SC3900 Flexible Vector Processor (FVP) Cluster Reference Manual

Supports
B4860
B4420
B4220

SC3900SUBRM
Rev D
03/2012

freescale™

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

Document Number: SC3900SUBRM
Rev D, 03/2012

Contents

Paragraph Number	Title	Page Number
	Audience	xxvii
	Organization	xxvii
	Suggested Reading	xxviii
	Conventions	xxviii
	Chapter 1 Introduction	
1.1	FVP subsystem and cluster features	1-1
1.2	Subsystem components	1-2
1.3	Cluster components	1-3
1.4	Internal architecture	1-4
1.4.1	SC3900 Flexible Vector Processor (FVP)	1-4
1.4.2	Instruction channel	1-5
1.4.2.1	Instruction Cache	1-6
1.4.3	Data channel	1-6
1.4.3.1	Read Data Cache	1-7
1.4.3.2	Store Gather Buffer (SGB)	1-8
1.4.4	Memory Management Unit	1-8
1.4.5	L2 Cache	1-9
1.4.6	Cache Management Engine	1-10
1.4.7	Enhanced Programmable Interrupt Controller	1-10
1.4.8	Debug and Trace Unit	1-10
1.4.9	Peripheral Bus	1-11
1.4.10	Timer	1-11
	Chapter 2 Memory Architecture	
2.1	Memory management	2-1
2.1.1	Memory protection	2-1
2.1.1.1	Benefits of memory protection	2-1
2.1.1.2	Memory protection settings	2-1
2.1.1.3	MATT miss exception	2-3
2.1.1.4	Memory-related error conditions	2-3
2.1.1.5	Guarded memory	2-4
2.1.2	Address translation	2-4
2.1.2.1	Address translation activation options	2-5
2.1.2.1.1	No translation	2-5
2.1.2.1.2	Single-mapped virtual addressing	2-6
2.1.2.1.3	Multi-mapped virtual addressing	2-7

Contents

Paragraph Number	Title	Page Number
2.1.3	Task ID and address structure	2-8
2.1.4	Task memory and system/shared task memory	2-10
2.2	Cache policies	2-10
2.2.1	Memory and peripheral address space	2-11
2.2.2	L1 Cache allocation policy	2-11
2.2.3	L2 Cache policies	2-11
2.2.3.1	Cacheable/noncacheable	2-11
2.2.3.2	Write-back policy	2-12
2.2.3.3	Write-through policy	2-12
2.3	Cache performance mechanisms	2-12
2.3.1	L1 cache performance mechanisms	2-12
2.3.1.1	L1 software prefetch	2-12
2.3.1.2	L1 hardware prefetch	2-13
2.3.2	L2 cache performance mechanisms	2-13
2.3.2.1	L2 partitioning	2-13
2.3.2.2	L2 touch support	2-13
2.3.2.3	L2 stashing	2-13
2.3.2.4	L2 locking	2-13
2.4	Cache and data coherency	2-14
2.4.1	Hardware coherency	2-14
2.4.1.1	MESI+L protocol	2-14
2.4.1.2	Coherency and caches	2-15
2.4.2	Accelerating software coherency	2-15
2.4.3	Memory access synchronization	2-16
2.4.4	Register access synchronization	2-17
2.4.5	Decorated instructions	2-17
2.4.6	Semaphore support	2-17
2.4.7	Messaging support	2-18
2.4.7.1	Sending and receiving messages	2-18
2.4.8	Program and data coherency	2-19

Chapter 3
Memory Management Unit

3.1	MMU features	3-2
3.2	MMU functional description	3-3
3.2.1	Internal architecture	3-4
3.2.2	Translation table and memory descriptors	3-5
3.2.2.1	Data MATT	3-6
3.2.2.2	Program MATT	3-8
3.2.2.3	Virtual segment base and size	3-9

Contents

Paragraph Number	Title	Page Number
3.2.2.3.1	Aligned segment programming model	3-9
3.2.2.3.2	Flexible segment programming model	3-12
3.2.2.4	Descriptors priority mechanism	3-13
3.2.2.5	MATT programming model	3-14
3.2.2.5.1	Read accesses from the MATT registers	3-15
3.2.2.5.2	Write accesses to the MATT registers	3-15
3.2.2.6	MATT configuration change indication	3-16
3.2.3	Attributes logic	3-16
3.2.4	Protection Unit	3-17
3.2.4.1	Protection violation	3-18
3.2.4.1.1	Task protection	3-18
3.2.4.1.2	Stack overrun protection	3-18
3.2.4.1.3	FVP subsystem protection	3-18
3.2.4.2	MMU errors and exceptions	3-18
3.2.4.2.1	Precise exceptions	3-18
3.2.4.2.2	Core data and program bus errors and exceptions	3-19
3.2.4.2.3	Cache commands support	3-20
3.2.4.2.4	ECC error support	3-22
3.2.4.2.5	Peripheral access size and alignment error	3-22
3.2.4.2.6	Semaphore access size and alignment error	3-23
3.2.4.2.7	Peripheral write access error	3-23
3.2.4.2.8	MMU behavior on error	3-23
3.2.5	Task ID support	3-24
3.2.5.1	System/shared task accesses	3-24
3.2.5.1.1	Access to the non-shared memory space	3-24
3.2.5.1.2	Access to shared memory space	3-24
3.2.5.1.3	RTOS access to non-shared memory space	3-24
3.2.6	MMU query mechanism	3-25
3.2.7	Multiple bank support	3-25
3.2.7.1	FVP subsystem control registers	3-27
3.2.7.2	Segment descriptors for different banks	3-27
3.3	Memory map and register definition	3-28
3.3.1	MMU Control Register (M_CR)	3-37
3.3.2	Data Violation PC Register (M_DVPC)	3-38
3.3.3	Data Violation Address Register (M_DVA)	3-39
3.3.4	Data Status Register (M_DSR)	3-40
3.3.5	Non-precise Data Violation Address Register (M_NDVR)	3-42
3.3.6	Non-precise Data Error Status Register (M_NDSR)	3-42
3.3.7	Platform Information Register (M_PIR)	3-43
3.3.8	Doorbell Level Register (M_DBL)	3-44
3.3.9	Doorbell Edge Register (M_DBE)	3-44

Contents

Paragraph Number	Title	Page Number
3.3.10	Data Exception Service Routine Address0 (M_DESRA0)	3-45
3.3.11	Data Exception Service Routine Address1 (M_DESRA1)	3-46
3.3.12	Data Exception Service Routine Select (M_DESRS)	3-46
3.3.13	Message Process ID Register (MSG_PIR)	3-48
3.3.14	Message Guest Process ID Register (MSG_GPIR)	3-48
3.3.15	Message Logical Process ID Register (MSG_LPIDR)	3-49
3.3.16	CCSR Base Address Register (CCSR_BASE)	3-49
3.3.17	Data Segment Descriptor VAL (M_DSDVAL)	3-50
3.3.18	Data Segment Descriptor MASK (M_DSDMASK)	3-50
3.3.19	Core Preload Register for Data Segment Descriptor A (M_DSDA_PL)	3-51
3.3.20	Core Preload Register for Data Segment Descriptor B (M_DSDB_PL)	3-51
3.3.21	Core Preload Register for Data Segment Descriptor C (M_DSDC_PL)	3-52
3.3.22	Core Preload Register for Data Segment Descriptor M (M_DSDM_PL)	3-52
3.3.23	Slave Preload Register for Data Segment Descriptor A (M_DSDA_PL_PB)	3-53
3.3.24	Slave Preload Register for Data Segment Descriptor B (M_DSDB_PL_PB)	3-53
3.3.25	Slave Preload Register for Data Segment Descriptor C (M_DSDC_PL_PB)	3-54
3.3.26	Slave Preload Register for Data Segment Descriptor M (M_DSDM_PL_PB)	3-54
3.3.27	Data MATT Programming Error Register (M_DMPER)	3-55
3.3.28	SGB Watermark Value (M_WMCFG)	3-55
3.3.29	Factory Debug Register (M_FDR)	3-56
3.3.30	Data Segment Descriptor Registers A (M_DSDA_<i>)	3-57
3.3.31	Data Segment Descriptor Registers B (M_DSDB_<i>)	3-59
3.3.32	Data Segment Descriptor Registers C (M_DSDC_<i>)	3-61
3.3.33	Program Violation Address Register (M_PVA)	3-62
3.3.34	Program Status Register (M_PSR)	3-63
3.3.35	Program Exception Service Routine Address0 (M_PESRA0)	3-64
3.3.36	Program Exception Service Routine Address1 (M_PESRA1)	3-64
3.3.37	Program Exception Service Routine Select (M_PESRS)	3-65
3.3.38	Program Segment Descriptor VAL (M_PSDVAL)	3-66
3.3.39	Program Segment Descriptor MASK (M_PSDMASK)	3-66
3.3.40	Core Preload Register for Program Segment Descriptor A (M_PSDA_PL)	3-67
3.3.41	Core Preload Register for Program Segment Descriptor B (M_PSDB_PL)	3-67
3.3.42	Core Preload Register for Program Segment Descriptor C (M_PSDC_PL)	3-68
3.3.43	Core Preload Register for Program Segment Descriptor M (M_PSDM_PL)	3-68
3.3.44	Slave Preload Register for Program Segment Descriptor A (M_PSDA_PL_PB)	3-69
3.3.45	Slave Preload Register for Program Segment Descriptor B (M_PSDB_PL_PB)	3-69
3.3.46	Slave Preload Register for Program Segment Descriptor C (M_PSDC_PL_PB)	3-70
3.3.47	Slave Preload Register for Program Segment Descriptor M (M_PSDM_PL_PB)	3-70
3.3.48	Program MATT Programming Error Register (M_PMPER)	3-71
3.3.49	Program Segment Descriptor Registers A (M_PSDA_<i>)	3-71
3.3.50	Program Segment Descriptor Registers B (M_PSDB_<i>)	3-74

Contents

Paragraph Number	Title	Page Number
3.3.51	Program Segment Descriptor Registers C (M_PSDC_<i>)	3-76
3.4	Default state of the MMU when exiting from reset	3-76

Chapter 4
L1 Caches

4.1	Main features of the L1 subsystem	4-5
4.1.1	Test state	4-6
4.2	Read cache	4-6
4.2.1	Functional description	4-6
4.2.1.1	Cache line replacement	4-7
4.2.1.1.1	Enhanced PLRU replacement	4-7
4.2.1.1.2	Enhanced PLRU update in case of hit or allocation	4-8
4.2.2	Cache coherency	4-9
4.2.3	Multitask support	4-10
4.2.4	Cache debug mode	4-10
4.3	SGB (Store Gather Buffer)	4-11

Chapter 5
L2 Cache

5.1	Introduction	5-1
5.2	L2 cache functional overview	5-2
5.2.1	L2 cache characteristics	5-2
5.2.2	Main features of L2 cache	5-3
5.2.3	Processing states	5-4
5.3	Functional description	5-4
5.3.1	Enable/disable cache	5-5
5.3.2	Cache global invalidate command	5-5
5.3.3	Cache global flush command	5-5
5.3.4	Cache global lock flush clear command	5-5
5.3.5	Replacement mechanism and cache partitioning scheme	5-6
5.3.5.1	Advanced programmable replacements	5-6
5.3.5.2	Support for stashing	5-6
5.3.6	Cache coherency operations	5-6
5.3.6.1	Memory barriers	5-7
5.3.6.2	Flush/Invalidate instructions	5-7
5.3.7	L2 bank arbitration priorities	5-7
5.3.8	Arbitration priorities towards CoreNet (via BIU)	5-8
5.3.9	Cache ECC support	5-8
5.4	Memory map and register definitions	5-8

Contents

Paragraph Number	Title	Page Number
5.4.1	L2 Cache Control and Status Register 0 (L2CSR0)	5-8
5.4.2	L2 Cache Control and Status Register 1 (L2CSR1)	5-10
5.4.3	L2 Configuration Register (L2CFG0)	5-11
5.4.4	L2 cache partitioning registers	5-12
5.4.4.1	L2 Cache Partitioning Identification Registers (L2PIRn)	5-13
5.4.4.2	L2 Cache Partitioning Allocation Registers (L2PARn)	5-14
5.4.4.3	L2 Cache Partitioning Way Registers (L2PWRn)	5-15
5.4.5	L2 Cache Error Disable Register (L2ERRDIS)	5-16
5.4.6	L2 Cache Error Detect Register (L2ERRDET)	5-17
5.4.7	L2 Cache Error Interrupt Enable Register (L2ERRINTEN)	5-18
5.4.8	L2 Cache Error Control Register (L2ERRCTL)	5-19
5.4.9	L2 Cache Error Capture Address Registers (L2ERRADDR and L2ERREADDR)	5-20
5.4.10	L2 Cache Error Capture Data Registers (L2CAPTDATALO and L2CAPTDATAHI)	5-20
5.4.11	L2 Cache Capture ECC Syndrome Register (L2CAPTECC)	5-20
5.4.12	L2 Cache Error Attribute Register (L2ERRATTR)	5-20
5.4.13	L2 Cache Error Injection Control Register (L2ERRINJCTL)	5-21
5.4.14	L2 Cache Error Injection Mask Registers (L2ERRINJLO and L2ERRINJHI)	5-22

Chapter 6
Cache Management

6.1	Cache management overview	6-1
6.2	Functional overview of the CME	6-1
6.2.1	Understanding cache control operations	6-2
6.2.1.1	Which cache instructions are implemented by the CME	6-2
6.2.1.2	How the channels access the cache	6-4
6.2.2	Main features of the CME	6-5
6.2.3	Considerations when entering processing states	6-6
6.3	Functional description of the CME	6-7
6.3.1	DCache maintenance instruction generation overview	6-7
6.3.2	CME block programming	6-7
6.3.2.1	Core block cache maintenance instructions	6-8
6.3.2.2	External CME programming	6-8
6.3.2.3	CME block channels organization and control	6-9
6.3.3	Query instructions	6-10
6.3.4	Using the external query channel	6-10
6.3.5	Using the debug channel	6-11
6.3.6	Understanding the arbitration logic	6-11
6.3.7	Next address adder	6-12

Contents

Paragraph Number	Title	Page Number
6.3.8	Core bus insertion mechanism	6-12
6.3.8.1	ICache maintenance instructions insertion	6-12
6.3.8.2	DCache maintenance instructions insertion	6-13
6.3.9	Maintenance instruction completion events	6-13
6.3.10	Understanding MMU Errors	6-14
6.4	Detailed block cache maintenance instructions description	6-14
6.4.1	Block software prefetch maintenance instructions	6-14
6.4.2	L2 cache lock management maintenance instructions	6-15
6.4.3	Coherency accelerator block instructions	6-15
6.5	Memory map and register definitions	6-16
6.5.1	CME Control Register (CME_CTR)	6-19
6.5.2	CME Debug Channel Control Programming Register (CME_DCC)	6-20
6.5.3	CME Debug Channel Address Programming Register (CME_DCA)	6-21
6.5.4	CME Debug Status Register (CME_DCR)	6-22
6.5.5	CME Debug Query Result Register 1 (CME_DQU1)	6-23
6.5.6	CME Debug Query Result Register 2 (CME_DQU2)	6-23
6.5.7	CME External Query Status Register (CME_QCR)	6-25
6.5.8	CME External Query Control Programming Register (CME_QCC)	6-26
6.5.9	CME External Query Address Programming Register (CME_QCA)	6-27
6.5.10	CME External Query Result Register 1 (CME_QU1)	6-27
6.5.11	CME External Query Result Register 2 (CME_QU2)	6-28
6.5.12	CME Block Control Programming Register (CME_CC)	6-28
6.5.13	CME Block Stride Programming Register (CME_CS)	6-29
6.5.14	CME Block Address Programming Register (CME_CA)	6-31
6.5.15	CME Block Programming Status Register (CME_CR)	6-32
6.5.16	CME Data Status Register (CME_DST)	6-33
6.5.17	CME Data Error Register (CME_DER)	6-33
6.5.18	CME Data Interrupt Status Register (CME_DIN)	6-34
6.5.19	CME Data External Doorbell Interrupt Status Register (CME_DMS)	6-35
6.5.20	CME Data Channel Control Register (CME_DC<i>)	6-35
6.5.21	CME Data Channel Stride Register (CME_DS<i>)	6-36
6.5.22	CME Data Channel Address Register (CME_DA<i>)	6-37
6.5.23	CME Program Status Register (CME_PST)	6-38
6.5.24	CME Program Error Register (CME_PER)	6-39
6.5.25	CME Program Interrupt Status Register (CME_PIN)	6-40
6.5.26	CME Program External Doorbell Interrupt Status Register (CME_PMS)	6-41
6.5.27	CME Program Channel Control Register (CME_PC<i>)	6-41
6.5.28	CME Program Channel Stride Register (CME_PS<i>)	6-42
6.5.29	CME Program Channel Address Register (CME_PA<i>)	6-43

Contents

Paragraph Number	Title	Page Number
	Chapter 7 Interrupts	
7.1	EPIC features	7-2
7.2	EPIC functional description	7-3
7.3	Memory map and registers	7-4
7.3.1	DSP subsystem interrupt table	7-4
7.3.1.1	DSP subsystem interrupt address	7-6
7.3.2	EPIC register summary	7-6
7.3.3	Register descriptions	7-11
7.3.3.1	EPIC Interrupt Priority Level Registers (P_IPL*x*)	7-11
7.3.3.2	EPIC Interrupt Dispatcher Selector Register (P_DISP*x*)	7-14
7.3.3.3	EPIC Interrupt Dispatcher Target Register (P_TRG_*x*)	7-16
7.3.3.4	EPIC Edge/Level Trigger Registers (P_ELR_*x*)	7-16
7.3.3.5	EPIC Interrupt Pending Registers (P_IPR_*x*)	7-19
7.3.3.6	EPIC Enable/Disable Interrupts Registers (P_ENDIS_*x*)	7-20
7.3.3.7	EPIC SW induced interrupt Register (P_SWII)	7-21
7.3.3.8	EPIC Disable Interrupts Register (P_DI)	7-21
	Chapter 8 Debug and Trace Support	
8.1	Overview of the debug and trace functions	8-1
8.1.1	Debug components at the cluster and subsystem level	8-3
8.2	Debug session management	8-5
8.2.1	DTU resource partitioning	8-6
8.2.2	Enabling and disabling the DTU	8-7
8.2.3	Debug configuration by an external host	8-8
8.2.3.1	Debug configuration when the core is halted	8-8
8.2.3.2	Debug configuration in the background while the core is running	8-9
8.2.4	Debug configuration by an on-core monitor	8-9
8.3	Debug events and filters	8-10
8.3.1	Start and stop events	8-14
8.3.2	Preciseness of synchronous direct events	8-15
8.4	Debug and trace register map	8-16
8.5	Understanding run control	8-20
8.5.1	Debug instructions	8-20
8.5.2	Debug mode	8-21
8.5.2.1	Entering and exiting debug mode	8-21
8.5.2.2	Operations supported during debug mode	8-22
8.5.3	Debug exceptions	8-22

Contents

Paragraph Number	Title	Page Number
8.5.4	Single stepping	8-23
8.5.5	Core commands	8-24
8.5.6	Debug run control registers	8-25
8.5.6.1	Core Command registers (CCR3-0)	8-25
8.5.6.2	Core Command Control Register (CCCR)	8-26
8.5.6.3	Core Command Data (CCD0–3)	8-26
8.5.6.4	PC_NEXT	8-27
8.5.6.5	Run Control Register (RCR)	8-28
8.5.6.6	Debug Mode Event Enabling Register (DMEER)	8-29
8.5.6.7	Debug Mode Reason Status Register (DMRSR)	8-30
8.5.6.8	Debug Mode Control and Status Register (DMCSR)	8-32
8.5.6.9	Debug Host Resource Reservation Register (DHRRR)	8-34
8.5.6.10	Debug Exception Event Enabling Register (DEEER)	8-36
8.5.6.11	Debug Exception Reason Status Register (DERSR)	8-37
8.5.6.12	Debug Exception Control and Status Register (DECSR)	8-38
8.5.6.13	Debug Exception Service Address Register (DESAR)	8-40
8.5.6.14	Debug Monitor Resource Reservation Register (DMRRR)	8-40
8.5.6.15	DTU Interface Control Register (DUICR)	8-42
8.5.6.16	Debug Resource Activity Status Register (DRASR)	8-44
8.5.6.17	Debug Event Status Register (DESR)	8-45
8.5.6.18	Subsystem Activity Status Register (SASR)	8-46
8.5.6.19	Debug and Trace Unit Version Register (DTUREV)	8-48
8.6	PC, Address and Data Detection	8-49
8.6.1	PC and Address Detection Groups	8-49
8.6.1.1	Address Range Detector Control Register n (ARDCRn)	8-50
8.6.1.2	Dual Exact PC Detector Control Register n (DEPCRn)	8-52
8.6.1.3	PC and Address Detector Reference Register An (PADRRAn)	8-53
8.6.1.4	PC and Address Detector Reference Register Bn (PADRRBn)	8-54
8.6.1.5	Semantics of PC and Address Detection	8-55
8.6.2	The Task ID comparator	8-55
8.6.2.1	Task ID Comparator Control Register (TIDCCR)	8-56
8.6.2.2	Task ID Comparator Reference Register 0 (TIDCRR0)	8-57
8.6.3	The Exception Detector	8-57
8.6.3.1	Exception Detector Control Register (EDCR)	8-57
8.6.3.2	Exception Detector Reference Value Register (EDRVR)	8-58
8.6.3.3	Exception Detector Mask Register (EDMR)	8-59
8.7	Indirect events	8-59
8.7.1	State bits and state change configuration	8-60
8.7.2	IEU programming model	8-63
8.7.2.1	Indirect Event Unit Control and Status Register (IECTLS)	8-64
8.7.2.2	Indirect Event Conditional Transition Configuration Registers (IECTRn)	8-65

Contents

Paragraph Number	Title	Page Number
8.7.2.3	Indirect Event Unconditional Transition Configuration Register (IEUTR)	8-67
8.7.2.4	IEU register configuration example	8-68
8.8	Event counting	8-69
8.8.1	Registers of the profiling unit	8-70
8.8.2	Triad registers	8-70
8.8.2.1	Profiling Counters Control and Status Register (PCCSR)	8-70
8.8.2.2	Profiling Triad Control Register A, B (PTCRA/B)	8-72
8.8.2.3	Summary of events counted in the profiling counters	8-75
8.8.2.4	Profiling Counter Value Registers An/Bn (PCVRAn/Bn)	8-80
8.8.2.5	Profiling Counter Snapshot Registers An/Bn (PCSRAn/Bn)	8-81
8.8.3	Reloadable counter registers	8-82
8.8.3.1	Reloadable Counter Control Register 0 (RCCR0)	8-82
8.8.3.2	Reloadable Counter Value Registers 0 (RCVR0)	8-85
8.8.3.3	Reloadable Counter Reload Registers 0 (RCRR0)	8-86
8.8.3.4	Reloadable Counter Snapshot Register 0 (RCSR0)	8-86
8.9	Understanding tracing support	8-87
8.9.1	Supported trace configurations	8-87
8.9.2	Messages and beats	8-88
8.9.3	NCS message buffers	8-89
8.9.4	Trace unit registers	8-89
8.9.4.1	Trace Control Register 1 (TC1)	8-90
8.9.4.2	Trace Status Register (TRSR)	8-91
8.9.4.3	Trace Control Register 3 (TC3)	8-92
8.9.4.4	Trace Control Register 4 (TC4)	8-93
8.9.4.5	Trace Profile Message Control Register (TPMCR)	8-94
8.9.4.6	Trace Watchpoint Mask Register (TWMSK)	8-95
8.9.4.7	Trace Overrun Control Register (TOCR)	8-95
8.9.4.8	TMDAT Image Register (TMDATI)	8-96
8.9.5	General message categories	8-97
8.9.5.1	Full program trace	8-97
8.9.5.2	Profiling trace	8-98
8.9.6	Supported trace messages	8-99
8.9.7	Timestamp counter	8-105
8.9.8	Mandatory trace messages	8-106
8.9.8.1	Debug status message	8-106
8.9.8.2	Error message	8-107
8.9.9	Ownership trace message	8-107
8.9.10	Program trace messages	8-108
8.9.10.1	Program trace—program address reporting	8-108
8.9.10.2	Program trace synchronization	8-108
8.9.10.3	Program trace—indirect branch history message	8-109

Contents

Paragraph Number	Title	Page Number
8.9.10.4	Treating RTS return address as a direct COF	8-110
8.9.10.5	Resource full message	8-111
8.9.10.6	Program trace—correlation message	8-111
8.9.10.7	I-CNT field	8-112
8.9.11	Data acquisition trace messages	8-113
8.9.12	Profiling trace messages	8-113
8.9.13	Watchpoint trace messages	8-114
8.9.14	Timestamp correlation message	8-115

Chapter 9
Interfaces

9.1	Introduction	9-1
9.2	Interfaces overview	9-1
9.3	Interfaces functional description	9-2
9.3.1	Core/Cache Interface Unit (CCIU)	9-2
9.3.2	CoreNet Bus Interface Unit (BIU)	9-3
9.3.3	Peripheral bus	9-3
9.3.3.1	Peripheral bus memory map	9-4

Chapter 10
Timers

10.1	Register summary	10-3
10.2	Timer registers	10-4
10.2.1	Timer 1 and Timer 3 Control Registers (TM_T1C and TM_T3C)	10-4
10.2.2	Timer 0 and Timer 2 Control Registers (TM_T0C and TM_T2C)	10-5
10.2.3	Timer Preload Registers (TM_TiP, i = 0,1,2,3)	10-6
10.2.4	Timer Value Registers (TM_TiV, i = 0,1,2,3)	10-7
10.2.5	Shadow Registers Control (TM_SC)	10-7
10.2.6	Timer Shadow Value Registers (TM_Si, i = 0,1)	10-8
10.2.7	WD Timer 0 control Register (WD_T0/1C)	10-9
10.2.8	Timer Preload Registers (WD_TiP, i = 0,1,2,3)	10-10
10.2.9	Shadow Registers Control (WD_SC0/1)	10-10
10.2.10	Timer Shadow Value Registers (WD_Si, i = 0,1,2,3)	10-11
10.3	Timer restrictions	10-11

Appendix A
Revision History

A.1	Changes from Revision *C* to Revision *D*	A-1

Contents

Paragraph Number	Title	Page Number
	Appendix B Code Restrictions	
B.1	General restrictions	B-1
B.2	Operation system restrictions	B-1
B.3	MMU restrictions	B-1
B.4	CME restrictions	B-2
B.5	EPIC restrictions	B-3

Figures

Figure Number	Title	Page Number
1-1	SC3900 FVP subsystem block diagram	1-3
1-2	SC3900 FVP subsystem top-level block diagram	1-4
2-1	Address translation	2-4
2-2	No translation, multitask memory partition example	2-6
2-3	Single-mapped translation, multitask memory partition example	2-7
2-4	Multi-mapped translation, multitask memory partition example	2-8
2-5	Effective, virtual, and physical addresses	2-9
3-1	Virtual and physical address ranges	3-1
3-2	MMU functional block diagram	3-4
3-3	MATT diagram	3-5
3-4	Generation of 36-bits physical address	3-5
3-5	Example of data address map	3-6
3-6	Data MATT diagram	3-7
3-7	Program MATT diagram	3-8
3-8	Descriptor priority mechanism not in use (example)	3-14
3-9	Using descriptors priority mechanism (example)	3-14
3-10	MMU attributes priority	3-17
3-11	Multibank data memory space	3-26
3-12	Single bank program memory space	3-27
3-13	MMU Control Register	3-37
3-14	Data Violation PC Register (M_DVPC)	3-38
3-15	Data Violation Address Register (M_DVA)	3-39
3-16	Data Status Register (M_DSR)	3-40
3-17	Non-precise Data Violation Address Register (M_NDVR)	3-42
3-18	Non-precise Data Error Status Register (M_NDSR)	3-42
3-19	Platform Information Register (M_PIR)	3-43
3-20	Doorbell Level Register (M_DBL)	3-44
3-21	Doorbell Edge Register (M_DBE)	3-44
3-22	Data Exception Service Routine Address0 (M_DESRA0)	3-45
3-23	Data Exception Service Routine Address1 (M_DESRA1)	3-46
3-24	Data Exception Service Routine Select (M_DESRS)	3-46
3-25	Message Process ID Register	3-48
3-26	Message Guest Process ID Register	3-48
3-27	Message Logical Process ID Register	3-49
3-28	CCSR Base Address Register	3-49
3-29	Data Segment Descriptor Value (M_DSDVAL)	3-50
3-30	Data Segment Descriptor Mask (M_DSDMASK)	3-50
3-31	Core Preload Register for Data Segment Descriptor Registers A (M_DSDA_PL)	3-51
3-32	Core Preload Register for Data Segment Descriptor Registers B (M_DSDB_PL)	3-51
3-33	Data Segment Descriptor Registers C (M_DSDC_PL)	3-52
3-34	Core Preload Register for Data Segment Descriptor M (M_DSDM_PL)	3-52

Figures

Figure Number	Title	Page Number
3-35	Slave Preload Register for Data Segment Descriptor Registers A (M_DSDA_PL_PB)	3-53
3-36	Slave Preload Register for Data Segment Descriptor Registers B (M_DSDB_PL_PB)	3-53
3-37	Slave Preload Register for Data Segment Descriptor Registers C (M_DSDC_PL_PB)	3-54
3-38	Slave Preload Register for Data Segment Descriptor M (M_DSDM_PL_PB)	3-54
3-39	Data MATT Programming Error Register	3-55
3-40	SGB Watermark Value (M_WMCFG)	3-55
3-41	Factory Debug Register (M_FDR)	3-56
3-42	Data Segment Descriptor Registers A(M_DSDA_<i>)	3-57
3-43	Data Segment Descriptor Registers C (M_DSDB_<i>)	3-59
3-44	Data Segment Descriptor Registers C (M_DSDC_<i>)	3-61
3-45	Program Violation Address Register (M_PVA)	3-62
3-46	Program Status Register (M_PSR)	3-63
3-47	Program Exception Service Routine Address0 (M_PESRA0)	3-64
3-48	Program Exception Service Routine Address1 (M_PESRA1)	3-64
3-49	Program Exception Service Routine Select (M_PESRS)	3-65
3-50	Program Segment Descriptor Value (M_PSDVAL)	3-66
3-51	Program Segment Descriptor Mask (M_PSDMASK)	3-66
3-52	Core Preload Register for Program Segment Descriptor A (M_PSDA_PL)	3-67
3-53	Core Preload Register for Program Segment Descriptor B (M_PSDB_PL)	3-67
3-54	Core Preload Register for Program Segment Descriptor C (M_PSDC_PL)	3-68
3-55	Core Preload Register for Program Segment Descriptor M (M_PSDM_PL)	3-68
3-56	Slave Preload Register for Program Segment Descriptor A (M_PSDA_PL_PB)	3-69
3-57	Slave Preload Register for Program Segment Descriptor B (M_PSDB_PL_PB)	3-69
3-58	Slave Preload Register for Program Segment Descriptor C (M_PSDC_PL_PB)	3-70
3-59	Slave Preload Register for Program Segment Descriptor M (M_PSDM_PL_PB)	3-70
3-60	Program MATT Programming Error Register	3-71
3-61	Program Segment Descriptor Registers A (M_PSDA_<i>)	3-72
3-62	Program Segment Descriptor Registers B (M_PSDB_<i>)	3-74
3-63	Program Segment Descriptor Registers C (M_PSDC_<i>)	3-76
4-1	Data channel block diagram	4-2
4-2	Instruction channel block diagram	4-4
4-3	Cache characteristics	4-6
4-4	PLRU replacement algorithm	4-8
4-5	Address comparison for extended tag match identification	4-10
5-1	L2 cache high-level block diagram	5-2
5-2	L2 Cache Control and Status Register 0 (L2CSR0)	5-8
5-3	L2 Cache Control and Status Register 1 (L2CSR1)	5-10
5-4	L2 Configuration Register (L2CFG0)	5-11
5-5	L2 Cache Partitioning Identification Registers (L2PIRn)	5-14
5-6	L2 Cache Partitioning Allocation Registers (L2PARn)	5-14

Figures

Figure Number	Title	Page Number
5-7	L2 Cache Partitioning Way Registers (L2PWRn)	5-16
5-8	L2 Cache Error Disable Register (L2ERRDIS)	5-16
5-9	L2 Cache Error Detect Register (L2ERRDET)	5-18
5-10	L2 Cache Error Interrupt Enable Register (L2ERRINTEN)	5-19
5-11	L2 Cache Error Control Register (L2ERRCTL)	5-19
5-12	L2 Cache Error Attribute Register (L2ERRATTR)	5-21
5-13	L2 Cache Error Injection Control Register (L2ERRINJCTL)	5-22
6-1	CME high-level block diagram	6-2
6-2	CME DCache maintenance instruction generation scheme	6-7
6-3	Channel state transition scheme	6-10
6-4	Two-dimensional address generation	6-12
6-5	CME Control Register (CME_CTR)	6-19
6-6	CME Debug Channel Control Programming Register (CME_DCC)	6-20
6-7	CME Debug Channel Address Programming Register (CME_DCA)	6-21
6-8	CME Debug Status Register (CME_DCR)	6-22
6-9	CME Debug Query Result Register 1 (CME_DQU1)	6-23
6-10	CME Debug Query Result Register 2 (CME_DQU2)	6-23
6-11	CME External Query Status Register (CME_QCR)	6-25
6-12	CME External Query Control Programming Register (CME_QCC)	6-26
6-13	CME External Query Address Programming Register (CME_QCA)	6-27
6-14	CME Query Result Register 1 (CME_QU1)	6-27
6-15	CME External Query Result Register 2 (CME_QU2)	6-28
6-16	CME Block Control Programming Register (CME_CC)	6-28
6-17	CME Block Stride Programming Register (CME_CS)	6-29
6-18	CME Stride Programming Register in Sequential Address Mode (One Dimensional)	6-30
6-19	CME Stride Programming Register in Two Dimensional Mode	6-30
6-20	CME Address Programming Register (CME_CA)	6-31
6-21	CME Programming Status Register (CME_CR)	6-32
6-22	CME Data Status Register (CME_DST)	6-33
6-23	CME Data Error Register (CME_DER)	6-33
6-24	CME Data Interrupt Status Register (CME_DIN)	6-34
6-25	CME Data External Doorbell Interrupt Status Register (CME_DMS)	6-35
6-26	CME Data Channel Control Register (CME_DC<i>)	6-35
6-27	CME Data Channel Stride Register (CME_DS<i>)	6-36
6-28	CME Data Channel Address Register (CME_DA<i>)	6-37
6-29	CME Program Status Register (CME_PST)	6-38
6-30	CME Program Error Register (CME_PER)	6-39
6-31	CME Program Interrupt Status Register (CME_PIN)	6-40
6-32	CME Program External Doorbell Interrupt Status Register (CME_PMS)	6-41
6-33	CME Program Channel Control Register (CME_PC<i>)	6-41
6-34	CME Program Channel Stride Register (CME_PS<i>)	6-42

Figures

Figure Number	Title	Page Number
6-35	CME Program Channel Address Register (CME_PA<i>)	6-43
7-1	EPIC block diagram	7-2
7-2	EPIC Interrupt Priority Level Register 0 (P_IPL_0)	7-11
7-3	EPIC Interrupt Priority Level Register i(P_IPL_i)	7-12
7-4	EPIC Interrupt Dispatcher Selector Register *x* (P_DISP_*x*)	7-14
7-5	EPIC Interrupt Dispatcher Selector Register *x* (P_TRG*x*)	7-16
7-6	EPIC Edge/Level Register 0 (P_ELR_0)	7-17
7-7	EPIC Edge/Level Register *x* (P_ELR_*x*)	7-17
7-8	EPIC Interrupt Pending Registers (P_IPR*x*)	7-19
7-9	EPIC Enable/Disable Interrupts Registers (P_ENDIS_*x*)	7-21
7-10	EPIC Disable Interrupts Register (P_SWII)	7-21
7-11	EPIC Disable Interrupts Register (P_DI)	7-21
8-1	SC3900 subsystem and cluster debug and trace logic block diagram	8-3
8-2	Activity states of a debug event generator	8-15
8-3	Core Command Registers (CCR3-0)	8-25
8-4	Core Command Control Register (CCCR)	8-26
8-5	Core Command Data (CCD0–3)	8-27
8-6	NEXT PC Register (PC_NEXT)	8-28
8-7	Run Control Register (RCR)	8-28
8-8	Debug Mode Event Enabling Register (DMEER)	8-29
8-9	Debug Mode Reason Status Register (DMRSR)	8-31
8-10	Debug Mode Control and Status Register (DMCSR)	8-32
8-11	Debug Host Resource Reservation Register (DHRRR)	8-34
8-12	Debug Exception Event Enabling Register (DEEER)	8-36
8-13	Debug Exception Reason Status Register (DERSR)	8-37
8-14	Debug Exception Control and Status Register (DECSR)	8-38
8-15	Debug Exception Service Address Register (DESAR)	8-40
8-16	Debug Monitor Resource Reservation Register (DMRRR)	8-41
8-17	DTU Interface Control Register (DUICR)	8-43
8-18	Debug Resource Activity Status Register (DRASR)	8-44
8-19	Debug Event Status Register (DESR)	8-45
8-20	Subsystem Activity Status Register (SASR)	8-47
8-21	Debug and Trace Unit Revision (DTUREV)	8-48
8-22	Address Range Detector Control Register n (ARDCRn)	8-50
8-23	Dual Exact PC Detector Control Register n (DEPCRn)	8-52
8-24	PC and Address Detector Reference Register An (PADRRAn)	8-54
8-25	PC and Address Detector Reference Register Bn (PADRRBn)	8-54
8-26	Task ID Comparator Control Register (TIDCCR)	8-56
8-27	Task ID Comparator Reference Register 0 (TIDCRR0)	8-57
8-28	Exception Detector Control Register (EDCR)	8-58
8-29	Exception Detector Reference Value Register (EDRVR)	8-58

Figures

Figure Number	Title	Page Number
8-30	Exception Detector Mask Register (EDMR)	8-59
8-31	Example 1: Identification of a sequence of 4 events (exclusive)	8-61
8-32	Example 2: Identification of a sequence of 4 events (pipelined)	8-61
8-33	Example 3: Two independent detection sequences	8-62
8-34	Example 4: Single input-output connection options	8-63
8-35	Indirect Event Unit Control and Status Register (IECTLS)	8-64
8-36	Indirect Event Conditional Transition Configuration Register n (IECTRn)	8-65
8-37	Indirect Event Unconditional Transition Configuration Register (IEUTR)	8-68
8-38	State transition diagram of the IEU configuration example	8-68
8-39	Profiling Counters Control and Status Register (PCCSR)	8-71
8-40	Profiling Triad Control Register A, B (PTCRA/B)	8-73
8-41	Profiling Counter Value An/Bn (PCVRAn/Bn)	8-81
8-42	Profiling Counter Snapshot An/Bn (PCSRAn/Bn)	8-81
8-43	Reloadable Counter Control Register 0/1 (RCCR0)	8-82
8-44	Reloadable Counter Value Register 0 (RCVR0)	8-85
8-45	Reloadable Counter Reload Register 0 (RCRR0)	8-86
8-46	Reloadable Counter Snapshot Register 0 (RCSR0)	8-87
8-47	Trace Control Register 1 (TC1)	8-90
8-48	Trace Status Register (TRSR)	8-91
8-49	Trace Control Register 3 (TC3)	8-92
8-50	Trace Control Register 4 (TC4)	8-94
8-51	Trace Profile Message Control Register (TPMCR)	8-94
8-52	Trace Watchpoint Mask Register (TWMSK)	8-95
8-53	Trace Overrun Control Register (TOCR)	8-96
8-54	TMDAT Image Register (TMDATI)	8-97
8-55	Bits traced in profiling messages and upon overflow	8-98
9-1	Memory buses interconnect	9-2
10-1	Timer 1 and Timer 3 Control Registers (TM_T1C and TM_T3C)	10-4
10-2	Timer 0 and Timer 2 Control Registers (TM_T0C and TM_T2C)	10-5
10-3	Timer i Preload Register (TM_TiP). i = 0, 1, 2, 3	10-6
10-4	Timer i Value Register (TM_TiV). i = 0, 1, 2, 3	10-7
10-5	Shadow Register Control (TM_SC)	10-7
10-6	Timer i Shadow Value Register (TM_Si). i=0,1	10-8
10-7	WD Timer 0 and WD Timer 1Control Registers (WD_T0C and WD_T1C)	10-9
10-8	Timer i Preload Register (WD_TiP). i = 0, 1,2,3	10-10
10-9	Shadow Register Control (WD_SC0/1)	10-10
10-10	Timer i Shadow Value Register (WD_Si). i=0,1,2,3	10-11

Tables

Table Number	Title	Page Number
2-1	Program address protection parameters	2-2
2-2	Data address protection parameters	2-2
2-3	Memory barriers supported by the SC3900 subsystem	2-16
2-4	Processor message types	2-18
3-1	Segment descriptor base and size	3-10
3-2	Physical descriptor base address	3-11
3-3	Flexible segment programming model	3-12
3-4	Default attributes	3-16
3-5	Core data bus errors	3-19
3-6	Core program bus errors	3-20
3-7	Erroneous cache command behavior	3-21
3-8	MMU Behavior for a precise error	3-23
3-9	MMU memory map	3-28
3-10	M_CR bit descriptions	3-37
3-11	M_DVPC bit descriptions	3-39
3-12	M_DVA bit descriptions	3-39
3-13	M_DSR bit descriptions	3-40
3-14	M_NDVR bit descriptions	3-42
3-15	M_NDSR Bit Descriptions	3-42
3-16	M_PIR bit descriptions	3-43
3-17	M_DBL bit descriptions	3-44
3-18	M_DBE bit descriptions	3-45
3-19	M_DESRA0 bit descriptions	3-45
3-20	M_DESRA1 bit descriptions	3-46
3-21	M_DESRS bit descriptions	3-47
3-22	MSG_PIR bit description	3-48
3-23	MSG_GPIR bit description	3-49
3-24	MSG_LPIDR bit description	3-49
3-25	CCSR_BASE bit description	3-50
3-26	M_DSDVAL bit descriptions	3-50
3-27	M_DSDMASK Bit Descriptions	3-51
3-28	M_DSDM_PL register bit description	3-52
3-29	M_DSDM_PL_PB register bit description	3-54
3-30	M_DMPER bit descriptions	3-55
3-31	SGB Watermark Value (M_WMCFG) bit description	3-56
3-32	Factory Debug Register (M_FDR) bit description	3-56
3-33	M_DSDA_<i> (0 .. 31) bit descriptions	3-57
3-34	DSVBAS region base and size	3-58
3-35	DSVBAS alignment and boundary restrictions	3-58
3-36	DAPS values and permission levels	3-59
3-37	M_DSDB_<i> (0 .. 31) bit descriptions	3-60

Tables

Table Number	Title	Page Number
3-38	DSPBA alignment and boundary restrictions	3-60
3-39	DSS alignment and boundary restrictions	3-60
3-40	M_DSDC_<i> (0 .. 31) bit descriptions	3-61
3-41	M_PVA bit descriptions	3-62
3-42	M_PSR bit descriptions	3-63
3-43	M_PESRA0 bit descriptions	3-64
3-44	M_PESRA1 bit descriptions	3-64
3-45	M_PESRS bit descriptions	3-65
3-46	M_PSDVAL bit descriptions	3-66
3-47	M_PSDMASK bit descriptions	3-67
3-48	M_PSDM_PL register bit description	3-68
3-49	M_PSDM_PL_PB register bit description	3-70
3-50	M_PMPER bit descriptions	3-71
3-51	M_PSDA_<i> bit descriptions	3-72
3-52	PSVBAS base address and size	3-73
3-53	PSVBAS alignment and boundary restriction	3-73
3-54	M_PSDB_<i> (0 .. 15) bit descriptions	3-75
3-55	M_PSDC_<i> (0 .. 15) bit descriptions	3-76
4-1	Data channel access summary	4-3
4-2	PLRU replacement way selection	4-8
4-3	Enhanced PLRU B0-B6 update on hit or allocation	4-9
5-1	L2 cache main features	5-3
5-2	L2 cache memory map	5-8
5-3	Register L2CSR0 bits description	5-9
5-4	Register L2CSR1 bits description	5-11
5-5	Register L2CFG0 bits description	5-12
5-6	L2PARn field descriptions	5-15
5-7	L2PWRn field descriptions	5-16
5-8	L2ERRDIS field descriptions	5-17
5-9	L2ERRDET field descriptions	5-18
5-10	L2ERRINTEN field descriptions	5-19
5-11	L2ERRCTL field descriptions	5-20
5-12	L2ERRATTR field descriptions	5-21
5-13	L2ERRINJCTL field descriptions	5-22
6-1	Summary of cache commands and their functionality	6-3
6-2	Block coherency accelerator summary	6-16
6-3	CME memory map	6-16
6-4	Register CME_CTR bits description	6-19
6-5	Register CME_DCC bits description	6-20
6-6	Register CME_DCA bits description	6-22

Tables

Table Number	Title	Page Number
6-7	Register CME_DCR bits description	6-22
6-8	Register CME_DQU1 bits description	6-23
6-9	Register CME_DQU2 bits description	6-24
6-10	Register CME_QCR Bits Description	6-25
6-11	Register CME_QCC bits description	6-26
6-12	Register CME_QCA bits description	6-27
6-13	Register CME_CC bits description	6-28
6-14	Register CME_CS bits description	6-30
6-15	Register CME_CA bits description	6-31
6-16	Register CME_CR bits description	6-32
6-17	Register CME_DST bits description	6-33
6-18	Register CME_DER bits description	6-34
6-19	Register CME_DIN bits description	6-34
6-20	Register CME_DMS bits description	6-35
6-21	Register CME_DC<i>(0 .. 7) bits description	6-36
6-22	Register CME_DS<i>(0 .. 7) bits description	6-37
6-23	Register CME_DA<i>(0 .. 7) bits description	6-38
6-24	Register CME_PST bits description	6-39
6-25	Register CME_PER bits description	6-40
6-26	Register CME_PIN bits description	6-40
6-27	Register CME_PMS bits description	6-41
6-28	Register CME_PC<i>(0 .. 7) bits description	6-42
6-29	Register CME_PS<i>(0 .. 7) bits description	6-43
6-30	Register CME_PA<i>(0 .. 7) bits description	6-44
7-1	DSP subsystem interrupt routing	7-5
7-2	Addresses of exception routines	7-6
7-3	EPIC register summary	7-6
7-4	Register P_IPL_<*i*> (i=1..63) bits descriptions	7-12
7-5	P_IPL_*x* interrupt group mapping	7-12
7-6	P_DISP_x bit descriptions	7-15
7-7	P_DISP_x interrupt mapping	7-15
7-8	P_TRGx bit descriptions	7-16
7-9	P_ELR_*x* bit descriptions	7-17
7-10	P_ELR_*x* mapping	7-18
8-1	Debug Event Generators in the DTU	8-4
8-2	DTU resource allocation states, per resource	8-6
8-3	Activation signals of the Debug Unit from the SoC	8-7
8-4	Debug events used as inputs for event selection	8-11
8-5	Filtering conditions for events and outcomes	8-12
8-6	Definition of the privilege level and mode filter	8-12
8-7	Outcomes of debug events	8-12

Tables

Table Number	Title	Page Number
8-8	Self-contained resource allocation in debug resource partitions	8-13
8-9	Allocation of DEBUGEV.n instructions for direct events	8-13
8-10	Allocation of PCDA detectors as filters	8-14
8-11	Preciseness of synchronous direct event combinations	8-16
8-12	General access permissions to DTU registers	8-18
8-13	Debug and trace register summary	8-19
8-14	SC3900 instructions for debug support and their affect by the DTU	8-21
8-15	Status values after single step operations	8-24
8-16	Field description of CCR3–0	8-26
8-17	Fields of the Core Command Control Register (CCCR)	8-26
8-18	Field description of CCD3–0	8-27
8-19	Field description of PC_NEXT	8-28
8-20	Field description of RCR	8-29
8-21	Field description of DMEER	8-29
8-22	Field description of DMRSR	8-31
8-23	Field description of DMCSR	8-33
8-24	Field description of DHRRR	8-35
8-25	Field description of DEEER	8-37
8-26	Field description of DERSR	8-38
8-27	Field description of DECSR	8-39
8-28	Field description of DESAR	8-40
8-29	Field description of DMRRR	8-41
8-30	Field description of DUICR	8-43
8-31	Field description of DRASR	8-44
8-32	Field description of DESR	8-46
8-33	Field description of SASR	8-47
8-34	Field description of DTUREV	8-48
8-35	Preciseness of event detection types	8-49
8-36	Field description of ARDCRn	8-51
8-37	Allocation of events for filtering, starting and stopping PC and address Detectors	8-52
8-38	Field description of DEPCRn	8-53
8-39	Field description of PADRRAn	8-54
8-40	Field description of PADRRBn	8-55
8-41	Field description of TIDCCR	8-56
8-42	Field description of TIDCRR0	8-57
8-43	Field description of EDCR	8-58
8-44	Field description of EDRVR	8-59
8-45	Field description of EDMR	8-59
8-46	Field description of IECTLS	8-64
8-47	Field description of IECTRn	8-65
8-48	Input events for the IEU	8-66

Tables

Table Number	Title	Page Number
8-49	Field description of IEUTR	8-68
8-50	Field Description of PCCSR	8-71
8-51	Field description of PTCRA/B	8-73
8-52	Allocation of events for filtering, starting and stopping profiling counters	8-74
8-53	Allocation of events for sampling and resetting profiling counters	8-74
8-54	Summary of event groups for profiling	8-76
8-55	Triad Counted Events (CEV) field values	8-80
8-56	Field description of PCVRAn/Bn	8-81
8-57	Field description of PCSRAn/Bn	8-82
8-58	Field description of RCCR0	8-83
8-59	Event allocation for filtering, start/stop and reloading a reloadable counter	8-83
8-60	Events counted by the reloadable counters	8-84
8-61	Field description of RCVR0	8-86
8-62	Field description of RCRR0	8-86
8-63	Field description of RCSR0	8-87
8-64	Supported trace combinations	8-88
8-65	Message priority at the entrance to the main message queue	8-89
8-66	Field description of TC1	8-90
8-67	Field description of TRSR	8-92
8-68	Field description of TC3	8-93
8-69	Allocation of events for filtering, starting and stopping trace messages	8-93
8-70	Field description of TC4	8-94
8-71	Field description of TPMCR	8-95
8-72	Field description of TWMSK	8-95
8-73	Field description of TOCR	8-96
8-74	Field description of TMDATI	8-97
8-75	Supported trace messages	8-99
8-76	Timestamp field (TSTAMP) format	8-105
8-77	Structure of the status field in the debug status message	8-106
8-78	Error Type (ETYPE) format	8-107
8-79	Error Code (ECODE) format	8-107
8-80	Structure of the process field in the ownership message	8-108
8-81	Hard program synchronization conditions	8-108
8-82	Indirect Branch Type (B-TYPE) format	8-110
8-83	Resources that are reported by the resource full message	8-111
8-84	Events reported in program correlation messages	8-112
8-85	Values of the SMP, SMP1 and SMP2 fields	8-114
8-86	Values of the GRP field in the watchpoint message	8-114
8-87	Address detector Watchpoint Hit (WPHIT) field formats	8-114
8-88	T-TYPE values in timestamp correlation messages	8-115
9-1	Peripheral access paths	9-3

Tables

Table Number	Title	Page Number
9-2	Cluster peripheral address map	9-4
9-3	Subsystem peripheral address map	9-5
10-1	Timer memory map	10-3
10-2	Register TM_T1C TM_T3C bits description	10-4
10-3	Register TM_T0C TM_T2C bits description	10-5
10-4	Register TM_T0P TM_T1P TM_T2P TM_T3P bits description	10-6
10-5	Register TM_T0V TM_T1V TM_T2V TM_T3V bits description	10-7
10-6	Register TM_SC Bits Description	10-8
10-7	Register TM_S0 TM_S1 bits description	10-8
10-8	Register WD_T0C WD_T1C bits description	10-9
10-9	Register WD_T0P WD_T1P WD_T2P WD_T3P bits description	10-10
10-10	Register WD_SC bits description	10-10
10-11	Register WD_S0 WD_S1 bits description	10-11

About This Book

This reference manual provides reference information for the SC3900 FVP subsystem and cluster, based on the StarCore SC3900 FVP core. The subsystem encompasses the StarCore SC3900 core and its immediate supporting units, which include instruction and data cache systems, interrupt controller, timers, interface units, and debug and profiling support units.

This manual also describes the subsystem and units supporting the core in detail, including the programmer's model and operation procedures. The SC3900 core is described only briefly; for details, see the *SC3900 FVP Core Reference Manual*.

Audience

WARNING

This document can be shared outside only with explicit approval from NMSG marketing, and under NDA.

This document is intended for technical teams from selected customers and for an internal Freescale audience, which includes system software developers, hardware designers, software tool developers and application developers.

Organization

The following is a summary and a brief description of the major parts of this reference manual:

- Chapter 1, "Introduction," summarizes subsystem features and components, and displays the functional block diagram of the SC3900 FVP subsystem.
- Chapter 2, "Memory Architecture," surveys the structure and components of the subsystem and how they interact in some detail. An example application at the system level is presented.
- Chapter 3, "Memory Management Unit," covers the features, architecture, and functionality of the MMU.
- Chapter 4, "L1 Caches," covers the features, architecture, operating states, overall functionality, and registers of the L1 cache.
- Chapter 5, "L2 Cache," covers the features, architecture, operating states, overall functionality, and registers of the L2 cache.
- Chapter 6, "Cache Management," covers the features, architecture, operating states, overall functionality, and registers of the cache management engine.
- Chapter 7, "Interrupts," covers the features, architecture, operating states, overall functionality, and registers of the advanced interrupt controller (EPIC).
- Chapter 8, "Debug and Trace Support," describes the features in the SC3900 subsystem for supporting debug and trace.

- Chapter 9, “Interfaces,” discusses the main interfaces, which are the eLink, slave, and CoreNet interfaces. After outlining the main features of each interface, the chapter considers processing states, clock ratio change, and core access timings.
- Chapter 10, “Timers,” is a chapter that briefly describes timer functionality and then presents definitions of the individual timer registers.
- Appendix A, “Revision History,” provides a list of the major differences between revisions of the *SC3900 FVP Cluster Reference Manual.*
- Appendix B, “Code Restrictions,” describes the software restrictions for the SC3900 FVP subsystem, which are in addition to the programming rules of the SC3900 listed in the *SC3900 FVP Core Reference Manual*

Suggested Reading

The following list contains suggested reading for additional information:

- *SC3900 FVP Core Reference Manual* describes the SC3900 StarCore architecture.

Conventions

This document uses certain conventions to assist in identifying, locating, and understanding information.

Table i list suffixes used to identify different numbering systems:

Table i. Suffix identification

Suffix	Meaning
b	Binary number. For example, the binary equivalent of the number 5 is written 101b.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch.

Table ii lists the notational conventions used throughout this document:

Table ii. Notational conventions

Example	Description
placeholder	Items in italics are placeholders for information that you provide. Italicized text is also used for the titles of publications and for emphasis.
code	Fixed-width type indicates text that must be typed exactly as shown. It is used for instruction mnemonics, symbols, subcommands, parameters, and operators. Fixed-width type is also used for example code.
[*n*]	A number enclosed in brackets represents a single bit in a register or in memory.
[*n:m*]	Numbers enclosed in brackets and separated by a colon represent the endpoints of a continuous range of bits in a register or in memory.
SR.SCM register.field	The way to represent a field name of a particular register. For instance, SR.SCM is the Scaling Mode (SCM) field of the Status Register (SR).

Table iii list terms that have special meanings.

Table iii. Special terms

Term	Meaning
asserted	Refers to the state of a signal as follows: • An active-high signal is asserted when high (1). • An active-low signal is asserted when low (0).
byte	An 8-bit data object
deasserted	Refers to the state of a signal as follows: • An active-high signal is deasserted when low (0). • An active-low signal is deasserted when high (1).
quad-long word	A 128-bit data object
double-long word	A 64-bit data object
long word	A 32-bit data object
word	A 16-bit data object

Chapter 1
Introduction

The SC3900 FVP subsystem is built around the StarCore SC3900 Flexible Vector Processor (FVP). Several subsystems combine with a shared L2 cache to form the SC3900 FVP cluster.

1.1 FVP subsystem and cluster features

Key features of this subsystem are as follows:

- Abstract software model
  - Address translation allows application development using virtual addresses unrelated to the actual mapping in memory. Advantages of this feature are that it makes application development easier and faster, promotes code reuse, and reduces code size in real-time applications.
  - Task and memory protection makes it easier to develop and debug multitask systems, enhances system reliability by giving higher mean time between failures (MTBF), and allows incorporation of code from external sources.
- High performance cache system
  - Cache architecture tailored for FVP algorithms
  - Hardware coherency throughout the memory system (user selectable)
  - Seamless unaligned access support
  - Hardware support for advance OS (such as Linux)
  - Advanced hardware prefetching mechanisms keep the cycle miss penalty low for a wide range of target applications
  - Background non-intrusive software prefetch, coherency accelerator, and L2 cache lock generation
  - Efficient bus protocol allowing high bandwidth
  - Advanced multicore support
  - Advanced peripheral bus connectivity with full external visibility of all peripherals
- Advanced debug and profiling support
  - SC3900 debug and trace unit (DTU) supporting PC and data address breakpoints, single stepping, injected core commands, and more.
  - Nexus compliant (IEEE ISTO 5001) tracing support from the SC3900 FVP
  - Cache array and control state (such as tags and valid bits) are readable and writable.
  - Six counters that enable counting from approximately 90 core and subsystem events.

— Advanced cross-triggering capabilities that enable flexible enabling, disabling, and filtering debug events and resources

1.2 Subsystem components

Components of the SC3900-based SC3900 FVP subsystem are as follows:

- SC3900 flexible vector processor
- Instruction channel built around a 32-Kbyte, 8 way instruction cache with a 128-byte line, which supports advanced prefetching
- Data channel built around a 32-Kbyte, 8 way read data cache with a 128-byte line, which supports advanced prefetching and a 1-Kbyte store gather buffer (SGB) with 16 entries of 64 byte each
- Memory management unit (MMU) for task protection and address translation supporting 32 data and 16 instruction descriptors
- Cache management engine (CME) providing 16 channel advanced software prefetch, coherency accelerator, and L2 cache lock features
- Extended peripheral bus grid allowing efficient internal and external access to subsystem and Cluster peripherals
- Quad timer for internal use (such as RTOS) and watchdog timer
- Advanced programmable interrupt controller (EPIC) supporting 512 interrupts
- Advanced debug support with the DTU
- NEXUS-based advanced multicore profiling support

Figure 1-1 shows the SC3900 FVP subsystem block diagram.

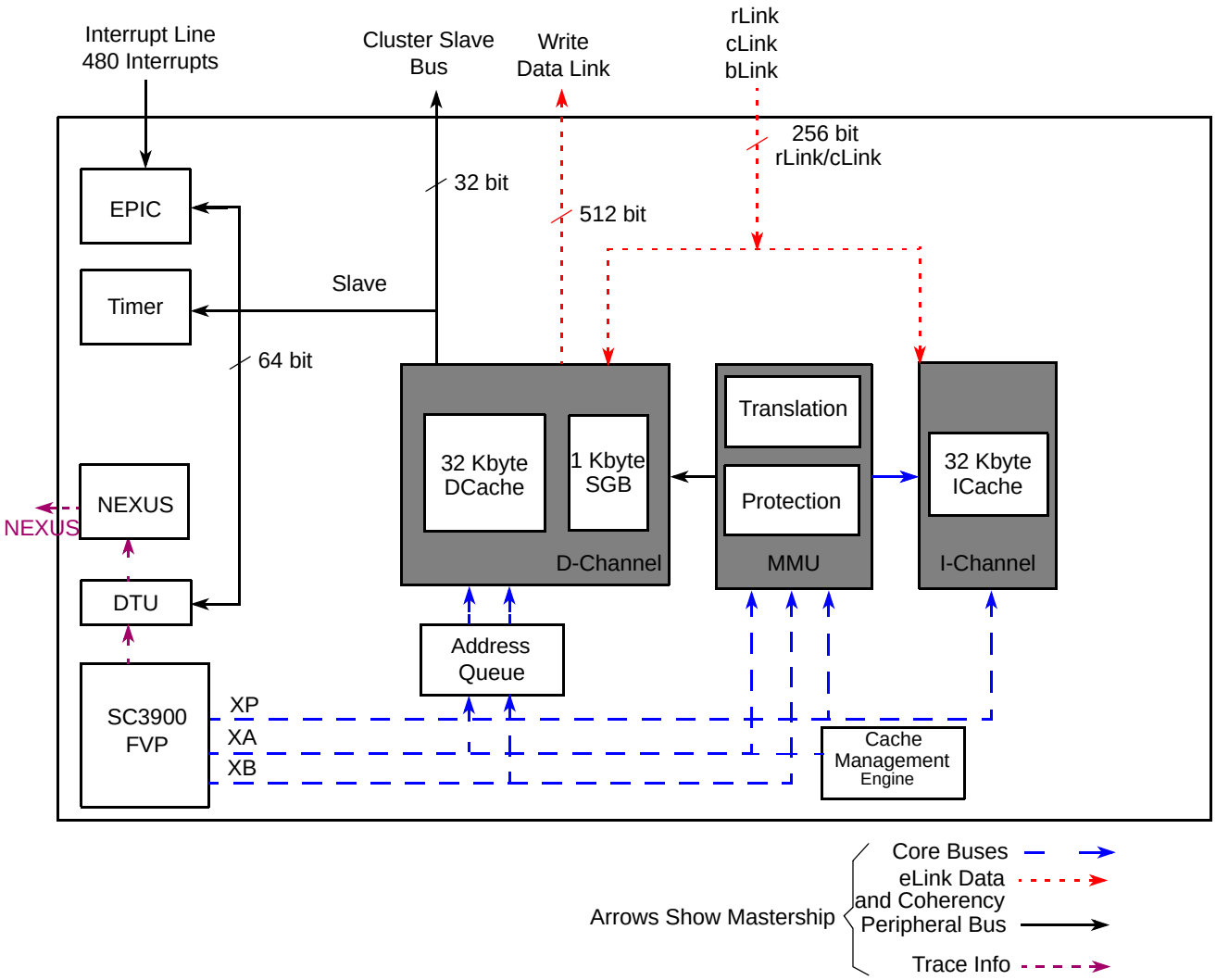

Figure 1-1. SC3900 FVP subsystem block diagram

1.3 Cluster components

The SC3900 cluster consists of several SC3900 subsystems connected to the L2 cache. The components of the SC3900 cluster are as follows:

- Two SC3900 subsystems
- Shared L2 cache
- CoreNet bus protocol interface
- Watchdog timers for each core
- Two AXI to ELINK bus converters (described in SoC documentation)

- Mesh for low latency intra-cluster control

Figure 1-2 shows the SC3900 FVP cluster top-level block diagram.

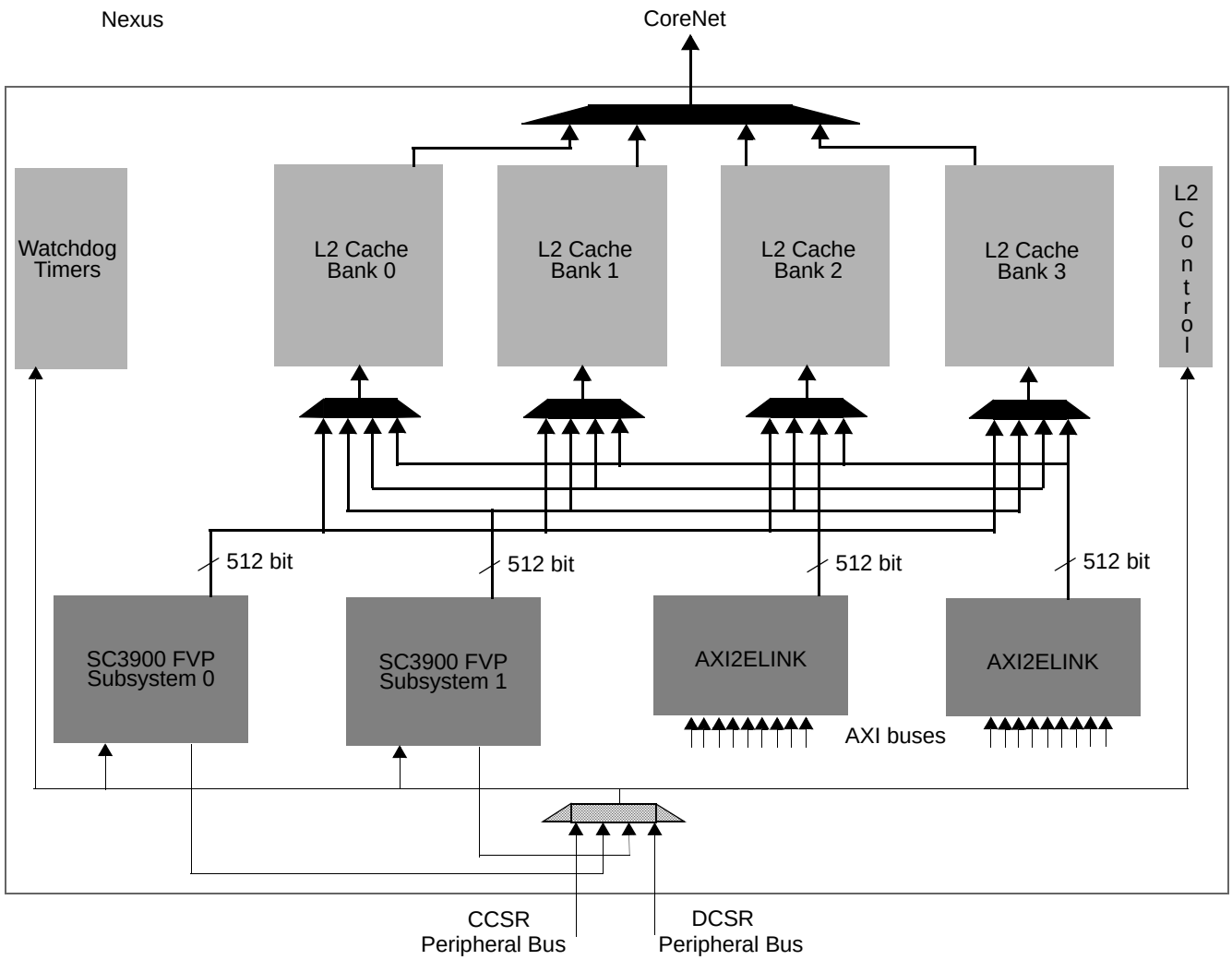

Figure 1-2. SC3900 FVP subsystem top-level block diagram

1.4 Internal architecture

Figure 1-1 depicts the functional modules and main buses of the SC3900 FVP subsystem, which are discussed in the remainder of this section.

1.4.1 SC3900 Flexible Vector Processor (FVP)

The SC3900 FVP is a flexible, programmable FVP core that handles compute-intensive communications applications and provides high performance, low power, and high code density. It is assembly-level compatible with the following DSPs:

- SC140, SC1400, SC3400 and SC3850 cores used in the single core MSC8101 and MSC8103 DSPs

- The quad-core MSC8102, MSC8122 and MSC8144 DSPs
- The single core MSC711x DSPs
- The six-core MSC8156, and MSC8157W DSP

The SC3900 FVP contains the following units:

- A data arithmetic and logic unit (DALU) with four data multiplication units (DMUs)
- An address generation unit (AGU) with two load-store units (LSU), one integer processing unit (IPU)
- A program control unit (PCU)

The core register set includes 64 data registers, each 40-bits wide (Dn), and 32 general-purpose registers, used for address calculation and integer arithmetic (Rn). In addition, the core has several other registers for status and control.

Each DMU supports eight 16-bit × 16-bit integer multipliers that can accumulate results into two 40-bit wide destination data registers. Each DMU performs eight MAC operations per clock cycle, so that a single core running at up to 1.2 GHz can perform 38.4 billion multiply-accumulates per second (GMACS). This rate is for both 16-bit operands, 8-bit operands, or mixed 8-bit and 16-bit operands. A single core can also perform up to 9.6 billion complex multiply-accumulates per second, or 9.6 billion 32 × 32 bit multiplications per second.

Each LSU in the AGU can perform one address calculation and drive one data memory access per cycle. Data access widths are flexible and can be between 8 to 512 bits wide. The AGU can support a throughput of up to 1228.8 Gbps between the core and the memory.

Arithmetic operations are performed using either fractional or integer native data types, enabling the user to choose between general C code development or to use coding techniques derived from an application-specific standard. Parts of many algorithms use data with reduced width (8 and 16 bits). For better efficiency, the SC3900 supports single instruction, multiple data (SIMD) instructions working on 16-bit, 32-bit, 64-bit, or 128-bit operands packed in a register, register pair or quad. This packing allows the core to perform 4 to 8 operations per instruction (maximum of 16 to 32 operation per VLES). In addition, the SC3900 includes special instructions to efficiently support typical FVP kernels in baseband, voice, and video applications.

For details on the SC3900 core features, see the *SC3900 FVP Core Reference Manual*.

1.4.2 Instruction channel

The instruction channel, which consists of the instruction cache (ICache), provides the core with instructions that are stored in higher-level memory. The ICache operates at core speed and stores recently accessed instructions. Whenever an addressed instruction (from the cacheable memory area) is found in the array, it is immediately made available to the core (ICache hit). When the required address is not found in the array, it is loaded to the ICache from the external (off-subsystem) memory (ICache miss). The ICache operates in parallel with the core to implement a hardware line prefetching algorithm that loads the ICache with information that has a high probability of being needed soon. This action reduces the number of cache misses. When an instruction is addressed from a noncacheable area, the ICache fetches it directly to the XP bus of the core without writing it to the cache.

1.4.2.1 Instruction Cache

The ICache has the following parameters:

- A size of 32 Kbytes with 128 bytes per cache line (total of 256 lines in the cache) arranged in an 8 way set-associative structure.

Upon a cache miss, the ICache fetches the required information. A pseudo-LRU replacement algorithm is used to select the line to be replaced when required. The programmer can flush the full contents of the ICache (meaning, invalidate all tags) or selectively flush its contents using dedicated core commands.

The hardware next line prefetch mechanism detects the presence (hit) of the next line in the cache (limited to a 4 Kbyte block). If the next line is a miss it will be prefetched to the cache depending on the MMU configuration. The hardware prefetch is a background operation that does not stall the core.

The ICache supports real-time and non-real-time debugging and profiling. In real time, the ICache provides information on misses and hits. In non-real-time, it supports access to all its internal information, namely, the tag, the replacement status, and the valid arrays. The cache array itself can be either read or written. This information is accessed through the peripheral bus interface in debug processing state.

The ICache supports a software prefetch, providing a way for programmers to preload critical program into the cache before its active usage. The prefetch mechanism is pre-programmed using the dedicated core instructions and works in the background without interfering with the core accesses.

For details on the ICache block, see Chapter 4, “L1 Caches.”

1.4.3 Data channel

The data channel consists of two separated cache systems: data cache for reads and store gather buffer (SGB) for writes. This separation allows optimized cache performance and simplified hardware memory coherency support.

The SGB is a data storage that contains 16 entries of 64 bytes each to hold the last written data. All core write accesses pass through the SGB on the way to memory. Writes that are defined as non-peripheral in the MMU are merged if possible to a single entry to save memory bandwidth. Accesses pass through the SGB in order (FIFO).

The read cache is physical with virtual allocation. The allocation is performed using a virtual address. In the case of a hit, a physical address is retrieved from the matched line and compared to the physical address from the memory management unit (MMU). If the addresses are different, the line is invalidated and a new physical line is loaded. In the case of a new line allocation, the physical address of the new allocation is checked against the existing lines in the cache. If the new physical address already resides in the cache, the matched line is invalidated in order to prevent duplication. This technique provides seamless physical cache behavior to the user.

The read data is a combination of two caches. The data may exist in both of them, but the SGB always has the more recent copy of it, and therefore the data from the SGB will be sent to the core. Each time a write from the SGB is written to memory, it is also updated in the read cache, if such a line exists. This is done to keep data coherency between the two caches.

The DCache, which operates at core speed, keeps the recently accessed data. When an addressed data (from a cacheable memory area) is found in the array, it is immediately made available to the core (DCache hit) in a read. When the required address read is not found in the array or in the SGB, a DCache miss occurs. The data is fetched to the DCache from the external (off-subsystem) memory and drives to the core.

A non-stalling non-align access support is provided by the cache as long as the access does not cross a 4 Kbyte boundary.

The data channel differentiates between cacheable and noncacheable addresses. The MMU provides cacheable/noncacheable indication for each access according to matched MATT configuration.

1.4.3.1 Read Data Cache

The DCache has the following parameters:

- A size of 32 Kbytes with 128 bytes per cache line (total of 256 lines in the cache) arranged in an 8 way set-associative structure.
- Seamless (non-stalling) unaligned access and next line hardware prefetch in a 4 Kbyte block size.

A data access that is identified as a hit in the DCache is served without freezing the core, with the exception of DCache contention (dual access from two core buses to the same module). If the requested data is not in the cache (DCache and SGB), the DCache fetches it. If the requested data belongs to a cache line that is not in the cache, the line is allocated at the expense of another line. A pseudo-LRU (PLRU) replacement algorithm selects the line to be replaced.

The hardware next line prefetch mechanism detects the presence (hit) of the next line in the cache (limited to a 4 Kbyte block). If the next line is a miss, it is prefetched to the cache depending on the MMU configuration. The hardware prefetch is a background operation that does not stall the core.

To implement hardware prefetch, each core access automatically generates a tag look up for the following cache line (if the next line is not crossing a 4 Kbyte boundary). If the next line is a miss, it is loaded to the cache, depending on the MMU setting. The hardware prefetch is performed in the background and does not stall the core.

A higher level cache coherency support is provided by the eLink protocol. Read cache support for the eLink is provided by supporting a back invalidate request from the higher level cache. Back invalidation enables the higher level cache to invalidate the stored data in the cache when the data is changed or when additional coherency is needed.

Because the read cache is not updated on an individual write (but on SGB write out), if the data is vacating the SGB and the same address (of the data) is present in the cache, the data needs to update the cache in order to keep internal data coherency. The update is performed using the physical address of the data.

A software prefetch is supported by the DCache providing a way for the programmer to preload critical data into the cache before its active usage. The prefetch mechanism is preprogrammed using dedicated core instructions and works in the background without interfering with the core accesses.

See Chapter 4, “L1 Caches,” for details on the DCache.

1.4.3.2 Store Gather Buffer (SGB)

The SGB has the following parameters:

- 16 entries of 64 bytes of data
- FIFO access ordering
- Support for two levels of priority
- Watermark mechanism ensuring quality of service

The SGB serves core write accesses. The write is stored in one entry in byte resolution (or two entries for unaligned writes). Whole entry is written out in one access. An entry can be updated (overwritten) by a new write. Whenever a write address is not matched to the stored data addresses in the SGB, a new entry needs to be allocated and the old one is thrashed if needed. As writes are usually local and may overwrite themselves the ratio of entries used compared to the number of writes is low. A memory barrier or special cache command is needed to make sure a write is written outside as the SGB doesn't empty itslef automaticaly.

The data read that is present in the SGB is sent to the core to be combined with the data read from the Dcache. The data from the SGB is the more recent data and therefore need to overwrite the data from the Dcache.

The eLink coherency protocol required a barrier command to cause SGB flush. The SGB writes out all relevant data and then sends the barrier command to the higher level destination.

The SGB acts also as write through buffer. If a write is marked by the MMU as “peripheral” the SGB does not merge the write data and sends it to the higher level destination.

See Chapter 4, “L1 Caches,” for details on the SGB.

1.4.4 Memory Management Unit

The memory management unit (MMU) performs three main functions:

- Memory hardware protection for instruction and data accesses .
- High-speed address translation from virtual to physical address to support memory relocation.
- Cache and bus controls for advanced memory management.

Memory protection increases the reliability of the system so that errant tasks cannot ruin the state of other tasks. The MMU checks each access to determine whether it matches the permissions defined for this task in the memory attributes and translation table (MATT). If it does not, the access is killed and a memory exception is generated.

The MMU performs address translation from virtual addresses (used by the software that runs on the core) to physical addresses (used by the system buses). Benefits of address translation include the following:

- Enables software to be written without consideration of the physical location of the code in memory, thereby providing a simpler software model that enhances modularity and reuse.
- Allows true dynamic code relocation without performance cost or overhead particularly in multicore devices.

The same effective addresses can be reused between tasks without a need to reconfigure MMU descriptors and flush the caches between tasks. The caches store the task ID in their line tags and thus have a unique memory image per task. The MMU translation tables include the task ID field, which provides a unique translation per task. Memory aliasing, when different virtual addresses point to the same physical address, is fully supported.

Protection and address translation are applied to memory segments defined in the MMU. A segment descriptor (SD) can set cacheable/noncacheable, prefetch policy, coherency, and more. The MMU controls up to 32 data and up to 16 program segment descriptors.

The SC3900 FVP subsystem provides two programming models for descriptor configuration:

- Aligned: descriptor memory space has a long-range variable mapping size designated in steps as a power of 2, starting from 4 Kbytes up to 4 Gbytes.
- Flexible: descriptor memory space has a middle-range variable mapping size starting from 4 Kbytes up to 16 Mbytes. The flexible model reduces aligned model constraints allowing reduction in the number of descriptors needed and memory waste.

See Chapter 3, “Memory Management Unit,” for details on the MMU.

1.4.5 L2 Cache

The L2 cache is composed of a number of independent banks that operate in parallel to process program and data accesses to external L3 cache/M3/DDR memory. Caching the accesses requested by the L1 subsystem reduces the average penalty of accessing the high latency external memories. The L2 cache also supports hardware coherency and provides an increase in snoop bandwidth.

The L2 cache consists of the following:

- Core-to-cache interface unit (CCIU), supporting four cores connecting to four L2 banks.
- Four L2 cache independent banks that operate in parallel to provide an increase in snoop and data bandwidth
- CoreNet bus interface unit

Features of the L2 cache are as follows:

- A size of 2 Mbytes (512 Kbytes per bank) with 64 bytes per cache line (total of 8192 lines in the cache) arranged in an 16 way set-associative structure.
- 64-byte line. Fetches the complete line at once. Writes back the complete line at once when a dirty line is thrashed from the cache.
- Physically addressed
- Support for stashing
- ECC support: tag, status, and data ECC (single-bit error correction, double-bit error detection)
- MESI+L hardware coherency support. Supporting for a coherency granule (CG) of 64 bytes
- Programmable partitioning control
- Selectable bank hashing
- Selectable index hashing

- Write policies:
  - — Cacheable and noncacheable on read and write (CA and NC)
  - — Cacheable write through; cacheable on read and noncacheable on write (WT).
  - — Cacheable write back; both read and write are cacheable (WB).

See Chapter 5, “L2 Cache,” for details on the L2 Cache.

1.4.6 Cache Management Engine

The cache management engine (CME) is an advanced cache maintenance instructions generator that allows the cache management to offload from the SC3900 FVP. The CME is the equivalent to a DMA in the tightly coupled memory (TCM) world. It operates in parallel to the core on its data and program buses and can be programmed using dedicated core instructions or external peripheral bus accesses. Usage of the CME can greatly improve performance and simplify cache management.

The SC3900 instruction set supports various types of cache maintenance instructions that are described in the *SC3900 FVP Core Reference Manual*. These cache instructions care for cache hierarchy performance optimization, coherency accelerator, operating system support and debug. The CME also provides the same functionality through peripheral bus configuration, allowing an external master to utilize available cache resources. The external master can also generate a data query instruction. External maintenance instructions are programmed using a set of dedicated registers.

See Chapter 6, “Cache Management,” for details on the Cache Management Engine.

1.4.7 Enhanced Programmable Interrupt Controller

The internal enhanced programmable interrupt controller (EPIC) manages internal and external interrupts. The EPIC handles up to 512 interrupts, 480 of which are external subsystem inputs. The rest of the interrupts serve internal subsystem conditions. The external interrupts can be configured as either maskable interrupts or critical interrupts (CIs). The EPIC can handle 32 levels of interrupt priorities, of which 31 levels are maskable at the core and 1 level is CI.

The EPIC decodes the index of the 480 external interrupts according to their priority, and the index is then used to look up the table of 16 interrupt dispatcher routines. Each entry holds the full virtual address of the interrupt service routine that the core jumps into. It is a user’s responsibility to configure the address for each interrupt.

See Chapter 7, “Interrupts,” for details on the EPIC.

1.4.8 Debug and Trace Unit

The main concepts of the SC3900 subsystem debug support are as follows:

- Unified support for the core and the subsystem
- Debug configuration by the external host is done via a generic slave bus; all configuration is done by accessing memory mapped registers.
- Unification of the breakpoint models between hardware and software breakpoints

- Dedicated core working mode for debug.
- Tracing is based on Nexus (IEEE ISTO 5001) standard, with an independent trace output bus.

The logic supporting debug and trace in SC3900 subsystem includes the following resources:

- Dedicated core instructions
- A debug exception
- An address detection unit, including the following:
  - Four address range detectors, which can be configured to detect up to four PC ranges or four data address ranges. Alternatively, each range detector can be configured to detect two exact PCs.
  - One exception detector, enabling the detection of service of a particular interrupt or group of interrupts
  - A task ID comparator
- A profiling unit, containing the following:
  - 6 event counters, organized in 2 triads, for profiling
  - A reloadable counter, for debug control
- An indirect event unit for advanced cross triggering, which enables implementing user-defined state machines and event sequence detectors based on debug events
- A trace unit, which supports (partial list):
  - Program flow trace
  - Watchpoint messages
  - User defined trace messages (data acquisition), based on writing to dedicated core registers
  - Snapshots of the profiling counters

See Chapter 8, “Debug and Trace Support,” for details on debug and trace support.

1.4.9 Peripheral Bus

The peripheral bus connects the subsystems and L2 cache control registers to the core and cluster slave port. The peripheral bus in the SC3900 cluster is organized into a separate grid. It allows effective intracluster peripheral accesses and full SoC access to the cluster and subsystem peripherals.

See Chapter 9, “Interfaces,” for details on the peripheral bus.

1.4.10 Timer

The timer block includes four 32-bit general-purpose counters with preloading capability. A pair of timers can be organized into one chained timer, thus providing two 64-bit timers. External synchronization allows for the easy synchronization of timers in different subsystems. Timers count clocks at the core frequency. They are intended for, but not limited to, operating system use. In addition to regular timers, there are watchdog timers for each core located on the cluster level.

See Chapter 10, “Timers,” for details on the timer block.

Chapter 2
Memory Architecture

This chapter provides an overview of the StarCore SC3900 FVP subsystem and its memory architecture. The description focuses on how software architects should view the subsystem and how its features can be used in an efficient FVP system.

2.1 Memory management

This section discusses the following:

- Section 2.1.1, “Memory protection”
- Section 2.1.2, “Address translation”
- Section 2.1.3, “Task ID and address structure”
- Section 2.1.4, “Task memory and system/shared task memory”

2.1.1 Memory protection

The SC3900 core has task-based protection that limits task activity in the memory space. Attempts to execute instructions or access registers that are not permitted result in a privilege exception.

2.1.1.1 Benefits of memory protection

The use of task protection in the subsystem increases the reliability of a multitask system because a problem with one task does not require the entire system to reboot. Therefore, the mean time between failures (MTBF) of the system is increased. Another benefit is that it is very helpful for detecting and debugging software errors.

Error exceptions are precise, so the system state is usually fully known when the exception is serviced. This is useful for debugging the error. However, in many cases the root cause of the error may be before the faulty memory access causing the exception. Therefore, the RTOS usually simply aborts an errant task.

2.1.1.2 Memory protection settings

Memory protection is implemented in the off-core MMU. The MMU contains the memory attributes and the translation table (MATT), which holds a set of program and data memory segment descriptors. Each descriptor defines the protection properties of a segment. A segment is $n \times$ 4-Kbyte memory range. In the

MATT, the segment descriptors are divided as independent segments into program segments and data segments. The protection definition scheme for program addresses is provided in this table.

Table 2-1. Program address protection parameters

Address ranges	Permissions in region
Program Segment 0	0—Not allowed 1—Allowed
...	—
Program Segment 15	0—Not allowed 1—Allowed

The protection definition scheme for data addresses is given in this table.

Table 2-2. Data address protection parameters

Address ranges	Permissions	
	Read	Write
Data Segment 0	0—Not allowed 1—Allowed	0—Not allowed 1—Allowed
...		
Data Segment 31	0—Not allowed 1—Allowed	0—Not allowed 1—Allowed

To describe the virtual and physical segments, the MMU supports the following two programming models:

- Aligned:
  - — Descriptor memory space has a long-range variable mapping size starting from 4 Kbytes up to 4 Gbytes.
  - — Size must be equal to a power of 2
  - — The base address must be aligned to the segment size.
- Flexible:
  - — Descriptor memory space has a middle-range variable mapping size starting from 4 Kbytes up to 16 Mbytes.

For more information on the segment alignment, see Section 3.2.2.3, “Virtual Segment Base and Size.”

The MATT implements a fixed priority between pairs of adjacent memory descriptor registers. This feature can be used to define segments that are not a power of 2 size. For example, a 12K segment can be implemented with a 4K segment having higher priority than an enveloping 16K segment.

The SC3900 architecture has a unified memory model. The split to different program and data permission tables occurs because memory is accessed either through the program bus as a program fetch request (read only) or through one of the data buses as a data memory access. The application developer can select from

all combinations to control the access permissions to a certain segment, much like using the Linux `chmod` command. As in Linux, not all combinations make sense. Also, the same memory addresses can be allowed in both tables if the application interleaves both program and data in the same memory area. The user should be aware of program/data cache coherency issues in such a case, as explained in Section 2.4.8, "Program and data coherency."

2.1.1.3 MATT miss exception

If the address of an access does not match any segment descriptors programmed in the MATT while protection is enabled (a condition termed MATT miss or segment miss), the MMU kills the access and generates an MMU exception. This exception is precise, meaning that on return from the exception the core re-executes the VLES that caused the memory exception. This feature allows the RTOS to update the MATT or perform other actions before returning and re-executing the access. Examples of such actions are as follows:

- Update an existing MATT descriptor to include the access that caused the exception
- Kill the task if the access is to a memory region not to be accessed
- Replace one or more MATT descriptors with others
- Swap the task with another task

The replacement task in the third bullet enables the RTOS to implement a virtual MATT consisting of more entries than are supported in the MMU hardware. The RTOS manages a virtual MATT consisting of all the descriptors needed for the task and swaps between them on a MATT miss. This procedure is in effect a software-assisted translation look-aside buffer (TLB). The software architect is not confined to the physical size of the MATT when defining the memory regions used by the software. This feature can be taken further to a full support of virtual paging by the RTOS.

