEnable** are calculated once per instruction.

If the processor executes two instructions in a cycle, the ETM can trace neither of them or both of them, but cannot trace only one of them. If **TraceEnable** indicates that one instruction can be traced, then trace is generated as if **TraceEnable** had indicated that both instructions on that cycle can be traced. As an example, consider the following case:

- Two instructions are executed in the same cycle.
- The first instruction causes a single address comparators to match with what is selected as a start address.
- The second instruction causes a single address comparators to match with what is selected as a stop address.
- The **TraceEnable** enabling event is true.
- **TraceEnable** is configured to use the start/stop resource.
- **TraceEnable** is configured to exclude regions.
- No address comparators are selected for exclude regions.

In this case, **TraceEnable** behaves as follows:

1. The first instruction is traced because the start/stop resource was active.
2. The second instruction is traced because the first instruction was traced.
3. Instructions are not traced on subsequent cycles because the start/stop resource is off.

### 14.5.2 ViewData

**ViewData** is Precise if all of the following are true:

- the **ViewData** enabling event is Precise
- the single address comparators and address range comparators selected to include regions are Precise, or **ViewData** is configured for exclude regions only
- the single address comparators and address range comparators selected to exclude regions are Precise.

The processor can perform two 32-bit data transfers in a cycle. The ETM treats a 64-bit data transfer as two 32-bit data transfers. The **ViewData** enabling event is calculated once per cycle. The other parts of **ViewData** are calculated once per data transfer.

If the processor performs two 32-bit data transfers in a cycle, the ETM can trace neither, one, or both of them. **ViewData** is recalculated for each transfer. However, because the enabling event is only calculated once per cycle, address comparators selected using the enabling event cause both data transfers to be traced as if a match occurs on each transfer.

### 14.5.3 Enabling events

The **TraceEnable** and **ViewData** enabling events are Precise if only the following is selected:

- Precise single address comparators
- Precise address range comparators
- instrumentation resources
- context ID comparator
- nonsecure state resource
- prohibited resource
- hard-wired resource, always true.

The following events are delayed by two cycles compared to the timing of their input events, and are Imprecise:

- counters at zero
- sequence state 1, 2, or 3
- trace start/stop resource.

The following events are Imprecise and have no fixed timing relationship with other events:

- external input
- extended external input selectors.

#### 14.5.4 Address comparators

Single address comparators and address range comparators are always Precise if the exact match bit for that comparator is cleared to 0. This includes cases where the address comparator is conditional on the context ID comparator matching.

## 14.6 Exact match bit

You can set the exact match bit in the address comparators to 1, to cause the ETM to wait before permitting the address comparator to match. Setting the exact match bit prevents it from an unintentional match. The exact match bit behaves as follows in the ETM:

- *Address comparators configured for instruction addresses*
- *Address comparators configured for data addresses*
- *Address range comparators.*

### 14.6.1 Address comparators configured for instruction addresses

If the exact match bit is set to 1, each instruction matches if both of the following are true:

- the instruction matches the address comparison conditions
- the instruction is not followed by a cancelling exception.

To determine whether the instruction is followed by a cancellation exception, the match does not take place until the next instruction. Therefore, matches do not occur in time to control tracing, but instructions that are cancelled by exceptions do not cause the comparator to match. This is useful when, for example, you want to count the number of times an instruction has been executed.

If the exact match bit is cleared to 0, each instruction matches if it matches the address comparison conditions. The match occurs at the time the instruction is traced and therefore, cannot consider if the instruction is subsequently cancelled. This is useful when you want to use the comparator to control tracing.

———— **Note** —————

Instructions that are cancelled by exceptions do not cause the comparator to match. This is useful when you want to count the number of times an instruction has been executed.

---

### 14.6.2 Address comparators configured for data addresses

The exact match bit does not affect whether a match occurs on a data address.

### 14.6.3 Address range comparators

If an address range comparator configured for instruction address matches on an instruction address, it continues to match until the next instruction addresses. If an address range comparator configured for data addresses matches on a data address, it continues to match until the next data addresses. This enables the use of address range



comparator in an imprecise manner such as to qualify performance monitoring events available as extended external input selectors. However, this is occasionally not required, for example when counting the number of data transfers performed in a region of memory.

If the exact match bit is set to 1, the address range comparator does not hold its value. An address range comparator configured for instruction addresses with its exact match bit cleared to 0 does not match on cycles in which no instructions are executed. In addition, an address range comparator configured for data addresses with its exact match bit cleared to 0 does not match on cycles in which no data transfers are performed.

## **14.7 Context ID tracing**

The ETM detects the MCR instruction that changes the context ID, and traces the appropriate number of bytes as a context ID packet instead of a normal data packet. As a result, if context ID tracing is enabled, an MCR instruction that changes the context ID does not have its data traced separately.

Because the ETM has a secure and a nonsecure context ID, the ETM outputs a context ID when switching between secure and nonsecure states.

## 14.8 Instrumentation instructions

The ETM implements four instrumentation resources. You can use these resources to control the behavior of the ETM by inserting instrumentation instructions into the code for the processor to execute. For more information, see the *ETM Architecture Specification*.

The ETM can predict whether an instrumentation instruction is canceled at the time it is traced. If an instrumentation instruction is canceled, it has no effect on the instrumentation resources.

## 14.9 Idle state control

The ETM implements an idle state that must be entered before you can power down the ETM. Before entry to the idle state, the following sequence occurs:

1. Trace is turned off.
2. The ETM waits for all trace that has already been generated to reach the FIFO.
3. The main FIFO is emptied.
4. The resynchronizing FIFO is emptied.
5. The ETM waits for any remaining trace on the ATB interface to be accepted.
6. The resynchronizing FIFO sets the read and write pointers that it uses to zero.

When in idle state, you can safely remove the power from the **ck\_gclke** or **ATCLK** domain. It is recommended that you use the OS Save and Restore Registers to save the registers in the **ck\_gclke** domain before removing the power and to restore the registers after restoring the power. See the *ETM Architecture Specification* for more information.

Following a reset of the **ck\_gclke** domain, the ETM is in idle state. The ETM is also in idle state when any of the following occur:

- the power down bit is set to 1
- the programming bit is set to 1
- the ETMEN bit is cleared to 0
- the OS Save and Restore Register lock is set
- a WFI idle request is encountered
- both the **NIDEN** and **DBGEN** inputs are LOW.

The ETM Status Register reports the programming bit as set to 1 if both:

- the programming bit, power down bit, or OS Save and Restore Register lock is set to 1
- the ETM is in idle state.

The standard method to turn off the ETM is to set the programming bit to 1 and wait for the ETM Status Register to report the programming bit as set to 1. This method ensures that the idle entry sequence is complete before you can perform more operations.

If the idle request is cancelled before the idle entry sequence is complete, the ETM behaves as if the idle request is maintained until the idle entry sequence is complete. For example, if the programming bit is set to 1 and 0 in quick succession without checking the ETM Status Register, the programming bit is not cleared to 0 internally until the idle entry sequence has completed.

When a WFI occurs, the processor waits for the idle entry sequence to complete before stopping the clock to the ETM.

## 14.10 Interaction with the Performance Monitoring Unit

The processor includes a *Performance Monitoring Unit* (PMU) that enables events, such as cache misses and instructions executed, to be counted over a period of time. This section describes how the PMU and ETM are used together.

### 14.10.1 Use of PMU events by the ETM

The PMU events are all available for use by the ETM using the extended external input facility. Each event is mapped to one or two extended external inputs. For more information on PMU events, see *c9, Event Selection Register* on page 3-110.

A PMU event uses two extended external inputs where two such events can occur in a cycle. Both extended external inputs are active in cycle when two events occur. The *ETM Architecture Specification* describes how to use extended external input selectors to make these events available to the rest of the ETM triggering and filtering logic.

Table 14-17 shows the mapping of the PMU event numbers to the ETM extended external input event numbers.

**Table 14-17 PMU event number mappings**

PMU event number	First ETM event number	Second ETM event number
0x0	-	-
0x1	0x1	-
0x2	0x2	-
0x3	0x3	-
0x4	0x5	-
0x5	0x6	-
0x6	0x7	-
0x7	0x8	-
0x8	0x9	0xa
0x9	0xb	-
0xa	0xc	-
0xb	0xd	-
0xc	0xe	-

Table 14-17 PMU event number mappings (continued)

PMU event number	First ETM event number	Second ETM event number
0xd	0xf	-
0xe	0x10	-
0xf	0x11	-
0x10	0x12	-
0x12	0x13	-
0x40	0x14	-
0x41	0x15	-
0x42	0x16	-
0x43	0x17	0x18
0x44	0x19	0x32
0x45	0x1a	-
0x46	0x1b	-
0x47	0x1c	-
0x48	0x1d	-
0x49	0x1e	-
0x4a	0x1f	-
0x4b	0x20	-
0x4c	0x21	-
0x4d	0x22	-
0x4e	0x23	-
0x4f	0x24	-
0x50	0x25	-
0x51	0x26	-
0x52	0x27	-
0x53	0x28	-

**Table 14-17 PMU event number mappings (continued)**

PMU event number	First ETM event number	Second ETM event number
0x54	0x29	-
0x55	0x2a	0x2b
0x56	0x2c	-
0x57	0x2d	0x2e
0x58	0x2f	-
0x59	0x30	-
0x5a	0x31	-

Table 14-18 shows the behavior of the ETM when two PMU events occur in a cycle.

**Table 14-18 PMU event cycle mappings**

PMU events in cycle	First ETM event active	Second ETM event active
0	No	No
1	Yes	No
2	Yes	Yes

### 14.10.2 Use of ETM events by the PMU

The PMU can count the two ETM external outputs as additional events by using the CTI. You must configure the CTI to connect the ETM external outputs to the PMU.

Because the CTI is implemented in the ATCLK clock domain, the ETM events must be resynchronized to ATCLK and back to the core clock before the PMU can use it. If the ETM events are too close together, the resynchronization causes some events to be lost.

The CTI outputs are normally held several cycles while synchronization takes place. CTI supports edge-detection logic that enables the PMU to count one event per ETM event. ARM recommends that you enable edge-detection for the PMU CTI outputs.



You can use the ETM to qualify PMU events and then count them using the ETM counters or pass them back to the PMU to be counted. You can count the number of cache misses caused by a particular region of instruction addresses as follows:

- Configure the ETM extended external input selectors to the PMU cache miss events you want to count.
- Configure an address range comparator to the required instruction address region, with the exact match bit cleared to 0.
- Configure the ETM external outputs as follows:
  - Event A is the extended external input selector.
  - Event B is the required address range comparator.
  - Function is A and B.
- Select the PMU external inputs to be counted in the PMU.



# Chapter 15

## Cross Trigger Interface

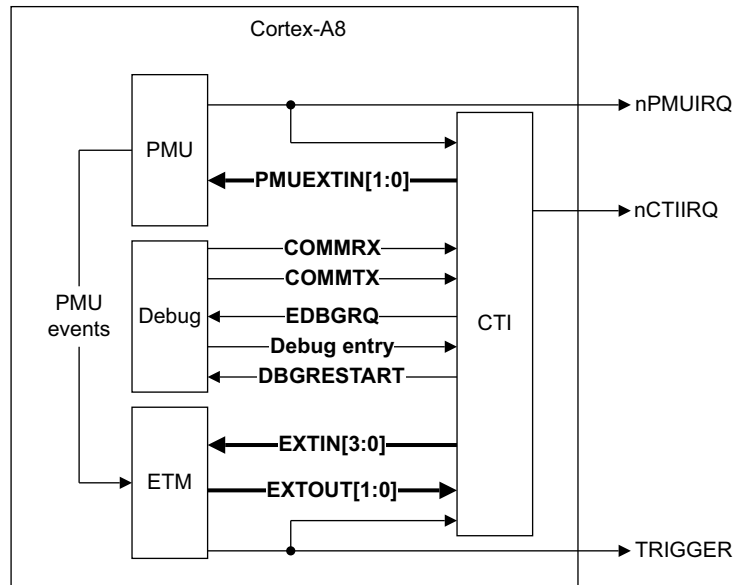
This chapter describes the *Cross Trigger Interface (CTI)*. It contains the following sections:

- *About the CTI* on page 15-2
- *Trigger inputs and outputs* on page 15-6
- *Connecting asynchronous channel interfaces* on page 15-8
- *About the CTI programmers model* on page 15-9
- *CTI register summary* on page 15-10
- *CTI register descriptions* on page 15-13
- *CTI Integration Test Registers* on page 15-24
- *CTI CoreSight defined registers* on page 15-30.

## 15.1 About the CTI

The CTI enables the debug logic, ETM, and PMU, to interact with each other and with other CoreSight components. This is called cross triggering. For example, you can configure the CTI to generate an interrupt when the ETM trigger event occurs.

The CTI is connected to a number of *trigger inputs* and *trigger outputs*. You can connect each trigger input to one or more trigger outputs. Figure 15-1 shows the debug system components and the available trigger inputs and trigger outputs.



**Figure 15-1 Debug system components**

The CTI also implements a synchronous channel interface as defined in the *CoreSight Architecture Specification* for communication with other CoreSight components.

### 15.1.1 How the CTI works

The CTI connects trigger inputs to trigger outputs using four channels. The following can cause a channel event:

- A trigger input event, if you have configured the channel for the trigger input using the CTIINEN registers. See *Trigger inputs and outputs* on page 15-6 for information on trigger inputs and outputs that are available to the CTI.
- An application trigger, using the CTIAPPSET, CTIAPPCLEAR, and CTIAPPULSE registers.

- An input event on the channel interface.

A channel event can cause the following to occur:

- A trigger output event, if you have configured the channel for the trigger output using the CTIOUTEN registers. See *Trigger inputs and outputs* on page 15-6 for information on trigger inputs and outputs that are available to the CTI.
- An output event on the channel interface, unless the channel interface output for that channel has been disabled using the CTICHGATE Register.

Figure 15-2 on page 15-4 shows the connections to the channels.

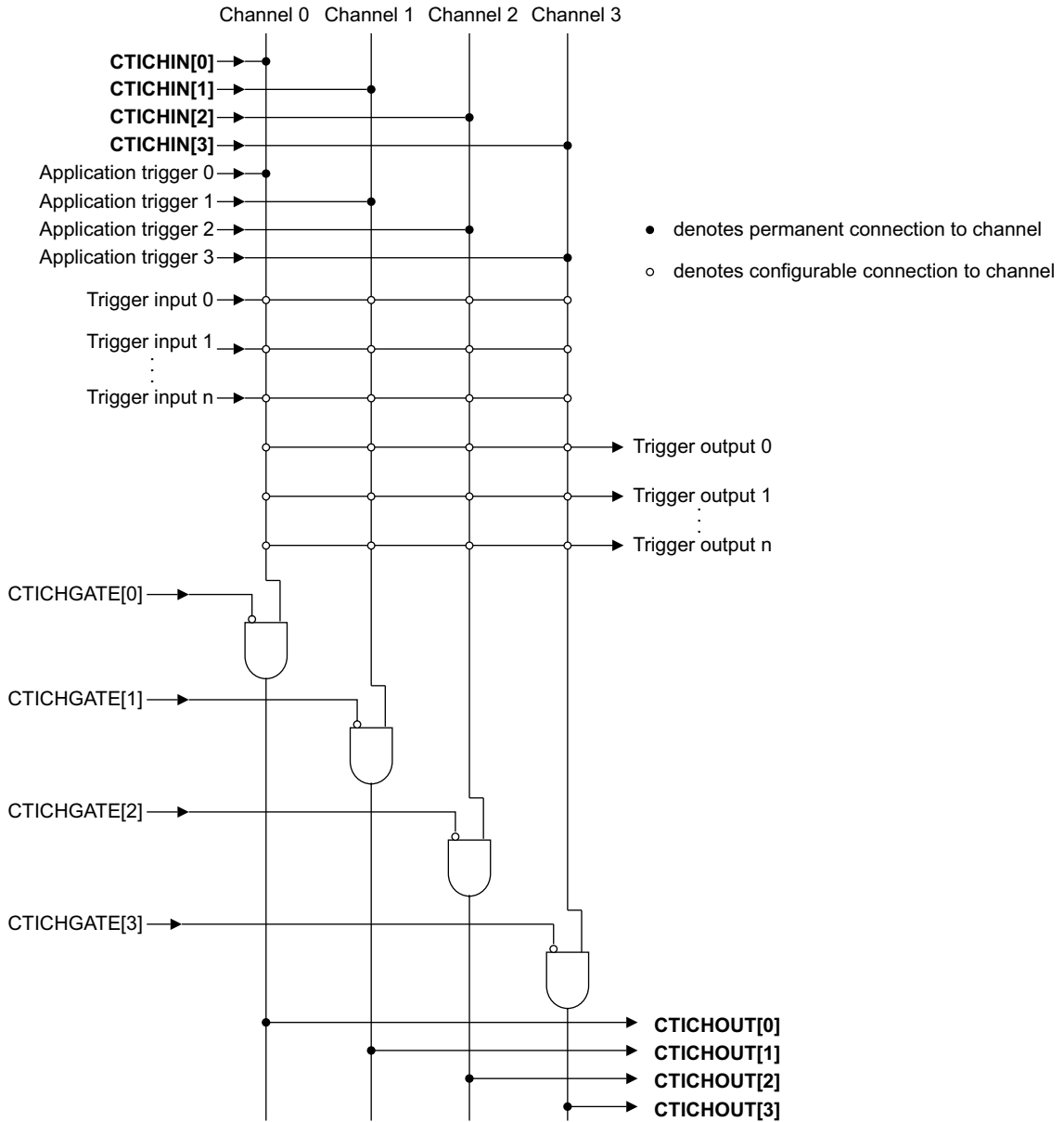


Figure 15-2 Cross Trigger Interface channels

### 15.1.2 The channel interface

The CTI can be:

- used on its own
- connected to another CTI using the channel interface
- connected to a CTM using the channel interface, enabling multiple CTIs to be linked together.

The CoreSight Design Kit for the processor includes CTI and CTM components. You must use a separate CTI to connect the channel interface to system-level trigger signals.

### 15.1.3 Trigger signal synchronization

The CTI operates in the ATCLK domain, and synchronizes the trigger inputs and outputs to ATCLK where required. The **EXTIN[3:0]** and **PMUEXTIN[1:0]** trigger outputs support edge detection, controlled by the CTI ASICCTL Register. See *ASIC Control Register, ASICCTL* on page 15-21.

## 15.2 Trigger inputs and outputs

This section describes the trigger inputs and outputs that are available to the CTI.

Table 15-1 shows the trigger inputs available to the CTI.

**Table 15-1 Trigger inputs**

Trigger input	Name	Clock domain	Description
0	Debug entry <sup>a</sup>	CLK	Pulsed on entry to debug state
1	!nPMUIRQ	CLK	PMU generated interrupt
2	EXTOUT[0]	CLK	ETM external output
3	EXTOUT[1]	CLK	ETM external output
4	COMMRX	CLK	Debug communication receive channel is full
5	COMMTX	CLK	Debug communication transmit channel is empty
6	TRIGGER	ATCLK	ETM trigger

- a. For revision r3 of the Cortex-A8 processor, this trigger is a pulse asserted on debug state entry. For revisions r0 through r2, this trigger is a level-sensitive signal asserted while the processor is in debug state. This level-sensitive signal is **DBGTRIGGER**.

Table 15-2 shows the trigger outputs available to the CTI.

**Table 15-2 Trigger outputs**

Trigger Output	Name	Clock domain	Edge detection enable	Description
0	EDBGRQ	CLK	-	Causes the processor to enter debug state.
1	EXTIN[0]	CLK	ASICCTL[0]	ETM external input.
2	EXTIN[1]	CLK	ASICCTL[1]	ETM external input.
3	EXTIN[2]	CLK	ASICCTL[2]	ETM external input.
4	EXTIN[3]	CLK	ASICCTL[3]	ETM external input.
5	PMUEXTIN[0]	CLK	ASICCTL[4]	PMU CTI event. This input can be selected by the Event Selection Register. See <i>c9, Event Selection Register</i> on page 3-110 for more information on PMU events.



Table 15-2 Trigger outputs (continued)

Trigger Output	Name	Clock domain	Edge detection enable	Description
6	PMUEXTIN[1]	CLK	ASICCTL[5]	PMU CTI event. This input can be selected by the Event Selection Register. See <i>c9, Event Selection Register</i> on page 3-110 for more information on PMU events.
7	DBGRESTART	CLK	-	Causes the processor to exit debug state.
8	nCTIIRQ	Asynchronous	-	Generates an interrupt if nCTIIRQ is connected appropriately to the interrupt controller.

---

**Note**

---

- In revision r3 of the Cortex-A8 processor, trigger outputs 0 and 8 must be cleared by software. See *CTI Interrupt Acknowledge Register, CTIINTACK* on page 15-13.
  - In revisions r0 through r2, only trigger output 8 must be cleared by software. Trigger output 0 is automatically cleared by the debug state entry event.
-

## 15.3 Connecting asynchronous channel interfaces

The CTI implements a synchronous channel interface. It is synchronous to ATCLK.

It is possible to convert between synchronous and asynchronous versions of the channel interface using the circuit shown in Figure 15-3, which also includes a **BYPASS** signal to enable the converter to be bypassed if not required.

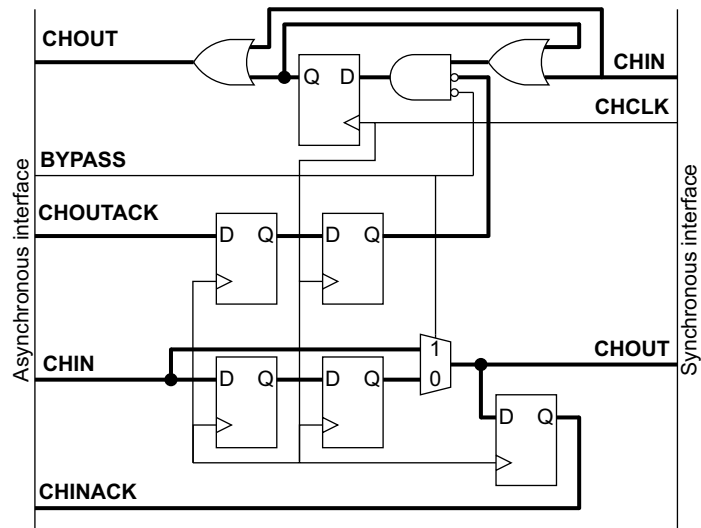


Figure 15-3 Asynchronous to synchronous converter

If you implement a synchronous to asynchronous converter, you increase the likelihood of merger of events. This is likely to happen if the events to be counted occur close together. For more information, see the *CoreSight Architecture Specification*.

## 15.4 About the CTI programmers model

The base addresses of the CTI are not fixed, and can be different for any particular system implementation. However, the offset of any particular register from the base address is fixed. Each CTI has a 4KB programmer's model and must be programmed separately. All unused memory space is reserved.

The following applies to all registers:

- reserved or unused bits of registers must be written as 0, and ignored on a read unless otherwise stated in the text
- all register bits are reset to 0 unless otherwise stated in the text
- all registers must be accessed as words and so are compatible with little-endian and big-endian systems.

## 15.5 CTI register summary

Table 15-3 shows the CTI programmable registers.

**Table 15-3 CTI register summary**

Address offset	Register name	Type	Width	Reset value	Description
0x000	CTICONTROL	R/W	1	0x0	CTI Control Register, CTICONTROL on page 15-13
0x010	CTIINTACK	W	9	-	CTI Interrupt Acknowledge Register, CTIINTACK on page 15-13
0x014	CTIAPPSET	R/W	4	0x0	CTI Application Trigger Set Register, CTIAPPSET on page 15-14
0x018	CTIAPPCLEAR	R/W	4	0x0	CTI Application Trigger Clear Register, CTIAPPCLEAR on page 15-15
0x01C	CTIAPPULSE	W	4	0x0	CTI Application Pulse Register, CTIAPPULSE on page 15-16
0x020-0x040	CTIINEN	R/W	4	0x00	CTI Trigger to Channel Enable Registers, CTIINEN0-8 on page 15-17
0x0A0-0x0C0	CTIOUTEN	R/W	4	0x00	CTI Channel to Trigger Enable Registers, CTIOUTEN0-8 on page 15-17
0x130	CTITRIGINSTATUS	R	9	-	CTI Trigger In Status Register, CTITRIGINSTATUS on page 15-18
0x134	CTITRIGOUTSTATUS	R	9	0x00	CTI Trigger Out Status Register, CTITRIGOUTSTATUS on page 15-19
0x138	CTICHINSTATUS	R	4	-	CTI Channel In Status Register, CTICHINSTATUS on page 15-20
0x140	CTICHGATE	R/W	4	0xF	CTI Channel Gate Register, CTICHGATE on page 15-20
0x144	ASICCTL	R/W	6	0x00	ASIC Control Register, ASICCTL on page 15-21
0x13C	CTICHOUTSTATUS	R	4	0x0	CTI Channel Out Status Register, CTICHOUTSTATUS on page 15-22
0xEE0	ITTRIGINACK	W	9	0x00	ITTRIGINACK, 0xEE0 on page 15-24
0xEE4	ITCHOUT	W	4	0x0	ITCHOUT, 0xEE4 on page 15-25

Table 15-3 CTI register summary (continued)

Address offset	Register name	Type	Width	Reset value	Description
0xEE8	ITTRIGOUT	W	9	0x00	<i>ITTRIGOUT</i> , 0xEE8 on page 15-26
0xEF0	ITTRIGOUTACK	R	9	0x00	<i>ITTRIGOUTACK</i> , 0xEF0 on page 15-27
0xEF4	ITCHIN	R	4	0x0	<i>ITCHIN</i> , 0xEF4 on page 15-27
0xEF8	ITTRIGIN	R	9	0x00	<i>ITTRIGIN</i> , 0xEF8 on page 15-28
0xEFC-0xF7C	-	-	-	-	Reserved
0xF00	ITCTRL	R/W	1	0x0	See the <i>CoreSight Design Kit Technical Reference Manual</i>
0xFA0	Claim Tag Set	R/W	4	0xF	
0xFA4	Claim Tag Clear	R/W	4	0x0	
0xFB0	Lock Access	W	32	-	
0xFB4	Lock Status	R	2	0x3	
0xFB8	Authentication Status	R	4	0xA	<i>CTI CoreSight defined registers</i> on page 15-30
0xFC0-0xFC4	-	-	-	-	Reserved
0xFC8	Device ID	R	20	0x40900	<i>CTI CoreSight defined registers</i> on page 15-30
0xFCC	Device Type Identifier	R	8	0x14	
0xFD0	PeripheralID4	R	8	0x04	<i>Peripheral Identification Registers</i> on page 15-31
0xFD4	PeripheralID5	R	8	0x00	
0xFD8	PeripheralID6	R	8	0x00	
0xFDC	PeripheralID7	R	8	0x00	
0xFE0	PeripheralID0	R	8	0x22	
0xFE4	PeripheralID1	R	8	0xB9	
0xFE8	PeripheralID2	R	8	0x6B	
0xFEC	PeripheralID3	R	8	0x10	

Table 15-3 CTI register summary (continued)

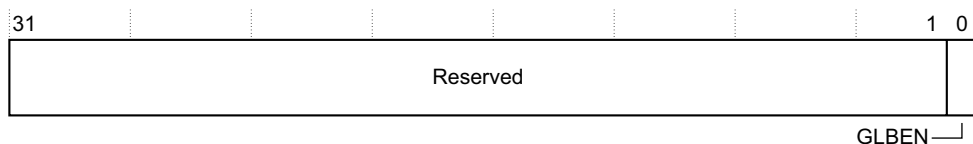
Address offset	Register name	Type	Width	Reset value	Description
0xFF0	ComponentID0	R	8	0x0D	<i>Component Identification Registers on page 15-32</i>
0xFF4	ComponentID1	R	8	0x90	
0xFF8	ComponentID2	R	8	0x05	
0xFFC	ComponentID3	R	8	0xB1	

## 15.6 CTI register descriptions

This section describes the CTI registers.