2.1.1.4 Memory-related error conditions

An error condition can occur in the following cases, resulting in a precise MMU exception:

- Access to a non-permitted memory area as programmed in the MMU.
- Dual-segment descriptor hit (caused by wrong MMU programming)
- Memory attribute and translation table (MATT) miss

For the full list of errors, see Chapter 3, "Memory Management Unit."

If the privilege violation occurs in a task, the protection model of the SC3900 core ensures that the state of other tasks is not harmed. Usually, the RTOS kills the task and continues with another task. However, shared memory may be affected by the errant task (if the task has write permission to it). Thus, the system designer should consider the potential damage and the necessary action to minimize its effect; for example, prevent untrusted code from accessing a shared area using the memory protection mechanism.

One use of these MMU errors is related directly to the SC3900 architecture. The SC3900 core generates speculative accesses in various circumstances to reduce pipeline stalls. These accesses may belong to instructions which never execute. When the memory protection is disabled, such speculative accesses may have an effect on the system, and the user should be aware of them. For example, when the core executes code from a program section near the boundary of physical memory, the core may speculatively fetch

ahead from unimplemented memory. In some systems, accesses to such memory creates an imprecise bus error exception, even though the core never uses the fetched instructions. When memory protection is enabled, accesses beyond the allowed areas cannot leave the SC3900 FVP subsystem, but no memory exception is caused if these accesses are not used. Enabling protection in the MMU, the user can prevent the speculative accesses to these areas from generating unwarranted exceptions, while keeping the normal flow of the application.

2.1.1.5 Guarded memory

The well-behaved memory is a memory location for which the effect of single access is indistinguishable from the effects of multiple identical accesses. Therefore, speculative read accesses can be done to the well-behaved memory without causing undesired side effects. Memory that is not well-behaved is called destructive memory and should be protected from speculative accesses. This protection is provided by the DCache for accesses to memory areas that have the guarded attribute in their MATT entry. A destructive load must be marked with SYNC.B.DL. For details, see the *SC3900 FVP Core Reference Manual*.

2.1.2 Address translation

Address translation is a hardware mechanism that separates the addresses that the program uses to access memory (effective address) from the actual (physical) addresses. Any 32-bit effective address generated by the core is translated to a 36-bit physical address. Up to 64 Gbytes of physical address space is supported.

This figure depicts the translation operation.

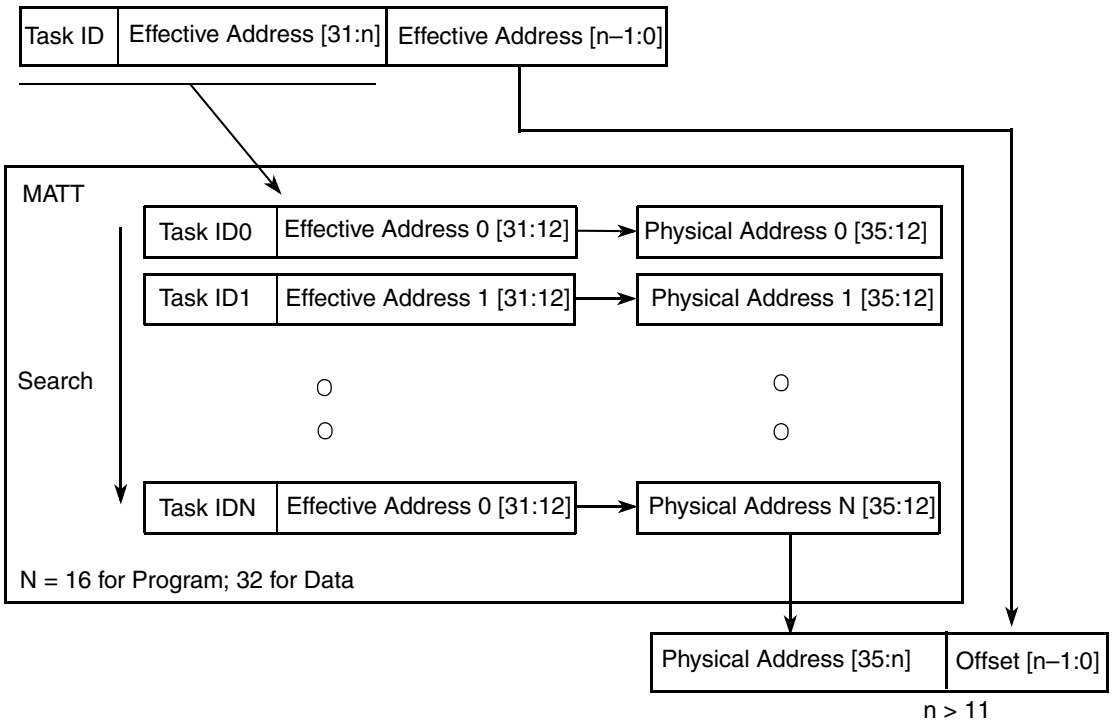

Figure 2-1. Address translation

Translation is performed only on the most significant part (MSP) of the address. The least significant part (LSP) remains unchanged. Translation is defined in the MMU per address range (memory segment). For each access, a set of comparators compares the virtual address with program or data ID and the virtual address of the enabled segments. When there is a hit, the matching translated MSP is driven in its place.

Address translation simplifies software development as follows:

- The program can be written without consideration of the physical memory partition and allocation policy of the system, thereby allowing reuse across systems and cleaner code.
- Relocatable programs can be written without affecting the way the code is written and its cycle performance.
- Use of the lower 64-Kbyte addresses can result in code that is more compactly encoded. With address translation, all tasks can reuse this virtual address space and enjoy the code size reduction.

The second point is of particular importance for FVP applications because FVP software frequently uses absolute addressing to avoid losing performance on pointer load-use interlocks. This absolute addressing creates a problem for dynamically relocatable code and in multicore devices. The problem becomes more severe when more than one instance of the same program is running. However, the problem also exists even if a single program is running in a multitask, dynamically-allocated environment in which the task cannot be sure if the memory segment allocated to it for global variables (implemented with absolute addresses) is available.

2.1.2.1 Address translation activation options

Although the MMU is designed to enable the implementation of a multitasked, protected software architecture, it also enables other types of software architectures. This approach may be advantageous in simpler systems with a software model that does not require the full capabilities of the MMU.

2.1.2.1.1 No translation

The simplest software model does not use memory translation. The 32-bits addresses used by the program extended with zeros and presents 36-bits physical address. This model is depicted in Figure 2-2. This model is only permitted for the supervisor and allows access to the lowest 4 Gbytes of 64 Gbyte physical address space. MMU protection mechanisms are not applied for such accesses.

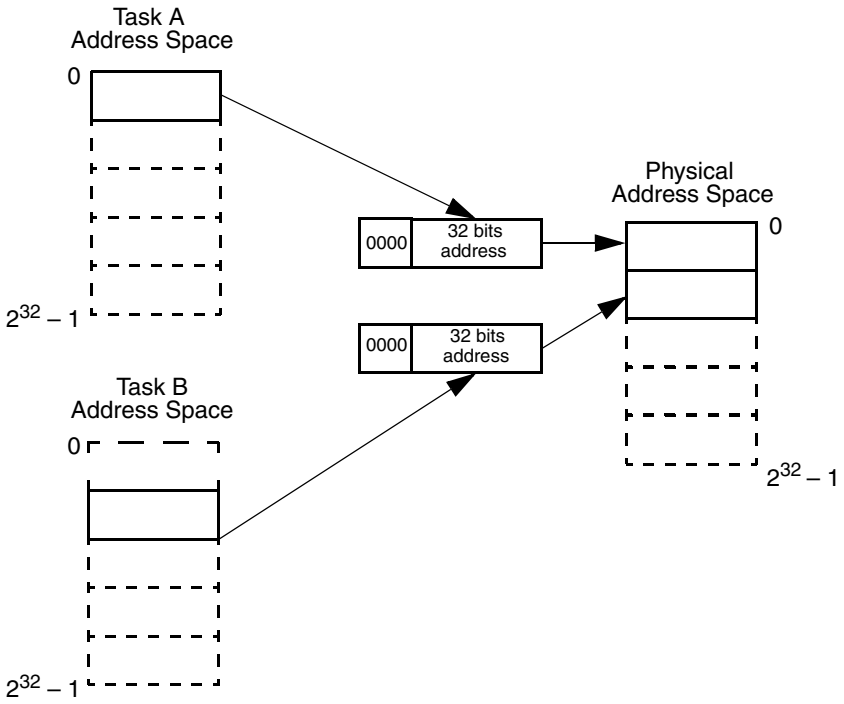

Figure 2-2. No translation, multitask memory partition example

2.1.2.1.2 Single-mapped virtual addressing

For single-mapped virtual addressing, address translation is enabled, but each address is uniquely used in the system so that different tasks and portions of the software do not use the same virtual address at the same time (refer to Figure 2-3). This software architecture assumes a global memory map that is at the disposal of the system. The memory map is divided among the tasks and other software components. The incremental advantage of this software architecture relative to the “no translation architecture” is that the software is made unaware of the system-dependent memory map. The same software architecture can be migrated with minimal changes to another system with another memory map.

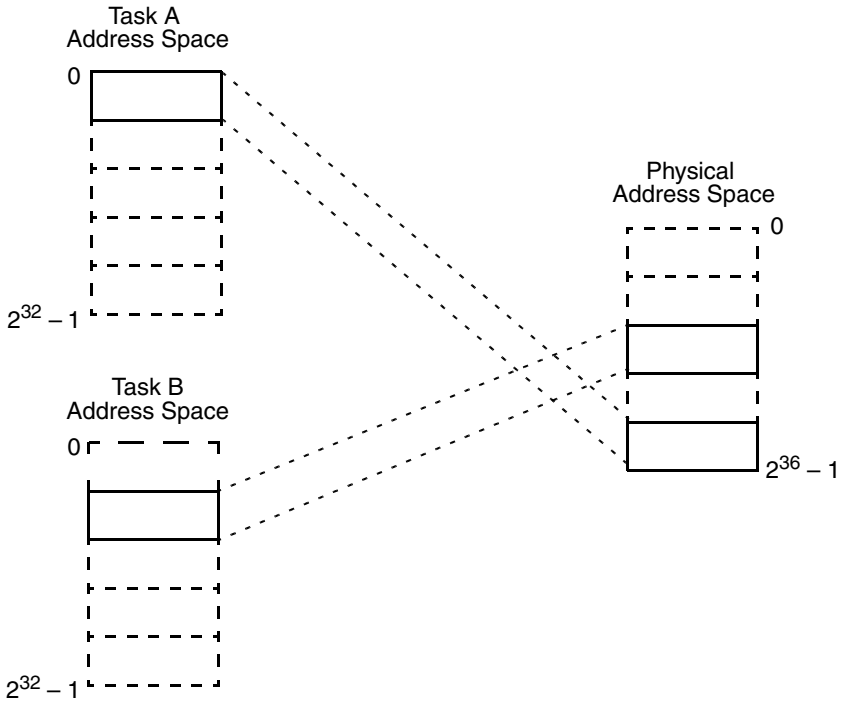

Figure 2-3. Single-mapped translation, multitask memory partition example

2.1.2.1.3 Multi-mapped virtual addressing

This software model for a system with multi-mapped virtual addressing includes:

- An RTOS and other code to manage the system.
- Software tasks that are isolated from the system and from each other. This isolation may be enforced with protection features and proper MMU programming.

The software model of the task includes a virtual memory map. The tasks can reuse the same virtual memory locations among them. Address translation maps the virtual addresses generated by different tasks to different physical memory so that the same virtual addresses are multi-mapped to different physical addresses in different tasks (see Figure 2-4).

This figure shows the multitask memory partition example of multi-mapped translation.

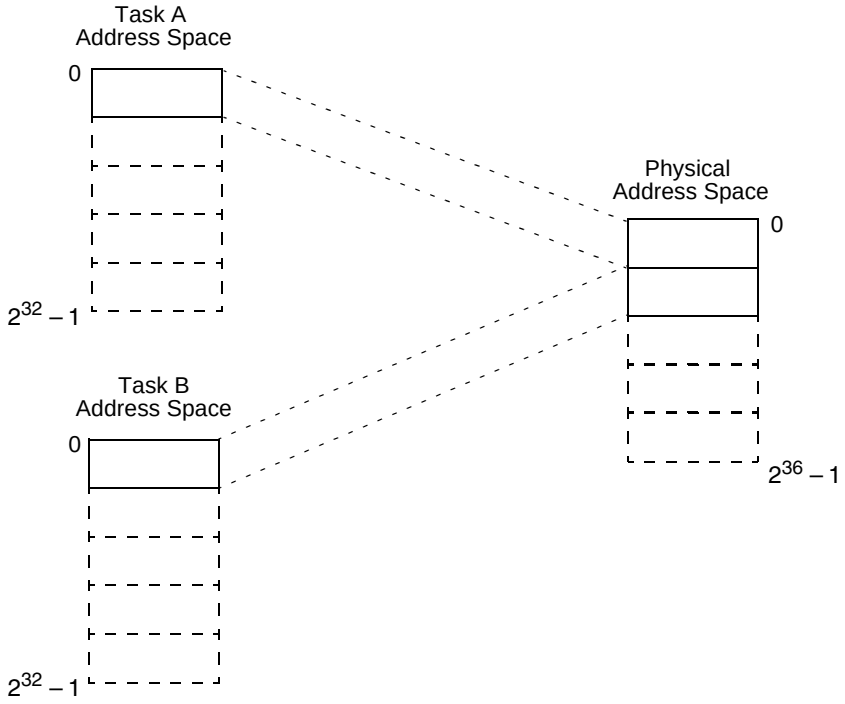

Figure 2-4. Multi-mapped translation, multitask memory partition example

2.1.3 Task ID and address structure

The SC3900 FVP has a task ID register used to identify the current task. This register includes two separate IDs for program and data, as follows:

- Program task ID (PID)
- Data task ID (DID)

The size of the PID and DID fields in the task ID registers is 8 bits each, which allows up to 255 program and data concurrent tasks. These IDs are used as address space identifiers. PID and DID are part of descriptor entry. This enables descriptors from different tasks to be enabled at the same time without risk of receiving a multi-hit error. The IDs are also partially stored as part of the cache tags. The task ID is used in the caches for performance reasons and not for logical as the physical tag match is checked as well. Tasks that share memory space should have an identical task ID that the cache stores and contrarily, tasks that do not share memory space should have different task IDs. The proposed task ID allocation is not required, but it is recommended for optimal cache operation.

An address used by the software has three representations in different components of the subsystem, as follows:

Effective address	The address represented in the program and issued on the address buses of the core.
Virtual address	The program or data virtual address, appended with the PID or DID, respectively. The virtual address is used inside the caches, in the MMU for descriptor match

calculation, and in some communication pathways between the caches and their supporting logic. In general, virtual addresses are hidden from the software and the RTOS.

Physical address — The virtual address after translation, which is the address on the subsystem external buses. The RTOS is aware of the physical addressing of tasks through the programming of the MMU translation tables.

Figure 2-5 shows the schematic address structure.

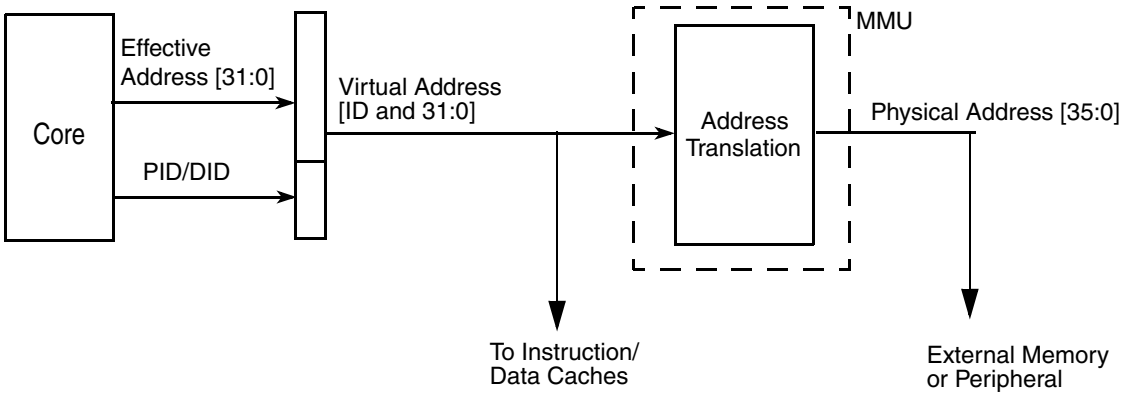

Figure 2-5. Effective, virtual, and physical addresses

Effective addresses can be reused among tasks. Extending the cache tags with a PID/DID enables this reuse without flushing the cache during each task switch. Thus, the benefits of using the caches are maintained in real-time, multitasked software environments. Each descriptor also has a TaskID field that allows overlapped (effective address) descriptors from different tasks to be enabled at the same time, removing the need of MATT reconfiguration during task switch.

The PID/DID separation is based on the nature of the common multichannel processing applications the subsystem supports. In these applications, the subsystem is the processing engine that should be ready to time-share the processing of dynamically-allocated channels (audio or data). Each channel can be in one of a set of protocols for which there is a predefined program to process it. Each channel can be instantiated by the system any number of times (up to the system/processor capacity), but each time with unique data to process. For example, the subsystem can process three channels at a time, with two in the V.92 modem application and one that uses the G.711 speech application. In this case, the processor should work with the following:

- Two program codes (one for V.92 and one for G.711), one of which is shared by two channels
- A total of three data sections that are independent of each other

Each of the three channels requires a unique DID, while only two PIDs are required. The two V.92 channels can share the same program cache entries, improving system performance. When one V.92 call replaces another, it can benefit from the instruction cache hits resulting from fetches in the preceding call.

Within the ID allocation framework, a system can select a variety of software architectures.For example, in a system that does not write to program memory and where all tasks heavily instantiate the same predefined code, it may be better to define this code under one PID.

2.1.4 Task memory and system/shared task memory

In each system, the software architect should partition the virtual memory space into three types, as follows:

- Task memory that is used by tasks as their unique memory and can be multi-mapped between tasks (that is, translated each time to different physical addresses)
- System/shared task memory (task ID equal zero) that is used by system code and should be always enabled in MMU descriptors (such as the ISRs). This memory is single-mapped (that is, cannot be reused with different physical addresses between tasks).
- System/shared task memory (task ID equal zero) that is used by system code (such as the RTOS kernel, MMU configuration code) and also memory that is shared among tasks. The caches support multi-mapped translation for such shared tasks, but with some performance degradation due to cache invalidations for overlapped addresses. The MMU does not support address overlap for system/shared tasks; therefore descriptors for overlapped system/shared memory areas may be enabled only one at a time.

Shared memory can be viewed as part of the system memory space for which the RTOS grants access permission to selected tasks. Shared memory is needed for the following reasons:

- A program is instanced more than once, and each instance is known to the RTOS as a different task. These programs share the program code and perhaps some data, such as fixed tables.
- Tasks can communicate with each other using shared memory.

Task memory versus system/shared task memory is partitioned in the MMU. For each descriptor in the MATT, the memory type is defined by the PID/DID field in the M_PSDCx/M_DSDCx registers. If this field equals zero, the respective memory segment is defined as system/shared task. If it is not zero, the segment is defined as task memory. See Chapter 3, “Memory Management Unit,” for information on MMU programming.

As for any other access, the MMU performs the checks on accesses to system/shared task memory to see if the access is permitted.

The subsystem registers and cluster register memory belongs to the system/shared task memory by definition. Other than that, the partition between task and system/shared task memory is in the hands of the software architect. This boundary is typically not changed during operation. Changing this boundary requires flushing both caches and reprogramming the MMU while running from noncacheable memory.

2.2 Cache policies

This section discusses the following:

- Section 2.2.1, “Memory and peripheral address space”
- Section 2.2.2, “L1 Cache allocation policy”

- Section 2.2.3, “L2 Cache policies”

2.2.1 Memory and peripheral address space

Address space can be separated into “memory” and “peripheral.” “Memory” address space can be cacheable or noncacheable. “Peripheral” address space is automatically noncacheable. The attributes of an access is programmable in the MMU descriptor, per address range.

- Subsystem and cluster memory-mapped control registers should be marked by the MMU and treated as “peripheral”.
- System control registers and special memory-mapped registers outside the subsystem should be marked by the RTOS as “peripheral” in the MMU.
- Memory locations to which writes must be performed deterministically before the program continues (such as destructive peripherals) should be marked by the RTOS as “peripheral” in the MMU.

Aside from these locations, the user is free to define memory ranges as cacheable or noncacheable. Because the L2 cache is inclusive, the same control is used for both L1 and L2 caches in MMU registers.

2.2.2 L1 Cache allocation policy

Writes are stored temporary in the SGB. If not peripheral, they are merged for up to 64 bytes until vacating the SGB. If a write is marked by the MMU as peripheral, the SGB does not merge it and sends it in order (among the peripheral writes) to the higher level cache. To improve merging SGB tries to keep write data inside as long as possible, writing it to the external memory only after reaching watermark, memory barrier or dedicated cache instruction.

A cacheable read is allocated in the DCache or Icache and cache triggers hardware next line prefetch (if next line is enabled in the MMU and does not cross the 4-Kbyte boundary). A noncacheable read is not allocated in the cache and can be accompanied by a guarded attribute.

2.2.3 L2 Cache policies

The L2 cache supports the following policies:

- Noncacheable on read and write (NC)
- Cacheable write back; cacheable on read and write
- Cacheable write through; cacheable on read and on write hit, and noncacheable on write miss (WT).

2.2.3.1 Cacheable/noncacheable

Noncacheable reads and writes are both processed by the cache without affecting it, i.e. without the allocation.

Cacheable reads/writes that are a hit in the L2Cache are read/written directly from the cache. Cacheable reads/writes (write back) that are a miss in the L2cache result in the following operations:

- The cache reads the cache missing line from higher level memory, without allocating the missing line in the cache
- After the data arrives, the cache allocates a line; the dirty (victim) line can be thrashed as a result from the cache.
- In the case of a read, the data from higher level memory is written into the cache. In the case of a write, the updated data is written into the cache

Write hit is not always written to the cache directly. It depends on coherency state. If the state is shared, the line must be invalidated first then a new fetch requested from CoreNet. Only after those steps are complete should the cache proceed with the write access.

2.2.3.2 Write-back policy

With the write-back policy, the accesses are update the cache if they are cache hits. If the write access is miss in the cache, its address is fetched from high-level memory to L2 cache and then write is treated like hit. Modified data is written to higher level memory on L2 cache line thrash.

2.2.3.3 Write-through policy

With the write-through policy, the accesses are written to higher-level memory and update the cache if they are cache hits. If the write access is miss in the cache, it is only written to higher level memory, no allocation is done, and the cache is unaffected.

2.3 Cache performance mechanisms

Both L1 and L2 caches provide different performance enhancing mechanisms that can be used for application specific optimizations.

2.3.1 L1 cache performance mechanisms

The L1 cache can perform a software or hardware prefetch. These are discussed in the following subsections.

2.3.1.1 L1 software prefetch

Two dimensional prefetch is a useful tool for bringing complex data structures into the cache. Cache software prefetch is a mechanism designed to fill the cache with upper level memory data before it is needed to increase the cache hit rate. This mechanism enables the prefetch of a specific address space as programmed in the cache maintenance instruction. For a description of how the cache maintenance instructions use the cache management engine (CME), see Chapter 6, “Cache Management.”

Prefetch can be instruction or data and to the L1 or to L2 cache (L1 prefetch updates L2 as well because of cache inclusiveness). Instruction and data L2 cache prefetch differ even when the L2 cache is shared: it gets different MMU translation, different attributes, and different coherency treatment.

2.3.1.2 L1 hardware prefetch

L1 cache allocates a line for each read miss and fills it to the end. Additionally, L1 cache implements hardware prefetch by automatically generating a tag look up for the following cache line (if next line is not crossing 4 Kbyte boundary). If next line is a miss it is loaded to the cache (depends on the MMU setting). The hardware prefetch is performed in the background and does not stall the core.

Thus each core read access can be configured to cause automatic (hardware) prefetch of up to two lines (256 bytes). Prefetch accesses are prioritized with core misses causing minimal interference.

2.3.2 L2 cache performance mechanisms

This section discusses the L2 cache performance mechanisms.

2.3.2.1 L2 partitioning

The L2 cache supports a flexible way partition/allocation control scheme. Each transaction that misses in the cache looks in a table to determine whether or not to allocate and which ways are available for allocation. Table entries are matched by comparing partition IDs that are composed from the static core ID and 2 bits configurable in MMU per descriptor. Allocation is then controlled based on instruction/data allocation control registers as well as the L2 WAY partitioning register. For a detailed description, see Chapter 5, “L2 Cache.”

2.3.2.2 L2 touch support

The L2 cache supports prefetching a line prior to its usage. The pre-fetch (touch) can be also locked in the L2 cache prior to its usage. The locked line can be released by the Lock Clear command from the core (via eLink). For a description of how cache maintenance instructions use the cache management engine (CME), see Chapter 6, “Cache Management.”

2.3.2.3 L2 stashing

The L2 cache supports stashing that is a snoop of address range on the CoreNet fabric. When the desired access is detected, it is loaded into the cache.

Stash commands can be followed by lock (per CG) to lock data that is brought by stash until the data is required by a miss from the L1 cache. The UNLOCK command from the CoreNet unlocks the cache block holding the referenced CG.

2.3.2.4 L2 locking

The L2 cache can be locked on a per line base (per 64 byte). All CME prefetch maintenance instructions have the following variant with L2 lock: prefetch is locked and will not be thrashed until unlock. This feature keeps frequently used data in the L2 cache so that all of its thrashing is not done by big and rarely used data. Because lock is actually a destructive instruction that can reduce the active L2 cache size if done improperly, it is limited to block prefetch maintenance instructions as they cannot be speculative. For a detailed description, see Chapter 5, “L2 Cache.”

2.4 Cache and data coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. This allows easy usage of shared memory/resources in the multicore system.

The SC3900 FVP subsystem implements memory coherency in the following ways:

- Hardware (HW) coherency—the SC3900 Cluster supports a coherency protocol (MESI + L) that allows the system to be coherent with minimal requirements from software.
- Software (SW) coherency—the SC3900 FVP subsystem supports cache coherency instructions that allow the software to maintain coherency by itself.

The discussion of coherency in this section applies to the L2 cache and the L1 data cache only. The instruction cache coherency is software based.

2.4.1 Hardware coherency

The SC3900 FVP subsystem maintains hardware coherency for the addresses where the MMU provides the memory coherency attribute. The system maintains coherency on the basis of coherency granules (64 bytes), which are an aligned block of memory. The coherency model is based on the per granule state information maintained by system caches and its prescribed transitions. To maintain coherency, the cache hierarchies must obey the coherency protocols. The coherency domain does not include unfinished write accesses, so users need to execute the memory barrier command to it at the end of the write out process. Generally coherency implementation does not replace the need to synchronize processes to make sure they access memory in appropriate order.

2.4.1.1 MESI+L protocol

The MESI+L protocol is a widely used cache coherency and memory coherence protocol. The MESI+L protocol requires that each cache line is marked with one of the following five states:

Modified The cache line is present only in the current cache and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the exclusive state.

Exclusive The cache line is present only in the current cache, but is clean (not modified); it matches main memory. It may be changed to the shared state at any time in response to a read request. Alternatively, it may be changed to the modified state when writing to it.

Shared The cache line may be stored in other caches of the SOC and is clean; it matches the main memory. The line may be discarded (changed to the invalid state) at any time.

Invalid The cache line is invalid.

Last This indicates the last shared state.

To allow reloading of the L2 cache and perform non-coherent read transactions for L1 instruction cache misses, the L2 cache supports a new coherency state (in addition to the MESI+L). This new state is called non-coherent (or “N,” for short).

2.4.1.2 Coherency and caches

The SC3900 FVP subsystem implements two different coherency protocols, as follows:

- L2 cache fully supports the MESI+L protocol toward output of the SC3900 FVP subsystem and an internal ELink protocol between L2 cache and L1 caches of the subsystem.
- L1 cache supports eLink protocol toward L2 cache. This permits implementing hardware coherency while optimizing bus bandwidth.

The MESI+L protocol is explained in section Section 2.4.1.1, “MESI+L protocol.” The L2 cache implements MESI+L with the CG size of 64 bytes. The ELink protocol between L1 and L2 is explained in the following paragraph.

The L2 cache is inclusive; all addresses that exist in the L1 cache also exist in the L2 cache. For each line allocated in the L2 cache, the L2 cache maintains the status of this address in the L1 caches of the cluster. This maintains coherency inside the cluster without implementing the complex MESI+L protocol in L1 caches. Coherency is maintained in the following way:

- Read access from the core causes a read request to the L2 cache. The L2 cache provides the data and updates the status bits of the line location in L1 caches.
- Write access from the core becomes visible to the other masters in the system and participates in the system coherency when it appears on the ELink. Therefore, cases when it is required to verify the access arrival to L2 coherency domain, the software has to use ordering instructions (barriers).
- L2 cache sends back-invalidate request to one or more L1 caches as a result of a change in the CG coherency status. This change may happen due to the internal cluster access by one of the cores or due to an external to cluster access. The back-invalidate request causes the L1 cache to invalidate the line.

2.4.2 Accelerating software coherency

The hardware coherency mechanism provides great performance across a range of applications, but it can be turned off—for example when porting legacy code. Users can use a special cache instruction to maintain coherency itself. For example, a coherency accelerator may be needed to maintain coherency for data that the MMU has not marked with the memory coherency attribute. It can also be used for hardware coherent areas as a method to push out the data, if needed. This instruction is efficient for keeping coherency for small amounts of data (for example, a few hundred Kbytes).

CME cache coherency accelerator instructions are a mechanism designed to support coherency accelerator algorithms, so they should be mainly used for memory areas with disabled hardware coherency, for coherency acceleration, or for debug. This mechanism enables invalidation and/or synchronization of a specific address space as defined in the maintenance instruction. Synchronization is needed if a data which is modified in the cache array is going to be read from the system level memory by an external module. Invalidation is useful if a data or instruction which is in the cache array will not be used and required if the data in the system level memory is more recently updated than the one in the cache array. Flush is a

combination of synchronization and invalidation, and it is needed if data that is modified in the cache array is going to be read from the system level memory and will not be used by the core.

There is no need to use the sweep operation on the cache as in SC3850 subsystem because the SC3900 data cache always works in write through mode and the flush command effectively clears the L1 cache.

2.4.3 Memory access synchronization

The SC3900 subsystem supports out of order memory accesses; therefore, the barriers are used to maintain ordering.

In a hardware coherent system like the MSC9164 (fully coherent from cluster level and up), there are several use cases that can manage with a lightweight memory barrier, which relies on the hardware coherency system. Lightweight barriers allow ordering to be achieved with minimal performance impact. In other use cases, heavyweight barrier are required to ensure ordering of accesses and events in the system.

The SC3900 subsystem supports the types of memory barriers shown in this table.

Table 2-3. Memory barriers supported by the SC3900 subsystem

Memory barrier	Description
DBAR.IBSS.L1	Behaves like a store command. This memory barrier is a lightweight L1 internal barrier. The DBAR.IBSS.L1 is issued between two store access. The barrier ensures that the two store access will not be packed into one entry in the SGB and it also drains all the prior entries in the SGB.
DBAR.IBSS.L12	A lightweight barrier for a cacheable store followed by a store and behaves like a store command. The DBAR.IBSS.L12 ensures that the first store is updated in the coherency domain before the second store is updated in the coherency domain.
DBAR.IBLL	A lightweight barrier used for cacheable load followed by a cacheable load and behaves like a load command. The DBAR.IBLL ensures that the first load is updated in the coherency domain before the second load is sent out.
DBAR.IBSL	A lightweight barrier used for cacheable store followed by a load and behaves like a load command.
DBAR.SCFG	is used after the stores to configuration registers. The DBAR.SCFG holds the core until the store to the peripheral bus (memory mapped register) is preformed. The DBAR.SCFG ensures that any access that flow it is preformed only after the store configuration is completed.
DBAR.L1SYNC	behave like a load command. This barrier is used to bring the DCache and peripheral bus into Idle state. When the DCache detects the DBAR.L1SYNC, it immediately holds the core. The hold is negated only after the DCache is idle and the all peripheral accesses are closed. DBAR.L1SYNC is a stronger version of DBAR.SCFG and should be used after MMU programing.
DBAR.EIEIO	A heavyweight barrier with no respond and it is used for a noncacheable store followed by a noncacheable load/store. The DBAR.EIEIO instruction provides an ordering function that ensures that all load and store instructions initiated prior to the DBAR.EIEIO instruction complete in main memory before any loads or stores subsequent to the DBAR.EIEIO instruction access memory.
DBAR.HWSYNC	A heavyweight barrier with respond is used for all other access combination.This is the only barrier that completes only after all synchronization operations have completed. The HWSYNC ensures that the first load/store reaches its final destination before serving the following load/store. HWSYNC affects the access flow in the same way as EIEIO does. In addition, it waits for a barrier done indication. The core is released and the following load/store can progress only after the L1 receives the barrier done indication.

Table 2-3. Memory barriers supported by the SC3900 subsystem (continued)

Memory barrier	Description
DQSYNC	A global invalidation of the DCache.
PBAR.L1SYNC, PBAR.HWSYNC, and PQSYNC	These are symmetric to the DBAR and are preformed on the program channel using the Cache Management Engine's query channel. See Chapter 6, "Cache Management."

2.4.4 Register access synchronization

There are various configuration registers in the cluster and SoC levels that need to be written before use. For cluster-level configuration registers, it is enough to put DBAR.SCFG after the write to ensure that the write is executed. For SoC-level registers, DBAR.HWSYNC should be executed after the write and then a read of that same configuration register. The configuration read will push the configuration write all the way to the destination.

2.4.5 Decorated instructions

Decorated load and store instructions provide load and store operations to memory addresses (integrated device-specific) that have additional semantics available other than the customary load (read) and store (write). A decoration is additional semantic information to be applied to the decorated storage operation. The device performs device-specific semantics using the decoration, address, size, and store data (for decorated store instructions) and returns load data (for decorated load instructions).

Decorated storage also provides a notify instruction. A notify is a type of access that is neither load or store because it does not provide a data value (store) and does not receive a data value (load). A notify sends the decoration with the address to the device associated with the memory address. The device uses the decoration to determine the semantic operation.

For more information on decoration, please see the SoC reference manual.

2.4.6 Semaphore support

The SC3900 core uses the following special instructions for semaphore support:

- Read-and-reserve instruction
- Write conditional instruction

The read-and-reserve instruction reads from an address and causes the system to mark this address as reserved. The reservation is done in 64 bytes of resolution even if the read-and-reserve instruction size is smaller. The conditional write instruction succeeds only if the reservation was not reset when the write access arrives. If the write is unsuccessful, the data is not updated in the memory and the fail status is indicated in an internal core status bit.

The specified storage location must be in memory that has the memory coherence attribute if the location may be modified by other processors or mechanisms. Semaphores must be defined in a non-cacheable memory segment.

2.4.7 Messaging support

Messaging instructions provide facilities for processors within a coherence domain to send doorbell like messages to other devices in the coherence domain. The facility provides a mechanism for sending interrupts to other processors that are not dependent on the interrupt controller and allow message filtering by the processors that receive the message.

Messaging initiated by processors to processors is topology-independent.

2.4.7.1 Sending and receiving messages

Processors initiate a message by executing the **msgsnd** instruction and specifying a message type and message payload in a general purpose register. Sending a message causes the message to be sent to all the processors including the sending processor, in the coherence domain in a reliable manner. The message is broadcast on the interconnect mechanism that connects all devices in the coherence domain and has a unique transaction type that cannot be generated using any other processor operations. The uniqueness of the transaction type insures that processors can only generate a message transaction using the defined instructions that send such messages.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a processor can send.

When a device or processor receives a message, the processor examines the message type and payload to determine whether the device or processor should accept the message. This is called message *filtering*. If after examining the payload the device or processor, decides that the message should be processed (i.e. has met all the appropriate criteria specified in the message type and payload), the device or processor *accepts* the message and processes it accordingly.

The SC3900 cluster filter and accept messages are defined in Table 2-4. Processors ignore (that is, filter and do not accept) any messages with other message types.

Table 2-4. Processor message types

Message type	Message type name	Message type description
0	DBELL	Doorbell. A processor doorbell interrupt is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message.
1	DBELL_CRIT	Doorbell Critical. A processor doorbell critical interrupt is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message.
2	G_DBELL	Guest Doorbell. A guest processor doorbell interrupt is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message.

Table 2-4. Processor message types (continued)

Message type	Message type name	Message type description
3	G_DBELL_CRIT	Guest Doorbell Critical. A guest processor doorbell critical interrupt is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message.
4	G_DBELL_MC	Guest Doorbell Machine Check. A guest processor doorbell machine check interrupt is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message.

Messages of the same type are not cumulative. That is, if a message of type *n* is accepted, and the interrupt is pending, further messages of type *n* accepted by the processor are ignored until the associated pending condition is cleared by taking the interrupt. In general, software is required to use memory to perform higher level messaging, using **msgsnd** to notify other processors that higher level messages are waiting for the accepting processor to process. The timing relationship between when a message is sent and when it is received is not defined.

2.4.8 Program and data coherency

The SC3900 architecture has a unified memory map, so an address can be either a program or a data location. Most FVP applications use this feature to provide software flexibility, which enables memory usage between tasks and applications to be changed. However, a program rarely uses this feature to actually modify itself. That is, the program uses data writes to change the contents of the code to be used as program instructions by that same program.

The hardware does not provide any means to ensure coherency between the instruction and data caches. The program and data fetch paths are completely diverged. If data is written to an address that is already fetched in the ICache, it is not reflected by the hardware. In such a case, the software should flush the relevant area in the ICache so that it can be reloaded with the modified program. For completeness, the software should execute the Flush BTB and perform a change-of-flow instruction to flush the core internal fetch pipe and reload new instruction fetch sets from the change-of-flow destination.

Chapter 3
Memory Management Unit

The memory management unit (MMU) controls memory accesses and translates virtual addresses to physical addresses. It performs threemain functions:

- Supplies hardware data and program access protection.
- Implements a high-speed address translation mechanism that translates from virtual to physical addresses to support memory relocation.
- Provides cache and bus controls for advanced memory management.

A high-speed address translation mechanism enables memory relocation and checks access permissions for core instructions and data buses. It also controls hardware task protection and provides cache and bus controls for advanced memory management. The MMU enables the system designer to integrate system resources and define a clean software model. For example, programming protected regions, address translation regions, cacheable regions, and so on can be combined. In addition, cache usage can be optimized, based on the specific attributes controlled by the MMU programming.

For memory protection, an RTOS supports protection, thereby protecting the operating system, task code, and data from being affected by an errant task.

For address translation, there is a software model in which the code is written in virtual addresses that are translated to physical addresses before they are issued to memory. Any address in 32-bits virtual address space could be translated to any address in 36-bits physical address space. The only limitation is that for every physical address there should be only one data MMU descriptor that points to it. As shown in Figure 3-1, the MMU provides the virtual memory software model.

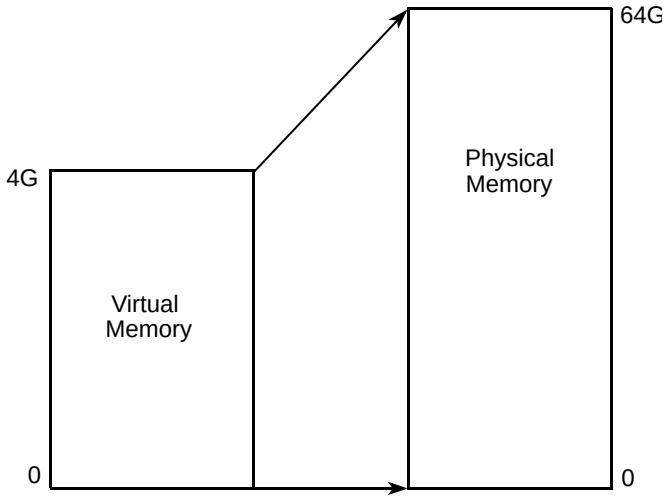

Figure 3-1. Virtual and physical address ranges

The core generates effective addresses that, together with the task ID, represent the virtual address used by the caches. The MMU translates between virtual and physical addresses during each core access (for details refer to Section 3.2.2, “Translation table and memory descriptors”).

The MMU provides control attributes for each core access per memory segment, such as prefetch enable, write-policy, cacheability, and so on. It handles all memory protection. If an attempted memory access is not permitted, the MMU kills it by sending abort signals to the relevant SC3900 FVP subsystem components and an MMU exception indication back to the core. The MMU also stores the address and attributes of an access that is not permitted (for details, refer to Section 3.2.4, “Protection Unit”). The MMU includes subsystem registers among its control and status registers. The MMU registers are memory mapped on the peripheral bus and can be programmed by the subsystem’s core or from any other external master.

3.1 MMU features

The MMU has the following functions and features:

- Memory attributes and translation table (MATT) composed of 32 data segment descriptors and 16 program segment descriptors.
- Each segment descriptor defines a memory region and its attributes, protection, and address translation.
- Address translation for each program/data memory region:
  - 32-bit effective address extended with PID/DID translated to 36-bit physical address.
- Two programming models:
  - Aligned: descriptor memory space has a long-range variable mapping size designated in steps as a power of 2, starting from 4 Kbytes up to 4 Gbytes. The base address must be aligned to the segment size.
  - Flexible: descriptor memory space has a middle-range variable mapping size starting from 4 Kbytes up to 16 Mbytes. The alignment restriction is much more flexible in this model.
- The memory region dedicated attributes support the following:
  - Cacheable access
  - Hardware prefetch policy
  - Write policy for data memory
  - Data memory/peripheral area
  - Memory coherency
  - Data guarded memory
  - Stack memory
  - L2 partitioning ID
- Hardware data and program access protection defined for each data/program memory region. The MMU generates kill signals for the core program and data buses for errant accesses. The MMU provides memory region support, as follows:
  - Program memory region: provides read allowed/not allowed
  - Data memory region: provides read/write allowed/not allowed

- Priority mechanism between descriptors, allowing overlapping of memory regions.
- Supports four exception service routines for precise MMU errors (two for program and two for data)
- Control and status registers include a subsystem ID register and doorbell-level register.
- Supports edge interrupt (doorbell edge) generation for fast intercore communication. Doorbell edge can be initiated by the core (slave write) and by the CME.
- Handles access error detection with support for misaligned semaphore access, misaligned peripheral access, peripheral accesses with unsupported sizes.
- Capture of error status bits enables a fast error diagnostic.
- Precise interrupts, allowing handling MMU memory attributes and translation table (MATT) miss, supporting a virtually paged operating system
- Error detection code (ECC) scheme
- Enable/disable ECC exception mechanism
- Query mechanism for program and data
- Address protection and translation for CME accesses

3.2 MMU functional description

The following sections describe the MMU functionality:

- Section 3.2.1, “Internal architecture”
- Section 3.2.2, “Translation table and memory descriptors”
- Section 3.2.3, “Attributes logic”
- Section 3.2.4, “Protection Unit”
- Section 3.2.5, “Task ID support”
- Section 3.2.6, “MMU query mechanism”
- Section 3.2.7, “Multiple bank support”

3.2.1 Internal architecture

Figure 3-2 shows a functional block diagram of the MMU, which is used for illustration only and does not represent the final implementation structure of the MMU. In this figure, XA/B and RP are the core address buses.

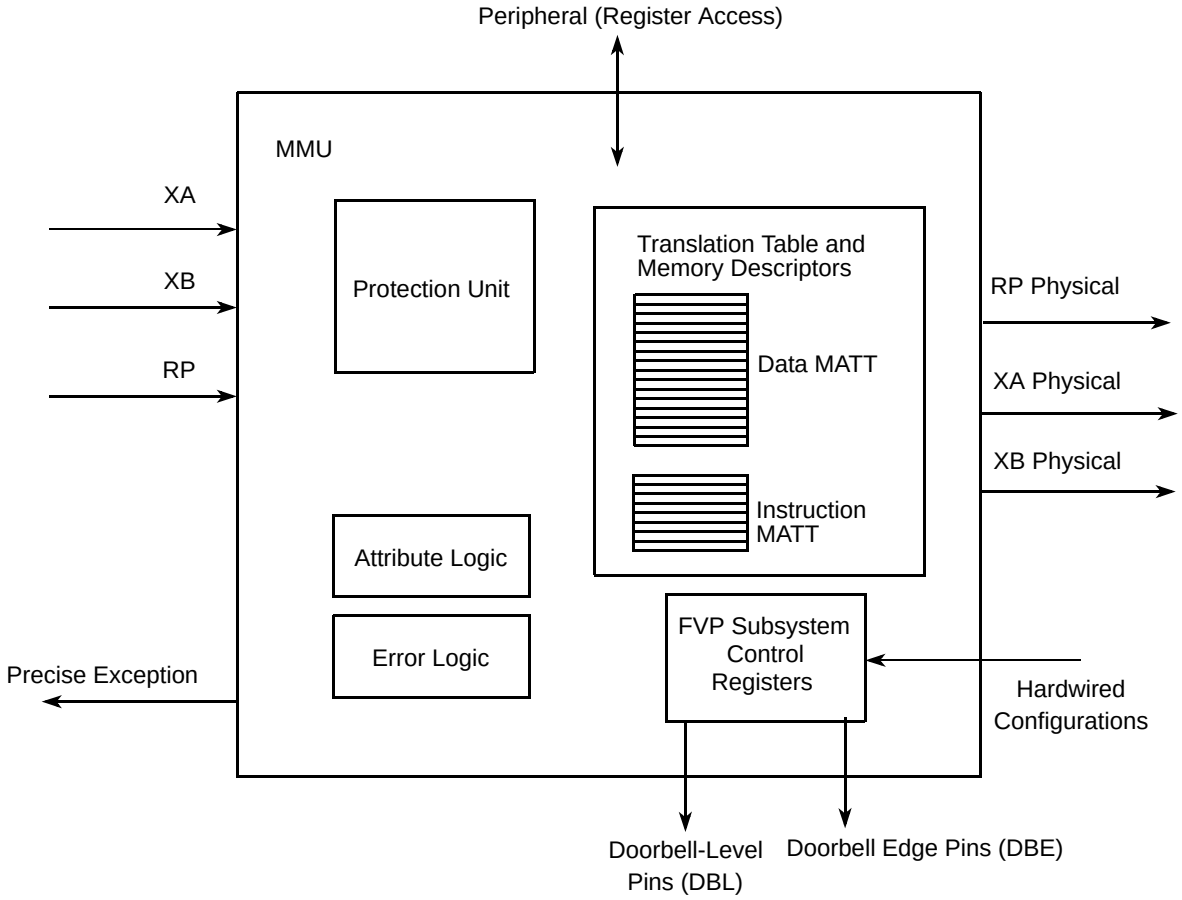

Figure 3-2. MMU functional block diagram

3.2.2 Translation table and memory descriptors

The MMU data memory attributes and translation table (MATT) and program MATT are depicted in this figure.

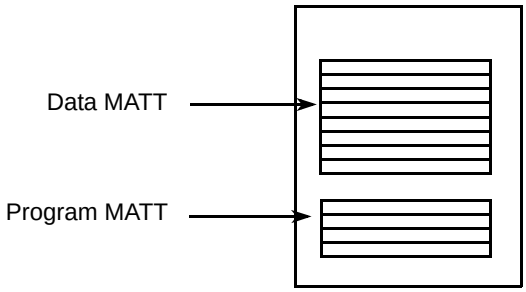

Figure 3-3. MATT diagram

The MATT enables different memory regions to be defined. The MMU supports two main memory regions:

- Task memory
- System/shared task memory

These two memory types are mutually exclusive with no overlapping memory between them. This separation also applies at the effective address level in the core (bits 31:0 of the virtual address). The application has to allocate different effective addresses to memory used by tasks, and to memory used for system and OS code. When an access arrives from the internal core buses, the MMU searches for the descriptor related to the access. When a hit on one of the memory descriptors occurs, the MMU generates the related physical address and all related attributes for the access.

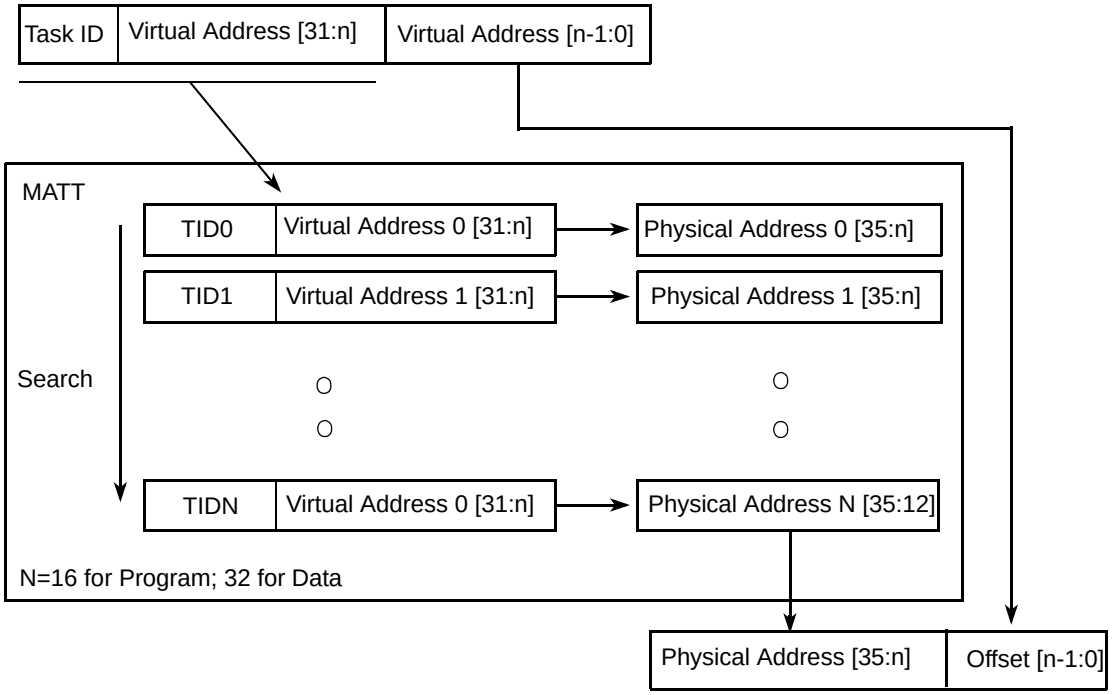

Figure 3-4. Generation of 36-bits physical address

In normal operation, every memory region is mapped by only one of the enabled segment descriptors (unless using the priority mechanism, for more details refer to Section 3.2.2.4, “Descriptors priority mechanism”). When there is a miss on the MATT while the protection unit is enabled (see Section 3.2.4, “Protection Unit”) or a double-hit on the MATT, a memory exception indication is sent back to the core, and the access is killed.

For every physical address there should be only one data MMU descriptor that points to it. The program MMU has no such limitations. Tasks can access the same physical address using a shared address space (for more details refer to Section 3.2.5.1, “System/shared task accesses). Example of the data address mapping is shown in Figure 3-6.

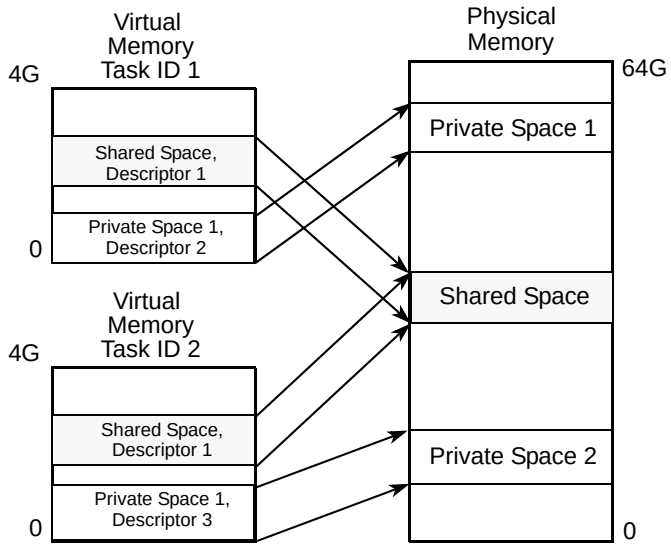

Figure 3-5. Example of data address map

When a miss on the MATT occurs, all attributes of the MMU are set to defaults. When a double hit on the MATT occurs (in non-coupled descriptors), the results of all MMU attributes are unknown.

On a MATT miss while the protection unit is enabled, recovery by software is possible due to the precise interrupt support (see Section 3.2.4.2.1, “Precise exceptions”). It requires programming one of the MATT descriptors with the memory attributes of the killed access and then re-executing the killed access.

3.2.2.1 Data MATT

The Data MATT enables users to define up to 32 different data memory regions, with each region having different memory characteristics and access rights. The Data MATT consists of 32 segment descriptors (SD), as shown in Figure 3-6.

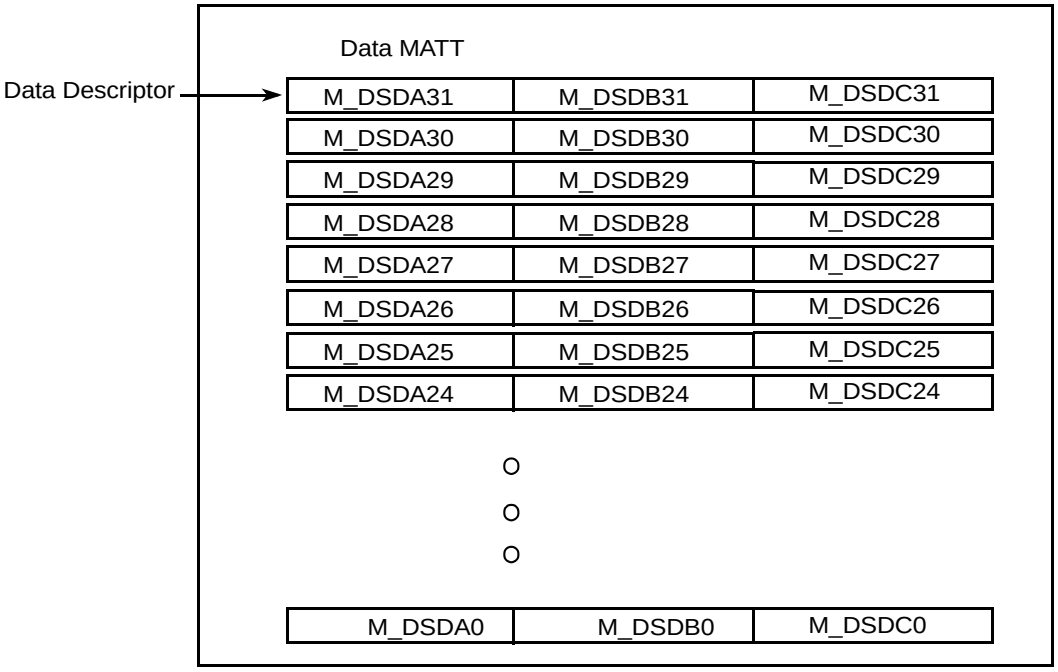

Figure 3-6. Data MATT diagram

Each data descriptor has the following content:

- Segment virtual base address
- Segment Size
- Physical base address
- Task ID
- permissions
- Cacheable
- Write policy
- Guarded
- Memory coherency
- Memory/peripheral access
- Prefetch policy
- Stack/normal
- L2 PID
- Programming model
- Valid

This content is distributed to three 32-bits registers: M_DSDA, M_DSDB, M_DSDC.

During data access on the core buses, the Data MATT provides information regarding the protection attributes of the access. If the access is permitted, the MMU provides the related bus and cache attributes. If the access is not permitted, the MMU generates an MMU exception and aborts the access.

The Data MATT has the following characteristics:

- It consists of 32 data memory descriptors (M_DSDA, M_DSDB and M_DSDC).
- Each descriptor describes a region in memory.
- A region is defined by a base address and size.
- It has the following two programming models:
  - Aligned: size can range from 4 Kbytes up to 4 Gbytes in powers of 2. The base address must be aligned to the segment size.
  - Flexible: size can range from 4 Kbytes up to 16 Mbytes. The alignment restriction is much more flexible in this model.
- Handles descriptors priority mechanism

3.2.2.2 Program MATT

The program MATT enables users to define up to 16 different data memory regions, with each region having different memory characteristics and access rights. The Program MATT consists of 16 segment descriptors (SD), as shown in Figure 3-7.

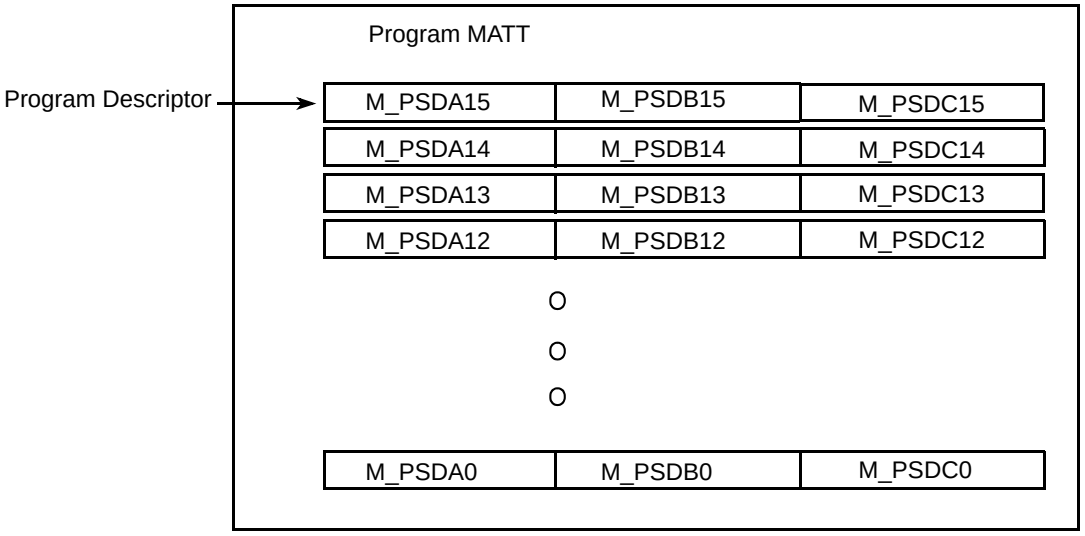

Figure 3-7. Program MATT diagram

Each program descriptor has the following content:

- Segment virtual base address
- segment size
- Physical base address
- Task ID
- permissions

- Cacheable
- Memory coherency
- Prefetch policy
- Programming model
- L2PID
- Valid

This content is distributed to three 32-bits registers: M_PSDA, M_PSDB, M_PSDC.

During an instruction fetch access through the internal program bus, the program MATT provides information regarding the protection for the access. If the access is permitted, the MMU provides the related bus and cache attributes. If the access is not permitted, the MMU generates an MMU exception and aborts the access.

The program MATT has the following characteristics:

- It consists of 16 program memory descriptors (M_PSDA, M_PSDB, and M_PSDC).
- Each descriptor describes a region in memory.
- The region is defined by a base address and size.
- It has the following two programming models:
  - Aligned: size can range from 4 Kbytes up to 4 Gbytes in powers of 2. The base address must be aligned to the segment size.
  - Flexible: size can range from 4 Kbytes up to 16 Mbytes. The alignment restriction is much more flexible in this model.
- It handles the descriptors priority mechanism.

3.2.2.3 Virtual segment base and size

The MMU supports two programming models to describe the virtual and physical segments:

- Aligned Segment Model
- Flexible Segment Model

Each segment descriptor supports both models. The M_PSDA_<i>.PSPM bit (for program) and the M_DSDA_<i>.DSPM bit (for data) defines the programming model per descriptor, for all three related registers (M_PSDA_<i>, M_PSDB_<i>, M_PSDC_<i>—for program, and M_DSDA_<i>, M_DSDB_<i>, M_DSDC_<i>—for data).

3.2.2.3.1 Aligned segment programming model

The aligned segment programming model supports segment sizes of 4 Kbytes up to 4 GBytes and is restricted by the base address (virtual and physical), which must be aligned to the segment size. This mode is kept for very large memory areas (>16 Mbytes). More flexibility is given by the Section 3.2.2.3.2, “Flexible segment programming model.”

Base address aligned to segment size:

The virtual segment descriptor register contains the definition of the base address and size of the related segment. The segment base address should always be aligned to the segment size. The minimum segment size is 4 Kbytes, while the maximum segment size is 4 Gbytes. The size is defined as $2^n \times 4$ Kbytes (n = 0..20). Table 3-1 summarizes the different programming options for the segment descriptor register base and size field. The base address and size column, represents the 20 MSBs of the base address required for the segment. The 12 LSBs of the base address are insignificant (as a consequence of the minimum segment size).

The ‘x’ values in Table 3-1 represent the MSBs of the base address, the LSBs of the base address are zero due to the alignment constraint. The 1 and 0 values represent the bits that must be programmed, based on the size of the segment.

Table 3-1. Segment descriptor base and size

No.	Size	Base address and size [31:11]
1	4 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_1
2	8 Kbytes	xxxx_xxxx_xxxx_xxxx_xxx1_0
3	16 Kbytes	xxxx_xxxx_xxxx_xxxx_xx10_0
4	32 Kbytes	xxxx_xxxx_xxxx_xxxx_x100_0
5	64 Kbytes	xxxx_xxxx_xxxx_xxxx_1000_0
6	128 Kbytes	xxxx_xxxx_xxxx_xxx1_0000_0
7	256 Kbytes	xxxx_xxxx_xxxx_xx10_0000_0
8	512 Kbytes	xxxx_xxxx_xxxx_x100_0000_0
9	1 Mbyte	xxxx_xxxx_xxxx_1000_0000_0
10	2 Mbytes	xxxx_xxxx_xxx1_0000_0000_0
11	4 Mbytes	xxxx_xxxx_xx10_0000_0000_0
12	8 Mbytes	xxxx_xxxx_x100_0000_0000_0
13	16 Mbytes	xxxx_xxxx_1000_0000_0000_0
14	32 Mbytes	xxxx_xxx1_0000_0000_0000_0
15	64 Mbytes	xxxx_xx10_0000_0000_0000_0
16	128 Mbytes	xxxx_x100_0000_0000_0000_0
17	256 Mbytes	xxxx_1000_0000_0000_0000_0
18	512 Mbytes	xxx1_0000_0000_0000_0000_0
19	1 Gbyte	xx10_0000_0000_0000_0000_0
20	2 Gbytes	x100_0000_0000_0000_0000_0
21	4 Gbytes	1000_0000_0000_0000_0000_0 0000_0000_0000_0000_0000_0

For example, a segment with a base address of 1 Mbyte (0x0010_0000) and a size equal to 256 Kbytes supports the basic requirement that the base is aligned to the size. The steps required to define this segment are as follows:

1. According to the size (256 Kbytes), choose row number 7 in Table 3-1. The 7 lowest bits as shown in Table 3-1 are 100_0000 (non-x bits in the table)
2. Write the base address in a 32-bit representation. The 1-Mbyte base is written as: 0000_0000_0001_0000_0000_0000_0000_0000.
3. Determine the base address bits for the segment descriptor. The upper bits are determined according to the remaining bits ([31:18], marked by x in the table) of the base address, which in this case are 0000_0000_0001_00.

This definition process results in base binary form, 0000_0000_0001_0010_0000_0.

Physical segment base address:

The physical segment descriptor register contains the definition of the physical base address of the related virtual segment. The physical address base should always be aligned to the segment size. Table 3-2 summarizes the different programming options for the physical segment descriptor register base address, based on the size of the segment size defined in the related virtual segment. The physical base address column represents the 24 MSBs of the physical base address required for the segment. The ‘x’ values in Table 3-2 represent the programmable base address, and the 0 values represent the bits that must be programmed.

Table 3-2. Physical descriptor base address

No.	Size	Physical base address [35:12]
1	4 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_xxxx
2	8 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_xxx0
3	16 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_xx00
4	32 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_x000
5	64 Kbytes	xxxx_xxxx_xxxx_xxxx_xxxx_0000
6	128 Kbytes	xxxx_xxxx_xxxx_xxxx_xxx0_0000
7	256 Kbytes	xxxx_xxxx_xxxx_xxxx_xx00_0000
8	512 Kbytes	xxxx_xxxx_xxxx_xxxx_x000_0000
9	1 Mbyte	xxxx_xxxx_xxxx_xxxx_0000_0000
10	2 Mbytes	xxxx_xxxx_xxxx_xxx0_0000_0000
11	4 Mbytes	xxxx_xxxx_xxxx_xx00_0000_0000
12	8 Mbytes	xxxx_xxxx_xxxx_x000_0000_0000
13	16 Mbytes	xxxx_xxxx_xxxx_0000_0000_0000
14	32 Mbytes	xxxx_xxxx_xxx0_0000_0000_0000
15	64 Mbytes	xxxx_xxxx_xx00_0000_0000_0000

Table 3-2. Physical descriptor base address (continued)

No.	Size	Physical base address [35:12]
16	128 Mbytes	xxxx_xxxx_x000_0000_0000_0000
17	256 Mbytes	xxxx_xxxx_0000_0000_0000_0000
18	512 Mbytes	xxxx_xxx0_0000_0000_0000_0000
19	1 Gbyte	xxxx_xx00_0000_0000_0000_0000
20	2 Gbytes	xxxx_x000_0000_0000_0000_0000
21	4 Gbytes	xxxx_0000_0000_0000_0000_0000

For example, a physical segment with a related virtual segment, which defines a segment size of 256 Kbytes:

1. According to the size (256 Kbytes), choose row number 7.
2. Replace the x with the msbs of the physical destination address.

3.2.2.3.2 Flexible segment programming model

The flexible segment programming model supports segment sizes of 4 Kbytes up to 16 Mbytes minus 256 Kbytes and allows for the definition of more flexible data and program segments with the following restrictions. For details, see Table 3-3.

Table 3-3. Flexible segment programming model

Segment size	Boundary restriction	Alignment restriction
4 Kbytes → 512 Kbytes–4 Kbytes	1 Mbyte	4 Kbytes
16 Kbytes → 2 Mbytes–16 Kbytes	4 Mbytes	16 Kbytes
64 Kbytes → 8 Mbytes–64 Kbytes	16 Mbytes	64 Kbytes
256 Kbytes → 16 Mbytes–256 Kbytes	64 Mbytes	256 Kbytes

Boundary restriction:

The virtual segment and physical segment must not cross multiples of the boundary restriction, according to the required row in the above table.

- Program virtual segment—M_PSDA_<i>.PSVBAS + M_PSDB_<i>.PSS
- Program physical segment—{M_PSDC_<i>.PSPBAS_EXT,M_PSDB_<i>.PSPBAS} + M_PSDB_<i>.PSS
- Data virtual segment—M_DSDA_<i>.DSVBAS + M_DSDB_<i>.DSS
- Data physical segment—{M_DSDC_<i>.DSPBAS_EXT,M_DSDB_<i>.DSPBAS} + M_DSDB_<i>.DSS

Alignment restriction:

The physical base address, virtual base address, and the size must be aligned according to the required row in the above table.

NOTE

The selection between the different cases (for the same size) is done automatically by the hardware to give the largest possible boundary according to the alignment of the given virtual base, physical base, and size.

3.2.2.4 Descriptors priority mechanism

The data and program MATT has a descriptors priority mechanism, allowing memory regions to overlap. This mechanism is handled by dividing the MATT into couples of descriptors. Each couple contains two sequential index descriptors. For example: descriptor 0 is coupled with descriptor 1, descriptor 2 is coupled with descriptor 3, etc. Each couple of descriptors has a fixed priority only between them. The priority is determined by the index number of the descriptor. The higher descriptor index number between the two coupled descriptors has a higher priority. For example: descriptor 3 has a higher priority over descriptor 2, but no priority over other descriptors. In case of double hit in two coupled descriptors, the higher priority descriptor will determined the attributes of the access. In case of double hit in two non-coupled descriptors, a double hit error occurs.

NOTE

If memory regions that are programmed in two coupled descriptors do not overlap, there is no priority relation between them.

The next MMU programming example, shown in Figure 3-8, illustrates the advantages of using the priority mechanism. For example, the user wants to define three memory regions, each with its unique sets of attributes:

- Address 0x0000–0xBFFF with attribute set A (48 Kbyte memory region)
- Address 0xC000–0xFFFF with attribute set B (16 Kbyte memory region)
- Address 0x10000–0x1FFFF with attribute set A (64 Kbyte memory region)

Without using the priority feature, this configuration requires four descriptors, for example:

- Descriptor 0: address 0x0000–0xBFFF, with attribute set A (48-Kbyte memory region)
- Descriptor 1: address 0xC000–0xFFFF, with attribute set B (16-Kbyte memory region)
- Descriptor 2: address 0x10000–0x1FFFF, with attribute set A (64-Kbyte memory region)

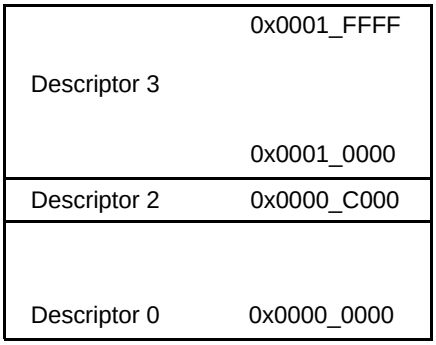

Figure 3-8. Descriptor priority mechanism not in use (example)

Using the descriptors priority character of two coupled descriptors requires only two descriptors, as shown in Figure 3-9. For example:

- Descriptor 0: address 0x0000–0x1FFFF, with attribute set A (128-Kbyte memory region)
- Descriptor 1: address 0xC000–0xFFFF, with attribute set B (16-Kbyte memory region)

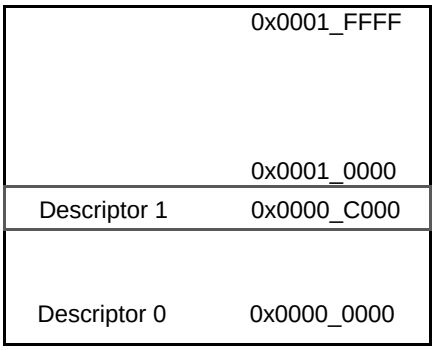

Figure 3-9. Using descriptors priority mechanism (example)

Because descriptor 0 and 1 are coupled and prioritized, overlapping between the memory regions is allowed. When the transaction accesses the memory region between 0xC000–0xFFFF, it receives attribute set B because descriptor 1 has a higher priority over descriptor 0. All accesses between 0x0000–0x1FFFF, except for 0xC000–0xFFFF, hit descriptor 0 only, and hence receive attribute set A.

NOTE

It is forbidden to map the same virtual address as shared and not shared, so both coupled descriptor should have the same shared attribute.

3.2.2.5 MATT programming model

Each MATT entry consists of three 32-bits registers (M_PSDA, M_PSDB, M_PSDC for program and M_DSDA, M_DSDB, M_DSDC for data). There is different programming model for write and read accesses to the MATT to ensure proper read and write of the MATT registers.

3.2.2.5.1 Read accesses from the MATT registers

MATT registers can be read directly by core accesses of 4 and 8 bytes or external master accesses of 4 bytes. The accesses must be aligned. 64-bit reads from the M_PSDA/M_DSDA register result in a simultaneous read from the M_PSDA + M_PSDB or M_DSDA + M_DSDB pairs. 64-bit reads cannot be performed from the M_PSDB/M_DSDB register address due to alignment restriction.

3.2.2.5.2 Write accesses to the MATT registers

The following preload registers perform MATT configuration.

There are two groups for program descriptors, and two groups for data descriptors configuration:

- M_PSDA_PL/M_PSDB_PL/M_PSDC_PL/M_PSDM_PL group are used for configuration of the program descriptors by the core, but not limited to the core.
- M_PSDA_PL_PB/M_PSDB_PL_PB/M_PSDC_PL_PB/M_PSDM_PL_PB group are used for configuration of the program descriptors by the peripheral bus by an external master, but not limited to it.
- M_DSDA_PL/M_DSDB_PL/M_DSDC_PL/M_DSDM_PL are used for configuration of the data descriptors by the core, but not limited to the core.
- M_DSDA_PL_PB/M_DSDB_PL_PB/M_DSDC_PL_PB/M_DSDM_PL_PB are used for configuration of the data descriptors by the peripheral bus by an external master, but not limited to it.

The following flow represents the core configuration of the data descriptors. Configuration of the program descriptors by the core and of the data/program descriptors by an external master work in a similar way.

The flow of the core's configuration of the data descriptors is as follows:

1. Write to M_DSDA_PL and write to M_DSDB_PL registers. This can be done by two 32 bits accesses or by a single 64 bit access. (External master access is limited to 32 bits)
2. Write to M_DSDC_PL and write to M_DSDM_PL registers. This can be done by two 32 bits accesses or by a single 64 bit access. (External master access is limited to 32 bits)
3. Write to the M_DSDM register result in updates of the following:
   — M_DSDA_<i> from the M_DSDA_PL register
   — M_DSDB_<i> from the M_DSDB_PL register
   — M_DSDC_<i> from the M_DSDC_PL register

Where <i> is defined in Register M_DSDM_PL. The update is enabled only if this descriptor <i> is owned by the source of the write access, in this example, if the core has ownership for this descriptor<i>. The descriptor ownership is defined in register M_DSD_MASK<i>. If the source of the write doesn't have ownership on the desired descriptor number, the descriptor will not be updated, and the MMU will send error indication on the Peripheral Bus and to the EPIC.

MMU is checking correct programming of MATT registers and indicates programming error in case of the wrong configuration values: violation of alignment or boundary restrictions in flexible programming model. The programming error status is reflected in M_PMPER and M_DMPER for program and data respectively. The descriptor is updated with the wrong configuration to enable easy debug of wrong

programming. In addition, the enable bit of such a descriptor is unconditionally cleared until its configuration is fixed and the error indication is sent on the Peripheral Bus and to the EPIC.

3.2.2.6 MATT configuration change indication

MMU provides MMU MATT configuration change (data and program) indication. This indication is sent to the DTU and captured there.

3.2.3 Attributes logic

The MMU allows the system designer to define different cache and external bus attributes for each memory region. When an access arrives on the internal core buses, the MMU searches for the access in the MATT descriptors and provides the region attributes related to the access.

MMU attributes support is used for the following:

- Cacheable memory and noncacheable memory
- Different write policies
- Memory coherency
- Guarded region
- Memory/peripheral region
- Different pre-fetch policies
- Different protection definitions
- L2 PID

There are some cases in which default attributes will be setting. This table describes the different cases attributes.

Table 3-4. Default attributes

Cases	Trans-lation	Peripheral	Cacheable	Guarded	WriteBack policy	Prefetch	L2PID	Coherency
Bank0	—	Yes	No	—	Yes	off	—	—
Segment Miss	No	Yes	No	Yes	Yes	off	zero	No
Barrier/Message	No	No	Yes	No	Yes	off	zero	Yes
Physical cache command access	No	No	Yes	No	Yes	off	zero	Yes
Core direct to B0	No	Yes	No	No	Yes	off	zero	No

The MMU attributes are prioritized, meaning that the lower priority attribute is being override whenever a higher attribute is set/cleared. For example, if the access is noncacheable, the prefetch policy will be no-prefetch. Access to peripheral space is always noncacheable. The MMU will fix and set the attributes

according to the priority, whenever it perform a MATT write configuration. Figure 3-10 illustrates the priority of the MMU attributes.

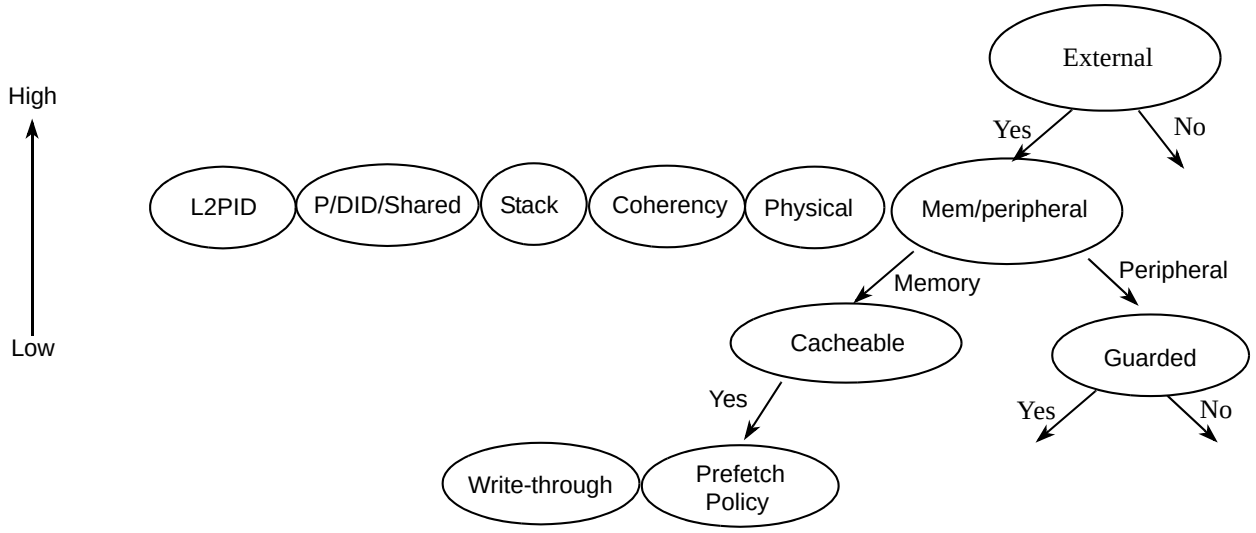

Figure 3-10. MMU attributes priority

3.2.4 Protection Unit

The MMU provides memory access permission checking for data reads, data writes, cache commands, and instruction fetches. The access permission definitions are programed in the MMU descriptors (data and program) for the related region.

When an access arrives on the internal XA/XB/RP buses, the MMU searches for the existence of the address in one of the regions described in the enabled memory descriptors. If an access hit the MATT, the MMU performs permission checking, based on the definition in the related descriptor. If the access is permitted, the MMU provides the related attributes for the access. If the access is not permitted or if there is a MATT miss, the memory access is killed and the MMU signals the core to abort. Indications describing the reason for the violation are captured and stored in the data/program status register.

NOTE

The core aborts only for non-speculative/taken-speculative access. If the speculative access is not taken, the core ignores the MMU abort request. The MMU captures error indications for a valid error only (non-speculative/taken-speculative access error), acknowledged by the core.

The protection unit is enabled or disabled by programming M_CR[MPE]. The reset value of the MPE is 0, meaning the protection unit is disabled at reset.

3.2.4.1 Protection violation

Non-permitted memory accesses are killed inside the subsystem, and a memory precise exception indication is sent back to the core. When a data protection violation occurs, the system does not change the state of the memory, and the access does not reflect to the external memory.

3.2.4.1.1 Task protection

The MMU provides a task protection mechanism. In the MMU descriptors, the memory boundaries of the task are programmed and limited. If the protection unit is enabled, accesses that do not hit the MATT are killed by the MMU, and a precise memory exception is generated.

3.2.4.1.2 Stack overrun protection

The MMU provides stack overrun protection if the feature is enabled (M_CR[SOEE] is set). The descriptor that includes stack memory inside it is marked with the stack attribute. Stack pointer based accesses that hit a non-stack descriptor results in a segment miss error. The access is killed and a precise exception is generated.

3.2.4.1.3 FVP subsystem protection

To protect the system, the MMU is responsible for the management of the subsystem and cluster control registers. The external SOC registers could be also protected by MMU if defined as peripheral in the MATT.

3.2.4.2 MMU errors and exceptions

This section discusses the following MMU errors and exceptions:

- Section 3.2.4.2.1, “Precise exceptions”
- Section 3.2.4.2.2, “Core data and program bus errors and exceptions”
- Section 3.2.4.2.3, “Cache commands support”
- Section 3.2.4.2.4, “ECC error support”
-
- Section 3.2.4.2.5, “Peripheral access size and alignment error”
- Section 3.2.4.2.7, “Peripheral write access error”
- Section 3.2.4.2.8, “MMU behavior on error”

3.2.4.2.1 Precise exceptions

The MMU exception to the core is a precise exception. The execution of the exception is not guaranteed, since the SC3900 core uses speculative accesses execution. Therefore, assertion of the exception request on a speculative execution not taken does not lead to an exception routine. On a taken MMU exception, the core aborts the transaction and jumps to the exception routine.

The MMU supports four types of precise interrupt requests: two for data and two for program. Each one leads to a different Interrupt Service Routine (ISR) address. The full (32-bit) ISR addresses are defined in

the M_PESRA0, M_PESRA1, M_DESRA0, and M_DESRA1 registers. Each MMU error can be related to one of two respective ISR addresses, according to configuration in M_PESRS and M_DESRS registers. Program exception is served when an MMU error/violation occurs in non-speculative/ taken-speculative dispatch attempt. Data exception is served when an MMU error/violation occurs in non-speculative/taken-speculative data access attempt.

3.2.4.2.2 Core data and program bus errors and exceptions

The MMU detects errors and exceptions on the core data and program buses. The errors are prioritized, and the MMU reflects only one error at a time. If there are two or more data/program errors, only the most recenterror per data/program domain is captured (that is, one for data and one for program). If there is an error on the two data buses at the same time, the error on bus XA (issued by LSU0 from the core) is captured for data. If an unaligned access crosses a segment boundary, it can obtain two different errors as it targets two memory regions. In this case, the first error is the one captured.

This table summarizes the errors on the core data buses and their severity.

Table 3-5. Core data bus errors

Error ID	Error/exception	Description	Severity
0000	Multiple Data Segment Descriptor Hit	When the data address matches more than one segment descriptor in non-coupled descriptors	Highest
0001	Segment Miss	When access does not match any of the enabled segment descriptors.	—
0010	Permission Violation	When a data access match a segment descriptor, but does not have sufficient permissions, as defined for that segment. (Bank 0 is always not-protected. Any master can access in Read or Write to Bank 0. Permission Violation Error will not occur on the accesses to Bank 0).	—
0011	Peripheral Access Error	For the accesses to peripheral memory: when the data address bus is not aligned to the access size. For the accesses to Bank0: when the data address bus is not aligned to the access size or the size is greater than 64 bits or smaller then 32 bits (except of QueryL1), or when the access is notify, or for queryL12. Notify access to cacheable area, or to Coherent area, or with Write-Through attributes, results in Error.	—
0101	Semaphore Access Size Error	For semaphore access when the data address bus is not aligned to the access size or the size is greater than 64 bits. For semaphore access to Bank0. Semaphore access must be with Write-Back policy. For noncacheable decoration access when the data address bus is not aligned to the access size or the size is greater than 64 bits.	—
0110	Stack Overrun	When stack based access matches non-stack descriptor.	—
0111	Noncacheable Cache Command	When core executes cache command to noncacheable memory (except for query).	—

Table 3-5. Core data bus errors (continued)

Error ID	Error/exception	Description	Severity
1000	Illegal access gathering	When the core executes Illegal access gathering: • Two read cache commands • 64 byte read, with another command to the cache (read, or read cache command). • Load_reserved with two reads. • Load with size 64B on bus B. • Load_reserved with size 64B. • Destructive load with size 64B.	—
1001	Noncacheable hit	When access with noncacheable attributes hits the cache.	—
1110	Data Read Error	When external data read access gets an error on the Rlink/Clink or peripheral bus.	—
1111	ECC	An ECC error reported by the data cache or internal memory.	Lowest

The core program bus errors are listed in this table.

Table 3-6. Core program bus errors

Error ID	Error/Exception	Description	Severity
0000	Multiple Program Segment Descriptor Hit	When the program address matches more than one segment descriptor in non-coupled descriptors	Highest
0001	Segment Miss	When access does not match any of the enabled segment descriptor.	—
0010	Permission Violation	When a program access match a segment descriptor, but does not have sufficient permissions, as defined for that segment	—
0100	Program Non-Aligned Access Error	SC3900 instructions are 16 bits (two bytes) and must be aligned. If the address on the program bus is not 16-bit aligned, a misaligned program error occurs.	—
0111	Noncacheable Cache Command	When core executes cache command to noncacheable memory (except for query).	—
1001	noncacheable hit	When access with noncacheable attributes hits the cache.	—
1110	Program Fetch Error	When instruction fetch gets an error on the Rlink/Clink bus	—
1111	ECC	An ECC error reported by the instruction cache or internal memory.	Lowest

3.2.4.2.3 Cache commands support

The MMU refers to data cache commands and program block cache commands as follows:

- The MMU cancels the following data cache commands issued by the core automatically when bit VCCC (fetch cache command cancel) in the M_CR is enabled:
  - DFETCHx
- VCCC bit in M_CR also controls fetch cache command generation in CME. CME can generate a voluntary cache command only when VCCC is cleared.
- The MMU always cancels the access and issues an error indication to the CME for all erroneous cache commands generated by CME and QUERY:

  - DFETCHx, DSYNC, DFLUSH, DINVALIDATE, DUNLOCK, DQUERY, PFETCHx, PINVALIDATE, PUNLOCK, PQUERY
- The MMU always cancels the access and issues an error indication to the CME for the following erroneous cache commands issued by the core:
  - DQUERY, PFETCHx, PINVALIDATE, PUNLOCK
- The MMU always cancels the access and issues an error indication to the core (precise error) for the following erroneous cache commands issued by the core.
  - DSYNC, DFLUSH, DINVALIDATE, DUNLOCK
- The MMU issues an error indication to the core (precise error) for the following erroneous cache commands issued by the core only if the bit VCEE (cache commands error enabled) in the M_CR is enabled and VCCC (cancel fetch commands enabled) in the M_CR is disabled. The erroneous access is always canceled independent of the VCCC, VCEE values.
  - DFETCHx
- If the core issues an erroneous data cache command, or error on Notify, M_DSR[DCCV] indicates the error. M_DSR[DAVD] will indicate if the error occurred on write cache command, Notify, or read cache command.

Table 3-7 summarizes erroneous cache commands behavior.

Table 3-7. Erroneous cache command behavior

Command	Condition for cancel	Precise error to the core	Error Indication to the CME
Commands issued by the core			
DFETCHx	Error or M_CR.VCCC = 1	Error and M_CR.VCCC = 0 and M_CR.VCEE = 1	—
DFLUSH	Error or noncacheable	Error	—
DSYNC	Error or noncacheable	Error	—
DINVALIDATE	Error or noncacheable	Error	—
DUNLOCK	Error or noncacheable	Error	—
DQUERY	Error	—	Error
PFETCHx	Error	—	Error
PINVALIDATE	Error	—	Error
PUNLOCK	Error	—	Error
Commands issued by the CME			
DFETCHx	Error	—	Error and M_CR.VCCC = 0
DFLUSH	Error	—	Error
DSYNC	Error	—	Error
DINVALIDATE	Error	—	Error

Table 3-7. Erroneous cache command behavior (continued)

Command	Condition for cancel	Precise error to the core	Error Indication to the CME
DUNLOCK	Error	—	Error
DQUERY	Error	—	Error
PFETCHx	Error	—	Error
PINVALIDATE	Error	—	Error
PUNLOCK	Error	—	Error
PQUERY	Error	—	Error

The MMU supports the following errors in cases of an erroneous cache command (for both program and data):

1. Multiple segment descriptor hit
2. Segment miss, when protection is enabled
3. Permission violation, if there is no read permission and protection is enabled.
4. Cache command to noncacheable memory (for granular data cache commands.

3.2.4.2.4 ECC error support

The SC3900 subsystem supports the ECC scheme by latching the access attributes causing the ECC error. The interrupt handler of the ECC exception can use this attributes for diagnostic and error correction. The MMU provides a configuration bit, M_CR[ECCEE] (ECC Exception Enable), to enable the ECC error diagnostic. This bit is cleared on reset, meaning that no exception is generated upon ECC error occurrence. Setting M_CR[ECCEE] invokes the ECC exception and error latching mechanism on ECC error.

The ECC has two types of precise interrupt requests: data and program. Each type of interrupt leads to a different interrupt vector. Program interrupts are served when an ECC error occurs in a non-speculative/taken-speculative dispatch attempt. A data interrupt is served when an ECC error occurs in a non-speculative/taken-speculative data read access attempt.

NOTE

The ECC error attributes are locked for the first ECC error and do not update unless M_CR[CMIR] is cleared.

3.2.4.2.5 Peripheral access size and alignment error

Accesses to the internal subsystem or cluster registers or any other memory with peripheral attribute must be aligned to their size. Additionally, the size of accesses to Bank0 space must not be greater than 64 bits or smaller then 32 bits (except of QueryL1). Additionally, write conditional/read reserved must not be to Bank0. Additionally, QueryL12 must not be to B0. Notify accesses must be to a non-cacheable area, and to a non-coherent area, and with Write-Back policy, and not to Bank0.

MMU kills the access and generates a precise memory exception for the accesses that violate the above restrictions.

3.2.4.2.6 Semaphore access size and alignment error

Semaphore accesses must be aligned to its size, and the size of the accesses must not be greater than 64 bits. Semaphore access must have Write-Back policy.

Decoration accesses to noncacheable memory must be aligned to its size, and the size of the accesses must not be greater than 64 bits.

MMU kills the access and generates a precise memory exception for accesses that violate the above restrictions.

3.2.4.2.7 Peripheral write access error

The MMU handles bus errors resulting from the core write accesses on the peripheral bus, or when a non-cacheable write hit the cache. The exception is non-precise and issued by the MMU to the EPIC. The latching error mechanism is handled by a unique control bit, M_NDSR[NDIR]. The error type is represented by a PWAE/NCWH bit in the M_NDSR. The error address is latched in M_NDVR.

3.2.4.2.8 MMU behavior on error

For a precise exception, the access address and attributes are latched in the MMU registers (refer to Table 3-8). The non-precise exception is level-triggered.

Table 3-8. MMU Behavior for a precise error

Error Type	Description	
Multiple Segment Descriptor Hit	A precise exception is generated and the access is killed.	
Miss on the MATT	If M_CR[MPE] is disabled, the access receives the default attributes values. No exception is involved and the access is not killed.	If M_CR[MPE] is enabled, a precise exception is generated and the access is killed.
Permission Violation	If MPE bit in the M_CR is disabled, no exception is involved and the access is not killed.	If MPE bit in the M_CR is enabled, a precise exception is generated and the access is killed.
Peripheral Access Size Error	A precise exception is generated and the access is killed.	
Semaphore Access Size Error	A precise exception is generated and the access is killed.	
Program Non-Align Access	A precise exception is generated and the access is killed.	
Stack Overrun	If M_CR[SOEE] is disabled no exception is involved and the access is not killed.	If M_CR[SOEE] is enabled, a precise exception is generated and the access is killed.
Noncacheable Cache Command	A precise exception is generated and the access is killed.	
noncacheable hit	A precise exception is generated and the access is killed.	

Table 3-8. MMU Behavior for a precise error (continued)

Error Type	Description	
Data Read, Peripheral Read, or Program Fetch Error	A precise exception is generated and the access is killed.	
Error Detection Code (ECC)	If M_CR[ECCEE] is disabled, no exception is involved.	If M_CR[ECCEE] is enabled, a precise exception is generated.

3.2.5 Task ID support

The core stores the currently running program ID (PID) and data ID (DID) in its registers. These registers are managed by the RTOS. The program or data ID together with the address from the core provide virtual address. The virtual address enables caches and MMU to distinguish between different tasks. The program ID consists of 8 bits, enables the SC3900 FVP subsystem, and supports up to 255 (1–255) program tasks (user/RTOS). The data ID consists of 8 bits, enabling the SC3900 FVP subsystem to support up to 255 (1–255) data tasks (user/RTOS).

3.2.5.1 System/shared task accesses

The MMU and cache system has a special mechanism to support system/shared task memory. Shared areas are configured by writing zero to the task ID field in the MATT. During descriptor match calculation for shared memory, the MMU disregards the PID/DID compare and makes an effective address compare only. Virtual address area defined as shared can not be defined as not shared with any task ID.

When a cache miss occurs for an access to a memory region marked as system/shared task in the MMU, the corresponding cache line has the shared attribute. If the cache line is shared, the caches ignore the task ID portion in the extended tag when comparing addresses for hit/miss detection. The cache verifies the physical address for a match and invalidates the line when there is no match to prevent aliasing. In this way, tasks with different task IDs can all view the same shared memory image in the cache.

3.2.5.1.1 Access to the non-shared memory space

When the MMU hardware identifies an access to a non-shared memory space, the cache system extends the effective address with the task ID to create a virtual address. The virtual address is used for comparison with the cache tags. In cases involving a miss, the task ID is saved as part of the tag.

3.2.5.1.2 Access to shared memory space

When the MMU hardware identifies an access to a system/shared space, the cache system sets the shared attribute to the cache line.

3.2.5.1.3 RTOS access to non-shared memory space

The RTOS may require the ability to access task-specific data written to task private memory space. This memory is not defined as shared. Therefore, images of it in the cache are task-extended with the ID of the

task. In order for the RTOS to access task memory without creating cache coherency problems, the RTOS should do it with correct ID. See the *SC3900 FVP Core Reference Manual* for details.

3.2.6 MMU query mechanism

The MMU query mechanism provides the MMU with information about a desired virtual address and generates translation and MATT queries.

- Translation query—makes a translation (physical address) of a required virtual address. This is useful for providing the physical address for devices not aware of the MMU address translation, and for debug, when one wants feedback on the address translation aspect of MMU MATT programming.
- MATT query—determines the related segment descriptor index for a desired virtual address. The MATT query is useful when virtual address-related attributes are needed—for example, access permissions.

To perform the MMU query, issue a DQUERY (for data) or PQUERY (for program) command, with the desired virtual address.

NOTE

Pay attention to the following while using the query mechanism:

- The data query supports a single query at a time. There must be no data access in parallel with the data query.
- On a query miss (virtual address is a miss in the MMU MATT), the data/program MATT miss status bit is set. The hit index filed is not valid.
- On a query command, when the virtual address matches more than one segment descriptor (in non-coupled descriptors), the data/program query status register reflects an error. The hit index field reflects the higher segment descriptor hit value. The query physical address register is not valid.

3.2.7 Multiple bank support

The data physical memory is divided into two banks. Banks are not programmable but rather determined in the MMU. Each bank represents a unique memory space and has its own chip select, as follows:

- Bank 0 represents register memory space that includes all memory mapped registers of the cluster. Bank 0 size is fixed and equal to 256 KByte.
- Bank 1: External memory space.

Bank0 is defined in the descriptor, in bit M_DSDC<i>[B0]. At reset, Bank0 is set to be in descriptor<0>, and it get the default attributes:

- Peripheral memory
- Shared

- Noncacheable
- No Guarded
- WriteBack
- Prefetch Off
- L2PID=0

Figure 3-11 illustrates the bank-related memory space for data access.

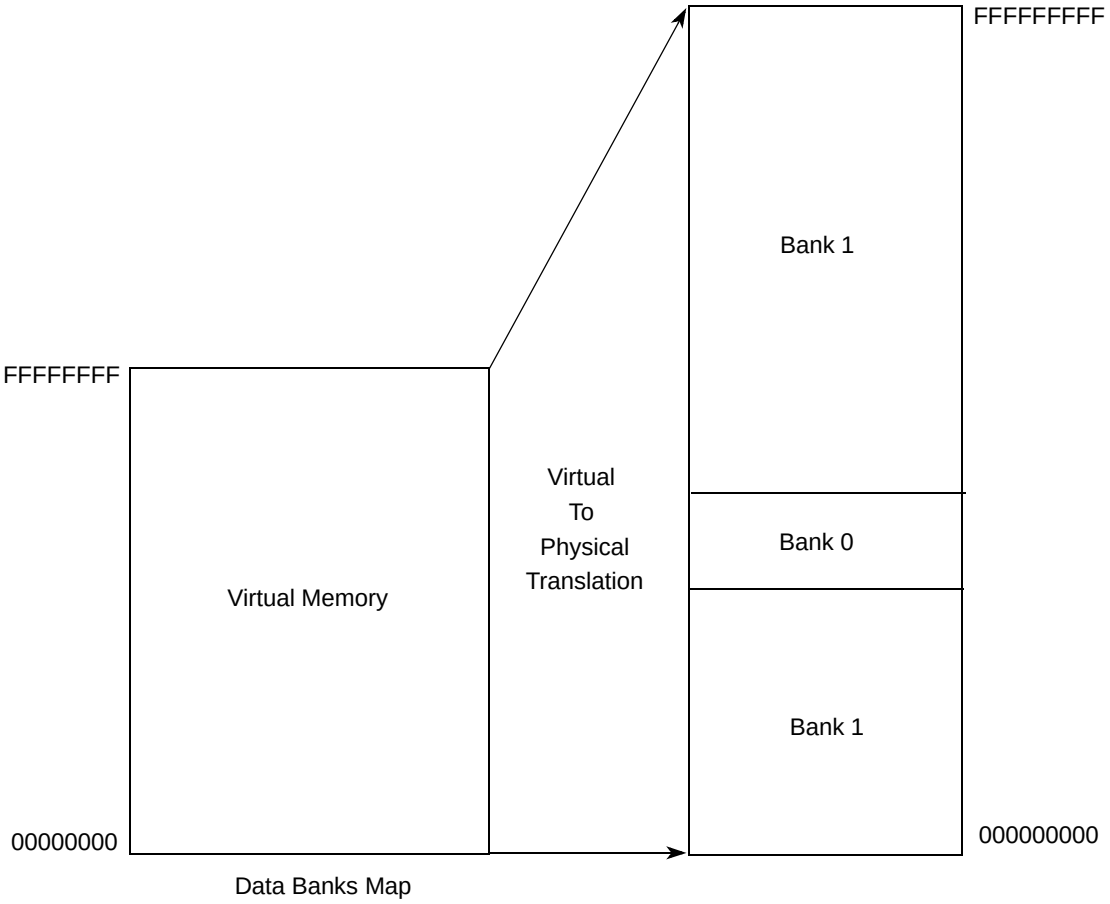

Figure 3-11. Multibank data memory space

All program physical memory is related to Bank1, or external memory. Figure 3-12 illustrates the bank's related memory space for a program access.

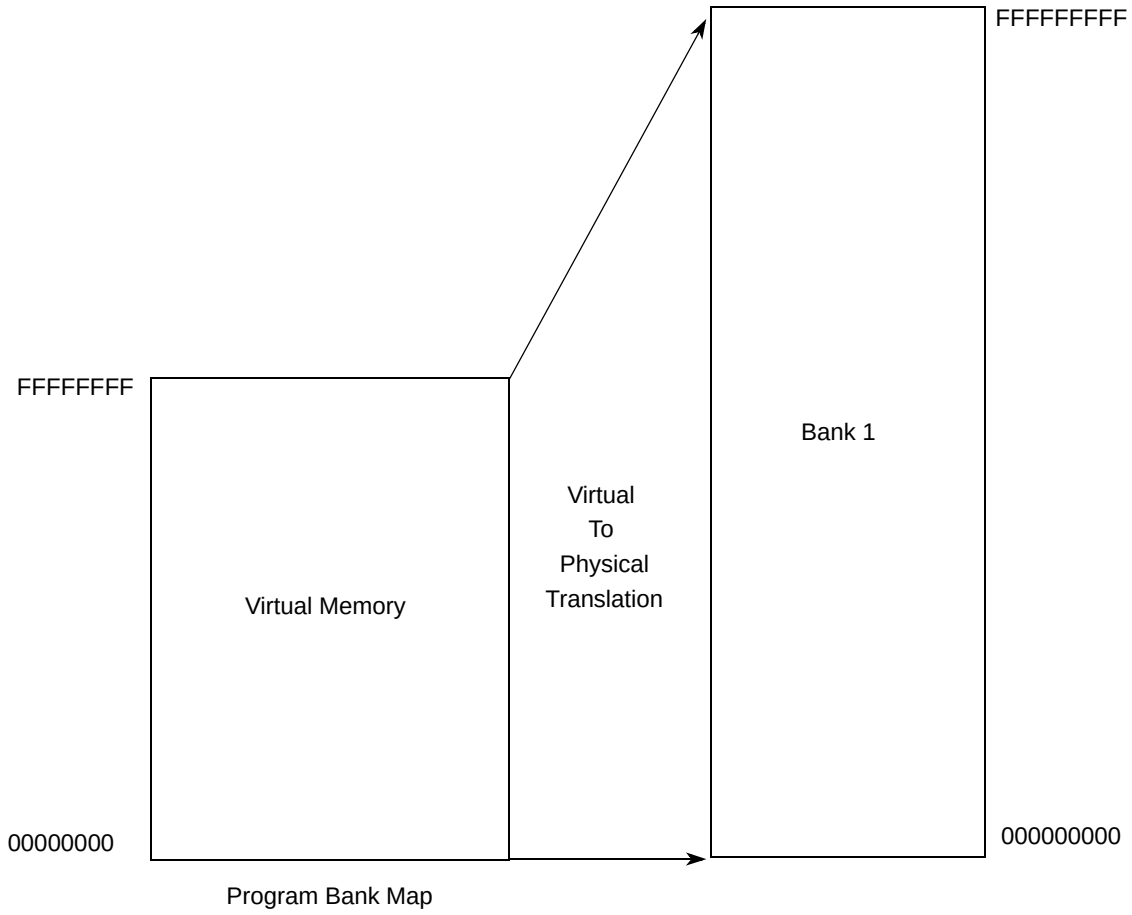

Figure 3-12. Single bank program memory space

3.2.7.1 FVP subsystem control registers

Most of the registers of the SC3900 FVP subsystem are placed in the subsystem control registers block of the MMU. Reads and writes to the control registers are performed through the peripheral bus. The MMU acts as a slave on the peripheral bus and belongs to bank 0.

3.2.7.2 Segment descriptors for different banks

The MMU requires defining segment descriptors for every memory space that is accessed by the core. This is true for all the banks. The difference is which attributes are programmable and which are predefined for each bank. Access that is done to the bank 0 is always noncacheable with a peripheral memory area attribute. These two attributes override any configuration in the relevant descriptor. All other attributes are

be generated according to the descriptor configuration. Bank 1 does not have attributes restrictions. The access to it acquires attributes according to descriptor configuration.

3.3 Memory map and register definition

The MMU registers are memory-mapped, and are accessed through the peripheral bus. These registers are accessible via a standard move in the normal or exception working modes. All registers are 32 bits, but every two sequential registers could be accessed by a single aligned 64 bits access.

Table 3-9 describes the MMU memory map.

Table 3-9. MMU memory map

Address offset from MMU base	Use	Section/page
General MMU registers		
000	MMU Control Register (M_CR)	3.3.1/3-37
004	Data Violation PC Register (M_DVPC)	3.3.2/3-38
008	Data Violation Address Register (M_DVA)	3.3.3/3-39
00C	Data Status Register (M_DSR)	3.3.4/3-40
010	Non-precise Data Violation Address Register (M_NDVR)	3.3.5/3-42
014	Non-precise Data Error Status Register (M_NDSR)	3.3.6/3-42
018	Platform Information Register (M_PIR)	3.3.7/3-43
01C	Reserved	
020	Doorbell Level Register (M_DBL)	3.3.8/3-44
024	Doorbell Edge Register (M_DBE)Doorbell Edge Register (M_DBE)	3.3.9/3-44
028	Data Exception Service Routine Address0 (M_DESRA0)	3.3.10/3-45
02C	Data Exception Service Routine Address1 (M_DESRA1)	3.3.11/3-46
030	Data Exception Service Routine Select (M_DESRS)	3.3.12/3-46
034	Message Process ID Register (MSG_PIR)	3.3.13/3-48
038	Message Guest Process ID Register (MSG_GPIR)	3.3.14/3-48
03C	Message Logical Process ID Register (MSG_LPIDR)	3.3.15/3-49
040	CCSR Base Address Register (CCSR_BASE)	3.3.16/3-49
Data Control Registers		
080	Data Segment Descriptor VAL (M_DSDVAL)	3.3.17/3-50
084	Data Segment Descriptor MASK (M_DSDMASK)	3.3.18/3-50
088	Core Preload Register for Data Segment Descriptor A (M_DSDA_PL)	3.3.19/3-51

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
08C	Core Preload Register for Data Segment Descriptor B (M_DSDB_PL)	3.3.20/3-51
090	Core Preload Register for Data Segment Descriptor C (M_DSDC_PL)	3.3.21/3-52
094	Core Preload Register for Data Segment Descriptor M (M_DSDM_PL)	3.3.22/3-52
098	Slave Preload Register for Data Segment Descriptor A (M_DSDA_PL_PB)	3.3.23/3-53
09C	Slave Preload Register for Data Segment Descriptor B (M_DSDB_PL_PB)	3.3.24/3-53
0A0	Slave Preload Register for Data Segment Descriptor C (M_DSDC_PL_PB)	3.3.25/3-54
0A4	Slave Preload Register for Data Segment Descriptor M (M_DSDM_PL_PB)	3.3.26/3-54
0A8	Data MATT Programming Error Register (M_DMPER)	3.3.27/3-55
0AC	SGB Watermark preload value (M_WMCFG)	3.3.28/3-55
Data MATT		
200	Data Segment Descriptor Registers A0 (M_DSDA0)	3.3.30/3-57
204	Data Segment Descriptor Registers B0 (M_DSDB0)	3.3.31/3-59
208	Data Segment Descriptor Registers C0 (M_DSDC0)	3.3.32/3-61
20C–20F	Reserved	
210	Data Segment Descriptor Registers A1 (M_DSDA1)	3.3.30/3-57
214	Data Segment Descriptor Registers B1 (M_DSDB1)	3.3.31/3-59
218	Data Segment Descriptor Registers C1 (M_DSDC1)	3.3.32/3-61
21C–21F	Reserved	
220	Data Segment Descriptor Registers A2 (M_DSDA2)	3.3.30/3-57
224	Data Segment Descriptor Registers B2 (M_DSDB2)	3.3.31/3-59
228	Data Segment Descriptor Registers C2 (M_DSDC2)	3.3.32/3-61
22C–22F	Reserved	
230	Data Segment Descriptor Registers A3 (M_DSDA3)	3.3.30/3-57
234	Data Segment Descriptor Registers B3 (M_DSDB3)	3.3.31/3-59
238	Data Segment Descriptor Registers C3 (M_DSDC3)	3.3.32/3-61

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
23C–23F	Reserved	
240	Data Segment Descriptor Registers A4 (M_DSDA4)	3.3.30/3-57
244	Data Segment Descriptor Registers B4 (M_DSDB4)	3.3.31/3-59
248	Data Segment Descriptor Registers C4 (M_DSDC4)	3.3.32/3-61
24C–24F	Reserved	
250	Data Segment Descriptor Registers A5 (M_DSDA5)	3.3.30/3-57
254	Data Segment Descriptor Registers B5 (M_DSDB5)	3.3.31/3-59
258	Data Segment Descriptor Registers C5 (M_DSDC5)	3.3.32/3-61
25C–25F	Reserved	
260	Data Segment Descriptor Registers A6 (M_DSDA6)	3.3.30/3-57
264	Data Segment Descriptor Registers B6 (M_DSDB6)	3.3.31/3-59
268	Data Segment Descriptor Registers C6 (M_DSDC6)	3.3.32/3-61
26C–26F	Reserved	
270	Data Segment Descriptor Registers A7 (M_DSDA7)	3.3.30/3-57
274	Data Segment Descriptor Registers B7 (M_DSDB7)	3.3.31/3-59
278	Data Segment Descriptor Registers C7 (M_DSDC7)	3.3.32/3-61
27C–27F	Reserved	
280	Data Segment Descriptor Registers A8 (M_DSDA8)	3.3.30/3-57
284	Data Segment Descriptor Registers B8 (M_DSDB8)	3.3.31/3-59
288	Data Segment Descriptor Registers C8 (M_DSDC8)	3.3.32/3-61
28C–28F	Reserved	
290	Data Segment Descriptor Registers A9 (M_DSDA9)	3.3.30/3-57
294	Data Segment Descriptor Registers B9 (M_DSDB9)	3.3.31/3-59
298	Data Segment Descriptor Registers C9 (M_DSDC9)	3.3.32/3-61
29C–29F	Reserved	
2A0	Data Segment Descriptor Registers A10 (M_DSDA10)	3.3.30/3-57
2A4	Data Segment Descriptor Registers B10 (M_DSDB10)	3.3.31/3-59
2A8	Data Segment Descriptor Registers C10 (M_DSDC10)	3.3.32/3-61

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
2AC–2AF	Reserved	
2B0	Data Segment Descriptor Registers A11 (M_DSDA11)	3.3.30/3-57
2B4	Data Segment Descriptor Registers B11 (M_DSDB11)	3.3.31/3-59
2B8	Data Segment Descriptor Registers C11 (M_DSDC11)	3.3.32/3-61
2BC–2BF	Reserved	
2C0	Data Segment Descriptor Registers A12 (M_DSDA12)	3.3.30/3-57
2C4	Data Segment Descriptor Registers B12 (M_DSDB12)	3.3.31/3-59
2C8	Data Segment Descriptor Registers C12 (M_DSDC12)	3.3.32/3-61
2CC–2CF	Reserved	
2D0	Data Segment Descriptor Registers A13 (M_DSDA13)	3.3.30/3-57
2D4	Data Segment Descriptor Registers B13 (M_DSDB13)	3.3.31/3-59
2D8	Data Segment Descriptor Registers C13 (M_DSDC13)	3.3.32/3-61
2DC–2DF	Reserved	
2E0	Data Segment Descriptor Registers A14 (M_DSDA14)	3.3.30/3-57
2E4	Data Segment Descriptor Registers B14 (M_DSDB14)	3.3.31/3-59
2E8	Data Segment Descriptor Registers C14 (M_DSDC14)	3.3.32/3-61
2EC–2EF	Reserved	
2F0	Data Segment Descriptor Registers A15 (M_DSDA15)	3.3.30/3-57
2F4	Data Segment Descriptor Registers B15 (M_DSDB15)	3.3.31/3-59
2F8	Data Segment Descriptor Registers C15 (M_DSDC15)	3.3.32/3-61
2FC–2FF	Reserved	
300	Data Segment Descriptor Registers A16 (M_DSDA16)	3.3.30/3-57
304	Data Segment Descriptor Registers B16 (M_DSDB16)	3.3.31/3-59
308	Data Segment Descriptor Registers C16 (M_DSDC16)	3.3.32/3-61
30C–30F	Reserved	
310	Data Segment Descriptor Registers A17 (M_DSDA17)	3.3.30/3-57
314	Data Segment Descriptor Registers B17 (M_DSDB17)	3.3.31/3-59
318	Data Segment Descriptor Registers C17 (M_DSDC17)	3.3.32/3-61

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
31C–31F	Reserved	
320	Data Segment Descriptor Registers A18 (M_DSDA18)	3.3.30/3-57
324	Data Segment Descriptor Registers B18 (M_DSDB18)	3.3.31/3-59
328	Data Segment Descriptor Registers C18 (M_DSDC18)	3.3.32/3-61
32C–32F	Reserved	
330	Data Segment Descriptor Registers A19 (M_DSDA19)	3.3.30/3-57
334	Data Segment Descriptor Registers B19 (M_DSDB19)	3.3.31/3-59
338	Data Segment Descriptor Registers C19 (M_DSDC19)	3.3.32/3-61
33C–33F	Reserved	
340	Data Segment Descriptor Registers A20 (M_DSDA20)	3.3.30/3-57
344	Data Segment Descriptor Registers B20 (M_DSDB20)	3.3.31/3-59
348	Data Segment Descriptor Registers C20 (M_DSDC20)	3.3.32/3-61
34C–34F	Reserved	
350	Data Segment Descriptor Registers A21 (M_DSDA21)	3.3.30/3-57
354	Data Segment Descriptor Registers B21 (M_DSDB21)	3.3.31/3-59
358	Data Segment Descriptor Registers C21 (M_DSDC21)	3.3.32/3-61
35C–35F	Reserved	
360	Data Segment Descriptor Registers A22 (M_DSDA22)	3.3.30/3-57
364	Data Segment Descriptor Registers B22 (M_DSDB22)	3.3.31/3-59
368	Data Segment Descriptor Registers C22 (M_DSDC22)	3.3.32/3-61
36C–36F	Reserved	
370	Data Segment Descriptor Registers A23 (M_DSDA23)	3.3.30/3-57
374	Data Segment Descriptor Registers B23 (M_DSDB23)	3.3.31/3-59
378	Data Segment Descriptor Registers C23 (M_DSDC23)	3.3.32/3-61
37C–37F	Reserved	
380	Data Segment Descriptor Registers A24 (M_DSDA24)	3.3.30/3-57
384	Data Segment Descriptor Registers B24 (M_DSDB24)	3.3.31/3-59
388	Data Segment Descriptor Registers C24 (M_DSDC24)	3.3.32/3-61

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
38C–38F	Reserved	
390	Data Segment Descriptor Registers A25 (M_DSDA25)	3.3.30/3-57
394	Data Segment Descriptor Registers B25 (M_DSDB25)	3.3.31/3-59
398	Data Segment Descriptor Registers C25 (M_DSDC25)	3.3.32/3-61
39C–39F	Reserved	
3A0	Data Segment Descriptor Registers A26 (M_DSDA26)	3.3.30/3-57
3A4	Data Segment Descriptor Registers B26 (M_DSDB26)	3.3.31/3-59
3A8	Data Segment Descriptor Registers C26 (M_DSDC26)	3.3.32/3-61
3AC–3AF	Reserved	
3B0	Data Segment Descriptor Registers A27 (M_DSDA27)	3.3.30/3-57
3B4	Data Segment Descriptor Registers B27 (M_DSDB27)	3.3.31/3-59
3B8	Data Segment Descriptor Registers C27 (M_DSDC27)	3.3.32/3-61
3BC–3BF	Reserved	
3C0	Data Segment Descriptor Registers A28 (M_DSDA28)	3.3.30/3-57
3C4	Data Segment Descriptor Registers B28 (M_DSDB28)	3.3.31/3-59
3C8	Data Segment Descriptor Registers C28 (M_DSDC28)	3.3.32/3-61
3CC–3CF	Reserved	
3D0	Data Segment Descriptor Registers A29 (M_DSDA29)	3.3.30/3-57
3D4	Data Segment Descriptor Registers B29 (M_DSDB29)	3.3.31/3-59
3D8	Data Segment Descriptor Registers C29 (M_DSDC29)	3.3.32/3-61
3DC–3DF	Reserved	
3E0	Data Segment Descriptor Registers A30 (M_DSDA30)	3.3.30/3-57
3E4	Data Segment Descriptor Registers B30 (M_DSDB30)	3.3.31/3-59
3E8	Data Segment Descriptor Registers C30 (M_DSDC30)	3.3.32/3-61
3EC–3EF	Reserved	
3F0	Data Segment Descriptor Registers A31 (M_DSDA31)	3.3.30/3-57
3F4	Data Segment Descriptor Registers B31 (M_DSDB31)	3.3.31/3-59
3F8	Data Segment Descriptor Registers C31(M_DSDC31)	3.3.32/3-61

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
Program Control Registers		
800	Program Violation Address Register (M_PVA)	3.3.33/3-62
804	Program Status Register (M_PSR)	3.3.34/3-63
808-80F	Reserved	
810	Program Exception Service Routine Address0 (M_PESRA0)	3.3.35/3-64
814	Program Exception Service Routine Address1 (M_PESRA1)	3.3.36/3-64
818	Program Exception Service Routine Select (M_PESRS)	3.3.37/3-65
81C	Reserved	
820	Program Segment Descriptor VAL (M_PSDVAL)	3.3.38/3-66
824	Program Segment Descriptor MASK (M_PSDMASK)	3.3.39/3-66
828	Core Preload Register for Program Segment Descriptor A (M_PSDA_PL)	3.3.40/3-67
82C	Core Preload Register for Program Segment Descriptor B (M_PSDB_PL)	3.3.41/3-67
830	Core Preload Register for Program Segment Descriptor C (M_PSDC_PL)	3.3.42/3-68
834	Core Preload Register for Program Segment Descriptor M (M_PSDM_PL)	3.3.43/3-68
838	Slave Preload Register for Program Segment Descriptor A (M_PSDA_PL_PB)	3.3.44/3-69
83C	Slave Preload Register for Program Segment Descriptor B (M_PSDB_PL_PB)	3.3.45/3-69
840	Slave Preload Register for Program Segment Descriptor C (M_PSDC_PL_PB)	3.3.46/3-70
844	Slave Preload Register for Program Segment Descriptor M (M_PSDM_PL_PB)	3.3.47/3-70
848	Program MATT Programming Error Register (M_PMPER)	3.3.48/3-71
Program MATT		
A00	Program Segment Descriptor Registers A0 (M_PSDA0)	3.3.49/3-71
A04	Program Segment Descriptor Registers B0 (M_PSDB0)	3.3.50/3-74
A08	Program Segment Descriptor Registers C0 (M_PSDC0)	3.3.51/3-76
A0C–A0F	Reserved	
A10	Program Segment Descriptor Registers A1 (M_PSDA1)	3.3.49/3-71
A14	Program Segment Descriptor Registers B1 (M_PSDB1)	3.3.50/3-74

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
A18	Program Segment Descriptor Registers C1 (M_PSDC1)	3.3.51/3-76
A1C–A1F	Reserved	
A20	Program Segment Descriptor Registers A2 (M_PSDA2)	3.3.49/3-71
A24	Program Segment Descriptor Registers B2 (M_PSDB2)	3.3.50/3-74
A28	Program Segment Descriptor Registers C2 (M_PSDC2)	3.3.51/3-76
A2C–A2F	Reserved	
A30	Program Segment Descriptor Registers A3 (M_PSDA3)	3.3.49/3-71
A34	Program Segment Descriptor Registers B3 (M_PSDB3)	3.3.50/3-74
A38	Program Segment Descriptor Registers C3 (M_PSDC3)	3.3.51/3-76
A3C–A3F	Reserved	
A40	Program Segment Descriptor Registers A4 (M_PSDA4)	3.3.49/3-71
A44	Program Segment Descriptor Registers B4 (M_PSDB4)	3.3.50/3-74
A48	Program Segment Descriptor Registers C4 (M_PSDC4)	3.3.51/3-76
A4C–A4F	Reserved	
A50	Program Segment Descriptor Registers A5 (M_PSDA5)	3.3.49/3-71
A54	Program Segment Descriptor Registers B5 (M_PSDB5)	3.3.50/3-74
A58	Program Segment Descriptor Registers C5 (M_PSDC5)	3.3.51/3-76
A5C–A5F	Reserved	
A60	Program Segment Descriptor Registers A6 (M_PSDA6)	3.3.49/3-71
A64	Program Segment Descriptor Registers B6 (M_PSDB6)	3.3.50/3-74
A68	Program Segment Descriptor Registers C6 (M_PSDC6)	3.3.51/3-76
A6C–A6F	Reserved	
A70	Program Segment Descriptor Registers A7 (M_PSDA7)	3.3.49/3-71
A74	Program Segment Descriptor Registers B7 (M_PSDB7)	3.3.50/3-74
A78	Program Segment Descriptor Registers C7 (M_PSDC7)	3.3.51/3-76
A7C–A7F	Reserved	
A80	Program Segment Descriptor Registers A8 (M_PSDA8)	3.3.49/3-71
A84	Program Segment Descriptor Registers B8 (M_PSDB8)	3.3.50/3-74

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
A88	Program Segment Descriptor Registers C8 (M_PSDC8)	3.3.51/3-76
A8C–A8F	Reserved	
A90	Program Segment Descriptor Registers A9 (M_PSDA9)	3.3.49/3-71
A94	Program Segment Descriptor Registers B9 (M_PSDB9)	3.3.50/3-74
A98	Program Segment Descriptor Registers C9 (M_PSDC9)	3.3.51/3-76
A9C–A9F	Reserved	
AA0	Program Segment Descriptor Registers A10 (M_PSDA10)	3.3.49/3-71
AA4	Program Segment Descriptor Registers B10 (M_PSDB10)	3.3.50/3-74
AA8	Program Segment Descriptor Registers C10 (M_PSDC10)	3.3.51/3-76
AAC–AAF	Reserved	
AB0	Program Segment Descriptor Registers A11 (M_PSDA11)	3.3.49/3-71
AB4	Program Segment Descriptor Registers B11 (M_PSDB11)	3.3.50/3-74
AB8	Program Segment Descriptor Registers C11 (M_PSDC11)	3.3.51/3-76
ABC–ABF	Reserved	
AC0	Program Segment Descriptor Registers A12 (M_PSDA12)	3.3.49/3-71
AC4	Program Segment Descriptor Registers B12 (M_PSDB12)	3.3.50/3-74
AC8	Program Segment Descriptor Registers C12 (M_PSDC12)	3.3.51/3-76
ACC–ACF	Reserved	
AD0	Program Segment Descriptor Registers A13 (M_PSDA13)	3.3.49/3-71
AD4	Program Segment Descriptor Registers B13 (M_PSDB13)	3.3.50/3-74
AD8	Program Segment Descriptor Registers C13 (M_PSDC13)	3.3.51/3-76
ADC–ADF	Reserved	
AE0	Program Segment Descriptor Registers A14 (M_PSDA14)	3.3.49/3-71
AE4	Program Segment Descriptor Registers B14 (M_PSDB14)	3.3.50/3-74
AE8	Program Segment Descriptor Registers C14 (M_PSDC14)	3.3.51/3-76
AEC–AEF	Reserved	
AF0	Program Segment Descriptor Registers A15 (M_PSDA15)	3.3.49/3-71

Table 3-9. MMU memory map (continued)

Address offset from MMU base	Use	Section/page
AF4	Program Segment Descriptor Registers B15 (M_PSDB15)	3.3.50/3-74
AF8	Program Segment Descriptor Registers C15 (M_PSDC15)	3.3.51/3-76

3.3.1 MMU Control Register (M_CR)

Figure 3-13 shows the MMU control register.

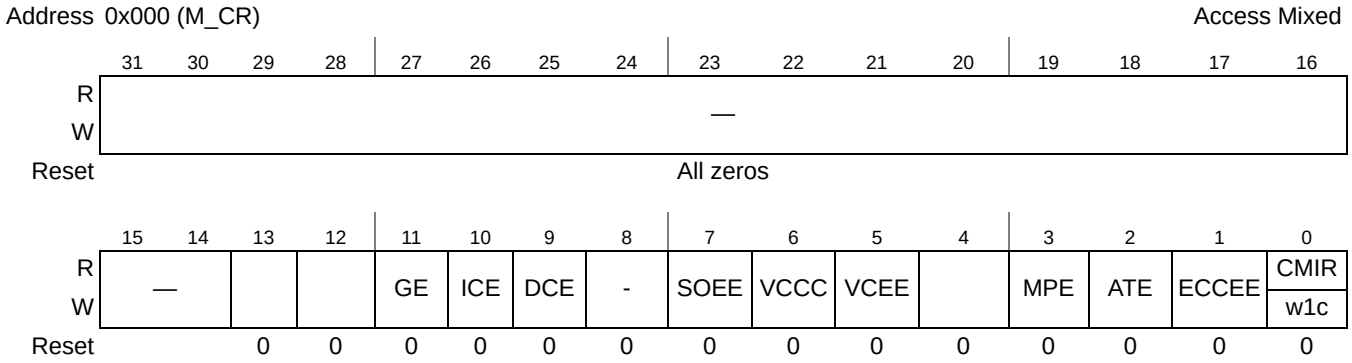

Figure 3-13. MMU Control Register

Table 3-10 describes the M_CR fields.

Table 3-10. M_CR bit descriptions

Field	Description
31–12	Reserved
11 GE	Gather Enable 0 SGB doesn't gather write accesses 1 SGB gathering function is enabled
10 ICE	Instruction Cache Enable 0 Instruction Cache is disabled. This bit overrides ICache and L2 cache policy to noncacheable for all instruction accesses. 1 Instruction Cache is enabled. Instruction accesses cache policy determined by relevant MMU descriptor. This bit is not a multiple-use enable/disable bit. The ICE can only be enabled once. To disable instruction cache again, the SC3900 FVP subsystem must be reset.
9 DCE	Data Cache Enable 0 Data Cache is disabled. This bit overrides DCache and L2 cache policy to noncacheable for all data accesses. 1 Data Cache is enabled. Data accesses cache policy determined by relevant MMU descriptor. This bit is not a multiple-use enable/disable bit. This bit is not a multiple-use enable/disable bit. The DCE can only be enabled once. To disable the data cache again, the SC3900 FVP subsystem must be reset.
7 SOEE	Stack Overrun Error Enabled If set, the stack related access that matches non-stack descriptor results in precise exception. 0 Disable 1 Enable

Table 3-10. M_CR bit descriptions (continued)

Field	Description
6 VCCC	Fetch Cache Commands Cancel If set, the MMU and CME cancel the following cache performance commands: DFETCHx. 0 No Cancel 1 Cancel
5 VCEE	Voulantary Cache Commands Error Enabled If set, generates precise exception on error on granular DFETCHx commands, unless VCCC bit is enabled. If clear, the MMU closes the access inside the platform, but does not inform the core of the error. 0 Disable 1 Enable
3 MPE	Memory Protection Enable A central bit that enables/disables the protection-checking function in all enabled segment descriptors. It also enables/disables the miss interrupt support on a miss access. The reset value is configured according to an external FVP subsystem plug. 0 Disable 1 Enable
2 ATE	Address Translation Enable Enables or disables the address translation mechanism. If disabled, addresses are not translated (such as, from a virtual address to a physical address). The reset value is configured according to external FVP subsystem plug. 0 Disable 1 Enable
1 ECCEE	Error Detection Code Exception Enable Enables the ECC exception. This bit is disabled after reset. 0 Disable 1 Enable
0 CMIR	Clear MMU Interrupt Request This bit is set for the first valid MMU/ECC error of any type. While this bit is set, the MMU does not change the status bits and the memory violation information in the M_PSR, M_DSR, M_PVA, M_DVA registers. Write this bit as 1 to clear the M_PSR, M_PVA, M_DSR, and M_DVA registers and to enable the MMU to latch the information for a new violation. 0 No pending reserved exception 1 Exception

3.3.2 Data Violation PC Register (M_DVPC)

M_DVPC, shown in Figure 3-14, is cleared when M_CR[CMIR] is set.

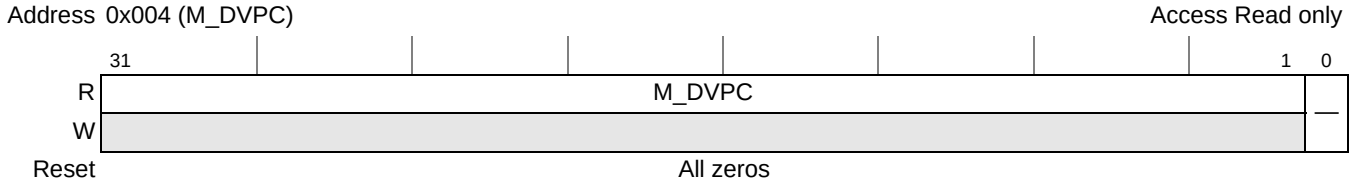

Figure 3-14. Data Violation PC Register (M_DVPC)

Table 3-11 describes the M_DVPC fields.

Table 3-11. M_DVPC bit descriptions

Field	Description
31–1 M_DVPC	PC of VLES with Data Violation If one of the data error status bits in the M_DSR register is set, this register contains the PC of violating VLES. M_DVPC is updated for the first non-permitted memory address. It is not updated unless the CMIR bit is clear. **Note:** Non-permitted memory addresses that arrive after the first address violation are not saved.
0	Reserved

3.3.3 Data Violation Address Register (M_DVA)

M_DVA, shown in Figure 3-15, is cleared when M_CR[CMIR] is set. For a double error on the XA/XB buses, the M_DVA register reflects the most severe error. For a double error of equal severity on the XA/XB buses, M_DVA reflects the XA error.

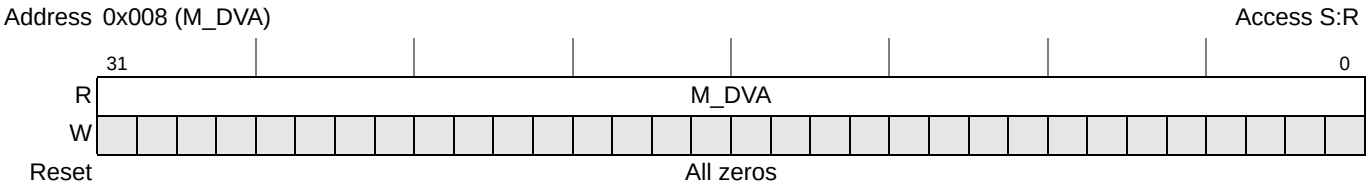

Figure 3-15. Data Violation Address Register (M_DVA)

Table 3-12 describes the M_DVA fields.

Table 3-12. M_DVA bit descriptions

Field	Description
31–0 M_DVA	Data Violation Address If one of the data error status bits in the M_DSR register is set, this register contains the violating data address. If the error occurred on a cross 4k boundary access (M_DSR[DAAL]=2'b01,2'b10), the captured address will be of the second part of the access. M_DVA is updated for the first non-permitted memory address. It is not updated unless the CMIR bit is clear. **Note:** Non-permitted memory addresses that arrive after the first address violation are not saved.

3.3.4 Data Status Register (M_DSR)

M_DSR status bits do not change as long as the M_CR[CMIR] is set. Setting CMIR reset M_DSR. M_DSR reflects only one error at a time. For two or more errors, only the most severe error is represented. Figure 3-16 shows the data status register (M_DSR).

Address 0x00C (M_DSR) Access Read only

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	DID								DAAL		DCCV	DAVW			DAVD	-
W																
Reset	All zeros															1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	—												DEID			
W																
Reset	All zeros															

Figure 3-16. Data Status Register (M_DSR)

Table 3-13 describes the M_DSR fields.

Table 3-13. M_DSR bit descriptions

Field	Description
31–24 DID	Data ID: Holds the access Data ID.
23–22 DAAL	Data Access Alignment 00 Access doesn't cross 4k boundary 01 Access cross 4k boundary and results in two memory transactions. Error on first transaction. 10 Access cross 4k boundary and results in two memory transactions. Error on second transaction. 11 Reserved
21 DCCV	Data Cache Command Violation Set when a data cache command or Notify violation occurred. 0 Read/Write 1 Data cache command, or Notify violation
20–18 DAVW	Data Access Violation Width Holds the access width of the violating access on the data bus. 000 1 byte 001 2 bytes 010 4 bytes 011 8 bytes 100 16 bytes 101 32 bytes 110 64 bytes 111 Reserved
17 DAVD	Data Access Violation Direction Set when the data access violation involves a write/write_cache_command/Notify. Clear when the data access violation involves a read/read_cache_command. 0 Read/Read_cache_command 1 Write/Write_cache_command/Notify

Table 3-13. M_DSR bit descriptions (continued)

Field	Description
15–4	Reserved
3–0 DEID	Data Error ID: 0000: Data Multiple Segment Descriptor Hit • Set when the data address matches more than one enabled data segment descriptors in non-coupled descriptors (for details, refer to Section 3.2.2.4, "Descriptors priority mechanism"). 0001: Data Segment Miss • Set when a data access does not match any of the enabled data segment descriptors or stack related access matches non-stack descriptor while M_CR[SOEE] is set and the protection unit is enabled (M_CR[MPE] is set). 0010: Data Permission Violation • Set when a data access matches a data segment descriptor but does not have sufficient permissions, as defined for that segment, and the protection unit is enabled (MCR[MPE] is set). 0011: Peripheral Access Size Error • Set when a data access with peripheral attribute is non-aligned to its size or the size of Bank0 access is greater than 64 bits. 0101: Semaphore Access Size Error • Set when semaphore access is unaligned to its size or the size is greater than 64 bits. 0110: Stack Overrun Violation • Set when stack based data access matches non-stack descriptor while M_CR[SOEE] is set. 0111: Noncacheable Cache Command • Set when core executes cache command to noncacheable memory. 1000: Illegal access gathering Error. 1001 NC hit: • Set when data access that has noncacheable attribute hits the cache 1110: Data Read Error • Set when the data read access gets an error on the RLink/CLink or peripheral bus interface. 1111: Data ECC • Set when there is a data read access with an ECC error and the M_CR[ECCEE] field is enabled.

NOTE

Please note the following conditions:

- As long as M_CR[CMIR] is set, the MMU does not change the status bits in the M_DSR.
- Setting M_CR[CMIR] clears the M_DSR register.
- The data status register (M_DSR) reflects only one error at a time. If there are two or more errors, only the most severe error is represented (bits 0–6 are mutually exclusive). For details, refer to Section 3.2.4.2.2, "Core data and program bus errors and exceptions."
- If there is a double error on the XA/XB buses, the M_DSR register reflects the most severe error.
- If there is a double error on the XA/XB buses with equal severity, the M_DSR register reflects the XA error.

3.3.5 Non-precise Data Violation Address Register (M_NDVR)

M_NDVR status bits do not change if M_NDSR[NDIR] is set. Figure 3-17 shows the non-precise data violation address register.

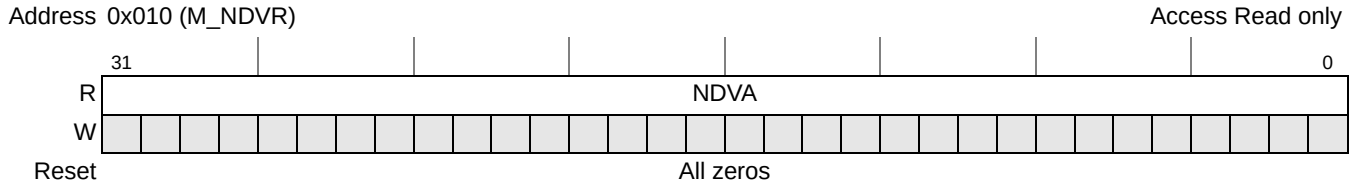

Figure 3-17. Non-precise Data Violation Address Register (M_NDVR)

Table 3-14 describes the M_NDVR fields

Table 3-14. M_NDVR bit descriptions

Field	Description
31–0 NDVA	Non-precise data violation address If M_NDSR[NDIR] is set and one of the error status bits in the M_NDSR is set, this register contains the error address. M_NDVR is updated for the first memory address error. It is not updated unless the NDIR bit is clear. In case the error is non-cacheable write hit—the M_NDVA contains 32 bits of the memory address error, In case the error is peripheral write error—the M_NDVA contains only the 18 LSB bits of the memory address error.

3.3.6 Non-precise Data Error Status Register (M_NDSR)

Figure 3-18 shows the non-precise data error status register.

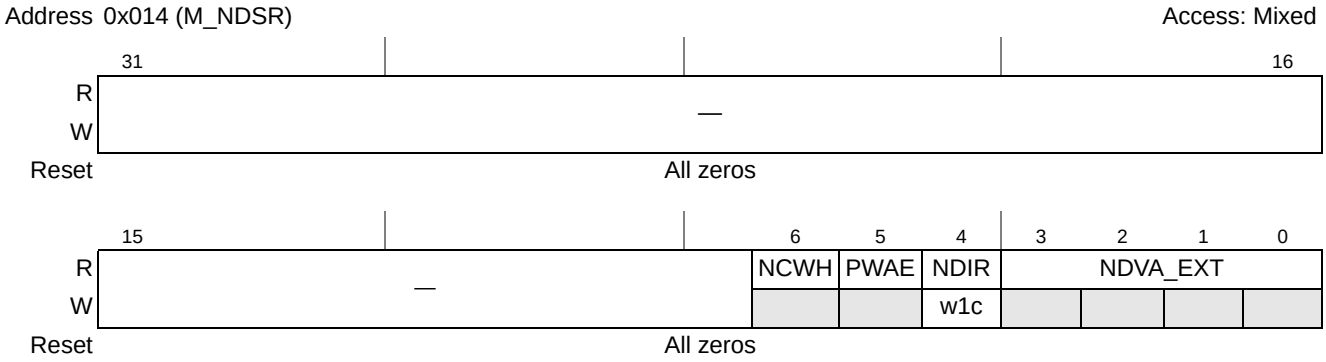

Figure 3-18. Non-precise Data Error Status Register (M_NDSR)

Table 3-15 describes the M_NDSR fields.

Table 3-15. M_NDSR Bit Descriptions

Field	Description
32–7	Reserved
6 NCWH	Non-cacheable Write Hit Error 0 - No Non-cacheable Write hit Error 1 - A Non-cacheable Write hit the Cache Error

Table 3-15. M_NDSR Bit Descriptions (continued)

Field	Description
5 PWAE	Peripheral Write Access Error Set when the Bank0 peripheral write access gets an error. 0 No Peripheral Write Error 1 Peripheral Write Error
4 NDIR	Non-precise Data Error Interrupt Request Set when non-precise data error occurs. This bit is set for the first Error Interrupt Request. As long as this bit is set, the MMU does not change the status bits of this register. Set NDIR to negate the interrupt request and to clear the status fields in the M_NDSR. This enables the MMU to latch the information for a new violation. Errors that occur while NDIR bit is set are not captured. 0 No pending error exception 1 Error exception
3–0 NDVA_EXT	Non-precise Data Violation Address Extension If M_NDSR[NDIR] is set and one of the error status bits in the M_NDSR is set, this register contains the 4 MSBs of error address. M_NDVR is updated for the first memory address error. It is not updated unless the NDIR bit is clear. M_NDVR contains the rest of the bits. In case the error is non-cacheable write hit—the NDVA_EXT are the 4 MSBs of error address. In case the error is peripheral write error—the NDVA_EXT are equal to zero.

3.3.7 Platform Information Register (M_PIR)

M_PIR, shown in Figure 3-19, is configured at production and contains FVP subsystem information that software may need for identification and version control. The reset value is hard-wired based on FVP subsystem instantiation.

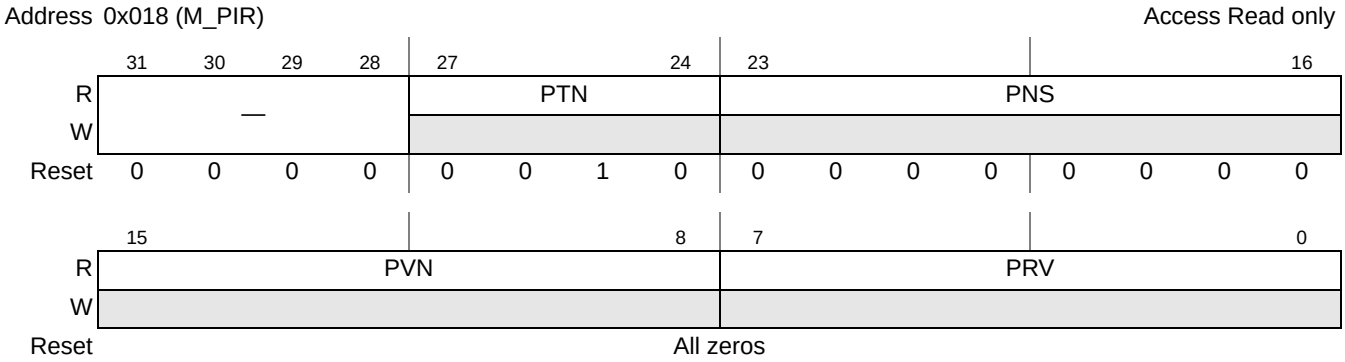

Figure 3-19. Platform Information Register (M_PIR)

Table 3-16 describes the M_PIR fields.

Table 3-16. M_PIR bit descriptions

Field	Description
31–28	Reserved
27–24 PTN	Platform Type Number Contains a unique value for each new FVP subsystem type. 0x02 SC3900 FVP subsystem

Table 3-16. M_PIR bit descriptions (continued)

Field	Description
23–16 PNS	Platform Number in the System Contains a unique value for each FVP subsystem. For a system with multiple FVP subsystems, this field contains a different number for each FVP subsystem. This field enables the software to execute conditionally, based on the core in which it is instanced (for example, for defining the boot processor).
15–8 PVN	Platform Version Number Contains the version of the FVP subsystem, which defines the top-level functional features, such as the blocks that are included, memory size, and so on. 0x00 SC3900 FVP Subsystem
7–0 PRV	Platform Revision Number Contains the revision of the FVP subsystem, which is an incremental change under the scope of one version. Different revisions may include bug corrections, and so on. 0x00 SC3900 FVP Subsystem

3.3.8 Doorbell Level Register (M_DBL)

Figure 3-20 shows the doorbell level register.

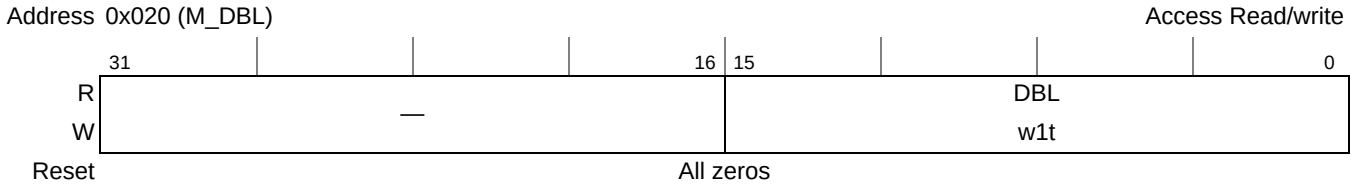

Figure 3-20. Doorbell Level Register (M_DBL)

Table 3-17 describes the M_DBL fields.

Table 3-17. M_DBL bit descriptions

Field	Description
31–16	Reserved
15–0 DBL	Doorbell Level Register. General pins output from the SC3900 FVP subsystem. Bits 0–15 are used for intercore interrupt mesh issuing. Write ‘1’ to bit n (n = 0..15) to toggle its value. Writing a ‘0’ to bit n does not change it.

3.3.9 Doorbell Edge Register (M_DBE)

Figure 3-21 shows the doorbell edge register.

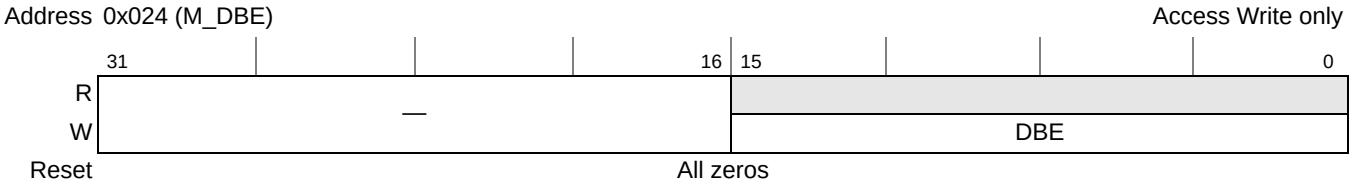

Figure 3-21. Doorbell Edge Register (M_DBE)

Table 3-18 describes the M_DBE fields.

Table 3-18. M_DBE bit descriptions

Field	Description
31–16	Reserved
15–0 DBE	Doorbell Edge Register. Writing 1 results in generating 16 cycles pulse in the appropriate output from the SC3900 FVP subsystem.

3.3.10 Data Exception Service Routine Address0 (M_DESRA0)

M_DESRA0, shown in Figure 3-22, contains the address for data service routine 0.

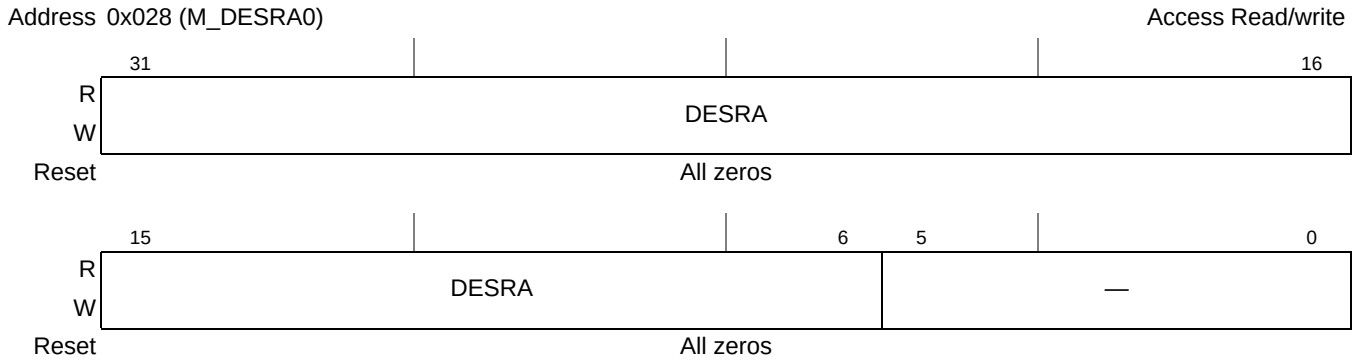

Figure 3-22. Data Exception Service Routine Address0 (M_DESRA0)

Table 3-19 describes the M_DESRA0 fields.

Table 3-19. M_DESRA0 bit descriptions

Field	Description
31–6 DESRA	Data Exception Service Routine Address 0
5–0	Reserved

3.3.11 Data Exception Service Routine Address1 (M_DESRA1)

M_DESRA1, shown in Figure 3-23, contains the address for data service routine 1.

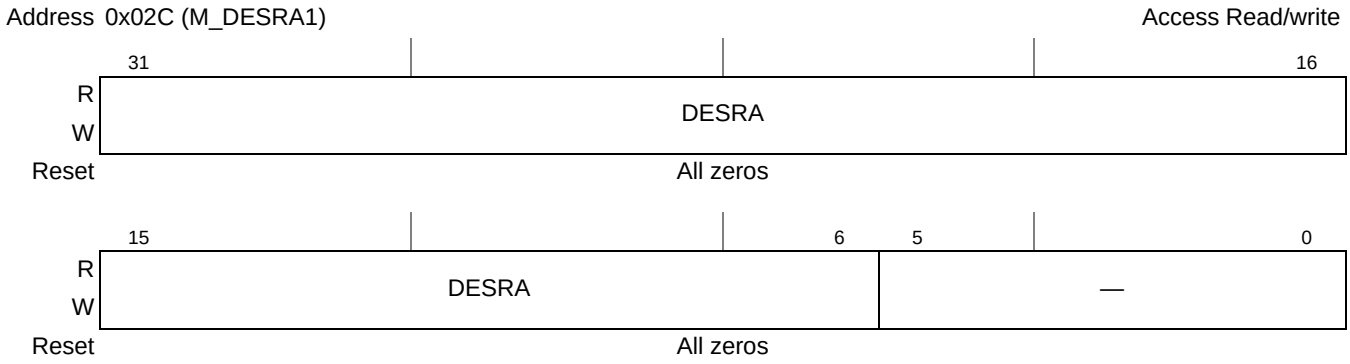

Figure 3-23. Data Exception Service Routine Address1 (M_DESRA1)

Table 3-20 describes the M_DESRA1 fields.

Table 3-20. M_DESRA1 bit descriptions

Field	Description
31–6 DESRA	Data Exception Service Routine Address 1
5–0	Reserved

3.3.12 Data Exception Service Routine Select (M_DESRS)

M_DESRS, shown in Figure 3-24, contains the mapping of data exceptions to service routine addresses.

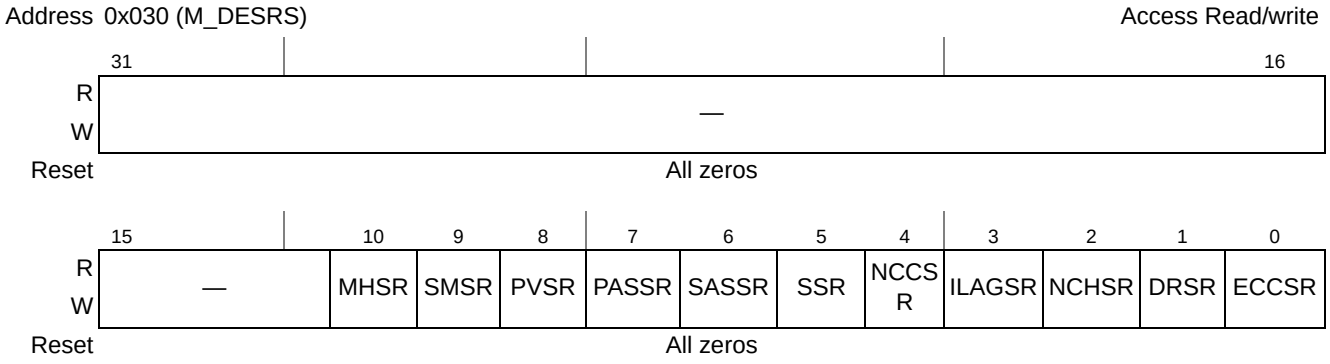

Figure 3-24. Data Exception Service Routine Select (M_DESRS)

Table 3-21 describes the M_DESRS fields.

Table 3-21. M_DESRS bit descriptions

Field	Description
31–11	Reserved
10 MHSR	Multiple segment descriptor hit Service Routine Select Selects service routine address for multiple segment descriptor hit exception. 0 M_DESRA0 1 M_DESRA1
9 SMSR	Segment Miss Exception Service Routine Select Selects service routine address for segment miss exception. 0 M_DESRA0 1 M_DESRA1
8 PVSR	Permission Exception Service Routine Select Selects service routine address for permission exception. 0 M_DESRA0 1 M_DESRA1
7 PASSR	Peripheral Access Size Exception Service Routine Select Selects service routine address for peripheral access size access exception. 0 M_DESRA0 1 M_DESRA1
6 SASSR	Semaphore Access Size Exception Service Routine Select Selects service routine address for peripheral access size access exception. 0 M_DESRA0 1 M_DESRA1
5 SSR	Stack Overrun Exception Service Routine Select Selects service routine address for stack overrun exception. 0 M_DESRA0 1 M_DESRA1
4 NCCSR	Noncacheable Cache Command Service Routine Select. Selects service routine address for Noncacheable Cache Command exception. 0 M_DESRA0 1 M_DESRA1
3 ILAGSR	Illegal access gathering Service Routine Select. Selects service routine address for illegal access gathering exception. 0 M_DESRA0 1 M_DESRA1
2 NCHSR	Noncacheable Hit Service Routine Select Selects service routine address for noncacheable hit exception. 0 M_DESRA0 1 M_DESRA1

Table 3-21. M_DESRS bit descriptions (continued)

Field	Description
1 DRSR	Data Read Exception Service Routine Select Selects service routine address for data read exception. 0 M_DESRA0 1 M_DESRA1
0 ECCSR	ECC Error Service Routine Select Selects service routine address for ECC exception. 0 M_DPESRA0 1 M_DESRA1

3.3.13 Message Process ID Register (MSG_PIR)

The MSG_PIR is read/write register, and can be read/write at any time. At reset the MSG_PIR sample a plug of the PIR bits. The MSG_PIR bits are reflected at the MMU output pins. Figure 3-25 describes the Message Process ID.

Figure 3-25. Message Process ID Register

Table describes the MSG_PIR fields.

Table 3-22. MSG_PIR bit description

Field	Description
31-14	Reserved
13–0 PIR	Contains Message Process ID value.

3.3.14 Message Guest Process ID Register (MSG_GPIR)

The MSG_GPIR is read/write register, and can be read/write at any time. At reset the MSG_GPIR sample a plug of the GPIR bits. The MSG_GPIR bits are reflected at the MMU output pins. Figure 3-26 describes the Message Guest Process ID.

Figure 3-26. Message Guest Process ID Register

Table describes the MSG_GPIR fields.

Table 3-23. MSG_GPIR bit description

Field	Description
31-14	Reserved
13–0 GPIR	Contains Message Guest Process ID value.

3.3.15 Message Logical Process ID Register (MSG_LPIDR)

The MSG_LPIDR is read/write register, and can be read/write at any time. At reset the MSG_LPIDR sample a plug of the LPIDR bits. The MSG_LPIDR bits are reflected at the MMU output pins. Figure 3-27, describe the Message Logical Process ID.

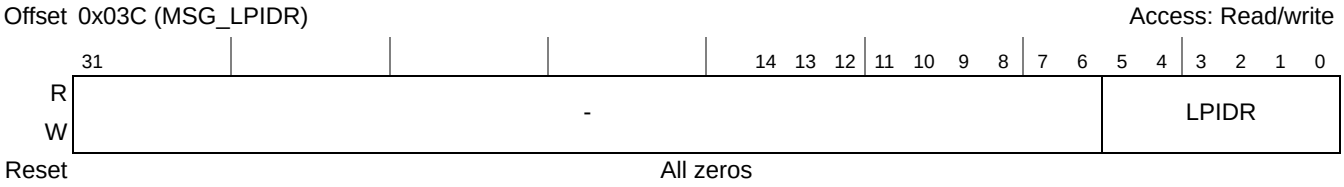

Figure 3-27. Message Logical Process ID Register

Table describes the MSG_LPIDR fields.

Table 3-24. MSG_LPIDR bit description

Field	Description
31-6	Reserved
5–0 LPIDR	Contains Message Logical Process ID value.

3.3.16 CCSR Base Address Register (CCSR_BASE)

CCSR_BASE, shown in Figure 3-28, contains the CCSR Base Address.

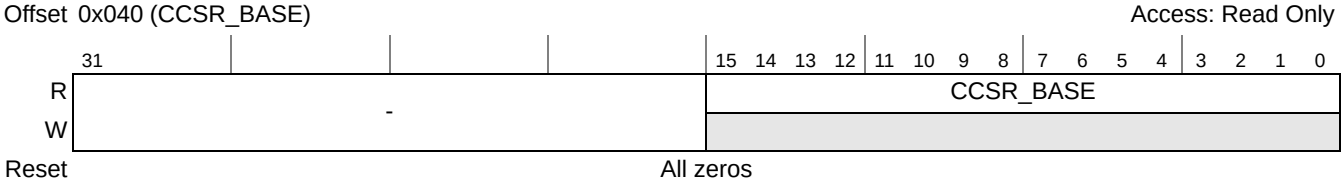

Figure 3-28. CCSR Base Address Register

Table 3-25 describes the CCSR_BASE fields.

Table 3-25. CCSR_BASE bit description

Field	Description
31-15	Reserved
15–0 CCSR_BASE	Hold the CCSR Base Address. (MSB bits 39-24).

3.3.17 Data Segment Descriptor VAL (M_DSDVAL)

M_DSDVAL, shown in Figure 3-29, contains 1 bit per data segment descriptor. When a source is writing to this register, each descriptor that is owned by this source (the ownership of each descriptors, is defined by the M_DSDMASK), will be set/clear the enable bit of the descriptor accordingly. By clustering all bits together, the RTOS can activate all the relevant data segments using one instruction.

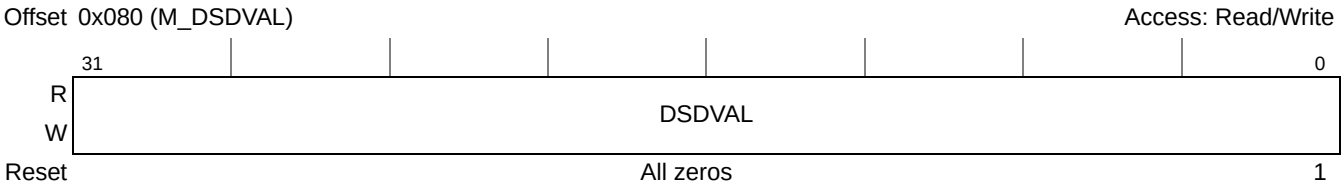

Figure 3-29. Data Segment Descriptor Value (M_DSDVAL)

Table 3-26 describes the M_DSDVAL fields.

Table 3-26. M_DSDVAL bit descriptions

Field	Description
31–0 DSDVAL*n*	Data Segment Descriptor Value. Each of these bits is an alias of the respective DE bit in the respective M_DSDA*n* register, For Example: bit 0 is related to M_DSDA 0. The state of the bit is observable and controllable from either path. If set, this data segment descriptor is enabled. 0 Clear the enable bit of the related data segment descriptor. 1 Enable bit of the related data segment descriptor is set.

3.3.18 Data Segment Descriptor MASK (M_DSDMASK)

M_DSDMASK, shown in Figure 3-30, contains 1 bit per data segment descriptor. Each bit define who has the ownership of the correlate data descriptor.

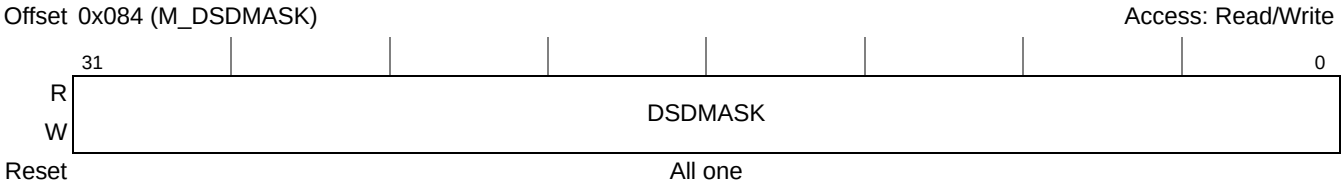

Figure 3-30. Data Segment Descriptor Mask (M_DSDMASK)

Table 3-27 describes the M_DSDEC bit fields.

Table 3-27. M_DSDMASK Bit Descriptions

Field	Description
31–0 DSDMASK*n*	Data Segment Descriptor MASK 0 The ownership on this data descriptor is not for the core. 1 The Core has the ownership on this data descriptor.

3.3.19 Core Preload Register for Data Segment Descriptor A (M_DSDA_PL)

M_DSDA_PL, shown in Figure 3-31, is a preload register that is used to configure data segment descriptor A registers.

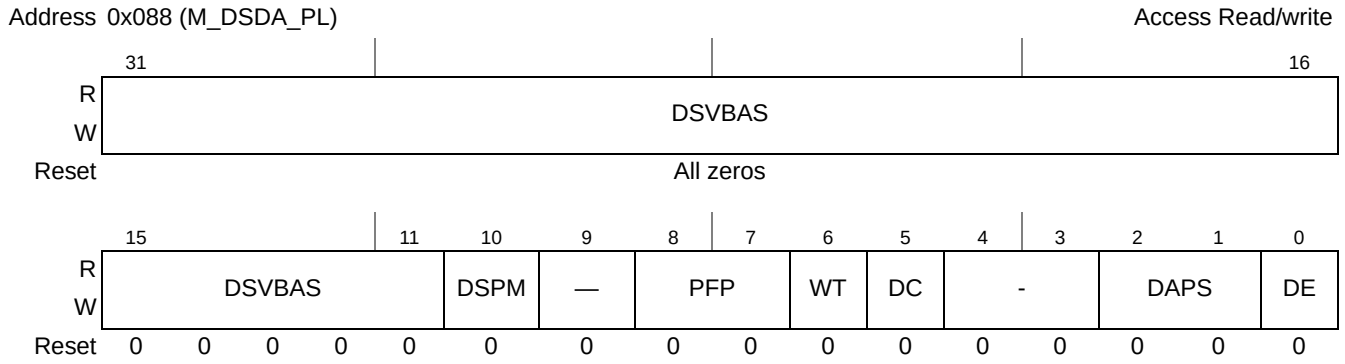

Figure 3-31. Core Preload Register for Data Segment Descriptor Registers A (M_DSDA_PL)

See Table 3-33 for the description of the M_DSDA_PL fields.

3.3.20 Core Preload Register for Data Segment Descriptor B (M_DSDB_PL)

M_DSDB_PL, shown in Figure 3-32, is a preload register that is used to configure data segment descriptor B registers.

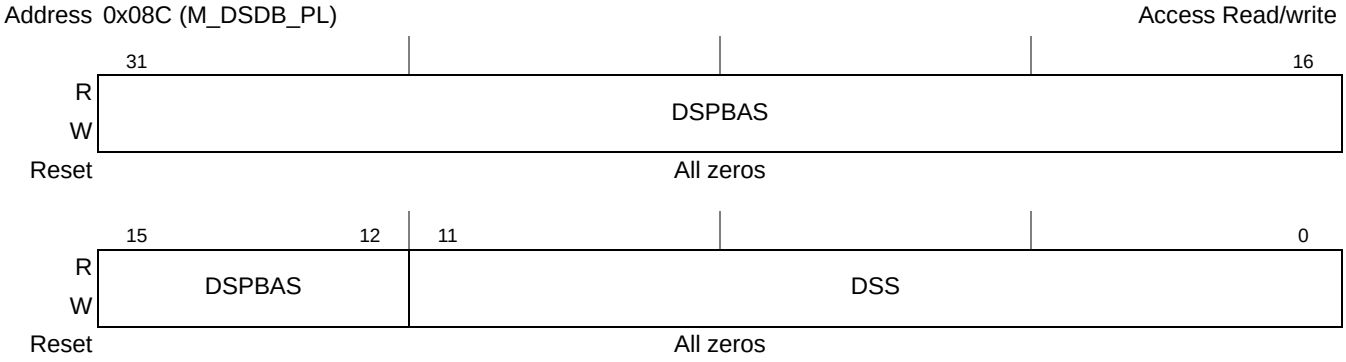

Figure 3-32. Core Preload Register for Data Segment Descriptor Registers B (M_DSDB_PL)

See Table 3-37 for the description of the M_DSDB_PL fields.

3.3.21 Core Preload Register for Data Segment Descriptor C (M_DSDC_PL)

M_DSDC_PL, shown in Figure 3-33, is a preload register that is used to configure data segment descriptor C registers.

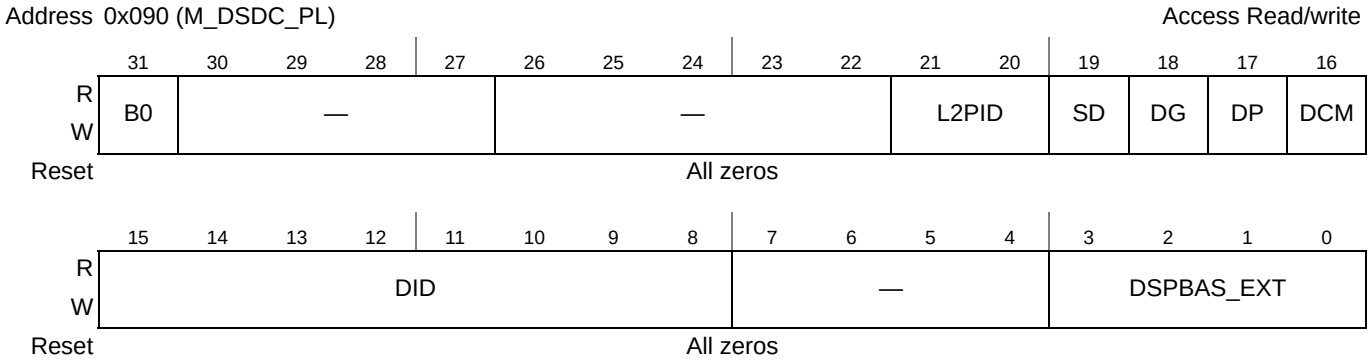

Figure 3-33. Data Segment Descriptor Registers C (M_DSDC_PL)

See Table 3-40 for the description of the M_DSDC_PL fields.

3.3.22 Core Preload Register for Data Segment Descriptor M (M_DSDM_PL)

M_DSDM_PL, shown in Figure 3-34, is a preload register that is used to configure data segment descriptor M registers.

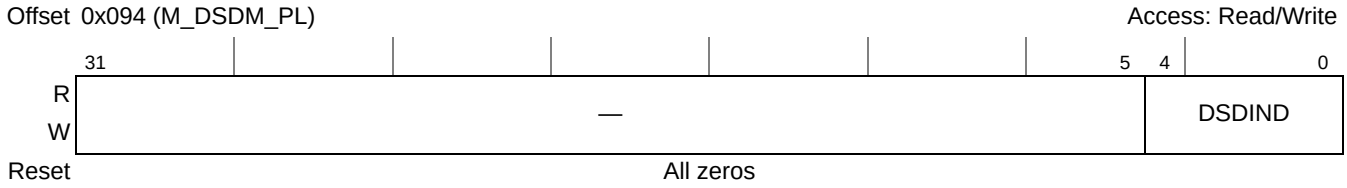

Figure 3-34. Core Preload Register for Data Segment Descriptor M (M_DSDM_PL)

Table 3-28 describes the M_DSDM_PL fields.

Table 3-28. M_DSDM_PL register bit description

Field	Description
31–5	Reserved
4–0 (DSDIND)	00000 Config Data Descriptor 0 00001 Config Data Descriptor 1 00010 Config Data Descriptor 2 ... 11111 Config Data Descriptor 31

3.3.23 Slave Preload Register for Data Segment Descriptor A (M_DSDA_PL_PB)

M_DSDA_PL_PB, shown in Figure 3-35, is a preload register that is used to configure the data segment descriptor A registers.

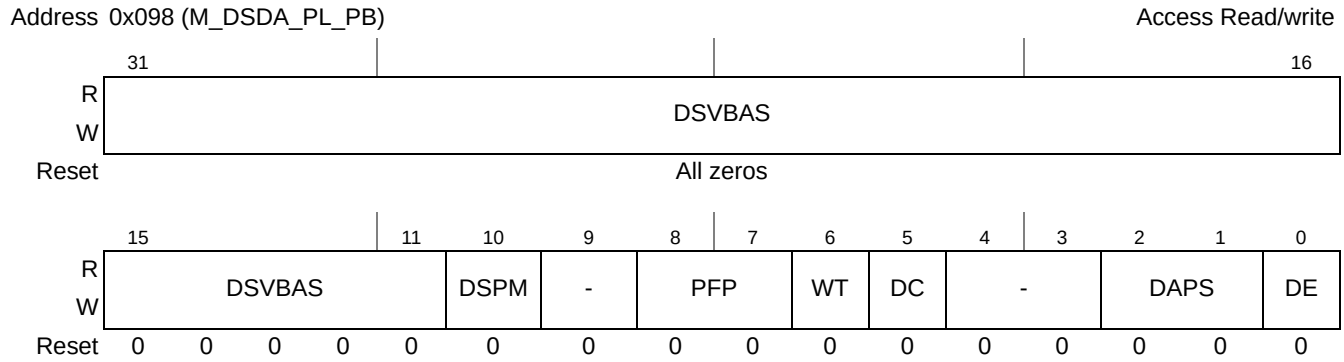

Figure 3-35. Slave Preload Register for Data Segment Descriptor Registers A (M_DSDA_PL_PB)

See Table 3-33 for the description of the M_DSDA_PL_PB fields.

3.3.24 Slave Preload Register for Data Segment Descriptor B (M_DSDB_PL_PB)

M_DSDB_PL_PB, shown in Figure 3-36, is a preload register that is used to configure data segment descriptor B registers.

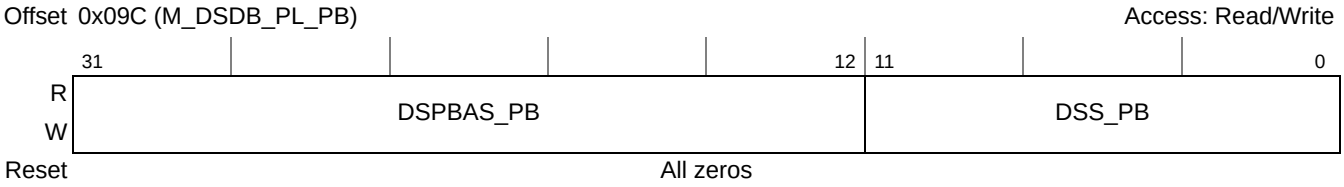

Figure 3-36. Slave Preload Register for Data Segment Descriptor Registers B (M_DSDB_PL_PB)

See Table 3-37 for the description of the M_DSDB_PL_PB fields.

3.3.25 Slave Preload Register for Data Segment Descriptor C (M_DSDC_PL_PB)

M_DSDC_PL_PB, shown in Figure 3-37, is a preload register that is used to configure data segment descriptor C registers.

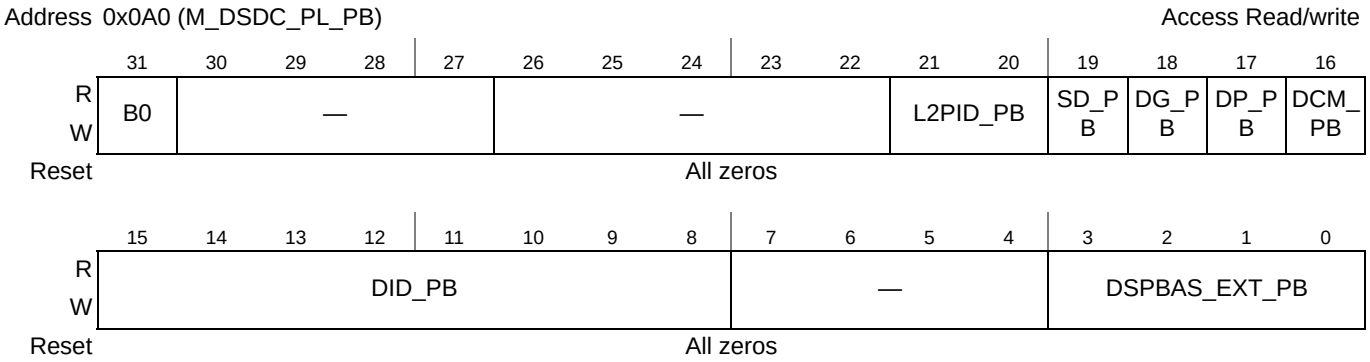

Figure 3-37. Slave Preload Register for Data Segment Descriptor Registers C (M_DSDC_PL_PB)

See Table 3-40 for the description of the M_DSDC_PL_PB fields.

3.3.26 Slave Preload Register for Data Segment Descriptor M (M_DSDM_PL_PB)

M_PSDM_PL_PB, shown in Figure 3-38, is a preload register that is used to configure data segment descriptor M registers.

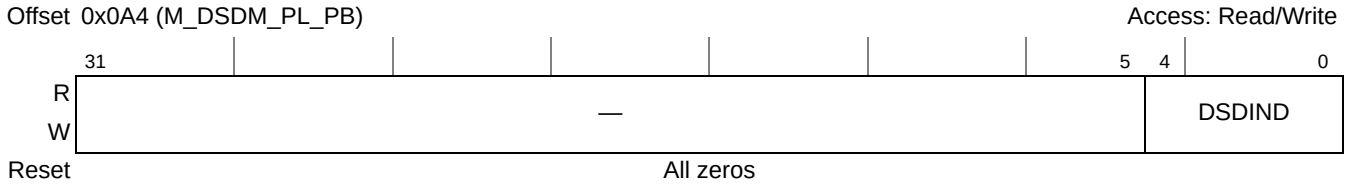

Figure 3-38. Slave Preload Register for Data Segment Descriptor M (M_DSDM_PL_PB)

Table 3-29 describes the M_PSDM_PL_PB fields.

Table 3-29. M_DSDM_PL_PB register bit description

Field	Description
31–5	Reserved
4–0 (DSDIND)	00000 Config Data Descriptor 0 00001 Config Data Descriptor 1 00010 Config Data Descriptor 2 ... 11111 Config Data Descriptor 31

3.3.27 Data MATT Programming Error Register (M_DMPER)

Figure 3-39 shows the Data MATT programming error register.

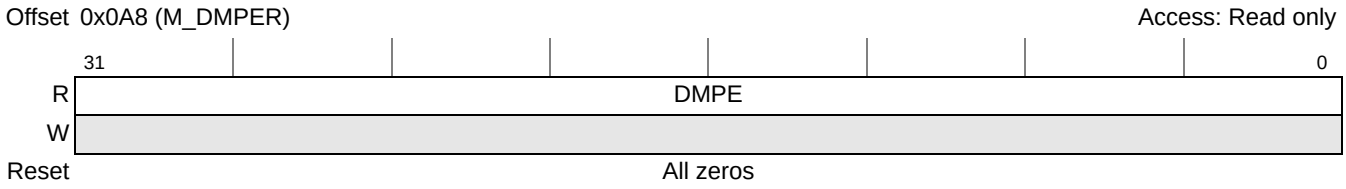

Figure 3-39. Data MATT Programming Error Register

Table 3-30 describes the M_DMPER fields.

Table 3-30. M_DMPER bit descriptions

Field	Description
31–0 DMPEn	Data MATT Programming Error (status bits) If set, indicates the following data MATT programming error in the appropriate descriptor: one of the M_DSDA_<i>.DSPM bit is set and there is an error in the programming of one of the related M_DSDA_<i>.DSVBA, M_DSDB_<i>.DSPBA, and M_DSDB_<i>.DSS registers. 0 No error 1 Error in descriptor *n*

3.3.28 SGB Watermark Value (M_WMCFG)

The watermark is a static configure. It can be configured only once. It cannot be dynamically reconfigured during a normal operation. Figure 3-40 shows the SGB watermark register. Watermark valid values can be configured from 9 to 16.

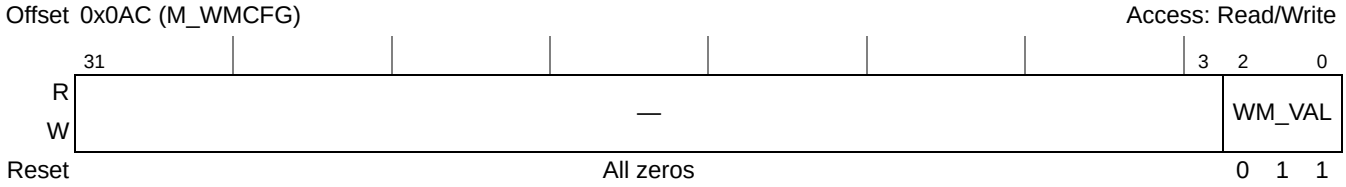

Figure 3-40. SGB Watermark Value (M_WMCFG)

Table 3-31 describes the M_WMCFG fields.