### 15.6.1 CTI Control Register, CTICONTROL

CTICONTROL is a read/write register that enables the CTI. Figure 15-4 shows the bit arrangement of the CTICONTROL Register.



**Figure 15-4 CTI Control Register format**

Table 15-4 shows how the bit values correspond with the CTICONTROL Register functions.

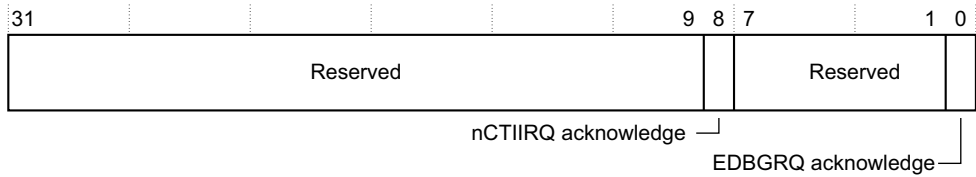
**Table 15-4 CTI Control Register bit functions**

Bits	Field	Function
[31:1]	-	Reserved. RAZ, SBZ.
[0]	GLBEN	Enables or disables the CTI: 0 = disable CTI (reset) 1 = enable CTI. When disabled, all cross triggering mapping logic functionality is disabled for this processor.

### 15.6.2 CTI Interrupt Acknowledge Register, CTIINTACK

CTIINTACK is a write-only register used to acknowledge the **nCTIIRQ** and **EDBGRQ** trigger outputs. When the **nCTIIRQ** trigger output is asserted, it continues to be asserted until you write to bit [8] of this register. When the **EDBGRQ** trigger output is asserted, it continues to be asserted until you write to bit [0] of this register.

Figure 15-5 on page 15-14 shows the bit arrangement of the CTIINTACK Register.



**Figure 15-5 CTI Interrupt Acknowledge Register format**

Table 15-5 shows how the bit values correspond with the CTIINTACK Register functions.

**Table 15-5 CTI Interrupt Acknowledge Register bit functions**

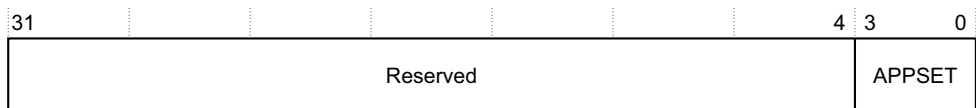
Bits	Field	Function
[31:9]	-	SBZ.
[8]	nCTIIRQ acknowledge	Acknowledges the <b>nCTIIRQ</b> output: 0 = no acknowledgement 1 = <b>nCTIIRQ</b> is acknowledged and is deasserted.
[7:1]	-	SBZ.
[0] <sup>a</sup>	EDBGRQ acknowledge	Acknowledges the <b>EDBGRQ</b> trigger output: 0 = no acknowledgement 1 = <b>EDBGRQ</b> trigger output is acknowledged and is deasserted.

a. Bit [0] EDBGRQ acknowledge of this register is only present in revision r3 of the Cortex-A8 processor. In revisions r0 through r2, this bit is SBZ.

### 15.6.3 CTI Application Trigger Set Register, CTIAPPSET

CTIAPPSET is a read/write register. A write to this register generates a channel event, corresponding to the bit written to.

Figure 15-6 shows the bit arrangement of the CTIAPPSET Register.



**Figure 15-6 CTI Application Trigger Set Register format**



Table 15-6 shows how the bit value corresponds with the CTIAPPSET Register functions.

**Table 15-6 CTI Application Trigger Set Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZ.
[3:0]	APPSET	Setting a bit HIGH generates an event for the selected channel. For read: 0 = application trigger inactive (reset) 1 = application trigger active. For write: 0 = no effect 1 = generate channel event. There is one bit of the register for each channel.

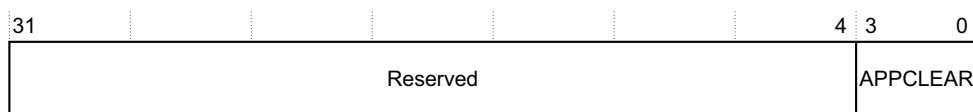
**Note**

The CTIINEN Registers do not affect the CTIAPPSET operation.

#### 15.6.4 CTI Application Trigger Clear Register, CTIAPPCLEAR

CTIAPPCLEAR is a read/write register. A write to this register clears a channel event, corresponding to the bit written to.

Figure 15-7 shows the bit arrangement of the CTIAPPCLEAR Register.



**Figure 15-7 CTI Application Trigger Clear Register format**

Table 15-7 shows how the bit values correspond with the CTIAPPCLEAR Register functions.

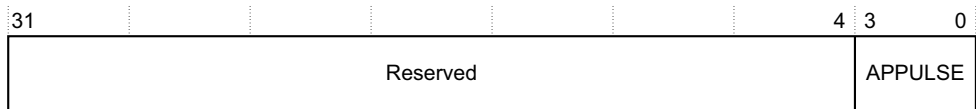
**Table 15-7 CTI Application Trigger Clear Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZ.
[3:0]	APPCLEAR	Clears corresponding bits in the CTIAPPSET Register: 0 = no effect 1 = application trigger disabled in the CTIAPPSET Register. There is one bit of the register for each channel.

### 15.6.5 CTI Application Pulse Register, CTIAPPULSE

CTIAPPULSE is a write-only register. A write to this register generates a channel event pulse, one **CTICK** period, corresponding to the bit written to. The pulse external to the CTI can be extended to multi-cycle by the handshaking interface circuits. This register clears itself immediately, so it can be repeatedly written to without software having to clear it.

Figure 15-8 shows the bit arrangement of the CTIAPPULSE Register.



**Figure 15-8 CTI Application Pulse Register format**

Table 15-8 shows how the bit values correspond with the CTIAPPULSE Register functions.

**Table 15-8 CTI Application Pulse Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved, SBZ.
[3:0]	APPULSE	Setting a bit HIGH generates a channel event pulse for the selected channel. For write: 0 = no effect 1 = channel event pulse generated for one <b>CTICK</b> period. There is one bit of the register for each channel.

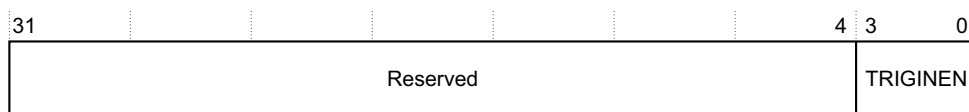
**Note**

The CTIINEN registers do not affect the CTIAPPULSE operation.

### 15.6.6 CTI Trigger to Channel Enable Registers, CTIINEN0-8

These registers are read/write registers that enable the signalling of an event on a CTM channel or CTM channels when the core issues a **CTITRIGIN** trigger input to the CTI. There is one register for each of the nine trigger inputs. Only seven trigger inputs are used, so CTIINEN7 and CTIINEN8 are present but not used. Within each register there is one bit for each of the four channels implemented. These registers do not affect the application trigger operations.

Figure 15-9 shows the bit arrangement of these registers.



**Figure 15-9** CTI Trigger to Channel Enable Registers format

Table 15-9 shows how the bit values correspond with these registers.

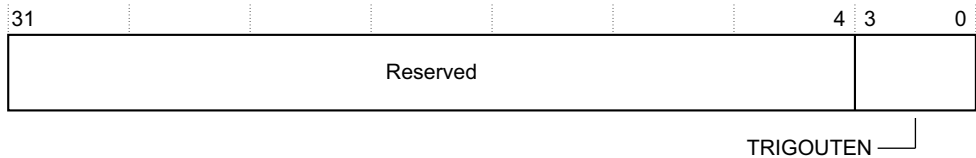
**Table 15-9** CTI Trigger to Channel Enable Registers bit functions

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZ.
[3:0]	TRIGINEN	Enables a cross trigger event to the corresponding channel when <b>CTITRIGIN</b> is activated: 0 = disables the <b>CTITRIGIN</b> signal from generating an event on the respective channel of the CTM 1 = enables the <b>CTITRIGIN</b> signal to generate an event on the respective channel of the CTM. There is one bit of the register for each of the four channels. For example, <b>TRIGINEN[0]</b> set to 1 in Register CTIINEN0, enables <b>CTITRIGIN</b> onto channel 0.

### 15.6.7 CTI Channel to Trigger Enable Registers, CTIOUTEN0-8

These registers are read/write registers that define which channel can generate a **CTITRIGOUT** output. There is one register for each of the nine **CTITRIGOUT** outputs. Within each register there is one bit for each of the four channels implemented. These registers affect the mapping from application trigger to trigger outputs.

Figure 15-10 on page 15-18 shows the bit assignments of these registers.



**Figure 15-10 CTI Channel to Trigger Enable Registers format**

Table 15-10 shows how the bit values correspond with these registers.

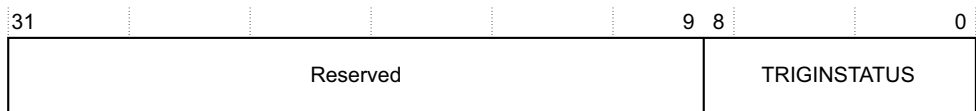
**Table 15-10 CTI Channel to Trigger Enable Registers bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZ.
[3:0]	TRIGOUTEN	Enables a channel event for the corresponding channel to generate an <b>CTITRIGOUT</b> output: 0 = the channel input <b>CTICHIN</b> from the CTM is not routed to the <b>CTITRIGOUT</b> output 1 = the channel input <b>CTICHIN</b> from the CTM is routed to the <b>CTITRIGOUT</b> output. There is one bit of the register for each of the four channels. For example, enabling bit [0] in Register CTIOUTEN0, enables <b>CTICHIN[0]</b> to cause a trigger event on the <b>CTITRIGOUT[0]</b> output.

### 15.6.8 CTI Trigger In Status Register, CTITRIGINSTATUS

CTITRIGINSTATUS is a read-only register that provides the status of the **CTITRIGIN** inputs.

Figure 15-11 shows the bit arrangement of the CTITRIGINSTATUS Register.



**Figure 15-11 CTI Trigger In Status Register format**

Table 15-11 shows how the bit values correspond with the CTITRIGINSTATUS Register functions.

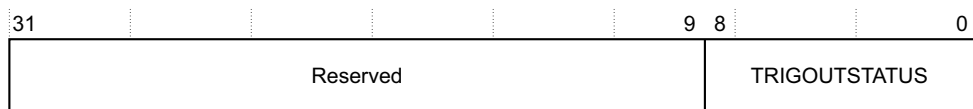
**Table 15-11 CTI Trigger In Status Register bit functions**

Bits	Field	Function
[31:9]	-	Reserved, RAZ.
[8:0]	TRIGINSTATUS	Displays the status of the <b>CTITRIGIN</b> inputs: 0 = <b>CTITRIGIN</b> is inactive 1 = <b>CTITRIGIN</b> is active. Because the register provides a view of the raw <b>CTITRIGIN</b> inputs, the reset value is unknown. There is one bit of the register for each trigger input.

### 15.6.9 CTI Trigger Out Status Register, CTITRIGOUTSTATUS

CTITRIGOUTSTATUS is a read-only register that provides the status of the **CTITRIGOUT** outputs.

Figure 15-12 shows the bit arrangement of the CTITRIGOUTSTATUS Register.



**Figure 15-12 CTI Trigger Out Status Register format**

Table 15-12 shows how the bit values corresponds with the CTITRIGOUTSTATUS Register functions.

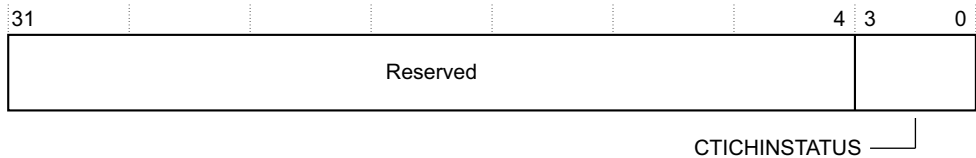
**Table 15-12 CTI Trigger Out Status Register bit functions**

Bits	Field	Function
[31:9]	-	Reserved, RAZ.
[8:0]	TRIGOUTSTATUS	Displays the status of the <b>CTITRIGOUT</b> outputs: 0 = <b>CTITRIGOUT</b> is inactive (reset) 1 = <b>CTITRIGOUT</b> is active. There is one bit of the register for each trigger output.

### 15.6.10 CTI Channel In Status Register, CTICHINSTATUS

CTICHINSTATUS is a read-only register that provides the status of the **CTICHIN** inputs.

Figure 15-13 shows the bit arrangement of the CTICHINSTATUS Register.



**Figure 15-13 CTI Channel In Status Register format**

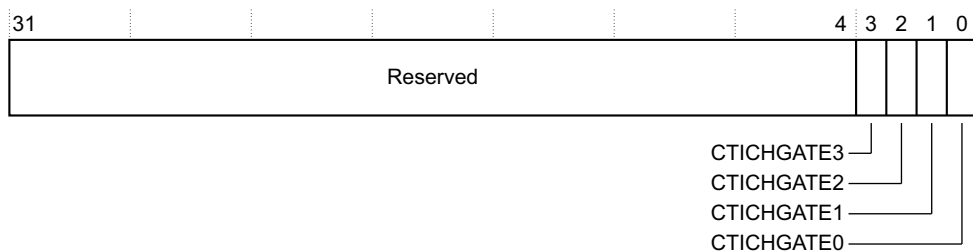
Table 15-13 shows how the bit values correspond with the CTICHINSTATUS Register functions.

**Table 15-13 CTI Channel In Status Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved, RAZ.
[3:0]	CTICHINSTATUS	Displays the status of the <b>CTICHIN</b> inputs: 0 = <b>CTICHIN</b> is inactive 1 = <b>CTICHIN</b> is active. Because the register provides a view of the raw <b>CTICHIN</b> inputs from the CTM, the reset value is unknown. There is one bit of the register for each channel input.

### 15.6.11 CTI Channel Gate Register, CTICHGATE

The CTICHGATE Register is a read/write register that controls the propagation of events to the channel interface. Figure 15-14 on page 15-21 shows the bit arrangement of the CTICHGATE Register.



**Figure 15-14 CTI Channel Gate Register format**

Table 15-14 shows how the bit values correspond with the CTICHGATE Register functions.

**Table 15-14 CTI Channel Gate Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved. RAZ, SBZ.
[3]	CTICHGATE3	Enable CTICHOUT3. Set to 0 to disable channel propagation.
[2]	CTICHGATE2	Enable CTICHOUT2. Set to 0 to disable channel propagation.
[1]	CTICHGATE1	Enable CTICHOUT1. Set to 0 to disable channel propagation.
[0]	CTICHGATE0	Enable CTICHOUT0. Set to 0 to disable channel propagation.

The Channel Gate Register prevents events from propagating through the channel interface to other CTIs. This enables local cross-triggering, such as causing an interrupt when the ETM trigger occurs. You can use the CTICHGATE Register with the CTIAPPSET, CTIAPPCLEAR and CTIAPPULSE Registers to assert trigger outputs by asserting channels, without affecting the rest of the system.

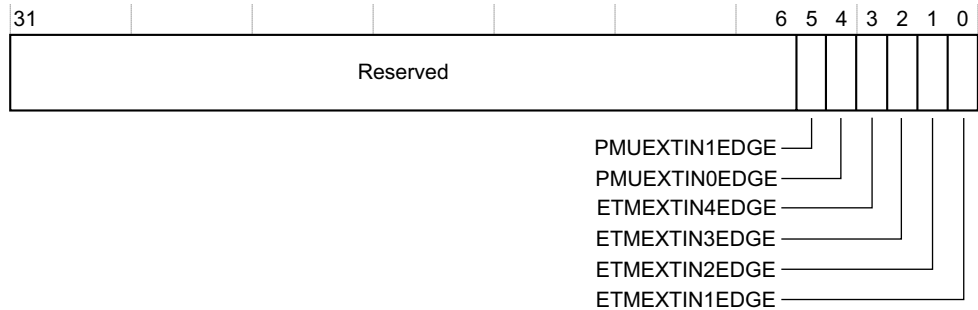
**Note**

This register is set to 0xF on reset, this causes channel interface propagation to be enabled for all channels.

See Figure 15-2 on page 15-4 for more information.

### 15.6.12 ASIC Control Register, ASICCTL

The ASICCTL Register is a read/write register that controls edge detection on trigger outputs. Figure 15-15 on page 15-22 shows the bit assignments of the ASIC Control Register.



**Figure 15-15 ASIC Control Register format**

Table 15-15 shows how the bit values correspond with the ASIC Control Register functions.

**Table 15-15 ASIC Control Register bit functions**

Bits	Field	Function
[31:6]	-	Reserved. RAZ, SBZ.
[5]	PMUEXTIN1EDGE	Enables edge detection for trigger output 6, PMU CTI event 1.
[4]	PMUEXTIN0EDGE	Enables edge detection for trigger output 5, PMU CTI event 0.
[3]	ETMEXTIN4EDGE	Enables edge detection for trigger output 4, ETM external input 4.
[2]	ETMEXTIN3EDGE	Enables edge detection for trigger output 3, ETM external input 3.
[1]	ETMEXTIN2EDGE	Enables edge detection for trigger output 2, ETM external input 2.
[0]	ETMEXTIN1EDGE	Enables edge detection for trigger output 1, ETM external input 1.

You can enable edge detection for each trigger output that is used in the CLK domain. If edge detection is enabled:

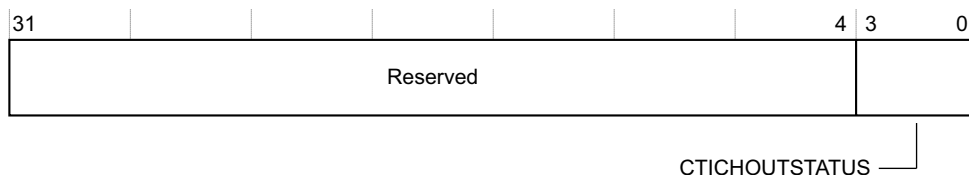
- a single PMU CTI event is generated for every rising edge of the trigger output
- the ETM external input is HIGH for one CLK cycle for every rising edge of the trigger output.

### 15.6.13 CTI Channel Out Status Register, CTICHOUTSTATUS

CTICHOUTSTATUS is a read-only register that provides the status of the CTI **CTICHOUT** outputs.

Figure 15-16 shows the bit arrangement of the CTICHOUTSTATUS Register.





**Figure 15-16 CTI Channel Out Status Register format**

Table 15-16 shows how the bit values correspond with the CTICHOUTSTATUS Register functions.

**Table 15-16 CTI Channel Out Status Register bit functions**

Bits	Field	Function
[31:4]	-	Reserved, RAZ.
[3:0]	CTICHOUTSTATUS	Displays the status of the <b>CTICHOUT</b> outputs: 0 = <b>CTICHOUT</b> is inactive (reset) 1 = <b>CTICHOUT</b> is active. There is one bit of the register for each channel output.

## 15.7 CTI Integration Test Registers

Integration Test Registers are provided to simplify the process of verifying the integration of the CTI with other devices in a CoreSight system. These registers enable direct control of outputs and the ability to read the value of inputs. You must only use these registers when the Integration Test Control Register bit [0] is set to 1.

See the *CoreSight Implementation and Integration Manual* for details of how to use these signals.

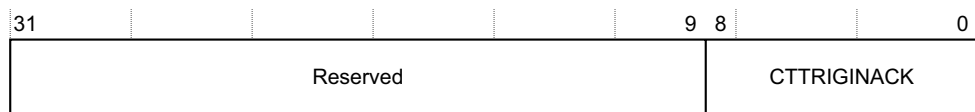
Table 15-17 shows the CTI Integration Test Registers.

**Table 15-17 CTI Integration Test Registers**

Address offset	Register	Access	Width	Description
0xEE0	ITTRIGINACK	W	9 bits	<i>ITTRIGINACK</i> , 0xEE0
0xEE4	ITCHOUT	W	4 bits	<i>ITCHOUT</i> , 0xEE4 on page 15-25
0xEE8	ITTRIGOUT	W	9 bits	<i>ITTRIGOUT</i> , 0xEE8 on page 15-26
0xEF0	ITTRIGOUTACK	R	9 bits	<i>ITTRIGOUTACK</i> , 0xEF0 on page 15-27
0xEF4	ITCHIN	R	4 bits	<i>ITCHIN</i> , 0xEF4 on page 15-27
0xEF8	ITTRIGIN	R	9 bits	<i>ITTRIGIN</i> , 0xEF8 on page 15-28

### 15.7.1 ITTRIGINACK, 0xEE0

ITTRIGINACK is a write-only register. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 15-17 shows the bit arrangement of the ITTRIGINACK Register.



**Figure 15-17 ITTRIGINACK Register format**



### 15.7.3 ITTRIGOUT, 0xEE8

ITTRIGOUT is a write-only register. This register controls signal outputs when bit [0] of the Integration Mode Control Register is set to 1. Figure 15-19 shows the bit arrangement of the ITTRIGOUT Register.



**Figure 15-19 ITTRIGOUT Register format**

Table 15-20 shows how the bit values correspond with the ITTRIGOUT Register functions.

**Table 15-20 ITTRIGOUT Register bit functions**

Bits	Field	Function
[31:9]	-	Reserved, SBZ
[8:0]	CTTRIGOUT	Sets the value of the <b>CTTRIGOUT</b> outputs

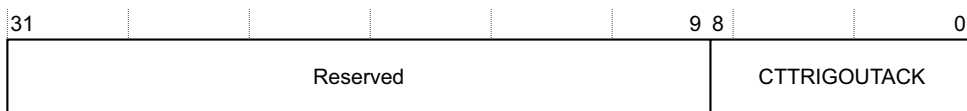
Each bit of the ITTRIGOUT Register corresponds to a trigger output. Table 15-21 shows how some of the bits of ITTRIGOUT are connected to other integration test registers in the processor.

**Table 15-21 ITTRIGOUT connections to other integration test registers**

Bits	Field	Connected to	Register name	Address bit
[8]	!nCTIIRQ	-	-	-
[7]	DBGRESTART	Debug	Integration Input Status Register, 0xEFC	[11]
[6]	PMUEXTIN[1]	Debug	Integration Input Status Register, 0xEFC	[9]
[5]	PMUEXTIN[0]	Debug	Integration Input Status Register, 0xEFC	[8]
[4]	EXTIN[3]	ETM	ITMISCIN Register, 0xEE0	[3]
[3]	EXTIN[2]	ETM	ITMISCIN Register, 0xEE0	[2]
[2]	EXTIN[1]	ETM	ITMISCIN Register, 0xEE0	[1]
[1]	EXTIN[0]	ETM	ITMISCIN Register, 0xEE0	[0]
[0]	DBGACK	Debug	Integration Input Status Register, 0xEFC	[10]

### 15.7.4 ITTRIGOUTACK, 0xEF0

ITTRIGOUTACK is a read-only register. This register enables the values of signal inputs to be read when bit [0] of the Integration Mode Control Register is set to 1. Figure 15-20 shows the bit arrangement of the ITTRIGOUTACK Register.



**Figure 15-20 ITTRIGOUTACK Register format**

Table 15-22 shows how the bit values correspond with the ITTRIGOUTACK Register functions.

**Table 15-22 ITTRIGOUTACK Register bit functions**

Bits	Field	Function
[31:9]	-	Reserved, RAZ
[8:0]	CTTRIGOUTACK	Reads the values of the <b>CTTRIGOUTACK</b> inputs

Each bit of the ITTRIGOUTACK Register corresponds to a bit on the ITTRIGOUT Register. It indicates when a trigger output has been received.

Table 15-23 shows how some of the bits of the ITTRIGOUTACK Register are connected to other integration test registers in the processor.

**Table 15-23 ITTRIGOUTACK connections to other integration test registers**

Bits	Field	Connected to	Register name	Address bit
[8]	-	-	-	-
[7]	DBGRESTARTED	Debug	Integration Internal Output Control Register, 0xEF4	[4]
[6:1]	-	-	-	-
[0]	DBGACK	Debug	Integration Internal Output Control Register, 0xEF4	[0]

### 15.7.5 ITCHIN, 0xEF4

ITCHIN is a read-only register. This register enables the values of signal inputs to be read when bit [0] of the Integration Mode Control Register is set to 1. Figure 15-21 on page 15-28 shows the bit arrangement of the ITCHIN Register.



Figure 15-21 ITCHIN Register format

Table 15-24 shows how the bit values correspond with the ITCHIN Register functions.

Table 15-24 ITCHIN Register bit functions

Bits	Field	Function
[31:4]	-	Reserved, RAZ
[3:0]	CTCHIN	Reads the values of the <b>CTCHIN</b> inputs

### 15.7.6 ITTRIGIN, 0xEF8

ITTRIGIN is a read-only register. This register enables the values of signal inputs to be read when bit [0] of the Integration Mode Control Register is set to 1. Figure 15-22 shows the bit arrangement of the ITTRIGIN Register.



Figure 15-22 ITTRIGIN Register format

Table 15-25 shows how the bit values correspond with the ITTRIGIN Register functions.

Table 15-25 ITTRIGIN Register bit functions

Bits	Field	Function
[31:9]	-	Reserved, RAZ
[8:0]	CTTRIGIN	Reads the values of the <b>CTTRIGIN</b> inputs

Each bit of the ITTRIGIN Register corresponds to a trigger input. Table 15-26 on page 15-29 shows how some of the bits of ITTRIGIN are connected to other integration test registers in the processor.

Table 15-26 ITTRIGIN connections to other integration test registers

Bits	Field	Connected to	Register name	Address bit
[8]	-	-	-	-
[7]	-	-	-	-
[6]	TRIGGER	ETM	ITTRIGGER Register, 0xEE8	[0]
[5]	COMMTX	Debug	Integration Internal Output Control Register, 0xEF4	[2]
[4]	COMMRX	Debug	Integration Internal Output Control Register, 0xEF4	[1]
[3]	EXTOUT[1]	ETM	ITMISCOUT Register, 0xEDC	[1]
[2]	EXTOUT[0]	ETM	ITMISCOUT Register, 0xEDC	[0]
[1]	!nPMUIRQ	Debug	Integration Internal Output Control Register, 0xEF4	[3]
[0]	Debug entry	Debug	Integration Internal Output Control Register, 0xEF4	[5]

## 15.8 CTI CoreSight defined registers

See the *CoreSight Architecture Specification* for definitions of CTI CoreSight defined registers. The information given in this section is specific to the CTI.

### 15.8.1 Authentication Status Register, 0xFB8

The Authentication Status Register reports the required security level. Table 15-27 shows how the bit values correspond with the Authentication Status Register functions.

**Table 15-27 Authentication Status Register bit functions**

Bits	Value	Function
[31:4]	0x0000000	Reserved, RAZ.
[3]	-	Noninvasive debug enabled, <b>DBGEN</b> or <b>NIDEN</b> . When this bit is LOW, all trigger inputs are disabled.
[2]	b1	Noninvasive debug features supported.
[1]	-	Invasive debug enabled, <b>DBGEN</b> . When this bit is LOW, the following trigger outputs are disabled: <ul style="list-style-type: none"> <li>ETM external inputs, <b>EXTIN[3:0]</b></li> <li>PMU external inputs, <b>PMUEXTIN[3:0]</b>.</li> </ul>
[0]	b1	Invasive debug features supported.

### 15.8.2 Device ID Register, 0xFC8

The Device ID Register reports the configuration of the CTI. For the Cortex-A8 processor, the CTI Device ID is 0x40900. Table 15-28 shows how the bit values correspond with the Device ID Register functions.