Table 3-31. SGB Watermark Value (M_WMCFG) bit description

Field	Description
31–3	Reserved
2–0 (WM_VAL)	000 Watermark value = 9 001 Watermark value = 10 010 Watermark value = 11 011 Watermark value = 12 100 Watermark value = 13 101 Watermark value = 14 110 Watermark value = 15 111 Watermark value = 16

3.3.29 Factory Debug Register (M_FDR)

Figure 3-41 shows the factory debug register.

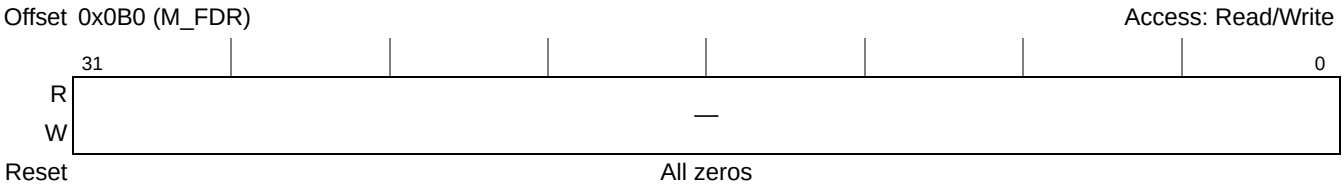

Figure 3-41. Factory Debug Register (M_FDR)

Table 3-32 describes the M_FDR fields.

Table 3-32. Factory Debug Register (M_FDR) bit description

Field	Description
31–0	Reserved

3.3.30 Data Segment Descriptor Registers A (M_DSDA_<i>)

M_DSDA_<i>, shown in Figure 3-42, is a group of registers in which the *i* in the register name is a number between 0 and 31. Changing any translation or other MMU attribute for a specific PID/DID requires the relevant virtual region from the caches to be flushed because the cache is virtually tagged and stores all relevant attributes of the lines.

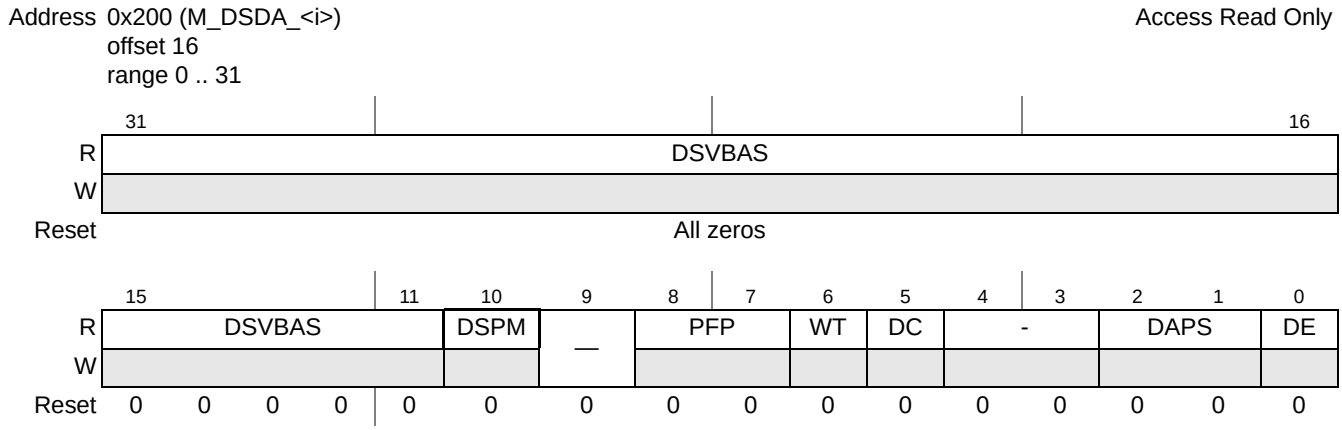

Figure 3-42. Data Segment Descriptor Registers A(M_DSDA_<i>)

Table 3-33 shows the field descriptions.

Table 3-33. M_DSDA_<i> (0 .. 31) bit descriptions

Field	Description
31–11 DSVBAS	When M_DSDA_<i>.DSPM bit is clear. Data Segment Virtual Base Address and Size. Indicates the virtual base address and size of the program segment. To determine the base address component, find the trailing one (from LSB to MSB) and replace it with a zero (the rest of the LSBs are zeros).The base address must be aligned to a power of 2. Size is $2^{\wedge n} \times 4$ Kbytes, where *n* is the number of zeros from LSB to MSB (not included) until the firsef1. The region base should always be aligned to the region size. See Table 3-34. When DSPM bit is set. Data Segment Virtual Base Address. Indicates the virtual base address of the data segment—bits [31:12] (bit 11 is Reserved).The segment size is specified in the corresponding M_DSDB_<i>.DSS. The data segment virtual base address must follow the alignment and boundary restrictions, according to Table 3-35. Boundary Restriction. Segment (M_DSDA_<i>.DSVBA + M_DSDB_<i>.DSS) must not cross multiples of the table specification according to the required size and alignment. For details, refer to Section 3.2.2.3, “Virtual segment base and size.”
10 DSPM	Data Segment Programming Model. This bit determine the segment base and size programming model. For more details, refer to Section 3.2.2.3, “Virtual segment base and size.” 0: Aligned segment model 1: Flexible segment model
8–7 PFP	Prefetch Policy. Program policy of prefetching current and next cache lines. Prefetch brings the current line with wrap around and also next line as defined by PFP field. 00: No prefetch 01: Prefetch on miss access 10: Prefetch on any access (hit or miss) 11: Reserved

Table 3-33. M_DSDA_<i> (0 .. 31) bit descriptions (continued)

Field	Description
6 DWT	Data Write-Through Segment. It controls the L2 write policy. If this bit is set, the segment gets write-through policy attribute. 0: Not write-through 1: Write-through
5 DC	Data Cacheability. If this bit is set, the segment is cacheable in the DCache and L2 Cache. 0: Noncacheable region 1: Cacheable region
2–1 DAPS	Data Access Permission Level. A two-bit field specifying if accesses in this segment are given data read and/or write permission (respectively). If there is no read or write permission and protection is enabled (not applicable for the QUERY command) a permission violation occurs. See Table 3-36.
0 DE	Descriptor Enable. If this bit is set, this program segment descriptor is enabled. This bit is aliased in the M_SDCR register. 0: Disable 1: Enable

Table 3-34. DSVBAS region base and size

Value	Base address	Size
0000 0000 0000 0000 0000 0	$0	4 Gbytes
0000 0000 0000 0000 0000 1	$0	4 Kbytes
0000 0000 0000 0000 0001 0	$0	8 Kbytes
0000 0000 0000 0000 0001 1	$1000	4 Kbytes
....	....	....
1111 1111 1111 1111 1111 0	$FFFFE000	8 Kbytes
1111 1111 1111 1111 1111 1	$FFFFF000	4 Kbytes

Table 3-35. DSVBAS alignment and boundary restrictions

Segment size	Boundary restriction	DSVBAS alignment
4K → 512K-4K	1M	4K
16K → 2M-16K	4M	16K
64K → 8MB-64K	16M	64K
256K → 16M–256K	64M	256K

Table 3-36. DAPS values and permission levels

Value	Permission Level
00	– –
01	– w
10	r –
11	r w

r = data reads permitted
w = data writes permitted

3.3.31 Data Segment Descriptor Registers B (M_DSDB_<i>)

M_DSDB_<i>, shown in Figure 3-43, is a group of registers for which the *i* in the register name is a number between 0 and 31.

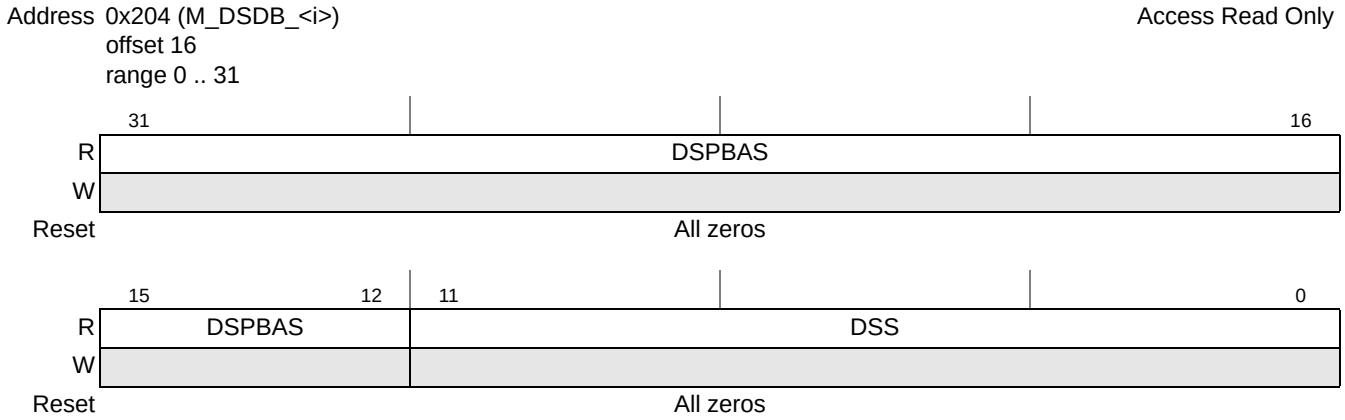

Figure 3-43. Data Segment Descriptor Registers C (M_DSDB_<i>)

Table 3-37 describes the M_DSDB_<i> fields.

Table 3-37. M_DSDB_<i> (0 .. 31) bit descriptions

Field	Description
31–12 DSPBA	When M_DSDA_<i>.DSPM bit is clear–Data Segment Physical Base Address. A 20-bit field together with 4-bit M_DSDC_<i>.DSPBA_EXT sets the MSB of the physical address used for translation. The size of the MSB is determined by the size defined by the DSBVAS bit. When M_DSDA_<i>.DSPM bit is set–Data Segment Physical Base Address. A 20-bit field together with 4-bit M_DSDC_<i>.DSPBA_EXT sets the MSP of the physical address used for translation. See Table 3-38. Boundary Restriction. Segment (M_DSDA_<i>.DSVBA + M_DSDB_<i>.DSS) must not cross multiples of the table specification according to the required size and alignment. For details, see Section 3.2.2.3, "Virtual segment base and size."
11–0 DSS	Data Segment Size. This field is valid only if M_DSDA.DSPM bit is set. It defines the size of the program segment in multiples of 4 Kbytes: Segment size = DSS × 4 Kbytes. The size must follow the alignment and boundary restrictions, according to Table 3-39. Boundary restriction. segment (M_DSDA_<i>.DSVBA + M_DSDB_<i>.DSS and M_DSDB_<i>.DSPBA + M_DSDB_<i>.DSS) must not cross multiples of the table specification according to the size and required alignment. 000: 4 Kbytes 001: 4 Kbytes 002: 8 Kbytes ... FFF: (16 Mbytes–256 Kbytes)

This table shows DSPBA alignment and boundary restrictions.

Table 3-38. DSPBA alignment and boundary restrictions

Segment Size	Boundary Restriction	DSVBAS Alignment
4 Kbytes → 512 Kbytes-4 Kbytes	1 Mbytes	4 Kbytes
16 Kbytes → 2 Mbytes-16 Kbytes	4 Mbytes	16 Kbytes
64 Kbytes →8 Mbytes–64 Kbytes	16 Mbytes	64 Kbytes
256 Kbytes → 16 Mbytes–256 Kbytes	64 Mbytes	256 Kbytes

This table shows DSS alignment and boundary restrictions.

Table 3-39. DSS alignment and boundary restrictions

Segment Size	Boundary Restriction	DSS Alignment
4 Kbytes → 512 Kbytes-4 Kbytes	1 Mbytes	4 Kbytes

Table 3-39. DSS alignment and boundary restrictions (continued)

16 Kbytes → 2 Mbytes-16 Kbytes	4 Mbytes	16 Kbytes
64 Kbytes → 8 Mbytes-64 Kbytes	16 Mbytes	64 Kbytes
256 Kbytes → 16 Mbytes–256 Kbytes	64 Mbytes	256 Kbytes

3.3.32 Data Segment Descriptor Registers C (M_DSDC_<i>)

M_DSDC_<i>, shown in Figure 3-44, is a group of registers for which the *i* in the register name is a number between 0 and 31.

Address 0x208 (M_DSDC_<i>) offset 16 range 0 .. 31 — Access Read Only

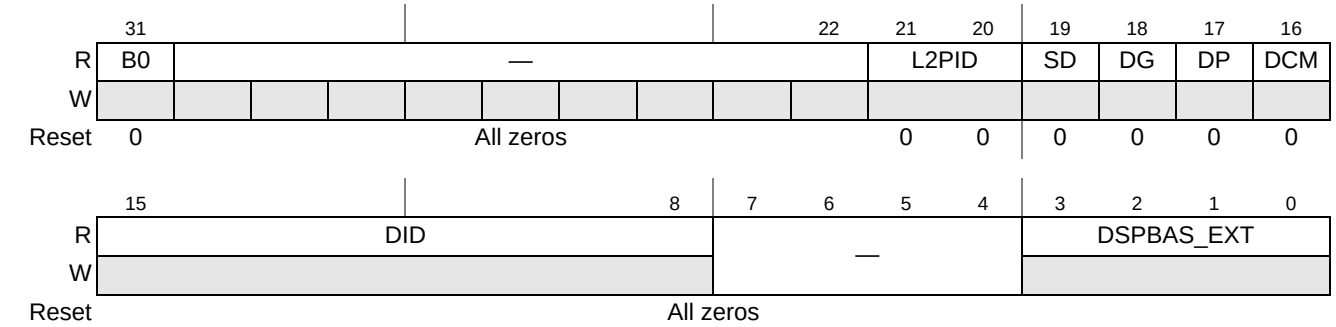

Figure 3-44. Data Segment Descriptor Registers C (M_DSDC_<i>)

Table 3-40 describes the M_DSDC_<i> fields.

Table 3-40. M_DSDC_<i> (0 .. 31) bit descriptions

Field	Description
31 B0	B0 0 Bank0 access is not permitted 1 Bank0 access permitted
21–20 L2PID	This field defines bits [1:0] of the L2 Partitioning ID. (Bits [4:3] of the L2 Partitioning ID are the core number inside the cluster. Bit [2] of the L2 Partition ID is always zero). For more details about L2 Partitioning ID, see L2 Cache chapter.
19 SD	Stack Descriptor If this bit is set, the segment gets stack attribute. That means, core accesses related to the stack pointer will be permitted to this segment. 0 Stack related accesses is not permitted 1 Stack related accesses is permitted

Table 3-40. M_DSDC_<i> (0 .. 31) bit descriptions (continued)

Field	Description
18 DG	Data Guarded Segment If this bit is set, the segment gets guarded attribute. 0 Not guarded 1 Guarded
17 DP	Data Peripheral Space If this bit is set, the segment gets peripheral space attribute. 0 Memory 1 Peripheral
16 DCM	Data Coherent Memory Segment If this bit is set, the segment gets memory coherency attribute. 0 No coherency 1 Memory coherency
15–8 DID	Data task ID Indicates the descriptor data ID. DID=0 used to define shared memory and matches any DID generated by the core. 0 Shared 1 255 DID
3–0 DSPBAS_EXT	This is 4 MSBs of 36-bit physical address used for translation.

3.3.33 Program Violation Address Register (M_PVA)

M_PVA, shown in Figure 3-45, is cleared when M_CR[CMIR] is set.

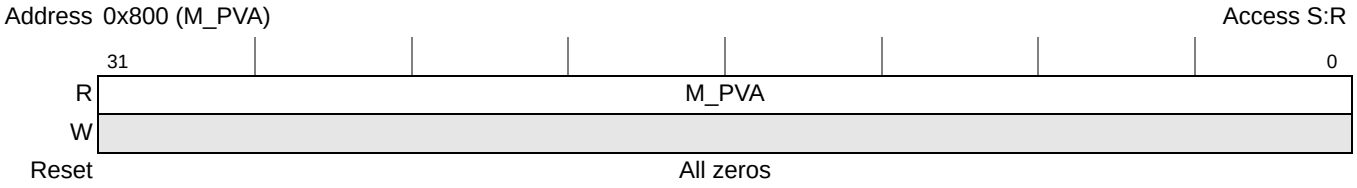

Figure 3-45. Program Violation Address Register (M_PVA)

Table 3-41 describes the M_PVA fields.

Table 3-41. M_PVA bit descriptions

Field	Description
31–0 M_PVA	Program Violation Address If one of the program error status bits in the M_PSR is set, this register contains the violating program address (word aligned). M_PVA is updated for the first non-permitted memory address. It is not updated unless the CMIR bit is clear. **Note:** Non-permitted memory addresses that arrive after the first address violation are not saved. **Note:** The M_PVA does not contain the address alignment information; this means M_PVA[0] is always zero.

3.3.34 Program Status Register (M_PSR)

M_PSR status bits do not change as long as the M_CR[CMIR] is set. Setting CMIR reset M_PSR. M_PSR reflects only one error at a time. For two or more errors, only the most severe error is represented (bits 0–5 are mutually exclusive). For details, refer to Section 3.2.4.2.2, “Core data and program bus errors and exceptions.”

Figure 3-46 shows the program status register.

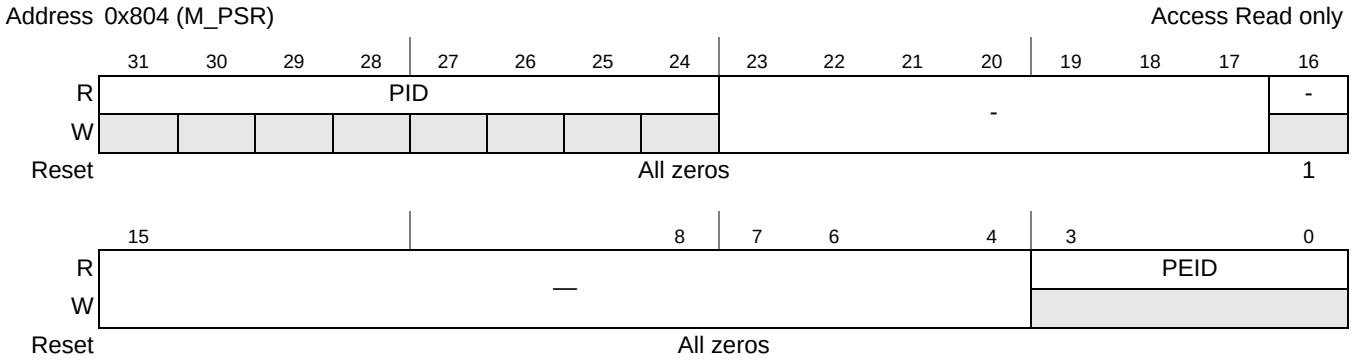

Figure 3-46. Program Status Register (M_PSR)

Table 3-42 describes the M_PSR fields.

Table 3-42. M_PSR bit descriptions

Field	Description
31-24 PID	Program ID: Holds the access Program ID.
3–0 PEID	Program Error ID: 0000: Program Multiple Segment Descriptor Hit • Set when the program address matches more than one program segment descriptor in non-coupled descriptors. 0001: Program Segment Miss • Set when a program access does not match any of the enabled program segment descriptors and protection unit is enabled (M_CR[MPE] is set). 0010: Program Permission Violation • Set when a program access matches a program segment but does not have sufficient permissions as defined for that segment and the protection unit is enabled (M_CR[MPE] is set). 0100: Program Non-Aligned Access Error • Set when a program non-aligned access occurs. 1001 NC hit: • Set when data access that has noncacheable attribute hits the cache 1110: Program Fetch Error • Set when the external fetch access gets an error on the RLink/CLink interface. 1111: Program ECC • Set when there is a program fetch access with an ECC error and ECCEE field in M_CR is enabled.

3.3.35 Program Exception Service Routine Address0 (M_PESRA0)

M_PESRA0, shown in Figure 3-47, contains the address of program service routine 0.

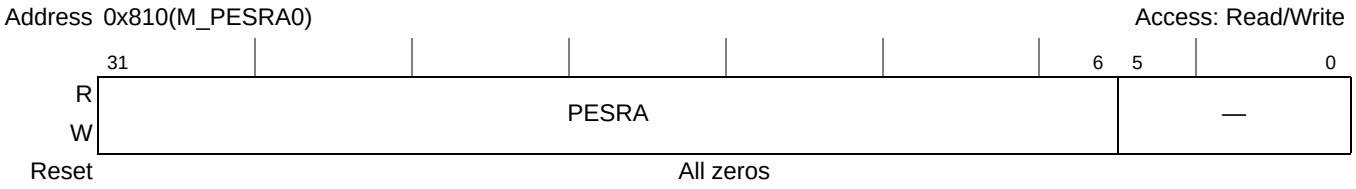

Figure 3-47. Program Exception Service Routine Address0 (M_PESRA0)

Table 3-43 describes the M_PESRA0 fields.

Table 3-43. M_PESRA0 bit descriptions

Field	Description
31–6 PESRA	Program Exception Service Routine Address 0
5–0	Reserved

3.3.36 Program Exception Service Routine Address1 (M_PESRA1)

M_PESRA1, shown in Figure 3-48, contains the address of program service routine 1.

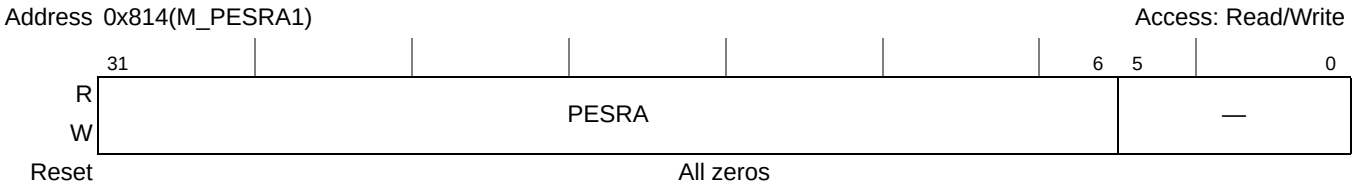

Figure 3-48. Program Exception Service Routine Address1 (M_PESRA1)

Table 3-44 describes the M_PESRA1 fields.

Table 3-44. M_PESRA1 bit descriptions

Field	Description
31–6 PESRA	Program Exception Service Routine Address 1
5–0	Reserved

3.3.37 Program Exception Service Routine Select (M_PESRS)

M_PESRS, shown in Figure 3-49, contains the mapping of program exceptions to service routine addresses.

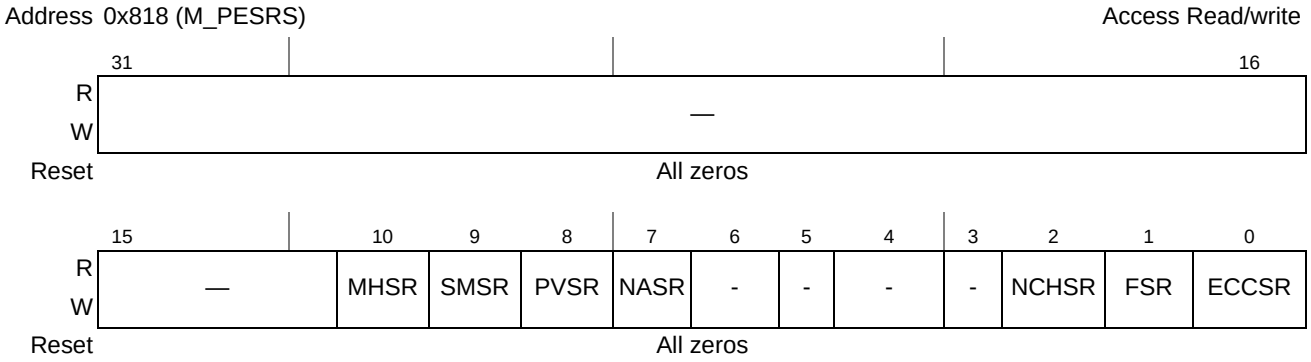

Figure 3-49. Program Exception Service Routine Select (M_PESRS)

Table 3-45 describes the M_PESRS fields.

Table 3-45. M_PESRS bit descriptions

Field	Description
31–11	Reserved
10 MHSR	Multiple segment descriptor hit service routine select Selects service routine address for multiple segment descriptor hit exception. 0 M_PESRA0 1 M_PESRA1
9 SMSR	Segment miss exception service routine select Selects service routine address for segment miss exception. 0 M_PESRA0 1 M_PESRA1
8 PVSR	Permission Exception service routine select Selects service routine address for permission exception. 0 M_PESRA0 1 M_PESRA1
7 NASR	Non-aligned access exception service routine select Selects service routine address for non-aligned access exception. 0 M_PESRA0 1 M_PESRA1
2 NCHSR	Noncacheable hit service routine select Selects service routine address for noncacheable hit exception. 0 M_PESRA0 1 M_PESRA1

Table 3-45. M_PESRS bit descriptions (continued)

Field	Description
1 FSR	Fetch exception service routine select Selects service routine address for fetch exception. 0 M_PESRA0 1 M_PESRA1
0 ECCSR	ECC error service routine select Selects service routine address for ECC exception. 0 M_PESRA0 1 M_PESRA1

3.3.38 Program Segment Descriptor VAL (M_PSDVAL)

M_PSDVAL, shown in Figure 3-50, contains 1 bit per program segment descriptor. Each bit, together with the correlate bit in M_PSDMASK, can enables/disables the respective segment descriptor. By clustering all bits together, the RTOS can activate all the relevant program segments using one instruction.

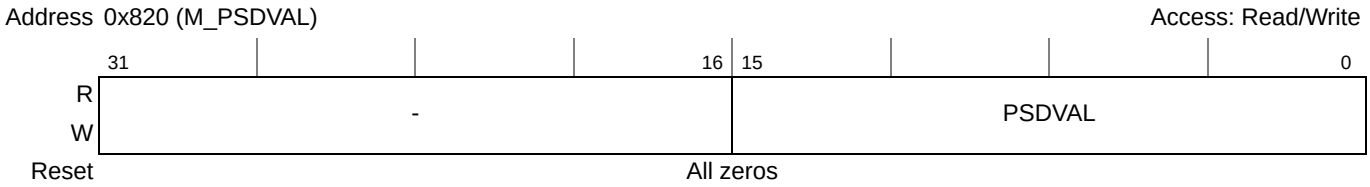

Figure 3-50. Program Segment Descriptor Value (M_PSDVAL)

Table 3-46 describes the M_PSDES fields.

Table 3-46. M_PSDVAL bit descriptions

Field	Description
15–0 PSDVAL*n*	Program segment descriptor enable Each of these bits is an alias of the respective DE bit in the respective M_PSDA*n* register. For example, bit 0 is related to M_PSDA 0. The state of the bit is observable and controllable from either path. If set, this program segment descriptor is enabled. 0 Clear the enable bit of the related data segment descriptor. 1 Enable bit of the related data segment descriptor is set.

3.3.39 Program Segment Descriptor MASK (M_PSDMASK)

M_PSDMASK, shown in Figure 3-51, contains 1 bit per program segment descriptor. Each bit define who has the ownership of the correlate program descriptor.

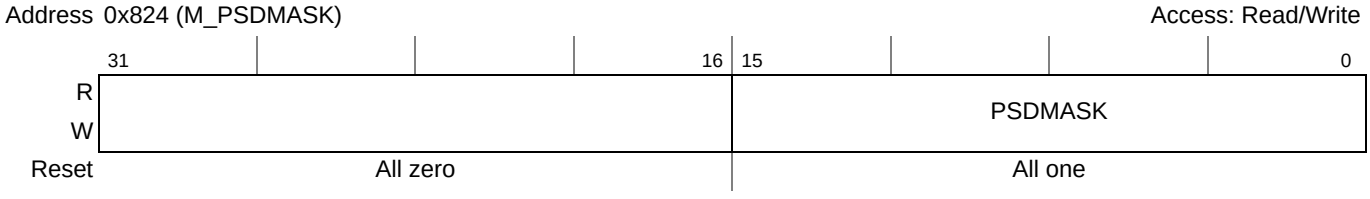

Figure 3-51. Program Segment Descriptor Mask (M_PSDMASK)

Table 3-47 describes the M_PSDMASK fields.

Table 3-47. M_PSDMASK bit descriptions

Field	Description
15–0 PSDMASK*n*	Program Segment Descriptor MASK 0 The ownership on this program descriptor is not for the core. 1 The Core has the ownership on this program descriptor.

3.3.40 Core Preload Register for Program Segment Descriptor A (M_PSDA_PL)

M_PSDA_PL, shown in Figure 3-52, is a preload register that is used to configure program segment descriptor A registers.

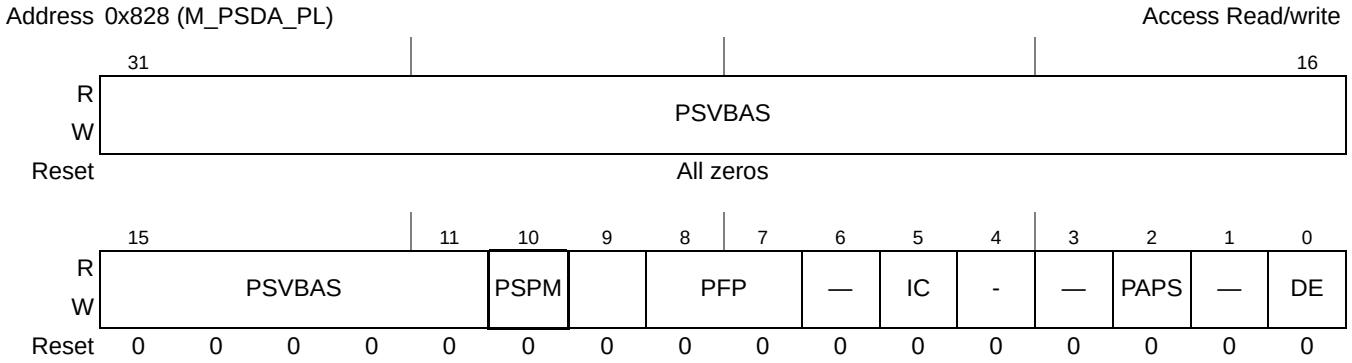

Figure 3-52. Core Preload Register for Program Segment Descriptor A (M_PSDA_PL)

See Table 3-51 for a description of the M_PSDA_PL fields.

3.3.41 Core Preload Register for Program Segment Descriptor B (M_PSDB_PL)

M_PSDB_PL, shown in Figure 3-53, is a preload register that is used to configure program segment descriptor B registers.

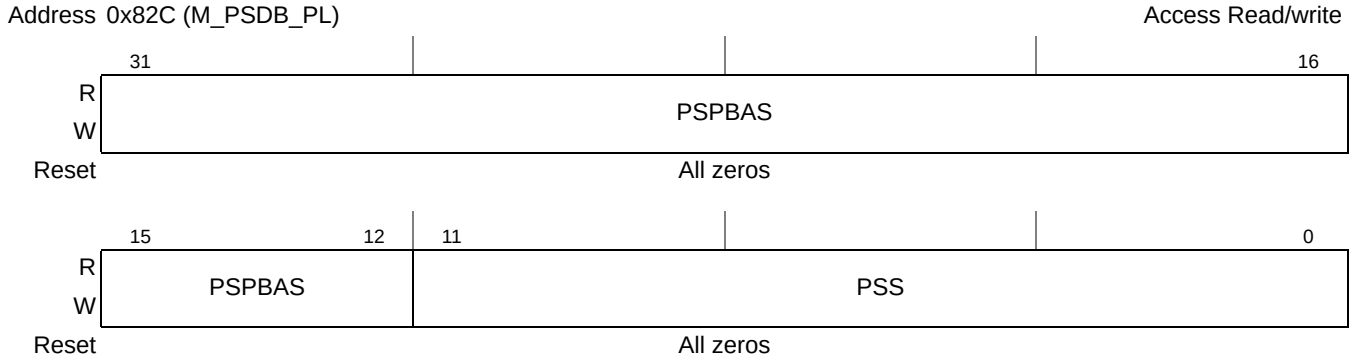

Figure 3-53. Core Preload Register for Program Segment Descriptor B (M_PSDB_PL)

See Table 3-54 for a description of the M_PSDB_PL fields.

3.3.42 Core Preload Register for Program Segment Descriptor C (M_PSDC_PL)

M_PSDC_PL, shown in Figure 3-54, is a preload register that is used to configure program segment descriptor C registers.

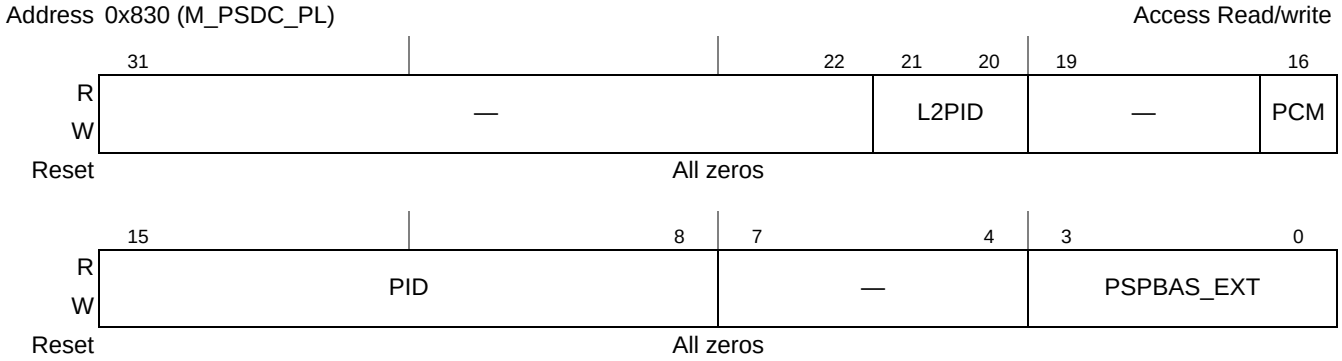

Figure 3-54. Core Preload Register for Program Segment Descriptor C (M_PSDC_PL)

See Table 3-55 for a description of the M_PSDC_PL fields.

3.3.43 Core Preload Register for Program Segment Descriptor M (M_PSDM_PL)

M_PSDM_PL, shown in Figure 3-55, is a preload register that is used to configure program segment descriptor M registers.

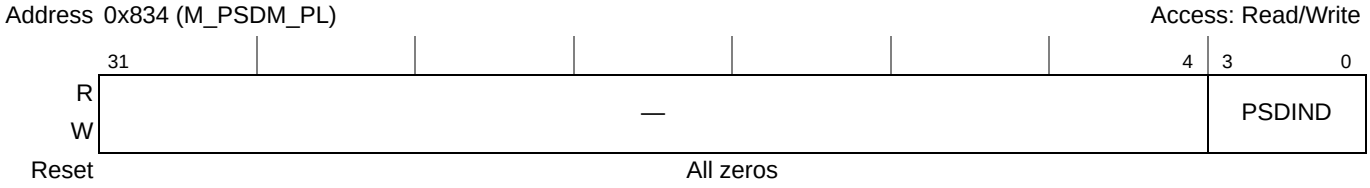

Figure 3-55. Core Preload Register for Program Segment Descriptor M (M_PSDM_PL)

See Table 3-48 for a description of the M_PSDM_PL fields.

Table 3-48. M_PSDM_PL register bit description

Field	Description
31–4	Reserved
3–0 (PSDIND)	0000 Config Program Descriptor 0 0001 Config Program Descriptor 1 0010 Config Program Descriptor 2 ... 1111 Config Program Descriptor 15

3.3.44 Slave Preload Register for Program Segment Descriptor A (M_PSDA_PL_PB)

M_PSDA_PL_PB, shown in Figure 3-56, is a preload register that is used to configure program segment descriptor A registers.

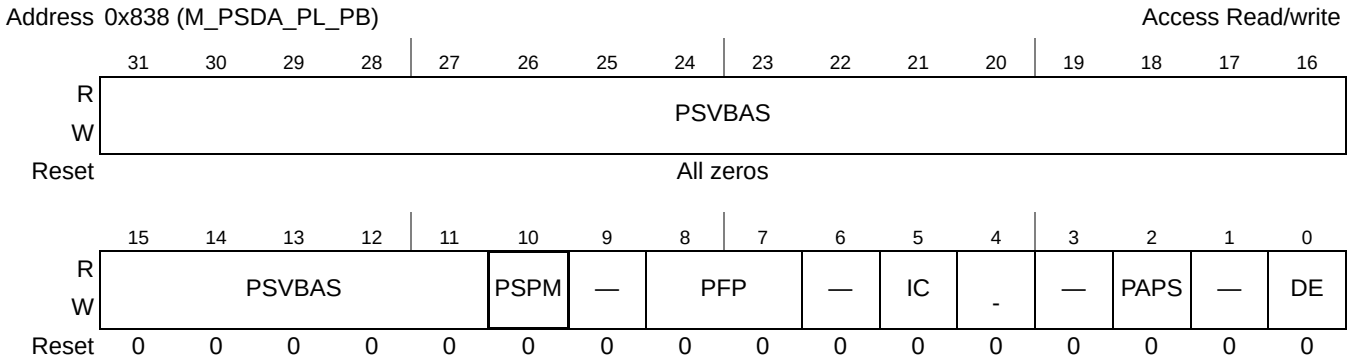

Figure 3-56. Slave Preload Register for Program Segment Descriptor A (M_PSDA_PL_PB)

See Table 3-51 for a description of the M_PSDA_PL_PB fields.

3.3.45 Slave Preload Register for Program Segment Descriptor B (M_PSDB_PL_PB)

M_PSDB_PL_PB, shown in Figure 3-57, is a preload register that is used to configure program segment descriptor B registers.

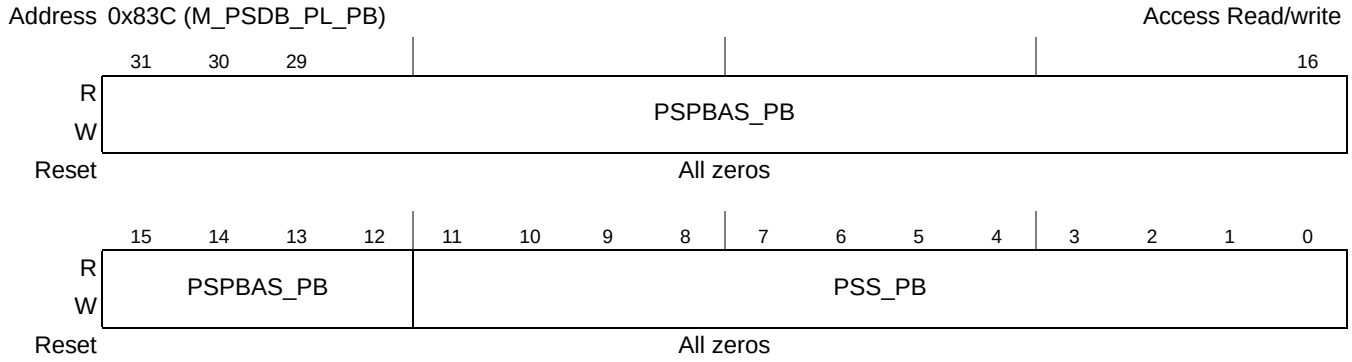

Figure 3-57. Slave Preload Register for Program Segment Descriptor B (M_PSDB_PL_PB)

See Table 3-54 for a description of the M_PSDB_PL_PB fields.

3.3.46 Slave Preload Register for Program Segment Descriptor C (M_PSDC_PL_PB)

M_PSDC_PL_PB, shown in Figure 3-58, is a preload register that is used to configure program segment descriptor C registers.

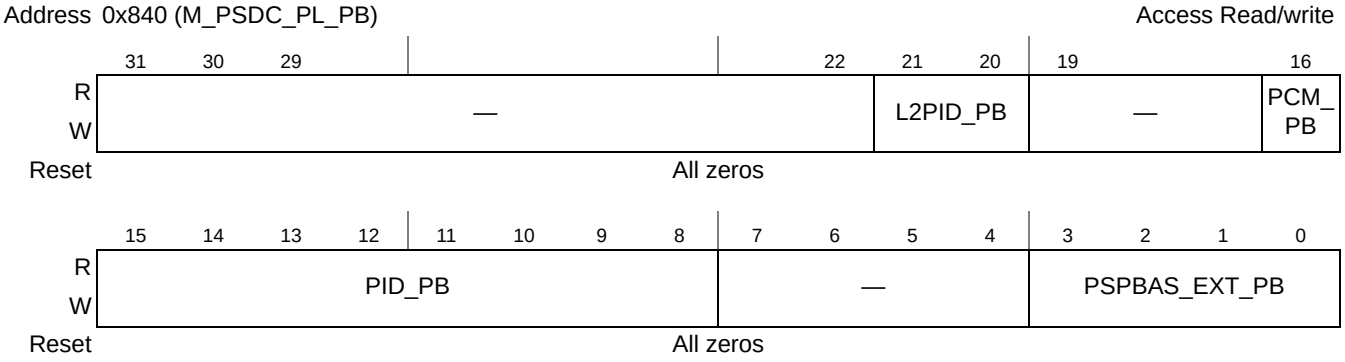

Figure 3-58. Slave Preload Register for Program Segment Descriptor C (M_PSDC_PL_PB)

See Table 3-55 for a description of the M_PSDC_PL_PB fields.

3.3.47 Slave Preload Register for Program Segment Descriptor M (M_PSDM_PL_PB)

M_PSDM_PL_PB, shown in Figure 3-59, is a preload register that is used to configure program segment descriptor M registers.

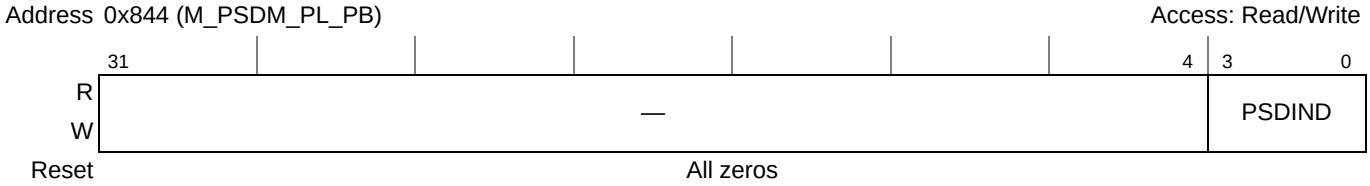

Figure 3-59. Slave Preload Register for Program Segment Descriptor M (M_PSDM_PL_PB)

Table 3-49 describes the M_PSDM_PL_PB fields.

Table 3-49. M_PSDM_PL_PB register bit description

Field	Description
31–4	Reserved
3–0 (PSDIND)	0000 Config Program Descriptor 0 0001 Config Program Descriptor 1 0010 Config Program Descriptor 2 ... 1111 Config Program Descriptor 15

3.3.48 Program MATT Programming Error Register (M_PMPER)

Figure 3-60 shows the program MATT programming error register.

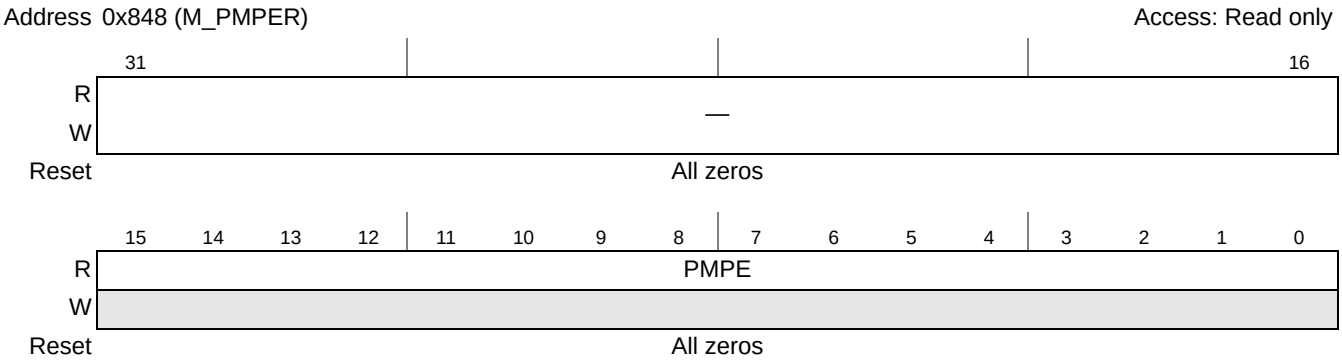

Figure 3-60. Program MATT Programming Error Register

Table 3-50 describes the M_PMPER fields.

Table 3-50. M_PMPER bit descriptions

Field	Description
31–16	Reserved
15–0 PMPEn	Program MATT programming error (status bits) if set, indicates the following program MATT programming error in the appropriate descriptor: one of the M_PSDA_<i>.PSPM bits is set and there is an error in the programming of one of the related M_PSDA_<i>.PSVBA, M_PSDB_<i>.PSPBA, and M_PSDB_<i>.PSS registers. 0 No error 1 Error in descriptor *n*

3.3.49 Program Segment Descriptor Registers A (M_PSDA_<i>)

M_PSDA_<i>, shown in Figure 3-61, is a group of registers for which the *i* variable in the register name is a number between 0 and 15. Changing any translation or other MMU attribute for a specific PID requires

the relevant virtual region from the caches to be flushed because the cache is virtually tagged and stores all the relevant attributes of the lines.

Address 0xA00 (M_PSDA_<i>)
offset 16
range 0..15

Access Read only

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	PSVBAS															
W																
Reset	All zeros															

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PSVBAS					PSPM		PFP		—	IC	-	—	PAPS	—	DE
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3-61. Program Segment Descriptor Registers A (M_PSDA_<i>)

Table 3-51 describes the M_PSDA_<i> fields

Table 3-51. M_PSDA_<i> bit descriptions

Field	Description
31–11 PSVBAS	Program segment virtual base address and size When M_PSDA_<i>.PSPM bit is clear, this field Indicates the virtual base address and size of the program segment. To determine the base address component, find the trailing one (from LSB to MSB) and replace it with a zero (the rest of the LSBs are zeros).The base address must be aligned to a power of 2. Size is $2^{n} \times 4$ Kbytes, where *n* is the number of zeros from LSB to MSB (not included) until the first 1. The region base should always be aligned to the region size. See Table 3-52. When PSPM bit is set, this field indicates the virtual base address of the program segment—bits [31:12] (bit 11 is reserved). The segment size is specified in the corresponding M_PSDB_<i>.PSS. The program segment virtual base address must follow the alignment and boundary restrictions, according to Table 3-53. Boundary Restriction. Segment (M_PSDA_<i>.PSVBA + M_PSDB_<i>.PSS) must not cross multiples of the table specification according to the required size and alignment. For details, refer to Section 3.2.2.3, “Virtual segment base and size.”
10 PSPM	Program segment programming model This bit determine the segment base and size programming model. For more details, refer to Section 3.2.2.3, “Virtual segment base and size.” 0 Aligned segment model 1 Flexible segment model
8-7 PFP	Prefetch Policy. Program policy of prefetching current and next cache lines. Prefetch brings the current line with wrap around and also next line as defined by PFP field. 00 No prefetch 01 Prefetch on miss access 10 Prefetch on any access (hit or miss) 11 Reserved
5 IC	Instruction Cacheability. If this bit is set, the segment is cacheable in the ICache and L2 Cache. 0 Noncacheable region 1 Cacheable region

Table 3-51. M_PSDA_<i> bit descriptions (continued)

Field	Description
2 PAPS	Program Access Permission Level. If this bit is set, the segment is given fetch permission for program accesses. If the PAPS bit is set, program protection checks are disabled for this segment. 0 No access permitted 1 Execute (instruction fetch) permitted
0 DE	Descriptor Enable. If this bit is set, this program segment descriptor is enabled. This bit is aliased in the M_SDCR register. 0 Disable 1 Enable

This table shows the PSVBAS base address and size.

Table 3-52. PSVBAS base address and size

Value	Base Address	Size
0000 0000 0000 0000 0000 0	$0	4 Gbytes
0000 0000 0000 0000 0000 1	$0	4 Kbytes
0000 0000 0000 0000 0001 0	$0	8 Kbytes
0000 0000 0000 0000 0001 1	$1000	4 Kbytes
....	....	....
1111 1111 1111 1111 1111 0	$FFFFE000	8 Kbytes
1111 1111 1111 1111 1111 1	$FFFFF000	4 Kbytes

This table shows PSVBAS alignment and boundary restrictions.

Table 3-53. PSVBAS alignment and boundary restriction

Segment Size	Boundary Restriction	PSVBAS Alignment
4 Kbytes → 512 Kbytes–4 Kbytes	1 Mbyte	4 Kbytes
16 Kbytes → 2 Mbytes–16 Kbytes	4 Mbytes	16 Kbytes
64 Kbytes → 8 Mbytes–64 Kbytes	16 Mbytes	64 Kbytes
256 Kbytes → 16 Mbytes–256 Kbytes	64 Mbytes	256 Kbytes

3.3.50 Program Segment Descriptor Registers B (M_PSDB_<i>)

M_PSDB_<i>, shown in Figure 3-62, is a group of registers for which the *i* in the register name is a number between 0 and 15.

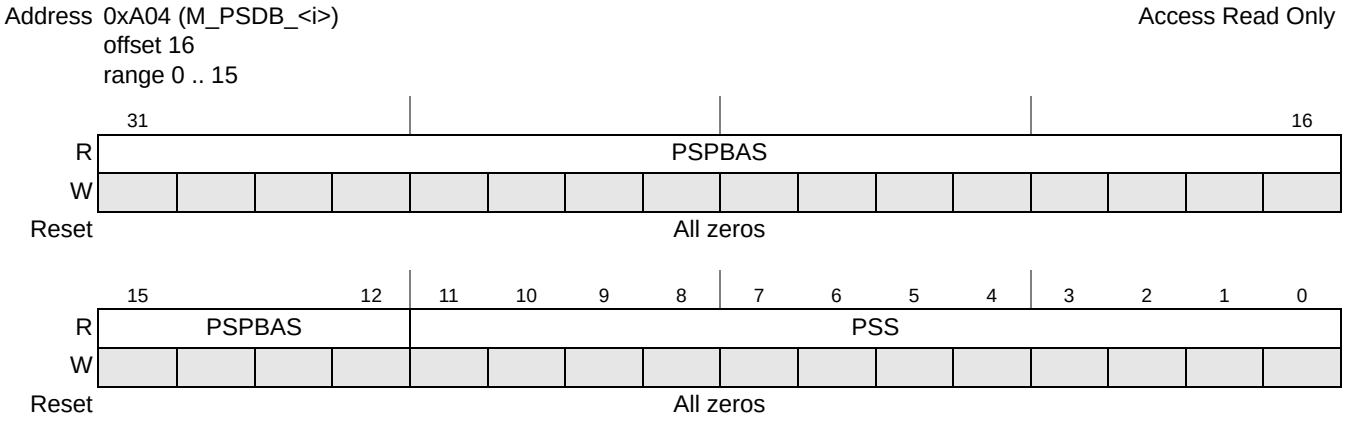

Figure 3-62. Program Segment Descriptor Registers B (M_PSDB_<i>)

Table 3-54 describes the M_PSDB_<i> fields.

Table 3-54. M_PSDB_<i> (0 .. 15) bit descriptions

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>31–12
PSPBA</td><td>Program Segment Physical Base Address
When M_PSDA_<i>.PSPM bit is clear, a 20-bit field together with 4-bit M_PSDC_<i>.PSPBA_EXT sets the MSP of the physical address used for translation. The size of the MSP is determined by the size defined by the PSBVAS bit.
When M_PSDA_<i>.PSPM bit is set, a 0-bit field together with 4-bit M_PSDC_<i>.PSPBA_EXT sets the MSP of the physical address used for translation.
<table>
<tr><th>Segment Size</th><th>Boundary Restriction</th><th>PSVBAS Alignment</th></tr>
<tr><td>4 Kbytes → 512 Kbytes–4 Kbytes</td><td>1 Mbyte</td><td>4 Kbytes</td></tr>
<tr><td>16 Kbytes → 2 Mbytes–16 Kbytes</td><td>4 Mbytes</td><td>16 Kbytes</td></tr>
<tr><td>64 Kbytes → 8 Mbytes–64 Kbytes</td><td>16 Mbytes</td><td>64 Kbytes</td></tr>
<tr><td>256 Kbytes → 16 Mbytes–256 Kbytes</td><td>64 Mbytes</td><td>256 Kbytes</td></tr>
</table>
Boundary Restriction. Segment (M_PSDA_<i>.PSVBA + M_PSDB_<i>.PSS) must not cross multiples of the table specification according to the required size and alignment.
For details, refer to Section 3.2.2.3, “Virtual segment base and size.”</td></tr>
<tr><td>11–0
PSS</td><td>Program Segment Size
This field is valid only if M_PSDA.PSPM bit is set. It defines the size of the program segment in multiples of 4 Kbytes:
Segment size = PSS × 4 Kbytes.
The size must follow the alignment and boundary restrictions, according to the next table:
<table>
<tr><th>Segment Size</th><th>Boundary Restriction</th><th>PSS Alignment</th></tr>
<tr><td>4 Kbytes → 512 Kbytes–4 Kbytes</td><td>1 Mbytes</td><td>4 Kbytes</td></tr>
<tr><td>16 Kbytes → 2 Mbytes–16 Kbytes</td><td>4 Mbytes</td><td>16 Kbytes</td></tr>
<tr><td>64 Kbytes → 8 Mbytes–64 Kbytes</td><td>16 Mbytes</td><td>64 Kbytes</td></tr>
<tr><td>256 Kbytes → 16 Mbytes–256 Kbytes</td><td>64 Mbytes</td><td>256 Kbytes</td></tr>
</table>
Boundary restriction. segment (M_PSDA_<i>.PSVBA + M_PSDB_<i>.PSS and M_PSDB_<i>.PSPBA + M_PSDB_<i>.PSS) must not cross multiples of the table specification according to the size and required alignment.
000: 4 Kbytes
001: 4 Kbytes
002: 8 Kbytes
...
FFF: (16 Mbytes–256 Kbytes)</td></tr>
</table>

3.3.51 Program Segment Descriptor Registers C (M_PSDC_<i>)

M_PSDC_<i>, shown in Figure 3-63, is a group of registers for which the *i* in the register name is a number between 0 and 15.

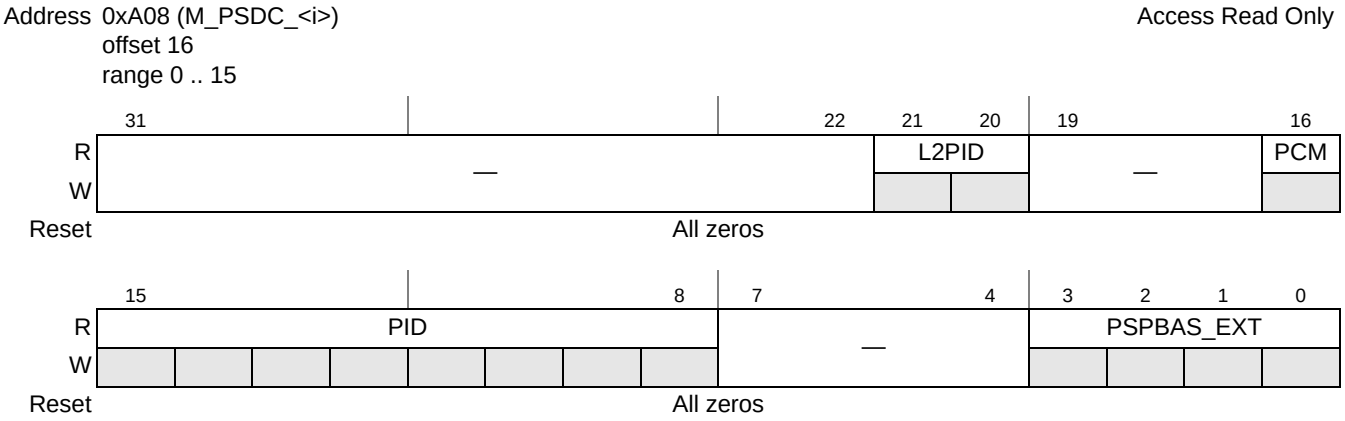

Figure 3-63. Program Segment Descriptor Registers C (M_PSDC_<i>)

Table 3-55 describes the M_PSDC_<i> fields.

Table 3-55. M_PSDC_<i> (0 .. 15) bit descriptions

Field	Description
21–20 L2PID	This field defines bits [1:0] of the L2 Partitioning ID. (Bits [4:3] of the L2 Partitioning ID are the core number inside the cluster. Bit [2] of the L2 Partition ID is always zero). For more details about L2 Partitioning ID, see L2 Cache chapter.
16 PCM	Program Coherent Memory Segment If this bit is set, the segment gets memory coherency attribute. 0 No coherency 1 Memory coherency
15–8 PID	Program task ID. Indicates the descriptor program ID. PID = 0 used to define shared memory and matches any PID generated by the core. 0 Shared 1 255 PID
3–0 PSPBAS_EXT	This is the 4 MSBs of the 36-bit physical address used for translation.

3.4 Default state of the MMU when exiting from reset

When the processor exits reset, only MMU descriptor 0 is enabled for base virtual address 0xFEC00000, size 256 KByte, shared between all tasks. Protection and address translation are disabled. Descriptor 0 is configured to cover the Bank0 address space of all cores in the cluster, see Section 9.3.3, “Peripheral bus.” This enables the boot sequence to configure the MMU of the current core and the MMU of the other cores of the same cluster. Program and data accesses to addresses different from Bank0 proceed without translation (memory protection is disabled). The 36-bit physical address is built from a 32-bit effective address extended with a 4-bit zero vector (MSB).

NOTE

The descriptor for bank 0 must always remain enabled. Disabling it prevents future access to the MMU and other subsystem registers from its core. In such a case, only the external master can access the subsystem registers.

Bank 0 is always not-protected. Any master can access in Read or Write to Bank 0.

The translation of Bank 0 can be reprogrammed without disabling the descriptor. Changing the virtual location of bank 0 should be done by setting another bank 0 descriptor at different virtual address and then disabling the first one, thus providing continuos bank 0 mapping.

Chapter 4
L1 Caches

The L1 subsystem consist of two cache-based channels: the data channel, which handles the data, and the instruction channel, which handles the instructions.

The data channel processes data accesses to the memory system. This includes accesses from the core and also from the cache management engine (CME), which is an advanced cache instructions generator. The data channel includes a read cache and a write cache called store gather buffer (SGB). It also manages noncacheable data accesses to the external memory or to the control memory space, FVP peripherals, or FVP subsystem memory-mapped registers. The main components of the data channel, illustrated in Figure 4-1, are the read cache and the write cache (SGB).

The read cache has the following characteristics:

- 32-Kbyte cache memory
- 8 ways
- 32 cache indices
- 128-byte cache line
- 256 cache lines (TAGs)
- Virtual allocation physical tag
- Next line hardware prefetch
- Seamless non-aligned read support

The write cache (SGB) has the following main characteristics:

- 1-Kbyte, including 16 entries of 64 bytes of data
- Efficiently gathers multiple individual stores
- Supports zero wait state read hit
- Has unaligned access support
- Has FIFO access ordering
- Supports two levels of priority
- Has a static watermark

Figure 4-1 shows the data channel block diagram.

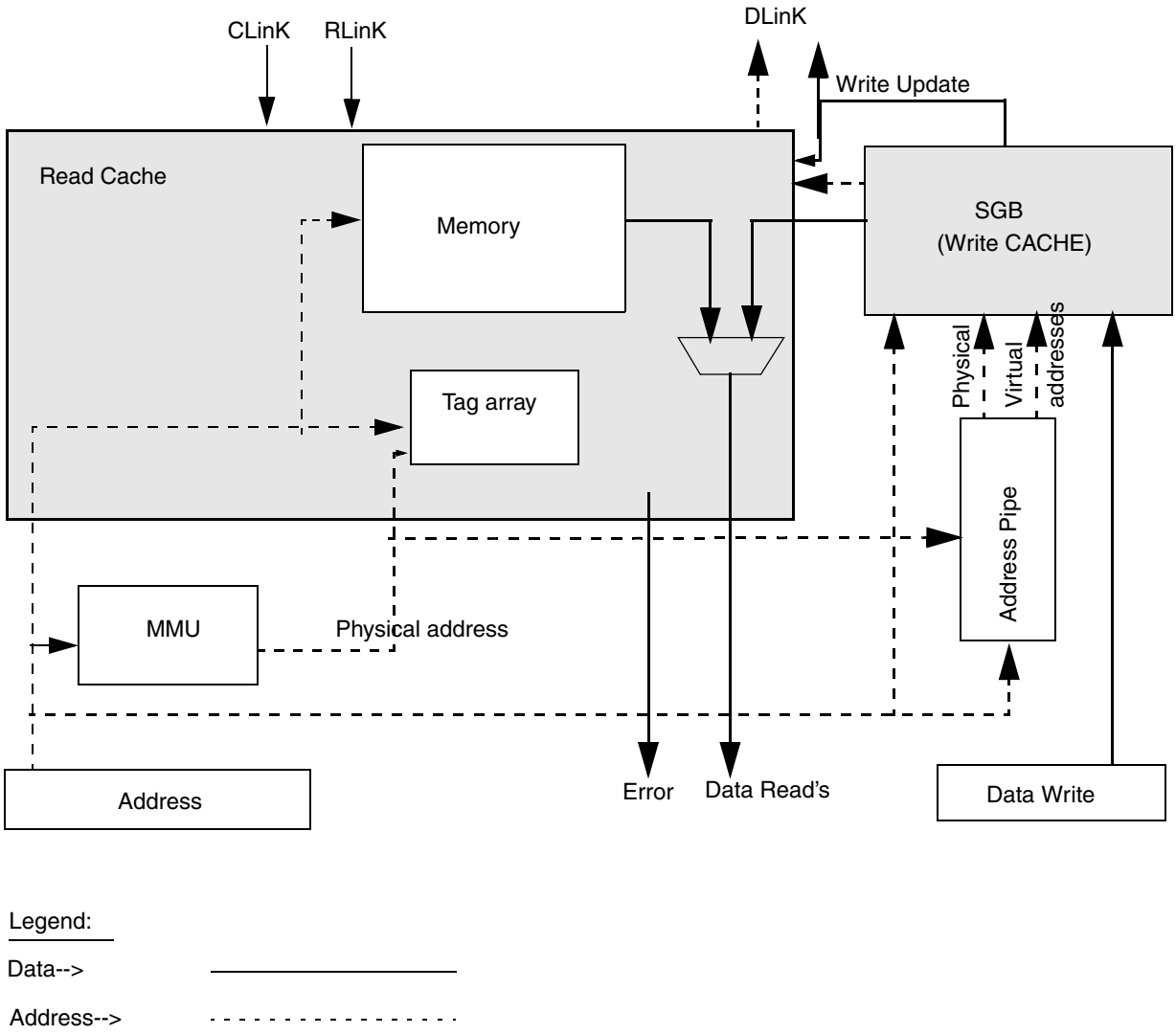

Figure 4-1. Data channel block diagram

The SGB is a data storage that contains 16 entries of 64 bytes each to hold the last written data. All core write accesses pass through the SGB on the way to memory. Cacheable writes are merged if possible to a single entry to save memory bandwidth. Accesses pass through the SGB in order (FIFO). Read after write hazards for cacheable accesses are resolved without penalty.

The read cache is physical with virtual allocation. The allocation is performed using a virtual address. In the case of a hit, a physical address is retrieved from the matched line and compared to the physical address from the memory management unit (MMU). If the addresses are different, there is an alias copy. The line is invalidated and a new physical line is loaded. In the case of a new line allocation, the physical address of the new allocation is checked against the existing lines in the cache. If the new physical address already

resides in the cache, the matched line is invalidated in order to prevent duplication. This technique provides seamless physical cache behavior to the user.

The read data is a combination of both caches. The SGB always has the most recent copy of the data even when the data exists in both caches. Therefore, the data from the SGB is sent to the core. Each time a write from the SGB is written to memory, it is also updated in the read cache, if such a line exists. This preserves data coherency between two caches.

The DCache, which operates at core speed, keeps the recently accessed read data. When an addressed data (from a cacheable memory area) is found in the array, it is immediately made available to the core (DCache hit) in a read. When the required address read is not found in either the array or the SGB, a DCache miss occurs. The data is fetched to the DCache from the external (off-subsystem) memory and then driven to the core. The SGB updates the DCache in case the write access is a hit in the cache at the time of outputting the write.

A non-stalling non-align access support is provided by the cache and SGB as long as the access does not cross 4 Kbyte boundary.

The data channel differentiates between cacheable and noncacheable addresses. The MMU provides cacheable/noncacheable indication for each access according to matched MATT configuration.

Table 4-1 summarizes the various types of accesses handled by the data channel.

Table 4-1. Data channel access summary

Access type		Allocation policy	Comments
Cacheable access	Read	Allocation in read cache	Triggers HW next line prefetching of data if not in cache and is enabled by MMU. Next line prefetch never cross 4 Kbyte boundary.
	Write	Allocation in SGB	Read cache ignores write accesses. SGB allocates a new entry or merges into existing entry in case of hit.
Noncacheable access	Read	Not allocated	—
	Write	Allocation in SGB	Read cache ignores write accesses. SGB keeps order between peripheral accesses and proceeds them to the higher level cache without merging.
Guarded (Noncacheable) access	Read	Not allocated	The cache wait for the access to be validated by the core and the MMU error before accessing the Dlink.
	Write	Allocation in SGB	Read cache ignores write accesses. SGB keeps order between peripheral accesses and proceeds them to the higher level cache without merging.

The instruction cache similar to the data cache with the main difference being instruction cache support for one read bus.

The read cache has the following characteristics:

- 32-Kbyte cache memory
- 8 ways
- 32cache indices
- 128-byte cache line
- 256 cache lines (TAGs)
- Virtual allocation physical tag
- Next line hardware prefetch

Figure 4-2 shows the instruction channel block diagram.

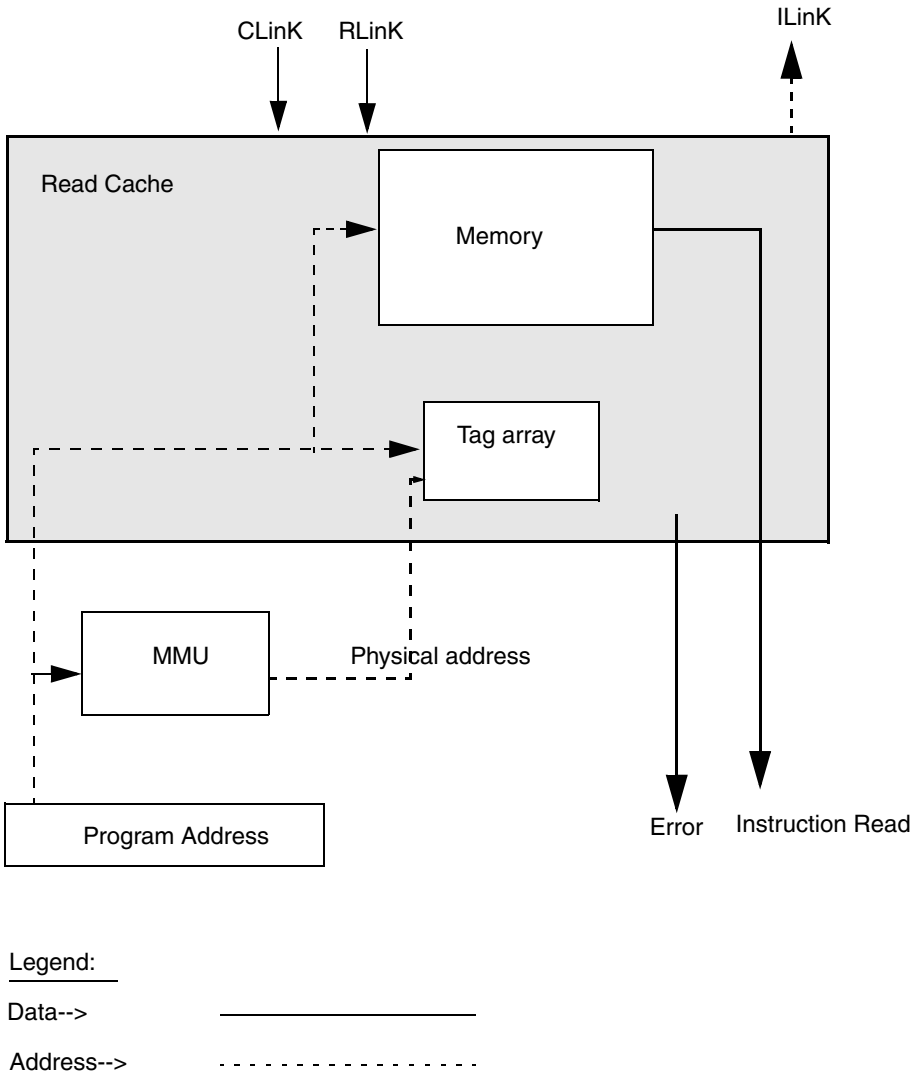

Figure 4-2. Instruction channel block diagram

4.1 Main features of the L1 subsystem

The following list provides the main features of the data channel:

- Handles two parallel core accesses (XA, XB)
- Handles an additional two parallel write buses (WA,WB)
- Supports 1-, 2-, 4-, 8-, 16-, 32-, or 64-byte accesses
- Supports cacheable, noncacheable, and peripheral accesses
- Supports task-extended virtually addressed cache. The task ID from the MMU is stored as part of the line tag which allows a task-specific cache image that is not overridden by other tasks and that uses the same virtual address. This feature supports the multitask mechanism.
- Serves both read and write cache hit access without wait states (except memory conflicts).
- Serves non-aligned access in most cases without wait states.
- Upon a cache miss, it issues 128 byte accesses to the higher level memory
- Automatic hardware next line prefetch on a miss or hit controlled by the MMU
- Supports SC3900 core cache maintenance instructions that are operable for both user and supervisor user levels:
  - — Prefetch instructions
  - — Coherency acceleration instructions
  - — Memory barrier instructions that notify the higher level cache for synchronization.
- Pseudo-LRU (PLRU) with streaming support as cache line replacement mechanism (LRM).
- Cache debug mode where the cache state (ETAG values, valid, PLRU state) can be read and written.
- ECC by software, inclusive in L2 (error detection) support
- Supports DQuery instruction providing cache status information

The following list provides the main features of the instruction channel:

- Handles a core program access
- Supports 32 bytes accesses
- Supports cacheable and noncacheable accesses
- Supports task-extended virtually addressed cache. The task ID from the MMU is stored as part of the line tag which allows a task-specific cache image that is not overridden by other tasks and that uses the same virtual address. This feature supports multitask mechanism.
- Serves read cache hit accesses without wait states (except memory conflicts).
- Upon a cache miss, issues 128 byte accesses to the higher level memory
- Automatic hardware next line prefetch on a miss or hit controlled by the MMU
- Supports SC3900 core cache maintenance instructions that are operable for both user and supervisor user levels:
  - — Prefetch instructions
  - — Coherency acceleration instructions
  - — Memory barrier instructions that notify the higher level cache for synchronization.

- Pseudo-LRU (PLRU) with streaming support as the cache line replacement mechanism (LRM).
- Cache debug mode where the cache state (ETAG values, valid, PLRU state) can be read and written.
- ECC by software, inclusive in L2 (error detection) support
- Supports PQuery instruction providing cache status information

4.1.1 Test state

During BIST mode the cache memory acts as a memory-mapped area and cache memory is accessed through the dedicated BIST port. Other ports are disabled/not used. The cache logic prevents cache memory accesses that are not BIST accesses.

4.2 Read cache

This section discusses the following elements of the read cache (both instruction and data):

- Section 4.2.1, “Functional description”
- Section 4.2.2, “Cache coherency”
- Section 4.2.3, “Multitask support”

4.2.1 Functional description

Figure 4-3 illustrates cache characteristics. Each cache way includes cache lines. A line is associated with a physical tag and a virtual partial tag and task ID. Each line includes 128 bytes of data.

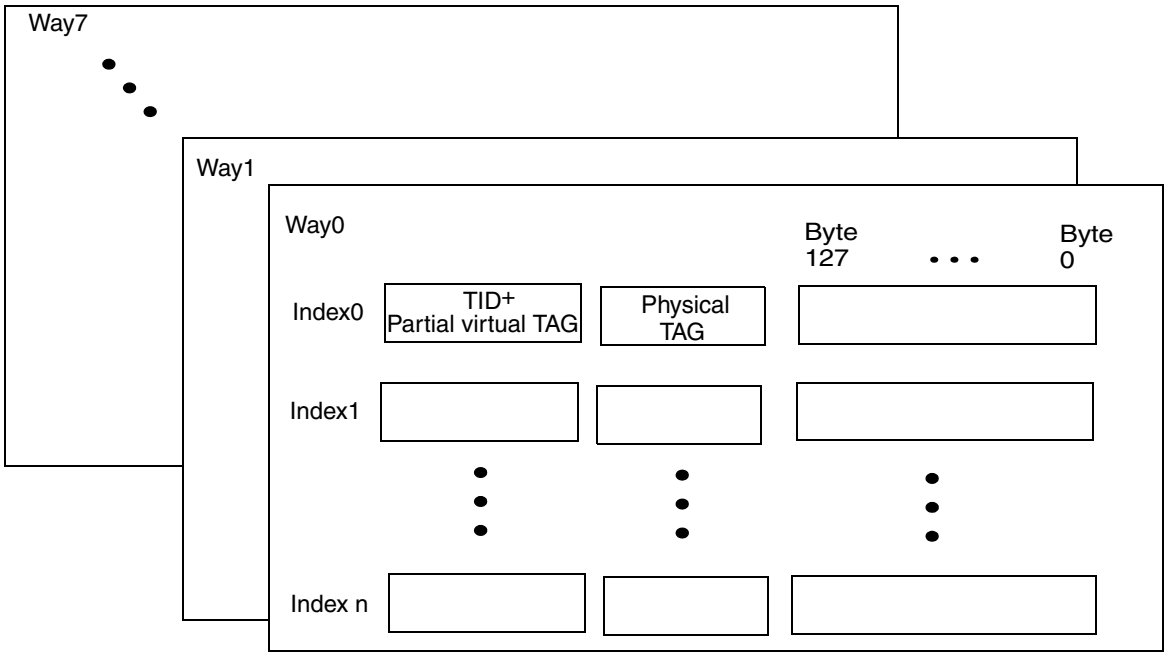

Figure 4-3. Cache characteristics

The partial virtual tag allows aliasing, meaning that not all upper bits of the address are compared during memory access. The physical address is fully compared to detect such cases. The cache line that is stored in the cache is invalidated if aliasing happens.

To locate the cache line that matches the access, the partial virtual tag and task ID are compared to those stored in the cache lines. All lines in the relevant set (index) and the following set (index) are compared simultaneously. In order to avoid impact on the critical path, the indices are split to odd and even sets. The index of the even set increments if the access is to the odd set (index). If the access is non-aligned, both sets are used to return the data. If the access is cache-line aligned, only one set is used (according to bit 0 of the index and bit 7 of the address). When there is a line match, the access is served by cache memory without latency penalty (except memory conflicts).

Not all the bits of the virtual address are used for the match calculation. The execution of a physical address check provides detection of virtual address aliasing. The cache line is invalidated in such cases. The assumption is that aliasing of 64 Mbyte is rare.

The use of the task ID in the cache is for performance reasons and not for logical because the physical tag is checked as well. The cache uses several bits of the task ID (not full task ID) to prevent frequent invalidation between tasks.

The usage of only part of the task ID and part of the address can result in aliasing that may degraded performance. The virtual memory map of the task is recommended to avoid aliasing of 64Mbyte.

4.2.1.1 Cache line replacement

When a line needs to be replaced in the cache, the pseudo-least-recently-used (PLRU) replacement algorithm is used. When a cache line is accessed, it is tagged as the most recently used line of the index. When a line miss occurs, the PLRU line is replaced by a new line, provided the cache is not completely locked or disabled. The PLRU bits in the cache are updated each time a cache line hit occurs based on the most recently used cache line.

4.2.1.1.1 Enhanced PLRU replacement

Line replacement is performed using a binary decision-tree, PLRU algorithm. There is an identifying bit for each cache way, L[0–7]. There are seven PLRU bits, B[0–6] for each index in the cache to determine the line to be replaced. The PLRU bits are updated when a new line is allocated or replaced and when there is a line hit. A line is selected for replacement according to the PLRU bit encoding shown in Table 4-2 (initial values of B[6–0] is all zeros).

Table 4-2. PLRU replacement way selection

PLRU bits						Way selected for replacement
B0	0	B1	0	B3	0	L0
	0		0		1	L1
	0		1	B4	0	L2
	0		1		1	L3
	1	B2	0	B5	0	L4
	1		0		1	L5
	1		1	B6	0	L6
	1		1		1	L7

Figure 4-4 shows the decision tree used to generate the victim line in the PLRU algorithm.

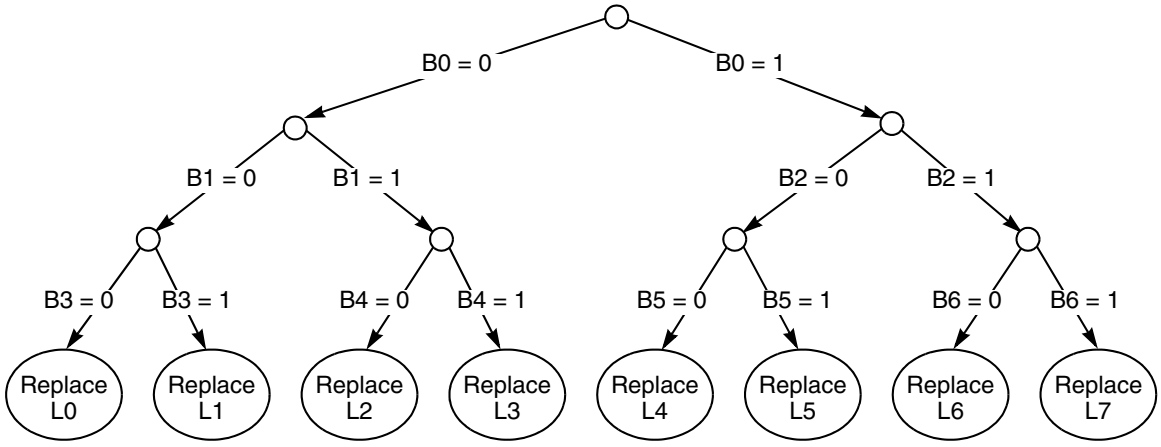

Figure 4-4. PLRU replacement algorithm

4.2.1.1.2 Enhanced PLRU update in case of hit or allocation

The PLRU is enhanced by a strong bit “S0” to augment the root of the tree.

Additional Strong bit to B0 to indicates that both the least and almost least are at the same side of the binary tree. PLRU B0–B6 and SO bits are updated according to the following rules:

Table 4-3. Enhanced PLRU B0-B6 update on hit or allocation

Line hit or allocated	PLRU bits					
L0	NEW B0	If(!S0&&(!B3&&!B1)) 1 else B0	NEW B1	If(!B3)1 else B1	NEW B3	1
L1		If(!S0&&(B3&&!B1)) 1 else B0		If(B3) 1 else B1		0
L2		If(!S0&&(!B4&& B1)) 1 else B0		If(!B4) 0 else B1	NEW B4	1
L3		If(!S0&&(!B4&& B1)) 1 else B0		If(B4) 0 else B1		0
L4		If(!S0&&(!B5&&!B2)) 0 else B0	NEW B2	If(!B5) 1 else B2	NEW B5	1
L5		If(!S0&&(B5&&!B2)) 0 else B0		If(B5) 1 else B2		0
L6		If(!S0&&(!B6&& B2)) 0 else B0		If(!B6) 0 else B2	NEW B6	1
L7		If(!S0&&(!B6&& B2)) 0 else B0		If(B6) 0 else B2		0

On next line pre-fetch allocation bit's B6–B3 keep their original value. Meaning next line pre-fetch is set 4th to be replaced as it is a speculative pre-fetch.

Upon a "streaming" access hit (for streaming see *SC3900 FVP Core Reference Manual)* the line is set to be second to be replaced. If two hits are streaming one will get to be the next for replacement and the other is second. This ensures that a large buffer that is read and marked as "streaming" will not thrash the whole cache but rather a small subset of it.

4.2.2 Cache coherency

The DCache block provides memory coherency support in the following ways:

- Hardware (HW) coherency support—supports back invalidation bus between L1 and L2 caches.
- Software (SW) coherency support—supports cache coherency instructions that allow the software to maintain coherency by itself.

The ICache supports coherency accelerator instruction.

The L1 cache supports the eLink protocol toward the L2 cache that allows the implementation of hardware coherency while optimizing the bus bandwidth. The L2 cache is inclusive: all addresses that exist in the L1 cache also exist in the L2 cache. For each line allocated in L2 cache, L2 cache maintains the status of this address in the L1 caches of the cluster. This allows maintaining coherency inside the cluster without implementing the complex MESI+L protocol in L1 caches. The coherency is maintained by using a back-invalidate request to one or more L1 caches as a result of a change in the CG coherency status. This change may happen due to internal cluster access by one of the cores or due to external to cluster access. The back-invalidate request cause the cache to invalidate the line.

Users have an option to maintain coherency manually by using the special cache instruction DFLUSH, or flush memory block. This instruction is efficient at maintaining coherency for amounts of data of several

hundred Kbytes or fewer. When DFLUSH is used, the cache writes back and invalidates a block of 64 bytes belonging to the specified address of the off platform memory.

DFLUSH may be used to implement coherency accelerator for data that is not marked with the memory coherency attribute by MMU or to accelerate hardware coherency by flushing the data beforehand (voluntary operation).

For more details on cache coherency, see Section 2.4, “Cache and data coherency.”

Both the hardware and coherency accelerator models require memory barrier support. Memory barriers ensure access ordering in certain cases. The types of barriers and cache behavior for each of them are described in Section 2.4.3, “Memory access synchronization.”

4.2.3 Multitask support

The cache address mechanism relates to virtual addresses. The extended virtual address is a combination of the address supplied by the SC3900 core and a task ID supplied by the memory management unit (MMU). Each cache line includes a tag and a task ID. The task ID identifies the task that initiates the core access. This mechanism enables multitask support.

An extended tag (ETAG) match occurs when access to an part of the address and part of the task ID match those of a valid stored cache line. Only part of the task ID and part of the address are used for the tag match which can result in aliasing. For a shared marked line, only the address is compared. The shared bit is set in cache lines allocated to shared memory space accesses. The MMU indicates a task shared memory space access by asserting a shared signal. This indication set the shared bit when there is no extended tag match and a new line is allocated. Figure 4-5 illustrates the two cases of address comparison for an extended tag match identification.

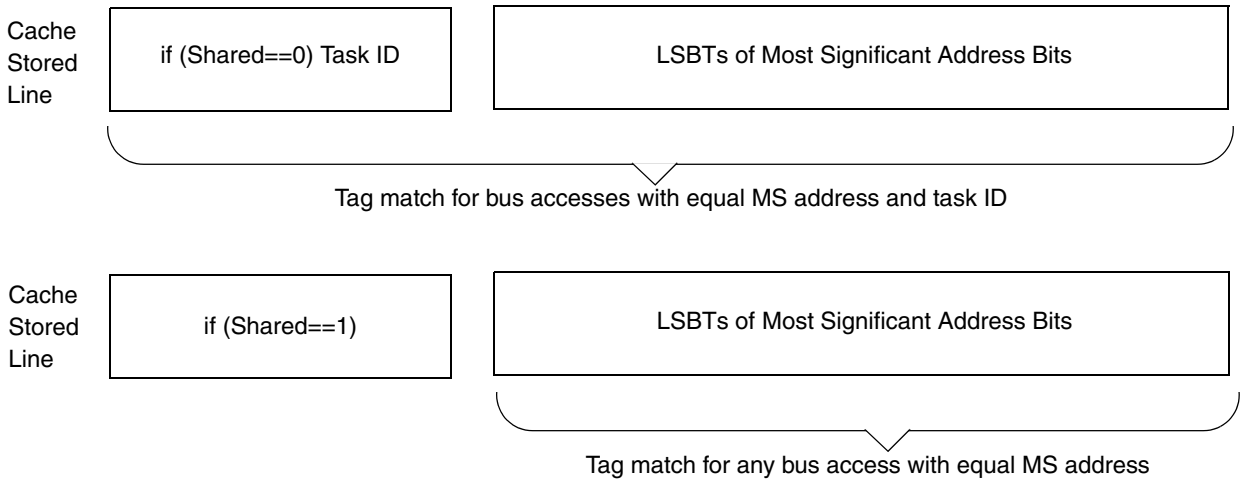

Figure 4-5. Address comparison for extended tag match identification

4.2.4 Cache debug mode

Cache debug mode is a special mode that allows observation of the cache internal state and control over the contents of the cache array.

During cache debug mode cache status will not be updated due to any access except for direct update from DCSR.

No hit data from the L1 cache will be available for load. The cache will treat all load access as non cacheable.

Cache memory array is accessible during cache debug mode using the SkyBlu dedicated interface.

4.3 SGB (Store Gather Buffer)

The SGB has the following characteristics:

- Main buffer includes 16 entries of 64 bytes of data
- Pre-buffer includes 4 address entries and two data entries of 32 bytes
- Single entry output buffer of 64 bytes of data
- Two input data write buses 256 bits each. Max throughput 2 × 256 bits per cycle.
- Single output bus: 36-bits address, 512 bits data. Max throughput 512 bits per cycle.
- Two output data read buses 256 bits each (to core via DCache).
- Propagation of all kind of writes from the core/AQ to the ELink bus (via DCache)
  - — Cacheable write-back
  - — Cacheable write-through
  - — Noncacheable
  - — Write conditional (semaphore)
  - — Guarded write accesses
  - — Non-write accesses: barriers, cache commands, messages
- Efficient gathering of multiple individual stores
- Supports zero wait state read hit
- Unaligned access support
- FIFO access ordering
- Support two levels of priority
- Programmable watermark

The SGB is a write buffer that promotes write accesses, cache commands, and memory barriers to the L2 cache. It supports two simultaneous write accesses of up to 32 bytes each or a single 64 byte write access. The accesses are promoted in order (FIFO) except for writes with a memory area attribute from the MMU that can merge into an existing valid entry if there is an address match and semaphore's write conditional access that can bypass the old entries. The SGB supports barriers that completely separates buffer entries from the new ones. It prevent new accesses merging with the old entries that were valid before barrier arrival and also prevents write conditional bypass.

The SGB monitors core read accesses and checks if the data is present in the buffer. The data is forwarded to the core in case of a cacheable read match to one of the entries. The SGB updates the DCache in case the write access is a hit in the cache at the time of outputting the write.

The SGB supports non-aligned accesses of up to 4K boundary for both read and write accesses. Non-aligned accesses that cross the 4K boundary are split into two 4K aligned accesses earlier by the core. The SGB freezes the core if the buffer is full.

The SGB asserts a rewind request to the core when a new read access matches more than a single entry in the buffer. This scenario may happen when two write accesses to the same 64 bytes are separated by a barrier command. When two writes to the same 64 bytes are located in the different entries (one in the main buffer and the other in the pre-buffer), the situation is resolved without rewind.

The SGB supports two priority levels: high (HP) and low (LP). High priority guarantees that no access from the DCache will arbitrate to the ELink before the SGB negates high priority. Low priority means that DCache misses have preference to arbitrate to the ELink before the SGB access, but prefetch will wait until SGB finishes all its requests.

The SGB promotes its content to the ELink in the following cases:

- Guarded access (HP)
- Barrier/flush in the buffer (LP or HP)
- Buffer full (LP)
- Conditional read and write in the SGB (LP)
- Read multi-hit (LP)
- Message in SGB (LP)
- Non-gatherable access in the SGB (peripheral, write conditional) (LP)
- Watermark (LP)

  The SGB supports a programable watermark. This means that it asserts write request to the CCM each time the number of buffer entries is greater than watermark threshold. The watermark configuration register is located in the MMU. Allowed values are 9 to 16, default value is 12.

Chapter 5
L2 Cache

5.1 Introduction

The L2 cache is composed of four independent banks that operate in parallel to the process program and data accesses to the external L3 cache/M3/DDR memory. Caching the accesses requested by the L1 subsystem reduces the average penalty of accessing the high latency external memories. The L2 cache also supports hardware coherency.

The L2 cache includes the following:

- A slave arbitration and tag unit
- Cache logic and arrays
- Buffers for write-through accesses, thrashes, and memory barriers
- Buffers for fetches to and from the off platform memory on a miss or a noncacheable access
- Arbiters to arbitrate between several sources for external requests.

5.2 L2 cache functional overview

This figure shows the L2 cache.

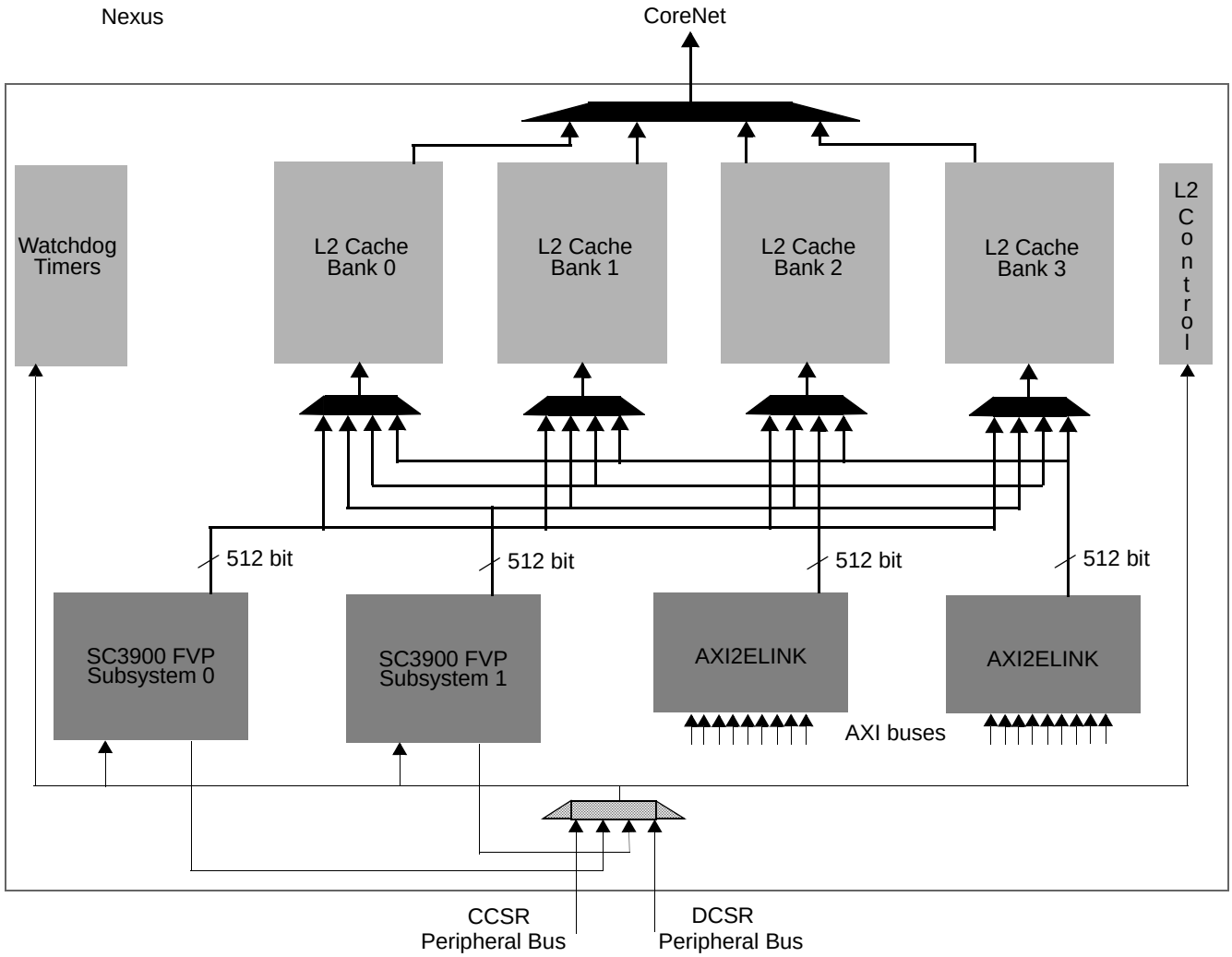

Figure 5-1. L2 cache high-level block diagram

Its main components are as follows:

- Core-to-cache interface unit (CCIU), supporting 1 to 4 cores (subsystems) connected to 4 L2 banks.
- Four L2 cache independent banks that operate in parallel to provide an increase in snoop and data bandwidth
- CoreNet bus interface unit, supporting asynchronous connection

5.2.1 L2 cache characteristics

The L2 cache characteristics are as follows:

- A size of 2 Mbytes (4 banks of 512 Kbytes) with 64 bytes per cache line (total of 8192 lines in the cache) arranged in a 16 way set-associative structure.
- 64-byte line. Can fetch the whole line at once and write back the whole line at once when a dirty line is thrashed from the cache.
- Physically addressed
- ECC support: tag, status, and data ECC (single-bit error correction, double-bit error detection)
- MESI+L hardware coherency support. Support for coherency granules (CG) of 64 bytes
- Programmable partitioning control
- Support for stashing transactions from CoreNet
- Programmable bandwidth allocation
- Selectable bank hashing
- Selectable index hashing
- WIMG attributes (Cache policies):
  - Cacheable/noncacheable on read and write (NC)
  - Cacheable write through; cacheable on read and noncacheable on write (WT).
  - Coherency required/not required
  - Guarded

5.2.2 Main features of L2 cache

Table 5-1 shows the main features of the L2 cache.

Table 5-1. L2 cache main features

Feature	Details
Input ports (Elink per each L1 subsystem)	• The L2 cache handles two input ports: ILInk and DLink. It makes an arbitration to each L2 bank with pipeline support. • The L2 cache supports partial bus width and full bus width accesses of 1, 2, 4, 8, 16, 32, 64 bytes for reads and any combination of bytes enables for writes. • An arbitration between ILink and DLink garantees minimum stall to the core. • The L2 banks arbiter arbitrates (per bank) between the core accesses, snoops, and internal reloads (from previous misses). In general, highest priority is given to snoops, then to internal reloads, and then to transactions from the L1 subsystems.
Output ports (CoreNet)	• Interfaces to the external memory via 256 bit wide master bus towards CoreNet and suppors asynchronous connection. • Transactions are fully pipelined and tagged, which allows out of order completion. • The number of beats in the burst is equal to the burst size divided by the bus size. The maximum accumulative burst size is 64 bytes, which is made in 2 beats of 256 bits. • Synchronization between two clock domains: fast clock cluster domain and slow clock CoreNet domain.

Table 5-1. L2 cache main features (continued)

Feature	Details
Access handling	• Supports the following cache policies, which are identified by the cache policy WIMG bits, defined by MMU programming and which are part of the accesses attributes signals. — Cacheable write-through—cacheable on read and on write hit and noncacheable on write miss (WT) — Cacheable write back—both read and write are cacheable (WB) — Noncacheable—noncacheable on read and write (NC). • Throughput of accesses is every other cycle by the use of cache tag pipeline and data arrays • Upon a cache miss, brings the entire line using critical word first and wraps on 64-byte boundaries. • Identifies data that is being fetched (prefetch hit), which reduces the number of wait states relative to a simple miss and reduces the external memory bus load. • Supports core reservation accesses to external memory. Reservation access to L2 cacheable location is not supported (will always succeed).
Replacement mechanism	• Uses pseudo-random cache line replacement mechanism (PLRU). • Allows cache locking by line resolution. • Partitioning mechanism by ways, which allows assignment of a subset of cache ways to reduce cache restoration penalty of a restored task. This mechanism is useful when rapid task switching is required, thereby preventing a situation in which a task thrashes important data or instructions associated with other tasks.
ECC support (in tag, status, and data)	• Single-bit error correction • Double-bit error detection
Cache programming	• Dedicated programmable cache control registers that control or reflect its operation. • Supports reading and writing memory mapped registers through memory mapped slave bus.
Hardware coherency support	• Coherency granule (CG) size of 64 bytes • Support for coherency states MESI+L • Support for an additional noncoherent state (N), in order to perform noncoherent transactions but still allow reloading of L2 cache

5.2.3 Processing states

Before enabling the L2 cache through the enable bit, the L2 cache tags, status, and lock bits should be cleared (for the exact description, see Section 5.3.1, “Enable/disable cache”). Once this occurs, it is safe to enable the L2 cache.

5.3 Functional description

Assuming the L2 bank size is 512 Kbytes and 16 ways, each cache way includes cache lines. A line is associated with a tag. Each line includes 64 bytes of data.

To locate the L2 cache controller line that matches the access, a tag is compared to those that are stored in the cache lines. All lines with the same index as the accesses are compared simultaneously. When there is a line match with a valid status for the appropriate line in the cache, the access is served by cache memory with a latency of five cycles unless there is a conflict in the cache resources.

5.3.1 Enable/disable cache

At reset negation, the cache is disabled and may be enabled by the user. When the cache is disabled, most cache activities are turned off.

Before enabling the L2 cache, the L2 cache tags, status, and lock bits should be cleared by setting L2CSR0[L2FI] and L2CSR0[L2LFC] using the same register write and then waiting until the hardware clears L2CSR0[L2FI]. Once this occurs, it is safe to enable the L2 cache.

5.3.2 Cache global invalidate command

Invalidation occurs regardless of the cache enabled. A cache invalidation operation is initiated by setting bit L2CSR0[L2FI]. Once complete, this bit is cleared automatically. All coherency granules in the L2 cache are invalidated.

L1 cache need to be invalidated using L1 cache management commands after global L2 cache invalidation.

NOTE

Writes to this bit during an invalidation operation are ignored.

5.3.3 Cache global flush command

Setting L2CSR0[L2FL] initiates an L2 flush operation, which causes all coherency granules in the modified state to be written back to backing store and all L2 entries changed to the invalid state. This bit is cleared automatically when the operation is complete.

NOTE

Writes to this bit while an L2 flush operations is in progress are ignored. Writing 1 to this bit when the cache is disabled is also ignored.

To flush the L2 cache and ensure that no valid entries exist after the flush, the following instruction sequence should be used:

1 Clear all bits of partitioning allocation register (L2PAR<0:n>) to prevent future operations from allocating in the L2 cache.
2 Set L2CSR0[L2FL].
3 Wait for L2CSR0[L2FL] to be cleared by hardware.

5.3.4 Cache global lock flush clear command

Setting L2CSR0[L2LFC] clears lock bits regardless of cache enablement. Once complete, this bit is cleared automatically. All coherency granules in the L2 cache are unlocked.

NOTE

Writes to this bit during a lock clear operation are ignored.

If L2FI and L2LFC are set simultaneously with the same register write operation, the L2 flash invalidate function and the L2 lock flash clear function are performed in parallel. In this case, hardware clears both the valid and lock bits for each coherency granule and all the tag and PLRU bits with a single pass through all indices of the cache, writing all entries in the tag, status, and victim arrays.

5.3.5 Replacement mechanism and cache partitioning scheme

The L2 cache controller uses a pseudo-random replacement strategy. When used in combination with the partitioning scheme, a deterministic replacement strategy can be achieved. The pseudo-random replacement strategy allocates empty lines first. If all the lines are valid, the line to thrash is chosen using a pseudo-random algorithm from the ways that are allocateable by the partition mechanism.

5.3.5.1 Advanced programmable replacements

The L2 cache introduces a programmable replacements scheme to keep certain address ranges from being replaced. The partition scheme is based on the number of segments (per instruction and data) that control the ways that are available for allocation.

Locking and unlocking in the L2 cache should be done only when there is no other activity in L2 cache, for example during boot sequence.

5.3.5.2 Support for stashing

The L2 cache supports the processor stash, which is a processor cache that also permits external entities to store data into it. Processor stashing is used to store latency-critical data or to control information delivered closer to a processor. L2 cache contains stash support for a cache identifier driven selection, which is an explicit stashing mode, which is the only mode supported by processor stash.

For the selected address transactions, the L2 bank acts as a snooper to provide a special snoop response and snarf the correlated data transactions. A processor stash provides stashing support only for transactions that have ASize() specification of a CG (64 bytes).

The L2 cache supports several stash transactions from the CoreNet, supported by the processor stash. To enable stash in L2 cache, enable L2PAR[CNSTASHEN] and set the stash ID unique to the specific L2 bank in L2CSR1 register.

5.3.6 Cache coherency operations

The L2 cache supports hardware coherency by supporting MESI+L coherency protocol. There is an additional state in the L2 cache, N, to support noncoherent accesses. The supported coherency resolution/ granule (CG) is 64 bytes.

The following mechanisms in the L2 cache are activated in software to keep coherency in the system:

- Memory barriers instructions
- Instructions-set flush/invalidate commands

5.3.6.1 Memory barriers

The SC3900 subsystem generally uses the following types of memory barriers:

- Memory barriers to synchronize storage operations in L1 subsystem—These cause storage operations from L1 subsystems to be synchronized in L2 coherency domain.
- Memory barriers to synchronize all storage operations globally—These cause all previous storage operations from the issuer and others to be globally performed in the coherency domain.

These types can be implemented differently in L2 cache.

5.3.6.2 Flush/Invalidate instructions

There are several instructions that arrive via ELink and that support flush/invalidate of CG in the L2 cache, as supported in the SC3900 ISA.

5.3.7 L2 bank arbitration priorities

The L2 banks arbiter arbitrates (per bank) between the core accesses, snoops, and internal reloads (from previous misses). An arbitration between the core accesses is done per L2 bank, according to the following three groups:

- Latency priority requests (LPR)—This group contains requests that would benefit from a shorter latency. For example, initial load or touch miss, or an initial STGB request due to an eviction caused by a new store request when the STGB is already full (for example, the STGB needs to drain an entry in order to make forward progress on newer stores) .
- Bandwidth priority requests (BWPR)—This group includes data or instruction pre-fetch operations or any STGB request that is not due to an STGB full condition.
- Default priority requests (DPR)—This groups includes all operations that do not fit into the LPR and BWPR groups or latency priority requests and bandwidth priority requests that have been downgraded by the L2 bank arbiter.
- Castout Buffer/Write Data Buffer (COB/WDB)

The castout buffer (COB) is a queue that serves the address transactions towards the BIU, as follows:

- Non cacheable writes
- Write-through accesses
- Flush or flush and invalidate instructions
- Castouts
- Stash
- Memory barriers

The write data buffer (WDB) serves data transactions for reads on behalf of a castout or flush and for writes on behalf of a write-through or cache-inhibited store.

5.3.8 Arbitration priorities towards CoreNet (via BIU)

The following transaction sources are subject to arbitration before they are issued to higher level memory.

- Fetch requests (from RLT)
- Cast out requests from COB/WDB
- COB/WDB are full
- Noncacheable writes, writes through, flushes, etc. (from COB/WDB)

5.3.9 Cache ECC support

Cache memory support ECC (error correction code) is done by adding seven check bits to each 64-bit memory word. ECC support

5.4 Memory map and register definitions

L2 cache status, control, and error handling is accomplished through MMRs. Shared L2 configuration and control uses the same general formats as the integrated backside L2 cache provided in previous Freescale cores, although those controls were performed through SPRs.

The memory offset of the L2 cache registers is defined in Table 5-2. This offset should be added to the base address of the SC3900 L2 cache to obtain an actual address. Note that the register access width is restricted to long size and must be aligned to the address.

Table 5-2. L2 cache memory map

Address offset from L2 cache base	Use	Section/page
000	L2 Cache Control and Status Register 0 (L2CSR0)	5.4.1/5-8

5.4.1 L2 Cache Control and Status Register 0 (L2CSR0)

The L2CSR0 register, shown in Figure 5-2, provides general control and status for the L2 cache of the processor.

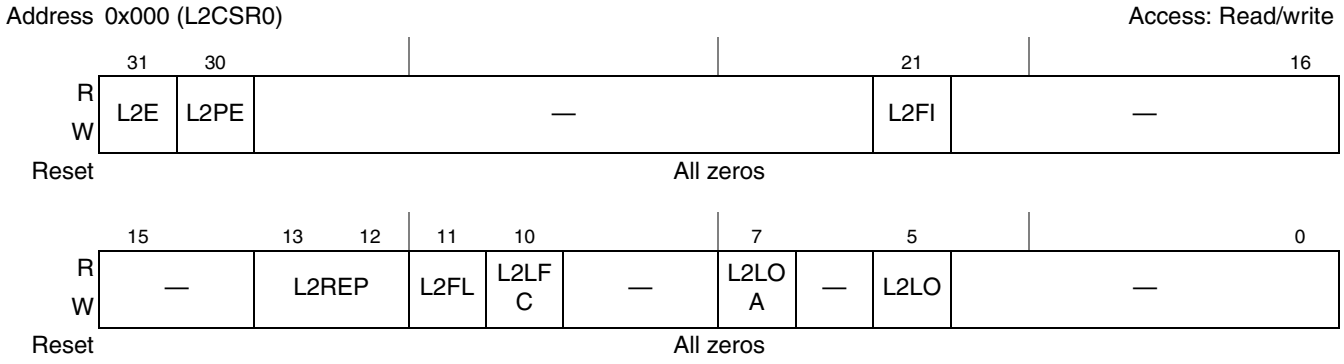

Figure 5-2. L2 Cache Control and Status Register 0 (L2CSR0)

This table describes the bit fields.

Table 5-3. Register L2CSR0 bits description

Field	Description
31 L2E	L2 cache enable. L2 cache requires software to continue to read this bit after setting it to ensure the desired value has been set before continuing on. **Note:** L2E should not be set when the L2 cache is disabled until after the L2 cache has been properly initialized by flash invalidating the cache and locks. This applies both to the first time the L2 cache is enabled as well as sequences that want to re-enable the cache after software has disabled it.
30 L2PE	L2 cache parity/ECC error checking enable. **Note:** L2PE should not bet set until after the L2 cache has been properly initialized out of reset by flash invalidation. Doing so can cause erroneous detection of errors because the state of the error detection bits are random out of reset. See Section 5.3.1, "Enable/disable cache," for more details on L2 cache initialization. **Note:** When an error injection is being performed, the value of L2PE and individual error disables are ignored and errors are always detected. Software should ensure that L2PE is set when performing error injection. **Note:** The value of L2PE must not be changed while the L2 cache is enabled.
21 L2FI	L2 cache flash invalidate. Note that Lock bits are not cleared by a L2 cache flash invalidate. Lock bits should be cleared by software at boot time to ensure that random states of the lock bits for each line do not limit allocation of those lines. See L2CSR0[L2LFC]. **Note:** Writing a 1 during any sequential operation causes undefined results. Writing a 0 during an invalidation operation is ignored. **Note:** If L2FI and L2LFC are set with the same register write operation, then the flash invalidate and the lock flash clear functions will be performed simultaneously.
13-12 L2REP	L2 line replacement algorithm. 00 SPLRU (Streaming Pseudo Least Recently Used) with Aging. With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit. On L2 cache allocations, the pseudo LRU state is updated to an intermediate state between least recently used and most recently used on most L2 cache allocations and to the most recently used state on the remainder of L2 cache allocations. 01 First-in-first-out (FIFO). 10 SPLRU (Streaming Pseudo Least Recently Used). With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit. On L2 cache allocations, the pseudo LRU state is updated to an intermediate state between least recently used and most recently used on all L2 cache allocations. 11 PLRU (Pseudo Least Recently Used).With this algorithm, the pseudo LRU state for a given index is updated to mark a given way most recently used on each L2 cache hit and all L2 cache allocations. Locks for cache lines locked with cache locking instructions are never selected for line replacement unless they are explicitly unlocked, regardless of the replacement algorithm.

Table 5-3. Register L2CSR0 bits description (continued)

Field	Description
11 L2FL	L2 cache flush. L2FL should not be set when the L2 cache is not currently enabled (L2E should already be 1). If L2FL is set and the L2 cache is not enabled, the flush will not occur and the L2FL bit will remain set. **Note:** To flush the L2 cache and ensure that no valid entries exist after the flush, the following sequence should be used: Clear all the bits of L2PAR0–L2PAR3 to prevent further allocations; Read L2PAR0–L2PAR3 to ensure that the changes are in effect, Set L2CSR0[L2FL], Continue to read L2CSR0[L2FL] until it reads 0.
10 L2LFC	L2 cache lock flash clear. On boot, the processor should set this bit to clear any lock state bits which may be randomly set out of reset, prior to enabling the L2 cache.
7 L2LOA	L2 cache lock overflow allocate. Note that cache line locking in L2 is persistent.
5 L2LO	L2 cache lock overflow.

5.4.2 L2 Cache Control and Status Register 1 (L2CSR1)

The L2CSR1 register, shown in Figure 5-3, provides general control and status for the L2 cache of the processor.

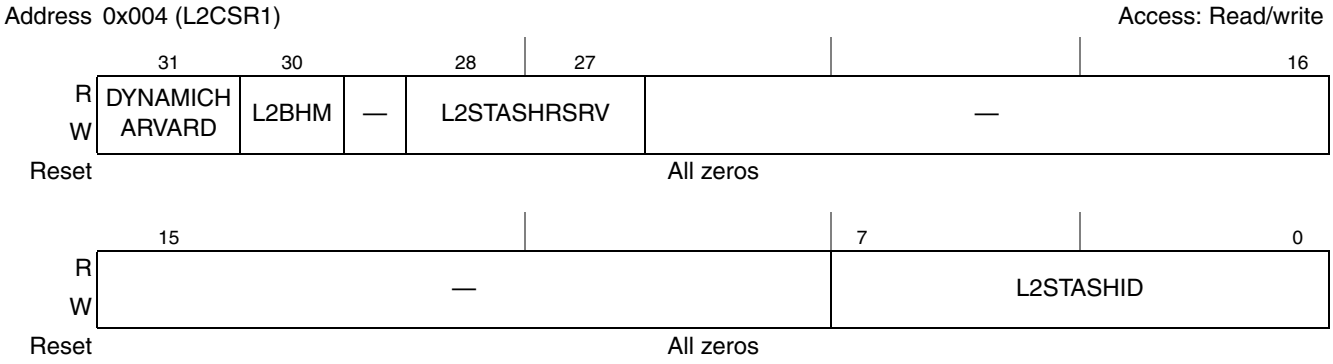

Figure 5-3. L2 Cache Control and Status Register 1 (L2CSR1)

This table describes the bit fields.

Table 5-4. Register L2CSR1 bits description

Field	Description
31 DYNAMICHARVARD	Dynamic harvard mode. 0 Enabled. Cacheable instruction fetches requested by the processor that miss are requested from CoreNet as non coherent (Memory coherence required = 0). When the line is allocated it is marked to allow a hit from instruction fetches but not data accesses. 1 Disabled. Cacheable instruction fetches requested by the processor that miss are requested from CoreNet as coherent (Memory coherence required = 1). When the line is allocated it is marked to allow a hit from instruction fetches and data accesses.
30 L2BHM	Bank hash mode. 0 Use decode hash (bits 7:6 of real address). 1 Use XOR hash (bits 21:6 of real address).
28-27 L2STASHRSRV	L2 stashing reserved resources. The number of resources per bank in which to allocate only stashes. 00 Allocate opportunistically with general resources(default) 01 One resource 10 Two resources 11 Three resources
7-0 L2STASHID	L2 cache stash ID. Contains the cache target identifier to be used for external stash operations directed to this processor's L2 cache. A value of 0 for L2STASHID prevents the L2 cache from accepting external stash operations. L2 cache supports only stash ID values of 8 and larger (that is values between 8 and 255); values from 1 to 7 are illegal.

5.4.3 L2 Configuration Register (L2CFG0)

L2CFG0, shown in Figure 5-4, provides configuration information for the L2 cache.

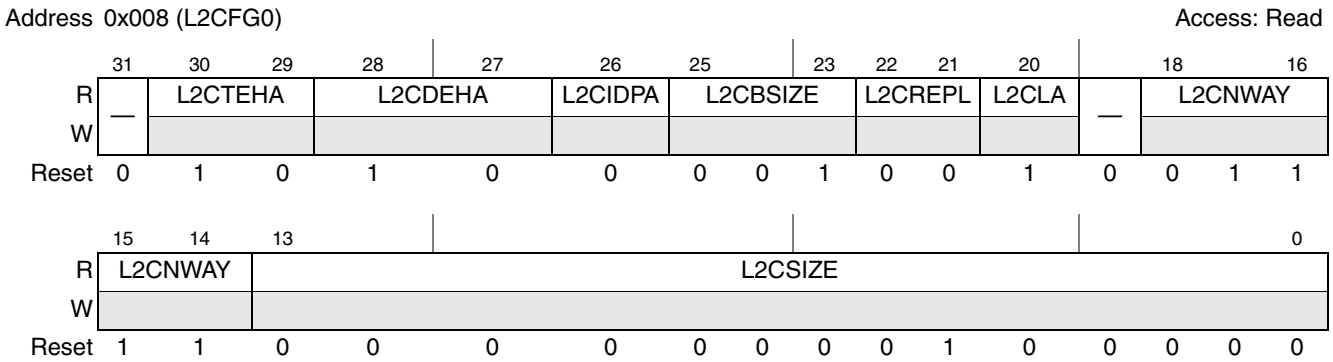

Figure 5-4. L2 Configuration Register (L2CFG0)

This table describes the bit fields.

Table 5-5. Register L2CFG0 bits description

Field	Description
30-29 L2CTEHA	L2 cache tags error handling available. 0b10 indicates single bit ECC correction, double bit ECC detection.
28-27 L2CDEHA	L2 cache data error handling available. 0b10 indicates single bit ECC correction, double bit ECC detection available.
26 L2CIDPA	Cache instruction and data partitioning available. 0 indicates not available.
25-23 L2CBSIZE	Cache line size. 1 indicates 64 bytes.
22-21 L2CREPL	Cache default replacement policy. This is the default line replacement policy at power-on-reset. If an implementation allows software to change the replacement policy it is not reflected here.0 indicates streaming pseudo-LRU.
20 L2CLA	Cache line locking available. 1 indicates available.
18-14 L2CNWAY	Number of cache ways minus 1. 15 indicates 16 ways.
13-0 L2CSIZE	Cache size as a multiple of 64 Kbytes. 32 indicates 2048-Kbyte cache.

5.4.4 L2 cache partitioning registers

L2PIR*n*, L2PAR*n*, and L2PWR*n* are sets of registers which are used to define how individual transactions performed by the L2 cache are allocated. The number of registers *n* may vary between implementations, but for any given value *n* supported by an implementation, the same number of registers exist for L2PIR, L2PAR, and L2PWR. The number of registers *n* implemented represents the number of different allocation policies that can be applied at any given time.

Each transaction sent to the L2 cache by a processor is tagged with an L2 Partition ID identifier. It is used to distinguish which allocation policies should be used when the L2 cache processes transactions. MMU segment descriptor programing and core number inside the cluster define L2 Partitioning ID value, allowing for a unique identifier for each task in the cluster. For more details, see Chapter 3, “Memory Management Unit.”

Let *id* be the identifier for a transaction presented to the L2 cache and *n* be the number of different allocation poilicies implemented (i.e. the number *n* of register implemented). 31 – *id* corresponds to a column of bits in the L2PIR*n* registers and is used to determine which allocation policies are to be applied as follows:

```
bit_num ← 31 - id
policy ← 0
ways ← 0
for reg_num = 0 to n - 1
    if L2PIR[reg_num]bit_num = 1 | stash then
        policy ← policy | L2PAR[reg_num]
        if instruction fetch & policyIRDALLOC then
```

```
            ways ← ways | L2PWR[reg_num]
        else if data read & policy_DRDALLOC then
            ways ← ways | L2PWR[reg_num]
        else if data store & policy_DSTALLOC then
            ways ← ways | L2PWR[reg_num]
        else if stash & policy_STALLOC then
            ways ← ways | L2PWR[reg_num]
endfor
if instruction fetch & policy_IRDALLOC then
    allocate line in ways
else if data read & policy_DRDALLOC then
    allocate line in ways
else if data store & policy_DSTALLOC then
    allocate line in ways
else if stash & policy_STALLOC then
    allocate line in ways
else
    line is not allocated
```

L2PIR*n* maps a possible set of 32 identifiers to specific allocation policies. L2PAR*n* and L2PWR*n* are used to process the allocation. L2PAR*n* determines what allocation policy is used. L2PWR*n* determines which ways the allocation may occur in.

Note that stash transactions that are targetted to the L2 cache in the cluster do not provide identifiers for the purpose of determining allocation policy and way selection. Instead, stashes behave as if the identifier for the transaction has bits set in all of L2PIR*n*.

L2 cache partitioning only affects when a line in the cache may be allocated or not and which ways in the cache it may be allocated in. Transactions to the L2 cache which do not require allocation (for example a load operation to an address that is present in the L2 cache) are unaffected by the settings of L2PIR*n*, L2PAR*n*, and L2PWR*n*.

5.4.4.1 L2 Cache Partitioning Identification Registers (L2PIR*n*)

L2PIR*n*, shown in Figure 5-5, provides controls for partitioning of the L2 cache based on identifiers attached to the L2 cache transactions from processors. L2PIR*n* is a set of registers each containing a bit vector of 32 bits. The identifier sent with each transaction to the L2 cache is used to select the same relative bit (31 - *id*) in each of the L2PIR*n* registers. If the bit is set in the register, then that register number is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

If more than one bit for each identifier is set among the group of L2PIR*n* registers, the allocation policy used is the logical OR of the corresponding L2PAR registers and the ways available for allocation is the logical OR of the L2PWR registers for which the corresponding L2PAR registers allow allcoation. For example, if bit 1 is set in L2PIR0 and bit 1 is set in L2PIR1 then the allocation policy is L2PAR0 | L2PAR1 and the ways available for allocation are defined by the OR of the L2PWR registers which corresond to L2PAR register which allow the allocation.

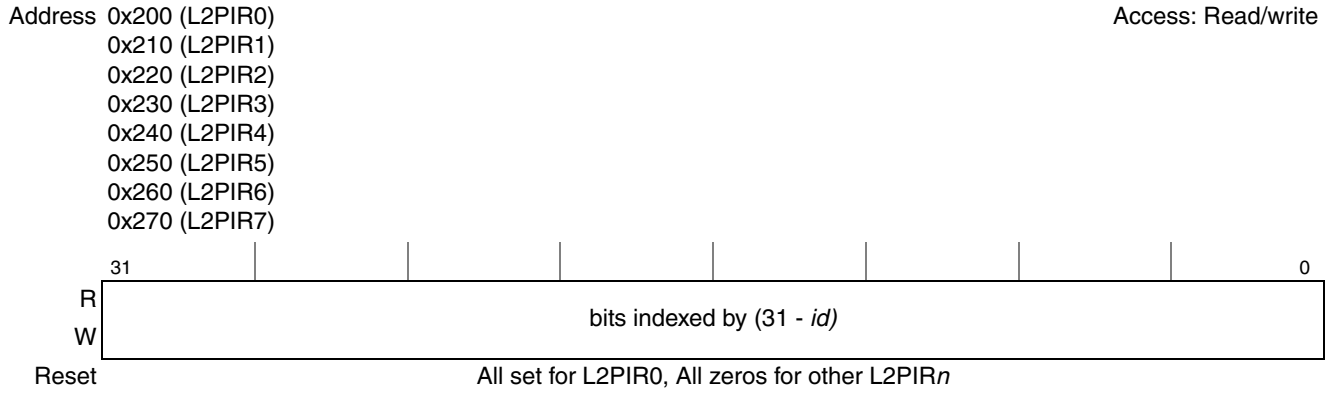

Figure 5-5. L2 Cache Partitioning Identification Registers (L2PIR*n*)

5.4.4.2 L2 Cache Partitioning Allocation Registers (L2PAR*n*)

L2PAR*n*, shown in Figure 5-6, provides controls for partitioning of the L2 cache based on which allocation policy is determined from the L2PIR*n* registers. If the bit associated with an *id* of a transaction sent to the L2 cache is set in one of the L2PIR*n* registers, then that register number (0 - *n*) is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

L2PAR*n* controls whether a line should be allocated based on the type of transaction to be performed by the L2 cache. The types of transactions which are distinguished are store type operations (store, store conditional), load type operations (load, dfetch and dfetch and lock set), instruction read and fetch, and stash operations targeted to the L2 cache.

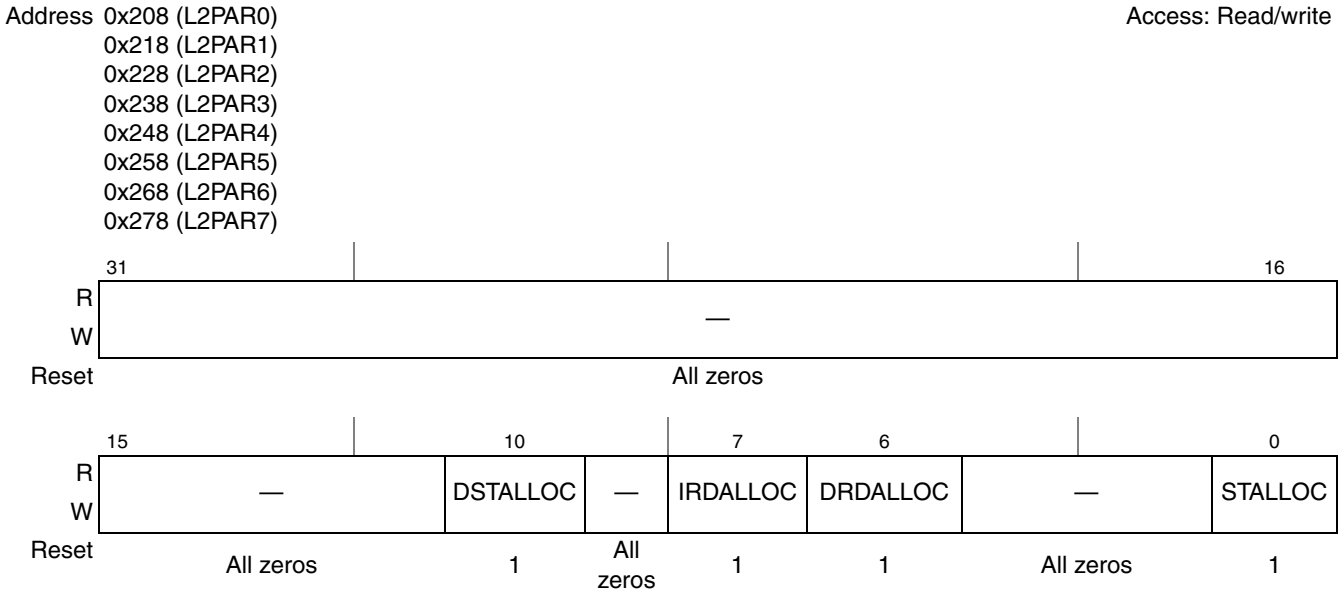

Figure 5-6. L2 Cache Partitioning Allocation Registers (L2PAR*n*)

This table describes the bit fields.

Table 5-6. L2PAR*n* field descriptions

Field	Description
10 DSTALLOC	Data store allocation control. 0 Cacheable store and store conditional instructions that miss in the L2 will not allocate unless enabled by another L2PAR*n*[DSTALLOC]. 1 Cacheable store and store conditional instructions that miss in the L2 will attempt to allocate in one of the ways defined by L2PWR*n*[WAY].
7 IRDALLOC	Instruction read and fetch allocation control. 0 Cacheable instruction fetches that miss in the L2 will not allocate unless enabled by another L2PAR*n*[IRDALLOC]. 1 Cacheable instruction fetches that miss in the L2 will attempt to allocate in one of the ways defined by L2PWR*n*[WAY].
6 DRDALLOC	Data read allocation control. 0 Cacheable load and dfetch instructions that miss in the L2 will not allocate unless enabled by another L2PAR*n*[DRDALLOC]. 1 Cacheable load and dfetch instructions that miss in the L2 will attempt to allocate in one of the ways defined by L2PWR*n*[WAY]. Any cache locking operation that have DRDALLOC set to 0 will not have the line locked because the L2 will not attempt to allocate the line.
0 STALLOC	Stashing allocation control. 0 Stash requests that miss in the L2 will not allocate unless enabled by another L2PAR*n*[STALLOC]. 1 Stash requests that miss in the L2 will attempt to allocate in one of the ways defined by L2PWR*n*[WAY]. Stash requests do not supply *id* values that index into L2PIR*n* registers, but instead examine all L2PARn registers to determine allocation policy.

5.4.4.3 L2 Cache Partitioning Way Registers (L2PWR*n*)

L2PWR*n*, shown in Figure 5-7, provides controls for partitioning of the L2 cache based on which allocation policy is determined from the L2PIR*n* registers. If the bit associated with an *id* of a transaction sent to the L2 cache is set in one of the L2PIR*n* registers, then that register number (0 - *n*) is used to index among the allocation policies represented by L2PAR*n* and L2PWR*n*.

L2PWR*n* controls which ways are available for a line to allocate into should allocation for the transaction be allowed by L2PAR*n*. The ways are represented as a bit vector where way *x* is represented by bit 31 - *x*. Only bits 31:32-*w* are implemented in each L2PWR*n* register, where *w* represents the number of ways in the L2 cache that are implemented.

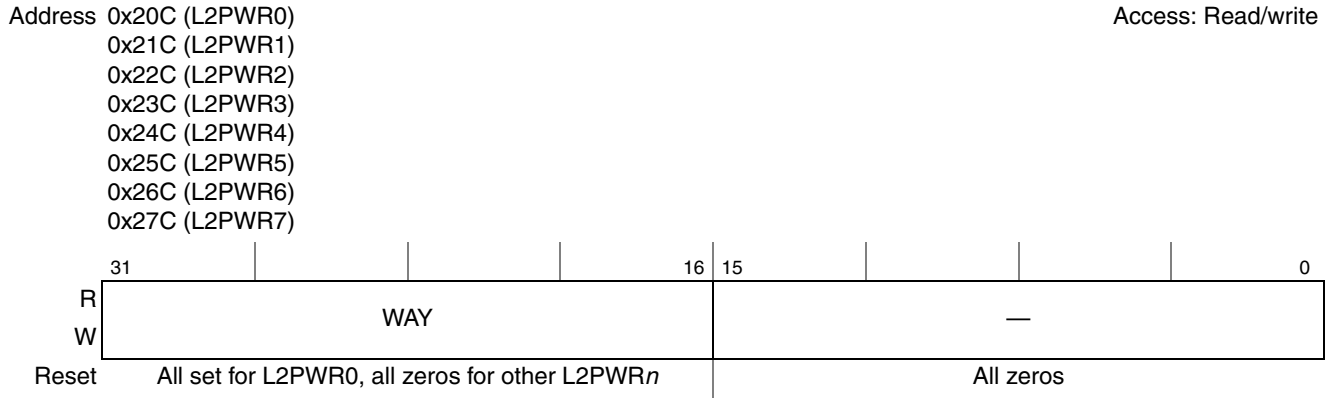

Figure 5-7. L2 Cache Partitioning Way Registers (L2PWR*n*)

This table describes the bit fields.

Table 5-7. L2PWR*n* field descriptions

Field	Description
31-16 WAY	Each bit that is set represents a way number into which the transaction can allocate. Multiple bits can be set, representing multiple ways that are available for allocation with this allocation policy. 0 The ways corresponding to bits set to 0 are not available for allocation using this allocation policy. 1 The ways corresponding to bits set to 1 are available for allocation using this allocation policy.
15-0	Reserved, should be cleared.

5.4.5 L2 Cache Error Disable Register (L2ERRDIS)

L2ERRDIS, shown in Figure 5-8, provides general control for disabling error detection in the L2 cache of the processor.

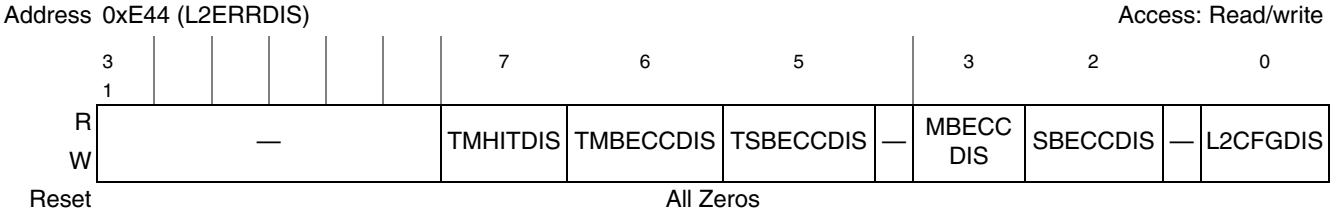

Figure 5-8. L2 Cache Error Disable Register (L2ERRDIS)

This table describes the bit fields.

Table 5-8. L2ERRDIS field descriptions

Field	Description
7 TMHITDIS	Tag/status multi-way hit error disable. 0 Tag multi-way hit detection enabled. 1 Tag multi-way hit error detection disabled. **Note:** When error injection is being performed, the value of TMHITDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and TMHITDIS is clear when performing error injection to the tags.
6 TMBECCDIS	Tag Multiple-bit ECC error disable. 0 Tag Multiple-bit ECC error detection enabled. 1 Tag Multiple-bit ECC error detection disabled. **Note:** When error injection is being performed, TMBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
5 TSBECCDIS	Tag ECC error disable. 0 Tag Single-bit ECC error detection enabled. 1 Tag Single-bit ECC error detection disabled. **Note:** When error injection is being performed, TSBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
3 MBECCDIS	Data Multiple-bit ECC error disable. 0 Data Multiple-bit ECC error detection enabled. 1 Data Multiple-bit ECC error detection disabled. **Note:** When error injection is being performed, the value of MBECCDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and MBECCDIS is clear when performing error injection to the data.
2 SBECCDIS	Data Single-bit ECC error disable. 0 Data Single-bit ECC error detection enabled. 1 Data Single-bit ECC error detection disabled. **Note:** When error injection is being performed, the value of SBECCDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software should ensure that L2PE is set and SBECCDIS is clear when performing error injection to the data.
0 L2CFGDIS	L2 configuration error disable 0 L2 configuration error detection enabled 1 L2 configuration error detection disabled

5.4.6 L2 Cache Error Detect Register (L2ERRDET)

L2ERRDET, shown in Figure 5-9, provides general status and information for errors detected in the L2 cache of the processor.

Address 0xE40 (L2ERRDET) Access: Write 1 to clear

	31							7	6	5		3	2		0
R	MULL2ERR							TMHITERR	TMBECCERR	TSBECCERR	—	MBECCERR	SBECCERR	—	L2CFGERR
W	w1c							w1c	w1c	w1c		w1c	w1c		w1c
Reset	All Zeros														

Figure 5-9. L2 Cache Error Detect Register (L2ERRDET)

This table describes the bit fields.

Table 5-9. L2ERRDET field descriptions

Field	Description
31 MULL2ERR	Multiple L2 errors. Writing a 1 to this bit location will reset the bit. 0 Multiple L2 errors of the same type were not detected. 1 Multiple L2 errors of the same type were detected.
7 TMHITERR	Tag multi-way hit detected. Writing a 1 to this bit location will reset the bit. 0 Tag multi-way hit not detected. 1 Tag multi-way hit detected.
6 TMBECCERR	Tag Multiple-bit ECC error detected. Writing a 1 to this bit location will reset the bit. 0 Tag Multiple-bit ECC error not detected. 1 Tag Multiple-bit ECC error detected.
5 TSBECCERR	Tag ECC error detected. Writing a 1 to this bit location will reset the bit. 0 Tag Single-bit ECC not detected. 1 Tag Single-bit ECC error detected.
3 MBECCERR	Data Multiple-bit ECC error detected. Writing a 1 to this bit location will reset the bit. 0 Tag Multiple-bit ECC error not detected. 1 Tag Multiple-bit ECC error detected.
2 SBECCERR	Data ECC error detected. Writing a 1 to this bit location will reset the bit. 0 Tag Single-bit ECC error not detected. 1 Tag Single-bit ECC error detected.
0 L2CFGERR	L2 configuration error detected. Writing a 1 to this bit location will reset the bit. 0 L2 configuration error not detected. 1 L2 configuration error detected.

5.4.7 L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

L2ERRINTEN, shown in Figure 5-10, provides general status and information for errors detected in the L2 cache of the processor.

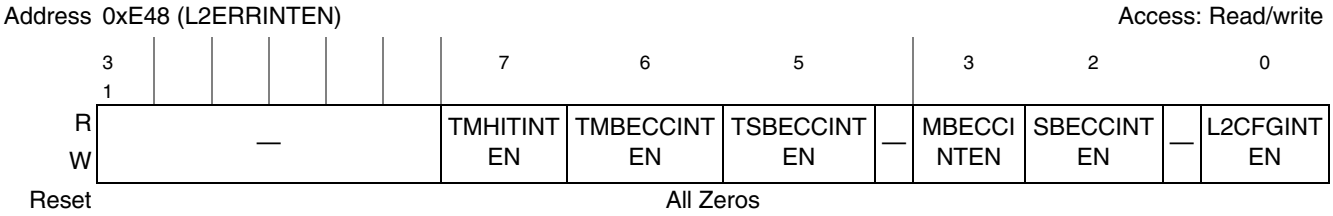

Figure 5-10. L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

This table describes the bit fields.

Table 5-10. L2ERRINTEN field descriptions

Field	Description
7 TMHITINTEN	Tag multi-way hit interrupt reporting enable. 0 Tag multi-way hit interrupt reporting disabled. 1 Tag multi-way hit interrupt reporting enabled.
6 TMBECCINTEN	Tag Multiple-bit ECC error interrupt reporting enable. 0 Tag multiple-bit ECC error interrupt reporting disabled. 1 Tag multiple-bit ECC error interrupt reporting enabled.
5 TSBECCINTEN	Tag ECC interrupt reporting enable. 0 Tag single-bit ECC error interrupt reporting disabled. 1 Tag single-bit ECC error interrupt reporting enabled.
3 MBECCINTEN	Data Multiple-bit ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data Multiple-bit ECC error interrupt reporting disabled. 1 Data Multiple-bit ECC error interrupt reporting enabled.
2 SBECCINTEN	Data ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data Single-bit ECC error interrupt reporting disabled. 1 Data Single-bit ECC error interrupt reporting enabled.
0 L2CFGINTEN	L2 configuration error interrupt reporting enable. 0 L2 configuration interrupt reporting disabled. 1 L2 configuration error interrupt reporting enabled.

5.4.8 L2 Cache Error Control Register (L2ERRCTL)

L2ERRCTL, shown in Figure 5-11, provides thresholds and counts for errors detected in the L2 cache of the processor.

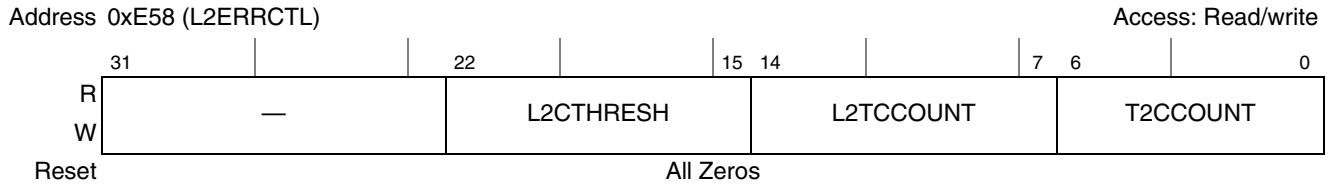

Figure 5-11. L2 Cache Error Control Register (L2ERRCTL)

This table describes the bit fields.

Table 5-11. L2ERRCTL field descriptions

Field	Description
22-15 L2CTHRESH	L2 cache threshold. Threshold value for the number of ECC single-bit errors that are detected before reporting an error condition. L2CTHRESH is compared to L2TCCOUNT and L2CCOUNT each time a single-bit ECC error is detected. A value of 0 in this field will cause the reporting of a single bit ECC error upon the first occurrence of such an error.
14-7 L2TCCOUNT	L2 tag ECC single-bit error count. Counts ECC single-bit errors in the L2 tags detected. If L2TCCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for tags is enabled. Software should clear this value when such an error is reported to reset the count.
6-0 T2CCOUNT	L2 data ECC single-bit error count. Counts ECC single-bit errors in the L2 data detected. If L2CCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for data is enabled. Software should clear this value when such an error is reported to reset the count.

5.4.9 L2 Cache Error Capture Address Registers (L2ERRADDR and L2ERREADDR)

L2ERRADDR and L2ERREADDR provides the real address of a captured error detected in the L2 cache of the processor. The real address is 36 bits.

5.4.10 L2 Cache Error Capture Data Registers (L2CAPTDATALO and L2CAPTDATAHI)

L2CAPTDATALO and L2CAPTDATAHI provides the array data of a captured error detected in the L2 cache of the processor. L2CAPTDATALO captures the lower 32 bits of the doubleword and L2CAPTDATAHI captures the upper 32 bits of the doubleword.

If the captured error is a data ECC error, then these registers contain the data associated with the error. If the captured error is a tag/status ECC error, then L2CAPTDATALO contains the following:

L2CAPTDATALO = low-order 19 bits of the tag || 0b00000 || status[7:0]

L2CAPTDATAHI = 0x000000 || high-order 8 bits of the tag

5.4.11 L2 Cache Capture ECC Syndrome Register (L2CAPTECC)

L2CAPTECC provides both the calculated and stored ECC syndrome of a captured error detected in the L2 cache of the processor. Tag and status ECC syndromes are left-padded with the appropriate number of zeros.

5.4.12 L2 Cache Error Attribute Register (L2ERRATTR)

L2ERRATTR, shown in Figure 5-12, provides extended information for errors detected in the L2 cache of the processor.

Address 0xE4C (L2ERRATTR) Access: Read/write

	31–28	27–20	19–16	15–13	12	11–3	2–1	0
R W	DWNUM	—	TRANSSRC	—	TRANS TYPE	—	CORE	VALINFO
Reset	All Zeros							

Figure 5-12. L2 Cache Error Attribute Register (L2ERRATTR)

This table describes the bit fields.

Table 5-12. L2ERRATTR field descriptions

Field	Description
31-28 DWNUM	For data ECC errors, contains the double-word number of the detected error. For tag/status ECC errors, contains which way of the tag/status encountered the error.
20-16 TRANSSRC	Transaction source for detected error 00000 External (snoop) 10000 Internal (instruction) 10001 Internal (data) 00001–01111 Not Implemented 10010–11111 Not Implemented
13-12 TRANSTYPE	Transaction type for detected error 00 Snoop 01 Write 10 Read 11 Not Implemented
3-1 CORE	Core ID that issued TRANSTYPE. If the transaction was from a snoop, this field is undefined.
0 VALINFO	L2 capture registers valid. 0 L2 capture registers contain no valid information or no enabled errors were detected. 1 L2 capture registers contain information of the first detected error which has reporting enabled. Software must clear this bit to unfreeze error capture so error detection hardware can overwrite the capture address/data/attributes for a newly detected error.

5.4.13 L2 Cache Error Injection Control Register (L2ERRINJCTL)

L2ERRINJCTL, shown in Figure 5-13, provides control for injecting errors into both the tags and data array for the L2 cache of the processor.

NOTE

When error injection is being performed, the value of specific error disables in L2ERRDIS and L2CSR0[L2PE] are ignored and errors are always detected. Software must ensure that L2PE is set and individual disables in L2ERRDIS are clear when performing error injection to the data or tags.

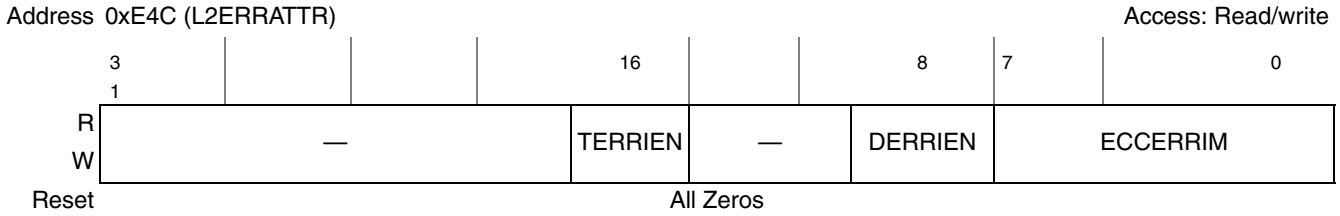

Figure 5-13. L2 Cache Error Injection Control Register (L2ERRINJCTL)

This table describes the bit fields.

Table 5-13. L2ERRINJCTL field descriptions

Field	Description
16 TERRIEN	L2 tag error injection. 0 No tag errors are injected. 1 All subsequent entries written to the L2 tag array have the tag ECC bits inverted as specified in the ECC error injection masks. Tag error injection is determined by L2ERRINJHI[17:0] and L2ERRINJLO[31:0].
8 DERRIEN	L2 data error injection. 0 No data errors are injected. 1 Subsequent entries written to the L2 data array have data or data ECC bits inverted as specified in the data and ECC error injection masks . Data error injection is determined by L2ERRINJHI[31:0] and L2ERRINJLO[31:0] and tag error injection is determined by L2ERRINJCTL[ECCERRIM].
7-0 ECCERRIM	Error injection mask for the ECC syndrome bits. When DERRIEN=1, the eight ECCERRIM bits map to the 8 data ECC bits for each 64 bits of data. When TERRIEN=1, the low order 7 ECCERIM bits map to the 7 tag ECC bits for each 40-bit tag.

5.4.14 L2 Cache Error Injection Mask Registers (L2ERRINJLO and L2ERRINJHI)

L2ERRINJLO and L2ERRINJHI provides the injection mask describing how errors are to be injected into the data path doubleword in the L2 cache of the processor. L2ERRINJLO provides the mask for the lower 32 bits of the doubleword and L2ERRINJHI provides the mask for the upper 32 bits of the doubleword. A set bit in the injection mask causes the corresponding data path bit to be inverted on data array writes when L2ERRINJCTL[DERRIEN] = 1 or tag array writes when L2ERRINJCTL[TERRIEN] = 1.

Chapter 6
Cache Management

6.1 Cache management overview

The SC3900 cache system supports a rich set of control functions (for example, pre-fetching operations, coherency operations, and status query operations) that the user can activate on the caches. The unit managing these operations is named the CME (cache management engine). The CME is divided into three subunits, one of which is an interface unit. These subunits are responsible for managing the instruction and data caches; the interface unit also handles configuration requests.

All cache operations are communicated to the caches via the address buses, are marked with attributes that identify them as a cache management operation, and specify the requested operation. In general, the CME holds a set of task channels that the user configures. The CME then monitors the respective address buses (program and data) and drives accesses on them from the active task channels using free access slots that are not used by functional core accesses.

6.2 Functional overview of the CME

For a summary of CME features, see Section 6.2.2, "Main features of the CME."

The CME has the following three main components:

Configuration Unit — This unit is responsible for the configuration of the CME through the core data bus and peripheral bus.

Data Generation Unit — This unit is responsible for DCache maintenance instructions generation and insertion into the core data buses.

Instruction Generation Unit — This unit is responsible for ICache maintenance instructions generation and insertion into the core instruction bus.

This figure shows the CME integration.

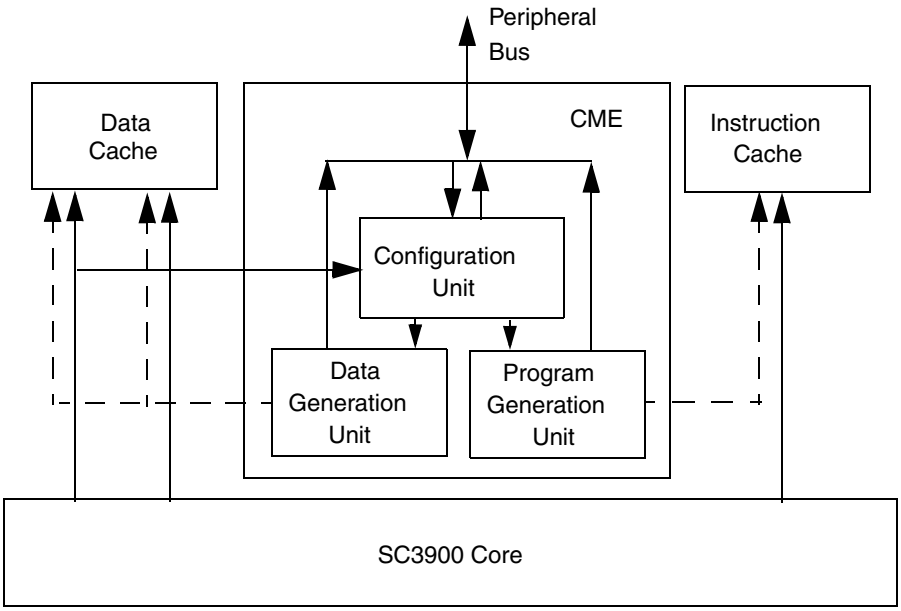

Figure 6-1. CME high-level block diagram

6.2.1 Understanding cache control operations

Cache control operations can be configured either by the core, using dedicated cache control instructions, or by configuration accesses to memory mapped registers on the peripheral bus that is used to configure all the subsystem memory mapped registers. Note that external masters on the peripheral bus can also configure the CME. For more information on the cache commands, see the *SC3900 FVP Core Reference Manual*.

Cache control commands can be characterized in the following ways:

- Granular cache instructions are instructions that operate on one cache granule, which is the minimum data unit on which cache coherency is managed. A cache granule is an aligned, contiguous range of 64-bytes.
- A block cache instruction is a more complex operation that can be defined on a set of cache granules, such as a contiguous memory range or even a two-dimensional memory range.

6.2.1.1 Which cache instructions are implemented by the CME

The SC3900 instruction set supports various types of granular cache maintenance instructions, which are described in the *SC3900 FVP Core Reference Manual*. These cache instructions allow for cache hierarchy performance optimization, coherency accelerator, operating system support, and debug. In addition to granular cache maintenance instructions, the SC3900 ISA defines block cache maintenance instructions with a similar functionality that works on the address range. The CME implements all block cache maintenance instructions. It also provides the same functionality through peripheral bus configuration, allowing external masters to utilize available cache resources. Using special query and debug channels,

external masters can also generate data query instructions and physical invalidation. External maintenance instructions are programmed by using a set of dedicated registers.

This table lists the instruction cache instructions, their variants, and their functionality.

Table 6-1. Summary of cache commands and their functionality

Functional group	Instruction	Type	Functionality
Query	DQUERY.L1	Granular	Queries the MMU and L1 data cache
	DQUERY.L12	Granular	Queries the MMU, L1 data cache and L2 cache
Pre-fetch	DFETCH.L12	Granular	Fetches the specified cache granule to the L2 cache and L1 data cache. Intend to write (ITW) attribute override can be added to optimize L2 performance.
	DFETCH.L2	Granular	Fetches the specified cache granule to the L2 cache using translation of data MMU. Intend to write (ITW) attribute override can be added to optimize L2 performance.
	DFETCHB.L12	Block	Fetches the specified block of cache granules to the L2 cache and L1 data cache. Intend to write (ITW) attribute override can be added to optimize L2 performance.
	DFETCHB.L2	Block	Fetches the specified block of cache granules to the L2 cache using translation of data MMU. Intend to write (ITW) attribute override can be added to optimize L2 performance.
	DFETCHB.LCK.L2	Block	Fetches the specified block of cache granules to the L2 cache using translation of data MMU and locks them there. Intend to write (ITW) attribute override can be added to optimize L2 performance.
	PFETCHB.L12	Block	Fetches the specified block of cache granules to the L2 cache and L1 program cache
	PFETCHB.L2	Block	Fetches the specified block of cache granules to the L2 cache using translation of program MMU
	PFETCHB.LCK.L2	Block	Fetches the specified block of cache granules to the L2 cache using translation of program MMU and locks them there
Unlocking	DUNLOCKB.L2	Block	Unlocks the specified block of cache granules in the L2 cache using translation of program MMU
	PUNLOCKB.L2	Block	Unlocks the specified block of cache granules in the L2 cache using translation of program MMU.

Table 6-1. Summary of cache commands and their functionality (continued)

Functional group	Instruction	Type	Functionality
Coherency	DCM.SYNC.L12	Granular	Synchronizes the specific L1 data cache and L2 cache granule with the memory.
	DCMB.SYNC.L1	Block	Synchronizes the whole L1 data cache granules with the L2 cache. Flushes SGB.
	DCM.FLUSH.L1	Granular	Flushes the specific L1 data cache granule to the L2 cache.
	DCM.FLUSH.L12	Granular	Flushes the specific L1 data cache and L2 cache granule to the memory.
	DCMB.FLUSH.L12	Block	Invalidates the specific block of L1 data cache granules and flushes this block from L2 cache to the memory. Does not flush SGB.
	DCM.INVAL.L1	Granular	Invalidates the specific L1 data cache granule.
	DCM.INVAL.L12	Granular	Invalidates the specific L1 data cache and L2 cache granule.
	DCMB.INVAL.L1	Block	Invalidates the specific block of L1 data cache granules. Does not flush SGB.
	DCMB.INVAL.L12	Block	Invalidates the specific block of L1 data cache and L2 cache granules. Does not flush SGB.
	PCMB.INVAL.L1	Block	Invalidates the specific block of L1 program cache granules.
	PCMB.INVAL.L12	Block	Invalidates the specific block of L1 program cache and L2 cache granules.
CME Control	CCMD.SUSP	CME Management	Suspends all active cache commands in the CME
	CCMD.RESM	CME Management	Resumes all suspended cache commands in the CME

6.2.1.2 How the channels access the cache

Granular cache maintenance instructions are treated like regular reads and writes, without significant CME intervention.

The data generation unit and program generation unit each support eight cache maintenance instructions channels and one cache access port. There are two additional channels for external query and debug. Channel allocation is dynamic and does not require complex knowledge during the instruction issue.

Each channel executes its cache instruction and is allowed access to the cache in order of programming. Actual cache access is inserted on the core bus into the appropriate slot. Cache maintenance instruction insertion is almost nonintrusive: the machine selects insertion slots from two data and program buses on cycles without core memory reads. The only side effect is an increased load on the external memory bus, which should not reduce performance due to a high cache hit ratio.

6.2.2 Main features of the CME

The CME's main features are as follows:

- Flexible state machine that allows the offloading of cache management tasks from the SC3900 FVP core
  - — Ten unique cache maintenance instructions
  - — Eight DCache maintenance block instructions channels
  - — Eight ICache maintenance block instructions channels
  - — External query channel that can execute regular L1, L2 and memory queries and ICache barriers
  - — Debug channel that can work in the debug mode and allows physical invalidation
  - — Machine state observability through peripheral bus registers
- Cache maintenance instruction initialization through dedicated core instructions or external peripheral bus accesses
  - — Initialization by core is very fast
  - — Full external controlability and observability
  - — Simple programming mode provides automatic channel allocation
- Advanced implementation of block cache maintenance instructions
  - — Two-dimensional cache instruction address generation
  - — Support for various types of block prefetch, invalidation and locking
    - – Software prefetch allows DMA like pre allocation optimized for different memory usage patterns (read or write after read)
    - – Full coherency accelerator support through flush/sync/invalidate cache maintenance instructions
    - – Support for L2 cache line resolution locking/unlocking
- Background maintenance instruction insertion into core buses
  - — Reuse bus slots without reads on two data and program core buses
  - — Quick suspend/resume functionality removes possible overhead in memory critical code sequences
  - — Cache memory bus saturation prevention using cache full indication
    - – Low cache activity, the lowest level, allows all accesses
    - – Half full and above allows only special channels and granular cache commands
- Advanced set of CME management commands
  - — Suspend/resume functionality for performance fine tuning
- Independent support for instruction and data caches
  - — One DCache and one ICache instruction per cycle
- Support for multilevel cache architectures
- Full MMU integration and support
  - — Allows both virtual and physical addressing

  - Segment miss, multiple segment descriptor hit and non cacheable instruction containment
- Various methods of maintenance instruction completion detection
  - Dedicated interrupt assertion
  - Doorbell inter-core interrupt mechanism
  - Polling through peripheral bus
- Allows advanced debugger features by allowing full cache control through external bus via dedicated channel
  - Automatic suspension of the block CME channel on entering debug mode or exception
  - Supports external query instructions
  - Physical address space invalidation allows easy breakpoint insertion
- DQUERY and PQUERY instructions support
  - Memory query functionality allows to read memory value as seen by the system (through L1 and L2 caches)

6.2.3 Considerations when entering processing states

Consider the CME as follows when entering different subsystem processing states:

- Debug Mode—when entering debug mode, the CME can automatically suspend all block cache channels when configured in DU. The debug channel remains active.

6.3 Functional description of the CME

6.3.1 DCache maintenance instruction generation overview

This figure shows the CME DCache maintenance instruction generation. The ICache maintenance instruction generation is very similar.

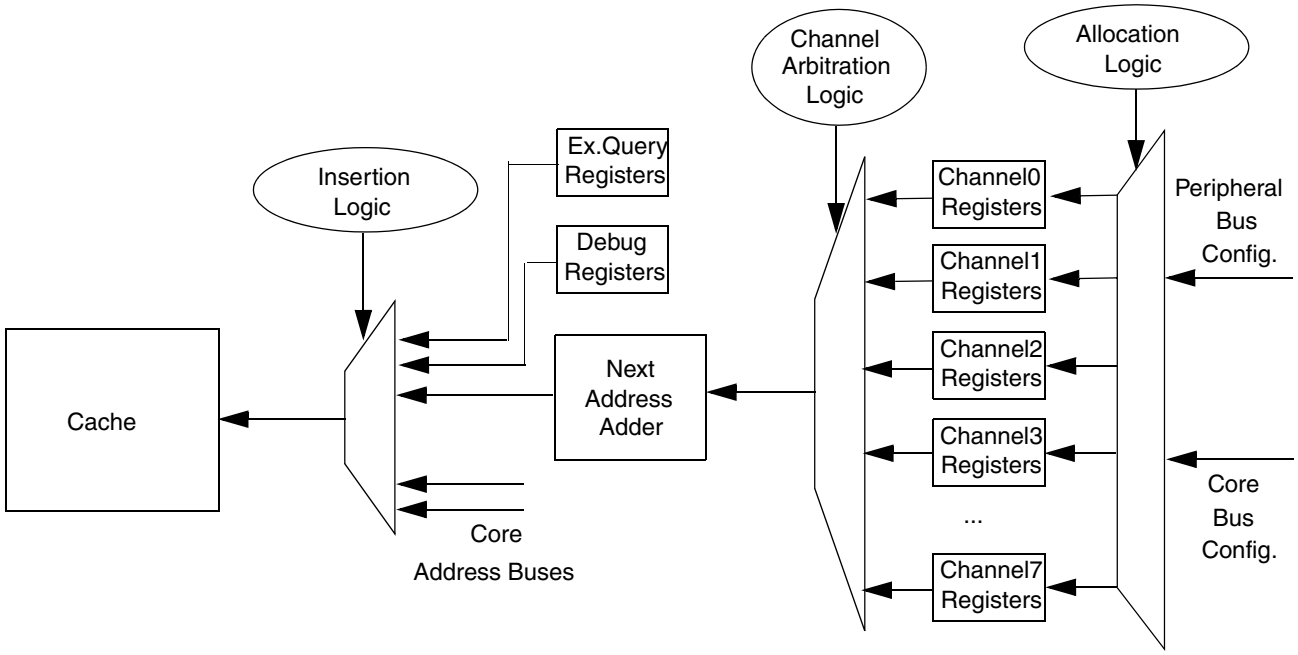

Figure 6-2. CME DCache maintenance instruction generation scheme

6.3.2 CME block programming

The CME can be programmed using dedicated SC3900 FVP cache instructions or through the peripheral bus (external configuration). The first method requires a simple execution of core instructions whenever an external configuration requires peripheral bus register updates.

Both methods follow the basic principles of dynamic allocation:

- If there is an empty channel, it will be used for a new cache maintenance instruction.
- If all channels are used by cache maintenance instructions, the new cache instruction fails.

The only exception to the above rules is when CME is in suspended or error state. In these states, allocation is permitted only into empty channels.

Programmability of the CME by core and external bus can be disabled by various bits in CME_CTRL register. Additional control is provided by the VCCC bit in the MMU M_CR register. It prevents FETCH cache commands to the caches. For more details see 3.3.1, “MMU Control Register (M_CR).”

In general, limiting access to the CME can be useful for debug or protection. The summary of the controls is below:

- The VCCC bit in the M_CR MMU register cancels the allocation of all FETCH cache commands in CME. Used to reduce bus load.
- The ECD bit in the CME_CTR CME register cancels the allocation of all external (from SkyBlue) cache commands in CME. Used to reserve CME for core cache commands only.

All of these bits cancel only CME allocation (programming attempts of the disabled type always fail CME insertion). If canceling is enabled during CME operation existing CME channels are not affected.

6.3.2.1 Core block cache maintenance instructions

The *SC3900 FVP Core Reference Manual* provides details about the core block cache maintenance instructions programming CME. Each block instruction uses a pair of core read-only registers to configure a channel in a read-like operation. The format of these registers is the same as that of external configuration registers CME_CS and CME_CA, as described in Section 6.5.13, “CME Block Stride Programming Register (CME_CS),” and in Section 6.5.14, “CME Block Address Programming Register (CME_CA).” About six stalls are added on each block configuration instruction, so small operations can be done more quickly using granular instructions. Maintenance instructions are checked by the MMU only upon exit from the CME; thus, the error is not precise.

Block cache maintenance instructions are load type instructions; the CME pushes respond information into the corresponding core register. See Section 6.5.15, “CME Block Programming Status Register (CME_CR),” for the format of the CME response to core block cache maintenance instructions.

6.3.2.2 External CME programming

External CME cache maintenance instruction programming is performed using three registers, as follows:

- CME_CC—contains the instruction’s controls
- CME_CS—contains the size and stride
- CME_CA—contains the start address

When CME_CA is written, all three registers are sampled and allocation proceeds. Allocation results are updated in the CME_CR register and can be read from there. All external cache maintenance instructions can fail to be allocated: it is possible to rewrite CME_CA value to have another try. Sometimes the allocation result is indicated as not ready; in this case the SME_CR register should be polled until the result is ready.

Often when configuring a sequence of cache instructions, the only difference between them is a start address. In this case, CME_CS and CME_CC are written once and only CME_CA is changed between instructions.

6.3.2.3 CME block channels organization and control

CME channels status is accessible through the external bus. Successful allocation by either way provides a channel index that can be used to access the channels status registers. Each channel has three read only registers in a format similar to external cache maintenance instruction programming interface. Data channel registers are CME_DA<n>, CME_DS<n>, CME_DC<n>, and instruction channel registers are CME_PA<n>, CME_PS<n> and CME_PC<n> with n from 0 to 7.

The channel status contains the current address and rows counter, allowing channel troubleshooting in case of an MMU error or for debug. CME_DC<n> and CME_PC<n> contain a STATE field that indicates the current state of the channel: empty or working. The whole CME can also be suspended by core instruction or by error.

The CME_DST and CME_PST registers contain aliases of all channels' STATE fields, and the status of all channels can be read in one operation. The register's SUSP field can be used to suspend data or program block channels.

The CME_DER and CME_PER registers contain information about the current CME error. After each error, the CME enters into a suspend on error state. To exit this state after the error's cause has been corrected (such as with MMU reprogramming), write a 0 into the D/PST field. This clears the error and resumes the channel. If the error is non-recoverable, write a 1 into the D/PST field. This clears the error and disables all the channels.

Channel reset and suspend differs in that a suspended channel can be forced to resume operation from the current point while a reset channels need to be reprogrammed. Error suspend differs from regular suspend only by the resume type operation: regular resume does not influence it.

This figure shows the channel state transition. In debug mode, all state transitions can be frozen if configured by the DU.

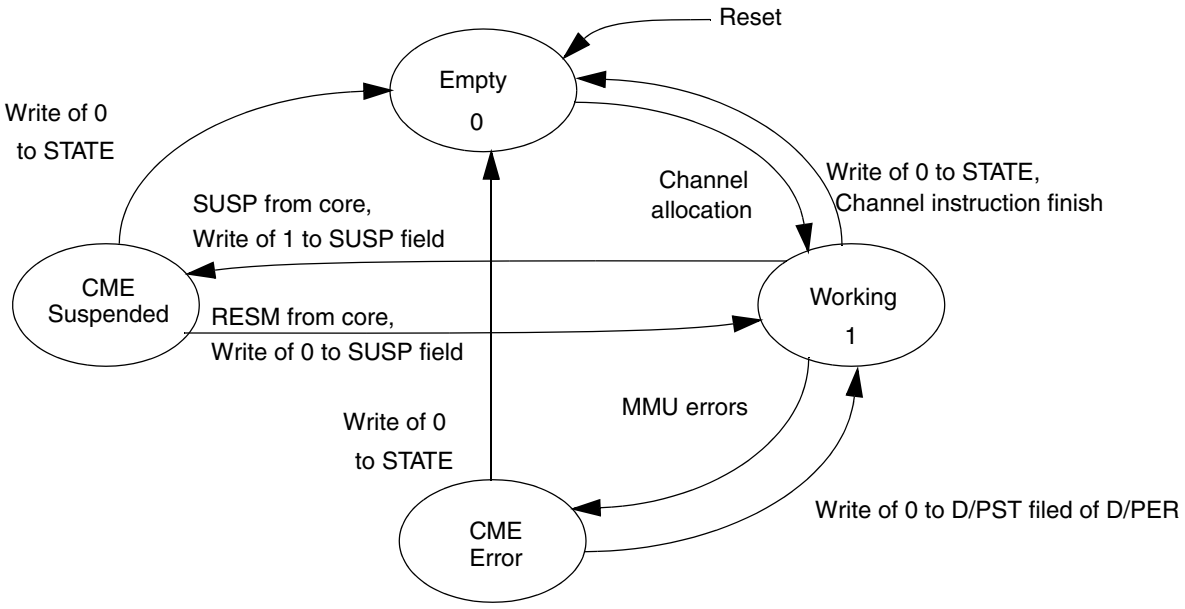

Figure 6-3. Channel state transition scheme

The core instructions SUSP and RESM have the same effect as a write to SUSP field of CME_D/PST register, forcing all program/data channels to suspend or resume respectively. See the *SC3900 FVP Core Reference Manual* for additional details about these instructions.

6.3.3 Query instructions

The core receives the query result into a pair of core read registers in a read-like operations DQUERY.L1/2 Rn,Ra:Rb. The format of Ra and Rb registers is the same as of CME registers CME_QU1 and CME_QU2 correspondingly. They are described in Section 6.5.10, "CME External Query Result Register 1 (CME_QU1)," and Section 6.5.11, "CME External Query Result Register 2 (CME_QU2)." See the *SC3900 FVP Core Reference Manual* for full details about core query instructions.

Core query instructions cannot fail or be canceled. Query instructions are stalled until the query result arrives.

6.3.4 Using the external query channel

The purpose of the external query channel is to allow external users to query the MMU. This channel also allows the performance of a memory query and program memory barriers that are not available in the core instruction set.

External query programming is done using two registers, as follows:

- CME_QCC—contains the query's controls

- CME_QCA—contains the query address

When CME_QCA is written, both registers are sampled and execution proceeds. Execution results are updated in the CME_QCR register and can be read from there. Sometimes the allocation result is indicated as not ready; in this case the SME_QCR register should be polled until the result is ready. After the execution has completed, the query result is sampled in the CME_QU1 and CME_QU2 registers.

Memory query (to read data using either program or data path) and program memory barriers are available only through peripheral bus programming. The L1 cache state is not changed by memory query, but the L2 cache treats it like a usual access.

6.3.5 Using the debug channel

The purpose of the dedicated debug channel is to allow the debugger to use the CME in the debug state and mode. This channel also allows the performance of a physical cache invalidation that is not available in the core instruction set. Only granular cache maintenance instruction are supported by the debug channel, as described in Section 6.5.2, “CME Debug Channel Control Programming Register (CME_DCC).” Access to debug channel registers is limited to debug masters on the peripheral bus.

Debug programming is done using two registers, as follows:

- CME_DCC—contains debug instruction controls
- CME_DCA—contains debug address

When CME_DCA is written, both registers are sampled and execution proceeds. Execution results are updated in the CME_DCR register and can be read from there. Sometimes the allocation result is indicated as not ready; in this case the SME_DCR register should be polled until the result is ready. Debug instructions can fail due to MMU error; in this case the DER field of CME_DCR register is updated. Debug channel also supports query commands and has dedicated registers to sample query result.

6.3.6 Understanding the arbitration logic

ICache and DCache instructions have separate arbitration schemes.

Cache maintenance instructions are ordered from high to low as follows:

- Debug channel instructions
- Query channel instructions
- Rest of maintenance instructions in their order of appearance (FIFO like)

If data or program cache maintenance instruction is suspended or receives an error, corresponding block channels are frozen. Users should resume CME execution by hand.

Caches provide the CME with the status of the external memory pipeline in order to prevent external memory bus saturation with low priority cache maintenance instructions. This prevents possible core performance loss due to a heavily loaded bus. When the bus is almost choked, only queries or debug can exit the CME.

6.3.7 Next address adder

Address of active block maintenance instruction is generated by the next address adder. It supports a two-dimensional model that is useful for block processing (for example, image processing frame).

Adder behavior is defined by the content of the channel registers as follows:

1. The accesses generator starts from the start address
2. The address is incremented in L1 cache line resolution (128 byte)
3. After incrementing the address according to the required size of row, the address is incremented to the next row according to the stride
4. The address is again incremented in coherency granule resolution according to the required row size.
5. When the address is incremented according to the required size of row and to the required number of rows, the process is done.

Figure 6-4 illustrates the frame mechanism parameters. The row size parameter defines the number of bytes fetched within a frame line. The stride parameter defines the stride between the start address of two sequential frame rows. The number of rows parameter defines how many frame rows are fetched. When simple linear address generation (single dimensional) is enabled, the stride is the same as the row size. For a two-dimensional model, the stride should be equal or greater than the row size.

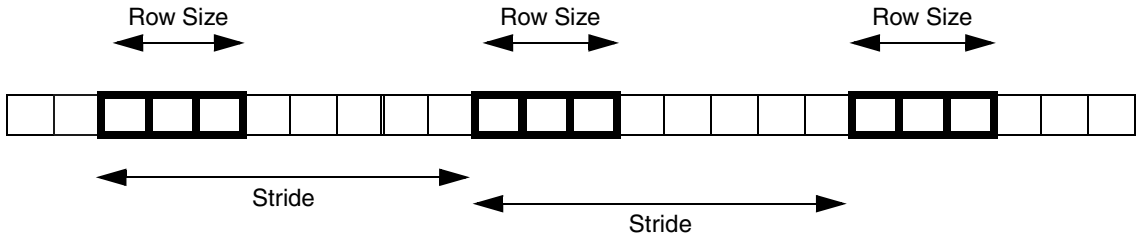

Figure 6-4. Two-dimensional address generation

6.3.8 Core bus insertion mechanism

All cache maintenance instructions generated by the CME are placed on the cache input buses that are shared with the core. In spite of the fact that the core has higher priority than CME, it is able to issue accesses on any cycle that is not used by the core for two reads and program fetch simultaneously. The implemented insertion algorithm does not interfere with the core and allows high performance. The CME can insert up to one ICache and one DCache maintenance instruction per cycle.

6.3.8.1 ICache maintenance instructions insertion

The SC3900 FVP instruction bus is not overloaded with core program fetching due to the optimizations made in the core prefetcher block. This allows the CME to insert instruction into the empty slots on the core instruction bus. The side effect is that sometimes the core is able to starve the CME, preventing it from accessing the ICache. Practically, such a situation cannot last for too long: to achieve full bus utilization the core should run maximum density code (each VLES has a size of 256 bits) without sequential loops or

to run non cacheable code. Such sequences are rare and short and when they are over, the CME is able to continue insertion.

6.3.8.2 DCache maintenance instructions insertion

The DCache has two data buses; the CME is able to place one maintenance instruction on either of them. There are two insertion algorithms for data instruction placement: empty slot insertion and write slot insertion. Insertion in the empty slots is performed in the same way as the ICache maintenance instructions, but some FVP kernels utilize data buses for a long time without any interruption. To work around this, the CME can insert DCache maintenance instructions during core write accesses to the memory. As there is no algorithm that continuously reads data without any write, CME is able to insert DCache instruction in any real circumstances.

Because the MMU needs to provide translation and protection for core writes, it is not available for CME accesses inserted instead of them. The CME solution is to cache physical address and MMU attributes for data channels. The CME provides stored physical address and attributes to the cache instead of the MMU.

The CME forces DCache maintenance instruction to use the empty bus slot for MMU translation in the following cases:

- First cache maintenance instruction issued after channel configuration
- First cache maintenance instruction issued after next address adder crossed 4 Kbyte boundary (minimal MMU segment size is 4 Kbyte and each such crossing can potentially hit another MMU segment)
- First cache maintenance instruction after resume from error

In all of these cases, there may be a slight delay in the data CME operation due to the wait for the empty slot on the data buses. Granular data instructions, like DQUERY, always use an empty bus slot because they need to access the MMU.

When memory attributes (coherency, cacheable, and so on) in the MMU descriptors or M_CR register are permanently changed, the CME should be reset using CME_CTR[CRB] to disable its caching mechanism.

6.3.9 Maintenance instruction completion events

Block cache maintenance instructions can take a long time to finish. Therefore, even when it is possible to poll of STATE flags, it is not always desirable. The CME provided solution is either an EPIC interrupt assertion to the core, a doorbell interrupt sent to another core through the MMU, or a memory barrier execution at the end of the cache maintenance instruction execution. All required fields are configured during cache maintenance instruction programming.

Because instructions are executed and finished in order, it is possible to configure a sequence of them with only the last having an interrupt or a message at finish. In this way it is possible, for example, to define multiblock (chained maintenance instructions) cache flush with a single completion interrupt at the end of the whole sequence instead of having an interrupt per each maintenance instruction, thus saving interrupt overhead. Also, a heavyweight sync memory barrier is automatically generated after finishing execution. An interrupt is asserted after barrier completion.

The CME_P/DIN and CME_P/DMS registers hold information about currently asserted EPIC and doorbell interrupts, respectively. Multiple channels can finish together waiting for resolution. The channel that asserted an interrupt has its corresponding bit asserted in the register. Writing 1 to these bits clears the completion indication and should be performed at the end of relevant interrupt routines.

6.3.10 Understanding MMU Errors

CME-issued cache maintenance instructions can result in an MMU error because of protection violation, a segment miss, a multiple segment descriptor hit, or noncacheable area hit. In all of these cases, an interrupt is asserted to the EPIC; the CME state is changed; and error information is sampled in the corresponding CME_ER register. Errors change the CME channels state to suspend by error; after an MMU programming fix, the channel can be resumed from the error point by using a write of 0 to the CME_P/DER STA field (error suspend is of a “break before make” type similar to the core’s precise exception). See Figure 6-3 for an illustration of the channel state transition. Cache maintenance instructions errors can be disabled using dedicated MMU bits.

CME_P/DER read only registers contains error type of the current error. The interrupt routine should check it to find the problematic channel. The channel’s current address from the CME_DA<n> or CME_PA<n> register indicates the source of the problem.

6.4 Detailed block cache maintenance instructions description

Cache instructions can be divided into three main groups: software prefetch, L2 cache lock management, and coherency accelerator instructions.

Block cache maintenance instruction is a way to run granular cache instructions on the address range. Each block instruction generates a sequence of granular maintenance instructions with the address sequence generated by next address adder. All generated instructions work on a 128 byte size, which is also the size of L1 line size.

6.4.1 Block software prefetch maintenance instructions

Cache software prefetch is a mechanism designed to fill the cache with upper level memory data before it is needed to increase the cache hit rate. This mechanism enables the prefetch of a specific address space as programmed in the cache maintenance instruction. Two-dimensional prefetch is a very useful tool to bring complex data structures into the cache.

Prefetch can carry an instruction or data to the L1 or to L2 cache (L1 prefetch updates L2 as well because of cache inclusiveness). Instruction and data L2 cache prefetches differ even if the L2 cache is shared: it obtains a different MMU translation, different attributes, and different coherency treatment.

Data prefetch can be optimized for the following data usage patterns:

- For a read-only sequence of data accesses, DFETCH should be used.
- For a read-write-read sequence of data accesses, DFETCH with intend to write should be used.

L2 prefetch instructions have a variant with L2 locking. L2 lock maintenance instructions are described in Section 6.4.2, “L2 cache lock management maintenance instructions.”

6.4.2 L2 cache lock management maintenance instructions

The L2 cache can be locked on a per two lines base (per 128 byte). L2 prefetch maintenance instructions have a variant with L2 lock in which the prefetch is locked and will not be thrashed until unlock. This feature is useful for frequently used data that should not be thrashed by big and rarely used data. Because lock is actually a destructive instruction that can reduce the active L2 cache size if done improperly, it is limited to blocking prefetch maintenance instructions only as they cannot be speculative. The SLD bit in the CME_CTR register allows disabling the L2 locking effect.

DUNLOCKB and PUNLOCKB are used to unlock L2 cache lines previously locked by prefetch. These instructions only work on the L2 cache. They differ in terms of MMU translation and attributes. DUNLOCKB is translated using data MMU descriptors and PUNLOCKB uses program MMU).