**Table 15-28 Device ID Register bit functions**

Bits	Value	Function
[31:20]	0x000	Reserved, RAZ
[19:16]	0x4	Number of CTI channels available
[15:8]	0x09	Number of CTI triggers available
[7:5]	b000	Reserved, RAZ
[4:0]	b00000	Number of multiplexing levels on CTI inputs and outputs



### 15.8.3 Device Type Identifier, 0xFCC

The Device Type Identifier Register indicates the type of CoreSight component. Table 15-29 shows how the bit values correspond with the Device Type Identifier Register functions.

**Table 15-29 Device Type Identifier Register bit functions**

Bits	Value	Function
[31:8]	0x000000	Reserved, RAZ
[7:4]	0x1	Minor type, cross trigger
[3:0]	0x4	Major type, debug control logic

### 15.8.4 Peripheral Identification Registers

The CTI Peripheral Identification Registers are a set of eight read-only registers, PeripheralID7 to PeripheralID0. Only bits [7:0] of each register are used.

Table 15-30 shows the bit field definitions of the Peripheral Identification Registers. The *CoreSight Architecture Specification* describes many of these fields in more detail.

**Table 15-30 Peripheral Identification Registers bit functions**

Register name	Offset	Bits	Value	Function
PeripheralID7	0xFDC	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID6	0xFD8	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID5	0xFD4	[31:8]	-	Unused, RAZ
		[7:0]	0x00	Reserved for future use, RAZ
PeripheralID4	0xFD0	[31:8]	-	Unused, RAZ
		[7:4]	0x0	Indicates that the ETM uses one 4KB block of memory
		[3:0]	0x4	JEP106 continuation code [3:0]
PeripheralID3	0xFEC	[31:8]	-	Unused, RAZ
		[7:4]	0x1	RevAnd (at top level)

**Table 15-30 Peripheral Identification Registers bit functions (continued)**

Register name	Offset	Bits	Value	Function
		[3:0]	0x0	Customer Modified 0x00 indicates from ARM
PeripheralID2	0xFE8	[31:8]	-	Unused, RAZ
		[7:4]	0x6	Revision number of Peripheral
		[3]	0x1	Indicates that a JEDEC assigned value is used
		[2:0]	0x3	JEP106 identity code [6:4]
PeripheralID1	0xFE4	[31:8]	-	Unused, RAZ
		[7:4]	0xB	JEP106 identity code [3:0]
		[3:0]	0x9	Part number 1 upper <i>Binary Coded Decimal</i> (BCD) value of Device number
PeripheralID0	0xFE0	[31:8]	-	Unused, RAZ
		[7:0]	0x22	Part number 0 middle and lower BCD value of Device number

**Note**

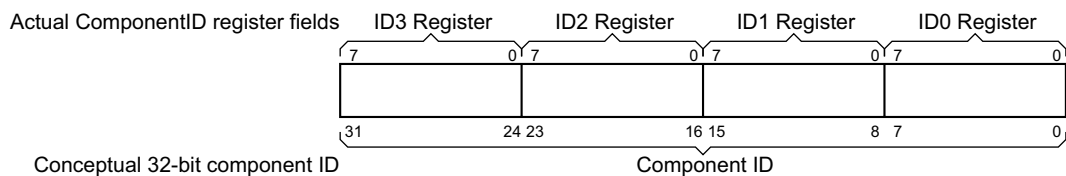
In Table 15-30 on page 15-31, the Peripheral Identification Registers are listed in order of register name, from most significant (ID7) to least significant (ID0). This does not match the order of the register offsets. Similarly, in Table 15-31 on page 15-33, the Component Identification Registers are listed in order of register name, from most significant (ID3) to least significant (ID0).

**15.8.5 Component Identification Registers**

There are four read-only Component Identification Registers, ComponentID3 to ComponentID0. Although these are implemented as standard 32-bit registers:

- the most significant 24 bits of each register are not used and *Read-As-Zero* (RAZ)
- the least significant eight bits of each register together make up the component ID.

Figure 15-23 on page 15-33 shows this concept of a single 32-bit component ID, obtained from the four Component Identification Registers.



**Figure 15-23 Mapping between the Component ID Registers and the component ID value**

Table 15-31 shows the bit field definitions of the Component Identification Registers. This register structure is defined in the *CoreSight Architecture Specification*.

**Table 15-31 Component Identification Registers bit functions**

Register	Register offset	Bits	Value	Function
ComponentID3	0xFFC	[31:8]	-	Unused, RAZ
		[7:0]	0xB1	Component identifier, bits [31:24]
ComponentID2	0xFF8	[31:8]	-	Unused, RAZ
		[7:0]	0x05	Component identifier, bits [23:16]
ComponentID1	0xFF4	[31:8]	-	Unused, RAZ
		[7:4]	0x9	Component class; component identifier, bits [15:12]
		[3:0]	0x0	Component identifier, bits [11:8]
ComponentID0	0xFF0	[31:8]	-	Unused, RAZ
		[7:0]	0x0D	Component identifier, bits [7:0]



# Chapter 16

## Instruction Cycle Timing

This chapter describes the cycle timings of instructions on the processor. It contains the following sections:

- *About instruction cycle timing* on page 16-2
- *Instruction-specific scheduling for ARM instructions* on page 16-3
- *Dual-instruction issue restrictions* on page 16-16
- *Other pipeline-dependent latencies* on page 16-17
- *Advanced SIMD instruction scheduling* on page 16-21
- *Instruction-specific scheduling for Advanced SIMD instructions* on page 16-23
- *VFP instructions* on page 16-44
- *Scheduling example* on page 16-49.

## 16.1 About instruction cycle timing

This chapter provides the information to estimate how much execution time particular code sequences require. The complexity of the processor makes it impossible to guarantee precise timing information with hand calculations. The timing of an instruction is often affected by other concurrent instructions, memory system activity, and additional events outside the instruction flow. Describing all possible instruction interactions and all possible events taking place in the processor is beyond the scope of this document. Only a cycle-accurate model of the processor can produce precise timings for a particular instruction sequence.

This chapter provides a framework for doing basic timing estimations for instruction sequences. The framework requires three main information components:

### **Instruction-specific scheduling information**

This includes the number of micro-operations for each main instruction and the source and destination requirements for each micro-operation. The processor can issue a series of micro-operations to the execution pipeline for each ARM instruction executed. Most ARM instructions execute only one micro-operation. More complex ARM instructions such as load multiples can consist of several micro-operations.

### **Dual issue restriction criteria**

This is the set of rules used to govern which instruction types can dual issue and under what conditions. This information is provided for dual issue of ARM instructions and Advanced SIMD instructions.

### **Other pipeline-dependent latencies**

In addition to the time taken for the scheduling and issuing of instructions, there are other sources of latencies that effect the time of a program sequence. The two most common examples are a branch mispredict and a memory system stall such as a data cache miss of a load instruction. These cases are the most difficult to predict and often must be ignored or estimated using statistical analysis techniques. Fortunately, you can ignore most of these additional latencies when creating an optimal hand scheduling for a code sequence. Hand scheduling is the most useful application of this cycle timing information.

## 16.2 Instruction-specific scheduling for ARM instructions

The tables in this section provide information to determine the best-case instruction scheduling for a sequence of instructions. The information includes:

- when source registers are required
- when destination registers are available
- which register, such as Rn or Rm, is meant for each source or destination
- the minimum number of cycles required for each instruction
- any additional instruction issue requirements or restrictions.

When a source register is required or a destination register is available depends on the availability of forwarding paths to route the required data from the correct source to the correct destination.

Special considerations and caveats concerning the instruction tables include:

- Source requirements are always given for the first cycle in a multi-cycle instruction.
- Destination available is always given with respect to the last cycle in a data processing multi-cycle instruction. This rule does not apply to load/store multiple instructions.
- Multiply instructions issue to pipeline 0 only.
- Flags from the CPSR Register are updated internally in the E2 stage.
- [Rd] as a source register indicates the destination register is required as a source if the instruction is conditional.
- {} on a source register indicate the register is required only if the instruction includes an accumulator operand.
- () on a destination register indicate the destination is required only if writeback is enabled.
- [] on a load instruction destination register indicate that the destination register is optional depending on the size of the data transferred.

### 16.2.1 Example of how to read ARM instruction tables

This section provides examples of how to read ARM instruction tables described in the chapter. See the *ARM Architecture Reference Manual* for assembly syntax of instructions.

Example 16-1 shows how to read an ADDEQ data-processing instruction from Table 16-1 on page 16-5.

---

#### Example 16-1 Data-processing instruction

---

ADDEQ R0, R1, R2 LSL#10

This is a conditional general data-processing instruction of type shift by immediate. Source1, in this case R1, is required in E2 and Source2, in this case R2, is required in E1. Because the instruction is conditional, the destination register R0 is also required as a source register and must be available in E2. The result, stored in R0 for this case, is available in E2 for the next subsequent instruction that requires this register as a source operand. Assuming no data hazards, the instruction takes a minimum of one cycle to execute as indicated by the value in the Cycles column.

---

Example 16-2 shows how to read an SMLAL multiply instruction from Table 16-4 on page 16-7.

---

#### Example 16-2 Multiply instruction

---

SMLAL R0, R1, R2, R3

This is a multiply accumulate instruction. Source1, in this case R2, and Source2, in this case R3, are both required in E1. Because this is an accumulate multiply instruction, the result registers, R0 and R1, in this case are both required as source registers in E1. The result, stored in R0 and R1, for this case is available in E5 for the next subsequent instruction that requires one or both of these registers as a source operand. Assuming no data hazards, the instruction takes a minimum of three cycles to execute as indicated by the value in the Cycles column.

---

Example 16-3 shows how to read an LDR PC load instruction from Table 16-9 on page 16-9.

---

#### Example 16-3 Load instruction

---

LDR PC, [R13,#4]

This is a load instruction of type immediate offset. However, it is also a branch instruction because the PC is the destination. Source1, in this case R13, is required in E1. Because writeback is enabled on this load instruction, Source1, in this case R13, is also required as a result destination register for writing back the new address. This result



is available in E2 for the next subsequent instruction that requires this register as a source operand. Assuming no data hazards, the instruction takes a minimum of one cycle to execute as indicated by the value in the Cycles column. To complete the timing calculation for this instruction, we use information for the branch instructions as shown in Table 16-11 on page 16-12. In this table, we can see that the instruction is unconditional, therefore no flags are required as a source in E3 for branch resolution. The Cycles column of Table 16-11 on page 16-12 indicates to add one cycle to the total execution time for all load instructions that are branches. Assuming no data hazards, the instruction takes a minimum of two cycles instead of one cycle.

## 16.2.2 Data-processing instructions

Data-processing instructions are divided into the following subcategories:

### Data-processing instructions with a destination

AND, EOR, SUB, RSB, ADD, ADC, SBC, RCSC, ORR, BIC

### Data-processing without a destination

TST, TEQ, CMP, CMN

### Move instructions

MOV, MVN

The data-processing instruction tables exclude cases where the PC is the destination. *Branch instructions* on page 16-11 describes these cases.

Table 16-1 shows the operation of data-processing instructions that use a destination.

**Table 16-1 Data-processing instructions with a destination**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Immediate	1	Rn:E2	[Rd:E2]	-	-	Rd:E2	-
Register	1	Rn:E2	Rm:E2	[Rd:E2]	-	Rd:E2	-
Shift by immediate, non-RRX	1	Rn:E2	Rm:E1	[Rd:E2]	-	Rd:E2	-
Shift by immediate, RRX <sup>a</sup>	1	Rn:E2	Rm:E1	[Rd:E2]	-	Rd:E2	-
Shift by register	1	Rn:E2	Rm:E1	Rs:E1	[Rd:E2]	Rd:E2	-

a. One-cycle stall required before instruction execution.

Table 16-2 shows the operation of data-processing instructions that do not use a destination.

**Table 16-2 Data-processing instructions without a destination**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Immediate	1	Rn:E2	-	-	-	-	-
Register	1	Rn:E2	Rm:E2	-	-	-	-
Shift by immediate, non-RRX	1	Rn:E2	Rm:E1	-	-	-	-
Shift by immediate, RRX <sup>a</sup>	1	Rn:E2	Rm:E1	-	-	-	-
Shift by register	1	Rn:E2	Rm:E1	Rs:E1	-	-	-

a. One-cycle stall required before instruction execution.

Table 16-3 shows the operation of MOV and MOVN instructions.

**Table 16-3 MOV and MOVN instructions**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Immediate <sup>a</sup>	1	[Rd:E2]	-	-	Rd:E1/E2	-	-
Register <sup>a</sup>	1	Rn:E1	[Rd:E2]	-	Rd:E1/E2	-	-
Shift by immediate, non-RRX <sup>a</sup>	1	Rn:E1	[Rd:E2]	-	Rd:E1/E2	-	-
Shift by immediate, RRX <sup>b</sup>	1	Rn:E1	[Rd:E2]	-	Rd:E1/E2	-	-
Shift by register	1	Rn:E1	Rs:E1	[Rd:E2]	Rd:E1/E2	-	-

a. Result is available in E2 if conditional.

b. Result is available in E2 if conditional. One-cycle stall required before instruction execution.

### 16.2.3 Multiply instructions

The ARM multiply instructions include MLA, MUL, SLMAxy, SMLAD, SMLAL, SMLALxy, SMLALD, SMLAWy, SMLD, SMLS LD, SMMLA, SMMLS, SMMUL, SMUAD, SMULxy, SMULL, SMULW, SMUSD, UMAAL, UMLAL, UMULL.

Table 16-4 shows the operation of multiply instructions.

**Table 16-4 Multiply instructions**

Multiply type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Normal: MUL	2	Rm:E1	Rs:E1	[Rd:E3]	{Rn:E4} <sup>a</sup>	Rd:E5	-
Long: SMULL, UMULL	3	Rm:E1	Rs:E1	{[RdLo:E3]}	{[RdHi:E3]}	RdLo:E5	RdHi:E5
Long: SMLAL, UMLAL, UMAAL	3	Rm:E1	Rs:E1	{[RdLo:E2]}	{[RdHi:E1]}	RdLo:E5	RdHi:E5
Halfword: SMLAxy, SMULxy	2	Rm:E1	Rs:E1	[Rd:E2]	{Rn:E4} <sup>a</sup>	Rd:E5	-
Halfword: SMLALxy	2	Rm:E1	Rs:E1	{[RdLo:E1]}	{[RdHi:E2]}	RdLo:E5	RdHi:E5
Word-halfword: SMULWy	1	Rm:E1	Rs:E1	[Rd:E2]	-	Rd:E5	-
Word-halfword: SMLAWy	2	Rm:E1	Rs:E1	[Rd:E2]	Rn:E4 <sup>a</sup>	Rd:E5	-
Most significant word	2	Rm:E1	Rs:E1	[Rd:E3]	{Rn:E4} <sup>a</sup>	Rd:E5	-
Dual halfword: SMUAD, SMUSD	1	Rm:E1	Rs:E1	[Rd:E2]	-	Rd:E5	-
Dual halfword: SMLAD, SMLSD	2	Rm:E1	Rs:E1	[Rd:E2]	{Rn:E4} <sup>a</sup>	Rd:E5	-
Dual halfword: SMLALD, SMLSLD	2	Rm:E1	Rs:E1	{[RdLo:E1]}	{[RdHi:E2]}	RdLo:E5	RdHi:E5

- a. A multiply that is followed by a MAC with a dependency on the accumulator, Rn register, triggers a special accumulator forwarding. This enables both instructions to issue back-to-back because Rn is required as a source in E4. If this accumulator forwarding is not used, Rn is required in E2.

## 16.2.4 Parallel arithmetic and DSP instructions

The parallel arithmetic instructions include ADD15, ADDSUBX, SUBADDX, SUB16, ADD8, SUB8, QDADD, QDSUB, QADD, QSUB.

Table 16-5 shows the operation of parallel arithmetic instructions.

**Table 16-5 Parallel arithmetic instructions**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Shifter required: ADDSUB, SUBADD, QD	1	Rm:E2	Rn:E1	[Rd:E2]	-	Rd:E3	-
No shifter required: all others	1	Rm:E2	Rn:E2	[Rd:E2]	-	Rd:E3	-

### 16.2.5 Extended instructions

The extended instructions include XTAB, XTAH, XTB, XTH.

Table 16-6 shows the operation of extended instructions.

**Table 16-6 Extended instructions**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
Versions without accumulate <sup>a</sup>	1	Rm:E1	[Rd:E2]	-	-	Rd:E1/E2	-
Versions with accumulate	1	Rm:E1	Rn:E2	[Rd:E2]	-	Rd:E2	-

a. If conditional, result is not available until E2.

### 16.2.6 Miscellaneous data-processing instructions

The miscellaneous data-processing instructions include PK, SAT, SEL.

Table 16-7 shows the operation of miscellaneous data-processing instructions.

**Table 16-7 Miscellaneous data-processing instructions**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
SAT <sup>a</sup>	1	Rm:E1	[Rd:E2]	-	-	Rd:E1/E2	-
CLZ	1	Rm:E2	[Rd:E2]	-	-	Rd:E2	-
USAD	1	Rm:E1	Rn:E1	[Rd:E2]	-	Rd:E5	-
PKT <sup>a</sup>	1	Rm:E1	Rn:E1	[Rd:E2]	-	Rd:E1/E2	-
SEL	1	Rm:E1	Rn:E1	[Rd:E2]	-	Rd:E2	-

a. If conditional, result is not available until E2.

## 16.2.7 Status register access instructions

The MRS, MSR, and CPS instructions modify the CPSR and SPSR registers. Table 16-8 shows the operation of the status register access instructions.

**Table 16-8 Status register access instructions**

Access type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
MRS <sup>a</sup>	1	Rd:E2	-	-	-	Rd:E1/E2	-
MSR <sup>b</sup>	1	Rm:E1	Rd:E2	-	-	-	-
CPS <sup>b</sup>	1	-	-	-	-	-	-
SETEND <sup>b</sup>	1	-	-	-	-	-	-

a. Serialize before the instruction.

b. Serialize between micro-operations. Force pipeline flush if updating CPSR. Serialize only for SPSR updates.

## 16.2.8 Load/store instructions

There are many key characteristics that define different load/store instructions including the addressing mode, the data type, data size, whether or not register writeback is enabled, and indexing mode. Table 16-9 and Table 16-10 on page 16-10 specify the timing for various load/store instruction types based on each of these characteristics, but only if that characteristic has an effect on timing. For example, data type and all data sizes except 64-bit offset do not affect instruction timing.

Table 16-9 shows the operation of load instructions.

**Table 16-9 Load instructions**

Addressing mode	Cycles	Source				Result		
		1	2	3	4	1	2	3
Immediate offset	1	Rn:E1	[Rd:E2]	-	-	Rd:E3	(Rn:E2)	-
Register offset	1	Rn:E1	Rm:E1	[Rd:E2]	-	Rd:E3	(Rn:E2)	-
Immediate 64-bit offset	2	Rn:E1	-	[Rd:E2]	-	Rd:E3	(Rn:E2)	[Rd+1]:E3, 2nd iteration

Table 16-9 Load instructions (continued)

Addressing mode	Cycles	Source				Result		
		1	2	3	4	1	2	3
Register 64-bit offset	2	Rn:E1	Rm:E1	[Rd:E2]	-	Rd:E3	(Rn:E2)	[Rd+1]:E3, 2nd iteration
Scaled register offset, LSL by 2	1	Rn:E1	Rm:E1	[Rd:E2]	-	Rd:E3	(Rn:E2)	-
Scaled register offset, other	2	Rn:E1	Rm:E1	[Rd:E2]	-	Rd:E3	(Rn:E2), 2nd iteration	-

Table 16-10 shows the operation of store instructions.

Table 16-10 Store instructions

Addressing mode	Cycles	Source1	Source2	Source3	Source4	Result1
Immediate offset	1	Rn:E1	Rd:E3	-	-	(Rn:E2)
Register offset	1	Rn:E1	Rm:E1	Rd:E3	-	(Rn:E2)
Immediate 64-bit offset	2	Rn:E1	Rd:E3	[Rd+1]:E3, 1st iteration	-	(Rn:E2)
Register 64-bit offset	2	Rn:E1	Rm:E1	Rd:E3	[Rd+1]:E3, 1st iteration	(Rn:E2)
Scaled register offset, LSL by 2	1	Rn:E1	Rm:E1	Rd:E3	-	(Rn:E2)
Scaled register offset, other	2	Rn:E1	Rm:E1	Rd:E3	-	(Rn:E2), 2nd iteration

### 16.2.9 Load multiple and store multiple instructions

The number of registers in the register list usually determines the number of cycles required to execute a load or store multiple instruction. The processor can load or store two 32-bit registers in each cycle. However, to access 64 bits, the address must be 64-bit aligned. Processor scheduling is static, and it is not possible to know the address alignment at schedule time. Therefore, scheduling for the first transfer of loads and the last transfer of stores must be done assuming the address might be unaligned.

The number of cycles required to execute an LDM or STM instruction is (number of registers / 2) or 2 cycles, whichever is greater.

If register writeback is enabled, it is done in the first iteration in the E2 stage, as it is done in normal load/store instructions.

## 16.2.10 Branch instructions

Any write to the PC is considered a branch. This section describes both standard B branch instructions in addition to different instruction types with the PC as the destination register. In general, branch instructions schedule very well and have very few hazards that prevent superscalar issue. There are several properties to the execution of branches that make them behave differently than other instructions.

### Conditional branches

Conditional branches are executed differently than other conditional instructions. Most conditional instructions take the destination register as an additional source and the condition codes are resolved in E2. Branches do not require the destination register, PC, as an additional source because they already use the PC as a source. They are also different than normal conditional operations because the flags resolve the condition codes in E3 rather than E2. This enables the pairing of a flag setting instruction and a branch in the same cycle.

### Branches with the PC as a source or destination

Using the PC as a source register does not generally result in scheduling hazards as for the case of a general-purpose register. This is because the PC values are predicted in the pipeline and are readily available to each instruction without any forwarding required. The only exception to this rule is that an instruction with a PC as a source register cannot be dual issued with an instruction that uses the PC as a destination register.

Other than the dual issue restriction, using the PC as a destination register does not result in a hazard for subsequent instructions for the same reason.

### Data processing-based branches

Data processing branches can have the same data hazards of nonbranch versions of these instructions for operands other than the PC.

### Load-based branches

An LDR PC or LDM PC instruction behaves like a normal load with the exception that it requires one additional cycle to execute.

Table 16-11 shows the behavior of branch instructions.

**Table 16-11 Branch instructions**

Shift type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
BCC	1	[Flags:E3]	-	-	-	R15:E4 <sup>a</sup>	-
BLCC, BLX	1	[Flags:E3]	-	-	-	R14:E3	R15:E4 <sup>a</sup>
BXCC	1	[Flags:E3]	Rm:E2	-	-	-	-
Data-processing branch <sup>b</sup>	Typically 1 <sup>c</sup>	[Flags:E3]	-	-	-	R15:E4 <sup>a</sup>	-
Load-based branch	Basic load plus one cycle <sup>d</sup>	[Flags:E3]	-	-	-	(Rn:E2)	-

a. Branch prediction resolution in E4.

b. ADD PC, R1, R2 and MOV PC, R4 are both examples of data-processing branches.

c. See *Data-processing instructions* on page 16-5 for more information on cycle counts and source registers.

d. See *Load/store instructions* on page 16-9 for more information on cycle counts and source registers.

### 16.2.11 Coprocessor instructions

The CP15 and CP14 instructions are used to access special-purpose registers that are distributed across the design. They also perform very specialized operations such as cache maintenance. The instructions affected are listed in Table 16-12 and in Table 16-13 on page 16-13. The minimum time to complete these CP15 and CP14 operations is 60 cycles. However, the timing of these instructions varies highly. It can take hundreds of cycles, depending on the operation and on the current processor activity.

**Table 16-12 Nonpipelined CP14 instructions**

Instruction	Op1	<Rd>	CRn	CRm	Op2
MCR/MRC p14	0	Rd	c0-c15	c0-c15	0-7



Table 16-13 Nonpipelined CP15 instructions

Instruction	Op1	<Rd>	CRn	CRm	Op2	Function
MCR p15	0	Rd	c1	c0	0	Control Register
MCR p15	0	Rd	c1	c0	1	Auxiliary Control Register
MCR p15	0	Rd	c2	c0	0	Translation Table Base 0 Register
MCR p15	0	Rd	c2	c0	1	Translation Table Base 1 Register
MCR p15	0	Rd	c2	c0	2	Translation Table Base Control Register
MCR p15	0	Rd	c3	c0	0	Domain Access Control Register
MCR/MRC p15	0	Rd	c5	c0	0	Data Fault Status Register
MCR/MRC p15	0	Rd	c5	c0	1	Instruction Fault Status Register
MCR/MRC p15	0	Rd	c5	c1	0	Data Auxiliary Fault Status Register
MCR/MRC p15	0	Rd	c5	c1	1	Instruction Auxiliary Fault Status Register
MCR/MRC p15	0	Rd	c6	c0	0	Data Fault Address Register
MCR/MRC p15	0	Rd	c6	c0	1	Instruction Fault Address Register
MCR p15	0	Rd	c7	c5	1	Invalidate I\$ Line by MVA to PoU
MCR p15	0	Rd	c7	c6	1	Invalidate D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c6	2	Invalidate D\$ Line by Set/Way
MCR p15	0	Rd	c7	c8	0-3	VA-to-PA translation in the Current World
MCR p15	0	Rd	c7	c8	4-7	VA-to-PA translation in the Other World
MCR p15	0	Rd	c7	c10	1	Clean D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c10	2	Clean D\$ Line by Set/Way
MCR p15	0	Rd	c7	c10	4	Data Synchronization Barrier
MCR p15	0	Rd	c7	c10	5	Data Memory Barrier
MCR p15	0	Rd	c7	c11	1	Clean D\$ Line by MVA to PoU
MCR p15	0	Rd	c7	c14	1	Clean and Invalidate D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c14	2	Clean and Invalidate D\$ Line by Set/Way

Table 16-13 Nonpipelined CP15 instructions (continued)

Instruction	Op1	<Rd>	CRn	CRm	Op2	Function
MCR p15	0	Rd	c8	c5	0	Invalidate I-TLB Unlocked Entries
MCR p15	0	Rd	c8	c5	1	Invalidate I-TLB Entry by MVA
MCR p15	0	Rd	c8	c5	2	Invalidate I-TLB Entry on ASID Match
MCR p15	0	Rd	c8	c6	0	Invalidate D-TLB Unlocked Entries
MCR p15	0	Rd	c8	c6	1	Invalidate D-TLB Entry by MVA
MCR p15	0	Rd	c8	c6	2	Invalidate D-TLB Entry on ASID Match
MCR p15	0	Rd	c8	c7	0	Invalidate Unified-TLB Unlocked Entries
MCR p15	0	Rd	c8	c7	1	Invalidate Unified-TLB Entry by MVA
MCR p15	0	Rd	c8	c7	2	Invalidate Unified-TLB Entry on ASID Match
MCR p15	1	Rd	c9	c0	0	L2\$ Lockdown Register
MCR p15	1	Rd	c9	c0	2	L2\$ Auxiliary Control Register
MCR p15	0	Rd	c10	c0	0	D-TLB Lockdown Register
MCR p15	0	Rd	c10	c0	1	I-TLB Lockdown Register
MCR p15	0	Rd	c10	c1	0	D-TLB Preload
MCR p15	0	Rd	c10	c1	1	I-TLB Preload
MCR p15	0	Rd	c10	c2	0	Primary Region Remap Register
MCR p15	0	Rd	c10	c2	1	Normal Memory Remap Register
MCR p15	0	Rd	c11	c1	0	PLE User Accessibility Register
MCR p15	0	Rd	c11	c2	0	PLE Channel Number Register
MCR p15	0	Rd	c11	c3	0-2	PLE Enable Register
MCR p15	0	Rd	c11	c4	0	PLE Control Register
MCR p15	0	Rd	c11	c15	0	PLE Context ID Register
MCR p15	0	Rd	c13	c0	0	FCSE ID Register
MCR p15	0	Rd	c13	c0	0	Context ID Register
MCR/MRC p15	-	Rd	c15	-	-	All Array Access instructions

Table 16-14 shows the CP15 instructions that have improved cycle timing if the Auxiliary Control Register bit[20] = 0,

**Table 16-14 CP15 instructions affected when ACTRL bit[20] = 0**

<b>Instruction</b>	<b>Op1</b>	<b>&lt;Rd&gt;</b>	<b>CRn</b>	<b>CRm</b>	<b>Op2</b>	<b>Function</b>
MCR p15	0	Rd	c7	c6	1	Invalidate D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c6	2	Invalidate D\$ Line by Set/Way
MCR p15	0	Rd	c7	c10	1	Clean D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c10	2	Clean D\$ Line by Set/Way
MCR p15	0	Rd	c7	c11	1	Clean D\$ Line by MVA to PoU
MCR p15	0	Rd	c7	c14	1	Clean and Invalidate D\$ Line by MVA to PoC
MCR p15	0	Rd	c7	c14	2	Clean and Invalidate D\$ Line by Set/Way

## 16.3 Dual-instruction issue restrictions

Calculating likely instruction pairings is part of the hand calculation process required to determine the timing for a sequence of instructions. The processor issues a pair of instructions unless it encounters an issue restriction. Table 16-15 shows the most common issue restriction cases. This table contains references to pipeline 0 and pipeline 1. The first instruction always issues in pipeline 0 and the second instruction, if present, issues in pipeline 1. If only one instruction issues, it always issues in pipeline 0.

**Table 16-15 Dual-issue restrictions**

Restriction type	Description	Example	Cycle	Restriction
Load/store resource hazard	There is only one LS pipeline. Only one LS instruction can be issued per cycle. It can be in pipeline 0 or pipeline 1	LDR r5, [r6]	1	
		STR r7, [r8]	2	Wait for LS unit
		MOV r9, r10	2	Dual issue possible
Multiply resource hazard	There is only one multiply pipeline, and it is only available in pipeline 0.	ADD r1, r2, r3	1	
		MUL r4, r5, r6	2	Wait for pipeline 0
		MUL r7, r8, r9	3	Wait for multiply unit
Branch resource hazard	There can be only one branch per cycle. It can be in pipeline 0 or pipeline 1. A branch is any instruction that changes the PC.	BX r1	1	
		BEQ 0x1000	2	Wait for branch
		ADD r1, r2, r3	2	Dual issue possible
Data output hazard	Instructions with the same destination cannot be issued in the same cycle. This can happen with conditional code.	MOVEQ r1, r2	1	
		MOVNE r1, r3	2	Wait because of output dependency
		LDR r5, [r6]	2	Dual issue possible
Data source hazard	Instructions cannot be issued if their data is not available. See the scheduling tables for source requirements and stages results.	ADD r1, r2, r3	1	
		ADD r4, r1, r6	2	Wait for r1
		LDR r7, [r4]	4	Wait two cycles for r4
Multi-cycle instruction	Multi-cycle instructions must issue in pipeline 0 and can only dual issue in their last iteration.	MOV r1, r2	1	
		LDM r3, {r4-r7}	2	Wait for pipeline 0, transfer r4
		LDM (cycle 2)	3	Transfer r5, r6
		LDM (cycle 3)	4	Transfer r7
		ADD r8, r9, r10	4	Dual issue possible on last transfer

## 16.4 Other pipeline-dependent latencies

This section describes a variety of other factors that can affect the timing of a code sequence. For the most part, these factors cannot be accurately predicted on a case by case basis, but can be accounted for statistically if determining the overall timing for a larger section of code.

### 16.4.1 Cycle penalty for instruction flow change

Whenever a control flow change occurs in the processor that the prefetch unit has not predicted, the pipeline must be flushed. This results in a cycle stall equal in number to the length of the integer pipeline. This branch mispredict penalty is 13 cycles. See Chapter 5 *Program Flow Prediction* for details on program execution prediction.

## 16.4.2 Memory system effects on instruction timings

Because the processor is a statically scheduled design, any stall from the memory system can result in the minimum of a 8-cycle delay. This 8-cycle delay minimum is balanced with the minimum number of possible cycles to receive data from the L2 cache in the case of an L1 load miss. Table 16-16 gives the most common cases that can result in an instruction replay because of a memory system stall.

**Table 16-16 Memory system effects on instruction timings**

Replay event	Delay	Description
Load data miss	8 cycles	<ol style="list-style-type: none"> <li>1. A load instruction misses in the L1 data cache.</li> <li>2. A request is then made to the L2 data cache.</li> <li>3. If a miss also occurs in the L2 data cache, then a second replay occurs. The number of stall cycles depends on the external system memory timing. The time required to receive the critical word for an L2 cache miss is 18 core cycles plus the number of cycles required by the external memory system. The minimum number of additional cycles required for the external system is 2 cycles, making the total minimum cycle count 20 cycles. However, 20 cycles are likely to be optimistic because this can only occur in a system with a 1:1 bus ratio and zero wait-state memory.</li> </ol>
Data TLB miss	24 cycles	<ol style="list-style-type: none"> <li>1. A table walk because of a miss in the L1 TLB causes a 24-cycle delay, assuming the translation table entries are found in the L2 cache.</li> <li>2. If the translation table entries are not present in the L2 cache, the number of stall cycles depends on the external system memory timing.</li> </ol>
Store buffer full	8 cycles plus latency to drain fill buffer	<ol style="list-style-type: none"> <li>1. A store instruction miss does not result in any stalls unless the store buffer is full.</li> <li>2. In the case of a full store buffer, the delay is at least eight cycles. The delay can be more if it takes longer to drain some entries from the store buffer.</li> </ol>
Unaligned load or store request	8 cycles	<ol style="list-style-type: none"> <li>1. If a load instruction address is unaligned and the full access is not contained within a 128-bit boundary, there is a 8-cycle penalty.</li> <li>2. If a store instruction address is unaligned and the full access is not contained within a 64-bit boundary, there is a 8-cycle penalty.</li> </ol>

## 16.4.3 Thumb-2 instructions

As a general rule, Thumb-2 instructions are executed with timing constraints identical to their ARM counterparts. However, there are some second order effects to the cycle timing that you must observe. First, the code footprint is smaller, which can reduce the number of instruction cache misses and therefore reduce the cycle count. Second, branch instructions tend to be more densely packed, slightly reducing the branch

prediction accuracy that is achieved and therefore increasing the number of branch mispredictions. Neither of these effects can be accurately measured using hand calculating techniques.

————— **Note** —————

The code footprint and densely packed branch instructions can have an impact on the performance of the processor. In most cases, the interaction of these effects might cancel with each other.

#### 16.4.4 ThumbEE instructions

The majority of the ThumbEE instruction set is identical in both encodings and behavior to the Thumb-2 instruction set and therefore the cycle timings are also identical to the Thumb-2 instruction timings. The behavior of some instructions are different when executed in ThumbEE state instead of in Thumb state. However, the behavior changes for these instructions do not result in any changes to their cycle timing. The only additional cycle timing information for ThumbEE is for the new instructions.

Table 16-17 shows the timing operation of the new ThumbEE instructions.

**Table 16-17 ThumbEE instructions**

Instruction type	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
ENTERX/LEAVE <sup>a</sup>	16	-	-	-	-	-	-
CHKA <sup>b</sup>	1	E2	E2	-	-	-	-
HB <sup>c</sup>	1	-	-	-	-	-	-
HBL <sup>d</sup>	1	-	-	-	-	R14:E3	-
HBP <sup>c</sup>	2	-	-	-	-	R8:E2	-
HBLP <sup>d</sup>	2	-	-	-	-	R8:E2	R14:E3
LDR [R9] <sup>e</sup>	1	R9:E1	-	-	-	Rd:E3	-
LDR [R10] <sup>e</sup>	1	R10:E1	-	-	-	Rd:E3	-
LDR [negative offset] <sup>e</sup>	1	Rn:E1	-	-	-	Rd:E3	-
STR [R9] <sup>f</sup>	1	Rn:E1	Rd:E3	-	-	-	-

- a. This instruction waits for all outstanding instructions to complete and then issues.  
b. If CHKA fails the array bounds check, then an exception is taken. Otherwise, this is a single cycle instruction.  
c. This instruction is predicted and behaves as a direct branch, B instruction.

- d. This instruction is predicted and behaves as a direct branch and link, BL instruction.
- e. Timing is identical to similar load instructions.
- f. Timing is identical to similar store instructions.

### ThumbEE memory check exceptions

All loads and stores in ThumbEE state have the additional functionality of checking the base register for a zero value. If the base register is zero, then the processor performs a branch to the address [HandlerBase – 4]. See the *ARM Architecture Reference Manual* for more information.

The processor handles this scenario in the same way as to an exception such as a data abort because it does not occur in the common case. If the base register is zero, the processor flushes the pipeline and branches to the correct address. The additional cycle time penalty for this is variable in length, but is at least 13 cycles. The CHKA instruction uses the same mechanism when the array bounds check fails. This is also a rare occurrence and therefore is not optimized for performance.

### Predicting ThumbEE branch type instructions

All ThumbEE branch type instructions are predicted in ThumbEE state in the same manner that they are predicted in ARM or Thumb state. In addition, the handler base branch instructions, HB[L][P], are also predicted using the same branch prediction hardware used for direct branch and branch link, B and BL instructions, respectively. Because the ThumbEE instruction set uses R9 as the base register rather than R13 as a stack pointer, LDR and STR instructions that read or write to the PC are written onto the return stack to aid in the prediction of these indirect branches. The usage model of the return stack in ThumbEE state, using R9 as the stack pointer, is identical to the usage model in ARM and Thumb state, using R13 as the stack pointer.

## 16.4.5 Conditional instructions

Because the processor is statically scheduled, it schedules conditional instructions on the basis that they pass their condition codes. This means multi-cycle instructions such as LDM and STM instructions can still complete all their iterations even if they fail their condition codes.

An additional point about conditional instructions is that the destination register of the instruction is treated as an additional source operand. This is done so the old value can be forwarded in the case when the instruction fails the condition codes. This additional source operand is required in the E2 stage of the machine.



## 16.5 Advanced SIMD instruction scheduling

Advanced SIMD instructions flow through the ARM pipeline and then enter the NEON instruction queue between the ARM and NEON pipelines. Although an instruction in the NEON instruction queue is completed from the point of view of the ARM pipeline, the NEON unit must still decode and schedule the instruction. The NEON instruction queue is 16 entries deep. There is also an 12-entry NEON data queue that holds entries for Advanced SIMD load instructions.

As long as these queues are not full, the processor can continue to run and execute both ARM and Advanced SIMD instructions. When the Advanced SIMD instruction or data queue is full, the processor stalls execution of the next Advanced SIMD instruction until there is room for this instruction in the queues. In this manner, the cycle timing of Advanced SIMD instructions scheduled in the NEON engine can affect the overall timing of the instruction sequence, but only if there are enough Advanced SIMD instructions to fill the instruction or data queue.

———— **Note** —————

When the processor is configured without NEON, all attempted Advanced SIMD and VFP instructions result in an Undefined Instruction exception.

### 16.5.1 Mixed ARM and Advanced SIMD instruction sequences

Advanced SIMD instruction scheduling only affects the overall timing sequence if there are enough Advanced SIMD instructions to fill the data or instruction queue. If the majority of instructions in a sequence are Advanced SIMD instructions, then the NEON unit dictates the time required for the sequence. Occasional ARM instructions in the sequence occur in parallel with the Advanced SIMD instructions. If most of the instructions in a sequence are ARM instructions, they dominate the timing of the sequence, and a Advanced SIMD data-processing instruction typically takes one cycle. In hand calculations of cycle timing, you must consider the type of instruction, ARM or Advanced SIMD, that dominates the sequence.

### 16.5.2 Passing data between ARM and NEON

Using MRC instructions to pass data from NEON to ARM takes a minimum of 20 cycles. The data transfers from the NEON register file at the back of the NEON pipeline to the ARM register file at the beginning of the ARM pipeline. You can hide some or all of this latency by doing multiple back-to-back MRC transfers. The processor continues to issue instructions following a MRC until it encounters an instruction that must read or write the ARM register file. At that point, instruction issue stalls until all pending register transfers from NEON to ARM are complete.

Using MCR instructions to pass data from ARM to NEON does not require any additional cycles. To the NEON unit, the transfers are similar to Advanced SIMD load instructions.

### **16.5.3 Dual issue for Advanced SIMD instructions**

The NEON engine has limited dual issue capabilities. A load/store, permute, MCR, or MRC type instruction can be dual issued with a Advanced SIMD data-processing instruction. A load/store, permute, MCR, or MRC executes in the NEON load/store permute pipeline. An Advanced SIMD data-processing instruction executes in the NEON integer ALU, Shift, MAC, floating-point add or multiply pipelines. This is the only dual issue pairing permitted.

The NEON engine can potentially dual issue on both the first cycle of a multi-cycle instruction (with an older instruction), and on the last cycle of a multi-cycle instruction (with a newer instruction). Intermediate cycles of a multi-cycle instruction cannot be paired and must be single issue.

## 16.6 Instruction-specific scheduling for Advanced SIMD instructions

The tables in this section use the same format presented in *Instruction-specific scheduling for ARM instructions* on page 16-3 and can be interpreted in the same way. The one difference between tables in this section and those in the *Instruction-specific scheduling for ARM instructions* on page 16-3 is that the execution pipeline consists of stages N1-N6 instead of E1-E5.

Advanced SIMD data-processing instructions are divided into the following subcategories:

- integer ALU instructions
- integer multiply instructions
- integer shift instructions
- floating-point add instructions
- floating-point multiply instructions.

Advanced SIMD load/store permute instructions are divided into the following subcategories:

- byte permute instructions
- load/store instructions
- register transfer instructions.

---

### Note

---

This document uses the older assembler language instruction mnemonics. See Appendix B *Instruction Mnemonics* for information about the *Unified Assembler Language* (UAL) equivalents of the Advanced SIMD instruction mnemonics. See the *ARM Architecture Reference Manual* for more information on the UAL syntax.

---

### 16.6.1 Example of how to read Advanced SIMD instruction tables

This section provides examples of how to read Advanced SIMD instruction tables described in the chapter. See the *ARM Architecture Reference Manual* for assembly syntax of instructions.

In these Advanced SIMD instruction tables, Q<n>Lo maps to D<2n> and Q<n>Hi maps to D<2n+1>.

Example 16-4 on page 16-24 shows how to read an Advanced SIMD integer ALU instruction from Table 16-18 on page 16-25.

---

**Example 16-4 Advanced SIMD integer ALU instruction**


---

VADD.I32.S16 Q2, D1, D2

This is an integer Advanced SIMD vector and long instruction. Source1, in this case D1, and Source2, in this case D2, are both required in N1. The result, stored in Q2 for this case, is available in N3 for the next subsequent instruction that requires this register as a source operand. Assuming no data hazards, the instruction takes a minimum of one cycle to execute as indicated by the value in the Cycles column.

---

Example 16-5 shows how to read an Advanced SIMD floating-point multiply instruction from Table 16-21 on page 16-32.

---

**Example 16-5 Advanced SIMD floating-point multiply instruction**


---

VMUL.F32 Q0, Q1, D4[0]

This is a floating-point Advanced SIMD vector multiply by scalar instruction. It is a multi-cycle instruction that has source operand requirements in both the first and second cycles. In the first cycle, Source1, in this case Q1Lo or D2, is required in N2. Source2, in this case D4, is required in N1. In the second cycle, Source1, in this case Q1Hi or D3, is required in N2. The result of the multiply, stored in Q0 for this case, is available in N5 for the next subsequent instruction that requires this register as a source operand. The low half of the result, Q0Lo or D0, is calculated in the first cycle. The high half of the result, Q0Hi or D1, is calculated in the second cycle. Assuming no data hazards, the instruction takes a minimum of two cycles to execute as indicated by the value in the Cycles column.

---

## 16.6.2 Advanced SIMD integer ALU instructions

Table 16-18 shows the operation of the Advanced SIMD integer ALU instructions.

**Table 16-18 Advanced SIMD integer ALU instructions**

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VADD	Dd, Dn, Dm	1	Dn:N2	Dm:N2	-	-	Dd:N3	-
VAND	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	QdLo:N3	QdHi:N3
VORR								
VEOR								
VBIC								
VORN								
VSUB	Dd, Dn, Dm	1	Dn:N2	Dm:N1	-	-	Dd:N3	-
	Qd, Qn, Qm	1	QnLo:N2	QmLo:N1	QnHi:N2	QmHi:N1	QdLo:N3	QdHi:N3
VADD	Qd, Dn, Dm (long)	1	Dn:N1	Dm:N1	-	-	QdLo:N3	QdHi:N3
VSUB								
	Qd, Qn, Dm (wide)	1	QnLo:N2	Dm:N1	QnHi:N2	-	QdLo:N3	QdHi:N3
VHADD	Dd, Dn, Dm	1	Dn:N2	Dm:N2	-	-	Dd:N4	-
VRHADD	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	QdLo:N4	QdHi:N4
VQADD								
VTST								
VADH	Dd, Qn, Qm (highhalf)	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	Dd:N4	-
VRADH								
VSBH	Dd, Qn, Qm (highhalf)	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N1	Dd:N4	-
VRSBH								

**Table 16-18 Advanced SIMD integer ALU instructions (continued)**

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VHSUB	Dd, Dn, Dm	1	Dn:N2	Dm:N1	-	-	Dd:N4	-
VQSUB	Qd, Qn, Qm	1	QnLo:N2	QmLo:N1	QnHi:N2	QmHi:N1	QdLo:N4	QdHi:N4
VABD								
VCEQ								
VCGE								
VCGT								
VMAX								
VMIN								
VFMX <sup>a</sup>								
VFMN <sup>a</sup>								
VNEG	Dd, Dm	1	-	Dm:N1	-	-	Dd:N3	-
	Qd, Qm	1	-	QmLo:N1	-	QmHi:N1	QdLo:N3	QdHi:N3
VQNEG	Dd, Dm	1	-	Dm:N1	-	-	Dd:N4	-
VQABS	Qd, Qm	1	-	QmLo:N1	-	QmHi:N1	QdLo:N4	QdHi:N4
VABD	Qd, Dn, Dm (long)	1	Dn:N2	Dm:N1	-	-	QdLo:N4	QdHi:N4
VABS	Dd, Dm	1	Dm:N2	-	-	-	Dd:N4	-
VCEQZ	Qd, Qm	1	QmLo:N2	-	QmHi:N2	-	QdLo:N4	QdHi:N4
VCGEZ								
VCGTZ								
VCLEZ								
VCLTZ								
VSUM	Dd, Dn, Dm	1	Dn:N1	Dm:N1	-	-	Dd:N3	-
	Dd, Dm (long)	1	Dm:N1	-	-	-	Dd:N3	-
	Qd, Qm (long)	1	QmLo:N1	QmHi:N1	-	-	QdLo:N3	QdHi:N3

Table 16-18 Advanced SIMD integer ALU instructions (continued)

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VNOT	Dd, Dm	1	-	Dm:N2	-	-	Dd:N3	-
VCLS								
VCLZ								
VCNT								
VNOT	Qd, Qm	1	-	QmLo:N2	-	QmHi:N2	QdLo:N3	QdHi:N3
VCLS	Qd, Qm	1	-	QmLo:N2	-	-	QdLo:N3	-
VCLZ		2	-	QmHi:N2	-	-	QdHi:N3	-
VCNT								
VMOV	Dd, #IMM	1	-	-	-	-	Dd:N3	-
VMVN	Qd, #IMM	1	-	-	-	-	QdLo:N3	QdHi:N3
VORR	Dd, #IMM	1	Dd:N2	-	-	-	Dd:N3	-
VBIC	Qd, #IMM	1	QdLo:N2	-	Qdb:N2	-	QdLo:N3	QdHi:N3
VBIT	Dd, Dn, Dm	1	Dn:N2	Dm:N2	Dd:N2	-	Dd:N3	-
VBIF	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	QdLo:N2	-	QdLo:N3	-
VBSL		2	QnHi:N2	QmHi:N2	QdHi:N2	-	QdHi:N3	-
VABA	Dd, Dn, Dm	1	Dn:N2	Dm:N1	Dd:N3	-	Dd:N6	-
	Qd, Qn, Qm	1	QnLo:N2	QmLo:N1	QdLo:N3	-	QdLo:N6	-
		2	QnHi:N2	QmHi:N1	QdHi:N3	-	QdHi:N6	-
	Qd, Dn, Dm (long)	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6
VSMA	Dd, Dm (long)	1	Dm:N1	-	Dd:N3	-	Dd:N6	-
	Qd, Qm (long)	1	QmLo:N1	QmHi:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6

a. VFMX and VFMN exist only for the Dd, Dn, Dm variant.

### 16.6.3 Advanced SIMD integer multiply instructions

Table 16-19 shows the operation of the Advanced SIMD integer multiply instructions.

**Table 16-19 Advanced SIMD integer multiply instructions**

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VMUL VQDMLH VQRDMLH	Dd, Dn, Dm (.8 normal) (.16 normal)	1	Dn:N2	Dm:N2	Dm:N2	-	Dd:N6	-
	Qd, Qn, Qm (.8 normal) (.16 normal)	1 2	QnLo:N2 QnHi:N2	QmLo:N2 QmHi:N2	- -	- -	QdLo:N6 QdHi:N6	- -
	Dd, Dn, Dm (.32 normal)	1 2	Dn:N2 -	Dm:N1 -	- -	- -	- Dd:N6	- -
	Qd, Qn, Qm (.32 normal)	1 2 3 4	QnLo:N2 - QnHi:N2 -	QmLo:N1 - QmHi:N1 -	- - - -	- - - -	- QdLo:N6 - QdHi:N6	- - - -
VMUL VQDMUL	Qd, Dn, Dm (.16.8 long) (.32.16 long)	1	Dn:N2	Dm:N2	-	-	QdLo:N6	QdHi:N6
	Qd, Dn, Dm (.64.32 long)	1 2	Dn:N2 -	Dm:N1 -	- -	- -	- QdLo:N6	- QdHi:N6



Table 16-19 Advanced SIMD integer multiply instructions (continued)

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VMLA <sup>a</sup>	Dd, Dn, Dm	1	Dn:N2	Dm:N2	Dd:N3	-	Dd:N6	-
VMLS <sup>a</sup>	(.8 normal)							
	(.16 normal)							
	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	QdLo:N3	-	QdLo:N6	-
	(.8 normal)	2	QnHi:N2	QmHi:N2	QdHi:N3	-	QdHi:N6	-
	(.16 normal)							
	Dd, Dn, Dm	1	Dn:N2	Dm:N1	Dd:N3	-	-	-
	(.32 normal)	2	-	-	-	-	Dd:N6	-
	Qd, Qn, Qm	1	QnLo:N2	QmLo:N1	QdLo:N3	-	-	-
	(.32 normal)	2	-	-	-	-	QdLo:N6	-
		3	QnHi:N2	QmHi:N1	QdHi:N3	-	-	-
		4	-	-	-	-	QdHi:N6	-
VMLA <sup>a</sup>	Qd, Dn, Dm	1	Dn:N2	Dm:N2	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6
VMLS <sup>a</sup>	(.16.8 long)							
VQDMLA <sup>a</sup>	(.32.16 long)							
VQDMLS <sup>a</sup>	Qd, Dn, Dm	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	-	-
	(.64.32 long)	2	-	-	-	-	QdLo:N6	QdHi:N6
VMUL	Dd, Dn, Dm[x]	1	Dn:N2	Dm:N1	-	-	Dd:N6	-
VQDMLH	(.16 scalar)							
VQRDMLH	Qd, Qn, Dm[x]	1	QnLo:N2	Dm:N1	-	-	QdLo:N6	-
	(.16 scalar)	2	QnHi:N2				QdHi:N6	
	Dd, Dn, Dm[x]	1	Dn:N2	Dm:N1	-	-	-	-
	(.32 scalar)	2	-	-	-	-	Dd:N6	-
	Qd, Qn, Dm[x]	1	QnLo:N2	Dm:N1	-	-	-	-
	(.32 scalar)	2	-	-	-	-	QdLo:N6	-
		3	Qnhi:N2	-	-	-	-	-
		4	-	-	-	-	QdHi:N6	-

Table 16-19 Advanced SIMD integer multiply instructions (continued)

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VMUL	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	-	-	QdLo:N6	QdHi:N6
VQDMUL	(.32.16 long scalar)							
	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	-	-	-	-
	(.64.32 long scalar)	2	-	-	-	-	QdLo:N6	QdHi:N6
VMLA <sup>a</sup>	Dd,Dn,Dm[x]	1	Dn:N2	Dm:N1	Dd:N3	-	Dd:N6	-
VMLS <sup>a</sup>	(.16 scalar)							
	Qd,Qn,Dm[x]	1	QnLo:N2	Dm:N1	QdLo:N3	-	QdLo:N6	-
	(.16 scalar)	2	QnHi:N2	-	QdHi:N3	-	QdHi:N6	-
	Dd,Dn,Dm[x]	1	Dn:N2	Dm:N1	Dd:N3	-	-	-
	(.32 scalar)	2	-	-	-	-	Dd:N6	-
	Qd,Qn,Dm[x]	1	QnLo:N2	Dm:N1	QdLo:N3	-	-	-
	(.32 scalar)	2	-	-	-	-	QdLo:N6	-
		3	QnHi:N2	-	QdHi:N3	-	-	-
		4	-	-	-	-	QdHi:N6	-
VMLA <sup>a</sup>	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6
VMLS <sup>a</sup>	(.32.16 long scalar)							
VQDMLA <sup>a</sup>	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	-	-
VQDMLS <sup>a</sup>	(.64.32 long scalar)	2	-	-	-	-	QdLo:N6	QdHi:N6

a.

If a multiply-accumulate follows a multiply or another multiply-accumulate, and depends on the result of that first instruction, then if the dependency between both instructions are of the same type and size, the processor uses a special multiplier accumulator forwarding. This special forwarding means the multiply instructions can issue back-to-back because the result of the first instruction in N5 is forwarded to the accumulator of the second instruction in N4. If the size and type of the instructions do not match, then Dd or Qd is required in N3. This applies to combinations of the multiply-accumulate instructions VMLA, VMLS, VQDMLA, and VQDMLS, and the multiply instructions VMUL and VQDMUL.

## 16.6.4 Advanced SIMD integer shift instructions

Table 16-20 shows the operation of the Advanced SIMD integer shift instructions.

**Table 16-20 Advanced SIMD integer shift instructions**

Instruction	Register format	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
VSHR	Dd, Dm, #IMM	1	Dm:N1	-	-	-	Dd:N3	-
VSHL	Qd, Qm, #IMM	1	QmLo:N1	QmHi:N1	-	-	QdLo:N3	QdHi:N3
VQSHL	Dd, Dm, #IMM	1	Dm:N1	-	-	-	Dd:N4	-
VRSHR	Qd, Qm, #IMM	1	QmLo:N1	QmHi:N1	-	-	QdLo:N4	QdHi:N4
VSHR	Dd, Qm, #IMM (narrow)	1	QmLo:N1	QmHi:N1	-	-	Dd:N3	-
VQSHR	Dd, Qm, #IMM (narrow)	1	QmLo:N1	QmHi:N1	-	-	Dd:N4	-
VQMOV								
VRSHR								
VQRSHR								
VSHL <sup>a</sup>	Qd, Dm, [#IMM] (long, wide)		Dm:N1	-	-	-	QdLo:N3	QdHi:N3
VMVH								
VSLI	Dd, Dm, #IMM	1	Dm:N1	Dd:N1	-	-	Dd:N3	-
VSRI	Qd, Qm, #IMM	1	QmLo:N1	QdLo:N1	-	-	QdLo:N3	-
		2	QmHi:N1	QdHi:N1	-	-	QdHi:N3	-
VSHL	Dd, Dm, Dn	1	Dm:N1	Dn:N1	-	-	Dn:N1	-
	Qd, Qm, Qn	1	QmLo:N1	QnLo:N1	-	-	QdLo:N3	-
		2	QmHi:N1	QnHi:N1	-	-	QdHi:N3	-
VQSHL	Dd, Dm, Dn	1	Dm:N1	Dn:N1	-	-	Dd:N4	-
VRSHL	Qd, Qm, Qn	1	QmLo:N1	QnLo:N1	-	-	QdLo:N4	-
VQRSHL		2	QmLo:N1	QnHi:N1	-	-	QdHi:N4	-
VSRA	Dd, Dm, #IMM	1	Dm:N1	-	Dd:N3	-	Dd:N6	-
VRSRA	Qd, Qm, #IMM	1	QmLo:N1	QmHi:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6

a. Only VSHL has the #IMM parameter.