6.4.3 Coherency accelerator block instructions

Cache coherency accelerator instructions are a mechanism designed to support coherency accelerator algorithms. This mechanism should be used mainly for memory areas with disabled hardware coherency, for coherency acceleration, or for debug. This mechanism enables invalidation and/or synchronization of a specific address space as defined in the maintenance instruction. Synchronization is needed if an external module reads data that is modified in the cache array from the system level memory. Invalidation is useful when data or an instruction that is in the cache array is not used. Invalidation is required if the data in the system level memory is more recently updated than the one in the cache array. Flush is a combination of synchronization and invalidation, and is needed if data that is modified in the cache array is read from system level memory and is not used by the core.

While using coherency accelerator operations, it is recommended that situations be prevented in which the core accesses the memory space on which a flush operation is executed. If such a situation does occur, however, the cache maintenance instruction or core access may override each other. The later event overrides an earlier one.

L1 synchronization only flushes SGB content to the L2 cache. L2 synchronization synchronizes the L2 cache with memory. However, the L2 synchronization does not clear the SGB. If it needs to be cleared, perform an L1 synchronization first.

The L1 invalidation of a coherency granule is executed by clearing the L1 cache line valid bit. L12 flavors invalidate L2 in addition to L1. However, the invalidation does not clear the SGB. If it needs to be cleared, perform an L1 synchronization first.

The L12 flush thrashes the L2 data and clears its dirty and valid bits. However, the L12 flush does not clear the SGB. If it needs to be cleared, perform an L1 synchronization first. Due to L2 cache inclusiveness L12 flush also invalidates all L1 caches in the cluster.

L1 synchronization (SGB flush) is a relatively costly procedure. It is not address dependant, so it is not included in all maintenance instructions option. However, it can be chained to them if needed by programming an L1 synchronization first. In many cases, a sequence of coherency instructions is executed, and there is no reason to flush the SGB more than one time at the start of the sequence.

Coherency accelerator instructions performed by the debug channel can be defined to work on the physical address space. In this case, the MMU translation is bypassed altogether and the cache invalidates the provided physical address.

This table summarizes the effect of the coherency accelerator instructions on the caches.

Table 6-2. Block coherency accelerator summary

Maintenance instruction	Cache level	SGB action	L1 DCache action	L2 Cache action*	L1 ICache action
DSYNC	L1	Flush	No	No	No
	L2	No	No	Thrash	No
DFLUSH	L12	No	Invalidate	Thrash and invalidate	No
DINVALID	L1	No	Invalidate	No	No
	L12	No	Invalidate	Invalidate	No
PINVALID	L1	No	No	No	Invalidate
	L12	No	No	Invalidate	Invalidate

Note: Some L2 cache implementation may choose to always perform thrash and invalidate operation even if only thrash or only invalidate were requested.

6.5 Memory map and register definitions

All CME registers are memory-mapped and can be written or read by the core or external master through the peripheral bus. Reserved or unused bits in all registers should be written as zeros, and the read value should be masked. Writing to unimplemented or read-only registers generates a peripheral bus error. Reading from unimplemented registers generates a peripheral bus error and undefined data.

The memory offset of the CME registers is defined in Table 6-3. This offset should be added to the base address of the SC3900 subsystem peripheral bus to obtain an actual address. Note that the register access width (long, 2long) is restricted to that shown in the following table and must be aligned to the address.

Table 6-3. CME memory map

Offset	Use	Access width	Section/page
6000	CME Control Register (CME_CTR)	long, 2long	6.5.1/6-19
6004-6007	Reserved		
6008	CME Debug Channel Control Programming Register (CME_DCC)	long, 2long	6.5.2/6-20
600C	CME Debug Channel Address Programming Register (CME_DCA)	long, 2long	6.5.3/6-21
6010	CME Debug Status Register (CME_DCR)	long, 2long	6.5.4/6-22
6014	CME External Query Status Register (CME_QCR)	long, 2long	6.5.7/6-25
6018	CME Debug Query Result Register 1 (CME_DQU1)	long, 2long	6.5.5/6-23
601C	CME Debug Query Result Register 2(CME_DQU2)	long, 2long	6.5.6/6-23
6020	CME External Query Control Programming Register (CME_QCC)	long, 2long	6.5.8/6-26

Table 6-3. CME memory map (continued)

Offset	Use	Access width	Section/page
6024	CME External Query Address Programming Register (CME_QCA)	long, 2long	6.5.9/6-27
6028	CME External Query Result Register 1 (CME_QU1)	long, 2long	6.5.10/6-27
602C	CME External Query Result Register 2 (CME_QU2)	long, 2long	6.5.11/6-28
6030	CME Block Control Programming Register (CME_CC)	long, 2long	6.5.12/6-28
6034	CME Block Stride Programming Register (CME_CS)	long, 2long	6.5.13/6-29
6038	CME Block Address Programming Register (CME_CA)	long, 2long	6.5.14/6-31
603C	CME Block Programming Status Register (CME_CR)	long, 2long	6.5.15/6-32
6040	CME Data Status Register (CME_DST)	long, 2long	6.5.16/6-33
6044	CME Data Error Register (CME_DER)	long, 2long	6.5.17/6-33
6048	CME Data Interrupt Status Register (CME_DIN)	long, 2long	6.5.18/6-34
604C	CME Data External Doorbell Interrupt Status Register (CME_DMS)	long, 2long	6.5.19/6-35
6050	CME Data Channel Control 0 Register (CME_DC0)	long, 2long	6.5.20/6-35
6054	CME Data Channel Stride 0 Register (CME_DS0)	long, 2long	6.5.21/6-36
6058	CME Data Channel Address 0 Register (CME_DA0)	long	6.5.22/6-37
605C-605F	Reserved		
6060	CME Data Channel Control 1 Register (CME_DC1)	long, 2long	6.5.20/6-35
6064	CME Data Channel Stride 1 Register (CME_DS1)	long, 2long	6.5.21/6-36
6068	CME Data Channel Address 1 Register (CME_DA1)	long	6.5.22/6-37
606C-606F	Reserved		
6070	CME Data Channel Control 2 Register (CME_DC2)	long, 2long	6.5.20/6-35
6074	CME Data Channel Stride 2 Register (CME_DS2)	long, 2long	6.5.21/6-36
6078	CME Data Channel Address 2 Register (CME_DA2)	long	6.5.22/6-37
607C-607F	Reserved		
6080	CME Data Channel Control 3 Register (CME_DC3)	long, 2long	6.5.20/6-35
6084	CME Data Channel Stride 3 Register (CME_DS3)	long, 2long	6.5.21/6-36
6088	CME Data Channel Address 3 Register (CME_DA3)	long	6.5.22/6-37
608C-608F	Reserved		
6090	CME Data Channel Control 4 Register (CME_DC4)	long, 2long	6.5.20/6-35
6094	CME Data Channel Stride 4 Register (CME_DS4)	long, 2long	6.5.21/6-36
6098	CME Data Channel Address 4 Register (CME_DA4)	long	6.5.22/6-37
609C-609F	Reserved		
60A0	CME Data Channel Control 5 Register (CME_DC5)	long, 2long	6.5.20/6-35
60A4	CME Data Channel Stride 5 Register (CME_DS5)	long, 2long	6.5.21/6-36

Table 6-3. CME memory map (continued)

Offset	Use	Access width	Section/page
60A8	CME Data Channel Address 5 Register (CME_DA5)	long	6.5.22/6-37
60AC-60AF	Reserved		
60B0	CME Data Channel Control 6 Register (CME_DC6)	long, 2long	6.5.20/6-35
60B4	CME Data Channel Stride 6 Register (CME_DS6)	long, 2long	6.5.21/6-36
60B8	CME Data Channel Address 6 Register (CME_DA6)	long	6.5.22/6-37
60BC-60BF	Reserved		
60C0	CME Data Channel Control 7 Register (CME_DC7)	long, 2long	6.5.20/6-35
60C4	CME Data Channel Stride 7 Register (CME_DS7)	long, 2long	6.5.21/6-36
60C8	CME Data Channel Address 7 Register (CME_DA7)	long	6.5.22/6-37
60CC-60FF	Reserved		
6100	CME Program Status Register (CME_PST)	long, 2long	6.5.23/6-38
6104	CME Program Error Register (CME_PER)	long, 2long	6.5.24/6-39
6108	CME Program Interrupt Status Register (CME_PIN)	long, 2long	6.5.25/6-40
610C	CME Program External Doorbell Interrupt Status Register (CME_DMS)	long, 2long	6.5.26/6-41
6110	CME Program Channel Control 0 Register (CME_PC0)	long, 2long	6.5.27/6-41
6114	CME Program Channel Stride 0 Register (CME_PS0)	long, 2long	6.5.28/6-42
6118	CME Program Channel Address 0 Register (CME_PA0)	long	6.5.29/6-43
611C-611F	Reserved		
6120	CME Program Channel Control 1 Register (CME_PC1)	long, 2long	6.5.27/6-41
6124	CME Program Channel Stride 1 Register (CME_PS1)	long, 2long	6.5.28/6-42
6128	CME Program Channel Address 1 Register (CME_PA1)	long	6.5.29/6-43
612C-612F	Reserved		
6130	CME Program Channel Control 2 Register (CME_PC2)	long, 2long	6.5.27/6-41
6134	CME Program Channel Stride 2 Register (CME_PS2)	long, 2long	6.5.28/6-42
6138	CME Program Channel Address 2 Register (CME_PA2)	long	6.5.29/6-43
613C-613F	Reserved		
6140	CME Program Channel Control 3 Register (CME_PC3)	long, 2long	6.5.27/6-41
6144	CME Program Channel Stride 3 Register (CME_PS3)	long, 2long	6.5.28/6-42
6148	CME Program Channel Address 3 Register (CME_PA3)	long	6.5.29/6-43
614C-614F	Reserved		
6150	CME Program Channel Control 4 Register (CME_PC4)	long, 2long	6.5.27/6-41
6154	CME Program Channel Stride 4 Register (CME_PS4)	long, 2long	6.5.28/6-42
6158	CME Program Channel Address 4 Register (CME_PA4)	long	6.5.29/6-43

Table 6-3. CME memory map (continued)

Offset	Use	Access width	Section/page
615C-615F	Reserved		
6160	CME Program Channel Control 5 Register (CME_PC5)	long, 2long	6.5.27/6-41
6164	CME Program Channel Stride 5 Register (CME_PS5)	long, 2long	6.5.28/6-42
6168	CME Program Channel Address 5 Register (CME_PA5)	long	6.5.29/6-43
616C-616F	Reserved		
6170	CME Program Channel Control 6 Register (CME_PC6)	long, 2long	6.5.27/6-41
6174	CME Program Channel Stride 6 Register (CME_PS6)	long, 2long	6.5.28/6-42
6178	CME Program Channel Address 6 Register (CME_PA6)	long	6.5.29/6-43
617C-617F	Reserved		
6180	CME Program Channel Control 7 Register (CME_PC7)	long, 2long	6.5.27/6-41
6184	CME Program Channel Stride 7 Register (CME_PS7)	long, 2long	6.5.28/6-42
6188	CME Program Channel Address 7 Register (CME_PA7)	long	6.5.29/6-43
618C-63FF	Reserved		

6.5.1 CME Control Register (CME_CTR)

The CME_CTR register, shown in Figure 6-5, contains the CME global disable bits.

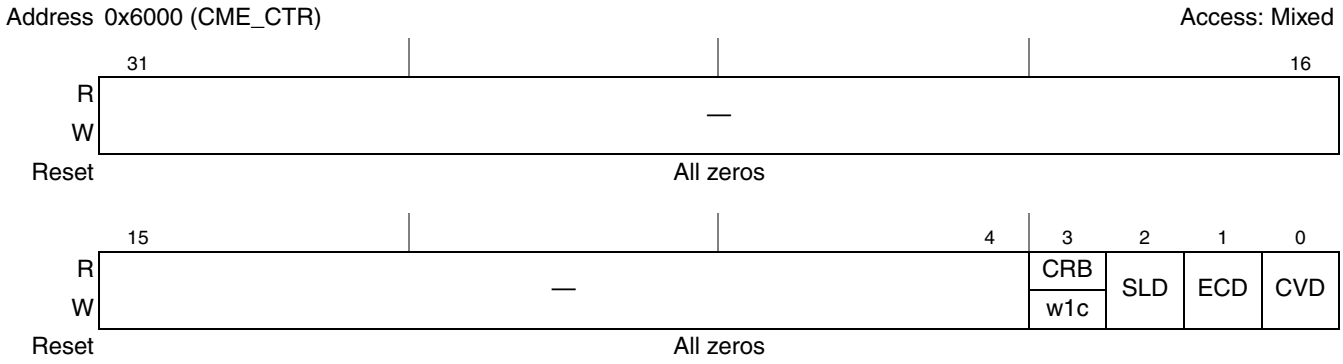

Figure 6-5. CME Control Register (CME_CTR)

This table describes the bit fields.

Table 6-4. Register CME_CTR bits description

Field	Description
31–4	Reserved
3 CRB	CRB bit is used to reset all CME channels at once. Write 1 to execute the reset. After reset CME can be programed as usual.

Table 6-4. Register CME_CTR bits description (continued)

Field	Description
2 SLD	Disables L2 locking effect in block fetch instructions. 0 Locking maintenance instructions are allowed. 1 All software prefetch with lock maintenance instructions exit to L2 cache without locking attribute.
1 ECD	Disables all block maintenance instructions from the peripheral bus. 0 External maintenance instructions are accepted and allocated. 1 External maintenance instructions are not allocated.
0 CVD	Disables all core fetch cache instructions. 0 Core fetch maintenance instructions are accepted and allocated. 1 Core fetch maintenance instructions are not allocated.

6.5.2 CME Debug Channel Control Programming Register (CME_DCC)

The CME_DCC register, shown in Figure 6-6, allows CME debug channel programming. Access to this register is limited to the DCSR accesses. Should be configured only if CME_DCR[DSTA] is not busy.

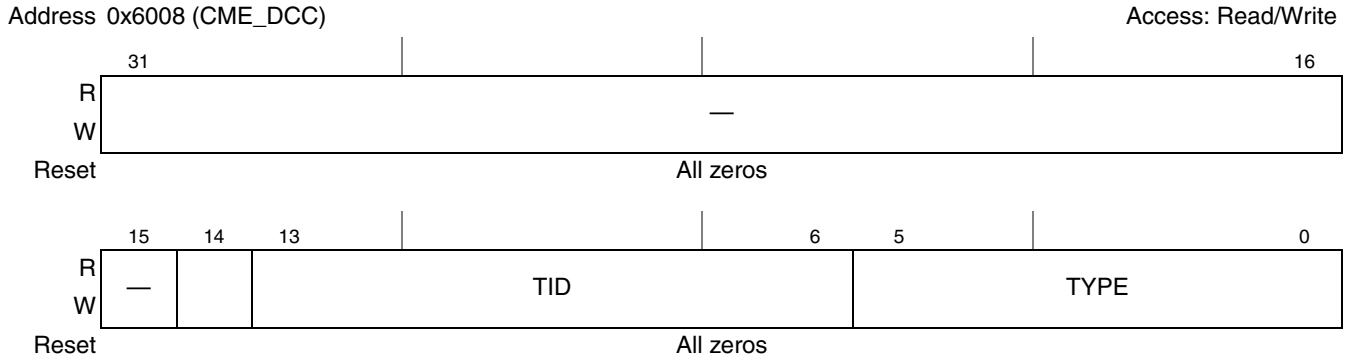

Figure 6-6. CME Debug Channel Control Programming Register (CME_DCC)

This table describes the bit fields.

Table 6-5. Register CME_DCC bits description

Field	Description
31–15	Reserved

Table 6-5. Register CME_DCC bits description

Field	Description
13–6 TID	Task ID or physical address extension of the cache debug instruction. If virtual addressing is used this field is mapped to PID or DID depending on the cache maintenance instruction type. If physical addressing is used low 4 bits of this field provide physical address bits [35:32] and upper 4 bits are ignored.
5–0 TYPE	Type of cache maintenance instruction. 010000: DQUERY.L1 010001: DQUERY.L12 010010: DQUERY.MEM 000111: DUNLOCK.L2 001000: DCM.SYNC.L1 (clears whole SGB) 001101: DCM.INVAL.L12 011110: DCM.INVAL.L12.PHY - uses physical addressing 011111: DBAR.HWSYNC (does not clear SGB; DCM.SYNC.L1 should be run if SGB clear is needed) 110000: PQUERY.L1 110001: PQUERY.L12 110010: PQUERY.MEM 100111: PUNLOCK.L2 101101: PCM.INVAL.L12 111110: PCM.INVAL.L12.PHY - uses physical addressing 111111: PBAR.HWSYNC

6.5.3 CME Debug Channel Address Programming Register (CME_DCA)

The CME_DCA register, shown in Figure 6-7, allows CME debug channel programming. Access to this register is limited to the DCSR accesses. When it is written, the contents of the CME_DCA and CME_DCC registers are sampled and used to configure the debug CME channel. Should be configured only if CME_DCR[DSTA] is not busy.

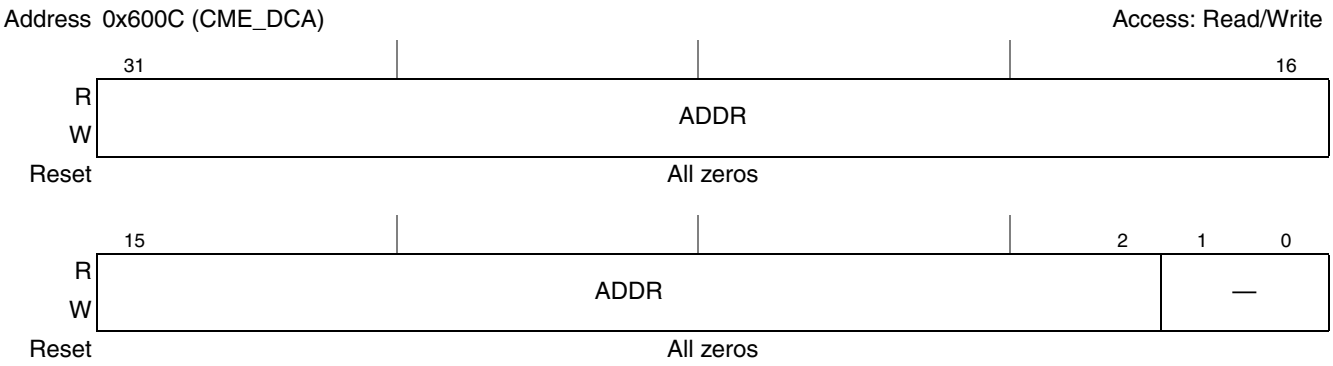

Figure 6-7. CME Debug Channel Address Programming Register (CME_DCA)

This table describes the bit fields.

Table 6-6. Register CME_DCA bits description

Field	Description
31–2 ADDR	Bits [31:2] of cache debug instruction address. Address bits [1:0] are always 0 (it has 4-byte resolution). The address is virtual for all instructions except for DCM.INVAL.L12.PHY and PCM.INVAL.L12.PHY, for which it is physical. It is not used by xBAR instructions.
1–0	Reserved

6.5.4 CME Debug Status Register (CME_DCR)

The CME_DCR register, shown in Figure 6-8, shows the result of CME debug channel. Access to this register is limited to the DCSR accesses. It is updated after a write to the CME_DCA register.

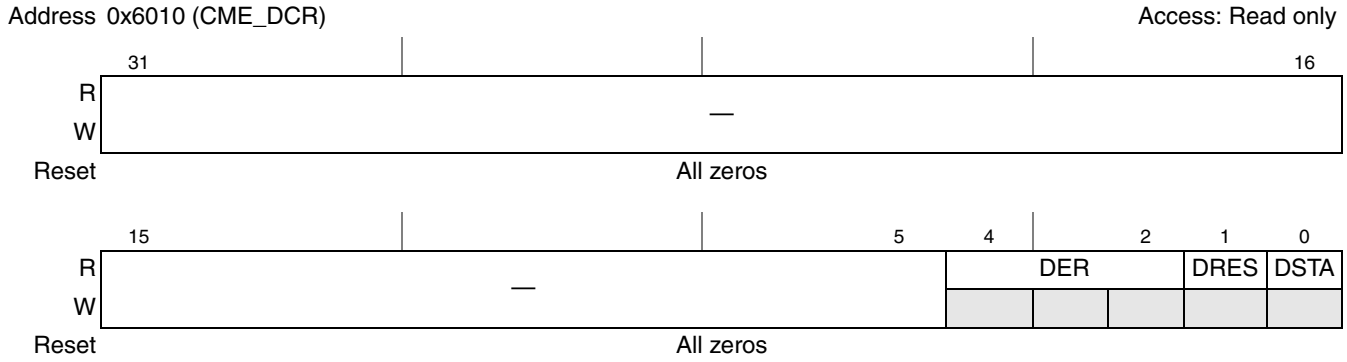

Figure 6-8. CME Debug Status Register (CME_DCR)

Table 6-7 describes the bit fields.

Table 6-7. Register CME_DCR bits description

Field	Description
31–5	Reserved
4–2 DER	Debug error type. Valid only if DSTA is 0 and DRES is 1. 000 Debug instruction has multiple segment descriptor hit error reported by MMU 010 Debug instruction has protection violation error reported by MMU 001 Debug instruction has segment miss error reported by MMU. 110 Debug instruction failed insertion because it is invalid (wrong TYPE) 111 Debug instruction has been programmed to non cacheable area
1 DRES	CME debug error. Valid only if DSTA indicates status done (is 0). 0 Success—debug instruction was executed without error. Query result is ready in CME_DQU registers. 1 Error—debug instruction execution failed due to MMU error; error is available in DER field. Query instruction never fails.
0 DSTA	CME debug status. 0 Done—debug instruction was executed and it's result is updated in DRES field and error is available in DER field. 1 Busy—debug instruction is executing.

6.5.5 CME Debug Query Result Register 1 (CME_DQU1)

The CME_DQU1 register, shown in Figure 6-9, provides the first part of a debug query result. Access to this register is limited to the DCSR accesses. It is updated after the execution of a debug query command.

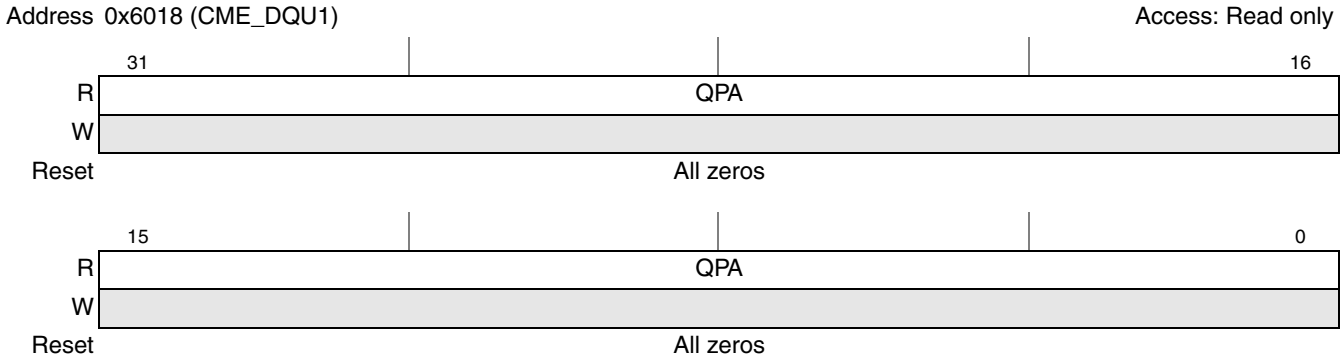

Figure 6-9. CME Debug Query Result Register 1 (CME_DQU1)

Table 6-8 describes the bit fields.

Table 6-8. Register CME_DQU1 bits description

Field	Description
31–0 QPA	For L1 and L2 query: Query Physical Address. Holds physical address [31:0] related to the last query virtual address. Not valid in case of error. For memory query: Memory query data. Holds 32 bit of data read from the query address aligned to 32 bit. Not valid in case of error. Does not contain data that is currently in SGB.

6.5.6 CME Debug Query Result Register 2 (CME_DQU2)

The CME_DQU2 register, shown in Figure 6-10, provides the second part of a debug query result. Access to this register is limited to the DCSR accesses. It is updated after the execution of a debug query command.

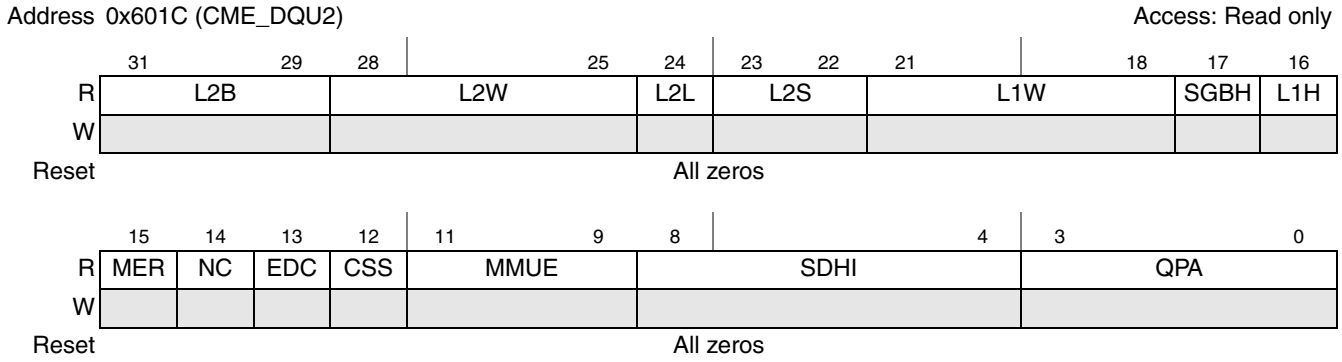

Figure 6-10. CME Debug Query Result Register 2 (CME_DQU2)

Table 6-9 describes the bit fields.

Table 6-9. Register CME_DQU2 bits description

Field	Description
31–29 L2B	L2 cache bank Reflects the L2 cache bank for the related query address. Only valid for the L2 query and cacheable address. Don't care in case of error.
28–25 L2W	L2 cache way Reflects the L2 cache way for the related query address. Only valid for the L2 query and cacheable address. Don't care in case of error.
24 L2L	L2 lock status Reflects the L2 cache lock status for the related query address. Only valid for the L2 query and cacheable address. Don't care in case of error.
23–22 L2S	L2 cache coherency status. Reflects the L2 cache hit and coherency status for the related query address. Value updated only for L2 query and cacheable address. Don't care in case of error. 00 Invalid 01 Shared (includes Shared-Last Reader) 10 Exclusive 11 Modified
21–18 L1W	L1 cache way Reflects the L1 cache way for the related query address. Don't care in case of error or non cacheable address.
17 SGBH	SGB hit indication Reflects SGB hit indication for the related query address. Not valid for PQUERY, memory queries, in case of error or L12 query to non cacheable address. Core query instruction has size of 1 byte for checking SGB match. CME initiated queries (from query and debug channels) has size of 4 byte. 0 No hit in SGB 1 Hit in SGB
16 L1H	L1 cache hit indication Reflects the L1 cache hit indication for the related query address. Don't care in case of error or L2 query and non cacheable address. 0 No hit in L1 cache 1 Hit in L1 cache
15 MER	L1 external memory error Valid only for memory query. Don't care in case of MMU error or L2 query and non cacheable address. 0 No external memory error in L1. 1 External memory error was detected.
14 NC	Non cacheable descriptor hit. Don't care in case of MMU error, except for peripheral error. 0 Cacheable descriptor hit. 1 Non cacheable descriptor hit. If L1 hit is also asserted then non cacheable hit error was detected.
13 EDC	L1 EDC error status Valid only for memory query. Don't care in case of MMU error. 0 No EDC error in L1. 1 L1 EDC error was detected. Returned data is not corrected.

Table 6-9. Register CME_DQU2 bits description (continued)

Field	Description
12 CSS	Bank 0 indication Set when a query address matches bank 0 address space. 0 for PQUERY. Don't care in case of error, except for peripheral error. 0 External access that goes through the L2 cache. 1 Bank 0 access. L2 cache statistics above are meaningless. For QUERY.L12 peripheral error is generated.
11–9 MMUE	MMU status. 000 Query has multiple segment descriptor hit error reported by MMU. 001 Query has segment miss error reported by MMU. 011 Query has peripheral error reported by MMU. DQUERY.L12 accesses bank 0 descriptor. 100 Query detected segment miss when protection is disabled. 111 Query has no errors reported by MMU.
8–4 SDHI	Segment descriptor hit index Reflects the segment descriptor hit index for the related query address. Don't care in case of error (except for peripheral error) and when detected segment miss in MMUE. When MSD is set, this field reflects the higher hit segment descriptor index.
3–0 QPA	Query physical address Holds full physical address [35–32] related to the last query virtual address. Don't care in case of error or memory query.

6.5.7 CME External Query Status Register (CME_QCR)

The CME_QCR register, shown in Figure 6-11, shows the result of CME external query. It is updated after a write to the CME_QCA register.

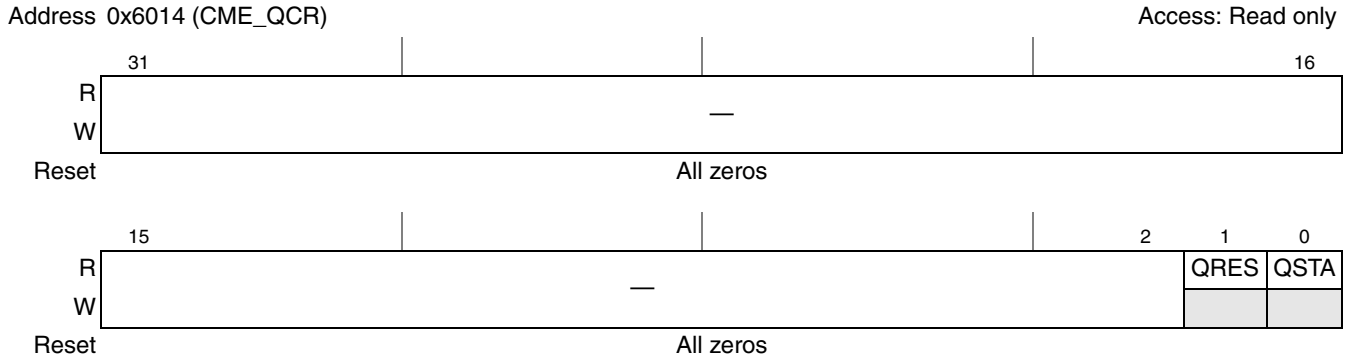

Figure 6-11. CME External Query Status Register (CME_QCR)

Table 6-10 describes the bit fields.

Table 6-10. Register CME_QCR Bits Description

Field	Description
31–2	Reserved

Table 6-10. Register CME_QCR Bits Description (continued)

Field	Description
1 QRES	CME block programming result. Valid if QSTA is 0. 0 External query or barrier instruction was successfully executed and query result is updated in CME_QU registers. 1 External query instruction failed because it had the wrong configuration.
0 QSTA	CME external query status. 0 Done—external query or barrier was executed and query's result is updated in QRES field. 1 Busy—external query or barrier is executing

6.5.8 CME External Query Control Programming Register (CME_QCC)

The CME_QCC register, shown in Figure 6-12, allows CME external query channel programming. Should be configured only if CME_QCR[QSTA] is not busy.

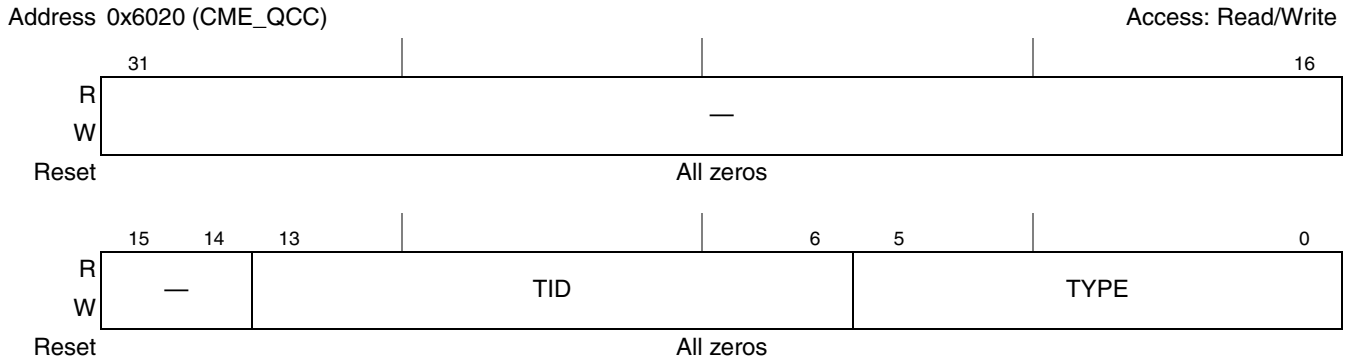

Figure 6-12. CME External Query Control Programming Register (CME_QCC)

Table 6-11 describes the bit fields.

Table 6-11. Register CME_QCC bits description

Field	Description
31–14	Reserved
13–6 TID	Task ID of the query instruction.
5–0 TYPE	Type of cache maintenance instruction. 010000: DQUERY.L1 010001: DQUERY.L12 010010: DQUERY.MEM 110000: PQUERY.L1 110001: PQUERY.L12 110010: PQUERY.MEM Memory barriers for ICache; do not use Task ID or address: 111100: PQSYNC - global ICache invalidation 111101: PBAR.L1SYNC - clearance of ICache internal FIFOs 111111: PBAR.HWSYNC - HWSYNC through ICache bus

6.5.9 CME External Query Address Programming Register (CME_QCA)

The CME_QCA register, shown in Figure 6-13, allows CME external query programming. When it is written, the contents of the CME_QCA and CME_QCC registers are sampled and used to configure external query CME channel. Should be configured only if CME_QCR[QSTA] is not busy.

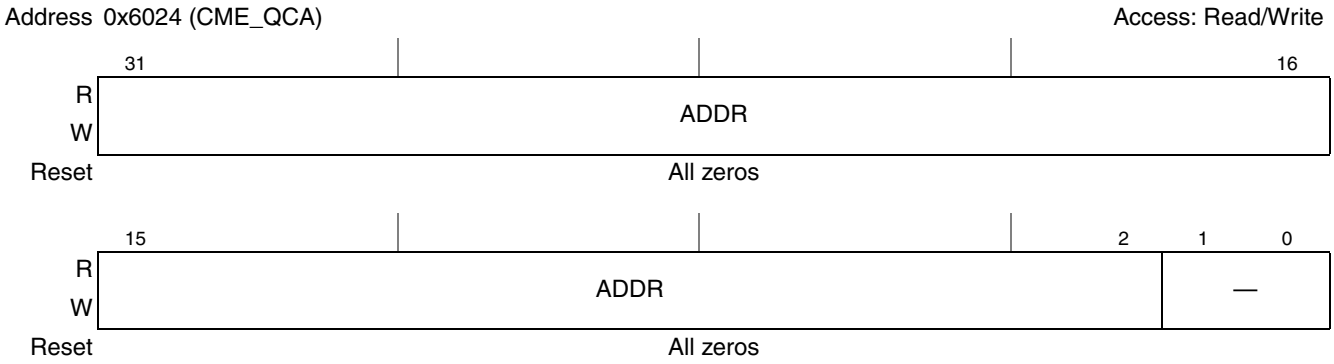

Figure 6-13. CME External Query Address Programming Register (CME_QCA)

Table 6-12 describes the bit fields.

Table 6-12. Register CME_QCA bits description

Field	Description
31–2 ADDR	Bits [31:2] of external query virtual address. Address bits [1:0] are always 0 (it has 4-byte resolution).
1–0	Reserved

6.5.10 CME External Query Result Register 1 (CME_QU1)

The CME_QU1 register, shown in Figure 6-14, provides the first part of an external query result. It is updated after the execution of an external query command.

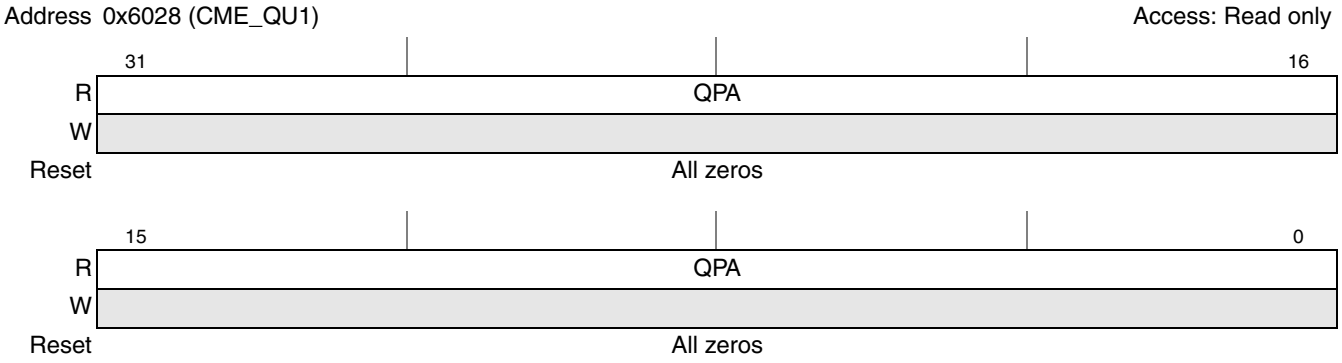

Figure 6-14. CME Query Result Register 1 (CME_QU1)

Table 6-8 describes the bit fields.

6.5.11 CME External Query Result Register 2 (CME_QU2)

The CME_QU2 register, shown in Figure 6-15, provides the second part of a external query result. It is updated after the execution of an external query command.

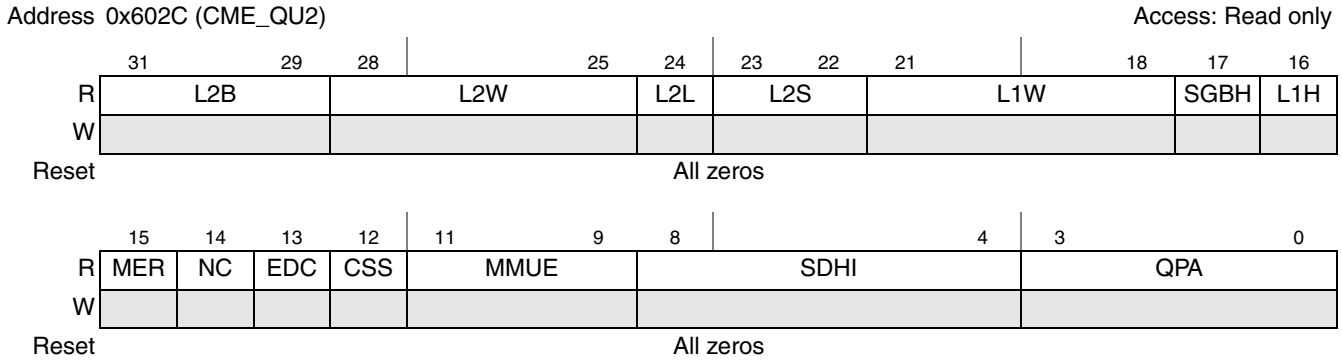

Figure 6-15. CME External Query Result Register 2 (CME_QU2)

Table 6-9 describes the bit fields.

6.5.12 CME Block Control Programming Register (CME_CC)

The CME_CC register, shown in Figure 6-16, allows CME block programming. Should be configured only if CME_CR[BSTA] is not busy.

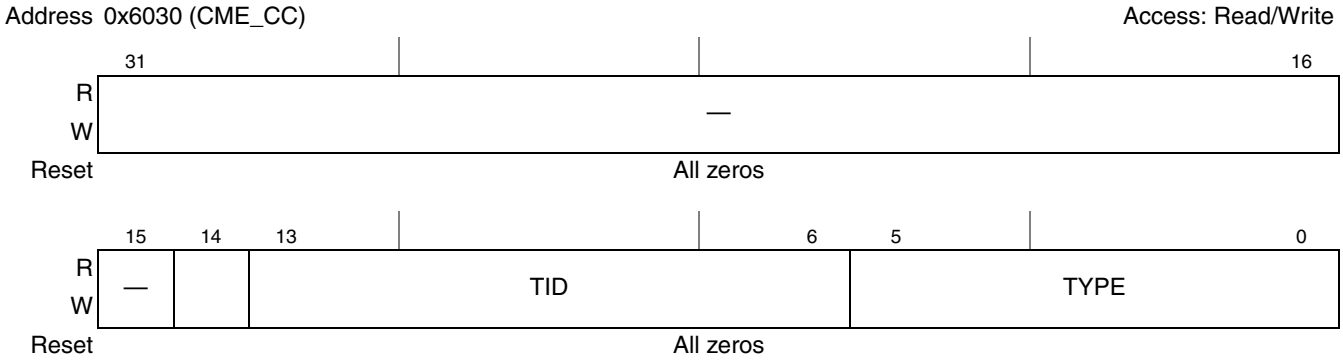

Figure 6-16. CME Block Control Programming Register (CME_CC)

Table 6-13 describes the bit fields.

Table 6-13. Register CME_CC bits description

Field	Description
31–16	Reserved

Table 6-13. Register CME_CC bits description (continued)

Field	Description
13–6 TID	Task ID of the cache maintenance instruction. This field is mapped to PID or DID depending on the cache maintenance instruction type.
5–0 TYPE	Type of cache maintenance instruction. 000000: DFETCHB.L12 000001: DFETCHB.L2 000010: DFETCHB.LCK.L2 000011: DFETCHB.L12.ITW 000100: DFETCHB.L2.ITW 000101: DFETCHB.LCK.L2.ITW 000111: DUNLOCKB.L2 001000: DCMB.SYNC.L1 001011: DCMB.FLUSH.L12 001100: DCMB.INVAL.L1 001101: DCMB.INVAL.L12 100000: PFETCHB.L12 100001: PFETCHB.L2 100010: PFETCHB.LCK.L2 100111: PUNLOCKB.L2 101100: PCMB.INVAL.L1 101101: PCMB.INVAL.L12

6.5.13 CME Block Stride Programming Register (CME_CS)

The CME_CS register, shown in Figure 6-17, allows block CME programming. This register presentation is dependant on the address generation mode (sequential or two dimensional) as defined by TDAG field. Different versions are presented in Figure 6-18 and Figure 6-19. Should be configured only if CME_CR[BSTA] is not busy.

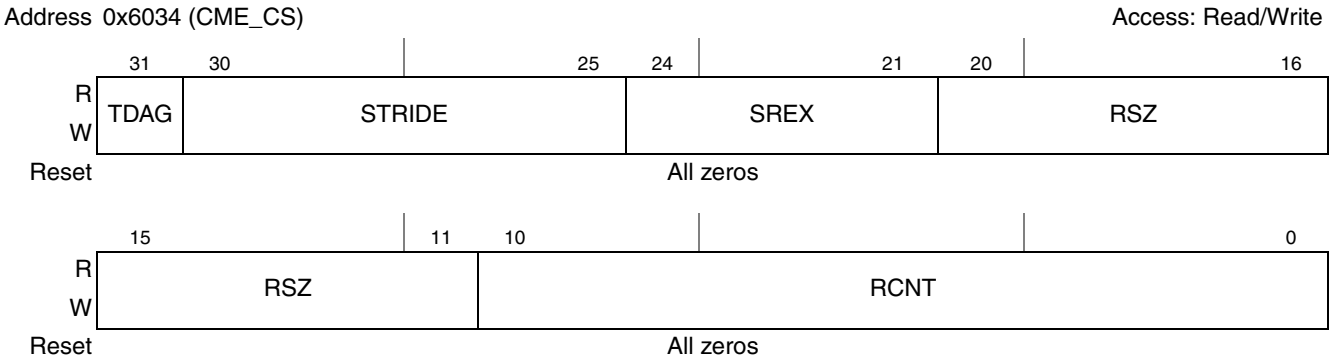

Figure 6-17. CME Block Stride Programming Register (CME_CS)

Table 6-14 describes the bit fields.

Table 6-14. Register CME_CS bits description

Field	Description
31 TDAG	Two dimensional address generation enable bit. 0 Address generation is strictly sequential (one dimensional). Stride size is forced to 0. 1 Address generation is two dimensional.
30–25 STRIDE	In case of two dimensional address generation this field represents stride size bits [15:10]; stride size bits [9:6] are taken from SREX. Stride size bits [6:0] should be always 0 (it has 128-byte resolution). This field is ignored for sequential address generation. Stride is illustrated on Figure 6-19. When valid stride should be equal or greater than row size to make sense.
24–21 SREX	Row size or stride bit extension in case of two dimensional address generation. For sequential address generation row size bits [19:16] are taken from SREX.
20–11 RSZ	Bits [15:6] of the row size. Row size bits [6:0] should be always 0 (it has 128-byte resolution). In case of two dimensional address generation row size bits [19:16] are 0. For sequential address generation row size bits [19:16] are taken from SREX. Sequential address row size is illustrated on Figure 6-18, two dimensional row size is illustrated on Figure 6-19.
10–0 RCNT	Number of rows

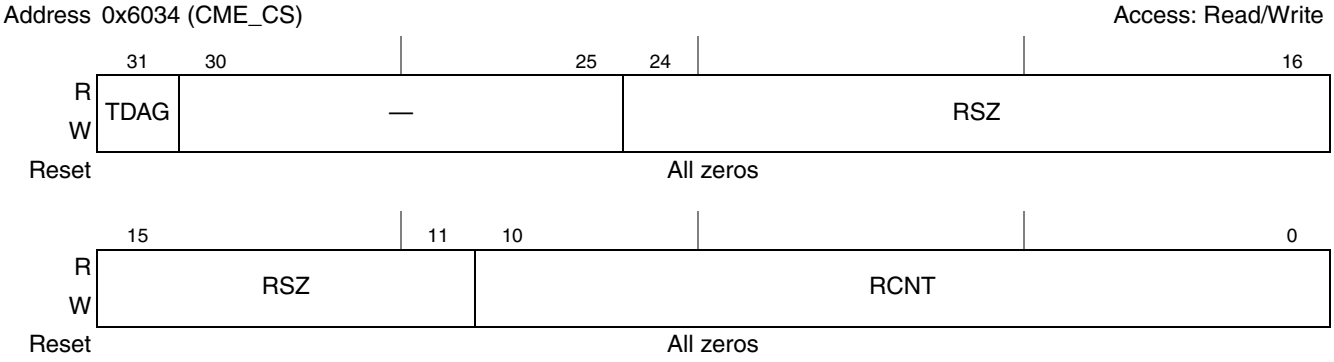

Figure 6-18. CME Stride Programming Register in Sequential Address Mode (One Dimensional)

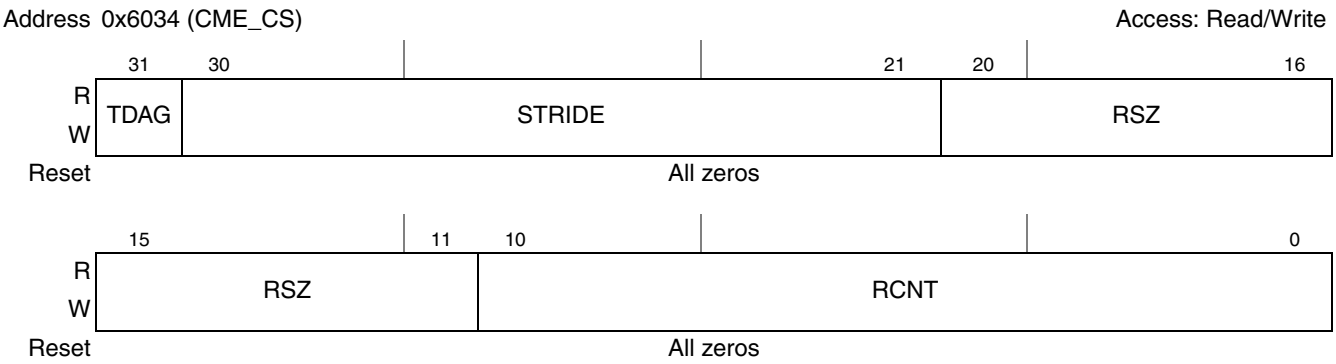

Figure 6-19. CME Stride Programming Register in Two Dimensional Mode

6.5.14 CME Block Address Programming Register (CME_CA)

The CME_CA register, shown in Figure 6-20, allows block CME programming. When it is written, the contents of the CME_CA, CME_CS, and CME_CC registers are sampled and used to configure CME channel. Should be configured only if CME_CR[BSTA] is not busy.

Figure 6-20. CME Address Programming Register (CME_CA)

Table 6-15 describes the bit fields.

Table 6-15. Register CME_CA bits description

Field	Description
31–6 ADDR	Bits [31:6] of cache maintenance instruction start address. Address bits [6:0] should be always 0 (it has 128-byte resolution). The address is virtual.
5–2 IDST	Maintenance instruction external interrupt destination. Used only if external interrupt completion event is chosen by CEV field below. Index in the doorbell register of MMU.
1–0 CEV	Maintenance instruction completion event. Controls special action that is executed after maintenance instruction completion. Ignored by query. 00 No special action is done on maintenance instruction completion. 01 Interrupt to local EPIC is asserted. Before interrupt heavyweight memory barrier is generated. 10 External interrupt is sent to destination defined by IDST. Before interrupt heavyweight memory barrier is generated. 11 Heavyweight memory barrier is generated.

6.5.15 CME Block Programming Status Register (CME_CR)

The CME_CR register, shown in Figure 6-21, shows the result of CME block programming. It is updated after a write to the CME_CA register. Core gets block programming result in the same form but this register is not updated.

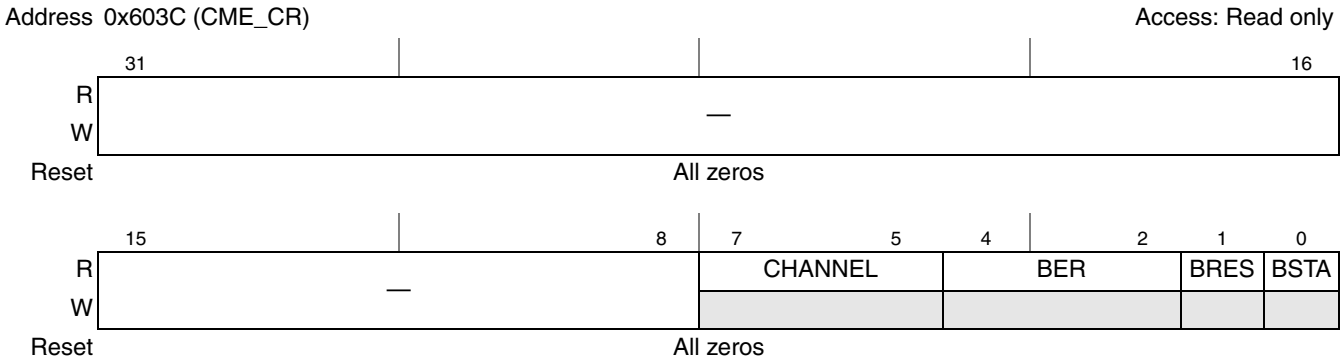

Figure 6-21. CME Programming Status Register (CME_CR)

Table 6-16 describes the bit fields.

Table 6-16. Register CME_CR bits description

Field	Description
31–8	Reserved
7–5 CHANNEL	Index of CME channel allocated for core cache maintenance instruction. Type of the channel (instruction or data) is defined by maintenance instruction's type. Valid if BRES and BSTA are 0 (BSTA is always 0 for core command result).
4–2 BER	CME block programming details Valid if BSTA is 0 (BSTA is always 0 for core command result). 000 Maintenance instruction was successfully inserted into empty CME channel 010 Maintenance instruction failed insertion into CME channel due to lack of space (all available channels are taken by maintenance instructions). 011 Maintenance instruction failed insertion into CME channel because CME is disabled by control registers, cache commands are disabled by MMU M_CR VCCC bit or full during error or suspend state 111 Maintenance instruction failed insertion into CME channel because it is invalid (wrong combination of parameters and values)
1 BRES	CME block programming result Valid if BSTA is 0 (BSTA is always 0 for core command result). 0 Maintenance instruction was successfully inserted 1 Maintenance instruction failed insertion into CME channel.
0 BSTA	CME block programming status Core command always return 1 in BSTA. 0 Done—allocation is finished and result is updated in BRES field and additional info in BER field. 1 Maintenance instruction channel allocation is still in progress (busy). Register must be polled until this value is changed.

6.5.16 CME Data Status Register (CME_DST)

The CME_DST register, shown in Figure 6-22, provides visibility and allows control of data CME block channels.

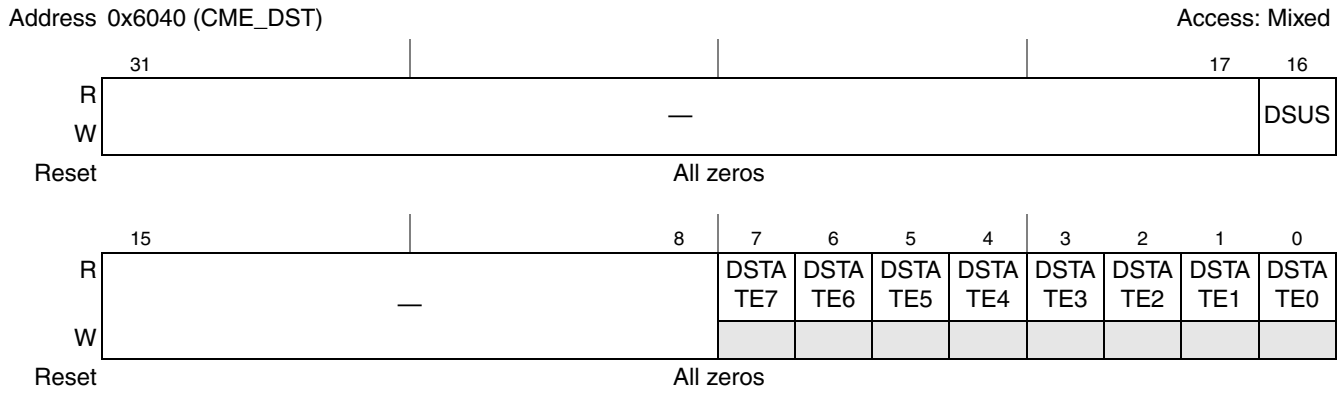

Figure 6-22. CME Data Status Register (CME_DST)

Table 6-17 describes the bit fields.

Table 6-17. Register CME_DST bits description

Field	Description
16 DSUS	Data channel suspend indication and control. Core SUSP instructions sets this bit and RESM instruction clears this bit. 0 Channels are not suspended and can be executed. 1 Channels are suspended.
7–0 DSTATE	Data channel status and control. Alias of the STATE field in the corresponding CME_DC register. 0 Channel is in empty state (idle). 1 Channel is in working state (active).

6.5.17 CME Data Error Register (CME_DER)

The CME_DER register, shown in Figure 6-23, provides the error status of data CME block channels.

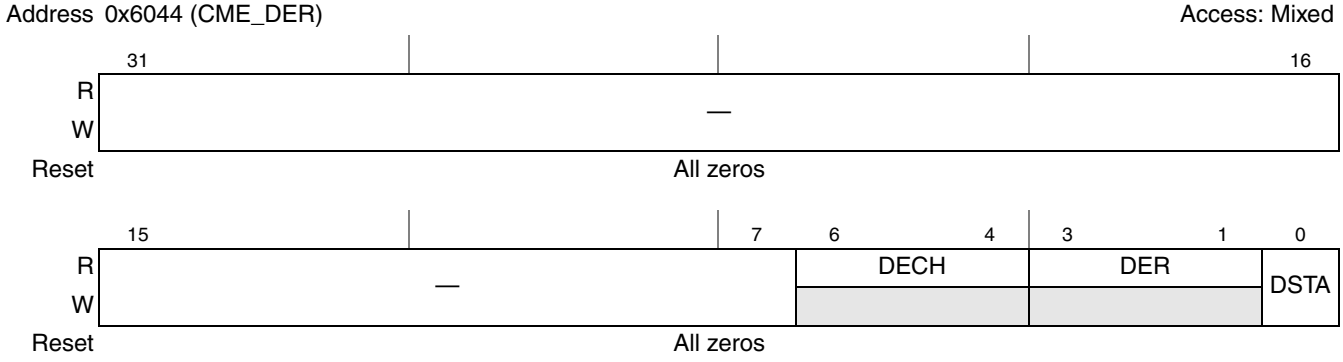

Figure 6-23. CME Data Error Register (CME_DER)

Table 6-18 describes the bit fields.

Table 6-18. Register CME_DER bits description

Field	Description
31–7	Reserved
6–4 DECH	Data channel with error number. Has no meaning if DSTA is 0.
3–1 DER	Data error type. Valid only if DSTA is 1. 000 Channel has multiple segment descriptor hit error reported by MMU 010 Channel has protection violation error reported by MMU 001 Channel has segment miss error reported by MMU. 111 Channel has been programmed to non cacheable area
0 DSTA	Data error status. 0 Channel has no errors. Writing this value clears the error and resumes the channel. 1 Channel has error reported by MMU. Details are in DER field. Writing 1 clears the error and disables all channels.

6.5.18 CME Data Interrupt Status Register (CME_DIN)

The CME_DIN register, shown in Figure 6-24, provides the interrupt status of data CME block channels.

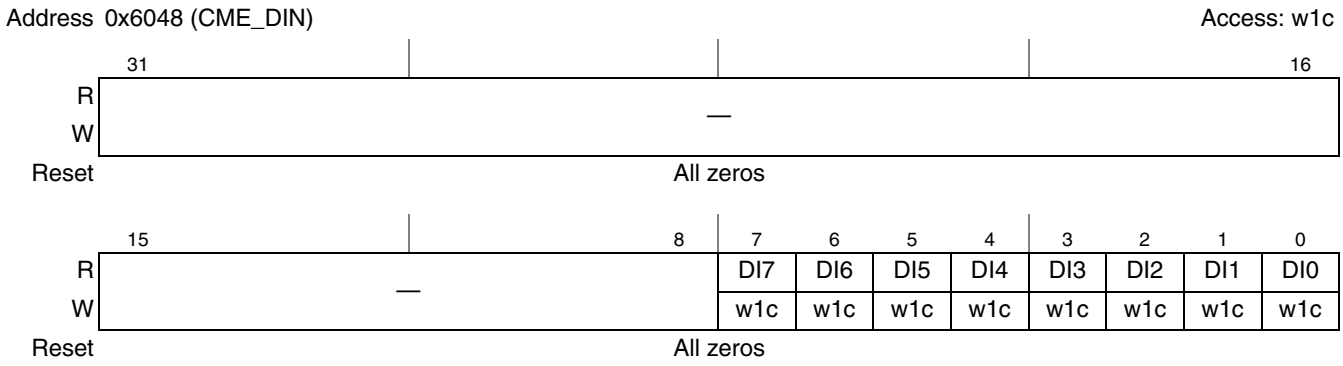

Figure 6-24. CME Data Interrupt Status Register (CME_DIN)

Table 6-19 describes the bit fields.

Table 6-19. Register CME_DIN bits description

Field	Description
31–6	Reserved
7–0 DI*n*	Data channel interrupt status. Asserted at the end of the cache maintenance instruction execution if defined by channel programming. Cleared by writing 1. 0 Channel has no asserted interrupt. 1 Channel has finished its operation and asserted interrupt.

6.5.19 CME Data External Doorbell Interrupt Status Register (CME_DMS)

The CME_DMS register, shown in Figure 6-25, provides the doorbell interrupts status of data CME block channels.

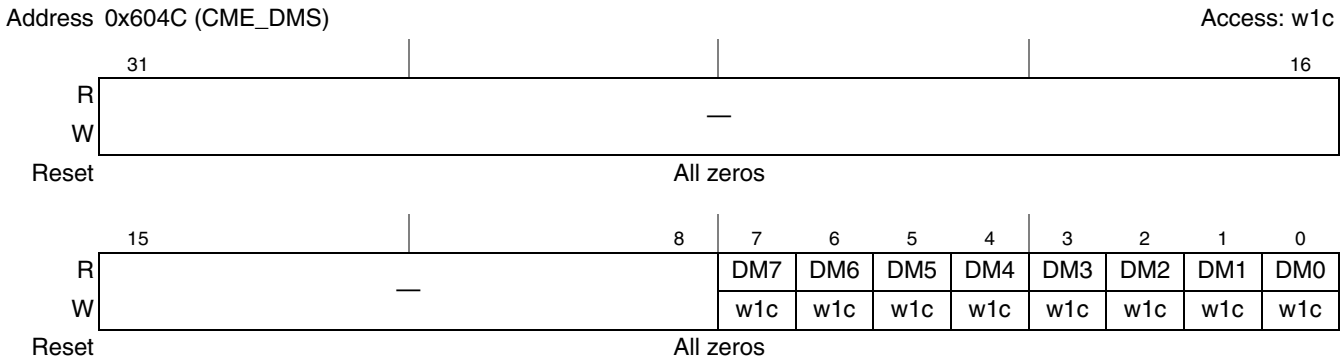

Figure 6-25. CME Data External Doorbell Interrupt Status Register (CME_DMS)

Table 6-20 describes the bit fields.

Table 6-20. Register CME_DMS bits description

Field	Description
31–8	Reserved
7–0 DM*n*	Data channel doorbell interrupt status. Asserted, if defined by channel programming, at the end of the cache maintenance instruction execution. Cleared by writing 1. 0 Channel has no asserted doorbell interrupt 1 Channel has finished its operation and asserted doorbell interrupt

6.5.20 CME Data Channel Control Register (CME_DC<i>)

The CME_DC<i> registers, shown in Figure 6-26, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. It contains the data channel current status.

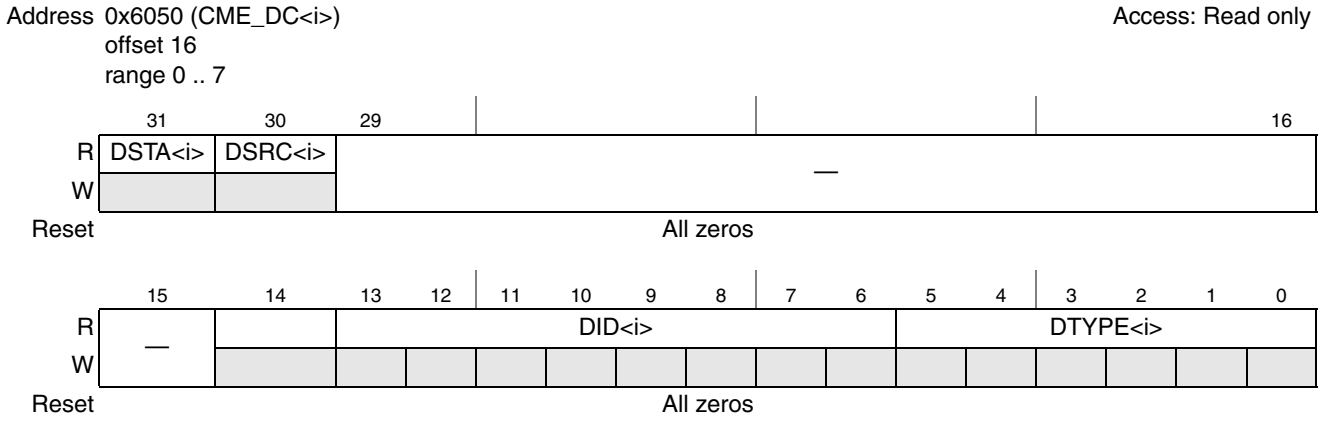

Figure 6-26. CME Data Channel Control Register (CME_DC<i>)

Table 6-21 describes the bit fields.

Table 6-21. Register CME_DC<i>(0 .. 7) bits description

Field	Description
31 DSTA<i>	State of the channel. Read only alias from the corresponding DSTATE field in CME_DST register.
30 DSRC<i>	Source of the current cache maintenance instruction. 0 Maintenance instruction was programmed by core. 1 Maintenance instruction was programmed by peripheral bus.
13–6 DID<i>	DID of the cache maintenance instruction.
5–0 DTYPE<i>	Type of DCache maintenance instructions: 000000: DFETCHB.L12 000001: DFETCHB.L2 000010: DFETCHB.LCK.L2 000011: DFETCHB.L12.ITW 000100: DFETCHB.L2.ITW 000101: DFETCHB.LCK.L2.ITW 000111: DUNLOCKB.L2 001000: DCMB.SYNC.L1 001011: DCMB.FLUSH.L12 001100: DCMB.INVAL.L1 001101: DCMB.INVAL.L12

6.5.21 CME Data Channel Stride Register (CME_DS<i>)

The CME_DS<i> registers, shown in Figure 6-27, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. They contain the data channel current status. Please check section Section 6.5.13, “CME Block Stride Programming Register (CME_CS),” for a more detailed explanation about stride/row size presentation.

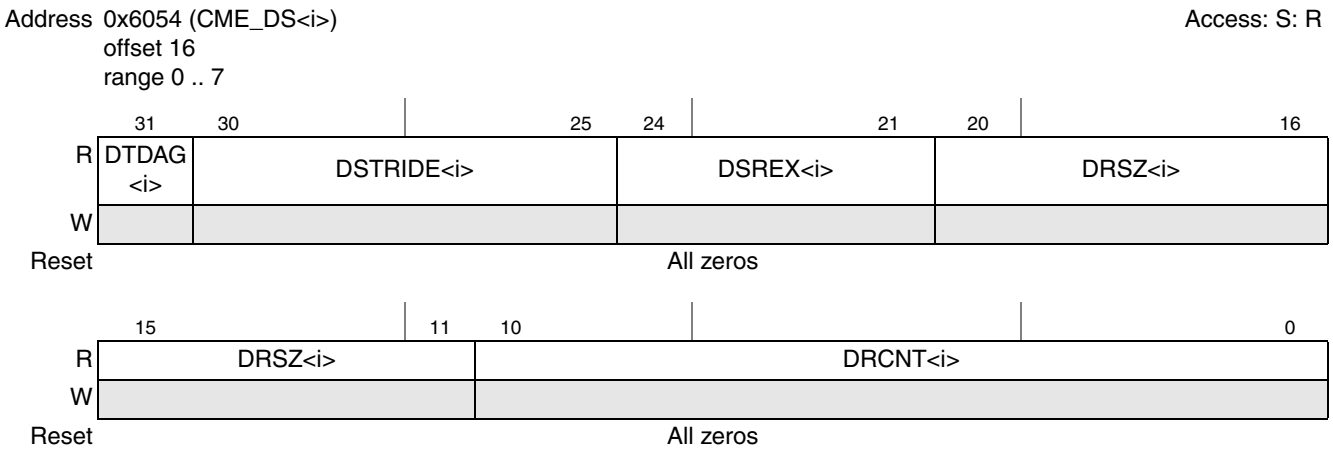

Figure 6-27. CME Data Channel Stride Register (CME_DS<i>)

Table 6-22 describes the bit fields.

Table 6-22. Register CME_DS<i>(0 .. 7) bits description

Field	Description
31 DTDAG<i>	Two dimensional address generation enable bit. 0 Address generation is strictly sequential (one dimensional). Stride size is forced to 0. 1 Address generation is two dimensional.
30–25 DSTRIDE<i>	In case of two dimensional address generation this field represents stride size bits [15:10], stride size bits [9:6] are taken from SREX. Stride size bits [5:0] are always 0. This field is ignored for sequential address generation. When valid stride should be equal or greater than row size to make sense.
24–21 DSREX<i>	Row size or stride bit extension. In case of two dimensional address generation. For sequential address generation row size bits [19:16] are taken from SREX.
20–11 DRSZ<i>	Bits [15:6] of the row size. Row size bits [5:0] are always 0. In case of two dimensional address generation row size bits [19:16] are 0. For sequential address generation row size bits [19:16] are taken from SREX.
10–0 DRCNT<i>	Number of rows left until end of the cache maintenance instruction operation, changed in real time. Can be used for debug. If there is no error, this number is decremented speculatively and can be slightly smaller than actually executed.

6.5.22 CME Data Channel Address Register (CME_DA<i>)

The CME_DA<i> registers, shown in Figure 6-28, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. They contain the data channel current status.

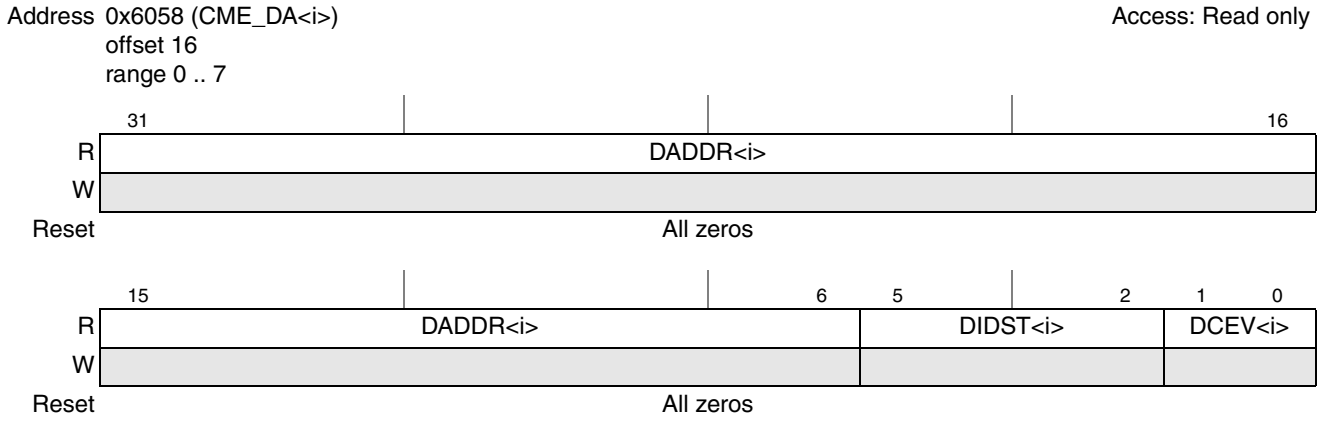

Figure 6-28. CME Data Channel Address Register (CME_DA<i>)

Table 6-23 describes the bit fields.

Table 6-23. Register CME_DA<i>(0 .. 7) bits description

Field	Description
31–6 DADDR<i>	Bits [31:6] of cache maintenance instruction start address. Address bits [6:0] are always 0 (it has 128-byte resolution). The address is virtual. It represents the real-time address that is used by this channel. Its value should be used to debug MMU errors. If there is no error, this address is incremented speculatively and can be slightly larger than actually executed by the cache.
5–2 DIDST<i>	Maintenance instruction external interrupt destination. Used only if external interrupt completion event is chosen by CEV field below. Index in the doorbell register of the MMU.
1–0 DCEV<i>	Maintenance instruction completion event. Controls the special action that is executed after maintenance instruction completion. Ignored by query. 00 No special action is done on maintenance instruction completion 01 Interrupt to local EPIC is asserted. Before interrupt heavyweight memory barrier is generated. 10 External interrupt is sent to destination defined by IDST. Before interrupt heavy weight memory barrier is generated. 11 Heavyweight memory barrier is generated.

6.5.23 CME Program Status Register (CME_PST)

The CME_PST register, shown in Figure 6-29, provides visibility and allows control of program CME block channels.

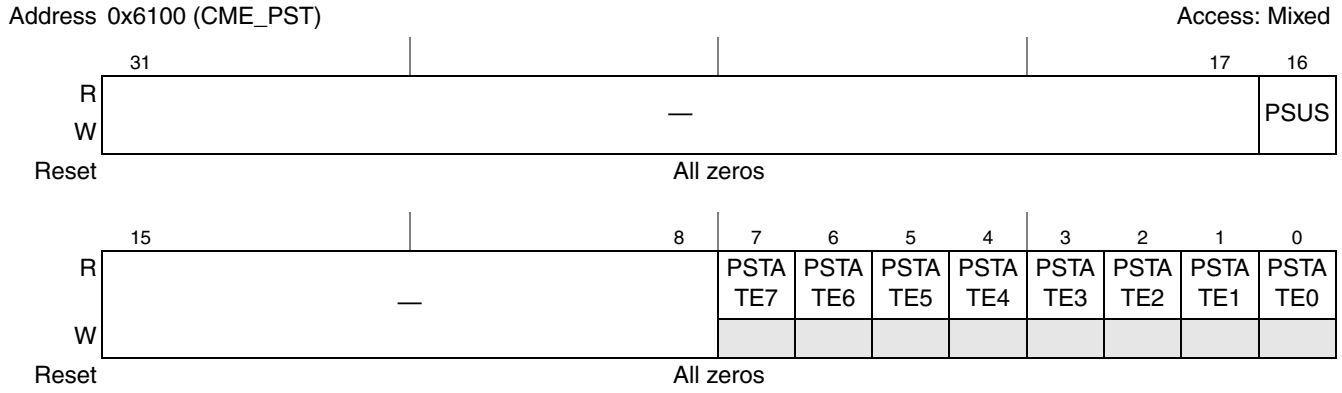

Figure 6-29. CME Program Status Register (CME_PST)

Table 6-24 describes the bit fields.

Table 6-24. Register CME_PST bits description

Field	Description
31–17	Reserved
16 PSUS	Program channel suspend indication and control. Core SUSP instructions sets this bit and RESM instruction clears this bit. 0 Channels are not suspended and can be executed. 1 Channels are suspended.
15–8	Reserved
7–0 PSTATE	Program channel status and control. Alias of the STATE field in the corresponding CME_PC register. 0 Channel is in empty state (idle). 1 Channel is in working state (active).

6.5.24 CME Program Error Register (CME_PER)

The CME_PER register, shown in Figure 6-30, provides the error status of program CME block channels.

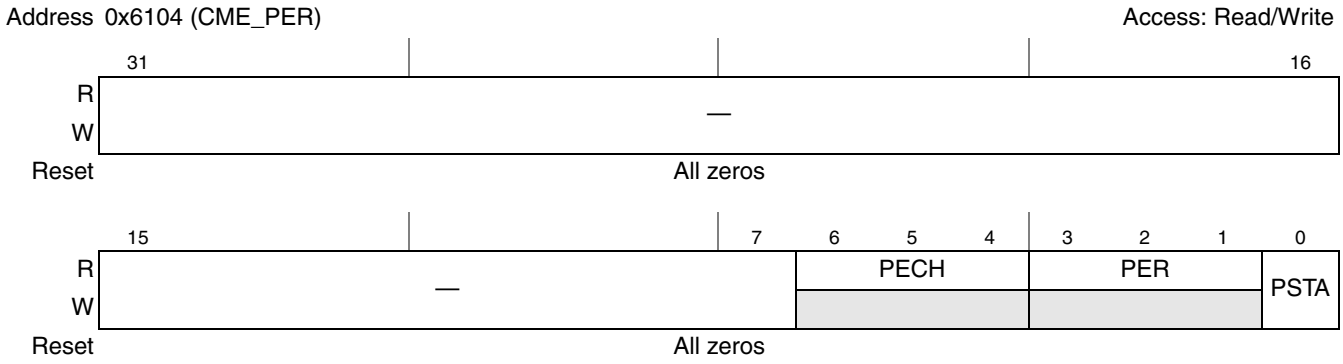

Figure 6-30. CME Program Error Register (CME_PER)

Table 6-25 describes the bit fields.

Table 6-25. Register CME_PER bits description

Field	Description
31–7	Reserved
6–4 PECH	Program channel with error number. Has no meaning if PSTA is 0.
3–1 PER	Program error type. Valid only if PSTA is 1. 000 Channel has multiple segment descriptor hit error reported by MMU 010 Channel has protection violation error reported by MMU 001 Channel has segment miss error reported by MMU. 111 Channel has been programmed to non cacheable area
0 PSTA	Program error status. 0 Channel has no errors. Writing this value clears the error and resumes the channel. 1 Channel has error reported by MMU. Details are in PER field. Writing 1 clears the error and disables all channels .

6.5.25 CME Program Interrupt Status Register (CME_PIN)

The CME_PIN register, shown in Figure 6-31, provides the interrupt status of program CME block channels.

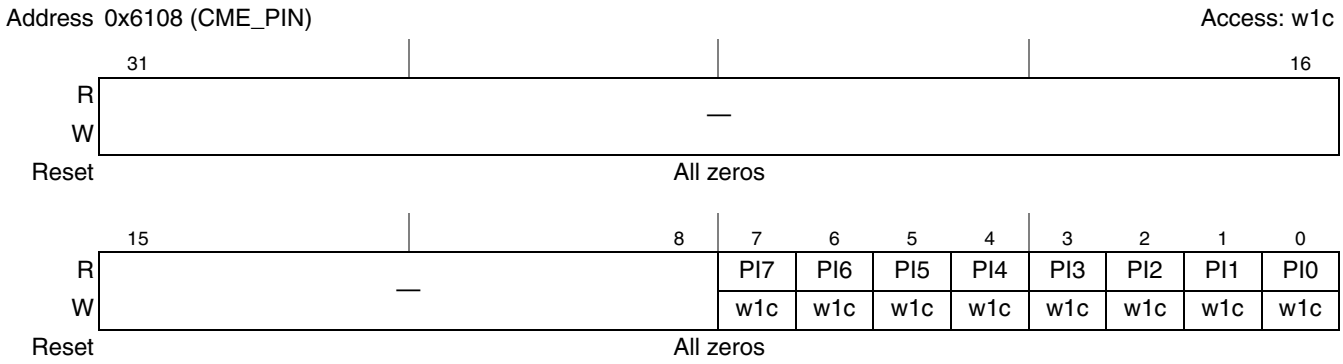

Figure 6-31. CME Program Interrupt Status Register (CME_PIN)

Table 6-26 describes the bit fields.

Table 6-26. Register CME_PIN bits description

Field	Description
31–8	Reserved
PI*n*	Program channel interrupt status. Asserted, if defined by channel programming, at the end of the cache maintenance instruction execution. Cleared by writing 1. 0 Channel has no asserted interrupt 1 Channel has finished its operation and asserted interrupt

6.5.26 CME Program External Doorbell Interrupt Status Register (CME_PMS)

The CME_PMS register, shown in Figure 6-32, provides the doorbell interrupts status of program CME block channels.

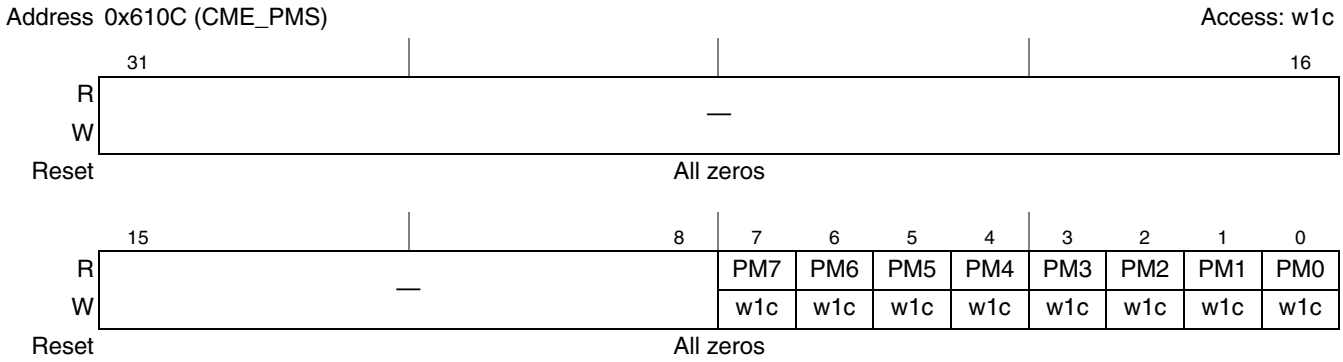

Figure 6-32. CME Program External Doorbell Interrupt Status Register (CME_PMS)

Table 6-27 describes the bit fields.

Table 6-27. Register CME_PMS bits description

Field	Description
31–8	Reserved
7–0 PM	Program channel doorbell interrupt status. Asserted, if defined by channel programming, at the end of the cache maintenance instruction execution. Cleared by writing 1. 0 Channel has no asserted doorbell interrupt 1 Channel has finished its operation and asserted doorbell interrupt

6.5.27 CME Program Channel Control Register (CME_PC<i>)

The CME_PC<i> registers, shown in Figure 6-33, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. It contains the program channel current status.

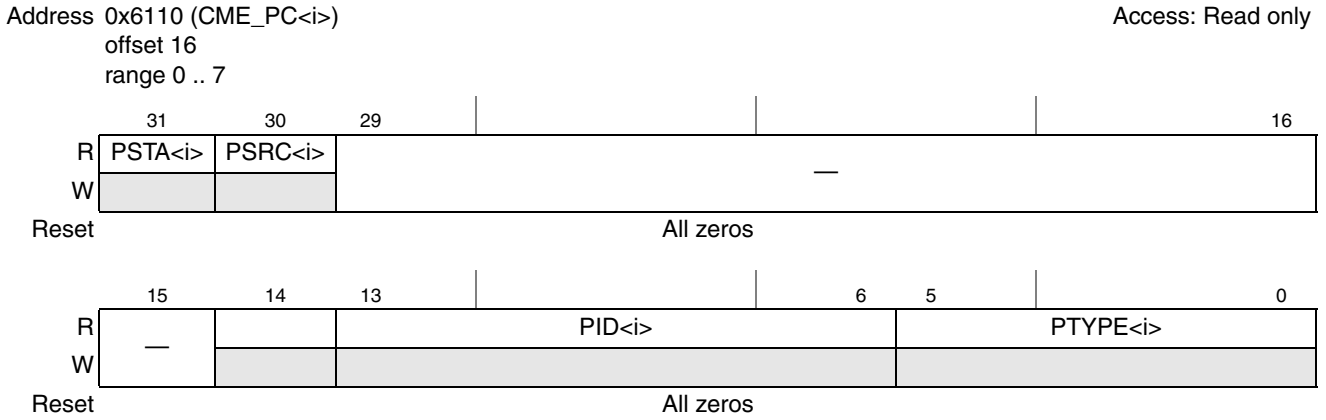

Figure 6-33. CME Program Channel Control Register (CME_PC<i>)

Table 6-28 describes the bit fields.

Table 6-28. Register CME_PC<i>(0 .. 7) bits description

Field	Description
31 PSTA<i>	State of the channel. Read only alias from the corresponding PSTATE field in CME_PST register.
30 PSRC<i>	Source of the current cache maintenance instruction. 0 Maintenance instruction was programmed by core. 1 Maintenance instruction was programmed by peripheral bus.
29–15	Reserved
13–6 PID<i>	PID of the cache maintenance instruction.
5–0 PTYPE<i>	Type of DCache maintenance instructions: 100000: PFETCHB.L12 100001: PFETCHB.L2 100010: PFETCHB.LCK.L2 100111: PUNLOCKB.L2 101100: PCMB.INVAL.L1 101101: PCMB.INVAL.L12

6.5.28 CME Program Channel Stride Register (CME_PS<i>)

The CME_PS<i> registers, shown in Figure 6-34, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. They contain the program channel current status. See Section 6.5.13, “CME Block Stride Programming Register (CME_CS),” for a more detailed explanation about stride/row size presentation.

Figure 6-34. CME Program Channel Stride Register (CME_PS<i>)

Table 6-29 describes the bit fields.

Table 6-29. Register CME_PS<i>(0 .. 7) bits description

Field	Description
31 PTDAG<i>	Two dimensional address generation enable bit. 0 Address generation is strictly sequential (one dimensional). Stride size is forced to 0. 1 Address generation is two dimensional.
30–25 PSTRIDE<i>	In case of two dimensional address generation this field represents stride size bits [15:10], stride size bits [9:6] are taken from SREX. Stride size bits [5:0] are always 0. This field is ignored for sequential address generation. When valid stride should be equal or greater than row size to make sense.
24–21 PSREX<i>	Row size or stride bit extension. In case of two dimensional address generation. For sequential address generation row size bits [19:16] are taken from SREX.
20–11 PRSZ<i>	Bits [15:6] of the row size. Row size bits [5:0] are always 0. In case of two dimensional address generation row size bits [19:16] are 0. For sequential address generation row size bits [19:16] are taken from SREX.
10–0 PRCNT<i>	Number of rows left until end of the cache maintenance instruction operation, changed in the real time. Can be used for debug. If there is no error this number is decremented speculatively and can be slightly smaller than actually executed.

6.5.29 CME Program Channel Address Register (CME_PA<i>)

The CME_PA<i> registers, shown in Figure 6-35, are a group of registers for which the *i* variable in the register name is a number between 0 and 7. They contain the program channel current status.

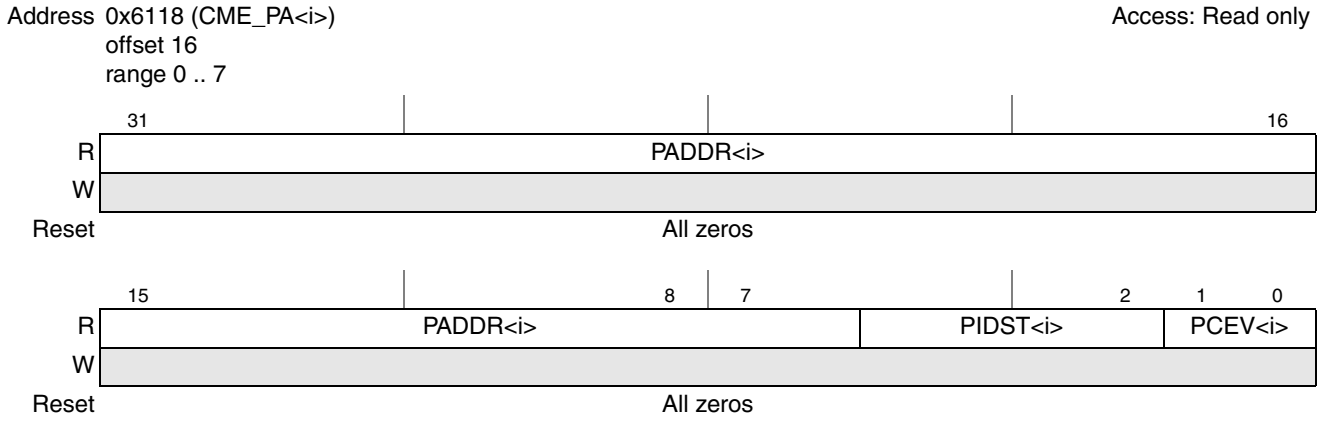

Figure 6-35. CME Program Channel Address Register (CME_PA<i>)

Table 6-30 describes the bit fields.

Table 6-30. Register CME_PA<i>(0 .. 7) bits description

Field	Description
31–6 PADDR<i>	Bits [31:6] of cache maintenance instruction start address. Address bits [6:0] are always 0 (it has 128-byte resolution). The address is virtual. It represents the real-time address that is used by this channel. Its value should be used to debug MMU errors. If there is no error this address is incremented speculatively and can be slightly larger than actually executed by the cache.
5–2 PIDST<i>	Maintenance instruction external interrupt destination. Used only if external interrupt completion event is chosen by CEV field below. Index in the Doorbell register of MMU.
1–0 PCEV<i>	Maintenance instruction completion event. Controls special action that is executed after maintenance instruction completion. Ignored by query. 00 No special action is done on maintenance instruction completion 01 Interrupt to local EPIC is asserted. Before it heavy weight memory barrier is generated. 10 External interrupt is sent to destination defined by IDST. Before it heavy weight memory barrier is generated. 11 Heavyweight memory barrier is generated.

Chapter 7
Interrupts

Exceptions and interrupts are generated by conditions inside the StarCore SC3900 core, the SC3900 DSP subsystem modules, or external sources. Generally, the non-core internal and the external interrupt sources are prioritized and arbitrated in the SC3900 DSP subsystem Enhanced Programmable Interrupt Controller (EPIC) module. The interrupt sources in the SC3900 DSP subsystem are as follows:

- SC3900 core internal exceptions

  Internal exception sources include trap, illegal instruction, debug exception, and DALU overflow. For details, see the *SC3900 FVP Core Reference Manual*.
- MMU interrupts

  The core has dedicated exception vectors activated by the MMU that are treated as internal core exceptions. The MMU has its own exception sources, some internal to the MMU (such as protection violation and MATT miss) and some external to the MMU (such as an EDC error from L1 caches). MMU exceptions are synchronized to core accesses and are therefore precise. For details, see Chapter 3, “Memory Management Unit.”
- EPIC

  The EPIC manages interrupts generated by the SC3900 DSP subsystem and external peripherals. It uses a fixed set of priority rules and passes interrupts with the highest priority to the core. The EPIC also manages the acknowledgment of edge-triggered interrupts. EPIC exceptions are imprecise. Their service is not directly related to the program flow.

Figure 7-1 shows the functional organization of the EPIC, including all external signals available to the user. The EPIC is a peripheral module that serves all interrupt requests (IRs) and critical interrupts (CIs) received from the SC3900 DSP subsystem peripherals and I/O pins. The EPIC registers are

memory-mapped to the SC3900 core and can be accessed through the SC3900 DSP subsystem peripheral bus and by any SoC master (for example, other core).

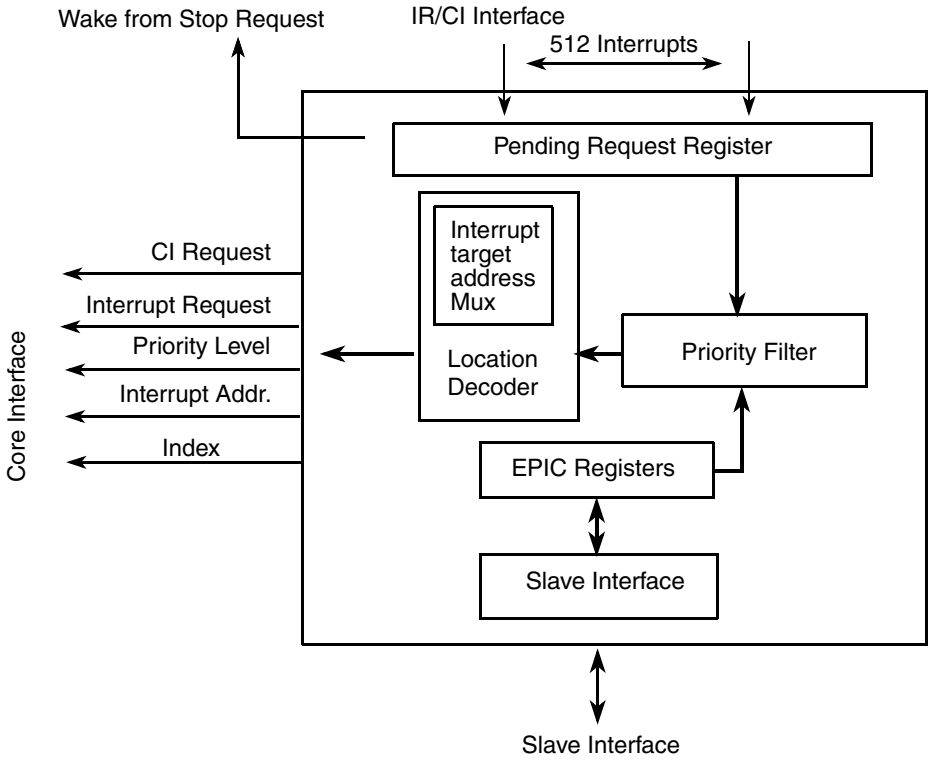

Figure 7-1. EPIC block diagram

The EPIC has an interrupt line interface external to the peripherals of the DSP subsystem. The EPIC provides an ideal RTOS solution that readily addresses SoC interrupt-related needs. It handles up to **512** interrupt requests (**480)** external to the DSP subsystem), which also can be configured as CIs.

The EPIC decodes the index of the 512 interrupts according to their configured priority. According to the interrupt index, the EPIC associates an interrupt with its ISR dispatcher handler 28 bit virtual address for fast advanced low latency interrupt handling. In addition to the dispatcher address, the EPIC also provides the index of the interrupt as information to the ISR. This prevents needing to query the EPIC for software interrupt source. Both the dispatcher address and the interrupt index greatly accelerate the interrupt service by reducing unnecessary change of flows and unnecessary query of the EPIC.

The EPIC can handle 33 levels of interrupt priorities, with 32 levels dedicated to IRs and one level for CIs.

7.1 EPIC features

The EPIC has the following features:

- **512** interrupt inputs, which are grouped internally in 256 groups of 2 interrupts. each group has the full flexibility configuration. The enable/disable configuration bit is provided for each interrupt.
- 16 configured dispatcher (ISR) full address.

- Fast interrupt handling mechanism. Core receives a full, 26 bit address and the original interrupt index.
- **32** (first) reserved interrupt ports for internal DSP subsystem use and 480 are external to the DSP subsystem.
- All interrupts, except for the SC3900 DSP subsystem CIs, can be configured as regular interrupts (IRs) or Critical Interrupts (CIs) using the P_IPL*x* registers.
- Interrupts can be quickly enabled or disabled using the P_ENDIS*x* registers. In extreme cases a single DI bit in the EPIC can be used to disable all the EPIC controlled interrupts (even CIs).
- Interrupts can be configured as edge or level triggered using the P_ELR*x* registers. The SC3900 DSP subsystem interrupts are hard configured to the appropriate edge/level triggering mode.
- 33 priority levels, as follows:
  - Priority level 0 (interrupt disabled)
  - Priority levels 1 through **31** (where **31** is the highest priority)
  - CI level (highest priority)
- There is fixed priority among interrupts at the same priority level. When more than one interrupt has the same priority, the interrupt with the lowest index has precedence.
- Support for software acknowledgment of all edge-triggered IRs and CIs.
- Visibility to all pending IRs. The status of an IR (asserted, negated) can be read from the memory-mapped P_IPRx registers.
- Software induced interrupts. The software can induce any of the 512 interrupts.
- EPIC registers can be accessed through the slave peripheral bus with respect to their memory mapped address in supervisor mode. This access can be done by the core or any other SoC peripheral bus master (i.e. one of the other cores)

7.2 EPIC functional description

The EPIC can serve a total of **512** interrupts. The interrupts are divided to 256 groups. Each group holds two adjacent pair interrupts. Each group has its own set of configuration registers, which provides flexibility from one side and compact EPIC from the other side. The enable/disable configuration is provided per interrupt to allow choosing precisely which interrupts each core serves.

Each group of interrupt requests (IR) can be configured as edge triggered or level triggered and assigned a priority between 0 and **31** or a CI priority, where priority 0 masks the interrupt. At reset, all external IRs are masked (set to priority 0) and configured as level triggered. The internal subsystem interrupts are hard configured to their appropriate edge/level triggering mode and the CIs are hard configured as CIs.

The SC3900 architecture enables programmers to ensure that specific execution sets are executed as an uninterruptible sequence by masking all interrupts using the DI (disable interrupts) instruction. The masking imposed by the DI is removed by issuing the EI (enable interrupts) instruction. CIs are not masked by the DI command.

Interrupts can also be masked to a specified priority level by selecting the priority level in the SC3900 core status register I[**4**:0]. The core handles only CIs or interrupts with an interrupt priority level (IPL) higher than the current interrupt mask value. At reset, these bits are set and all maskable interrupts are disabled.

The interrupt mask bits reflect the current IPL of the core. For details on the core status register, consult the *SC3900 FVP Core Reference Manual*.

The EPIC selects one of the 512 pending interrupts according to their group priority and CI configuration. The lower interrupt index wins if groups have the same priority. According to the interrupt group dispatchers index, the EPIC selects its dispatcher target. The EPIC then provides the core with a full 28 bit virtual address to allow a direct jump to the interrupt service routine (ISR). It is the programmer's responsibility to configure the correct dispatcher index for each group of interrupts. The initial values of the dispatcher's targets are all zeros and can be configured by software during the boot sequence. All the interrupts are disabled from reset until the end of boot sequence, and they are enabled by boot at the end of its sequence.

The EPIC supports a software induced interrupt. A slave transaction that specifies the index of any of the 512 interrupts and a set bit cause the EPIC to respond as if it was a real interrupt.

The EPIC provides the following signals to the core:

- An IR or CI signals to the SC3900 core indicating that an IR or CI input has requested interrupt service from the SC3900 core.
- An IPL[4:0] signal indicating the priority of the IR
- A TRG[31:6] signal indicating full virtual OS dispatcher handler address to which the core should jump to directly (with no address calculation).
- An interrupt index[9:0].

7.3 Memory map and registers

The SC3900 DSP subsystem interrupt programming model consists of the following:

- Interrupt masking field in the core status register
- EPIC P_IPL*i* registers for IPL and CI values configurations for each interrupt
- EPIC P_DISPi registers for selecting the dispatcher for each interrupt
- EPIC P_TRGi registers for sixteen optional dispatchers target.
- EPIC P_ELR*i* registers for programming the triggering mode of each interrupt
- EPIC P_ENDIS*i* registers for enabling and disabling interrupts
- EPIC P_IPR*i* registers for acknowledging edge triggered interrupts
- EPIC P_SWII register for software induced interrupt.
- EPIC P_DI register for disabling all interrupts with one write access

7.3.1 DSP subsystem interrupt table

Table 7-1 summarizes how interrupts are routed. For details on the interrupt scheme of the core, see the *SC3900 FVP Core Reference Manual.*

Upon interrupt service, ETP and EID are sampled into the EIDR (core register) and available for SW inspection in the ISR. This is useful since interrupt share dispatchers between them so differentiation cannot be done based on the vector address.

Table 7-1. DSP subsystem interrupt routing

ETP	EID	Name	Description
Core exceptions			
00_xxxx	xx_xxxx_xxxx	core exception	Trap, illegal, and so on. See the *SC3900 FVP Core Reference Manual*.
MMU exceptions			
01_xxxx	xx_xxxx_xxxx	MMU exception	See the *SC3900 FVP Core Reference Manual*.
Debug exceptions			
10_xxxx	xx_xxxx_xxxx	debug exception	Debug events. See the *SC3900 FVP Core Reference Manual*.
EPIC interrupts (imprecise)			
11_n000[1]	0x0	CBE	Cme block error
11_n000[1]	0x1	WE	Write error on SB or non cacheable write hit
11_n000[1]	0x2-3	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x4	L2E	L2 cache error
11_n000[1]	0x5-7	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x8	CMS	Critical message
11_n000[1]	0x9	CGMS	Critical guest message
11_n000[1]	0xA	WDTI	WDT interrupt
11_n000[1]	0xB-0xF	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x10	MS	message
11_n000[1]	0x11	GMS	Guest message
11_n000[1]	0x12	MMS	Machine check message
11_n000[1]	0x13	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x14	I_TM0	Timer 0 interrupt
11_n000[1]	0x15	I_TM1	Timer 1 interrupt
11_n000[1]	0x16	I_TM2	Timer 2 interrupt
11_n000[1]	0x17	I_TM3	Timer 3 interrupt
11_n000[1]	0x18	CBC	Cme block completion
11_n000[1]	0x19	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x1A–0x1F	RESERVED	Reserved for internal DSP subsystem use
11_n000[1]	0x20–0x1FF	IRQ32–IRQ511	External-to-the-DSP subsystem interrupts

[1] n - 0 for critical interrupt, 1 for normal interrupt

7.3.1.1 DSP subsystem interrupt address

The address of the exception routine is programmed in different locations for the different interrupt sources the SC3900 core serves. Table 7-3 lists the registers that hold the exception address per source.

Table 7-2. Addresses of exception routines

Exception	Address is set by	Remarks
Core	CESRA 0/1	See the *SC3900 FVP Core Reference Manual*.
MMU	M_DESRA 0/1 and M_PESRA0/1	See the "Memory Management Unit" chapter of this document.
Debug	DESAR	See the *SC3900 FVP Core Reference Manual*.
EPIC	P_TRG_n	See on page 7-16.

7.3.2 EPIC register summary

This table provides the register summary.

Table 7-3. EPIC register summary

Address offset from EPIC base	Use	Section/page
General registers		
000	EPIC interrupt priority level register 0	7.3.3.1/11
004	EPIC interrupt priority level register 1	7.3.3.1/11
008	EPIC interrupt priority level register 2	7.3.3.1/11
00C	EPIC interrupt priority level register 3	7.3.3.1/11
010	EPIC interrupt priority level register 4	7.3.3.1/11
014	EPIC interrupt priority level register 5	7.3.3.1/11
018	EPIC interrupt priority level register 6	7.3.3.1/11
01C	EPIC interrupt priority level register 7	7.3.3.1/11
020	EPIC interrupt priority level register 8	7.3.3.1/11
024	EPIC interrupt priority level register 9	7.3.3.1/11
028	EPIC interrupt priority level register 10	7.3.3.1/11
02C	EPIC interrupt priority level register 11	7.3.3.1/11
030	EPIC interrupt priority level register 12	7.3.3.1/11
034	EPIC interrupt priority level register 13	7.3.3.1/11
038	EPIC interrupt priority level register 14	7.3.3.1/11
03C	EPIC interrupt priority level register 15	7.3.3.1/11
040	EPIC interrupt priority level register 16	7.3.3.1/11

Table 7-3. EPIC register summary (continued)

Address offset from EPIC base	Use	Section/page
044	EPIC interrupt priority level register 17	7.3.3.1/11
048	EPIC interrupt priority level register 18	7.3.3.1/11
04C	EPIC interrupt priority level register 19	7.3.3.1/11
050	EPIC interrupt priority level register 20	7.3.3.1/11
054	EPIC interrupt priority level register 21	7.3.3.1/11
058	EPIC interrupt priority level register 22	7.3.3.1/11
05C	EPIC interrupt priority level register 23	7.3.3.1/11
060	EPIC interrupt priority level register 24	7.3.3.1/11
064	EPIC interrupt priority level register 25	7.3.3.1/11
068	EPIC interrupt priority level register 26	7.3.3.1/11
06C	EPIC interrupt priority level register 27	7.3.3.1/11
070	EPIC interrupt priority level register 28	7.3.3.1/11
074	EPIC interrupt priority level register 29	7.3.3.1/11
078	EPIC interrupt priority level register 30	7.3.3.1/11
07C	EPIC interrupt priority level register 31	7.3.3.1/11
080	EPIC interrupt priority level register 32	7.3.3.1/11
084	EPIC interrupt priority level register 33	7.3.3.1/11
088	EPIC interrupt priority level register 34	7.3.3.1/11
08C	EPIC interrupt priority level register 35	7.3.3.1/11
090	EPIC interrupt priority level register 36	7.3.3.1/11
094	EPIC interrupt priority level register 37	7.3.3.1/11
098	EPIC interrupt priority level register 38	7.3.3.1/11
09C	EPIC interrupt priority level register 39	7.3.3.1/11
0A0	EPIC interrupt priority level register 40	7.3.3.1/11
0A4	EPIC interrupt priority level register 41	7.3.3.1/11
0A8	EPIC interrupt priority level register 42	7.3.3.1/11
0AC	EPIC interrupt priority level register 43	7.3.3.1/11
0B0	EPIC interrupt priority level register 44	7.3.3.1/11
0B4	EPIC interrupt priority level register 45	7.3.3.1/11
0B8	EPIC interrupt priority level register 46	7.3.3.1/11
0BC	EPIC interrupt priority level register 47	7.3.3.1/11

Table 7-3. EPIC register summary (continued)

Address offset from EPIC base	Use	Section/page
0C0	EPIC interrupt priority level register 48	7.3.3.1/11
0C4	EPIC interrupt priority level register 49	7.3.3.1/11
0C8	EPIC interrupt priority level register 50	7.3.3.1/11
0CC	EPIC interrupt priority level register 51	7.3.3.1/11
0D0	EPIC interrupt priority level register 52	7.3.3.1/11
0D4	EPIC interrupt priority level register 53	7.3.3.1/11
0D8	EPIC interrupt priority level register 54	7.3.3.1/11
0DC	EPIC interrupt priority level register 55	7.3.3.1/11
0E0	EPIC interrupt priority level register 56	7.3.3.1/11
0E4	EPIC interrupt priority level register 57	7.3.3.1/11
0E8	EPIC interrupt priority level register 58	7.3.3.1/11
0EC	EPIC interrupt priority level register 59	7.3.3.1/11
0F0	EPIC interrupt priority level register 60	7.3.3.1/11
0F4	EPIC interrupt priority level register 61	7.3.3.1/11
0F8	EPIC interrupt priority level register 62	7.3.3.1/11
0FC	EPIC interrupt priority level register 63	7.3.3.1/11
100	EPIC interrupt dispatcher selector register 0	7.3.3.2/14
104	EPIC interrupt dispatcher selector register 1	7.3.3.2/14
108	EPIC interrupt dispatcher selector register 2	7.3.3.2/14
10C	EPIC interrupt dispatcher selector register 3	7.3.3.2/14
110	EPIC interrupt dispatcher selector register 4	7.3.3.2/14
114	EPIC interrupt dispatcher selector register 5	7.3.3.2/14
11C	EPIC interrupt dispatcher selector register 7	7.3.3.2/14
120	EPIC interrupt dispatcher selector register 8	7.3.3.2/14
124	EPIC interrupt dispatcher selector register 9	7.3.3.2/14
128	EPIC interrupt dispatcher selector register 10	7.3.3.2/14
12C	EPIC interrupt dispatcher selector register 11	7.3.3.2/14
130	EPIC interrupt dispatcher selector register 12	7.3.3.2/14
134	EPIC interrupt dispatcher selector register 13	7.3.3.2/14
138	EPIC interrupt dispatcher selector register 14	7.3.3.2/14
13C	EPIC interrupt dispatcher selector register 15	7.3.3.2/14

Table 7-3. EPIC register summary (continued)

Address offset from EPIC base	Use	Section/page
140	EPIC interrupt dispatcher selector register 16	7.3.3.2/14
144	EPIC interrupt dispatcher selector register 17	7.3.3.2/14
148	EPIC interrupt dispatcher selector register 18	7.3.3.2/14
14C	EPIC interrupt dispatcher selector register 19	7.3.3.2/14
150	EPIC interrupt dispatcher selector register 20	7.3.3.2/14
154	EPIC interrupt dispatcher selector register 21	7.3.3.2/14
158	EPIC interrupt dispatcher selector register 22	7.3.3.2/14
15C	EPIC interrupt dispatcher selector register 23	7.3.3.2/14
160	EPIC interrupt dispatcher selector register 24	7.3.3.2/14
164	EPIC interrupt dispatcher selector register 25	7.3.3.2/14
168	EPIC interrupt dispatcher selector register 26	7.3.3.2/14
16C	EPIC interrupt dispatcher selector register 27	7.3.3.2/14
170	EPIC interrupt dispatcher selector register 28	7.3.3.2/14
174	EPIC interrupt dispatcher selector register 29	7.3.3.2/14
178	EPIC interrupt dispatcher selector register 30	7.3.3.2/14
17C	EPIC interrupt dispatcher selector register 31	7.3.3.2/14
180	EPIC interrupt dispatcher target register 0	7.3.3.3/16
184	EPIC interrupt dispatcher target register 1	7.3.3.3/16
188	EPIC interrupt dispatcher target register 2	7.3.3.3/16
18C	EPIC interrupt dispatcher target register 3	7.3.3.3/16
190	EPIC interrupt dispatcher target register 4	7.3.3.3/16
194	EPIC interrupt dispatcher target register 5	7.3.3.3/16
198	EPIC interrupt dispatcher target register 6	7.3.3.3/16
19C	EPIC interrupt dispatcher target register 7	7.3.3.3/16
1A0	EPIC interrupt dispatcher target register 8	7.3.3.3/16
1A4	EPIC interrupt dispatcher target register 9	7.3.3.3/16
1A8	EPIC interrupt dispatcher target register 10	7.3.3.3/16
1AC	EPIC interrupt dispatcher target register 11	7.3.3.3/16
1B0	EPIC interrupt dispatcher target register 12	7.3.3.3/16
1B4	EPIC interrupt dispatcher target register 13	7.3.3.3/16
1B8	EPIC interrupt dispatcher target register 14	7.3.3.3/16

Table 7-3. EPIC register summary (continued)

Address offset from EPIC base	Use	Section/page
1BC	EPIC interrupt dispatcher target register 15	7.3.3.3/16
1C0	EPIC edge/level trigger register 0	7.3.3.4/16
1C4	EPIC edge/level trigger register 1	7.3.3.4/16
1C8	EPIC edge/level trigger register 2	7.3.3.4/16
1CC	EPIC edge/level trigger register 3	7.3.3.4/16
1D0	EPIC edge/level trigger register 4	7.3.3.4/16
1D4	EPIC edge/level trigger register 5	7.3.3.4/16
1D8	EPIC edge/level trigger register 6	7.3.3.4/16
1DC	EPIC edge/level trigger register 7	7.3.3.4/16
1E0	EPIC interrupt pending register 0	7.3.3.5/19
1E4	EPIC interrupt pending register 1	7.3.3.5/19
1E8	EPIC interrupt pending register 2	7.3.3.5/19
1EC	EPIC interrupt pending register 3	7.3.3.5/19
1F0	EPIC interrupt pending register 4	7.3.3.5/19
1F4	EPIC interrupt pending register 5	7.3.3.5/19
1F8	EPIC interrupt pending register 6	7.3.3.5/19
1FC	EPIC interrupt pending register 7	7.3.3.5/19
200	EPIC interrupt pending register 8	7.3.3.5/19
204	EPIC interrupt pending register 9	7.3.3.5/19
208	EPIC interrupt pending register 10	7.3.3.5/19
20C	EPIC interrupt pending register 11	7.3.3.5/19
210	EPIC interrupt pending register 12	7.3.3.5/19
214	EPIC interrupt pending register 13	7.3.3.5/19
218	EPIC interrupt pending register 14	7.3.3.5/19
21C	EPIC interrupt pending register 15	7.3.3.5/19
220	EPIC enable/disable interrupts register 0	7.3.3.6/20
224	EPIC enable/disable interrupts register 1	7.3.3.6/20
228	EPIC enable/disable interrupts register 2	7.3.3.6/20
22C	EPIC enable/disable interrupts register 3	7.3.3.6/20
230	EPIC enable/disable interrupts register 4	7.3.3.6/20
234	EPIC enable/disable interrupts register 5	7.3.3.6/20

Table 7-3. EPIC register summary (continued)

Address offset from EPIC base	Use	Section/page
238	EPIC enable/disable interrupts register 6	7.3.3.6/20
23C	EPIC enable/disable interrupts register 7	7.3.3.6/20
240	EPIC enable/disable interrupts register 8	7.3.3.6/20
244	EPIC enable/disable interrupts register 9	7.3.3.6/20
248	EPIC enable/disable interrupts register 10	7.3.3.6/20
24C	EPIC enable/disable interrupts register 11	7.3.3.6/20
250	EPIC enable/disable interrupts register 12	7.3.3.6/20
254	EPIC enable/disable interrupts register 13	7.3.3.6/20
258	EPIC enable/disable interrupts register 14	7.3.3.6/20
25C	EPIC enable/disable interrupts register 15	7.3.3.6/20
260	EPIC software induced interrupt	7.3.3.7/21
264	EPIC disable interrupts register	7.3.3.8/21

7.3.3 Register descriptions

This section describes epic registers in detail.

7.3.3.1 EPIC Interrupt Priority Level Registers (P_IPL*x*)

P_IPL*x* (where *x* is a value of 63–1) define the IPL for four groups interrupts in each register and specify whether the group is non critical or critical (can be regarded as priority 33) as shown in Figure 7-3. Before an IPL of an edge-triggered interrupt is changed from 0 to another value, it must be cleared (in P_IPR*x*), because the EPIC keeps track of interrupts with IPL0 to enable recording their history during the period when the IPL is 0 (interrupt is not effective).

Address 0x000 (P_IPL_<y>) offset 4 range y=0..1 — Access: R

	31	30	29	28	24	23	22	21	20	15	14	13	12	8	7	6	5	4	0
R	CId	—		IPLd		CIc	—		IPLc	CIb	—			IPLb	CIa	—		IPLa	
W																			
Reset	1	0	0	0	0 0 0 0	1	0	0	0 0 0 0 0	1	0	0	0	0 0 0 0	1	0	0	0	0 0 0 0

Figure 7-2. EPIC Interrupt Priority Level Register 0 (P_IPL_0)

addre ss 0x010 (P_IPL_<i>) offset 4 range i=2..63 — Access: RW

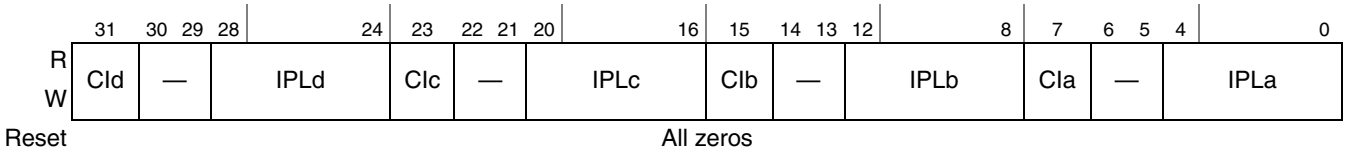

Figure 7-3. EPIC Interrupt Priority Level Register i(P_IPL_i)

Table 7-4. Register P_IPL_<*i*> (i=1..63) bits descriptions

Name	Description	Settings
CIy 31, 23, 15, 7	Critical or nonCritical interrupt configuration	0 NonCritical (IR) 1 Critical interrupt (CI)
IPLy 28–24 20–16 12–8 4–0	Priority level for IR input y. Defines the interrupt priority level (IPL). IPL 31 is the highest priority. If the value is 0, interrupts are disabled on this input. When CIy is set this field has no effect. The SC3900 DSP subsystem internal CIs are set as CIs and cannot be configured, so P_IPL0 has an unchangeable value of 0x8080_8080.	00000 -IPL0 (IR is off) 00001 -IPL1 (lowest priority) 00010 -IPL2 11111 -IPL31 (highest priority)

Table 7-5 map the interrupt group. Each group index maps two interrupt 2*Index and 2*index + 1.

Table 7-5. P_IPL_*x* interrupt group mapping

Register Index	Field d	Field c	Field b	Field a
	Read-only field			
0	3	2	1	0
1	7	6	5	4
2	11	10	9	8
3	15	14	13	12
4	19	18	17	16
5	23	22	21	20
6	27	26	25	24
7	31	30	29	28
8	35	34	33	32
9	39	38	37	36
10	43	42	41	40
11	47	46	45	44
12	51	50	49	48
13	55	54	53	52
14	59	58	57	56

Table 7-5. P_IPL_*x* interrupt group mapping (continued)

Register Index	Field d	Field c	Field b	Field a
15	63	62	61	60
16	67	66	65	64
17	71	70	69	68
18	75	74	73	72
19	79	78	77	76
20	83	82	81	80
21	87	86	85	84
22	91	90	89	88
23	95	94	93	92
24	99	98	97	96
25	103	102	101	100
26	107	106	105	104
27	111	110	109	108
28	115	114	113	112
29	119	118	117	116
30	123	122	121	120
31	127	126	125	124
32	131	130	129	128
33	135	134	133	132
34	139	138	137	136
35	143	142	141	140
36	147	146	145	144
37	151	150	149	148
38	155	154	153	152
39	159	158	157	156
40	163	162	161	160
41	167	166	165	164
42	171	170	169	168
43	175	174	173	172
44	179	178	177	176
45	183	182	181	180
46	187	186	185	184
47	191	190	189	188

Table 7-5. P_IPL_*x* interrupt group mapping (continued)

Register Index	Field d	Field c	Field b	Field a
48	195	194	193	192
49	199	198	197	196
50	203	202	201	200
51	207	206	205	204
52	211	210	209	208
53	215	214	213	212
54	219	218	217	216
55	223	222	221	220
56	227	226	225	224
57	231	230	229	228
58	235	234	233	232
59	239	238	237	236
60	243	242	241	240
61	247	246	245	244
62	251	250	249	248
63	255	254	253	252

7.3.3.2 EPIC Interrupt Dispatcher Selector Register (P_DISP*x*)

P_DISP*x* (where *x* is a value of 63–0) define the Dispatcher selector for eight groups of interrupt. Each field specify which dispatcher target should be associated with each group of interrupt.

addre ss 0x100 (P_DISP_<i>) offset 4 range 0..31 — Access: RW

	31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
R W	DISPh	DISPg	DISPf	DISPe	DISPd	DISPc	DISPb	DISPa
Reset	All zeros							

Figure 7-4. EPIC Interrupt Dispatcher Selector Register *x* (P_DISP_*x*)

This table maps the interrupt group index. Each group index maps 2 interrupts: 2*(index), 2*(index)+1.

Table 7-6. P_DISP_x bit descriptions

Name	Description	Settings
DISPy 31-28 27–24 23–20 19–16 15-12 11-08 7-4 3-0	The DISPy defines to which dispatcher handler the interrupt is associated.	0000 -select dispatcher 0 0001 -select dispatcher 1 0010 -select dispatcher 2 ... 1111 -select dispatcher 15

Table 7-7. P_DISP_x interrupt mapping

Register index	Field h	Field g	field f	Field e	Field d	Field c	Field b	Field a
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56
8	71	70	69	68	67	66	65	64
9	79	78	77	76	75	74	73	72
10	87	86	85	84	83	82	81	80
11	95	94	93	92	91	90	89	88
12	103	102	101	100	99	98	97	96
13	111	110	109	108	107	106	105	104
14	119	118	117	116	115	114	113	112
15	127	126	125	124	123	122	121	120
16	135	134	133	132	131	130	129	128
17	143	142	141	140	139	138	137	136
18	151	150	149	148	147	146	145	144
19	159	158	157	156	155	154	153	152
20	167	166	165	164	163	162	161	160
21	175	174	173	172	171	170	169	168

Table 7-7. P_DISP_x interrupt mapping (continued)

Register index	Field h	Field g	field f	Field e	Field d	Field c	Field b	Field a
22	183	182	181	180	179	178	177	176
23	191	190	189	188	187	186	185	184
24	199	198	197	196	195	194	193	192
25	207	206	205	204	203	202	201	200
26	215	214	213	212	211	210	209	208
27	223	222	221	220	219	218	217	216
28	231	230	229	228	227	226	225	224
29	239	238	237	236	235	234	233	232
30	247	246	245	244	243	242	241	240
31	255	254	253	252	251	250	249	248

7.3.3.3 EPIC Interrupt Dispatcher Target Register (P_TRG_*x*)

P_TRG*x* (where *x* is a value of 15–0) define the Dispatcher handler target (full virtual address) that the core should jump to when serving the interrupt.

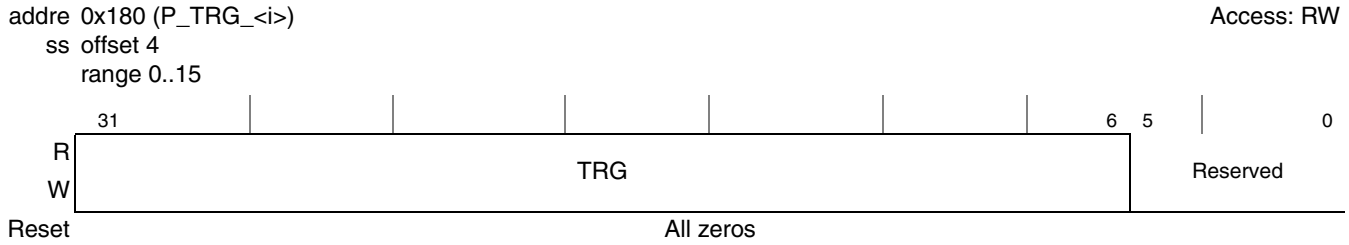

Figure 7-5. EPIC Interrupt Dispatcher Selector Register *x* (P_TRG*x*)

Table 7-8. P_TRGx bit descriptions

Name	Description	Settings
TRGy 31-6	The TRGx defines the target (dispatcher handler).	—

7.3.3.4 EPIC Edge/Level Trigger Registers (P_ELR_*x*)

P_ELR*x* (where *x* can be a value of 7–0) configure the interrupt capturing mechanism to detect edge-triggered or level-triggered interrupts. The configuration should match the interrupt type as generated by the interrupt source. A value of 1 in the relevant bit configures the interrupt to be treated as edge-triggered. The EPIC detects the positive edge of the interrupt and, only after the interrupt is cleared in the P_IPR*x*, is it negated internally. Clearing the relevant bit configures the interrupt as level-triggered, and the EPIC reflects the state of the interrupt in the input as it changes. For edge-triggered interrupts

without the override logic, subsequent interrupt requests that are asserted before the user clears the P_IPR*x* bit are lost. Therefore, the interrupt is serviced only once for the first event that sets the P_IPR*x* bit.

The SC3900 DSP subsystem internal interrupts (not including the reserved interrupts) are hard configured to their appropriate edge/level triggering mode and cannot be changed. The value of the P_ELR0 at reset is 0xFFFF_FFFF. The P_ELR0 register is shown in this figure.

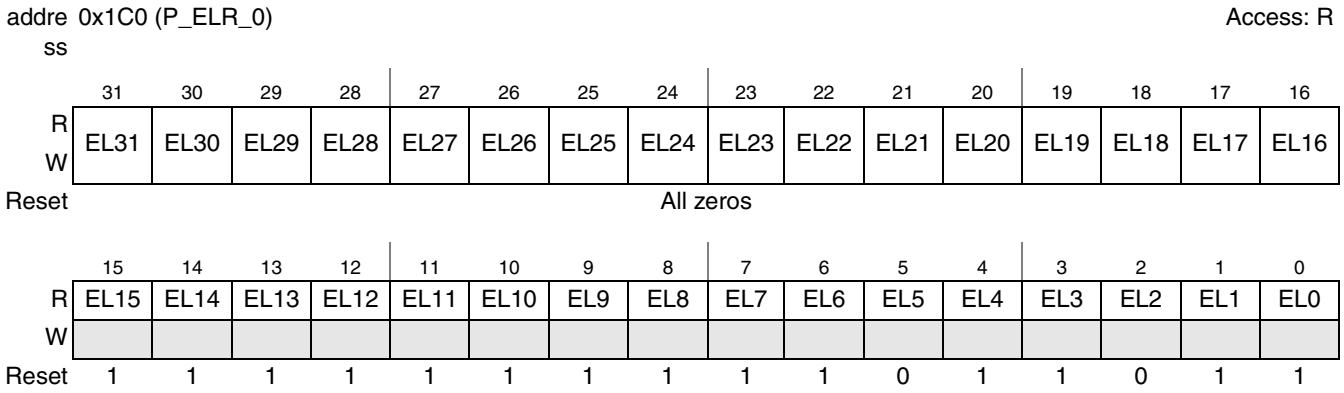

Figure 7-6. EPIC Edge/Level Register 0 (P_ELR_0)

Figure 7-7. EPIC Edge/Level Register *x* (P_ELR_*x*)

This table shows the P_ELRx bit descriptions.

Table 7-9. P_ELR_*x* bit descriptions

Name	Description	Settings
ELxx 31–0	Triggering Mode. each bit controls a group of two interrupts.	0 Level-triggered interrupt 1 Positive edge-triggered interrupt

Table 7-10 shows the interrupt specified by a particular register index (*x*) and bit index. each bit controls the triggering mode of group of two interrupts: 2*(bit index), 2*(bit index)+1 for example bit 31 in register 7 controls triggering mode of interrupts 510, 511.

Table 7-10. P_ELR_*x* mapping

Bit Index/Reg Index	0	1	2	3	4	5	6	7
0	0	32	64	96	128	160	192	224
1	1	33	65	97	129	161	193	225
2	2	34	66	98	130	162	194	226
3	3	35	67	99	131	163	195	227
4	4	36	68	100	132	164	196	228
5	5	37	69	101	133	165	197	229
6	6	38	70	102	134	166	198	230
7	7	39	71	103	135	167	199	231
8	8	40	72	104	136	168	200	232
9	9	41	73	105	137	169	201	233
10	10	42	74	106	138	170	202	234
11	11	43	75	107	139	171	203	235
12	12	44	76	108	140	172	204	236
13	13	45	77	109	141	173	205	237
14	14	46	78	110	142	174	206	238
15	15	47	79	111	143	175	207	239
16	16	48	80	112	144	176	208	240
17	17	49	81	113	145	177	209	241
18	18	50	82	114	146	178	210	242
19	19	51	83	115	147	179	211	243
20	20	52	84	116	148	180	212	244
21	21	53	85	117	149	181	213	245
22	22	54	86	118	150	182	214	246
23	23	55	87	119	151	183	215	247
24	24	56	88	120	152	184	216	248
25	25	57	89	121	153	185	217	249
26	26	58	90	122	154	186	218	250
27	27	59	91	123	155	187	219	251
28	28	60	92	124	156	188	220	252
29	29	61	93	125	157	189	221	253

Table 7-10. P_ELR_*x* mapping (continued)

Bit Index/Reg Index	0	1	2	3	4	5	6	7
30	30	62	94	126	158	190	222	254
31	31	63	95	127	159	191	223	255

7.3.3.5 EPIC Interrupt Pending Registers (P_IPR_*x*)

P_IPR*x* (where *x* is a value from 15–0) monitor pending interrupts and clear edge-triggered interrupts. The user can read the P_IPR*x* to view the status of all current IRs and CIs. Each bit in the registers represents one of the 512 input interrupt requests. If an interrupt is configured as level-triggered, its corresponding IP bit reflects the status of the input IR signal. When the interrupt is configured as edge-triggered the corresponding IP bit is set when an edge is detected. To clear (acknowledge) the IR or CI, a value of one is written to the corresponding bit in the register. Clearing a bit has no effect on its status. This feature is used for both IRs and CIs to indicate to the EPIC that the SC3900 core has acknowledged the corresponding edge-triggered interrupt source. The relation between each bit index (31–0) to the matching interrupt is shown in Table 7-10.

addre ss 0x1E0 (P_IPR_<i>) offset 4 range 0..15 — Access: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	IPR31	IPR30	IPR29	IPR28	IPR27	IPR26	IPR25	IPR24	IPR23	IPR22	IPR21	IPR20	IPR19	IPR18	IPR17	IPR16
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c
Reset	All zeros															

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	IPR15	IPR14	IPR13	IPR12	IPR11	IPR10	IPR9	IPR8	IPR7	IPR6	IPR5	IPR4	IPR3	IPR2	IPR1	IPR0
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c
Reset	All zeros															

Figure 7-8. EPIC Interrupt Pending Registers (P_IPR*x*)

Bit Index/Register Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480
1	1	33	65	97	129	161	193	225	257	289	321	353	385	417	449	481
2	2	34	66	98	130	162	194	226	258	290	322	354	386	418	450	482
3	3	35	67	99	131	163	195	227	259	291	323	355	386	419	451	483
4	4	36	68	100	132	164	196	228	260	292	324	356	387	420	452	484
5	5	37	69	101	133	165	197	229	261	293	325	357	389	421	453	485
6	6	38	70	102	134	166	198	230	262	294	326	356	390	422	454	486

Bit Index/Register Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	7	39	71	103	135	167	199	231	263	295	327	359	391	423	455	487
8	8	40	72	104	136	168	200	232	264	296	328	360	392	424	456	488
9	9	41	73	105	137	169	201	233	265	297	329	361	393	425	457	489
10	10	42	74	106	138	170	202	234	266	298	330	362	394	426	458	490
11	11	43	75	107	139	171	203	235	267	299	331	363	395	427	459	491
12	12	44	76	108	140	172	204	236	268	300	332	364	396	428	460	492
13	13	45	77	109	141	173	205	237	269	301	333	365	397	429	461	493
14	14	46	78	110	142	174	206	238	270	302	334	366	398	430	462	494
15	15	47	79	111	143	175	207	239	271	303	335	367	399	431	463	495
16	16	48	80	112	144	176	208	240	272	304	336	368	400	432	464	496
17	17	49	81	113	145	177	209	241	273	305	337	369	401	433	465	497
18	18	50	82	114	146	178	210	242	274	306	338	370	402	434	466	498
19	19	51	83	115	147	179	211	243	275	307	339	371	403	435	467	499
20	20	52	84	116	148	180	212	244	276	308	340	372	404	436	468	500
21	21	53	85	117	149	181	213	245	277	309	341	373	405	437	469	501
22	22	54	86	118	150	182	214	246	278	310	342	374	406	438	470	502
23	23	55	87	119	151	183	215	247	279	311	343	375	407	439	471	503
24	24	56	88	120	152	184	216	248	280	312	344	376	408	440	472	504
25	25	57	89	121	153	185	217	249	281	313	345	377	409	441	473	505
26	26	58	90	122	154	186	218	250	282	314	346	378	410	442	474	506
27	27	59	91	123	155	187	219	251	283	315	347	379	411	443	475	507
28	28	60	92	124	156	188	220	252	284	316	348	380	412	444	476	508
29	29	61	93	125	157	189	221	253	285	317	349	381	413	445	477	509
30	30	62	94	126	158	190	222	254	286	318	350	382	414	446	478	510
31	31	63	95	127	159	191	223	255	287	319	351	383	415	447	479	511

7.3.3.6 EPIC Enable/Disable Interrupts Registers (P_ENDIS_*x*)

P_ENDIS*x* (where *x* can be a value of 15–0) enables/disables each of the 512 interrupts. Setting a bit enables the interrupt. Note that the reset value of this register is all zeros. P_ENDIS*x* makes it convenient to enable/disable interrupts, and supports fast software priority handling of IRQs. Disabling an interrupt is equivalent to setting it to IPL 0, except that the disable also has an effect on CIs. Setting IPL 0 to an CI has no effect. In functional mode, the interrupt is recorded in the relevant bit of the P_IPR*x* but is not passed to the core. A disabled pending interrupt does not prevent enabled interrupts with a lower priority from being reflected to the core. Before an edge-triggered interrupt is enabled, it must be cleared (in P_IPR*x*) if

its history is not relevant because the EPIC records disabled interrupts. The relation between each bit index (31–0) to the matching interrupt same as P_IPR_x Table.

addre ss: Ox220 (P_ENDIS_<i>) offset 4 range 0..15 — Access: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	ED31	ED30	ED29	ED28	ED27	ED26	ED25	ED24	ED23	ED22	ED21	ED20	ED19	ED18	ED17	ED16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	ED15	ED14	ED13	ED12	ED11	ED10	ED9	ED8	ED7	ED6	ED5	ED4	ED3	ED2	ED1	ED0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7-9. EPIC Enable/Disable Interrupts Registers (P_ENDIS_*x*)

7.3.3.7 EPIC SW induced interrupt Register (P_SWII)

addre ss: Ox260 (P_SWII) — Access: RW

	31	30														8														0
R W	set	clr																											Index	
Reset	All zeros																													

Figure 7-10. EPIC Disable Interrupts Register (P_SWII)

The SW induced register hold the index on the interrupt that the programer request to induce and a set bit. The programmer can set any of the 512 interrupts by SkyBlue transaction that specify the interrupt index and a set bit for go.

In order to assert interrupt by software set interrupt index and assert set bit to “1”.

In order to negate it (for level interrupt) - write “1” to clr bit.

7.3.3.8 EPIC Disable Interrupts Register (P_DI)

addre ss: Ox264 (P_DI) — Access: RW

	31																											1	0
R W	—																												DI
Reset	All zeros																												

Figure 7-11. EPIC Disable Interrupts Register (P_DI)

The P_DI hold the DI bit which with one write access can disable all the EPIC controlled interrupts (even CI) from being passed to the core. This bit is useful for ensuring an interrupt safe region in the code. It is stronger than the core DI instruction as it also blocks CIs which the core DI does not. An example for using this bit is a save routine for S&R PG sequence that wants to guarantee no interruption from a certain point. Setting the bit to one disables all the interrupt.

Chapter 8
Debug and Trace Support

8.1 Overview of the debug and trace functions

This section describes the features in the SC3900 subsystem for supporting debug and trace. The main concepts of the SC3900 subsystem debug support are:

- Debug configuration by the external host is done via a generic slave bus (Sky Blue); all configuration is done by accessing memory mapped registers.
- Three debug management schemes, as follows:
  - — By an external debug host, activating the debug functions by accessing the debug register through the slave port
    - – Static configuration during debug mode, when it has full access to all core and memory resources
    - – Dynamic configuration while the core is running—in limited use-cases
  - — By an on-core software monitor, which manages the debug functions during debug exceptions, in limited use cases.
- The external host and the monitor can run concurrently, sharing the hardware debug resources between them.
- Support for multi-core, real time tracing, based on the Nexus standard (IEEE ISTO 5001), with an independent trace output bus.

The logic supporting debug and trace in SC3900 subsystem includes of the following resources:

- Dedicated core instructions
- A dedicated, configurable debug exception vector
- An address detection unit which includes:
  - — 4 general purpose event detection groups, each able to support up to two PC-only breakpoints/events, or up to two PC ranges, or two data address range (or with some combinations)
  - — One task ID filter
  - — One exception detector
- A profiling unit, which contains:
  - — Six 32-bit profiling counters, organized in 2 triads
  - — One reloadable counter, for debug event control and validating real-time intervals
- An indirect event unit, supporting 4 event-generating state bits
- A trace unit, which supports (partial list):
  - — Program flow trace, with exact cycle measurement
  - — Profiling counter tracing
  - — Watchpoint messages
  - — User defined trace messages (data acquisition), based on writing to dedicated core registers

The following debug functions are handled by logic in the SoC, conforming to the QorIQ conventions:

- The JTAG interface is handled at the SoC level. JTAG transactions by the external host are translated to memory accesses on the Sky Blue bus. Note that the SC3900 core is not aware what is the interface the debug host works through
- A central run control and event cross triggering blocks which controls debug request and resume signals per debug target, as well as SoC debug event selection and cross triggering
- Central collection and handling of the Nexus trace messages from all cores and trace generators

Activating some of this SoC logic is essential for managing a debug session on the SC3900 core. For a description of the SoC-level debug logic, see the SoC documentation.

8.1.1 Debug components at the cluster and subsystem level

The debug and profiling high level block diagram at the SC3900 cluster and subsystem level is illustrated in Figure 8-1. The illustration is of a two subsystem cluster.

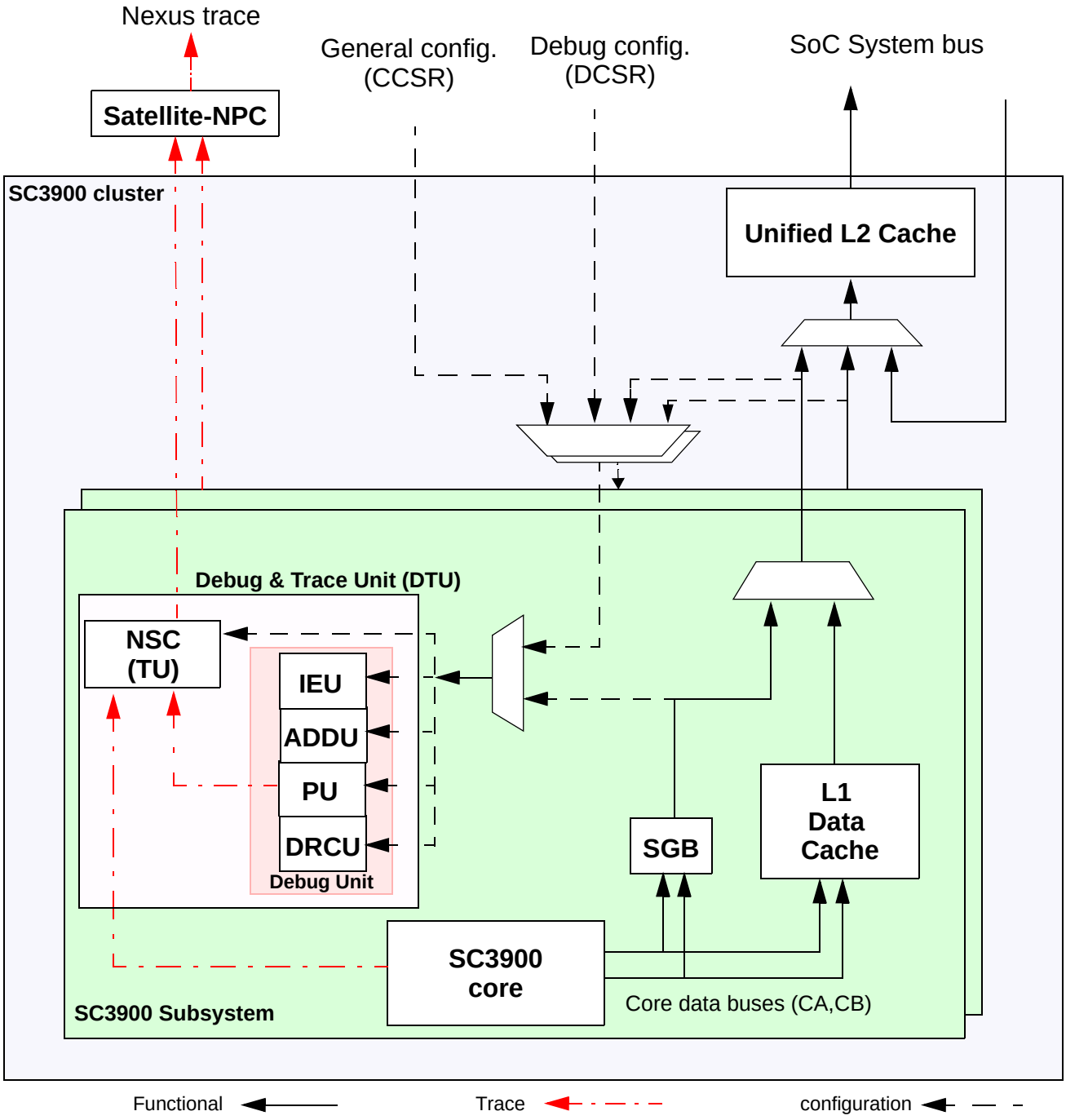

Figure 8-1. SC3900 subsystem and cluster debug and trace logic block diagram

The Debug and Trace Unit (DTU) is composed of two main subunits:

- The Debug Unit (DU) includes logic that manages run control, profiling, event detection and cross triggering. The DU includes the following functional blocks:
  - The Debug Run Control Unit (DRCU) is responsible to control the core on issues like entry and exit from debug, core command insertion, and single stepping.
  - The Address and Data Detection Unit (ADDU) is responsible to detect events, based on the match on the PC, data address, and the ID of an exception starting to execute. The output events can be used to set breakpoints and watchpoints, or to cause other outcomes.
  - Indirect Event Unit (IEU) is a unit that collects all debug events and allows the user to flexibly monitor them, to detect complex scenarios and sequences, and to generate debug events in response.
  - The Profiling Unit (PU) includes a set of counters that can be configured to count between a large number of input events from the subsystem and the core.
- The Nexus StarCore Client (NSC), also referred to as the Trace Unit (TU), is responsible for generating, using the Nexus (IEEE ISTO 5001) standard, trace messages based on core and subsystem events.

The functional blocks include debug resources that are named “Debug Event Generators” (DEG), and Debug filtering units (also referred to as “filtering units”). A DEG is a resource that can generate a debug event, and can be individually enabled by the user. Debug filter units are debug resources that can be individually enabled, but cannot generate an event on their own. They are used in conjunction with DEGs, which limits the events they can generate. Note that in some cases, events generated by DEGs can also be used as filters (for more information, see Section 8.3, “Debug events and filters”).

This table provides a list of the Debug Event Generators in the DTU.

Table 8-1. Debug Event Generators in the DTU

Unit	Debug Event Generators
DRCU	• Debug mode entry control • Debug exception entry control • Four interface event signals
ADDU	• 4 PC/Data Address detector groups (each can detect 2 exact PCs, or 2 PC/data address range) • 1 exception detector
IEU	4 state bits
PU	• 6 profiling counters • One reloadable counter

Debug filtering units in the DTU (in the ADDU):

- One task ID filter

8.2 Debug session management

The SC3900 supports two schemes of managing a debug session:

- Debug by an external host
- Debug by an on-core monitor software

An external host is any element (other than the target core) that can master the system bus and configure the DTU register bank through the debug configuration port in the DCSR memory space. It does not matter if it is an off-SoC host working through the JTAG port, or if it is any other unit that can master the SoC system bus (for example, another core in the SoC).

A debug monitor is a software driver that manages all debug activities while running on the debug core target itself. This driver is pre-programmed to perform debug tasks such as breakpoint insertion, event management, and so on. The monitor software operates during debug exceptions. Debug exceptions have a dedicated stack pointer (DSP) and link register (LR2) both of which minimize interference with core resources used by the application, and help having a short latency to activate a debug monitor function.

Some of the actions that can be performed by an external host can be performed equivalently by the monitor. However, the detailed implementation of each action can be different. For example, instruction jamming with core commands can be done only in debug mode. The equivalent action in an exception-based session would be to jump to an agreed, non-cacheable location in memory and perform the code written at that location. Some actions could be done only by an external host while the core is in debug mode; for example, the ability to probe the detailed internal state of caches and change it (including tags, valid bits, and so on). This action actively uses the caches and is not supported while the core is running.

The SC3900 supports the monitor and the external host managing debug sessions concurrently. This is done by allowing them to allocate the debug resources of the DTU between them in a way that they do not interfere with each other. Some of these resources (for example, the debug instructions) are rigidly allocated to a debug management scheme. Others (for example, PC detectors) are allocated under software control, and rely on a hardware-assisted partition of the debug resources between the two session managers. The DTU can be seen as having two partitions, each of which holds resources supporting one of these session management schemes.

The support in hardware for resource partition is as follows:

- All resources of a partition can be enabled or disabled together by way of global signals from the SoC. See Section 8.2.2, “Enabling and disabling the DTU.”
- Control registers of resources that are allocated to a partition cannot be written to by the other session manager while the core is running.

Debug resources (DEGs and filter units) that can be configured to belong to either partition are as follows:

- Address and data detector groups (each group individually)
- Indirect event states (each state individually)
- The task ID filter
- Exception detector
- Profiling counters (all together)

- The reloadable counter
- Trace unit
- Interface event signals to the SoC (each one individually)

8.2.1 DTU resource partitioning

The resources that could be allocated to a partition could be seen as belonging to a shared bank of resources. When not in debug mode, the host and monitor have more or less equal rights, and one cannot use or modify the debug resources that are not allocated to it. Each debug session manager (host or monitor) can request a free resource by posting a request in a dedicated register (one register per each partition: DHRRR for the host, and DMRRR for the monitor). Each allocatable resource is represented by a bit in these registers. An allocatable resource can be in one of the following three states:

- not allocated,
- allocated to the external host,
- or allocated to the monitor.

This table describes the way these states are reflected (per resource) in the DHRRR and DMRRR.

Table 8-2. DTU resource allocation states, per resource

Allocation state of the resource	State of the resource bit in DHRRR	State of the resource bit in DMRRR
Not allocated	0	0
Allocated to the external host	1	0
Allocated to the monitor	0	1

To add a resource to its partition, a debug session manager should follow this procedure:

1. Read the value of the respective bit in the resource reservation register of the partition and identify a resource that is not yet allocated to the partition (having a value of 0). A value of 1 means the resource is already allocated to this partition.
2. Write 1 to this bit, which means a request for allocation.
3. Read back the value of the bit.
   — If the value that was read is 0, the resource is allocated to the other partition and cannot be taken.
   — If the value is 1, the resource is allocated to the requestor.

Resource partitioning should be static; a resource should be allocated either to the host or the monitor when the debug session is established, and remain so until the next reset.

The DHRRR and DMRRR are each read and written as one entity, not bit by bit. When writing the register, bits of resources that should not change their allocation state should be written back with the same value with which they were read (writing back 1 to an allocated resource bit keeps it allocated). The hardware ensures that a resource is owned by one partition at most. When the two session managers ask for the same resource at the same cycle, the resource is allocated to the external host. For more information on these

registers, see Section 8.5.6.9, “Debug Host Resource Reservation Register (DHRRR),” and Section 8.5.6.14, “Debug Monitor Resource Reservation Register (DMRRR).”

The DTU resources that are allocated to the external host, as well as those allocated to the monitor, are usually accessible from different memory spaces. For more information, see Section 8.4, “Debug and trace register map.”

8.2.2 Enabling and disabling the DTU

The Debug and Trace Unit is disabled by default. When it is disabled, debug support instructions are executed either as NOP or as the cause of an illegal exception (see Table 8-14). DTU registers cannot be accessed; debug event generation and tracing is also inactive. Enabling and disabling the DTU is performed by writing to a register in the SoC. To enable access to the DTU registers, first enable the DTU clock. When the DTU clock is disabled, the user cannot read from or write to the DTU registers. Read accesses will return undefined data, and write accesses will have no effect[1]. An access error indication will be returned to the originator of the access. To save power when there is no need for debug functions, keep the clocks disabled. The DTU clock must be activated before configuring the DTU and activating debug functionality.

Each partition in the DTU is enabled by an independent signal controlled from an SoC control register. Assert the enable signal for a DTU partition to enable writing by the respective session manager to the DTU control registers that are allocated to this partition and to enable the generation of events controlled by these DTU control registers.

Once activated, the DTU clock enable, the host partition enable, and the monitor partition enable should not be disabled until the next reset. The DTU clock enable must be asserted before or with the host enable and monitor enable signals. All three signals can be asserted at once.

This table summarizes the enabled functionality in each activation state.

Table 8-3. Activation signals of the Debug Unit from the SoC

DTU state	DTU register access by the respective debug master	Register state when entering this state	Core functionality	DTU functionality
DTU clock disabled	Any register access returns an error.	Reset value	Same as when host disabled and monitor disabled	DTU Disabled
DTU host disabled (with clocks enabled)	Reads are supported, writes return an error.	Reset value	Cannot enter debug mode SWB causes Illegal exception DEBUGM.*n* treated as NOP	Host partition is disabled.

1. In previous StarCore generations, the EOnCE or OCE included registers that may have been needed by the application code, such as the register holding the version number of the core. In SC3900, these registers were all moved elsewhere.

Table 8-3. Activation signals of the Debug Unit from the SoC (continued)

DTU state	DTU register access by the respective debug master	Register state when entering this state	Core functionality	DTU functionality
DTU host enabled	Reads are supported, writes are supported to registers in resources allocated to the host partition.	Reset value	Core can enter debug mode by SWB or any DTU event configured to do so DEBUGM.*n* cause entry to debug mode if configured	Host partition is enabled. Debug resources could be allocated to the partition and operate as configured.
DTU monitor disabled (with clocks enabled)	Reads are supported, writes return an error.	Reset value	Cannot enter debug exception DEBUGE.*n* treated as NOP	The monitor partition is disabled
DTU monitor enabled	Reads are supported, writes are supported to registers in resources allocated to the host partition.	Reset value	Core can enter debug exception by any DTU event configured to do so DEBUGE.*n* cause entry to debug exception if configured	The monitor partition is enabled. Debug resources could be allocated to the partition and operate as configured.

The SoC control register controlling these inputs is also accessible by an SoC master while the core is held before the first fetch; thus, the DTU could be enabled and configured as required from the first program fetch.

8.2.3 Debug configuration by an external host

Configuration by an external host can take place while the core is running or halted (in idle state or in debug mode, see Section 8.5.2, “Debug mode”).

8.2.3.1 Debug configuration when the core is halted

The DTU registers can be configured when the core is in debug mode or when held before the first fetch. The core may be held by the SoC before it issues the first program fetch of the reset address. While the core is in this state, the external master can configure the DTU through the debug configuration port, so that this configuration takes effect from the first fetch the core issues after it is released from reset and starts to execute instructions.

The common flow to configure the debug registers is while the core is halted, as this ensures a known architectural state after which the configuration takes effect; it is also ensured that there is no contention between actions performed by the external host and those performed by the application. This assurance allows minimal limitations on the configuration process and, in general, allows for the performance of the full scope of debug actions. The downside is the need to halt the core, which unless performed before exit from reset, may disrupt the real time relations of the application.

Remember that, when the debug host is a local core or an SoC master, the process of entering the core into debug mode, performing some configuration accesses of debug registers, and exiting debug, may be as low as a few hundred cycles.

8.2.3.2 Debug configuration in the background while the core is running

The DTU registers can be accessed by an off-core host through the debug configuration bus in parallel to the core running the application code. The off-core host can also access all the other memory mapped registers in the SC3900 subsystem (for example, the MMU and cache control registers) by way of the CCSR memory space.

While the core is running, DTU configuration can be performed under the following conditions:

- The host debugger must ensure that during this process there are no debug requests asserted from the SoC.
- The code that the core is running does not include instructions that cause entry into debug mode or an exception (for example, SWB); this is reasonable because these instructions cause an illegal exception when the DTU is not enabled. Note that the DEBUGE.*n* and DEBUGM.*n* instructions are assumed to be disabled, as this is the default DMCSR and DECSR configuration when the DTU is enabled.

If it cannot be guaranteed that there are no such instructions in the code, or that the code is not currently running sections that may encounter them, the core must first be forced into an IDLE state or interrupted to execute a code section where these conditions are ensured before the DTU is enabled.

Once the host partition is enabled, the DTU can be configured from the SoC using the SkyBlue port and the DCSR memory space. However, this configuration must ensure that the outcome events will not be activated during configuration, only afterwards. For example, if the configured state includes several parallel event chains that may interact, there is danger is that portions of the event chains will fire before the configuration is complete because of the order of configuration. This situation may cause a false detection of events.

In this case, a safe configuration involves two safety measures:

- Enable the primary events (events that trigger secondary events) last.
- Condition the primary events to start based on an IEU event, which is triggered by one of the two general debug input events from the SoC. This event will be triggered by the debugger at the SoC level after the configuration is completed.

If it is not ensured that the first debug event the DTU is generating will occur only after configuration is complete, the correctness of the events generated as a result are not guaranteed to be as expected.

8.2.4 Debug configuration by an on-core monitor

The core software can access the DTU registers of the monitor partition when the partition is enabled. In general, it is recommended that the user limit the monitoring functions to debug exceptions (when SR2.IDE is set). This allows for a more robust code partitioning, so that the application code does not depend on the state of the DTU and its enablement. However, in some cases, embedding monitor functions inside the code allows for better performance and functionality; for example, when embedding

user-defined trace functions in the code, or when using OS services for debug-related communication protocols.

Configuring debug registers while the core is running requires careful access ordering and inserting memory barriers when enabling, disabling or changing a configuration.

8.3 Debug events and filters

Debug events are cycle-specific, or VLES-specific events, that are defined by the user and identified by components in the DTU to potentially cause a specific action. A debug event causing an action in the DTU is termed in this chapter an “input event,” the unit which is affected by it is the “target unit,” and the resulting action is the “outcome,” or “output event.” Many of the outcomes can, in turn, be used as input events by another target. Debug events can be generated by components of the DTU, which are called the Debug Event Generators (DEGs). However, there are other sources of debug events; for example, SoC triggers and core operations.

Debug filters are conditions that are used to limit or qualify event detection or outcome generation. Filters cannot generate an outcome on their own. In most cases, filtering conditions are relatively longer term conditions, such as when the core is in exception mode, or matches a certain Task ID value. However, there are cases in which the filtering condition may be VLES-specific. Filter conditions may be generated by dedicated DTU components (filter units). Also, some of the debug events of the DEGs may be used as filters. In addition, in some cases (for example, the attributes of a memory access) some conditions generated outside the DTU can be used as filters.

Event Selection is the general term for configuration of control registers—to define which input events will cause what outcomes. The SC3900 DTU supports the following two general methods for event selection:

Directly	The input event is configured to perform an outcome in the configuration register of the target sub-unit.
Indirectly	The input event modifies dedicated state bits in the Indirect Event Unit (IEU). State transitions are output events that in turn can be configured as inputs to cause outcomes in other units directly.

An example of a direct event is a PC detector, which is configured in a control register in the trace unit to cause tracing to start. An example of an indirect event is an SoC input signal assertion, which is configured to change the state of bit Q0 in the IEU to 1. An event that causes a transition in the state of Q0 can be configured in the control register of Triad A profiling counters to reset them. The Indirect Event Unit is described in detail in Section 8.7, “Indirect events.”

Connecting events directly has the following advantages over indirect connection through the IEU:

- Simpler configuration
- Precise architectural relation between the input and outcome events, when such a relation is applicable, see Section 8.3.1, “Start and stop events.”

However, only a limited set of events can be configured to cause a specific outcome directly. The available input events are described in each control and configuration register of the target sub-unit.

Filters that are defined for a DEG (task ID comparison, Privilege and Mode Filter, etc.) do not affect start and stop events (see Section 8.3.1, “Start and stop events”), they only filter the output events of the unit.

Most DEGs can be configured to respond to several start and stop events. When more than one start or stop events are enabled, the conditions are OR-ed, meaning that the DEG is enabled or disabled if any of the conditions occurred (respectively).

Similarly, most DEGs can be configured with several filter conditions. When more than one filter is enabled, the conditions in general are AND-ed, meaning that only if the event is not filtered by all the enabled filters the outcome occurs. An exception to this principle is if there are several independently enabled filters of the same type, namely:

- Program range filter in the Trace unit. Two PCAD detectors can be configured as filters for several message types, see Section 8.9.4.3, “Trace Control Register 3 (TC3).”

For each of the filter types listed above (independently), the generated event passes the type qualification (for example task ID filtering) if one of the filters of that type approves the event. For example, if an event is conditioned by two PCDA detectors and a task ID comparator, the event is qualified if one of the two PCDA detectors approves it, AND it was qualified by the task ID comparator.

This table summarizes all the events that are used as inputs for event selection.

Table 8-4. Debug events used as inputs for event selection

Type of the input event	Number of events in type	Generating unit	Synchronous?	Direct target for outcome (for a selected subset)
PC and Data Address detection	8	ADDU	Synchronous	DRCU, ADDU, PU, TU, IEU
Servicing an exception	1	core	Synchronous	DRCU, IEU
Interrupt/RTE/subroutine call/return	1	core	Synchronous	PU
DEBUGEV.*n*	8	core	Synchronous	DRCU, ADU, PU, TU, IEU
DEBUGM.*n*, SWB	4+1	core	Synchronous	DRCU
DEBUGE.*n*	4	core	Synchronous	DRCU
Profiling counter overflow	6+2	PU	Asynchronous	DRCU, TU, IEU
Reloadable counter reached zero	1	PU	Asynchronous	DRCU, PU, TU, IEU
Transition of state bit Q*n*	4	IEU	Asynchronous	DRCU, ADU, PU, TU
Assertion of SoC external debug mode request	1	SoC	Asynchronous	DRCU
Assertion of SoC debug event input	2	SoC	Asynchronous	DRCU, IEU

This table summarizes all the conditions used as filters for other events or outcomes.

Table 8-5. Filtering conditions for events and outcomes

Filtering condition	Number of detectors of the condition	Direct targets
PC range detection	4	Data address detection, counter events (all), Trace events (relevant messages)
Privilege and mode	1	counter events (all)
Task ID comparator match	1	ADDU events (all), counter events (all), trace events (all), IEU transitions

Privilege and mode is a filter that appears in the configuration registers of the profiling counters, named PMF (Privilege and Mode Filter). The standard set of values for this filter are described in this table.

Table 8-6. Definition of the privilege level and mode filter

Value	Condition	Meaning	Notes
000	—	No filter	—
100	SR2.EXP = 0 & SR2.IDE = 0	In a task	—
101	SR2.EXP = 1 & SR2.IDE = 0	Exception/OS code	Should not be used with PROCID or PID task filtering. These filters include EXP=0 in their condition.

This table summarizes all the outcomes that could be generated by input debug events.

Table 8-7. Outcomes of debug events

Outcome	Target sub-unit	Number of variants
Entry into debug mode	Core	1
Entry into debug exception	Core	1
Start/stop PC and data address detection	ADDU	8
Start/stop exception detection	ADDU	1
Start/stop count in profiling counters	PU	1
Sample the profiling counters to the shadow registers	PU	1
Reset the profiling counters	PU	1
Start/stop count in the reloadable counter	PU	1
Reload the reloadable counter	PU	1
Start/stop trace	TU	1
Generate a Watchpoint message	TU	1
Generate a program correlation message	TU	1

Table 8-7. Outcomes of debug events (continued)

Outcome	Target sub-unit	Number of variants
Generate a pulse on SoC output signals	DRCU	4
Change state of an IEU state bit	IEU	4

The rest of the section includes summary tables of the allocation of detectors and DEBUGEV.*n* instructions as direct events and filters. This information is also described in the control registers of each target unit, and is listed here for reference.

Note that, when allocating resources to partitions (see Section 8.2.1, “DTU resource partitioning”), all resources that are used for starting, stopping, filtering (and so on) other resources must be allocated together to the same partition. In order to ease this allocation, the definition assumes a symmetric allocation of resources into the two partitions, as specified in Table 8-8. The resources that are allocated to a certain partition in the table are ensured to have all direct cross-triggering connection types consistent within the group. In many cases, there are additional cross-triggering and filtering options from resources in the “other” partition; therefore, the user is not obliged to follow this resources allocation, in particular when not requiring all the cross triggering and filtering options that add dependencies with other resources.

Table 8-8. Self-contained resource allocation in debug resource partitions

Resource	Partition A	Partition B
DEBUGEV.*n* instructions	0-3	4-7
PCDA detectors	0-3	4-7
Exception detector	—	yes
Profiling counters	yes	
Reloadable counter	yes	
Trace Unit	yes	yes

This table summarizes the specific direct outcomes allocated to DEBUGEV.*n* instructions.

Table 8-9. Allocation of DEBUGEV.*n* instructions for direct events

DEBUGEV.*n*	Profiling counters	Reloadable counter	Trace Unit	DRCU - SoC pulse assertion	Counted event in	IEU event
DEBUGEV.0	Start counting	Start counter	Start trace group A	debug_event_out 0	A0, the reload counter	Yes
DEBUGEV.1	Stop counting	Stop counter	Stop trace group A	debug_event_out 1	The reload counter	Yes
DEBUGEV.2	Sample counters	—	—	debug_event_out 2	A1, the reload counter	Yes
DEBUGEV.3	—	—	—	debug_event_out 3	A2, the reload counter	Yes

Table 8-9. Allocation of DEBUGEV.*n* instructions for direct events (continued)

DEBUGEV.*n*	Profiling counters	Reloadable counter	Trace Unit	DRCU - SoC pulse assertion	Counted event in	IEU event
DEBUGEV.4				debug_event_out 0	B0, the reload counter	Yes
DEBUGEV.5				debug_event_out 1	The reload counter	Yes
DEBUGEV.6		—	—	debug_event_out 2	B1, the reload counter	Yes
DEBUGEV.7	—	—	—	debug_event_out 3	B2, the reload counter	Yes

This table summarizes the allocation of PCDA detectors as filters. Unless otherwise specified, the PCDA should be configured as PC range detector, hence only even-indexed PCDA detectors are allocated. This information is also specified in the description of the relevant configuration registers in each affected unit.

Table 8-10. Allocation of PCDA detectors as filters

PCDA detector		Profiling counters	Reloadable counter	Trace Unit
PCDA detector 0		Filter counters	Filter counter	PC range filter
PCDA detector 2	—	—	—	
PCDA detector 4	—			PC range filter
PCDA detector 6	—	—	—	

8.3.1 Start and stop events

Input events can be configured in their configuration registers to dynamically start and stop the operation of most Debug Event Generators. The following terms apply to the TU and to nearly all DEGs in the DTU (such as address detectors, counters, and so on):

Enabled — The DEG was configured and ready to operate without further user intervention. A unit is placed in this state at the end of the configuration process by setting an “Enable” bit in its’ configuration register.

Active — The DEG is enabled, and in addition the dynamic conditions that were pre-configured for it to operate were met. The Active state is reflected by a read-only “Active” status bit in its’ configuration or status register.

When a DEG is active, it performs the function it was configured to do, that is, to issue trace messages, count events, or compare an address in the search for a match with a reference value. If no start event is configured for the DEG, then, by definition, it is active upon enabling it. If a start event is configured for it, then the DEG waits until the start event occurs before it turns active. Similarly, a stop event suspends

the DEG until another start event occurs. This feature enables, for example, the activation of counting only for a specific routine—the start event is a PC detection event on the start address of the routine, and the stop event is another PC detection event on the end address of the routine. This configuration dynamically activates and de-activates the counter whenever the routine is re-executed.

A DEG can be configured to operate without start and stop events (in which case, it turns active upon enabling), or configured with start events, stop events, or both. In most DEGs, it is possible to configure more than one event as a start event, and more than one event to stop. If more than one event is configured to perform a start event, these events are OR-ed; this is the same for stop events.

If a start and a stop event occur on the same cycle, the stop event has precedence. This means that if the DEG was active, it changes its state to Enabled, and if it was Enabled, it does not change its state.

A DEG can be enabled only if it is allocated to a partition (host or monitor). This allocation may be fixed, in which case no allocation programming is required, or it could be configurable, in which case the DEG should be allocated first (see Section 8.2.1, “DTU resource partitioning”). If a DEG is not allocated, it cannot be enabled.

This figure illustrates the state transitions of enabling and activating a DEG.

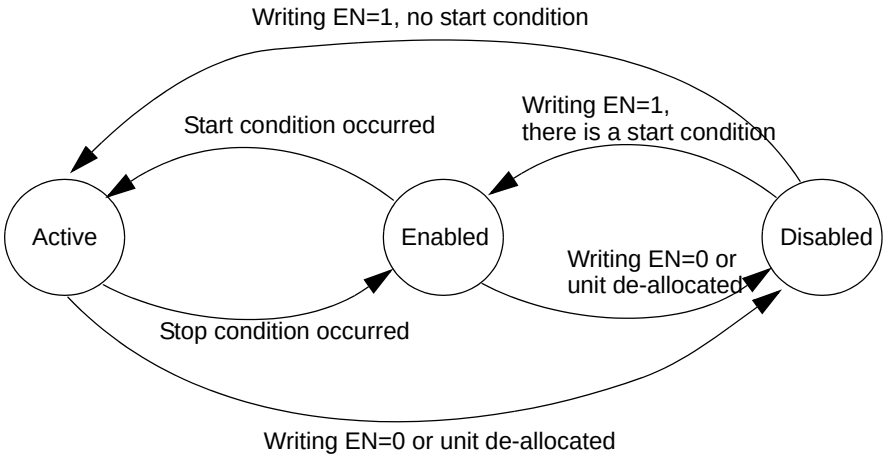

Figure 8-2. Activity states of a debug event generator

Filters of a DEG do not affect start and stop events, only the outcome events of the DEG. If the user wishes to filter a start or stop event, the start or stop event should be configured at the unit that generates that event.

8.3.2 Preciseness of synchronous direct events

Synchronous events are events that can be attributed to a specific instruction or VLES in the VLES stream. Ideally, a synchronous event, used as an input, causes a precise output event. Precise events can be either “break-before-make” or “break-after-make,” which means that the outcome occurred right before or right after the VLES that caused the input event. For example, a PC detector causes the core to enter debug mode on the exact PC that was detected, with the core state reflecting the state before the execution of the PC causing the event (break-before-make).

Some events are not precise, in which case one or more VLES will execute before the outcome takes effect. Table 8-11 lists the preciseness of all the synchronous event combinations supported by the event selector. This table relates only for events that change the state of the destination unit (start, stop, generate trace, and so on). It does not relate to events that are used as filters for other events, nor relates to events that operate via the IEU.

Table 8-11. Preciseness of synchronous direct event combinations

Causing event → Outcome ↓	PC detection	Instruction (DEBUGEV, and so on)	Address detection	Exception detection
Debug mode	P (B)	P (B)	P (B)	P (A)
start/stop PC detection	—	—	—	—
start/stop Data Address detection	—	P (A)	—	—
start/stop/reset counter	P (A)	P (A)	—	—
start/stop/message trace	P (A)	P (A)	—	—

The marking “P(B)” for outcomes of entry to debug mode stands for “Precise - Before”, meaning that the entry to debug mode or exception is at the architectural state as it was before executing the VLES causing the event. The marking “P(A)” stands for “Precise–After,” which means that the outcome occurs when the core is in the architectural state following VLES causing the event. For example, the relation P(A) between PC detection and enabling trace means that trace starts in the VLES following that pointed to by the PC detector. Therefore, if PC detectors mark the first and last VLES of a subroutine, then the first trace message will reflect the VLES following the start of the subroutine, while the last trace message will reflect the RTS and will not trace the destination VLES of the RTS.

8.4 Debug and trace register map

Nearly all Debug and Trace logic is controlled through memory mapped registers. The SoC has two physical memory spaces for accessing memory mapped registers:

- CCSR memory space: used for memory mapped registers of functional peripherals
- DCSR memory space: used for memory mapped registers of debug and test logic

Each memory spaced is accessed by way of a dedicated SkyBlue bus at the SoC level. Each SC3900 cluster is connected to these two buses. At the subsystem level, these two buses are united, and the registers see one physical bus (see Figure 8-1).

In addition, the local registers of the SC3900 subsystem (which include those of the DTU) are accessible by the core directly, outside the physical memory map using a local connection without going out to the system bus. This local access occurs in one of two cases:

- An MMU descriptor is allocated with a virtual address segment, which is marked by a “B0” (“Bank0”) attribute in the configuration. This attribute causes the access to be directed to the local

subsystem register space. The least significant 15 address bits determine which register is accessed in the 256KB local cluster register space. The higher order address bits are ignored. This configuration is typically how the monitor program should access the DTU registers

- In debug mode, core commands could be given a “Local” attribute that causes the access to be directed to the subsystem and cluster registers (the same 256KB). Only the 15 least significant address bits are used. The host debugger should use this method to access the DTU register by core commands, for example when transferring data from core registers to DTU registers.

Any access to the CCSR or DCSR memory space that does not use one of the two methods mentioned above are checked and translated by the MMU and will go out of the SC3900 subsystem to the SoC fabric, optionally returning through the SkyBlue SoC port. In this case, the physical address for each DTU is unique in the SoC memory map.

The DTU registers are mapped in both the DCSR and CCSR memory spaces. However, in some cases, registers may respond only to accesses from one of the buses. Registers allocated to the host partition should be accessed by the external host using the DCSR memory space, or accessed in debug mode by core commands using the Local attribute. A monitor program that functions as an external host of another core, should also access the DTU registers of that other core using the DCSR memory space. Registers that belong to the monitor partition could be accessed from the CCSR space, or accessed locally by the core if marked in the MMU as belonging to bank0.

This definition implies that registers that could be configured to belong to either partition should be accessed with different addresses, depending on their allocation. These two addresses have the same offset, based on the different physical base address of the CCSR and DCSR memory spaces.

The DCSR is an independent fabric that can support an access from the SoC also in the following conditions:

- When the core is in debug mode or a halted low power mode
- When the core is held due to a pending access on its main master bus
- When the SoC is held due to a pending access on CCSR to another core

Access permissions to DTU registers are controlled in two layers:

- At the memory master level: each core has an MMU, and each non-core bus master has a PAMU, where the memory access permissions for this master are configured. Each bus master that can function as a debug host must be allowed to access the DCSR memory space. An SC3900 MMU that supports a monitor should also allow access to the DTU registers in CCSR
- The DTU partition level: registers that are allocated to the monitor can be accessed only through CCSR or by a local access. Registers that are allocated to the external host can be accessed only through DCSR or, in debug mode, by a local access. Note that the external host can access the monitor registers in CCSR. The monitor can access the host registers by accessing DCSR, if the MMU allows it to access that space; It normally should not be allowed such access.

When the core attempts to write registers for which the MMU allows it to access, but the access is not allowed because the register is not allocated to the monitor partition, the register will not be affected, and an optional imprecise subsystem interrupt will be generated. Unless explicitly defined otherwise, the host and monitor can read registers of the other partition.

This table summarizes the general access permissions for DTU registers:

Table 8-12. General access permissions to DTU registers

Access generator	DTU register in the external host partition	DTU register in the monitor partition
Core command in debug mode marked as Local	Read and write	Read only
Monitor (core access marked in the MMU as “bank0”)	Read only	Read and write
SoC access (through DCSR)	Read and write	Read only
SoC access (through CCSR)	Read only	Read and Write

Note that specific DTU registers may have additional access restrictions, for example some registers are accessible only when the core is in debug mode. These specific conditions are in addition to what is specified in Table 8-12. Also note that access right violations for core accesses are detected and blocked at the MMU.

The DTU registers should be accessed only by 32-bit accesses. A few DTU registers support both 32-bit accesses (LD.L or ST.L) and 64-bit accesses (using LD.2L or ST.2L) when accessed directly from the core. Accesses from DCSR and CCSR are performed 32 bits at a time because this is the width of the buses from the SoC. The core may issue only 32-bit accesses that go through the SoC to DCSR or CCSR.

When a program running on the core writes to DTU registers, a memory barrier is required after the last write of the configuration sequence before the configuration may be considered as done by the code section following the barrier. DTU register configuration require the DBAR.SCG barrier followed by another SYNC.B instruction, for example:

```
st.l d0,(r0)      ; write DECSR to enable the DEBUGE.0 instruction
dbar.scfg sync.b
sync.b
debuge.0          ; this instruction is now enabled
```

The DTU registers in SC3900 subsystem are summarized in Table 8-13. Note that this list does not include registers used by the debugger that are in other blocks of the SC3900 subsystem, such as CME and cache control registers.

In the accessibility column, “host” or “monitor” means that these registers rigidly belong to the respective partition. “Config.” means that they could be configured, as a whole register, to belong to either partition. “Both—RO” means that this is a Read-Only register that by its content may be of interest to both partitions. “Both—field config.” means that the register is writable by both masters, but individual fields in the register are writable by only one of the masters, as configured for the respective resource. “Host—debug” means that this register belongs to the host partition, and in addition can be written to only when the core is in debug mode.

Table 8-13. Debug and trace register summary

Unit	Abbreviation	Function	Link to section/page	Accessibility	Address offset
Run Control	CCR3-0	Core Command Registers 3 through 0	8.5.6.1/8-25	Host—debug	0x10 - 0x1C
	CCCR	Core Command Control Register	8.5.6.2/8-26	Host—debug	0x20
	CCD0-3	Core Command Data registers	8.5.6.3/8-26	Host—debug	0x30 - 0x3C
	PC_NEXT	PC for next unexecuted VLES	8.5.6.4/8-27	Host—debug	0x50
	RCR	Run Control Register	8.5.6.5/8-28	Host—debug	0x54
	DMEER	Debug Mode Event Enabling Register	8.5.6.6/8-29	Host	0x70
	DMRSR	Debug Mode Reason Status Register	8.5.6.7/8-30	Host—debug	0x78
	DMCSR	Debug Mode Control and Status Register	8.5.6.8/8-32	Host	0x80
	DHRRR	Debug Host Resources Reservation Register	8.5.6.9/8-34	Host	0xA0
	DEEER	Debug Exception Event Enabling Register	8.5.6.10/8-36	Monitor	0xB0
	DERSR	Debug Exception Reason Status Register	8.5.6.11/8-37	Monitor	0xC0
	DECSR	Debug Exception Control and Status Register	8.5.6.12/8-38	Monitor	0xC8
	DESAR	Debug Exception Service Address Register	8.5.6.13/8-40	Monitor	0xD0
	DMRRR	Debug Monitor Resources Reservation Register	8.5.6.14/8-40	Monitor	0xF0
	DUICR	DTU Interface Control Register	8.5.6.15/8-42	Both—field config	0x100
	DRASR	Debug Resource Activity Status Register	8.5.6.16/8-44	Both—RO	0x108
	DESR	Debug Event Status Register	8.5.6.17/8-45	Both—field config	0x110
	SASR	Subsystem Activity Status Register	8.5.6.18/8-46	Both—RO	0x120
	DTUREV	Debug and Trace Unit Revision Register	8.5.6.19/8-48	Both—RO	0x140
Address and Data Detection Unit	ARDCR*n*	Address Range Detector Control Register *n*	8.6.1.1/8-50	Config per *n*=0..3	0x250 + *n* x 0x10
	DEPCR*n*	Dual Exact PC Detector Control Register	8.6.1.2/8-52	Config per *n*=0..3	0x254 + *n* x 0x10
	PADRRA*n*	PC & Address Detector Reference Register A	8.6.1.3/8-53	Config per n=0..3	0x258 + *n* x 0x10
	PADRRB*n*	PC & Address Detector Reference Register B	8.6.1.3/8-53	Config per n=0..3	0x25C + *n* x 0x10
	TIDCCR	Task ID Comparator Control Register	8.6.2.1/8-56	Both—field config	0x350
	TIDCRR0	Task ID Comparator Reference Register 0	8.6.2.2/8-57	Config	0x354
	EDCR	Exception Detector Control Register	8.6.3.1/8-57	Config	0x370
	EDRVR	Exception Detector Reference Value Register	8.6.3.2/8-58	Config	0x374
	EDMR	Exception Detector Mask Register	8.6.3.3/8-59	Config	0x378

Table 8-13. Debug and trace register summary (continued)

Unit	Abbreviation	Function	Link to section/page	Accessibility	Address offset
Indirect Event Unit	IECTLS	Indirect Control and Status Register	8.7.2.1/8-64	Both—field config	0x3A0
	IECTR*n*	Indirect Event Conditional Transition Configuration Register *n*	8.7.2.2/8-65	Config	0x3B0 - 0x3C8 (in jumps of 8)
	IEUTR	Indirect Event Unconditional Transition Configuration Register	8.7.2.3/8-67	Both—field config	0x3E0
Profiling Unit	PCCSR	Profiling Counters Control and Status Register	8.8.2.1/8-70	Both—field config	0x400
	PTCRA/B	Profiling Triad Control Register A/B	8.8.2.2/8-72	Config per Triad A/B	A: 0x410 B: 0x480
	PCVRA*n*/B*n*	Profiling Counter Value Register A*n*/B*n*	8.8.2.2/8-72	Config per Triad A/B	A: 0x420 + *n* x 4 B: 0x490 + *n* x 4
	PCSRA*n*/B*n*	Profiling Counter Snapshot Registers A*n*/B*n*	8.8.2.5/8-81	Config per Triad A/B	A: 0x440 + *n* x 4 B: 0x4B0 + *n* x 4
	RCCR0,1	Reloadable Counter Control Register 0	8.8.3.1/8-82	Config	0x500
	RCVR0,1	Reloadable Counter Value Register 0	8.8.3.2/8-85	Config	0x508
	RCRR0,1	Reloadable Counter Reload Register 0	8.8.3.3/8-86	Config	0x50C
	RCSR0,1	Reloadable Counter Snapshot Register 0	8.8.3.4/8-86	Config	0x510
Trace Unit	TC1	Trace Control Register 1	8.9.4.1/8-90	Config	0x600
	TRSR	Trace Status Register	8.9.4.2/8-91	Config	0x604
	TC3	Trace Control Register 3	8.9.4.3/8-92	Config	0x60C
	TC4	Trace Control Register 4	8.9.4.4/8-93	Config	0x610
	TPMCR	Trace Profiling Message Control Register	8.9.4.5/8-94	Config	0x614
	TWMSK	Trace Watchpoint Mask Register	8.9.4.6/8-95	Config	0x618
	TMDATI	Image of the TMDAT core register	8.9.4.8/8-96	Both—RO	0x630

8.5 Understanding run control

8.5.1 Debug instructions

The SC3900 core supports several instructions that are dedicated to debug support. These instructions allow the user to enter debug mode, call a debug exception, or generate a debug event who's exact function is configured in the DTU. The debug function that these instructions perform require that the DTU is enabled, and in addition the relevant partition is enabled. Table 8-14 below describes the dependency of these instructions on the configuration of the DTU. For more information on these instructions, see the debug chapter in the *SC3900 FVP Core Reference Manual*.