## 16.6.5 Advanced SIMD floating-point instructions

Table 16-21 shows the operation of the Advanced SIMD floating-point instructions.

**Table 16-21 Advanced SIMD floating-point instructions**

Instruction	Register format	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
VADD	Dd, Dn, Dm	1	Dn:N2	Dm:N2	-	-	Dd:N5	-
VSUB	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	-	-	QdLo:N5	-
VABD		2	QnHi:N2	QmHi:N2	-	-	QdHi:N5	-
VMUL								
VCEQ								
VCGE								
VCGT								
VCAGE								
VCAGT								
VMAX								
VMIN								
VABS	Dd, Dm	1	Dm:N2	-	-	-	Dd:N5	-
VNEG	Qd, Qm	1	QmLo:N2	-	-	-	QdLo:N5	-
VCEQZ		2	QmHi:N2	-	-	-	QdHi:N5	-
VCGEZ								
VCGTZ								
VCLEZ								
VCLTZ								
VRECPE								
VRSQRTE								
VCVT								
VSUM	Dd, Dn, Dm	1	Dn:N1	Dm:N1	-	-	Dd:N5	-
VFMX								
VPMN								
VMUL	Dd, Dn, Dm[x] (scalar)	1	Dn:N2	Dm:N1	-	-	Dd:N5	-
	Qd, Qn, Dm[x] (scalar)	1	QnLo:N2	Dm:N1	-	-	QdLo:N5	-
		2	QnHi:N2	-	-	-	QdHi:N5	-

Table 16-21 Advanced SIMD floating-point instructions (continued)

Instruction	Register format	Cycles	Source1	Source2	Source3	Source4	Result1	Result2
VMLA <sup>a</sup>	Dd, Dn, Dm	1	Dn:N2	Dm:N2	Dd:N3	-	Dd:N9	-
VMLS <sup>a</sup>	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	QdLo:N3	-	QdLo:N9	-
		2	QnHi:N2	QmHi:N2	QdHi:N3	-	QdHi:N9	-
	Dd, Dn, Dm[x] (scalar)	1	Dn:N2	Dm:N1	Dd:N3	-	Dd:N9	-
		2	QnLo:N2	-	QdHi:N3	-	QdHi:N9	-
VRECPS <sup>a</sup>	Dd, Dn, Dm	1	Dn:N2	Dm:N2	-	-	Dd:N9	-
VRSQRTS <sup>a</sup>	Qd, Qn, Qm	1	QnLo:N2	QmLo:N2	-	-	QdLo:N9	-
		2	QnHi:N2	QmHi:N2	-	-	QdHi:N9	-

- a. The VMLA.F, VMLS.F, VRECPS.F, VRSQRTS.F instructions begin execution on the floating-point multiply pipeline. The floating-point multiply result is then forwarded to the floating-point add pipeline to complete the accumulate portion of the instructions. Therefore, these instructions are pipelined across ten stages, N1 through N10, where N10 is the writeback stage.

———— **Note** —————

The VMLA.F and VMLS.F type instructions have additional restrictions that determine when they can be issued:

- If a VMLA.F is followed by a VMLA.F with no RAW hazard, the second VFMLA.F will issue with no stalls.
- If a VMLA.F is followed by an VADD.F or VMUL.F with no RAW hazard, the VADD.F or VMUL.F will stall 4 cycles before issue. The 4 cycle stall preserves the in-order retirement of the instructions.
- A VMLA.F followed by any NEON floating-point instruction with RAW hazard will stall for 8 cycles.

## 16.6.6 Advanced SIMD byte permute instructions

Table 16-22 shows the operation of the Advanced SIMD byte permute instructions.

**Table 16-22 Advanced SIMD byte permute instructions**

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VMOV	Dd, Qm (narrow)	1	QmLo:N1	QmHi: N1	-	-	Dd:N2	-
VMOV	Dd, Dm[x] (scalar)	1	Dm:N1	-	-	-	Dd:N2	-
	Qd, Dm[x] (scalar)	1	Dm:N1	-	-	-	QdLo:N2	QdHi:N2
VTRN	Dd, Dm	1	Dd:N1	Dm:N1	-	-	Dd:N2	Dm:N2
VSWP	Qd, Qm	1	QdLo:N1	QmLo:N1	-	-	QdLo:N2	QmLo:N2
		2	QdHi:N1	QmHi:N1	-	-	QdHi:N2	QmHi:N2
VZIP	Dd, Dm	1	Dd:N1	Dm:N1	-	-	Dd:N2	Dm:N2
		1	QdLo:N1	QmLo:N1	-	-	-	-
		2	QdHi:N1	QmHi:N1	-	-	QdLo:N2	QdHi:N2
		3	-	-	-	QmLo:N2	QmHi:N2	
VUZP	Dd, Dm	1	Dd:N1	Dm:N1	-	-	Dd:N2	Dm:N2
		1	QdLo:N1	QdHi:N1	-	-	-	-
		2	QmLo:N1	QmHi:N1	-	-	QdLo:N2	QmLo:N2
		3	-	-	-	QdHi:N2	QmHi:N2	
VREV	Dd, Dm	1	Dm:N1	-	-	-	Dd:N2	-
			QmLo:N1	QmHi:N1	-	-	QdLo:N2	QdHi:N2
VEXT	Dd, Dn, Dm, #IMM	1	Dn:N1	Dm:N1	-	-	Dd:N2	-
			QnLo:N1	QnHi:N1	-	-	-	-
		2	QmLo:N1	QmHi:N1	-	-	QdLo:N2	QdHi:N2

Table 16-22 Advanced SIMD byte permute instructions (continued)

Instruction	Register format	Cycles	Source				Result		
			1	2	3	4	1	2	
VTBL	Dd, {Dn}, Dm	1	-	Dm:N1	-	-	-	-	
		2	Dn:N1	-	-	-	Dd:N2	-	
	Dd, {Dn, Dn1}, Dm	1	-	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	Dd:N2	-	
	Dd, {Dn, Dn1, Dn2}, Dm	1	-	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	-	-	
		3	Dn2:N1	-	-	-	Dd:N2	-	
	Dd, {Dn, Dn1, Dn2, Dn3}, Dm	1	-	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	-	-	
		3	Dn2:N1	Dn3:N1	-	-	Dd:N2	-	
	VTBX	Dd, {Dn}, Dm	1	Dd:N1	Dm:N1	-	-	-	-
			2	Dn:N1	-	-	-	Dd:N2	-
Dd, {Dn, Dn1}, Dm		1	Dd:N1	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	Dd:N2	-	
Dd, {Dn, Dn1, Dn2}, Dm		1	Dd:N1	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	-	-	
		3	Dn2:N1	-	-	-	Dd:N2	-	
Dd, {Dn, Dn1, Dn2, Dn3}, Dm		1	Dd:N1	Dm:N1	-	-	-	-	
		2	Dn:N1	Dn1:N1	-	-	-	-	
		3	Dn2:N1	Dn3:N1	-	-	Dd:N2	-	

### 16.6.7 Advanced SIMD load/store instructions

Advanced SIMD load/store instructions can be divided into the following subcategories:

- VLDR and VSTR register load/store single
- VLDM and VSTM register load/store multiple
- VLD and VST multiple 1-element or 2, 3, 4-element structure
- VLD and VST single 1-element or 2, 3, 4-element structure to one lane
- VLD single 1-element or 2, 3, 4-element structure to all lanes.

VLDR and VSTR instructions transfer a single 64-bit register and require two issue cycles. Processor scheduling is static, and it is not possible to know the address alignment at schedule time. Therefore, scheduling of the VLDR and VSTR instructions must be done assuming the load/store address is not 128-bit aligned.

VLDM and VSTM instructions transfer multiple 64-bit registers. The number of registers in the register list determines the number of cycles required to execute a load or store multiple. The NEON unit can load or store two 64-bit registers in each cycle. The number of cycles required to execute a VLDM or VSTM instruction is given by the following formula:

$$(\text{number of registers}/2) + \text{mod}(\text{number of registers}, 2) + 1$$

For example, VLDM and VSTM transfer of one or two registers require two cycles, three or four registers require three cycles, five or six registers require four cycles, and 15 or 16 registers require nine cycles.

VLD and VST element and structure load/store instructions transfer one up to four 64-bit registers. The number of cycles required to execute a VLD or VST instruction depends on both the number of registers in the register list and the alignment requirement. Typically, you can reduce the number of cycles if you use a stronger alignment. For example, a 2-register VLD2.16@64 requires two cycles but VLD2.16@128 requires only one cycle.

Table 16-23 shows the operation of the Advanced SIMD load/store instructions.

**Table 16-23 Advanced SIMD load/store instructions**

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VLDR and VSTR register load/store <sup>a</sup> :					
VLDR	Dd, <addr>	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N1	-
VSTR	Dd, <addr>	1	Dd:N1	-	-	-	-	-
		2	-	-	-	-	-	-



Table 16-23 Advanced SIMD load/store instructions (continued)

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VLD and VST multiple 1-element or 2, 3, 4-element structure <sup>b</sup> :					
VLD1	1-reg (unaligned)	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N1	-
	1-reg (@64)	1	-	-	-	-	Dd:N1	-
		2	-	-	-	-	Dd:N1	Dd+1:N1
	2-reg (@128)	1	-	-	-	-	Dd:N1	Dd+1:N1
		2	-	-	-	-	Dd:N1	Dd+1:N1
	3-reg (unaligned, @64)	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N1	Dd+1:N1
		3	-	-	-	-	Dd+2:N1	-
	4-reg (unaligned, @64)	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N1	Dd+1:N1
		3	-	-	-	-	Dd+2:N1	Dd+3:N1
	4-reg (@128, @256)	1	-	-	-	-	Dd:N1	Dd+1:N1
		2	-	-	-	-	Dd+2:N1	Dd+3:N1
VLD2	2-reg (unaligned, @64)	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N2	Dd+1:N2
	2-reg (@128)	1	-	-	-	-	Dd:N2	Dd+1:N2
		2	-	-	-	-	Dd:N2	Dd+1:N2
	4-reg (unaligned, @64)	1	-	-	-	-	-	-
		2	-	-	-	-	Dd:N2	Dd+2:N2
		3	-	-	-	-	Dd+1:N2	Dd+3:N2
	4-reg (@128, @256)	1	-	-	-	-	Dd:N2	Dd+2:N2
		2	-	-	-	-	Dd+1:N2	Dd+3:N2

**Table 16-23 Advanced SIMD load/store instructions (continued)**

Instruction	Register list (alignment)	Cycles	Source				Result		
			1	2	3	4	1	2	
VLD3	3-reg (unaligned, @64)	1	-	-	-	-	-	-	
		2	-	-	-	-	-	-	
		3	-	-	-	-	Dd:N2	Dd+1:N2	
		4	-	-	-	-	Dd+2:N2	-	
VLD4	4-reg (unaligned, @64)	1	-	-	-	-	-	-	
		2	-	-	-	-	-	-	
		3	-	-	-	-	Dd:N2	Dd+1:N2	
		4	-	-	-	-	Dd+2:N2	Dd+3:N2	
	4-reg (@128, @256)	1	-	-	-	-	-	-	
		2	-	-	-	-	Dd:N2	Dd+1:N2	
		3	-	-	-	-	Dd+2:N2	Dd+3:N2	
VST1	1-reg (unaligned)	1	Dd:N1	-	-	-	-	-	
		2	-	-	-	-	-	-	
	1-reg (@64)	1	Dd:N1	-	-	-	-	-	
	2-reg (unaligned, @64)	1	Dd:N1	Dd+1:N1	-	-	--	--	
		2	-	-	-	-			
	2-reg (@128)	1	Dd:N1	Dd+1:N1	-	-	-	-	
	3-reg (unaligned)	1	Dd:N1	Dd+1:N1	-	-	-	-	
		2	Dd+2:N1	-	-	-	-	-	
		3	-	-	-	-	-	-	
	3-reg (@64)	1	Dd:N1	Dd+1:N1	-	-	-	-	
		2	Dd+2:N1	-	-	-	-	-	
4-reg (unaligned, @64)	1	Dd:N1	Dd+1:N1	-	-	-	-		
	2	Dd+2:N1	Dd+3:N1	-	-	-	-		
	3	-	-	-	-	-	-		
4-reg (@128, @256)	1	Dd:N1	Dd+1:N1	-	-	-	-		
	2	Dd+2:N1	Dd+3:N1	-	-	-	-		

Table 16-23 Advanced SIMD load/store instructions (continued)

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VST2	2-reg (unaligned, @64)	1	Dd:N1	Dd+1:N1	-
		2	-	-	-	-	-	-
	2-reg (@128)	1	Dd:N1	Dd+1:N1	-	-	-	-
	4-reg (unaligned, @64)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	-	-
		4	-	-	-	-	-	-
	4-reg (@128, @256)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	-	-
VST3	3-reg (unaligned)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	-	-	-	-	-
		3	-	-	-	-	-	-
		4	-	-	-	-	-	-
	3-reg (@64)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	-	-	-	-	-
		3	-	-	-	-	-	-
VST4	4-reg (unaligned, @64)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	-	-
		4	-	-	-	-	-	-
	4-reg (@128, @256)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	-	-

**Table 16-23 Advanced SIMD load/store instructions (continued)**

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VLD and VST single 1-element or 2, 3, 4-element structure to one lane <sup>c</sup> :					
VLD1	1-reg (.8 unaligned, .16@16, .32@32)	1	Dd:N1	-	-	-	-	-
		2	-	-	-	-	Dd:N2	-
	1-reg (.16 unaligned, .32 unaligned)	1	Dd:N1	-	-	-	-	-
		2	-	-	-	-	-	-
		3	-	-	-	-	Dd:N2	-
	VLD2	2-reg (unaligned)	1	Dd:N1	Dd+1:N1	-	-	-
2			-	-	-	-	-	-
3			-	-	-	-	Dd:N2	Dd+1:N2
2-reg (.8@16, .16@32, .32@64)		1	Dd:N1	Dd+1:N1	-	-	-	-
		2	-	-	-	-	Dd:N2	Dd+1:N2
VLD3	3-reg (unaligned)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	-	-	-	-	-
		3	-	-	-	-	-	-
		4	-	-	-	-	Dd:N2	Dd+1:N2
		5	-	-	-	-	Dd+2:N2	-
VLD4	4-reg (unaligned, .32@64)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	-	-
		4	-	-	-	-	Dd:N2	Dd+1:N2
		5	-	-	-	-	Dd+2:N2	Dd+3:N1
	4-reg (.8@32, .16@64, .32@128)	1	Dd:N1	Dd+1:N1	-	-	-	-
		2	Dd+2:N1	Dd+3:N1	-	-	-	-
		3	-	-	-	-	Dd:N2	Dd+1:N2
		4	-	-	-	-	Dd+2:N2	Dd+3:N2

Table 16-23 Advanced SIMD load/store instructions (continued)

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VST1	1-reg (.16 unaligned, .32 unaligned)	1 2	Dd:N1 -	- -	- -
	1-reg (.8 unaligned, .16@16, .32@32)	1	Dd:N1	-	-	-	-	
VST2	2-reg (unaligned)	1 2	Dd:N1 -	Dd+1:N1 -	- -	- -	- -	
	2-reg (.8@16, .16@32, .32@64)	1	Dd:N1	Dd+1:N1	-	-	-	
VST3	3-reg (unaligned)	1 2 3	Dd:N1 Dd+2:N1 -	Dd+1:N1 - -	- - -	- - -	- - -	
VST4	4-reg (unaligned, .32@64)	1 2 3	Dd:N1 Dd+2:N1 -	Dd+1:N1 Dd+3:N1 -	- - -	- - -	- - -	
	4-reg (.8@32, .16@64, .32@128)	1 2	Dd:N1 Dd+2:N1	Dd+1:N1 Dd+3:N1	- -	- -	- -	

Table 16-23 Advanced SIMD load/store instructions (continued)

Instruction	Register list (alignment)	Cycles	Source				Result	
			1	2	3	4	1	2
			VLD single 1-element or 2, 3, 4-element structure to all lanes <sup>b</sup> :					
VLD1	1-reg	1	-	-	-	-	-	-
	(.16 unaligned, .32 unaligned)	2	-	-	-	-	Dd:N2	-
	1-reg	1	-	-	-	-	Dd:N2	-
	(.8 unaligned, .16@16, .32@32)							
	2-reg	1	-	-	-	-	-	-
(.16 unaligned, .32 unaligned)	2	-	-	-	-	Dd:N2	Dd+1:N2	
2-reg	1	-	-	-	-	Dd:N2	Dd+1:N2	
(.8 unaligned, .16@16, .32@32)								
VLD2	2-reg	1	-	-	-	-	-	-
	(unaligned)	2	-	-	-	-	Dd:N2	Dd+1:N2
	2-reg	-	-	-	-	-	Dd:N2	Dd+1:N2
(.8@16, .16@32, .32@64)								
VLD3	3-reg	1	-	-	-	-	-	-
	(unaligned)	2	-	-	-	-	Dd:N2	Dd+1:N2
		3	-	-	-	-	Dd+2:N2	-
VLD4	4-reg	1	-	-	-	-	-	-
	(unaligned, .32@64)	2	-	-	-	-	Dd:N2	Dd+1:N2
		3	-	-	-	-	Dd+2:N2	Dd+3:N2
	4-reg	1	-	-	-	-	Dd:N2	Dd+1:N2
	(.8@32, .16@64, .32@128)	2	-	-	-	-	Dd+2:N2	Dd+3:N2

a. This table lists the VLDR instruction scheduling for little-endian mode. For VLDR in big-endian mode, results are available in N2 and not N1.

- b. This table lists the VLD instruction scheduling for little-endian mode. For VLD1 multiple 1-element in big-endian mode, results are available in N2 and not N1. For VLD2, VLD3, VLD4 results are available in N2 regardless of the endianness configuration. This table lists only the single-spaced register transfer variants. For single-spaced register transfer variants, the source and destination registers are Dd, Dd+1, Dd+2, and Dd+3. For double-spaced register transfer variants, the source and destination registers are Dd, Dd+2, Dd+4, and Dd+6.
- c. This table lists only the single-spaced register transfer variants. For single-spaced register transfer variants, the source and destination registers are Dd, Dd+1, Dd+2, and Dd+3. For double-spaced register transfer variants, the source and destination registers are Dd, Dd+2, Dd+4, and Dd+6.

## 16.6.8 Advanced SIMD register transfer instructions

Table 16-24 shows the operation of the Advanced SIMD register transfer instructions.

**Table 16-24 Advanced SIMD register transfer instructions**

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VMOV <sup>a</sup> (MCR/MCRR)	Dn, Rd	-	-	-	-	-	Dn:N2	-
	Qn, Rd	-	-	-	-	-	QnLo:N2	QnHi:N2
	Dn[], Rd	1	Dn:N1	-	-	-	-	-
		2	-	-	-	-	Dd:N2	-
Dm, Rd, Rn	1	-	-	-	-	-	-	
	2	-	-	-	-	Dm:N2	-	
VMOV <sup>b</sup> (MRC/MRRC)	Rd, Dn[]	-	Dn:N1	-	-	-	-	-
	Rd, Rn, Dm	1	Dm:N1	-	-	-	-	-
		2	-	-	-	-	-	-

a. MCRR instruction is scheduled as two back-to-back MCR instructions.

b. MRRC instruction is scheduled as two back-to-back MRC instructions.

## 16.7 VFP instructions

All VFP data-processing instructions can execute on the VFP coprocessor. A special subset of the VFP instructions can execute on the *NEON Floating-Point* (NFP) pipeline.

---

**Note**

This document uses the older assembler language instruction mnemonics. See Appendix B *Instruction Mnemonics* for information about the *Unified Assembler Language* (UAL) equivalents of the VFP data-processing instruction mnemonics. See the *ARM Architecture Reference Manual* for more information on the UAL syntax.

---

### 16.7.1 VFP instruction execution in the VFP coprocessor

The VFP coprocessor is a nonpipelined floating-point execution engine that can execute any VFPv3 data-processing instruction. Each instruction runs to completion before the next instruction can issue, and there is no forwarding of VFP results to other instructions. Two cycles of decode, stages M2 and M3, are required between consecutive VFP instructions. These decode cycles are included in the cycle timing of this section.

The number of cycles required to complete an instruction depends on both the instruction and the input data operands. Floating-point operands can be divided into three broad categories:

- normal
- subnormal
- special.

Most numbers are normal and have an internal format that consists of a sign, a fractional number between one and two, and an exponent. Subnormal numbers are too small to represent in the normal space. A subnormal number consists of a sign, a fractional number between zero and one, and a zero in the exponent field. Special numbers are zeros, NaNs, and infinities.



Table 16-25 shows the range of cycle times for VFPv3 data-processing instruction with normal numbers. Subnormal numbers usually take more time as the Subnormal penalty column in Table 16-25 shows. Special numbers are handled by separate logic, and usually take less time than what is indicated in this table.

**Table 16-25 VFP Instruction cycle counts**

<b>Instruction</b>	<b>Single precision cycles</b>	<b>Double precision cycles</b>	<b>Subnormal penalty</b>
FADD	9-10	9-10	operand/result
FSUB	9-10	9-10	operand/result
FMUL	10-12	11-17	operand/result
FNMUL	10-12	11-17	operand/result
FMAC	18-21	19-26	operand/intermediate/result
FNMAC	18-21	19-26	operand/intermediate/result
FMSC	18-21	19-26	operand/intermediate/result
FNMSC	18-21	19-26	operand/intermediate/result
FDIV	20-37	29-65	operand/result
FSQRT	19-33	29-60	operand
FCONST	4	4	none
FABS	4	4	none
FCPY	4	4	none
FNEG	4	4	none
FCMP	4 or 7	4 or 7	none
FCMPE	4 or 7	4 or 7	none
FCMPZ	4 or 7	4 or 7	none
FCMPEZ	4 or 7	4 or 7	none
FCVTDS	5	-	operand
FCVTSD	-	7	intermediate
FSITO	9	9	none

Table 16-25 VFP Instruction cycle counts (continued)

Instruction	Single precision cycles	Double precision cycles	Subnormal penalty
FUITO	9	9	none
FTOSI	8	8	none
FTOUI	8	8	none
FTOSIZ	8	8	none
FTOUIZ	8	8	none
FSHTO	9	9	none
FUHTO	9	9	none
FSLTO	9	9	none
FULTO	9	9	none
FTOSH	6	8	none
FTOUH	6	8	none
FTOSL	6	8	none
FTOUL	6	8	none

The Instruction column of Table 16-25 on page 16-45 indicates the specific VFPv3 data-processing instruction. The Single precision cycles column indicates the number of cycles required for normal single-precision inputs of the associated instruction. The Double precision cycles column indicates the number of cycles required for normal double-precision inputs of the associated instruction. For example, a double-precision FMUL instruction takes anywhere between 11 and 17 cycles, depending on the data. A single- or double-precision FCMP instruction takes either four or seven cycles, depending on the data.

The reason for the wide range of cycles required for normal data is because the VFP coprocessor can detect when a given problem does not require additional computation. For example, if the VFP coprocessor multiplies 3 times 3, the operation takes less time than when it multiplies  $\pi$  times  $\pi$ .

The Subnormal penalty column indicates whether additional cycles are required for subnormal operands, subnormal intermediate values, or subnormal final results. This penalty only applies when the VFP coprocessor has flush-to-zero mode disabled.

For operations that have the result penalty, six to seven additional cycles are required to format the final result.

For operations that have the operand penalty:

- one subnormal operand requires five to six additional cycles
- two subnormal operands require nine to ten additional cycles
- three subnormal operands require nine to ten additional cycles plus an intermediate penalty.

All 3-input operations FMAC, FNMAC, FMSC, or FNMSC are variations of multiply-add, that is, a multiplication followed by an addition. The multiplication produces an intermediate result that might itself be subnormal. This intermediate subnormal has a penalty that is the same as the output penalty (applied to the multiply) plus the input penalty (applied to the addition), which amounts to an additional 11-13 cycles.

A slightly simpler way to look at 3-input operation is to split them into equivalent multiply and add instructions. A 3-input operation takes the same amount of time as its component multiplication and addition, usually minus one cycle.

An FMAC operation with three normal operands might have a multiplication that takes 12 cycles and an addition that takes nine cycles. The corresponding multiply followed by add instruction takes:

$$12 + 9 - 1 = 20 \text{ cycles}$$

For a multiplication of a normal number with a subnormal number that results in a product that is also subnormal, this operation has an operand and result penalty and takes a total of 21 to 25 cycles. We then add the subnormal product to another subnormal number, resulting in a normal sum. This addition has two operand penalties, and takes a total of 18 to 20 cycles. The total time the two operations take is between:

$$10 + 5 + 6 + 18 = 39 \text{ cycles and } 12 + 6 + 7 + 20 = 45 \text{ cycles}$$

The corresponding FMAC multiply followed by add instruction has two operand penalties of nine to 10 cycles, an intermediate penalty of 11 to 13 cycles, and the cost of the multiply-add of 18 to 21 cycles. The total time is between:

$$9 + 11 + 18 = 38 \text{ cycles and } 10 + 13 + 21 = 44 \text{ cycles}$$

## 16.7.2 VFP instruction execution in the NFP pipeline

The NFP pipeline can execute a subset of the VFPv3 data-processing instructions more quickly than the VFP coprocessor. The following constraints define which VFP instructions are executable by the NFP pipeline:

- single-precision data-processing operations only
- RunFast mode must be enabled
- scalar only or non-short vector instructions

If these constraints are met, the following instructions can execute in the NFP pipeline:

- FADDs, FSUBS
- FABSS, FNEGS
- FMULS, FNMULS
- FMACS, FNMACS
- FMSCS, FNMSCS
- FCMPs, FCMPEs
- FCMPEZs, FCMPEZS
- FUITOs, FSITOs
- FTOUTIs, FTOSIS
- FTOUTIZs, FTOSIZs
- FSHTOs, FSLTOS
- FUHTOs, FULTOs
- FTOSHs, FTOSLS
- FTOUTHs, FTOUTLS

VFP instructions that execute in the NFP pipeline have results that are 32-bit single-precision writes to the upper or lower half of the 64-bit register value. A restriction that applies to VFP instructions executing in the NFP pipeline is that instruction results cannot be forwarded early to subsequent instructions. Each VFP instruction takes 7 cycles to execute in the NFP pipeline because of this restriction.

## 16.8 Scheduling example

Example 16-6 shows a sample code segment and how the processor might schedule it.

**Example 16-6 Dual issue instruction sequence for integer pipeline**

Cycle	PC	Opcode	Instruction	Timing description
1	0x0000ed0:	0xe12fff1e	BX r14	Dual issue pipeline 0
1	0x0000ee4:	0xe3500000	CMP r0,#0	Dual issue in pipeline 1
2	0x0000ee8:	0xe3a03003	MOV r3,#3	Dual issue pipeline 0
2	0x0000eec:	0xe3a00000	MOV r0,#0	Dual issue in pipeline 1
3	0x0000ef0:	0x05813000	STREQ r3,[r1,#0]	Dual issue in pipeline 0, r3 not needed until E3
3	0x0000ef4:	0xe3520004	CMP r2,#4	Dual issue in pipeline 1
4	0x0000ef8:	0x979ff102	LDRLS pc,[pc,r2,LSL #2]	Single issue pipeline 0, +1 cycle for load to pc, no extra cycle for shift since LSL #2
5	0x0000f2c:	0xe3a00001	MOV r0,#1	Dual issue with 2nd iteration of load in pipeline 1
6	0x0000f30:	0xea000000	B {pc}+8	#0xf38 dual issue pipeline 0
6	0x0000f38:	0xe5810000	STR r0,[r1,#0]	Dual issue pipeline 1
7	0x0000f3c:	0xe49df004	LDR pc,[r13],#4	Single issue pipeline 0, +1 cycle for load to pc
8	0x000017c:	0xe284200c	ADD r2,r4,#0xc	Dual issue with 2nd iteration of load in pipeline 1
9	0x0000180:	0xe5960004	LDR r0,[r6,#4]	Dual issue pipeline 0
9	0x0000184:	0xe3a0100a	MOV r1,#0xa	Dual issue pipeline 1
12	0x0000188:	0xe5900000	LDR r0,[r0,#0]	Single issue pipeline 0: r0 produced in E3, required in E1, so +2 cycle stall
13	0x000018c:	0xe5840000	STR r0,[r4,#0]	Single issue pipeline 0 due to LS resource hazard, no extra delay for r0 since produced in E3 and consumed in E3
14	0x0000190:	0xe594000c	LDR r0,[r4,#0xc]	Single issue pipeline 0 due to LS resource hazard
15	0x0000194:	0xe8bd4070	LDMFD r13!,{r4-r6,r14}	Load multiple loads r4 in 1st cycle, r5 and r6 in 2nd cycle, r14 in 3rd cycle, 3 cycles total
17	0x0000198:	0xea000368	B {pc}+0xda8	#0xf40 dual issue in pipeline 1 with 3rd cycle of LDM
18	0x0000f40:	0xe2800002	ADD r0,r0,#2 ARM	Single issue in pipeline 0
19	0x0000f44:	0xe0810000	ADD r0,r1,r0 ARM	Single issue in pipeline 0, no dual issue due to hazard on r0 produced in E2 and required in E2

Example 16-7 shows a sample instruction sequence for the NEON pipeline.

**Example 16-7 Instruction sequence for the NEON pipeline**

Cycle	PC	Opcode	Instruction	Timing description
1	0x00003690:	0xf2dbeac8	VMULL.S16 q15,d27,d0[1]	;4X16 SIMD multiply

2	0x00003694: 0xf2daaac8	VMULL.S16 q13,d26,d0[1]	;independent from previous multiply, issued in back-to-back cycles
2	0x00003698: 0xf4402a5d	VST1.16 {d18,d19},[r0@64]!	;128bit 2-issue cycle store (1st issue cycle is dual issued with previous instruction)
3	0x0000369c: 0xf2d7685a	VRSHRN.I32 d22,q5,#9	;shift operation (dual issued with 2nd issue cycle of previous store)
4	0x000036a0: 0xf2d7785c	VRSHRN.I32 d23,q6,#9	;independent from previous shift, executed in back-to-back cycles
5	0x000036a4: 0xf29caac0	VMULL.S16 q5,d28,d0[0]	;4X16 SIMD multiply
6	0x000036a8: 0xf29dcac0	VMULL.S16 q6,d29,d0[0]	;independent from previous multiply, issued in back-to-back cycles
7	0x000036ac: 0xf26aa8c6	VADD.I32 q13,q13,q3	;4x32 (128bit) VADD uses result of multiply from cycle 2.
8	0x000036b0: 0xf26ee8c8	VADD.I32 q15,q15,q4	;4x32 (128bit) independent from previous add, issued in back-to-back cycles
9	0x000036b4: 0xf29e6260	VMLAL.S16 q3,d14,d0[2]	;independent multiply
9	0x000036bc: 0xf4004a5d	VST1.16 {d4,d5},[r0@64]!	;128bit 2-issue cycle store (1st issue cycle is dual issued with previous instruction)
10	0x000036c0: 0xf2d7a87a	VRSHRN.I32 d26,q13,#9	;shift operation (dual issued with 2nd issue cycle of previous store)

Example 16-8 shows an instruction sequence for the VFP pipeline.

#### Example 16-8 Instruction sequence for VFP pipeline

Cycle	PC	Opcode	Instruction	Timing description
4	0x00002c44:	0xeeb01a49	FCPYS s2,s18	;4 cycle single precision register move
8	0x00002c48:	0xeef00a68	FCPYS s1,s17	;4 cycle single precision register move
12	0x00002c4c:	0xeeb00a48	FCPYS s0,s16	;4 cycle single precision register move
12	0x00002c50:	0xeb000116	BL {pc}+0x460	;branch executed 'for free' in ARM pipeline, not seen by Neon
19	0x000030b0:	0xee200a21	FMULS s0,s0,s3	;7 cycle single precision multiply operation
30	0x000030b4:	0xee000a82	FMACS s0,s1,s4	;11 cycle single precision multiply accumulate (uses NFP multiply and add pipelines with bypassing of add format stage)
41	0x000030b8:	0xee010a22	FMACS s0,s2,s5	;11 cycle single precision multiply accumulate (uses NFP multiply and add pipelines with bypassing of add format stage)
41	0x000030bc:	0xe12fff1e	BX lr	;branch executed 'for free' in ARM pipeline, not seen by Neon
45	0x00002c54:	0xeeb01a4a	FCPYS s2,s20	;4 cycle single precision register move
80	0x00002c58:	0xeeb10ac0	FSQRTS s0,s0	;35 cycles to execute single precision square root function (number of cycles is data dependent)
112	0x00002c5c:	0xeec10a00	FDIVS s1,s2,s0	;32 cycles to execute single precision divide function (number of cycles is data dependent)
116	0x00002c60:	0xeeb00a69	FCPYS s0,s19	;4 cycle single precision register move

123    0x00002c64: 0xee600a20 FMULS    s1,s0,s1    ;7 cycle single precision multiply operation

---





# Chapter 17

## AC Characteristics

This chapter describes the timing parameters of the processor signals:

- *About setup and hold times* on page 17-2
- *AXI interface* on page 17-4
- *ATB and CTI interfaces* on page 17-6
- *APB interface and miscellaneous debug signals* on page 17-7
- *L1 and L2 MBIST interfaces* on page 17-9
- *L2 preload interface* on page 17-10
- *DFT interface* on page 17-11
- *Miscellaneous signals* on page 17-12.

## 17.1 About setup and hold times

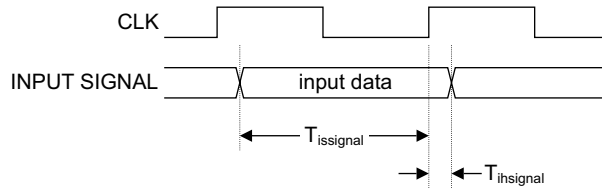
The setup and hold times of processor interface signals are necessary timing parameters for analyzing processor performance. This chapter specifies the setup and hold times of the processor interface signals.

The notation for setup and hold times of input signals is:

**T<sub>is</sub>** Input setup time. T<sub>is</sub> is the amount of time the input data is valid before the next rising clock edge.

**T<sub>ih</sub>** Input hold time. T<sub>ih</sub> is the amount of time the input data is valid after the next rising clock edge.

Figure 17-1 shows the setup and hold times of an input signal.



**Figure 17-1 Input timing parameters**

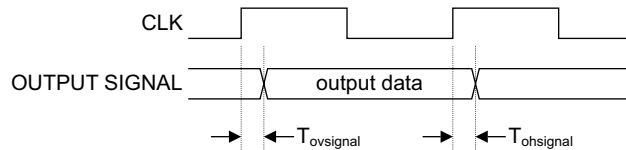
The time during which the processor can sample input data is **T<sub>isignal</sub>**.

The notation for setup and hold times of output signals is:

**T<sub>ov</sub>** Output valid time. T<sub>ov</sub> is the amount of time after the rising clock edge before valid output data appears.

**T<sub>oh</sub>** Output hold time. T<sub>oh</sub> is the amount of time the output data is valid after the next rising clock edge.

Figure 17-2 shows the setup and hold times of an output signal.



**Figure 17-2 Output timing parameters**

The timing parameter tables in this chapter show setup and hold parameters of each signal as percentages of the relevant clock as shown in Table 17-1.

**Table 17-1 Format of timing parameter tables**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
INPUT	CLK	$T_{\text{isinput}}$	50%	$T_{\text{ihinput}}$
OUTPUT	PCLK	$T_{\text{ovoutput}}$	30%	$T_{\text{ohoutput}}$

The setup parameter values are based on the *Slow-Slow* (SS) corner under the following conditions:

- 125 °C
- $V_{\text{DD}}$  = nominal operating voltage – 10%
- target frequency =  $f_{\text{max}}$ .

The hold parameter values are based on the *Fast-Fast* (FF) corner under the following conditions:

- -40 C
- $V_{\text{DD}}$  = nominal operating voltage + 10%
- target frequency =  $f_{\text{max}}$ .

The nominal operating voltage for the process is defined to be  $V_{\text{dd}}$ .

———— **Note** —————

The hold time requirements for the macrocell I/O are not specified in this document. The hold time is specific to process and implementation requirements and therefore, are controlled by the implementor.

## 17.2 AXI interface

Table 17-2 shows the setup and hold times for the AXI interface signals.

**Table 17-2 Timing parameters of AXI interface**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
A64n128 <sup>a</sup>	CLK	-	-	-
ACLKEN	CLK	T <sub>isacklen</sub>	30%	T <sub>ihaclken</sub>
ARADDR[31:0]	CLK	T <sub>ovaraddr</sub>	30%	T <sub>oharaddr</sub>
ARBURST[1:0]	CLK	T <sub>ovarburst</sub>	30%	T <sub>oharburst</sub>
ARCACHE[3:0]	CLK	T <sub>ovarcache</sub>	30%	T <sub>oharcache</sub>
ARID[3:0]	CLK	T <sub>ovarid</sub>	30%	T <sub>oharid</sub>
ARLEN[3:0]	CLK	T <sub>ovarlen</sub>	30%	T <sub>oharlen</sub>
ARLOCK[1:0]	CLK	T <sub>ovarlock</sub>	30%	T <sub>oharlock</sub>
ARPROT[2:0]	CLK	T <sub>ovarprot</sub>	30%	T <sub>oharprot</sub>
ARSIZE[2:0]	CLK	T <sub>ovarsize</sub>	30%	T <sub>oharsize</sub>
ARVALID	CLK	T <sub>ovarvalid</sub>	30%	T <sub>oharvalid</sub>
ARREADY	CLK	T <sub>isarready</sub>	30%	T <sub>iharready</sub>
RDATA[127:0]	CLK	T <sub>isrdata</sub>	30%	T <sub>ihrdata</sub>
RID[3:0]	CLK	T <sub>isrid</sub>	30%	T <sub>ihrid</sub>
RLAST	CLK	T <sub>isrlast</sub>	30%	T <sub>ihrlast</sub>
RRESP[1:0]	CLK	T <sub>isrresp</sub>	30%	T <sub>ihrresp</sub>
RVALID	CLK	T <sub>isrvalid</sub>	30%	T <sub>ihrvalid</sub>
RREADY	CLK	T <sub>ovrready</sub>	30%	T <sub>ohrready</sub>
AWADDR[31:0]	CLK	T <sub>ovawaddr</sub>	30%	T <sub>ohawaddr</sub>
AWBURST[1:0]	CLK	T <sub>ovawburst</sub>	30%	T <sub>ohawburst</sub>
AWCACHE[3:0]	CLK	T <sub>ovawcache</sub>	30%	T <sub>ohawcache</sub>

Table 17-2 Timing parameters of AXI interface (continued)

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
<b>AWID[3:0]</b>	<b>CLK</b>	$T_{ovawid}$	30%	$T_{ohawid}$
<b>AWLEN[3:0]</b>	<b>CLK</b>	$T_{ovawlen}$	30%	$T_{ohawlen}$
<b>AWLOCK[1:0]</b>	<b>CLK</b>	$T_{ovawlock}$	30%	$T_{ohawlock}$
<b>AWPROT[2:0]</b>	<b>CLK</b>	$T_{ovawprot}$	30%	$T_{ohawprot}$
<b>AWSIZE[2:0]</b>	<b>CLK</b>	$T_{ovawsize}$	30%	$T_{ohawsize}$
<b>AWVALID</b>	<b>CLK</b>	$T_{ovawvalid}$	30%	$T_{ohawvalid}$
<b>AWREADY</b>	<b>CLK</b>	$T_{isawready}$	30%	$T_{ihawready}$
<b>WDATA[127:0]</b>	<b>CLK</b>	$T_{ovwdata}$	30%	$T_{ohwdata}$
<b>WID[3:0]</b>	<b>CLK</b>	$T_{ovwid}$	30%	$T_{ohwid}$
<b>WLAST</b>	<b>CLK</b>	$T_{ovwlast}$	30%	$T_{ohwlast}$
<b>WSTRB[15:0]</b>	<b>CLK</b>	$T_{ovwstrb}$	30%	$T_{ohwstrb}$
<b>WVALID</b>	<b>CLK</b>	$T_{ovwvalid}$	30%	$T_{ohwvalid}$
<b>WREADY</b>	<b>CLK</b>	$T_{iswready}$	30%	$T_{ihwready}$
<b>BID[3:0]</b>	<b>CLK</b>	$T_{isbid}$	30%	$T_{ihbid}$
<b>BRESP[1:0]</b>	<b>CLK</b>	$T_{isbresp}$	30%	$T_{ihbresp}$
<b>BVALID</b>	<b>CLK</b>	$T_{isbvalid}$	30%	$T_{ihbvalid}$
<b>BREADY</b>	<b>CLK</b>	$T_{ovbready}$	30%	$T_{ohbready}$

a. This is a static input to the processor.

## 17.3 ATB and CTI interfaces

Table 17-3 shows the setup and hold times for:

- the ATB interface
- the CTI.

**Table 17-3 Timing parameters of ATB and CTI interfaces**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
AFREADYM	ATCLK	T <sub>ovafreadym</sub>	30%	T <sub>ohafreadym</sub>
AFVALIDM	ATCLK	T <sub>isafvalidm</sub>	30%	T <sub>ihafvalidm</sub>
ASICCTL[7:0]	CLK	T <sub>ovasicctl</sub>	30%	T <sub>ohasicctl</sub>
ATRESETn <sup>a,b</sup>	ATCLK	-	-	-
ATCLKEN	ATCLK	T <sub>isatclken</sub>	30%	T <sub>ihatclken</sub>
ATREADYM	ATCLK	T <sub>isatreadym</sub>	30%	T <sub>ihatreadym</sub>
ATBYTESM[1:0]	ATCLK	T <sub>ovatbytesm</sub>	30%	T <sub>ohatbytesm</sub>
ATDATAM[31:0]	ATCLK	T <sub>ovatdatam</sub>	30%	T <sub>ohatdatam</sub>
ATIDM[6:0]	ATCLK	T <sub>ovatidm</sub>	30%	T <sub>ohatidm</sub>
ATVALIDM	ATCLK	T <sub>ovatvalidm</sub>	30%	T <sub>ohatvalidm</sub>
TRIGGER	ATCLK	T <sub>ovtrigger</sub>	30%	T <sub>ohtrigger</sub>
CTICHOUT[3:0]	ATCLK	T <sub>ovctichout</sub>	30%	T <sub>ohctichout</sub>
CTICHIN[3:0]	ATCLK	-	-	-
nCTIIRQ <sup>a</sup>	CLK	T <sub>ovnctiirq</sub>	30%	T <sub>ohnctiirq</sub>

a. This signal has multiple end-points and must be treated as level-sensitive.

b. Figure 10-6 on page 10-6 shows how this signal must be set up.

## 17.4 APB interface and miscellaneous debug signals

Table 17-4 shows the setup and hold times for:

- the APB interface
- the miscellaneous debug signals.

**PCLK** is the clock for the APB interface and some miscellaneous debug signals, and **CLK** is the clock for all other miscellaneous debug signals.

**Table 17-4 Timing parameters of APB interface and miscellaneous debug signals**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
COMMRX <sup>a</sup>	CLK	T <sub>ovcommrx</sub>	30%	T <sub>ohcommrx</sub>
COMMTX <sup>a</sup>	CLK	T <sub>ovcommtx</sub>	30%	T <sub>ohcommtx</sub>
DBGACK	CLK	T <sub>ovdbgack</sub>	30%	T <sub>ohdbgack</sub>
DBGNOCLKSTOP	CLK	T <sub>isdbgnoclkstop</sub>	30%	T <sub>ihdbgnoclkstop</sub>
DBGROMADDR[31:12] <sup>b</sup>	CLK	T <sub>isdbgromaddr</sub>	30%	T <sub>ihdbgromaddr</sub>
DBGROMADDRV <sup>b</sup>	CLK	T <sub>isdbgromaddrv</sub>	30%	T <sub>ihdbgromaddrv</sub>
DBGSELFADDR[31:12] <sup>b</sup>	CLK	T <sub>isdbgselfaddr</sub>	30%	T <sub>ihdbgselfaddr</sub>
DBGSELFADDRV <sup>b</sup>	CLK	T <sub>isdbgselfaddrv</sub>	30%	T <sub>ihdbgselfaddrv</sub>
EDBGRQ <sup>a</sup>	PCLK	-	-	-
DBGEN	PCLK	-	-	-
DBGOSLOCKINIT <sup>b</sup>	PCLK	T <sub>isdbgoslockinit</sub>	30%	T <sub>ihdbgoslockinit</sub>
DBGNOPWRDWN <sup>a</sup>	PCLK	T <sub>ovdbgnopwrdsn</sub>	30%	T <sub>ohdbgnopwrdsn</sub>
DBGPWRDWNREQ <sup>a</sup>	PCLK	-	-	-
ETMPWRDWNREQ <sup>a,c</sup>	PCLK	-	-	-
DBGPWRDWNACK	PCLK	T <sub>ovdbgpwrdsnack</sub>	30%	T <sub>ohdbgpwrdsnack</sub>
ETMPWRDWNACK <sup>c</sup>	PCLK	T <sub>ovetmpwrdsnack</sub>	30%	T <sub>ohetmpwrdsnack</sub>
PRESET <sub>n</sub> <sup>a,d</sup>	PCLK	-	-	-
PCLKEN	PCLK	T <sub>ispclken</sub>	30%	T <sub>ihpclken</sub>

**Table 17-4 Timing parameters of APB interface and miscellaneous debug signals (continued)**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
<b>PADDR31</b>	<b>PCLK</b>	$T_{ispaddr31}$	30%	$T_{ihpaddr31}$
<b>PADDR11TO2[11:2]</b>	<b>PCLK</b>	$T_{ispaddr11to2}$	30%	$T_{ihpaddr11to2}$
<b>PENABLE</b>	<b>PCLK</b>	$T_{ispenable}$	30%	$T_{ihpenable}$
<b>PSELCTI<sup>e</sup></b>	<b>PCLK</b>	$T_{ispselecti}$	30%	$T_{ihpselecti}$
<b>PSELDBG</b>	<b>PCLK</b>	$T_{ispseldbq}$	30%	$T_{ihpseldbq}$
<b>PSELETM<sup>d</sup></b>	<b>PCLK</b>	$T_{ispselectm}$	30%	$T_{ihpselectm}$
<b>PWRITE</b>	<b>PCLK</b>	$T_{ispwrite}$	30%	$T_{ihpwrite}$
<b>PRDATA[31:0]</b>	<b>PCLK</b>	$T_{ovprdata}$	30%	$T_{ohprdata}$
<b>PWDATA[31:0]</b>	<b>PCLK</b>	$T_{ispwdata}$	30%	$T_{ihpwdata}$
<b>PREADY</b>	<b>PCLK</b>	$T_{ovpready}$	30%	$T_{ohpready}$
<b>PSLVERR</b>	<b>PCLK</b>	$T_{ovpslverr}$	30%	$T_{ohpslverr}$
<b>NIDEN<sup>a</sup></b>	<b>PCLK</b>	-	-	-
<b>SPIDEN<sup>a</sup></b>	<b>PCLK</b>	-	-	-
<b>SPNIDEN<sup>a</sup></b>	<b>PCLK</b>	-	-	-

a. This signal has multiple end-points and must be treated as level-sensitive.

b. This is a static input to the processor.

c. This signal is not required because debug and the ETM use the same power domain.

d. Figure 10-6 on page 10-6 shows how this signal must be set up.

e. This signal is not present when the processor is configured without the ETM.



## 17.5 L1 and L2 MBIST interfaces

Table 17-5 shows the setup and hold times for the L1 and L2 interfaces

**Table 17-5 Timing parameters of the L1 and L2 MBIST interface**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
MBISTDATAINL1	CLK	$T_{ismbistdatain1}$	30%	$T_{ihmbistdatain1}$
MBISTDSHIFTL1	CLK	$T_{ismbistdshift1}$	30%	$T_{ihmbistdshift1}$
MBISTMODEL1	CLK	$T_{ismbistmodel1}$	30%	$T_{ihmbistmodel1}$
MBISTRUNL1	CLK	$T_{ismbistrun1}$	30%	$T_{ihmbistrun1}$
MBISTSHIFTL1	CLK	$T_{ismbistshift1}$	30%	$T_{ihmbistshift1}$
MBISTRESULTL1[2:0]	CLK	$T_{ovmbistresult1}$	30%	$T_{ihmbistresult1}$
MBISTDATAINL2	CLK	$T_{ismbistdatain2}$	30%	$T_{ihmbistdatain2}$
MBISTDSHIFTL2	CLK	$T_{ismbistdshift2}$	30%	$T_{ihmbistdshift2}$
MBISTMODEL2	CLK	$T_{ismbistmodel2}$	30%	$T_{ihmbistmodel2}$
MBISTRUNL2	CLK	$T_{ismbistrun2}$	30%	$T_{ihmbistrun2}$
MBISTSHIFTL2	CLK	$T_{ismbistshift2}$	30%	$T_{ihmbistshift2}$
MBISTRESULTL2[2:0]	CLK	$T_{ovmbistresult2}$	30%	$T_{ohmbistresult2}$
MBISTUSERINL2[18:0]	CLK	$T_{ismbistuserin2}$	30%	$T_{ihmbistuserin2}$
MBISTUSEROUTL2[4:0]	CLK	$T_{ovmbistuserout2}$	50%	$T_{ohmbistuserout2}$

## 17.6 L2 preload interface

Table 17-6 shows the setup and hold times of the interrupt sources from the L2 preload engine.

**Table 17-6 Timing parameters of the L2 preload interface**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
<b>nDMAEXTERRIRQ<sup>a</sup></b>	<b>CLK</b>	$T_{ovndmaexterrirq}$	30%	$T_{ohndmaexterrirq}$
<b>nDMAIRQ<sup>a</sup></b>	<b>CLK</b>	$T_{ovndmairq}$	30%	$T_{ohndmairq}$
<b>nDMASIRQ<sup>a</sup></b>	<b>CLK</b>	$T_{ovndmasirq}$	30%	$T_{ohndmasirq}$

a. This signal has multiple end-points and must be treated as level-sensitive.

## 17.7 DFT interface

Table 17-7 shows the setup and hold times for the DFT interface.

**Table 17-7 Timing parameters of the DFT interface**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
SE	CLK	$T_{isse}$	30%	$T_{ihse}$
SERIALTEST	CLK	$T_{isserialtest}$	30%	$T_{ihserialtest}$
SHIFTWR	CLK	$T_{isshiftwr}$	30%	$T_{ihshiftwr}$
CAPTUREWR	CLK	$T_{iscapturewr}$	30%	$T_{ihcapturewr}$
TESTMODE	CLK	$T_{itestmode}$	30%	$T_{ihitestmode}$
TESTCGATE	CLK	$T_{itestcgate}$	30%	$T_{ihitestcgate}$
TESTEGATE	CLK	$T_{itestegate}$	30%	$T_{ihitestegate}$
TESTNGATE	CLK	$T_{itestngate}$	30%	$T_{ihitestngate}$
WSE	CLK	$T_{iswse}$	30%	$T_{ihwse}$
WEXTEST	CLK	$T_{iswextest}$	30%	$T_{ihwextest}$
WINTEST	CLK	$T_{iswintest}$	30%	$T_{ihwintest}$

## 17.8 Miscellaneous signals

Table 17-8 shows the setup and hold times of miscellaneous signals not described in the previous sections.

**Table 17-8 Timing parameters of miscellaneous signals**

Signal	Clock	Setup parameter	Percent of clock period	Hold parameter
nPORESET <sup>a,b</sup>	CLK	-	-	-
ARESETn <sup>a,b</sup>	CLK	-	-	-
ARESETNEONn <sup>a,b</sup>	CLK	-	-	-
L1RSTDISABLE <sup>c</sup>	CLK	-	-	-
L2RSTDISABLE <sup>c</sup>	CLK	-	-	-
CLKSTOPREQ	CLK	-	-	-
CLKSTOPACK	CLK	T <sub>ovclkstopack</sub>	30%	T <sub>ohclkstopack</sub>
SECMONOUTEN <sup>d</sup>	CLK	-	-	-
SECMONOUT[86:0]	CLK	T <sub>ovsecmonout</sub>	30%	T <sub>ohsecmonout</sub>
STANDBYWFI	CLK	T <sub>ovstandbywfi</sub>	30%	T <sub>ohstandbywfi</sub>
nFIQ <sup>a</sup>	CLK	-	-	-
nIRQ <sup>a</sup>	CLK	-	-	-
VINITHI <sup>c</sup>	CLK	-	-	-
CFGTE <sup>c</sup>	CLK	-	-	-
CFGEND0 <sup>c</sup>	CLK	-	-	-
CFGNMFI <sup>c</sup>	CLK	-	-	-
CP15SDISABLE	CLK	-	-	-
CPEXIST[13:0] <sup>c</sup>	CLK	-	-	-
SILICONID[31:0] <sup>c</sup>	CLK	-	-	-
nPMUIRQ <sup>a</sup>	CLK	T <sub>ovnpmuirq</sub>	30%	T <sub>ohnpmuirq</sub>

a. This signal has multiple end-points and must be treated as level-sensitive.

- b. Figure 10-6 on page 10-6 shows how this signal must be set up.
- c. This is a static input to the processor.
- d. This signal is sampled only during reset.



# Appendix A

## Signal Descriptions

This appendix describes the signals of the processor. It contains the following sections:

- *AXI interface* on page A-2
- *ATB interface* on page A-3
- *MBIST and DFT interface* on page A-4
- *Preload engine interface* on page A-7
- *APB interface* on page A-8
- *Miscellaneous signals* on page A-10
- *Miscellaneous debug signals* on page A-14
- *Miscellaneous ETM and CTI signals* on page A-17.

———— **Note** —————

For each output signal of the processor, the value in the Reset column in the signal tables can either be defined as a logic 1 or 0 or undefined during reset.

—————

## A.1 AXI interface

For complete descriptions of AXI interface signals, see the *AMBA AXI Protocol Specification*.

Table A-1 shows the AXI interface signals that have been added or that have different definitions for the Cortex-A8 processor.

**Table A-1 AXI interface**

Signal	I/O	Reset	Description
<b>A64n128</b>	I	-	Statically selects 64-bit or 128-bit AXI bus width: 0 = 128-bit bus width 1 = 64-bit bus width. This pin is only sampled during reset of the processor.
<b>ACLKEN</b>	I	-	AXI clock gate enable: 0 = AXI clock disabled 1 = AXI clock enabled.  <div style="text-align: center;"> <p>———— <b>Note</b> —————</p> <p>The rising edge of the internal <b>ACLK</b> signal comes two <b>CLK</b> cycles after the <b>CLK</b> cycle in which <b>ACLKEN</b> is asserted. See Chapter 10 <i>Clock, Reset, and Power Control</i>.</p> </div>
<b>ARCACHE[3:0]</b> and <b>AWCACHE[3:0]</b>	O	Undefined	Read or write cache type: b0000 = strongly ordered b0001 = device b0010 = reserved b0011 = normal noncacheable b0100 and b0101 = reserved b0110 = cacheable write-through, allocate on reads only b0111 = cacheable write-back, allocate on reads only b1000 to b1110 = reserved b1111 = cacheable write-back, allocate on both reads and writes.



## A.2 ATB interface

Table A-2 shows the signals of the ATB interface.

**Table A-2 ATB interface**

Signal	I/O	Reset	Description
<b>AFREADYM</b>	O	b0	Buffer ready for new data: 0 = buffer flush not complete 1 = buffer flush complete.
<b>AFVALIDM</b>	I	-	Flush request: 0 = no ETM data flush request 1 = ETM data flush request.
<b>ATBYTESM[1:0]</b>	O	b00	Size of data: b00 = 1 byte b01 = 2 bytes b10 = 3 bytes b11 = 4 bytes.
<b>ATCLK</b>	I	-	ATB clock.
<b>ATCLKEN</b>	I	-	ATB clock enable: 0 = clock not enabled 1 = clock enabled.
<b>ATDATAM[31:0]</b>	O	0x00000000	ATB data bus.
<b>ATIDM[6:0]</b>	O	b0000000	Identification tag of current trace source.
<b>ATREADYM</b>	I	-	ATB device ready. Previous data accepted: 0 = not ready 1 = ready.
<b>ATRESETn</b>	I	-	ATB reset input.
<b>ATVALIDM</b>	O	b0	ATB valid data: 0 = no valid data 1 = valid data.

## A.3 MBIST and DFT interface

This section describes:

- *MBIST interface*
- *DFT pins and additional MBIST pin requirements during MBIST testing on page A-5.*

### A.3.1 MBIST interface

Table A-3 shows the MBIST interface signals. All MBIST interface signals are registered. All MBIST signals from Table A-3 must be controllable from external SoC pins for use by ATE.

**Table A-3 MBIST interface**

Signal	I/O	Reset	Description
<b>MBISTDATAINL1</b>	I	-	Serial data input for loading the L1 MBIST Instruction Register
<b>MBISTDSHIFTL1</b>	I	-	Enables download of the L1 MBIST Datalog Register on <b>MBISTRESULTL1[0]</b> when <b>MBISTSHIFTL1</b> is set to 0.
<b>MBISTRUNL1</b>	I	-	Enables execution of the loaded L1 MBIST instruction.
<b>MBISTSHIFTL1</b>	I	-	Enables serial loading of the L1 MBIST Instruction Register when <b>MBISTDSHIFTL1</b> is set to 0 or enables serial loading of the L1 MBIST GO-NOGO Instruction Register when <b>MBISTDSHIFTL1</b> is set to 1.
<b>MBISTRESULTL1[2:0]</b>	O	Undefined	L1 MBIST controller status and data output: <b>MBISTRESULTL1[0]</b> = address expire flag or L1 MBIST Datalog Register output <b>MBISTRESULTL1[1]</b> = fail flag <b>MBISTRESULTL1[2]</b> = test complete flag.
<b>MBISTDATAINL2</b>	I	-	Serial data input for loading the L2 MBIST Instruction Register.
<b>MBISTDSHIFTL2</b>	I	-	Enables download of the L2 MBIST Datalog Register on <b>MBISTRESULTL2[0]</b> .
<b>MBISTRUNL2</b>	I	-	Enables execution of the loaded L2 MBIST instruction.
<b>MBISTSHIFTL2</b>	I	-	Enables serial loading of the L2 MBIST Instruction Register.

Table A-3 MBIST interface (continued)

Signal	I/O	Reset	Description
<b>MBISTRESULTL2[2:0]</b>	O	Undefined	L2 MBIST controller status and data output: <b>MBISTRESULTL2[0]</b> = address expire flag or L2 MBIST Datalog Register output <b>MBISTRESULTL2[1]</b> = fail flag <b>MBISTRESULTL2[2]</b> = test complete flag.
<b>MBISTUSERINL2[18:0]</b>	I	-	L2 MBIST configuration pins reserved for future expansion. Tie these pins LOW.
<b>MBISTUSEROUTL2[4:0]</b>	O	Undefined	L2 MBIST configuration pins reserved for future expansion. Ignore these pins.
<b>CLK</b>	I	-	Clock input.
<b>ARESETn</b>	I	-	Reset input <sup>a</sup> .
<b>nPORESET</b>	I	-	Reset input <sup>a</sup> .

a. Reset input is controlled in the same way during MBIST mode as during functional mode. See *Reset domains* on page 10-5 for information on reset.

### A.3.2 DFT pins and additional MBIST pin requirements during MBIST testing

Table A-4 shows the signals necessary for DFT. It also shows the additional pins required during MBIST testing.

Table A-4 DFT and additional MBIST pin requirements

Signal	I/O	Value during functional mode	Value during MBIST mode	Description
<b>MBISTMODEL1</b>	I	0	1	Configures L1 for MBIST mode and disables instruction fetch after reset.
<b>MBISTMODEL2</b>	I	0	1	Configures L2 for MBIST mode and disables instruction fetch after reset.
<b>TESTMODE</b>	I	0	0	Indicates ATPG test mode. Deassert during MBIST mode.
<b>TESTCGATE</b>	I	0	1	Controls core clock gating during test mode or MBIST mode.
<b>TESTEGATE</b>	I	0	0	Controls ETM clock gating. Deassert to save power during MBIST mode.

Table A-4 DFT and additional MBIST pin requirements (continued)

Signal	I/O	Value during functional mode	Value during MBIST mode	Description
TESTNGATE	I	0	0	Controls NEON clock gating. Deassert to save power during MBIST mode.
SE	I	0	0	Scan enable signal. Ensures safe shifting of scan chains.
SAFESHIFTRAMIF	I	0	0	Prevents the RAM in the instruction fetch unit from performing a write operation during scan shifting.
SAFESHIFTRAMLS	I	0	0	Prevents the RAM in the load/store unit from performing a write operation during scan shifting.
SAFESHIFTRAML2	I	0	0	Prevents the RAM in the L2 cache unit from performing a write operation during scan shifting.
SERIALTEST	I	0	0	Concatenates the wrapper boundary register scan cells into a single scan chain.
SHIFTWR	I	0	0	IEEE 1500 standard shift signal.
CAPTUREWR	I	0	0	IEEE 1500 standard capture signal.
WINTEST	I	0	0	Enables internal testing during ATPG.
WEXTTEST	I	0	0	Enables external testing during ATPG.
WSE	I	0	0	Wrapper scan enable. Enables serial shifting of the wrapper scan chain.
PRESETn	I	-	0	Active-LOW APB reset input.
ACLKEN	I	-	1	AXI clock enable signal. This signal must be driven HIGH for at least one clock cycle during reset. The value after reset does not affect the MBIST operation.

## A.4 Preload engine interface

Table A-5 shows the L2 preload signals.

**Table A-5 Preload engine interface**

Signal	I/O	Reset	Description
<b>nDMAEXTERRIRQ</b>	O	b1	Active-LOW interrupt on error for all transfers: 0 = interrupt active 1 = interrupt not active.
<b>nDMAIRQ</b>	O	b1	Active-LOW interrupt on completion for nonsecure transfers: 0 = interrupt active 1 = interrupt not active.
<b>nDMASIRQ</b>	O	b1	Active-LOW interrupt on completion for secure transfers: 0 = interrupt active 1 = interrupt not active.

## A.5 APB interface

Table A-6 shows the APB interface signals.

**Table A-6 APB interface**

Signal	I/O	Reset	Description
<b>PRESETn</b>	I	-	Active-LOW APB reset input: 0 = reset APB 1 = do not reset APB.
<b>PCLK</b>	I	-	APB clock.
<b>PCLKEN</b>	I	-	APB clock enable: 0 = not enabled 1 = enabled.
<b>PADDR31</b>	I	-	APB address bus bit [31]: 0 = not an external debugger access 1 = external debugger access.
<b>PADDR11TO2[11:2]</b>	I	-	APB address bus bits [11:2].
<b>PENABLE</b>	I	-	APB transfer complete flag: 0 = APB not in ENABLE cycle 1 = APB in ENABLE cycle. <b>PENABLE</b> remains asserted for only one cycle.
<b>PSELCTI<sup>a</sup></b>	I	-	CTI registers select: 0 = CTI registers not selected 1 = CTI registers selected.
<b>PSELDBG</b>	I	-	Debug registers select: 0 = debug registers not selected 1 = debug registers selected.
<b>PSELETM<sup>a</sup></b>	I	-	ETM registers select: 0 = ETM registers not selected 1 = ETM registers selected.
<b>PWRITE</b>	I	-	APB read or write signal: 0 = reads from APB 1 = writes to APB.
<b>PRDATA[31:0]</b>	O	Undefined	APB read data.

Table A-6 APB interface (continued)

Signal	I/O	Reset	Description
<b>PWDATA[31:0]</b>	I	-	APB write data.
<b>PREADY</b>	O	b0	APB slave ready. An APB slave can assert <b>PREADY</b> to extend a transfer.
<b>PSLVERR</b>	O	b0	APB slave transfer error: 0 = no transfer error 1 = transfer error.

- a. This signal is not present when the processor is configured without the ETM.

## A.6 Miscellaneous signals

Table A-7 shows the signals not included in the previous tables.

**Table A-7 Miscellaneous signals**

Signal	I/O	Reset	Description
<b>nPORESET</b>	I	-	Active-LOW power-on reset input: 0 = apply power-on reset 1 = do not apply power-on reset.
<b>ARESETn</b>	I	-	Active-LOW AXI reset input: 0 = apply AXI reset 1 = do not apply AXI reset.
<b>ARESETNEONn</b>	I	-	Active-LOW NEON reset input: 0 = apply NEON reset 1 = do not apply NEON reset.
<b>SECMONOUTEN</b>	I	-	Security monitor output enable: 0 = disables <b>SECMONOUT[86:0]</b> 1 = enables <b>SECMONOUT[86:0]</b> . This pin is only sampled during reset of the processor.
<b>L1RSTDISABLE</b>	I	-	L1 hardware reset disable input: 0 = the L1 valid RAM contents are reset by hardware 1 = the L1 valid RAM contents are not reset by hardware.
<b>L2RSTDISABLE</b>	I	-	L2 hardware reset disable input: 0 = the L2 valid RAM contents are reset by hardware 1 = the L2 valid RAM contents are not reset by hardware.
<b>CLKSTOPREQ</b>	I	-	Clock stop request: 0 = do not stop the internal clocks 1 = cause the processor to stop the internal clocks and to assert the <b>CLKSTOPACK</b> output.
<b>CLKSTOPACK</b>	O	0	Clock stop acknowledge: 0 = the internal clocks are not stopped 1 = the internal clocks are stopped.



Table A-7 Miscellaneous signals (continued)

Signal	I/O	Reset	Description
<b>SECMONOUT[86:0]</b>	O	Undefined	Security monitor output: [19:0] = pipeline 0 instruction address bits[31:12] [39:20] = pipeline 1 instruction address bits[31:12] [59:40] = L1 data address bits[31:12] [64:60] = exception encoding [69:65] = CPSR[4:0] = mode bits, M[4:0] [73:70] = CPSR[8:5] = bits A, I, F, and T [74] = CPSR[24] = J bit [75] = CP15 Secure Configuration Register bit[0], NS [76] = CP15 Secure Control Register bit[0], M [77] = CP15 Secure Control Register bit[2], C [78] = CP15 Secure Control Register bit[12], I [79] = IMB instruction executed flag [80] = DMB or DWB instruction executed flag [81] = pipeline 0 instruction address valid flag [82] = pipeline 1 instruction address valid flag [83] = condition code fail pipeline 0 valid flag [84] = condition code fail pipeline 1 valid flag [85] = exception valid flag [86] = L1 data address valid flag.
<b>STANDBYWFI</b>	O	b0	Standby mode flag generated by WFI operation: 0 = processor not in standby mode 1 = processor in standby mode.
<b>nFIQ</b>	I	-	Active-LOW asynchronous fast interrupt request: 0 = activate fast interrupt 1 = do not activate fast interrupt. The processor treats the <b>nFIQ</b> input as level sensitive. The <b>nFIQ</b> input must be asserted until the processor acknowledges the interrupt.
<b>nIRQ</b>	I	-	Active-LOW asynchronous interrupt request: 0 = activate interrupt 1 = do not activate interrupt. The processor treats the <b>nIRQ</b> input as level sensitive. The <b>nIRQ</b> input must be asserted until the processor acknowledges the interrupt.

Table A-7 Miscellaneous signals (continued)

Signal	I/O	Reset	Description
<b>VINITHI</b>	I	-	Controls the location of the exception vectors at reset: 0 = starts exception vectors at address 0x00000000 1 = starts exception vectors at address 0xFFFF0000. This pin is only sampled during reset of the processor.
<b>CFGTE</b>	I	-	Controls the state of TE bit in the CP15 c1 Control Register at reset: 0 = TE bit is LOW 1 = TE bit is HIGH. This pin is only sampled during reset of the processor.
<b>CFGEND0</b>	I	-	Controls the state of EE bit in the CP15 c1 Control Register at reset: 0 = EE bit is LOW 1 = EE bit is HIGH. This pin is only sampled during reset of the processor.
<b>CFGNMFI</b>	I	-	Configures fast interrupts to be nonmaskable: 0 = clears the NMFI bit in the CP15 c1 Control Register 1 = sets the NMFI bit in the CP15 c1 Control Register. This pin is only sampled during reset of the processor.
<b>CP15SDISABLE</b>	I	-	Disables CP15.
<b>CPEXIST[13:0]</b>	I	-	Enables access programming of coprocessors 0-13: 0 = CP0-CP13 cannot be programmed for access 1 = CP0-CP13 can be programmed for access. This pin is only sampled during reset of the processor. See <i>c1</i> , <i>Coprocessor Access Control Register</i> on page 3-67 for more details.
<b>SILICONID[31:0]</b>	I	-	Defines the reset value of the CP15 Silicon ID Register. See <i>c0</i> , <i>Silicon ID Register</i> on page 3-53 for more information. This pin is only sampled during reset of the processor.
<b>nPMUIRQ</b>	O	b1	Active-LOW PMU interrupt signal: 0 = PMU interrupt active 1 = PMU interrupt not active.
<b>CLK</b>	I	-	Clock input.

Table A-7 Miscellaneous signals (continued)

Signal	I/O	Reset	Description
<b>CLAMPCOREOUT<sup>a</sup></b>	I	-	Activates the clamps to force the core outputs to benign values: 0 = core clamps not active 1 = core clamps active.
<b>CLAMPNEONOUT<sup>a</sup></b>	I	-	Activates the clamps to force the NEON outputs to benign values: 0 = NEON clamps not active 1 = NEON clamps active.
<b>CLAMPDBGOUT<sup>a</sup></b>	I	-	Activates the clamps to force the debug <b>PCLK</b> , <b>ETM CLK</b> , and <b>ETM ATCLK</b> outputs to benign values: 0 = debug and ETM clamps not active 1 = debug and ETM clamps active.

a. This signal might not be present depending on the IEM support configurable options of the processor.

## A.7 Miscellaneous debug signals

Table A-8 shows the miscellaneous debug signals.

**Table A-8 Miscellaneous debug signals**

Signal	I/O	Reset	Description
<b>COMMRX</b>	O	b0	Receive portion of Data Transfer Register full flag: 0 = empty 1 = full.
<b>COMMTX</b>	O	b0	Transmit portion of Data Transfer Register empty flag: 0 = full 1 = empty.
<b>DBGACK</b>	O	b0	<b>EDBGRQ</b> acknowledge: 0 = external debug request not acknowledged 1 = external debug request acknowledged.
<b>DBGNOCLKSTOP</b>	I	-	Debug clock control signal: 0 = debug disabled while in WFI low-power state 1 = debug enabled while in WFI low-power state.
<b>DBGROMADDR[31:12]</b>	I	-	Debug ROM base address. This pin is only sampled during reset of the processor.
<b>DBGROMADDRV</b>	I	-	Debug ROM base address valid: 0 = address not valid 1 = address valid. This pin is only sampled during reset of the processor.
<b>DBGSELFADDR[31:12]</b>	I	-	2's complement offset from the debug ROM base address. This pin is only sampled during reset of the processor.
<b>DBGSELFADDRV</b>	I	-	Debug port base address valid bit: 0 = address not valid 1 = address valid. This pin is only sampled during reset of the processor.

Table A-8 Miscellaneous debug signals (continued)

Signal	I/O	Reset	Description
<b>EDBGRQ</b>	I	-	External debug request: 0 = no external debug request 1 = external debug request. The processor treats the <b>EDBGRQ</b> input as level sensitive. The <b>EDBGRQ</b> input must be asserted until the processor asserts <b>DBGACK</b> .
<b>DBGEN</b>	I	-	Invasive debug enable: 0 = not enabled 1 = enabled.
<b>DBGOSLOCKINIT</b>	I	-	Reset value for the OS lock: 0 = not locked 1 = locked. This pin is only sampled during reset of the processor.
<b>DBGNOPWRDWN</b>	O	b0	No power down: 0 = do not save state of debug registers 1 = save state of debug registers.
<b>DBGPWRDWNREQ</b>	I	-	Processor power-down request: 0 = no request for processor power down 1 = request for processor power down.
<b>ETMPWRDWNREQ<sup>a</sup></b>	I	-	ETM power-down request: 0 = no request for ETM power down 1 = request for ETM power down.
<b>DBGPWRDWNACK</b>	O	b0	Processor power-down acknowledge 0 = no acknowledge for processor power-down request 1 = acknowledge for processor power-down request.
<b>ETMPWRDWNACK<sup>b</sup></b>	O	b0	ETM power-down acknowledge 0 = no acknowledge for ETM power-down request 1 = acknowledge for ETM power-down request.

Table A-8 Miscellaneous debug signals (continued)

Signal	I/O	Reset	Description
<b>NIDEN</b>	I	-	Noninvasive debug enable: 0 = not enabled 1 = enabled.
<b>SPIDEN</b>	I	-	Secure privileged invasive debug enable: 0 = not enabled 1 = enabled.
<b>SPNIDEN</b>	I	-	Secure privileged noninvasive debug enable: 0 = not enabled 1 = enabled.

- a. This signal is not required because debug and the ETM use the same power domain. **ETMPWRDWNREQ** must be tied to 0. See Chapter 10 *Clock, Reset, and Power Control* for information on the Cortex-A8 supported power domain configurations.
- b. This signal is not required because debug and the ETM use the same power domain. See Chapter 10 *Clock, Reset, and Power Control* for information on the Cortex-A8 supported power domain configurations.

## A.8 Miscellaneous ETM and CTI signals

Table A-9 shows the ETM and CTI signals.

**Table A-9 Miscellaneous ETM and CTI signals**

Signal	I/O	Reset	Description
ASICCTL[7:0]	O	0x00	ETM ASIC Control Register outputs.
TRIGGER	O	b0	ETM trigger flag: 0 = no trigger occurs 1 = trigger occurs.
CTICHOUT[3:0]	O	b0000	CTI channel output status. Each bit represents a valid channel output: 0 = channel output inactive 1 = channel output active.
CTICHIN[3:0]	I	-	CTI channel input status. Each bit represents a valid channel input: 0 = channel input inactive 1 = channel input active.
nCTIIRQ	O	b1	Active-LOW CTI interrupt output: 0 = interrupt active 1 = interrupt not active.





# Appendix B

## Instruction Mnemonics

This appendix lists the *Unified Assembler Language* (UAL) equivalents of the legacy Advanced SIMD data-processing and VFP data-processing assembly language mnemonics used in this manual. It contains the following sections:

- *Advanced SIMD data-processing instructions* on page B-2
- *VFP data-processing instructions* on page B-5.

## B.1 Advanced SIMD data-processing instructions

Table B-1 lists the UAL equivalents of the legacy Advanced SIMD data-processing assembly language mnemonics used in this manual. This table lists only those mnemonics that are different in the UAL syntax.

**Table B-1 Advanced SIMD mnemonics**

Legacy	UAL
Three registers of the same length:	
VFMX	VPMAX
VFMN	VPMIN
VQ{R}DMLH	VQ{R}DMULH
VSUM	VPADD
VCAGE	VACGE
VACGT	VACGT
Three registers of different lengths:	
VADD long	VADDL
VSUB long	VSUBL
VADD wide	VADDW
VSUB wide	VSUBW
V{R}ADH	V{R}ADDHN
VABA long	VABAL
V{R}SBH	V{R}SUBHN
VABD long	VABDL
VMLA long	VMLAL
VQDMLA long	VQDMLAL
VMLS long	VMLSL
VQDMLS long	VQDMLSL
VMUL long	VMULL

**Table B-1 Advanced SIMD mnemonics (continued)**

<b>Legacy</b>	<b>UAL</b>
VQDMUL long	VQDMULL
VMUL polynomial	VMULL
Two registers and a scalar:	
VMLA long	VMLAL
VQDMLA long	VQDMLAL
VMLS long	VMLSL
VQDMLS long	VQDMLSL
VMUL long	VMULL
VQDMUL long	VQDMULL
VQ{R}DMLH	VQ{R}DMULH
Two registers and a shift amount:	
VQSHL	VQSHL{U}
V{R}SHR narrow	V{R}SHRN
VQ{R}SHR narrow	VQ{R}SHR{U}N
VSHL wide	VSHLL
Two registers, miscellaneous:	
VSUM long	VPADDL
VNOT	VMVN
VSMA long	VPADAL
VCGTZ	VCGT #0
VCGEZ	VCGE #0
VCEQZ	VCEQ #0
VCLEZ	VCLE #0
VCLTZ	VCLT #0
VMOV narrow	VMOVN

**Table B-1 Advanced SIMD mnemonics (continued)**

<b>Legacy</b>	<b>UAL</b>
VQMOV narrow	VQMOV{U}N
VMVH wide	VSHLL
Move data element to all lanes of a register:	
VMOV	VDUP

## B.2 VFP data-processing instructions

Table B-2 lists the UAL equivalents of the legacy VFP data-processing assembly language mnemonics used in this manual. The table lists only those mnemonics that are different in the UAL syntax.

**Table B-2 VFP data-processing mnemonics**

<b>Legacy</b>	<b>UAL</b>
FADD	VADD
FSUB	VSUB
FMUL	VMUL
FNMUL	VNMUL
FMAC	VMLA
FNMAC	VMLS
FMSC	VNMLS
FNMSC	VNMLA
FDIV	VDIV
FSQRT	VSQRT
FCONST	VMOV
FABS	VABS
FCPY	VMOV
FNEG	VNEG
FCMP{E}{Z}	VCMP{E}
FCVT	VCVT
F[US]ITO	VCVT
F[US][HL]TO	VCVT
FTO[US]I{Z}	VCVT
FTO[US][HL]	VCVT



# Appendix C

## Revisions

This appendix describes the technical changes between released issues of this book.

**Table C-1 Differences between issue F and issue G**

<b>Change</b>	<b>Location</b>
Added description of product documentation and architecture	<i>Product documentation and architecture</i> on page 1-12
Updated reset value of Main ID Register	<ul style="list-style-type: none"><li>• Table 3-3 on page 3-9</li><li>• <i>c0, Main ID Register</i> on page 3-25.</li></ul>
Updated bit assignments and description of Auxiliary Control Register	<i>c1, Auxiliary Control Register</i> on page 3-61
Show effect of improved cache maintenance	Table 3-73 on page 3-90
Changed description of values for predefined events 0x45 and 0x46	Table 3-97 on page 3-112
Expanded description of DT field in PLE Control Register	Table 3-126 on page 3-144

**Table C-1 Differences between issue F and issue G (continued)**

<b>Change</b>	<b>Location</b>
Updated description of L1 memory system	<ul style="list-style-type: none"> <li>• <i>About the L1 memory system</i> on page 7-2</li> <li>• <i>Cache organization</i> on page 7-3.</li> </ul>
Added section on instruction cache maintenance	<i>Instruction cache maintenance</i> on page 7-4
Reorganized tables for AXI ID assignments	<i>AXI identifiers</i> on page 9-4
Updated descriptions of AXI address channel for data transactions	Table 9-7 on page 9-10
Added table showing number of transfers on AXI write channel for an eviction	<i>Evictions</i> on page 9-8
Clarified timing diagram of STANDBYWFI deassertion	Figure 10-9 on page 10-12
Updated field values for Debug ID Register	<i>CP14 c0, Debug ID Register</i> on page 12-18
Updated field values for Peripheral ID Register 2	<ul style="list-style-type: none"> <li>• Table 12-48 on page 12-68</li> <li>• Table 14-2 on page 14-8</li> <li>• Table 14-6 on page 14-14</li> <li>• Table 15-3 on page 15-10</li> <li>• Table 15-30 on page 15-31.</li> </ul>
Updated ID Register bit assignments	Figure 14-2 on page 14-10
Added tables to show the effect of CP15 cache maintenance	<i>Coprocessor instructions</i> on page 16-12
Added footnote to clarify back-to-back execution of certain multiply and multiply-accumulate instructions	Table 16-19 on page 16-28
Added Note to clarify operation of Advanced SIMD floating-point instructions	<i>Advanced SIMD floating-point instructions</i> on page 16-32
Expanded description of VFP instruction execution	<i>VFP instruction execution in the NFP pipeline</i> on page 16-48

**Table C-2 Differences between issue G and issue H**

<b>Change</b>	<b>Location</b>
Amended function description for invalidate instruction cache line	Table 3-73 on page 3-90



**Table C-3 Differences between issue H and issue I**

<b>Change</b>	<b>Location</b>	<b>Affects</b>
Updated Product revisions information	<i>Product revisions</i> on page 1-14	All revisions
Updated Main ID Register to reflect revision change	<ul style="list-style-type: none"> <li>• Table 3-3 on page 3-9</li> <li>• <i>c0, Main ID Register</i> on page 3-25.</li> </ul>	r3p0
Modified description for bits [7:4] and [3:0] in the Debug ID Register	Table 12-11 on page 12-19	All revisions
Modified description for the Revision field in the Peripheral Identification Registers	Table 12-45 on page 12-67	All revisions
Updated description and reset value for bits [7:4] in the Peripheral ID Register 3	<ul style="list-style-type: none"> <li>• Table 12-49 on page 12-68</li> <li>• Table 14-2 on page 14-8</li> <li>• Table 14-6 on page 14-14</li> <li>• Table 15-3 on page 15-10</li> <li>• Table 15-30 on page 15-31.</li> </ul>	r3p0
Updated value for bits [3:0] of the FPSID Register	Table 13-7 on page 13-14	r3p0
Changed the name for trigger input 0 from DBGTRIGGER to Debug entry.	<ul style="list-style-type: none"> <li>• Table 15-1 on page 15-6</li> <li>• Table 15-26 on page 15-29.</li> </ul>	r3p0
Added text to clarify description for trigger input 0	Table 15-1 on page 15-6	All revisions
Added a note to clarify description for trigger outputs 0 and 8	<i>Trigger inputs and outputs</i> on page 15-6	All revisions
Updated the value and description for bits [4:0] of the CTI Device ID Register	<ul style="list-style-type: none"> <li>• Table 15-3 on page 15-10</li> <li>• <i>Device ID Register, 0xFC8</i> on page 15-30</li> <li>• Table 15-28 on page 15-30.</li> </ul>	All revisions
Added text to clarify description for bit [0] of the CTI Interrupt Acknowledge Register	Table 15-5 on page 15-14	All revisions
Updated description for the load data miss replay event	Table 16-16 on page 16-18	All revisions
Clarified description of dual issue for Advanced SIMD instructions	<i>Dual issue for Advanced SIMD instructions</i> on page 16-22	All revisions



# Glossary

This glossary describes some of the terms used in technical documents from ARM.

- Abort** A mechanism that indicates to a core that the value associated with a memory access is invalid. 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 or Data Abort, and an internal or 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.
- Addressing modes** A mechanism, shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.
- Advanced eXtensible Interface (AXI)** A bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple

outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure. The AXI protocol also includes optional extensions to cover signaling for low-power operation.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

**Advanced High-performance Bus (AHB)**

A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave IP developments and ARM recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

*See also* Advanced Microcontroller Bus Architecture and AHB-Lite.

**Advanced Microcontroller Bus Architecture (AMBA)**

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that describes a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

**Advanced Peripheral Bus (APB)**

A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

**AHB**

*See* Advanced High-performance Bus.

**Aligned**

A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

**AMBA**

*See* Advanced Microcontroller Bus Architecture.

**Advanced Trace Bus (ATB)**

A bus used by trace devices to share CoreSight capture resources.

**APB**

*See* Advanced Peripheral Bus.

**Application Specific Integrated Circuit (ASIC)**

An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

**Architecture**

The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv7 architecture.

**Arithmetic instruction**

Any VFPv3 *Coprocessor Data Processing* (CDP) instruction except FCPY, FABS, and FNEG.

*See also* CDP instruction.

**ARM instruction**

A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

**ARM state**

A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.

**ASIC**

*See* Application Specific Integrated Circuit.

**ATB**

*See* Advanced Trace Bus.

**ATPG**

*See* Automatic Test Pattern Generation.

**Automatic Test Pattern Generation (ATPG)**

The process of automatically generating manufacturing test vectors for an ASIC design, using a specialized software tool.

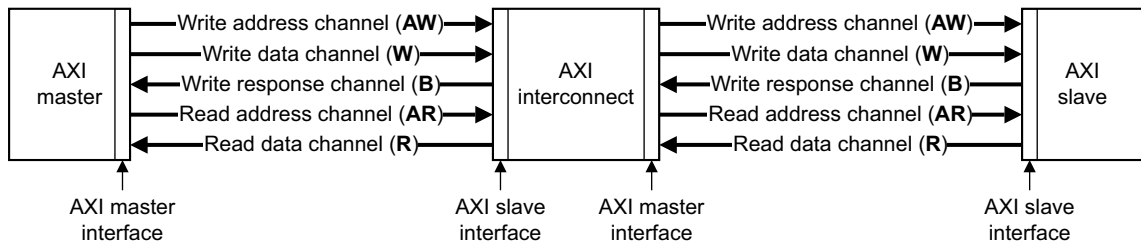
**AXI**

*See* Advanced eXtensible Interface.

**AXI channel order and interfaces**

The block diagram shows:

- the order in which AXI channel signals are described
- the master and slave interface conventions for AXI components.



## AXI terminology

The following AXI terms are general. They apply to both masters and slaves:

### Active read transaction

A transaction for which the read address has transferred, but the last read data has not yet transferred.

### Active transfer

A transfer for which the **xVALID**<sup>1</sup> handshake has asserted, but for which **xREADY** has not yet asserted.

### Active write transaction

A transaction for which the write address or leading write data has transferred, but the write response has not yet transferred.

### Completed transfer

A transfer for which the **xVALID/xREADY** handshake is complete.

**Payload** The non-handshake signals in a transfer.

**Transaction** An entire burst of transfers, comprising an address, one or more data transfers and a response transfer (writes only).

**Transmit** An initiator driving the payload and asserting the relevant **xVALID** signal.

**Transfer** A single exchange of information. That is, with one **xVALID/xREADY** handshake.

The following AXI terms are master interface attributes. To obtain optimum performance, they must be specified for all components with an AXI master interface:

### Combined issuing capability

The maximum number of active transactions that a master interface can generate. This is specified instead of write or read issuing capability for master interfaces that use a combined storage for active write and read transactions.

---

1. The letter **x** in the signal name denotes an AXI channel as follows:

<b>AW</b>	Write address channel.
<b>W</b>	Write data channel.
<b>B</b>	Write response channel.
<b>AR</b>	Read address channel.
<b>R</b>	Read data channel.

**Read ID capability**

The maximum number of different **ARID** values that a master interface can generate for all active read transactions at any one time.

**Read ID width**

The number of bits in the **ARID** bus.

**Read issuing capability**

The maximum number of active read transactions that a master interface can generate.

**Write ID capability**

The maximum number of different **AWID** values that a master interface can generate for all active write transactions at any one time.

**Write ID width**

The number of bits in the **AWID** and **WID** buses.

**Write interleave capability**

The number of active write transactions for which the master interface is capable of transmitting data. This is counted from the earliest transaction.

**Write issuing capability**

The maximum number of active write transactions that a master interface can generate.

**Banked registers**

Those physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.

**Base register**

A register specified by a load or store instruction that is used to hold the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

**Base register write-back**

Updating the contents of the base register used in an instruction target address calculation so that the modified address is changed to the next higher or lower sequential address in memory. This means that it is not necessary to fetch the target address for successive instruction transfers and enables faster burst accesses to sequential memory.

**Beat**

Alternative word for an individual transfer within a burst. For example, an INCR4 burst comprises four beats.

*See also* Burst.

**Big-endian** Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

*See also* Little-endian and Endianness.

**Big-endian memory** Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

*See also* Little-endian memory.

**Block address** An address that comprises a tag, an index, and a word field. The tag bits identify the way that contains the matching cache entry for a cache hit. The index bits identify the set being addressed. The word field contains the word address that can be used to identify specific words, halfwords, or bytes within the cache entry.

*See also* Cache terminology diagram on the last page of this glossary.

**Branch prediction** The process of predicting if branches are to be taken or not in pipelined processors. Successfully predicting if branches are to be taken enables the processor to prefetch the instructions following a branch before the branch is fully resolved. Branch prediction can be done in software or by using custom hardware. Branch prediction techniques are categorized as static, in which the prediction decision is decided before run time, and dynamic, in which the prediction decision can change during program execution.

**Breakpoint** 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.

*See also* Watchpoint.

**Burst** A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AMBA are controlled using signals to indicate the length of the burst and how the addresses are incremented.

*See also* Beat.

**Byte** An 8-bit data item.



- Byte-invariant** In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access. The ARM architecture supports byte-invariant systems in ARMv6 and later versions. When byte-invariant support is selected, unaligned halfword and word memory accesses are also supported. Multi-word accesses are expected to be word-aligned.
- See also* Word-invariant.
- Byte lane strobe** A signal that is used for unaligned or mixed-endian data accesses to determine which byte lanes are active in a transfer. One bit of this signal corresponds to eight bits of the data bus.
- Cache** 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 increase the average speed of memory accesses and so improve processor performance.
- See also* Cache terminology diagram on the last page of this glossary.
- Cache hit** A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.
- Cache line** The basic unit of storage in a cache. It is always a power of two words in size (usually four or eight words), and is required to be aligned to a suitable memory boundary.
- See also* Cache terminology diagram on the last page of this glossary.
- Cache line index** The number associated with each cache line in a cache way. Within each cache way, the cache lines are numbered from 0 to (set associativity) -1.
- See also* Cache terminology diagram on the last page of this glossary.
- Cache lockdown** 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.
- Cache miss** 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.
- Cache set** A cache set is a group of cache lines (or blocks). A set contains all the ways that can be addressed with the same index. The number of cache sets is always a power of two.
- See also* Cache terminology diagram on the last page of this glossary.

- Cache way** A group of cache lines (or blocks). It is 2 to the power of the number of index bits in size.  
*See also* Cache terminology diagram on the last page of this glossary.
- CAM** *See* Content Addressable Memory.
- Cast out** *See* Victim.
- CDP instruction** Coprocessor data processing instruction. For the VFP coprocessor, CDP instructions are arithmetic instructions and FCPY, FABS, and FNEG.  
*See also* Arithmetic instruction.
- Clean** A cache line that has not been modified while it is in the cache is said to be clean. To clean a cache is to write dirty cache entries into main memory. If a cache line is clean, it is not written on a cache miss because the next level of memory contains the same data as the cache.  
*See also* Dirty.
- Clock gating** Gating a clock signal for a macrocell with a control signal and using the modified clock that results to control the operating state of the macrocell.
- Coherency** *See* Memory coherency.
- Cold reset** Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required.  
*See also* Warm reset.
- Communications channel**  
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 ARMv7 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.
- Conditional execution**  
If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.
- Content Addressable Memory (CAM)**  
Memory that is identified by its contents. Content Addressable Memory is used in CAM-RAM architecture caches to store the tags for cache entries.

CAM includes comparison logic with each bit of storage. A data value is broadcast to all words of storage and compared with the values there. Words that match are flagged in some way. Subsequent operations can then work on flagged words. It is possible to read the flagged words out one at a time or write to certain bit positions in all of them.

- Context** The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.
- See also* Fast context switch.
- Control bits** The bottom eight bits of a Program Status Register. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.
- Coprocessor** 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.
- Copy back** *See* Write-back.
- Core** A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.
- Core reset** *See* Warm reset.
- CoreSight** The infrastructure for monitoring, tracing, and debugging a complete system on chip.
- CPSR** *See* Current Program Status Register
- Cross Trigger Interface (CTI)** Part of an Embedded Cross Trigger device. The CTI provides the interface between a core/ETM and the CTM within an ECT.
- Cross Trigger Matrix (CTM)** The CTM combines the trigger requests generated from CTIs and broadcasts them to all CTIs as channel triggers within an Embedded Cross Trigger device.
- CTI** *See* Cross Trigger Interface.
- CTM** *See* Cross Trigger Matrix.
- Current Program Status Register (CPSR)** The register that holds the current operating processor status.

- Data Abort** An indication from a memory system to the core of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.
- See also* Abort, External Abort, and Prefetch Abort.
- Data cache** 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 increase the average speed of memory accesses and so improve processor performance.
- Debugger** A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.
- Default NaN mode** A mode in which all operations that result in a NaN return the default NaN, regardless of the cause of the NaN result. This mode is compliant with the IEEE 754 standard but implies that all information contained in any input NaNs to an operation is lost.
- Direct-mapped cache** A one-way set-associative cache. Each cache set consists of a single cache line, so cache look-up selects and checks a single cache line.
- Direct Memory Access (DMA)** An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.
- Dirty** A cache line in a write-back cache that has been modified while it is in the cache is said to be dirty. A cache line is marked as dirty by setting the dirty bit. If a cache line is dirty, it must be written to memory on a cache miss because the next level of memory contains data that has not been updated. The process of writing dirty data to main memory is called cache cleaning.
- See also* Clean.
- DMA** *See* Direct Memory Access.
- DNM** *See* Do Not Modify.
- Do Not Modify (DNM)** In Do Not Modify fields, the value must not be altered by software. DNM fields read as Unpredictable values, and must only be written with the same value read from the same field on the same processor. DNM fields are sometimes followed by RAZ or RAO in parentheses to show which way the bits must read for future compatibility, but programmers must not rely on this behavior.

<b>Double-precision value</b>	Consists of two 32-bit words that must appear consecutively in memory and must both be word-aligned, and that is interpreted as a basic double-precision floating-point number according to the IEEE 754-1985 standard.
<b>Doubleword</b>	A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.
<b>Doubleword-aligned</b>	A data item having a memory address that is divisible by eight.
<b>ECT</b>	<i>See</i> Embedded Cross Trigger.
<b>Embedded Cross Trigger (ECT)</b>	The ECT is a modular component to support the interaction and synchronization of multiple triggering events with an SoC.
<b>EmbeddedICE-RT</b>	The JTAG-based hardware provided by debuggable ARM processors to aid debugging in real-time.
<b>Embedded Trace Buffer</b>	The ETB provides on-chip storage of trace data using a configurable sized RAM.
<b>Embedded Trace Macrocell (ETM)</b>	A hardware macrocell that, when connected to a processor core, outputs instruction and data trace information on a trace port. The ETM provides processor driven trace through a trace port compliant to the ATB protocol.
<b>Endianness</b>	Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.  <i>See also</i> Little-endian and Big-endian
<b>ETB</b>	<i>See</i> Embedded Trace Buffer.
<b>ETM</b>	<i>See</i> Embedded Trace Macrocell.
<b>Event</b>	1 (Simple) An observable condition that can be used by an ETM to control aspects of a trace.  2 (Complex) A boolean combination of simple events that is used by an ETM to control aspects of a trace.
<b>Exception</b>	A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

**Exception service routine**

*See* Interrupt handler.

**Exception vector**

*See* Interrupt vector.

**Exponent**

The component of a floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number.

**External Abort**

An indication from an external memory system to a core that the value associated with a memory access is invalid. 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**

In a multitasking system, the point at which the time-slice allocated to one process stops and the one for the next process starts. If processes are switched often enough, they can appear to a user to be running in parallel, in addition to being able to respond quicker to external events that might affect them.

In ARM processors, a fast context switch is caused by the selection of a non-zero PID value to switch the context to that of the next process. A fast context switch causes each Virtual Address for a memory access, generated by the ARM processor, to produce a Modified Virtual Address that is sent to the rest of the memory system to be used in place of a normal Virtual Address. For some cache control operations Virtual Addresses are passed to the memory system as data. In these cases no address modification takes place.

*See also* Fast Context Switch Extension.

**Fast Context Switch Extension (FCSE)**

An extension to the ARM architecture that 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.

*See also* Fast context switch.

**FCSE**

*See* Fast Context Switch Extension.

**Fd**

The destination register and the accumulate value in triadic operations. Sd for single-precision operations and Dd for double-precision.

**Flat address mapping**

A system of organizing memory in which each Physical Address contained within the memory space is the same as its corresponding Virtual Address.

**Flush-to-zero mode**

In this mode, the VFP coprocessor treats the following values as positive zeros:

- arithmetic operation inputs that are in the subnormal range for the input precision

- arithmetic operation results, other than computed zero results, that are in the subnormal range for the input precision before rounding.

The VFP coprocessor does not interpret these values as subnormal values or convert them to subnormal values.

The subnormal range for the input precision is  $-2^{E_{min}} < x < 0$  or  $0 < x < 2^{E_{min}}$ .

<b>Fm</b>	The second source operand in dyadic or triadic operations. Sm for single-precision operations and Dm for double-precision
<b>Fn</b>	The first source operand in dyadic or triadic operations. Sn for single-precision operations and Dn for double-precision.
<b>Formatter</b>	The formatter is an internal input block in the ETB and TPIU that embeds the trace source ID within the data to create a single trace stream.
<b>Fraction</b>	The floating-point field that lies to the right of the implied binary point.
<b>Fully-associative cache</b>	A cache that has one cache set that consists of the entire cache. The number of cache entries is the same as the number of cache ways.  <i>See also</i> Direct-mapped cache.
<b>Halfword</b>	A 16-bit data item.
<b>Halting debug-mode</b>	One of two mutually exclusive debug modes. In halting debug-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 using the debug port.  <i>See also</i> Monitor debug-mode.
<b>High vectors</b>	Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.
<b>Hit-Under-Miss (HUM)</b>	A buffer that enables program execution to continue, even though there has been a data miss in the cache.
<b>Host</b>	A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.
<b>HUM</b>	<i>See</i> Hit-Under-Miss.

<b>IEEE 754 standard</b>	<i>IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985.</i> The standard that defines data types, correct operation, exception types and handling, and error bounds for floating-point systems. Most processors are built in compliance with the standard in either hardware or a combination of hardware and software.
<b>IGN</b>	<i>See Ignore.</i>
<b>Ignore (IGN)</b>	Must ignore memory writes.
<b>Illegal instruction</b>	An instruction that is architecturally Undefined.
<b>IMB</b>	<i>See Instruction Memory Barrier.</i>
<b>Implementation-defined</b>	The behavior is not architecturally defined, but is defined and documented by individual implementations.
<b>Implementation-specific</b>	The behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.
<b>Index</b>	<i>See Cache index.</i>
<b>Infinity</b>	In the IEEE 754 standard format to represent infinity, the exponent is the maximum for the precision and the fraction is all zeros.
<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 increase the average speed of memory accesses and so improve processor performance.
<b>Instruction cycle count</b>	The number of cycles for which an instruction occupies the Execute stage of the pipeline.
<b>Instruction Memory Barrier (IMB)</b>	An operation to ensure that the prefetch buffer is flushed of all out-of-date instructions.
<b>Internal scan chain</b>	A series of registers connected together to form a path through a device, used during production testing to import test patterns into internal nodes of the device and export the resulting values.
<b>Interrupt handler</b>	A program that control of the processor is passed to when an interrupt occurs.
<b>Interrupt vector</b>	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 handler.



<b>Invalidate</b>	To mark a cache line as being not valid by clearing the valid bit. This must be done whenever the line does not contain a valid cache entry. For example, after a cache flush all lines are invalid.
<b>Jazelle architecture</b>	The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Java state to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications is present. When in Java state, the processor fetches and decodes Java bytecodes and maintains the Java operand stack.
<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>LE</b>	Little endian view of memory in both byte-invariant and word-invariant systems. <i>See</i> also Byte-invariant, Word-invariant.
<b>Line</b>	<i>See</i> Cache line.
<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>Little-endian memory</b>	Memory in which: <ul style="list-style-type: none"> <li>• a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address</li> <li>• a byte at a halfword-aligned address is the least significant byte within the halfword at that address.</li> </ul> <i>See also</i> Big-endian memory.
<b>Load/store architecture</b>	A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.
<b>Load Store unit (LS)</b>	The part of a processor that handles load and store transfers.
<b>LS</b>	<i>See</i> Load Store unit.

- Macrocell** A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.
- Memory bank** One of two or more parallel divisions of interleaved memory, usually one word wide, that enable reads and writes of multiple words at a time, rather than single words. All memory banks are addressed simultaneously and a bank enable or chip select signal determines which of the banks is accessed for each transfer. Accesses to sequential word addresses cause accesses to sequential banks. This enables the delays associated with accessing a bank to occur during the access to its adjacent bank, speeding up memory transfers.
- Memory coherency** A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.
- Memory Management Unit (MMU)** Hardware that controls caches and access permissions to blocks of memory, and translates virtual addresses to physical addresses.
- Microprocessor** *See* Processor.
- Miss** *See* Cache miss.
- MMU** *See* Memory Management Unit.
- Modified Virtual Address (MVA)** A Virtual Address produced by the ARM processor can be changed by the current Process ID to provide a *Modified Virtual Address* (MVA) for the MMUs and caches.  
  
*See also* Fast Context Switch Extension.
- Monitor debug-mode** One of two mutually exclusive debug modes. In Monitor debug-mode the 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.  
  
*See also* Halting debug-mode.
- MVA** *See* Modified Virtual Address.
- NaN** Not a number. A symbolic entity encoded in a floating-point format that has the maximum exponent field and a nonzero fraction. An SNaN sets the *Invalid Operation Cumulative* (IOC) flag if used in an arithmetic instruction and the instruction returns a QNaN. A QNaN propagates through almost every arithmetic operation without signaling exceptions and has a most significant fraction bit of one.

<b>PA</b>	See Physical Address.
<b>Penalty</b>	The number of cycles in which no useful Execute stage pipeline activity can occur because an instruction flow is different from that assumed or predicted.
<b>Power-on reset</b>	See Cold reset.
<b>Prefetching</b>	In pipelined processors, the process of fetching instructions from memory to fill up the pipeline before the preceding instructions have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.
<b>Prefetch Abort</b>	An indication from a memory system to the core that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.  See also Data Abort, External Abort and Abort.
<b>Processor</b>	A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.
<b>Physical Address (PA)</b>	The MMU performs a translation on <i>Modified Virtual Addresses</i> (MVA) to produce the <i>Physical Address</i> (PA) that is given to the AMBA bus to perform an external access. The PA is also stored in the data cache to avoid the necessity for address translation when data is cast out of the cache.  See also Fast Context Switch Extension.
<b>RAZ</b>	See Read-As-Zero.
<b>Read-As-Zero (RAZ)</b>	Appear as zero when read.
<b>Read</b>	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.
<b>RealView ICE</b>	A system for debugging embedded processor cores using a JTAG interface.
<b>Region</b>	A partition of instruction or data memory space.

- Remapping** Changing the address of physical memory or devices after the application has started executing. This is typically done to permit RAM to replace ROM when the initialization has been completed.
- 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 read as 0.
- Rounding mode** The IEEE 754 standard requires all calculations to be performed as if to an infinite precision. For example, a multiply of two single-precision values must accurately calculate the significand to twice the number of bits of the significand. To represent this value in the destination precision, rounding of the significand is often required. The IEEE 754 standard specifies four rounding modes.
- In round-to-nearest mode, the result is rounded at the halfway point, with the tie case rounding up if it would clear the least significant bit of the significand, making it even.
- Round-towards-zero mode chops any bits to the right of the significand, always rounding down, and is used by the C, C++, and Java languages in integer conversions.
- Round-towards-plus-infinity mode and round-towards-minus-infinity mode are used in interval arithmetic.
- RunFast mode** RunFast mode speeds up floating-point computations by flushing subnormal inputs and outputs to zero, and enabling the default NaN mode.
- Saved Program Status Register (SPSR)** The register that holds the CPSR of the task immediately before the exception occurred that caused the switch to the current mode.
- SBO** *See* Should-Be-One.
- SBZ** *See* Should-Be-Zero.
- SBZP** *See* Should-Be-Zero or Preserved.
- Scalar operation** A VFP coprocessor operation involving a single source register and a single destination register.
- See also* Vector operation.
- Scan chain** A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

**Set** *See* Cache set.

**Set-associative cache**

In a set-associative cache, lines can only be placed in the cache in locations that correspond to the modulo division of the memory address by the number of sets. If there are  $n$  ways in a cache, the cache is termed  $n$ -way set-associative. The set-associativity can be any number greater than or equal to 1 and is not restricted to being a power of two.

**Short vector operation**

A VFP coprocessor operation involving more than one destination register and perhaps more than one source register in the generation of the result for each destination.

**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.

**Should-Be-Zero or Preserved (SBZP)**

Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

**Significand**

The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of the implied binary point and a fraction field to the right.

**SPSR**

*See* Saved Program Status Register

**Stride**

The stride field, FPSCR[21:20], specifies the increment applied to register addresses in short vector operations. A stride of 00, specifying an increment of +1, causes a short vector operation to increment each vector register by +1 for each iteration, while a stride of 11 specifies an increment of +2.

**Subnormal value**

A value in the range  $(-2^{E_{\min}} < x < 2^{E_{\min}})$ , except for 0. In the IEEE 754 standard format for single-precision and double-precision operands, a subnormal value has a zero exponent and a nonzero fraction field. The IEEE 754 standard requires that the generation and manipulation of subnormal operands be performed with the same precision as normal operands.

**Synchronization primitive**

The memory synchronization primitive instructions are those instructions that are used to ensure memory synchronization. That is, the LDR{B,H,D}EX, STR{B,H,D}EX, SWP, and SWPB instructions.

- Tag** The upper portion of a block address used to identify a cache line within a cache. The block address from the CPU is compared with each tag in a set in parallel to determine if the corresponding line is in the cache. If it is, it is said to be a cache hit and the line can be fetched from cache. If the block address does not correspond to any of the tags, it is said to be a cache miss and the line must be fetched from the next level of memory.
- See also* Cache terminology diagram on the last page of this glossary.
- TCM** *See* Tightly coupled memory.
- Thumb instruction** A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.
- Thumb state** A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.
- Tightly coupled memory (TCM)**  
An area of low latency memory that provides predictable instruction execution or data load timing in cases where deterministic performance is required. TCMs are suited to holding:
- critical routines such as for interrupt handling
  - scratchpad data
  - data types whose locality is not suited to caching
  - critical data structures, such as interrupt stacks.
- Tiny** A nonzero result or value that is between the positive and negative minimum normal values for the destination precision.
- TLB** *See* Translation Look-aside Buffer.
- Trace hardware** A term for a device that contains an Embedded Trace Macrocell.
- Trace port** A port on a device, such as a processor or ASIC, used to output trace information.
- Trace Port Analyzer (TPA)**  
A hardware device that captures trace information output on a trace port. This can be a low-cost product designed specifically for trace acquisition, or a logic analyzer.
- Translation Lookaside Buffer (TLB)**  
A cache of recently used translation table entries that avoid the overhead of translation table walking on every memory access. Part of the Memory Management Unit.
- Translation table** A table, held in memory, that contains data that defines the properties of memory areas of various fixed sizes.
- Translation table walk**  
The process of doing a full translation table lookup. It is performed automatically by hardware.

<b>Trap</b>	An exceptional condition in a VFP coprocessor that has the respective exception enable bit set in the FPSCR register. The user trap handler is executed.
<b>Trigger instruction</b>	The VFP coprocessor instruction that causes a bounce at the time it is issued. A potentially exceptional instruction causes the VFP11 coprocessor to enter the exceptional state. A subsequent instruction, unless it is an FMXR or FMRX instruction accessing the FPEXC, FPINST, or FPSID register, causes a bounce, beginning exception processing. The trigger instruction is not necessarily exceptional, and no processing of it is performed. It is retried at the return from exception processing of the potentially exceptional instruction.  <i>See also</i> Bounce, Potentially exceptional instruction, and Exceptional state.
<b>Unaligned</b>	A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.
<b>Undefined</b>	Indicates an instruction that generates an Undefined instruction trap. See the <i>ARM Architecture Reference Manual</i> for more details on ARM exceptions.
<b>UNP</b>	<i>See</i> Unpredictable.
<b>Unpredictable</b>	Means that the behavior of the ETM cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is Unpredictable. Unpredictable behavior can affect the behavior of the entire system, because the ETM is capable of causing the core to enter debug state, and external outputs can be used for other purposes.
<b>Unpredictable</b>	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.
<b>VA</b>	<i>See</i> Virtual Address.
<b>Vector operation</b>	A VFP coprocessor operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination.  <i>See also</i> Scalar operation.
<b>Victim</b>	A cache line, selected to be discarded to make room for a replacement cache line that is required because of a cache miss. The way that the victim is selected for eviction is processor-specific. A victim is also known as a cast out.

**Virtual Address (VA)**

The MMU uses its translation tables to translate a Virtual Address into a Physical Address. The processor executes code at the Virtual Address, that might be located elsewhere in physical memory.

*See also* Fast Context Switch Extension, Modified Virtual Address, and Physical Address.

**Warm reset**

Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

**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 enable 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.

**Way**

*See* Cache way.

**WB**

*See* Write-back.

**Word**

A 32-bit data item.

**Word-invariant**

In a word-invariant system, the address of each byte of memory changes when switching between little-endian and big-endian operation, in such a way that the byte with address A in one endianness has address A EOR 3 in the other endianness. As a result, each aligned word of memory always consists of the same four bytes of memory in the same order, regardless of endianness. The change of endianness occurs because of the change to the byte addresses, not because the bytes are rearranged. The ARM architecture supports word-invariant systems in ARMv3 and later versions. When word-invariant support is selected, the behavior of load or store instructions that are given unaligned addresses is instruction-specific, and is in general not the expected behavior for an unaligned access. It is recommended that word-invariant systems use the endianness that produces the required byte addresses at all times, apart possibly from very early in their reset handlers before they have set up the endianness, and that this early part of the reset handler must use only aligned word memory accesses.

*See also* Byte-invariant.

**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. This is 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.
- 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.

#### Cache terminology diagram

The diagram illustrates the following cache terminology:

- block address
- cache line
- cache set
- cache way
- index