Table 8-14. SC3900 instructions for debug support and their affect by the DTU

Instruction	DTU partition it belongs to	Functionality when the respective partition is enabled	Functionality when the respective partition is disabled	DTU registers affecting the instruction
SWB	host	enter debug mode	Illegal exception	None
DEBUGM.*n* (*n*=0..3)	host	enter debug mode	NOP	DMCSR: a bit enabling each instruction variant (*n*), see Section 8.5.6.8, "Debug Mode Control and Status Register (DMCSR)"
DEBUGE.*n* (*n*=0..3)	monitor	enter debug exception	NOP	DECSR: a bit enabling each instruction variant (*n*), see Section 8.5.6.12, "Debug Exception Control and Status Register (DECSR)"
DEBUGEV.*n* (*n*=0..7)	per the allocation of the function it affects	—	NOP	In the control register of each DTU function it may affect, see Table 8-9

8.5.2 Debug mode

Debug mode is a special processing state of the core, where the pipeline is flushed, and dispatching of new instructions is suspended until the SoC informs the core to continue, via interface signals. Debug mode can be used only by the external host, so the ability to enter this mode requires the host partition to be enabled, see Section 8.2.2, "Enabling and disabling the DTU."

While the core is in debug mode, debug event generation, event counting and tracing are suspended. The subsystem timers, the core-associated watchdog timer and the trace timestamp are frozen while the core is in debug mode.

8.5.2.1 Entering and exiting debug mode

If the host partition is enabled, the core can enter debug mode because of one of the following reasons:

- Assertion of a debug request signal from the SoC run control block
- Execution of the SWB instruction
- Execution of one of the DEBUGM.*n* instructions which are individually enabled. See Section 8.5.6.8, "Debug Mode Control and Status Register (DMCSR)."
- Occurrence of a debug event, which is configured to cause entry into debug mode. See Section 8.5.6.6, "Debug Mode Event Enabling Register (DMEER)."
- Occurrence of an irrecoverable core exception, when already inside a routine servicing an irrecoverable exception (SR2.IRR set). For a description of the irrecoverable exception. See the *SC3900 FVP Core Reference Manual.*

Debug mode can be exited only when a debug resume signal is asserted from the SoC run control block, or when the core is reset.

When the external host asserts a debug request through the SoC control block when the core is in a low power mode, the core usually enters debug mode (with the core clocks activated), and when the SoC signals it to resume, the core returns to the low power mode it was in. This event sequence is either internal in the subsystem or managed by the external run control block so that the result is more or less transparent to the host debugger. However, the specific low power modes for which these transitions are supported are specific to the implementation. In the current implementation of the SC3900 subsystem, automatic entry to and exit from debug mode is supported only for the DOZE low power mode.

When exiting debug mode, the core performs a COF operation to the address programmed in the register PC_NEXT, see Section 8.5.6.4, "PC_NEXT."

8.5.2.2 Operations supported during debug mode

When the core is in debug mode, the host debugger can perform the following actions:

- Single step—execute one VLES as pointed by PC_NEXT. See Section 8.5.4, "Single stepping."
- Force-execute a VLES supplied by the host (called "core command" in this document). With core commands, the host debugger can upload or change the core state (registers), read or write memory using the virtual memory as seen by the application and more, or generate a direct memory access to the local memory mapped registers. See Section 8.5.5, "Core commands."
- Read or write the debug control registers of the DTU via the DCSR port. Configurations will take effect after the core exits from debug mode.
- Read or write from main memory or subsystem configuration registers directly as a master of the system bus. Through this path the core can also perform memory and cache management operations using physical addresses.

8.5.3 Debug exceptions

The SC3900 core has a single debug exception, with a user-configured address for the service routine. Debug exceptions can be used only by the monitor, so the ability to enter this exception requires the monitor partition to be enabled, see Section 8.2.2, "Enabling and disabling the DTU."

A debug exception can be generated by one of the following conditions:

- The DEBUGE.*n* instruction - compiled-in software debug exception. There are 4 instruction variants (*n*=0..3) which could each be individually enabled when configuring the DTU.
- Any debug event could be configured to generate a debug exception, however only a small subset of them can do this directly - the general way to generate a debug exception is through an IEU event, see Section 8.5.6.10, "Debug Exception Event Enabling Register (DEEER)."

Upon entering an exception, some information regarding the source of the exception is sampled into EIDR. For more information on the values sampled into EIDR see Section 5.4.4, "Exception Handling."

The identification of the events that caused the exception could be deduced by reading the Debug Exception Status Register, see Section 8.5.6.11, "Debug Exception Reason Status Register (DERSR)." In addition, DEBUGE.*n** has an immediate 10-bit operand that is also sampled upon servicing the exception into the EID field in EIDR. This enables the user to quickly communicate to the ISR the type of requested debug service. The service routine should check all the reasons that caused the exception, and service all

of them before returning. When servicing a debug exception cause, the user should clear the respective reason bit in DERSR. see Section 8.5.6.11, “Debug Exception Reason Status Register (DERSR).” Please refer to the *SC3900 FVP Core Reference Manual* for a description of the core-level actions that should be performed when servicing a debug exception.

Functions of the debug unit (trace, counting, event detection) are not automatically disabled during debug exceptions, unlike in debug mode.

8.5.4 Single stepping

Single stepping by the external host is performed, upon receiving the “exit from debug” strobe from the SoC, by performing a COF to the address programmed in PC_NEXT, and immediately entering debug mode again. PC_NEXT is updated automatically to point to the next un-executed VLES. The host debugger can change the value of PC_NEXT between steps (see Section 8.5.6.4, “PC_NEXT”).

Memory accesses that were generated by the single stepped VLES will have the attributes and privilege of the native code section (as configured in SR2). In particular, single stepped instructions would not have extra privilege that would enable them to execute privileged instructions or access otherwise protected memory.

It should be noted that the VLES that is executed upon receiving the exit from debug strobe from the SoC may not complete due to various micro-architectural situations. These include, for example, termination due to a pending interrupt, pipe re-wind and more. In case of an exception, PC_NEXT will point to the first VLES of the ISR (before execution). The original PC was killed. In other cases, PC_NEXT remains unchanged signifying that the VLES did not complete but no change of flow occurred. Upon failure, the PCK bit in DMCSR is set (see Section 8.5.6.8, “Debug Mode Control and Status Register (DMCSR)”). This bit is also communicated to the SoC run control block. The host debugger should check this bit after each single step to ensure it was executed. In some cases when the step did not complete, the host debugger should attempt another single step until successful completion. In other cases it should not. In any case the host debugger may choose not to re-step after such a failure. Table 8-15 describes the status bit settings that are related to single step operations, which can help the host debugger define the algorithm for managing it.

Table 8-15. Status values after single step operations[1]

DMCSR.PCK	DMCSR.PS	DMCSR.DINS	Case	Debugger action
0	x[2]	x	Last Single Step succeeded	Continue
1	0001 to 0011	x	Last Single Step killed due to an exception	Continue, Next Single Step will be from the ISR
	0100 to 0101	x	Last Single Step killed due to a rewind condition	Continue, next Single Step will re-execute this VLES and will probably succeed
	0000	not 0	Last Single Step killed due to a debug instruction (EDBUGM.*n*, SWB)	Per the status in DMCSR.DINS, if the instruction was • DEBUGM.n: skip this VLES by changing PC_NEXT • - SWB: per debugger flow for software breakpoints
		0	Last Single Step was killed due to another reason:	
			The core is in low-power mode.	Check SASR.SS (Table 8-33) to see what is the power mode, and optionally exit from this state by generating the appropriate condition in the SoC.
			There is a pending request to enter debug mode in DMRSR[3] (see Table 8.5.6.7).	Clear DMRSR and the external debug request source if relevant.

1 See Table 8-23 for a description of the fields of DMCSR.

2 "X"—value is not relevant (don't care).

3 A new event in DMRSR is only an external debug request. New internal events are blocked in debug mode and single stepping.

During single stepping the DTU ignores debug events that may be generated in parallel, such as trace generation, address detection, event counting etc. Note. however, that single stepping instructions that generate debug exceptions (such as DEBUGE.*n*) will perform a jump to the ISR.

8.5.5 Core commands

A core command is executed by presenting the core with an encoded VLES occupying up to 128-bits configured in DTU registers. This way flexible types of VLES can be executed - including multiple instructions, using any of the core execution units[1]. The encoded VLES is written in 4 memory mapped registers (CCR3-CCR0), which are accessible only in debug mode, see Section 8.5.6.1, "Core Command registers (CCR3-0)." Only one execution set can be executed from this encoded data, so in most cases less than 128-bits could be configured in these registers. The hardware identifies the "end of VLES" encoding that is embedded in the 128-bit data and knows to ignore the rest of the data in the Core Command Registers.

After configuring the CCRs, actual activation of the core command is done by writing to a bit in CCCR. For core commands that perform load or store instructions, an additional bit in CCCR enables to select if the access will be issued directly to the local cluster memory mapped registers (also known as "Bank0" registers), using only the 18 least significant address bits to specify the offset in the 256KB space of these

1. In previous StarCore architectures, only a single AGU instruction was supported in core commands.

registers. For more information on accessing bank0 registers, see Section 9.3.3, “Peripheral bus.” The MMU will not perform a protection check nor translate an access specified as local. For more information on the options to activate a core command, see Section 8.5.6.2, “Core Command Control Register (CCCR).”

The core command may be terminated due to architectural or micro-architectural reasons, for example an Illegal or MMU exception, and a re-wind condition. The execution status of the core command is reflected in the PCK bit in DMCSR is set, as well as the reason for failure (see Section 8.5.6.8, “Debug Mode Control and Status Register (DMCSR)”). The host debugger should check this bit after each execution attempt to ensure it was completed. In case the core command did not complete, in some cases, another execution strobe should be attempted until successful completion (it can be done without re-writing the CCRs). However, if the reason for failure was some kind of illegal condition (for example if the encoding of the core command was faulty), it should not be attempted again. The core ignores a VLES that caused an error. Exceptions of any kind are not serviced during core commands. Also, pending interrupts and exceptions do not abort the core command (except for errors that originate in the core command itself, such as MMU errors or illegal conditions). Note, however, that if the debugger issued a memory access that caused an non-precise interrupt (for example a Sky-Blue bus write error), this interrupt will remain pending and will be serviced by the core when it exits debug mode.

For more information on core commands as seen by the core see the *SC3900 FVP Core Reference Manual.*

8.5.6 Debug run control registers

8.5.6.1 Core Command registers (CCR3-0)

Four 32-bit registers (CCR3–CCR0), belonging to the host partition, accessible only in debug mode, in which the external host writes an encoded VLES to be executed as a core command. The VLES could be up to 128 bits long, but not all registers have to be configured for a shorter VLES. Assuming the VLES would have been encoded in program memory from address 0, CCR0 holds program bytes in addresses 0–3, CCR1 holds program bytes 4–7 and so on up to CCR3. Program words outside the scope of the VLES could be written with zeros or left unwritten.

Address 0x010 (CCR<i>)
offset:4
range: 3..0

Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CC<i>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CC<i>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-3. Core Command Registers (CCR3-0)

The fields of the CCR<i> registers are described in Table 8-16.

Table 8-16. Field description of CCR3–0

Field	Description
31-0 CC<i>	The respective encoding of the VLES of the core command. The lower program address portion should be programmed in CCR0, growing into CCR1 up to CCR3. A VLES shorter than 128 bits can be programmed in fewer registers, starting from CCR0.

8.5.6.2 Core Command Control Register (CCCR)

A 32-bit register, belonging to the host partition, can only be written to when the core is in debug mode.

Address 0x0020 (CCCR) Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R														LRMA		CCMD
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-4. Core Command Control Register (CCCR)

The fields of this register are described in Table 8-17.

Table 8-17. Fields of the Core Command Control Register (CCCR)

Field	Description
0 CCMD	Core Command: execute the core command as configured in CCR0-3 0: Don't execute 1: Execute (write only; always read as 0)
2 LRMA	Local Register Memory Access: marks any load and store instructions in the core command to be directed to the local register memory space (which includes the DTU, MMU and other memory mapped registers of the SC3900 subsystem). 0: Memory accesses are issued as virtual, going through the MMU as regular accesses 1: Memory accesses are issued to the local SC3900 subsystem registers. The least significant 18 bits of the address determine the register address offset in the cluster 256KB register space (bank0). The 14 MS address bits are ignored. The bit affects both accesses in the VLES. It should not be set if the core commands includes instructions accessing the memory other than LD* or ST*.

8.5.6.3 Core Command Data (CCD0–3)

Four 32-bit registers (CCD0–CCD3), belonging to the host partition, that could be read or written to only when the core is in debug mode. They are intended as intermediate data registers, when transferring data to or from the core using core commands.

When the host debugger wishes to upload the core state, it can initialize an R register to point to the CCD0, and perform store operations (this R register should be saved separately, see Section 8.9.4.8, "TMDAT

Image Register (TMDATI)”). The host debugger can then pick up these values from this location to the external host. Similarly when uploading the state of the core registers from the external host, the data values can be written to the CCD registers, and then core commands can load them up to the destination core registers.

Address 0x030 (CCD<i>)
offset: 4
range: 0..3

Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CCD<i>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CCD<i>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-5. Core Command Data (CCD0–3)

The fields of the CCD<i> registers are described in Table 8-18.

Table 8-18. Field description of CCD3–0

Field	Description
31-0 CCD<i>	Holds the data written by the debugger. Typically the debugger can use these registers as private temporary memory for store and load instructions issued by core commands

8.5.6.4 PC_NEXT

A 32 bit read-write register, belonging to the host partition, writable only in debug mode. Upon entry to debug mode, the register holds the PC of the next un-executed VLES in the program flow. In case of break before make events, it holds the PC of the event causing the event. Single stepping and exit from debug mode require PC_NEXT to hold a valid value. Upon an “exit from debug” signal from the SoC, the core performs a COF to the address stored in PC_NEXT. In case of a single step, it immediately enters debug mode again. Serial single stepping could be done without explicitly updating PC_NEXT. The external host can write a different address to this register and the core will jump to it upon the next assertion of the “exit from debug” signal. Note that the PC core register in debug mode may not always hold the address of the next VLES to execute, so the debugger must only use PC_NEXT to get this information.

The register is readable, but not writable, when the core is running, in which case it will return the PC of the next uncompleted VLES when it was read. When the core is not held, it could be seen as an imprecise sampling of the PC value because the read access of the external host is not synchronized with the activity of the core. However when the core is stuck on a memory access generated by a non-speculative VLES, PC_NEXT will hold the PC of the VLES causing the faulty access, which could be useful when analyzing this failure. If the VLES was speculated, PC_NEXT will show a PC value that may belong to an earlier VLES, right before starting the speculated flow. If the core was stuck due to a speculative data access, PC_NEXT may point to a few VLES before the VELS with the faulty access. If the core was stuck on a

speculative program fetch, then PC_NEXT may point to a VLES much earlier than the VLES speculated with the faulty program address.

Address 0x050 (PC_NEXT) Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PC_N															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PC_N															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-6. NEXT PC Register (PC_NEXT)

The fields of the PC_NEXT register are described in Table 8-19.

Table 8-19. Field description of PC_NEXT

Field	Description
31-0 PC_N	When in debug mode, holds the PC of the next unexecuted VLES. Can be written in debug mode with the PC for the next single step or PC to resume execution. During execution, continuously reflects the committed PC, allowing sampling by the external host

8.5.6.5 Run Control Register (RCR)

A 32-bit register, belonging to the host partition. Can only be accessed in debug mode. This register selects the action that the core performs when the “exit from debug” strobe is sent from the SoC. SoC triggering is required in order to allow synchronized single stepping or exit from debug mode in several cores. In both “Exit debug” and “Single step” selection, the action performs a change of flow (COF) to the PC that is stored in PC_NEXT, see Section 8.5.6.4, “PC_NEXT.”

Address 0x054 (RCR) Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R														ES		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-7. Run Control Register (RCR)

The fields of this register are described in Table 8-20.

Table 8-20. Field description of RCR

Field	Description
2-0 ES	Execution Select: The action to be performed when the “exit debug” strobe arrives from the SoC. 000: NOP 001: Exit debug 010: Single Step 011: Reserved 1xx: Reserved

8.5.6.6 Debug Mode Event Enabling Register (DMEER)

A 32-bit read-write register, belonging to the host partition, that controls what input events are enabled to cause the core to enter into debug mode. Each input event is controlled by a bit. A bit which is set enables entry into debug mode as a result of the specified event. If more than one input event is enabled, the events are OR-ed. Some of the events are precise, and some are imprecise, as described per event. Note that there are other unmaskable events that may cause entry into debug mode, which are not controlled by this register.

Address 0x070 (DMEER) Ownership: Host:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R					IE3	IE2	IE1	IE0		RCE0		CTOA		EXD		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PCDA7	PCDA6	PCDA5	PCDA4	PCDA3	PCDA2	PCDA1	PCDA0								
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-8. Debug Mode Event Enabling Register (DMEER)

The fields of this register are described in Table 8-21.

Table 8-21. Field description of DMEER

Field	Description
8 PCDA0	Enable entry into debug mode as a result of an event from PC or Data Address detector 0. For Data address detection, the PCDA0 and PCDA1 bits must both be set
9 PCDA1	Enable entry into debug mode as a result of an event from PC or Data Address detector 1. For Data address detection, the PCDA0 and PCDA1 bits must both be set
10 PCDA2	Enable entry into debug mode as a result of an event from PC or Data Address detector 2. For Data address detection, the PCDA2 and PCDA3 bits must both be set
11 PCDA3	Enable entry into debug mode as a result of an event from PC or Data Address detector 3. For Data address detection, the PCDA2 and PCDA3 bits must both be set

Table 8-21. Field description of DMEER (continued)

Field	Description
12 PCDA4	Enable entry into debug mode as a result of an event from PC or Data Address detector 4. For Data address detection, the PCDA4 and PCDA5 bits must both be set
13 PCDA5	Enable entry into debug mode as a result of an event from PC or Data Address detector 5. For Data address detection, the PCDA4 and PCDA5 bits must both be set
14 PCDA6	Enable entry into debug mode as a result of an event from PC or Data Address detector 6. For Data address detection, the PCDA6 and PCDA6 bits must both be set
15 PCDA7	Enable entry into debug mode as a result of an event from PC or Data Address detector 7. For Data address detection, the PCDA6 and PCDA7 bits must both be set
18 EXD	Enable entry into debug mode as a result of an event from the Exception Detector
20 CTOA	Enable entry into debug mode as a result of a Counter Triad A Overflow event
22 RCE0	Enable entry into debug mode as a result of a Reloadable Counter 0 event
24 IE0	Enable entry into debug mode as a result of Indirect event 0
25 IE1	Enable entry into debug mode as a result of Indirect event 1
26 IE2	Enable entry into debug mode as a result of Indirect event 2
27 IE3	Enable entry into debug mode as a result of Indirect event 3

8.5.6.7 Debug Mode Reason Status Register (DMRSR)

A 32-bit read-write register, belonging to the host partition, which could be accessed only when the core is in debug mode. The register reflects which debug events caused entry into debug mode. Reasons reflected in this register are mostly enabled in DMEER, but others are unconditional (such as some instructions and the external debug request signal from teh SoC). Per bit, the position in this register corresponds to the position of the respective bit in DMEER if relevant. When entering debug mode, several reason bits could be set, and the host debugger should examine all reasons before returning to execution. If an event bit of a precise reason is set, the entry into debug mode is precise. Imprecise reasons may relate to earlier VLES, due to the delay in asserting them. The read-only bits in this register are cleared automatically when the core exits from debug mode. Bits for imprecise reasons, marked by “W1C” in Figure 8-9 should be written by 1 by the host debugger in order to clear them. If their value is not 0 when exiting from debug mode, the core will enter debug mode again.

Address 0x078 (DMRSR)

Ownership: Host:DM
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INS	SFDM			IE3	IE2	IE1	IE0		RCE0		CTOA		EXD		
W	R	w1c			w1c	w1c	w1c	w1c		w1c		w1c		R		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PCDA7	PCDA6	PCDA5	PCDA4	PCDA3	PCDA2	PCDA1	PCDA0								
W	R	R	R	R	R	R	R	R								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-9. Debug Mode Reason Status Register (DMRSR)

The fields of this register are described in Table 8-22.

Table 8-22. Field description of DMRSR

Field	Description
8 PCDA0	The core entered debug mode as a result of an event from PC or Data Address detector 0
9 PCDA1	The core entered debug mode as a result of PC detection from PCDA1
10 PCDA2	The core entered debug mode as a result of an event from PC or Data Address detector 2
11 PCDA3	The core entered debug mode as a result of PC detection from PCDA3
12 PCDA4	The core entered debug mode as a result of an event from PC or Data Address detector 4
13 PCDA5	The core entered debug mode as a result of PC detection from PCDA5
14 PCDA6	The core entered debug mode as a result of an event from PC or Data Address detector 6
15 PCDA7	The core entered debug mode as a result of PC detection from PCDA7
18 EXD	The core entered debug mode as a result of an event from the Exception Detector
20 CTOA	The core entered debug mode as a result of a Counter Triad A Overflow event
22 RCE0	The core entered debug mode as a result of a Reloadable Counter 0 event
24 IE0	The core entered debug mode as a result of Indirect event 0
25 IE1	The core entered debug mode as a result of Indirect event 1

Table 8-22. Field description of DMRSR (continued)

Field	Description
26 IE2	The core entered debug mode as a result of Indirect event 2
27 IE3	The core entered debug mode as a result of Indirect event 3
30 SFDM	The core entered debug mode as a result of an assertion of the SoC Force Debug Mode signal
31 INS	The core entered debug mode as a result of executing a debug core instruction, see the DMCSR. DINS field for the exact instruction

8.5.6.8 Debug Mode Control and Status Register (DMCSR)

A 32-bit read-write register, belonging to the host partition. The register holds a combination of control bits that affect various aspects of debug mode functionality, as well as status bits that reflect the state of the core when debug mode was entered. The status bits are updated upon entry to debug mode, or execution of a core command or single step. Entry to debug mode involves killing a VLES and flushing the pipe; The status reflects if this process occurred while the killed VLES was being terminated by another reason as well. If the killed VLES was also killed by a pending exception, then the exception processing process has already started (sampling into the link registers, EIDR update, and PC_NEXT pointing to the first VLES in the ISR). If the killed VLES suffered a rewind condition, PC_NEXT points to the VLES that suffered the rewind. After performing a core command or a single step, the killing status reflects that of the core command or the single-stepped VLES.

Address 0x080 (DMCSR)

Ownership: Host
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	DBGM3	DBGM2	DBGM1	DBGM0	DCDM	ICDM	FCME	DIDM								
W	RW	RW	RW	RW	RW	RW	RW	RW								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	DMMUC	IMMUC						CS	PS				PCK	DINS		
W	w1c	w1c						R	R				R	R		
Reset	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-10. Debug Mode Control and Status Register (DMCSR)

The fields in this register are described in Table 8-23.

Table 8-23. Field description of DMCSR

Field	Description
2-0 DINS	The instruction which was aborted when entering debug mode. Non-zero values possible only if DMRSR.INS = 1. 000 - Other instructions - not part of the following options 001 - SWB 010 - Nested irrecoverable exception condition 011 - Reserved 100 - DEBUGM.0 101 - DEBUGM.1 110 - DEBUGM.2 111 - DEBUGM.3
3 PCK	The previous core command or single step VLES did not execute (killed). In case of a core command, it is set in case of an illegal encoding, MMU error or rewind. The PS field gives more information on the reason the VLES did not complete
7-4 PS	The Pipe Status after single stepping and core command 0000 - The debug point is not before the service of the first VLES of an exception and not before re-executing of a VLES that was rewound. 0001 - Jumped to an exception using LR0, halted before executing the 1st VLES 0010 - Jumped to an exception using LR1 (MMU), halted before executing the 1st VLES 0011 - Jumped to an exception using LR2 (Debug), halted before executing the 1st VLES 0100 - Rewind: program cache situation. Halted on the VLES before re-execution 0101 - Rewind: data cache situation. Halted on the VLES before re-execution All other combinations - reserved Note: values that are possible after a core command are only 0000 (no special state), 0101 (data rewind)
8 CS	Core Status when in debug mode 0 - Core is idle, ready to accept another core command or single step 1 - Core is active processing a core command or single step Same bit as in SASR.CS (see Section 8.5.6.18, “Subsystem Activity Status Register (SASR)”) used to determine if a new core command or single step can be issued
14 IMMUC	Set if a descriptor was changed in the Instruction MMU since the last time this bit was cleared
15 DMMUC	Set if a descriptor was changed in the Data MMU since the last time this bit was cleared
24 DIDM	Disable Interrupts in Debug mode: If set, will disable maskable interrupts while the core is in debug mode, and during single stepping. It could be seen as having the same function like the SR2.DI bit in disabling maskable interrupts.
25 FCME	Freeze CME while in debug mode. Outside of debug mode this bit is ignored by the CME. If the debugger wishes to freeze the CME, it should set this bit before debug mode is entered or when the CME is not active. This bit should not be set by the debugger when in debug mode and the CME is active. The debugger may clear this bit while in debug mode, in which case the CME would resume its operation. 0 - When entering debug mode, the CME continues to operate as configured 1 - When entering debug mode, the CME is frozen
26 ICDM	When the core is in debug mode, place the L1 instruction cache in cache debug mode. In this mode, the debugger is not allowed to access the cache through the core or by the CME.

Table 8-23. Field description of DMCSR (continued)

Field	Description
27 DCDM	When the core is in debug mode, place the L1 data cache in cache debug mode. In this mode, the debugger is not allowed to access the cache through the core or by the CME.
28 DBGM0	Enable the DEBUGM.0 instruction 0 - DEBUGM.0 is disabled, executes as a NOP 1 - DEBUGM.0 is enabled, when executes causes the core to enter debug mode (break before make)
29 DBGM1	Enable the DEBUGM.1 instruction 0 - DEBUGM.1 is disabled, executes as a NOP 1 - DEBUGM.1 is enabled, when executes causes the core to enter debug mode (break before make)
30 DBGM2	Enable the DEBUGM.2 instruction 0 - DEBUGM.2 is disabled, executes as a NOP 1 - DEBUGM.2 is enabled, when executes causes the core to enter debug mode (break before make)
31 DBGM3	Enable the DEBUGM.3 instruction 0 - DEBUGM.3 is disabled, executes as a NOP 1 - DEBUGM.3 is enabled, when executes causes the core to enter debug mode (break before make)

8.5.6.9 Debug Host Resource Reservation Register (DHRRR)

A 32-bit read-write register, allocated to the host partition. It holds a bit per debug resource that is allocatable to either the host or monitor partitions. A value of 0 in a bit signifies that the resource is not allocated to the host. When the host wishes to allocate the resource, it writes a “1” to the bit, and then reads it. If the bit was read as 1, the resource is now allocated to the host. If the bit was read as 0 then the allocation request failed (used by the monitor) and the host cannot use this resource.

Address 0x0A0 (DHRRR)

Ownership: Host:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OES3	OES2	OES1	OES0	IEQ3	IEQ2	IEQ1	IEQ0		RC0	CTB	CTA	TU	EXD		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		TIDC0			PADG3	PADG2	PADG1	PADG0								
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-11. Debug Host Resource Reservation Register (DHRRR)

Table 8-24 below lists the resources and the respective registers who’s write privileges are affected. Note that in some cases the granularity of reservation is to a group of resources. For more information on the resource allocation protocol see Section 8.2.1, “DTU resource partitioning.”

Table 8-24. Field description of DHRRR

Field	Description
8 PADG0	PC and address detector group 0: affecting registers ARDCR0, DEPCR0, PADRRA0, PADDRB0 0 - not allocated to the external host 1 - allocated to the external host
9 PADG1	PC and address detector group 1: affecting registers ARDCR1, DEPCR1, PADRRA1, PADDRB1 0 - not allocated to the external host 1 - allocated to the external host
10 PADG2	PC and address detector group 2: affecting registers ARDCR2, DEPCR2, PADRRA2, PADDRB2 0 - not allocated to the external host 1 - allocated to the external host
11 PADG3	PC and address detector group 3: affecting registers ARDCR3, DEPCR3, PADRRA3, PADDRB3 0 - not allocated to the external host 1 - allocated to the external host
14 TIDC0	Task ID Comparator 0: affecting registers TIDCCR (partial), TIDCRR0 0 - not allocated to the external host 1 - allocated to the external host
18 EXD	The Exception Detector: affecting registers EDCR, EDRVR, EDMR 0 - not allocated to the external host 1 - allocated to the external host
19 TU	The Trace Unit: affecting registers TC1, TRSR, TC3, TC4, TPMCR, TWMSK, TROCR 0 - not allocated to the external host 1 - allocated to the external host
20 CTOA	Counter Triad A: affecting registers PCCSR (partial), TCRA, PCVRA0-3, PCSRA0-3 0 - not allocated to the external host 1 - allocated to the external host
21 CTOB	Counter Triad B: affecting registers PCCSR (partial), TCRB, PCVRB0-3, PCSRB0-3 0 - not allocated to the external host 1 - allocated to the external host Must be selected with the same state as CTOA
22 RC0	Reloadable Counter 0: affects registers RCCR0, RCVR0, RCRR0, RCSR0 0 - not allocated to the external host 1 - allocated to the external host
24 IEQ0	Indirect Event Unit bit Q0: affects registers IECTLS (partial), IECTR0, IEUTR (partial) 0 - not allocated to the external host 1 - allocated to the external host

Table 8-24. Field description of DHRRR (continued)

Field	Description
25 IEQ1	Indirect Event Unit bit Q1: affects registers IECTLS (partial), IECTR1, IEUTR (partial) 0 - not allocated to the external host 1 - allocated to the external host
26 IEQ2	Indirect Event Unit bit Q2: affects registers IECTLS (partial), IECTR2, IEUTR (partial) 0 - not allocated to the external host 1 - allocated to the external host
27 IEQ3	Indirect Event Unit bit Q3: affects registers IECTLS (partial), IECTR3, IEUTR (partial) 0 - not allocated to the external host 1 - allocated to the external host
28 OES0	Output Event Signal 0: affects register DUICR (partial) 0 - not allocated to the external host 1 - allocated to the external host
29 OES1	Output Event Signal 1: affects register DUICR (partial) 0 - not allocated to the external host 1 - allocated to the external host
30 OES2	Output Event Signal 2: affects register DUICR (partial) 0 - not allocated to the external host 1 - allocated to the external host
31 OES3	Output Event Signal 3: affects register DUICR (partial) 0 - not allocated to the external host 1 - allocated to the external host

8.5.6.10 Debug Exception Event Enabling Register (DEEER)

A 32-bit read-write register, belonging to the monitor partition, that controls what input events are enabled to cause the core to enter into debug exception. Each input event is controlled by a bit. A bit which is set enables entry into debug exception as a result of the specified event. If more than one input event is enabled, the events are OR-ed.

Address 0x0B0 (DEEER)

Ownership: Monitor:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R					IE3	IE2	IE1	IE0								
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-12. Debug Exception Event Enabling Register (DEEER)

The fields in this register are described in Table 8-25.

Table 8-25. Field description of DEEER

Field	Description
24 IE0	Enable entry into debug exception as a result of Indirect event 0
25 IE1	Enable entry into debug exception as a result of Indirect event 1
26 IE2	Enable entry into debug exception as a result of Indirect event 2
27 IE3	Enable entry into debug exception as a result of Indirect event 3

8.5.6.11 Debug Exception Reason Status Register (DERSR)

A 32-bit read-write register, belonging to the monitor, that reflects which input events caused the recent entry of the core to into debug exception. Upon entry into debug exception, the state of the register reflects what input events caused it. The bits in this register correspond to the reason bits specified for DEEER, with the addition of the some other reasons, such as unmaskable exceptions (such as DEBUGE.*n*). Instructions causing debug exceptions are reflected by one bit, the exact type of instruction is reflected in DECSR. If an event bit of a precise reason is set, the entry into debug mode is precise. Imprecise reasons may be set as well. In the ISR servicing the debug exception, the software should record all reasons causing the entry into debug exception and clear the independent (writable) status bits before returning from the exception or enabling nested debug exceptions, so that the same event would not cause entry into debug exception again. Clearing a bit that was set is done by writing “1” back to it. Writing “0”, or writing “1” to a bit that was clear has no effect. This behavior is required to ensure that a new imprecise exception that was set after the ISR has read the DERSR and before clearing it will not be accidentally cleared thus loosing the indication that the new exception condition had occurred.

Address 0x0C0 (DERSR) Ownership: Monitor: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	INS				IE3	IE2	IE1	IE0								
W	R				w1c	w1c	w1c	w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-13. Debug Exception Reason Status Register (DERSR)

Table 8-26 describes the bits in this register.

Table 8-26. Field description of DERSR

Field	Description
24 IE0	The core entered debug exception as a result of Indirect event 0
25 IE1	The core entered debug exception as a result of Indirect event 1
26 IE2	The core entered debug exception as a result of Indirect event 2
27 IE3	The core entered debug exception as a result of Indirect event 3
31 INS	The core entered debug exception as a result of executing a debug exception core instruction (not including RTE.STP), see the DECSR.DINS field for the exact instruction

8.5.6.12 Debug Exception Control and Status Register (DECSR)

A 32-bit read-write register, belonging to the monitor partition, that allows configuring some functions related to debug exceptions, and also reflects the core status at the entry point of the debug exception. Entry to every debug exception involves killing a VLES and jumping to the ISR instead. The status reflects what was the killed instruction (if it was debug-related) and also if this process occurred while the killed VLES was being terminated by another reason as well. If the killed VLES was also killed by a pending exception, then exception processing has already started (sampling into the link registers, EIDR update, and the return PC of the debug exception is that of the first VLES in the ISR). If the killed VLES was due to a rewind condition, the return PC points to the VLES that suffered the rewind.

Address 0x0C8 (DECSR) Ownership: Monitor: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	DBGE3	DBGE2	DBGE1	DBGE0				TFDE								
W	RW	RW	RW	RW				RW								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	DMMUC	IMMUC							PS					DINS		
W	w1c	w1c							R					R		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-14. Debug Exception Control and Status Register (DECSR)

The fields in this register are described in Table 8-27.

Table 8-27. Field description of DECSR

Field	Description
2-0 DINS	The instruction which caused the entry into debug exception. Non-zero values possible only if DERSR.INS = 1. 000 - Other instructions - not part of the following options 001 - Reserved 010 - Reserved 011 - Reserved 100 - DEBUGE.0 101 - DEBUGE.1 110 - DEBUGE.2 111 - DEBUGE.3
7-4 PS	The Pipe Status during entry into debug exception 0000 - The debug exception point is not before the service of the first VLES of an exception and not before re-executing of a VLES that was rewound. 0001 - Jumped to an exception using LR0, serviced the debug exception before executing the 1st VLES 0010 - Jumped to an exception using LR1 (MMU), serviced the debug exception before executing the 1st VLES 0100 - Rewind: program cache situation, serviced the debug exception before re-executing the VLES 0101 - Rewind: data cache situation. serviced the debug exception before re-executing the VLES All other combinations - reserved
14 IMMUC	Set if a descriptor was changed in the Instruction MMU since the last time this bit was cleared
15 DMMUC	Set if a descriptor was changed in the Data MMU since the last time this bit was cleared
24 TFDE	Freeze subsystem timer - Setting this bit disables the subsystem timer during debug exceptions (SR2.IDE set). This freeze condition is added on the general DTU output that freezes the subsystem timer in debug mode. The watchdog timer is not affected by this bit.
28 DBGE0	Enable the DEBUGE.0 instruction 0 - DEBUGE.0 is disabled, execute as a NOP 1 - DEBUGE.0 is enabled, when execute cause the core to enter debug exception (break before make)
29 DBGE1	Enable the DEBUGE.1 instruction 0 - DEBUGE.1 is disabled, execute as a NOP 1 - DEBUGE.1 is enabled, when execute cause the core to enter debug exception (break before make)
30 DBGE2	Enable the DEBUGE.2 instruction 0 - DEBUGE.2 is disabled, execute as a NOP 1 - DEBUGE.2 is enabled, when execute cause the core to enter debug exception (break before make)
31 DBGE3	Enable the DEBUGE.3 instruction 0 - DEBUGE.3 is disabled, execute as a NOP 1 - DEBUGE.3 is enabled, when execute cause the core to enter debug exception (break before make)

8.5.6.13 Debug Exception Service Address Register (DESAR)

A 32-bit read-write registers, belonging to the monitor partition, determining the PC to which the core will jump to a debug exception. The 6 LS bits [5:0] are tied to zero in the register. The bits written to the reserved portion of the register must be written as zero for future compatibility.

Address 0x0D0 (DESAR) Ownership: Monitor: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	DESA															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	DESA										0					
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-15. Debug Exception Service Address Register (DESAR)

The fields in this register are described in Table 8-28.

Table 8-28. Field description of DESAR

Field	Description
31-6 DESA	Debug Exception Service Address bits 31-6.

8.5.6.14 Debug Monitor Resource Reservation Register (DMRRR)

A 32-bit read-write register, allocated to the monitor partition. It holds a bit per debug resource that is allocatable to either the host or monitor partitions. A value of 0 in a bit signifies that the resource is not allocated to the monitor. When the monitor wishes to allocate the resource, it writes a “1” to the bit, and then reads it. If the bit was read as 1, the resource is now allocated to the monitor. If the bit was read as 0 then the allocation request failed (used by the host) and the monitor cannot use this resource.

Address 0x0F0 (DMRRR) Ownership: Monitor:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OES3	OES2	OES1	OES0	IEQ3	IEQ2	IEQ1	IEQ0		RC0	CTB	CTA	TU	EXD		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		TIDC0			PADG3	PADG2	PADG1	PADG0								
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-16. Debug Monitor Resource Reservation Register (DMRRR)

Table 8-29 lists the resources and the respective registers who's write privileges are affected. Note that in some cases the granularity of reservation is to a group of resources. For more information on the resource allocation protocol see Section 8.2.1, "DTU resource partitioning."

Table 8-29. Field description of DMRRR

Field	Description
8 PADG0	PC and address detector group 0: affecting registers ARDCR0, DEPCR0, PADRRA0, PADDRB0 0 - not allocated to the monitor 1 - allocated to the monitor
9 PADG1	PC and address detector group 1: affecting registers ARDCR1, DEPCR1, PADRRA1, PADDRB1 0 - not allocated to the monitor 1 - allocated to the monitor
10 PADG2	PC and address detector group 2: affecting registers ARDCR2, DEPCR2, PADRRA2, PADDRB2 0 - not allocated to the monitor 1 - allocated to the monitor
11 PADG3	PC and address detector group 3: affecting registers ARDCR3, DEPCR3, PADRRA3, PADDRB3 0 - not allocated to the monitor 1 - allocated to the monitor
14 TIDC0	Task ID Comparator 0: affecting registers TIDCCR (partial), TIDCRR0 0 - not allocated to the monitor 1 - allocated to the monitor
18 EXD	The Exception Detector: affecting registers EDCR, EDRVR, EDMR 0 - not allocated to the monitor 1 - allocated to the monitor
19 TU	The Trace Unit: affecting registers TC1, TRSR, TC3, TC4, TPMCR, TWMSK, TROCR 0 - not allocated to the monitor 1 - allocated to the monitor
20 CTOA	Counter Triad A: affecting registers PCCSR (partial), TCRA, PCVRA0-3, PCSRA0-3 0 - not allocated to the monitor 1 - allocated to the monitor
21 CTOB	Counter Triad B: affecting registers PCCSR (partial), TCRB, PCVRB0-3, PCSRB0-3 0 - not allocated to the monitor 1 - allocated to the monitor Must be selected with the same state as CTOA

Table 8-29. Field description of DMRRR (continued)

Field	Description
22 RC0	Reloadable Counter 0: affects registers RCCR0, RCVR0, RCRR0, RCSR0 0 - not allocated to the monitor 1 - allocated to the monitor
24 IEQ0	Indirect Event Unit bit Q0: affects registers IECTLS (partial), IECTR0, IEUTR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
25 IEQ1	Indirect Event Unit bit Q1: affects registers IECTLS (partial), IECTR1, IEUTR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
26 IEQ2	Indirect Event Unit bit Q2: affects registers IECTLS (partial), IECTR2, IEUTR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
27 IEQ3	Indirect Event Unit bit Q3: affects registers IECTLS (partial), IECTR3, IEUTR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
28 OES0	Output Event Signal 0: affects register DUICR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
29 OES1	Output Event Signal 1: affects register DUICR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
30 OES2	Output Event Signal 2: affects register DUICR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor
31 OES3	Output Event Signal 3: affects register DUICR (partial) 0 - not allocated to the monitor 1 - allocated to the monitor

8.5.6.15 DTU Interface Control Register (DUICR)

A 32-bit read-write register, belonging both to the host and monitor partitions, which controls the four output signals from the DTU to the SoC, typically the SoC debug event profiling and cross triggering block. Each signal can be optionally triggered by up to 4 triggers (DEBUGEV.*n* instructions or IEU events). The four triggers are OR-ed before generating an output pulse. The output signal remains asserted until acknowledged by and acknowledge signal from the receiving unit, hence two close events may be seen as one by the recipient if it did not manage to acknowledge the first one in time.

Each signal can be independently allocated to the host or the monitor partition, so both the host and the monitor can write to this register. However each debug master can write only the field controlling the output signals that are allocated to it.

Address 0x100 (DUICR) Ownership: Both:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SPDEO3				SPDEO2				SPDEO1				SPDEO0			
W	RW				RW				RW				RW			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-17. DTU Interface Control Register (DUICR)

Table 8-30 describes the fields of this register.

Table 8-30. Field description of DUICR

Field	Description
3-0 SPDEO0	Send Pulse On Debug_Event_Out_0. This field is independently allocated to either the host or the monitor partition. 0000 - debug_event_out_0 not enabled xxx1 - pulse sent when DEBUGEV.0 is executed xx1x - pulse sent when DEBUGEV.4 is executed x1xx - pulse sent when IEU Q0 event occurred 1xxx - pulse sent when IEU Q2 event occurred
7-4 SPDEO1	Send Pulse On Debug_Event_Out_1. This field is independently allocated to either the host or the monitor partition. 0000 - debug_event_out_0 not enabled xxx1 - pulse sent when DEBUGEV.1 is executed xx1x - pulse sent when DEBUGEV.5 is executed x1xx - pulse sent when IEU Q1 event occurred 1xxx - pulse sent when IEU Q3 event occurred
11-9 SPDEO0	Send Pulse On Debug_Event_Out_2. This field is independently allocated to either the host or the monitor partition. 0000 - debug_event_out_0 not enabled xxx1 - pulse sent when DEBUGEV.2 is executed xx1x - pulse sent when DEBUGEV.6 is executed x1xx - pulse sent when IEU Q0 event occurred 1xxx - pulse sent when IEU Q3 event occurred
15-12 SPDEO0	Send Pulse On Debug_Event_Out_3. This field is independently allocated to either the host or the monitor partition. 0000 - debug_event_out_0 not enabled xxx1 - pulse sent when DEBUGEV.3 is executed xx1x - pulse sent when DEBUGEV.7 is executed x1xx - pulse sent when IEU Q1 event occurred 1xxx - pulse sent when IEU Q2 event occurred

8.5.6.16 Debug Resource Activity Status Register (DRASR)

A 32-bit read-only register, accessible from both host and monitor partitions, captures the global activity status of all debug resources in the DTU. Each bit is set if the respective DEG is in the "active" state, see Section 8.3.1, "Start and stop events."

Address 0x108 (DRASR) Ownership: Both: Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R			EHEN	MONEN		TU				RC0	CTB	CTA		EXD		
W			R	R		R				R	R	R		R		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PCDA7	PCDA6	PCDA5	PCDA4	PCDA3	PCDA2	PCDA1	PCDA0								
W	R	R	R	R	R	R	R	R								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-18. Debug Resource Activity Status Register (DRASR)

Table 8-31 describes the bits in this register. A "1" in a bit value means that the respective detector or unit is active.

Table 8-31. Field description of DRASR

Field	Description
8 PCDA0	PC or Data Address detector 0 is active. If configured as a dual exact PC detector, same bit as DEPCR0.ACTA. If configured as a range detector, same bit as ARDCR0.ACT.
9 PCDA1	PC or Data Address detector 1 is active. If configured as a dual exact PC detector, same bit as DEPCR0.ACTB. If configured as a range detector, same bit as ARDCR0.ACT.
10 PCDA2	PC or Data Address detector 2 is active. If configured as a dual exact PC detector, same bit as DEPCR1.ACTA. If configured as a range detector, same bit as ARDCR1.ACT.
11 PCDA3	PC or Data Address detector 3 is active. If configured as a dual exact PC detector, same bit as DEPCR1.ACTB. If configured as a range detector, same bit as ARDCR1.ACT.
12 PCDA4	PC or Data Address detector 4 is active. If configured as a dual exact PC detector, same bit as DEPCR2.ACTA. If configured as a range detector, same bit as ARDCR2.ACT.
13 PCDA5	PC or Data Address detector 5 is active. If configured as a dual exact PC detector, same bit as DEPCR2.ACTB. If configured as a range detector, same bit as ARDCR2.ACT.
14 PCDA6	PC or Data Address detector 6 is active. If configured as a dual exact PC detector, same bit as DEPCR3.ACTA. If configured as a range detector, same bit as ARDCR3.ACT.
15 PCDA7	PC or Data Address detector 7 is active. If configured as a dual exact PC detector, same bit as DEPCR3.ACTB. If configured as a range detector, same bit as ARDCR3.ACT.
18 EXD	The Exception Detector is active, same bit as EDCR.ACT
20 CTA	Counter Triad A is active, same bit as PCCSR.ACTA

Table 8-31. Field description of DRASR (continued)

Field	Description
21 CT**B**	Counter Triad A is active, same bit as PCCSR.ACTB
22 RC0	Reloadable Counter 0 is active, same bit as RCCR0.ACT
24 TU	The Trace Unit is active, same bit as the logic OR of all the ACTTM bits of TRSR
28 MONEN	Monitor Enabled - reflects the state of the Monitor Enable SoC input
29 EHEN	External Host Enabled - reflects the state of the Host Enable SoC input

8.5.6.17 Debug Event Status Register (DESR)

A 32-bit read-write register, belonging to both partitions, that captures the occurrence of debug events. Each bit captures one event, regardless of the outcome that event is configured to cause, if at all. Each bit is “sticky”, keeping the fact that the respective event occurred until explicitly cleared by writing “1” to it. A bit is also cleared when the respective event generator is disabled. These event bits can be used for polling-based event detection. Events that are a direct result of VLES execution (such as PCDA events) will cause the respective bit in DESR to be set only if this VLES was executed and committed. If the event is configured to cause entry into debug mode or debug exception, the VLES is considered as killed and the respective bit will not be set in DESR. This behavior is different than that of the respective bit in DMRSR and DERSR, which captures VLES-related events that cause entry into debug mode or debug exception although the VLES itself was killed in the process (by definition).

The register is shared for both the external host and monitor partitions, and includes events that are allocatable to either of them. Each bit can only be cleared by an access originating from the session manager it is allocated to. Clearing the bits has no effect on the logic that generated the event, for example the bit marking an IEU event marks the transition into state *n*, and not the value of the IEU state bit (see Section 8.7.1, “State bits and state change configuration”).

Address 0x110 (DESR) Ownership: Both: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R					IE3	IE2	IE1	IE0		RCE0	CTOB	CTOA		EXD		
W					w1c	w1c	w1c	w1c		w1c	w1c	w1c		w1c		
Reset	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PCDA7	PCDA6	PCDA5	PCDA4	PCDA3	PCDA2	PCDA1	PCDA0								
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-19. Debug Event Status Register (DESR)

Table 8-32 describes the bits in this register.

Table 8-32. Field description of DESR

Field	Description
8 PCDA0	An event from PC or Data Address detector 0 had occurred since this bit was last cleared.
9 PCDA1	A PC detection event from PCDA1 had occurred since this bit was last cleared.
10 PCDA2	An event from PC or Data Address detector 2 had occurred since this bit was last cleared.
11 PCDA3	A PC detection event from PCDA3 had occurred since this bit was last cleared.
12 PCDA4	An event from PC or Data Address detector 4 had occurred since this bit was last cleared.
13 PCDA5	A PC detection event from PCDA5 had occurred since this bit was last cleared.
14 PCDA6	An event from PC or Data Address detector 6 had occurred since this bit was last cleared.
15 PCDA7	A PC detection event from PCDA7 had occurred since this bit was last cleared.
18 EXD	An event from the Exception Detector had occurred since this bit was last cleared
20 CTOA	A Counter Triad A Overflow event had occurred since this bit was last cleared
21 CTO**B**	A Counter Triad B Overflow event had occurred since this bit was last cleared
22 RCE0	An event from Reloadable Counter 0 had occurred since this bit was last cleared
24 IE0	An Indirect Event Unit event 0 had occurred since this bit was last cleared
25 IE1	An Indirect Event Unit event 1 had occurred since this bit was last cleared
26 IE2	An Indirect Event Unit event 2 had occurred since this bit was last cleared
27 IE3	An Indirect Event Unit event 3 had occurred since this bit was last cleared

8.5.6.18 Subsystem Activity Status Register (SASR)

A 32-bit read-only register, accessible by both host and monitor partitions, which reflects the activity status of the various components of the SC3900 subsystem. This status information enables the external host or monitor to ascertain if the various components are idle, or in case they are active, what are they doing. The information is reported per subsystem unit, and is implementation dependent.

Address 0x120 (SASR) Ownership: Both:
Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R									SGBI			CMEI				
W									R			R				
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		L1DCI				L1ICI			CHFS			CS	DM	SS		
W		R				R			R			R	R	R		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-20. Subsystem Activity Status Register (SASR)

Table 8-33 describes the fields of this register.

Table 8-33. Field description of SASR

Field	Description
2-0 SS	Subsystem Status: 000 - Execution 001 - Reserved 010 - Reserved 011 - DOZE 100 - NAP 101 - reserved 11x - reserved
3 DM	Subsystem Debug Mode: 0 -The subsystem is not in debug mode 1 - The subsystem is in debug mode
4 CS	Core status when in debug mode: 0 - No active instruction, the core is ready to accept a new core command or single step 1 - The core did not finish processing a single step or core command When not in debug mode, this bit is always 0.
7-5 CHFS	Core Hold and Freeze Status 000 - The core is neither in hold nor freeze xx1 - Program hold is asserted x1x - Data hold is asserted 1xx - Data freeze is asserted
10-8 L1ICI	L1 Instruction Cache Idle 1xx - All reasons the L1 Icache waits for response from L2 (miss, HWSYNC and more) x1x - Elink Queue is not empty xx1 - The L1 instruction cache is idle
14-12 L1DCI	L1 Data Cache Idle 1xx - All reasons the L1 Dcache waits for response from L2 (miss, HWSYNC and more) x1x - Elink Queue is not empty xx1 - The L1 data cache is idle

Table 8-33. Field description of SASR (continued)

Field	Description
20-16 CMEI	CME Idle: 1xxxx - Active data block channel (may be suspended) x1xxx - Active instruction block channel (may be suspended) xx1xx - Active debug channel (cannot be suspended) xxx1x - Active - other reasons (cannot be suspended) xxxx1 - Architectural suspension: block operations suspended by any of: CCMD.SUSP instruction, suspension in debug mode by DMCSR.FCME, block error, low power mode
23-22 SGBI	Store Gather Buffer Idle: 1x - SGB flush not done x1 - Valid accesses not flushed

8.5.6.19 Debug and Trace Unit Version Register (DTUREV)

A 32-bit read-only register, accessible by both host and monitor partitions, that holds information regarding the hardware components of the Debug and Trace Unit, such as the number and type of detectors, version of the trace hardware, etc. The information in this register is hard-wired.

Address 0x140 (DTUREV) Ownership: Both:
Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R						TUREV				IEUSN				CNTN		
W						R				R				R		
Reset	0	0	0	0	0	U	U	U	U	U	U	U	0	U	U	U

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R							PADGN						DUREV			
W							R						R			
Reset	0	0	0	0	0	0	U	U	U	0	0	0	U	U	U	U

Figure 8-21. Debug and Trace Unit Revision (DTUREV)

The fields of this register are described in Table 8-34.

Table 8-34. Field description of DTUREV

Field	Description
3-0 DUREV	Version number of the Debug Unit (superset of all features in DRCU, ADDU, IEU, PU) Value for B4860 = 0
9-7 PADGN	Number of PC and Address detection groups: 000 - none 001 - 4 detectors 010 - 8 detectors all other combinations - reserved Value for B4860 = 001

Table 8-34. Field description of DTUREV (continued)

Field	Description
18-16 CNTN	Number of profiling counters 000 - none 001 - 3counters 010 - 6 counters all other combinations: reserved Value for B4860 = 010
22-20 IEUSN	Number of IEU state bits 000 - none 001 - 4 all other combinations: reserved Value for B4860 = 001
26-23 TUREV	Version number of the Trace Unit - both message format and programming model Value for B4860 = 0

8.6 PC, Address and Data Detection

The Debug Unit of SC3900 includes an Address and Data Detection Unit (ADDU), which provides a set of dynamic comparison and monitoring features. The main sub-units of the ADDU include:

- 4 PC and data address detector groups
- One task ID comparator filter
- One exception detector

A “detector” in the list above can generate independent debug events (one or more). A “filter” in the list above optionally joins another event as an additional qualifier (see more on events and filters in Section 8.3, “Debug events and filters”). A task ID filter can join nearly any unit (such as a PC detector, trace unit and others). Table 8-35 lists the preciseness of the events that are generated by the various detectors when entering debug mode.

Table 8-35. Preciseness of event detection types

Detection	Preciseness	Comments
PC	Precise	Break before make
Data address	Precise	Break before make
Exception	Precise	Break after sampling into LR and changing PC_NEXT to the ISR, before executing the first VLES of the ISR

8.6.1 PC and Address Detection Groups

The ADDU has four PC and data Address Detection Groups (PADG). Each detection group can be set to one of the following configurations:

- Two independent exact PC detectors “A” and “B”
- One PC range detector

- One data address range detector, implicitly monitoring both core buses. A range detector is also an exact address detector (on both core buses) if the upper and lower reference values are configured to be equal

Each PADG can be allocated as a unit to either the host or monitor partitions.

The detectors in the PADG and their respective output events are numbered serially and named PCAD0 to 7. PADG0 generates events PCAD0 and 1, and so on. Even numbered events are generated by address range detector events, or by Exact PC detector “A” in the PADG. The odd numbers are used by Exact PC detector B in the four PADGs.

Each PADG has 4 control registers:

- Address Range Detector Control Register - controlling how the PADG is configured, and controlling it when configured as a PC or an address range detector
- Dual Exact PC Detector Control Register - controlling the PADG when configured as two independent exact PC detectors
- Two Reference registers, one for each exact PC or one for the upper and one for the lower boundary

8.6.1.1 Address Range Detector Control Register *n* (ARDCR*n*)

A 32-bit read-write register, allocatable to either the host or monitor partitions, and that is used to configure the detection type of the PADG (dual exact PC, PC range or a data address range). It also holds the configuration options for range detection (PC or data address), if so selected. Some of the configuration fields are specific for data address range detection, and do not affect PC range detection. Note that detection of an exact data address is configured as range detection when the upper and lower bounds are equal.

Address 0x250 (ARDCR<*n*>)
offset: <16**n*>
range: *n*=0..3

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	CSEL									LWOSS				LS		IOR
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		STOP				STRT								ID0	ACT	EN
W															R	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-22. Address Range Detector Control Register *n* (ARDCR*n*)

The fields in the register are described in this table.

Table 8-36. Field description of ARDCR*n*

Field	Description
0 EN	Enable 0: Disabled 1: Enabled
1 ACT	Active (read only) 0: Not active 1: Active
2 ID0	Task ID filter 1. Match only of the task ID comparison in comparator 0 matches, see Section 8.6.2.1, “Task ID Comparator Control Register (TIDCCR)” 0: Filter disabled 1: Filter enabled
10-8 STRT	Events for starting detection. See Table 8-37.
14-12 STOP	Events for stopping detection. See Table 8-37.
16 IOR	In or out of range, in range detection: 0: In range 1: Out of range
18-17 LS	Load-Store select: For data address detection, detect only loads, stores or both x1: Detect loads 1x: Detect stores
22 LWOSS	Load Without a Special SYNC: For data address detection, this filter conditions detection to accesses that are not grouped with SYNC.B.DL. This filter is useful for detecting destructive loads that violate the requirement to be grouped with SYNC.B.DL. This bit can be set only load detection. 0: Detection is not conditioned on the existence of SYNC.B.DL in the VLES 1: Only accesses in VLES that is not grouped with SYNC.B.DL are detected
31-30 CSEL	Configuration select: Selects the configuration of the detector: 00: Data address range detection 01: PC address range detection 10: Dual exact PC detection (if selected, the rest of the bits have no effect) 11: Reserved

Each PCDA detector can optionally be activated by 2 start events, and stopped by 2 stop events. These events are IEU state transitions. The list of the detectors and events allocated as filters and start/stop events for each PCDA is listed in Table 8-37. When configured as range detectors, only the even-numbered PCDAs apply.

Table 8-37. Allocation of events for filtering, starting and stopping PC and address Detectors

PCDA #	Start events			Stop events		
	STRT[2]	STRT[1]	STRT[0]	STOP[2]	STOP[1]	STOP[0]
0	IEU Q1	IEU Q0	Reserved	IEU Q1	IEU Q0	Reserved
1	IEU Q1	IEU Q0	Reserved	IEU Q1	IEU Q0	Reserved
2	IEU Q1	IEU Q0	Reserved	IEU Q1	IEU Q0	Reserved
3	IEU Q1	IEU Q0	Reserved	IEU Q1	IEU Q0	Reserved
4	IEU Q3	IEU Q2	Reserved	IEU Q3	IEU Q2	Reserved
5	IEU Q3	IEU Q2	Reserved	IEU Q3	IEU Q2	Reserved
6	IEU Q3	IEU Q2	Reserved	IEU Q3	IEU Q2	Reserved
7	IEU Q3	IEU Q2	Reserved	IEU Q3	IEU Q2	Reserved

8.6.1.2 Dual Exact PC Detector Control Register *n* (DEPCR*n*)

A 32-bit read-write register, allocatable to either the host and monitor partitions, that is used to configure the PADG when configured to operate as a dual exact PC detector in the respective ADCR*n* register, see Section 8.6.1.1, "Address Range Detector Control Register n (ARDCRn)." The register holds two identical portions, each one controlling one exact PC detector.

Address 0x254 (DEPCR<*n*>)
offset: <16**n*>
range: *n*=0..3

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R		STOPB				STRTB								ID0B	ACTB	ENB
W															R	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		STOPA				STRTA								ID0A	ACTA	ENA
W															R	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-23. Dual Exact PC Detector Control Register *n* (DEPCR*n*)

The fields in the register are described in this table.

Table 8-38. Field description of DEPCR*n*

Field	Description
0 ENA	Enable Exact PC detector A 0: Disabled 1: Enabled
1 ACTA	Exact PC detector A is Active (read only) 0: Not active 1: Active
2 ID0A	Task ID filter 0 for exact PC detector A. Match only of the task ID comparison in comparator 0 matches, see Section 8.6.2.1, "Task ID Comparator Control Register (TIDCCR)" 0: Filter disabled 1: Filter enabled
10-8 STRTA	Events for starting detection for exact PC detector A. See Table 8-37.
14-12 STOPA	Events for stopping detection for exact PC detector A. See Table 8-37.
16 ENB	Enable Exact PC detector B 0: Disabled 1: Enabled
17 ACTB	Exact PC detector B is Active (read only) 0: Not active 1: Active
18 ID0B	Task ID filter 0 for exact PC detector B. Match only of the task ID comparison in comparator 0 matches, see Section 8.6.2.1, "Task ID Comparator Control Register (TIDCCR)" 0: Filter disabled 1: Filter enabled
26-24 STRTB	Events for starting detection for exact PC detector B. See Table 8-37.
30-28 STOPB	Events for stopping detection for exact PC detector B. See Table 8-37.

8.6.1.3 PC and Address Detector Reference Register A*n* (PADRRA*n*)

A-32 bit read-write register, allocatable to either the host or monitor partitions, which is used to configure a reference address for PADG that is configured by the respective ADCR*n*. The PADRRA*n* register is used for one of the following functions, depending on the configuration of the CONFIG field in the respective ADCR*n* register:

- If the PADG*n* is configured as PC or data address range, then PADRRA*n* is used to specify the lower bound of the PC or address range
- If the PADG is configured as dual exact PC detectors, then PADRRA*n* is used to specify the exact PC reference for PC detector A (see Figure 8-23).

Address 0x258 (PADRRA<*n*>)
offset: <16**n*>
range: *n*=0..3

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PADRA															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PADRA															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-24. PC and Address Detector Reference Register A*n* (PADRRA*n*)

The fields of this register are described in Table 8-39.

Table 8-39. Field description of PADRRA*n*

Field	Description
31-0 PADRA	PC and Address Reference Value A

8.6.1.4 PC and Address Detector Reference Register B*n* (PADRRB*n*)

A-32 bit read-write register, allocatable to either the host or monitor partitions, which is used to configure a reference address for PADG that is configured by the respective ADCR*n*. The PADRRB*n* register is used for one of the following functions, depending on the configuration of the CONFIG field in the respective ADCR*n* register:

- If the PADG*n* is configured as PC or data address range, then PADRRB*n* is used to specify the upper bound of the PC or address range
- If the PADG is configured as dual exact PC detectors, then PADRRB*n* is used to specify the exact PC reference for PC detector B.

Address 0x25C (PADRRB<*n*>)
offset: <16**n*>
range: *n*=0..3

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PADRB															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PADRB															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-25. PC and Address Detector Reference Register B*n* (PADRRB*n*)

The fields of this register are described in Table 8-40.

Table 8-40. Field description of PADRRB*n*

Field	Description
31-0 PADRB	PC and Address Reference Value B

8.6.1.5 Semantics of PC and Address Detection

This section describes the rules and conventions used in PC and data address detection. PC and data address comparison are performed on effective addresses (addressed as seen by the programmer inside the core). It is possible to filter detection according to the task ID thereby emulating the detection to be on virtual addresses, see Section 8.6.2, “The Task ID comparator.” There is no option to perform detection on the physical address.

Data addresses are compared without specifying which buses they are to be compared with. The PADG monitors both buses and correctly detects the access at whatever bus it uses.

The values configured in a reference register represent a single byte address, both for exact detection or when used as a boundary in range detection. Exact data address detection is configured by specifying an address range where the upper and lower boundaries are equal.

For exact data detection, a match is detected for any relevant access whose address includes the byte configured in the reference. For example, if the reference value is 0x100, all the following accesses will be a detector match:

```
st.l d0,(r0)      ; r0 holds the address 0xFE
st.b d0,(r1)      ; r1 holds the address 0x100
st.2l d0:d1,(r2)  ; r2 holds the address 0xFB
st.x d0,(r3)      ; r3 holds the address 0xFB. st.x is a 64-bit access
```

For range detection, an access is considered “in range” if it satisfies both the lower and upper boundary conditions. The lower boundary condition is if the access includes a byte greater or equal to the lower boundary; the upper boundary condition is if the access includes a byte smaller or equal to the upper boundary condition.

In range detection, there is an option to configure detection as “not in range”. An access is considered “not in range” if either the lower or upper boundary conditions is not met. The lower and upper boundary conditions are as specified earlier for “in range” detection.

Accesses that cross the 4KB non-alignment boundary are split by the memory system into two accesses. Data accesses are monitored in the ADDU as the core issued them, meaning, before the split.

8.6.2 The Task ID comparator

The ADDU has a task ID comparators, which can be used as filters of other event generators:

- PCDA detectors

- The Indirect Event Unit (outside the ADDU)
- The Trace unit (outside the ADDU)

8.6.2.1 Task ID Comparator Control Register (TIDCCR)

A 32-bit read-write register, Accessible to both host and monitor partitions, which is used to configure the task ID comparator. The comparator could be allocated to either the host or monitor partition, so both are allowed to write to the register, however only the debug master to which a certain task ID comparator was allocated will be able to modify the configuration fields that control that comparator.

Address 0x350 (TIDCCR) Ownership: Both: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R				MASK0						IDSEL0						EN0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-26. Task ID Comparator Control Register (TIDCCR)

The fields of this register are described in Table 8-41.

Table 8-41. Field description of TIDCCR

Field	Description
0 EN0	Enable TID comparator 0. The field is writable only for the partition to which this comparator is allocated to. 0: Disabled 1: Enabled
6-4 IDSEL0	ID select for comparator 0. The field is writable only for the partition to which this comparator is allocated to. Only PROCID could be selected. 000: reserved 001: reserved 010: reserved 011: PROCID 1xx: reserved
12-8 MASK0	Comparison mask for comparator 0. (2^{mask} - 1) will be used to mask the reference value for comparator 0. A field value of 0 means unmasked. The field is writable only for the partition to which this comparator is allocated to.

When PROCID is selected, it is implicitly conditioned with SR2.EXP = 0. This implicit condition ensures that the detection is performed when the task is issued, and the comparison will not catch “residue” events of the task switch routine itself.

8.6.2.2 Task ID Comparator Reference Register 0 (TIDCRR0)

A 32-bit read-write register, allocatable either to the host or monitor partitions, holding the value that is used as reference for the task ID comparator.

Address 0x354 (TIDCRR<*n*>)
offset: 4
range: 0

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	TIDCF0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	TIDCF0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-27. Task ID Comparator Reference Register 0 (TIDCRR0)

The fields of this register are described in Table 8-42.

Table 8-42. Field description of TIDCRR0

Field	Description
31-0 TIDCR0	TIDCR0 holds the reference value for TID comparator 0

8.6.3 The Exception Detector

The exception detector can generate an event when a particular exception has occurred. The detector can be configured to identify a specific exception, or an exception class, based on a 16-bit value that is sampled to the Exception ID Register (EIDR) in the core. Note that interrupts from the interrupt controller in the subsystem (RPIC) are driven with their index number, identifying them as one of the 512 interrupts that enter the EPIC. This number is sampled into the EID field in EIDR. For a description of the EIDR register and of the values that are sampled into it see the *SC3900 FVP Core Reference Manual*.

The programming model of the Exception Detector includes a control register, a reference value register and a mask register.

8.6.3.1 Exception Detector Control Register (EDCR)

A 32-bit read-write register, allocatable either to the host or the monitor partition, that controls the operation of the Exception Detector.

Address 0x370 (EDCR) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R															ACT	EN
W															R	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-28. Exception Detector Control Register (EDCR)

The fields of this register are described in Table 8-43.

Table 8-43. Field description of EDCR

Field	Description
0 EN	Enable 0: Disabled 1: Enabled
1 ACT	Active (read only) 0: Not active 1: Active

Note that this detector does not have the usual settings for task ID filtering, this is because Task ID comparison is inherently disabled during an exception (SR2.EXP=1).

8.6.3.2 Exception Detector Reference Value Register (EDRVR)

A 32-bit read-write register, allocatable to either the host or the monitor, that holds the value the exception detector will compare with EIDR when an exception occurs. The format of the register is identical to that of the EIDR core register.

Address 0x374 (EDRVR) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	ETP_REF						EID_REF									
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-29. Exception Detector Reference Value Register (EDRVR)

The fields of this register are described in Table 8-42.

Table 8-44. Field description of EDRVR

Field	Description
9-0 EID_REF	Reference value for the EID field in the core register EIDR
15-10 ETP_REF	Reference value for the ETP field in the core register EIDR

8.6.3.3 Exception Detector Mask Register (EDMR)

A 32-bit read-write register, that holds a 16-bit mask that defines, per bit, which bits of EIDR and EDRVR will be compared upon an exception. The mask bits affect the respective positions of EDRVER. This mask enables to identify a set of exceptions, in particular when applied in positions that match the ETP field. This enables to detect, for example, only MMU exceptions, only TRAP.*n* exceptions for which the user specified a specific immediate field, or only non-critical interrupts with a power-of-2-aligned index range.

A bit that is set in the mask enables the comparison of the respective bit in EIDR and EDRVR.

Address 0x378 (EDMR)

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	ETPM						EIDM									
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-30. Exception Detector Mask Register (EDMR)

The fields of this register are described in Table 8-45.

Table 8-45. Field description of EDMR

Field	Description
9-0 EIDM	Mask for the EID_REF field and the EID field in the core register EIDR
15-10 ETPM	Mask for the EID_REF field and ETP field in the core register EIDR

8.7 Indirect events

Indirect events work through four intermediate state bits (Q0–Q3). Instead of specifying the outcome of each causing events directly, the user specifies for input (causing) events what state bits are set as a result, and in what condition. The transitions into a state can in turn generate an outcome in a target unit (such as

enabling trace), which is configured directly in the target unit. The advantages of indirect event configurations, relative to direct configuration, are:

- Much more connectivity options between input events and outcomes - all debug events are connected to the IEU and can cause state transitions. In case the specific cause-outcome connectivity is missing in direct configuration, it can be configured indirectly
- Having 4 states enables to configure user-defined state machines, and detect complex event sequences (not just simple cause-outcome connections)

8.7.1 State bits and state change configuration

The IEU has 4 state bits, named Q0 through Q3. These bits hold the event detection history, as configured by the user. The user configures which input events, and in what conditions, set or clear a state bit. Transition into a state can trigger an outcome in the target units (as configured in the target unit).

The input events are configured to modify the state bits in one of two ways:

- Unconditionally: an input event causes the specified state bit (the destination state) to be set, regardless of the previous state of the Q bits.
- Conditionally: an input event causes the specified state bit (the destination state) to be set, only if a specified state bit (the source state) was set when the event occurred. The source state is cleared in the process.

The following bullets summarize the conditions in which state bits are set and cleared.

A state bit can be set in one of the following conditions:

- By direct initialization, when the IEU is configured
- Upon occurrence of an event, configured to unconditionally set this state bit
- Upon occurrence of an event, configured to conditionally set this state bit, if the source state on which it was conditioned was set

A state bit can be cleared in one of the following conditions:

- By direct initialization, when the IEU is configured
- Upon occurrence of an event, configured to conditionally leave this state if it was set to another state, if there is no other event attempting to set this state bit

Conditional events can be used to identify event sequences and map them to state transitions. Figure 8-31 below illustrates an example where a sequence of four events is identified.

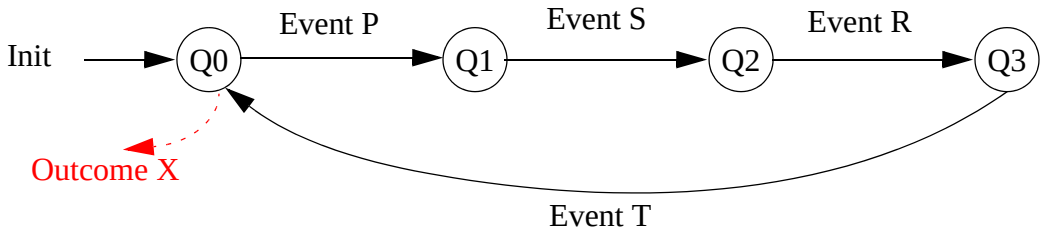

Figure 8-31. Example 1: Identification of a sequence of 4 events (exclusive)

In this example, the IEU is initialized with the state of Q0 being “1”, and all the other state bits as “0”. Events P, S, R and T are configured as conditional events as follows:

- Event P is conditioned on Q0 being set. When P occurs, if Q0 was set, Q1 is set and Q0 is cleared
- Event S is conditioned on Q1 being set. When S occurs, if Q1 was set, Q2 is set and Q1 is cleared
- Event R is conditioned on Q2 being set. When R occurs, if Q2 was set, Q3 is set and Q2 is cleared
- Event T is conditioned on Q3 being set. When T occurs, if Q3 was set, Q0 is set and Q3 is cleared

In the destination unit, outcome X is enabled upon a transition to Q0 (initialization is not a transition). This is a classic state machine implementation, and of course any transition scheme using 4 states can be defined (not only a simple serial sequence). Note that the configuration above detect an exclusive sequence, meaning that only one sequence can be monitored at a time; If another P event occurs before the transition from Q3 back to Q0, it is ignored.

Unconditional events enable to detect non-exclusive, or pipelined, event sequences. Figure 8-32 describes a configuration to detect the same 4-event sequence but this time allowing several sequences to be monitored concurrently.

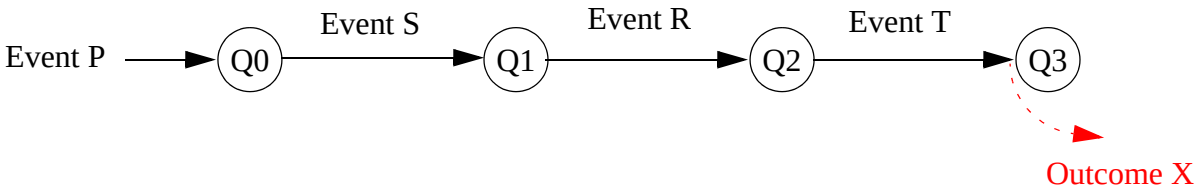

Figure 8-32. Example 2: Identification of a sequence of 4 events (pipelined)

In this example, the IEU is initialized with all the states as “0”. Event P is configured as an unconditional event, while S, R and T are configured as conditional events as follows:

- Event P is unconditional. Whenever P occurs, Q0 is set
- Event S is conditioned on Q0 being set. When S occurs, if Q0 was set, Q1 is set and Q0 is cleared
- Event R is conditioned on Q1 being set. When R occurs, if Q1 was set, Q2 is set and Q1 is cleared
- Event T is conditioned on Q2 being set. When T occurs, if Q2 was set, Q3 is set and Q2 is cleared

In the destination unit, outcome X is enabled upon a transition to Q3. Note that in this configuration, there is no event that clears Q3. After the first sequence is detected, Q3 remains set. However, the outcome event is identified upon the occurrence of an event into Q3 (regardless if Q3 was set or not). Also, there is no event that is conditioned on Q3, hence there is no need to clear it. Upon any occurrence of P, a new sequence can advance in the “pipe” of states Q0 to Q3, enabling to monitor several event sequences concurrently (as long as they retain their order and do not bypass each other).

Having 4 state bits with direct initialization enables to split the configuration into independent sequences that are not connected to each other. Figure 8-33 Illustrates such a case.

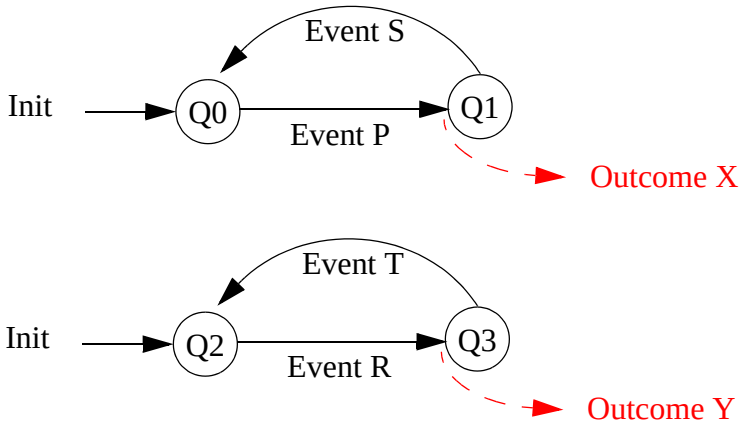

Figure 8-33. Example 3: Two independent detection sequences

In this example, the IEU is initialized with Q0 and Q2 as “1”, and Q1 and Q3 as “0”. All events P, S, R and T are configured as conditional events as follows:

- Event P is conditioned on Q0 being set. When P occurs, if Q0 was set, Q1 is set and Q0 is cleared
- Event S is conditioned on Q1 being set. When S occurs, if Q1 was set, Q0 is set and Q1 is cleared
- Event R is conditioned on Q2 being set. When R occurs, if Q2 was set, Q3 is set and Q2 is cleared
- Event T is conditioned on Q3 being set. When T occurs, if Q3 was set, Q2 is set and Q3 is cleared

Outcome X is configured to occur upon a transition into Q1, while outcome Y is configured to occur upon transition to state Q3. Of course there is no obligation to configure the two independent sequences in such a symmetrical way as in this example.

Figure 8-34 shows two equivalent configurations to specify a simple connection of an input event to an outcome using one state bit (Q0 as an example). This is the most “degenerate” form of state configuration.

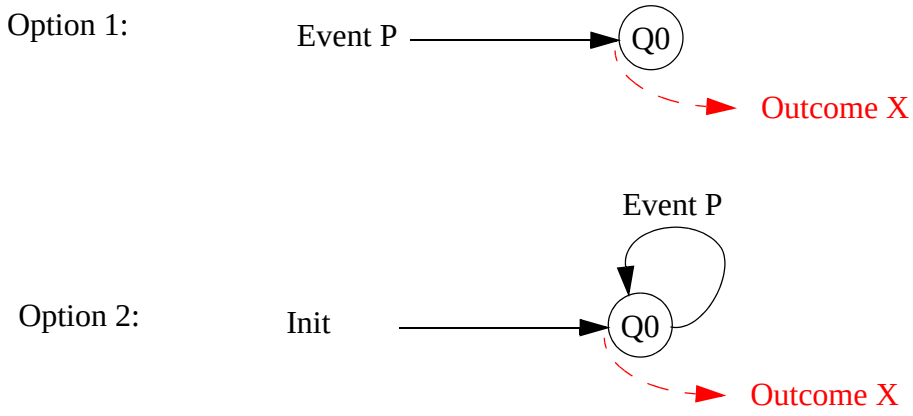

Figure 8-34. Example 4: Single input-output connection options

In option 1, the IEU is initialized so that Q0 is “0”, however it is not mandatory (the configuration will work regardless of the initial state of Q0). Event P is configured as an unconditional event: whenever P occurs, Q0 is set. In option 2, the IEU is initialized so that Q0 is “1”, and event P is configured as a conditional event causing a “transition” to Q0.

In both options, the transition to Q0 is configured in the target unit to cause Outcome X. After P occurs, Q0 remains set until the IEU is re-configured. The other state bits can be used for other configurations (as 3 other individual input-outcome connections, or as part of a more complex configuration).

8.7.2 IEU programming model

The IEU programming model has six registers, as follows:

- IECTLS—Control and Status Register
- IECTR*n*—4 Conditional Transition configuration Registers (*n* = 0..3)
- IEUTR—Unconditional Transition configuration Register

Input events are configured to cause state transitions. Each Transition register handles 3 independent events, each enabling to perform a change in a destination state bit. Conditional transitions that share the same condition (the source state bit they depend on) are grouped in the same IECTR. So for example register IECTA0 is used to configure up to 3 input events that may cause a state transition from Q0. Each input event configured in this register will cause a state transition if it occurred while Q0 was set. Similarly, IECTA1 is used to configure up to 3 input events that may cause a state transition from Q1. IEUTR is used to configure unconditional events.

Each IEU bit is considered as an independent resource that could be allocated to either the monitor or the external host partition. The IECTR*n* registers are related to each state bit, so writing to them is restricted to the debug master they are allocated to. The IECTLS and IEUTR registers are shared between the two masters so could be written to by both, the separation between the external host and the monitor is enforced at the field level in these registers.

The following sections describe the structure of each of these registers.

8.7.2.1 Indirect Event Unit Control and Status Register (IECTLS)

A 32-bit read/write register, accessible to both the host and the monitor, which holds the state of the 4 state bits. In addition it holds the ID comparator filter settings for each debug master (monitor and host) for the unconditional events of IEUTR. Each of the state bits in this register (Q*n*) is writable only by the debug master it is allocated to (host or monitor). The fields defining the filter setting for IEUTR are writable only by the respective debug master. Note that the IEU does not have an enabling bit - it is active when configured to respond to an event.

Address 0x3A0 (IECTLS) Ownership: Both: Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																ID0UH
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								ID0UM					Q3	Q2	Q1	Q0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-35. Indirect Event Unit Control and Status Register (IECTLS)

The fields of this register are described in this table.

Table 8-46. Field description of IECTLS

Field	Description
0 Q0	Reflects the state of the Q0 bit. Could be read or written to. Writing is possible only for the partition to which it is allocated.
1 Q1	Reflects the state of the Q1 bit. Could be read or written to. Writing is possible only for the partition to which it is allocated.
2 Q2	Reflects the state of the Q2 bit. Could be read or written to. Writing is possible only for the partition to which it is allocated.
3 Q3	Reflects the state of the Q3 bit. Could be read or written to. Writing is possible only for the partition to which it is allocated.
8 ID0UM	Task ID filter using comparator 0 for unconditional events (monitor). This filter affects events defined in IEUTR which relate to Qn bits that are allocated to the monitor partition 0: ID filer 0 not active 1: ID filter 0 Active
16 ID0UH	Task ID filter using comparator 0 for unconditional events (host). This filter affects events defined in IEUTR which relate to Qn bits that are allocated to the host partition 0: ID filer 0 not active 1: ID filter 0 Active

8.7.2.2 Indirect Event Conditional Transition Configuration Registers (IECTR*n*)

A 32-bit read-write register, allocatable either to the host or monitor partition. The register with index *n* configures up to three events that have an effect only if the state bit Q*n* is set (*n* = 0–3). The task ID filter setting is shared between the events in each register. It is the user's responsibility to set the destination states (SD*n* fields) only to state bits that belong to the same debug master (host or monitor)

Address 0x3B0 (IECTR<*n*>)
offset: 8
range: 0..3

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	SD2				IN2					SD1			IN1			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	IN1		SD0		IN0											ID0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-36. Indirect Event Conditional Transition Configuration Register *n* (IECTR*n*)

The fields of this register are described in this table.

Table 8-47. Field description of IECTR*n*

Field	Description
0 ID0	If set, condition input events on a match on task ID comparator 0
11-5 IN0	Specifies input event 0. If this event occurs and Q*n* (*n* is the register index) is set, will perform the transition to the state specified in SD0. The list of events is specified in Table 8-48.
13-12 SD0	State destination: if the event specified in IN0 is qualified, Q*n* is cleared and the Q bit specified in SD0 will be set. 00: Q0 01: Q1 10: Q2 11: Q3
20-14 IN1	Specifies input event 1. If this event occurs and Q*n* (*n* is the register index) is set, will perform the transition to the state specified in SD1. The list of events is specified in Table 8-48.
22-21 SD1	State destination: if the event specified in IN1 is qualified, Q*n* is cleared and the Q bit specified in SD1 will be set. 00: Q0 01: Q1 10: Q2 11: Q3

Table 8-47. Field description of IECTR*n* (continued)

Field	Description
29-23 IN2	Specifies input event 2. If this event occurs and Q*n* (*n* is the register index) is set, will perform the transition to the state specified in SD2. The list of events is specified in Table 8-48.
30-31 SD2	State destination: if the event specified in IN2 is qualified, Q*n* is cleared and the Q bit specified in SD2 will be set. 00: Q0 01: Q1 10: Q2 11: Q3

The input events that could be configured in fields IN0–IN2 are listed in this table.

Table 8-48. Input events for the IEU

Code	Event	Comments	Code	Event	Comments
0	disabled	No change of state	60	Triad counter A0 overflow	—
1	always	The state will change on the next cycle	61	Triad counter A1 overflow	—
2-7	reserved	—	62	Triad counter A2 overflow	—
8	reserved	—	63	Triad A overflow	—
9	reserved	—	64	Triad counter B0 overflow	—
10	reserved	—	65	Triad counter B1 overflow	—
11	reserved	—	66	Triad counter B2 overflow	—
12	reserved	—	67	reserved	—
13	reserved	—	68	Reloadable counter 0 RZ	—
14	reserved	—	69	reserved	—
15	reserved	—	74	reserved	—
16	pcadd[0]	—	75	reserved	—
17	pcadd[1]	—	76	Entry to DOZE	—
18	pcadd[2]	—	77	reserved	—
19	pcadd[3]	—	78-79	Reserved	—
20	pcadd[4]	—	80	Entry to low power mode (doze)	—
21	pcadd[5]	—	81	Reserved	—

Table 8-48. Input events for the IEU (continued)

Code	Event	Comments	Code	Event	Comments
22	pcadd[6]	—	82	Task switch	Previous TID/PROCID change reported upon SR2.EXP 1 → 0 (same condition as for ownership trace)
23	pcadd[7]	—	83-85	Reserved	—
24-27	Reserved	—	86-87	SOC input 0-1	—
28-31			88-89	Reserved	—
32	reserved	—	90	reserved	—
33	reserved	—	91	PCU false BTB hit	—
34	Exception detection	—	92-95	Reserved	—
35-39	Reserved	—	96-127	Reserved	—
40	DALU overflow	—	—	—	—
41	DALU saturation	—	—	—	—
42-49	Reserved	—	—	—	—
50	DEBUGEV.0	—	—	—	—
51	DEBUGEV.1	—	—	—	—
52	DEBUGEV.2	—	—	—	—
53	DEBUGEV.3	—	—	—	—
54	DEBUGEV.4	—	—	—	—
55	DEBUGEV.5	—	—	—	—
56	DEBUGEV.6	—	—	—	—
57	DEBUGEV.7	—	—	—	—
58-59	Reserved	—	—	—	—

8.7.2.3 Indirect Event Unconditional Transition Configuration Register (IEUTR)

A 32-bit read-write register, which is accessible for both the host and monitor partitions. The register configures events that can unconditionally set each state bit. Each event field is writable only by the debug master to which the respective state bit is allocated. The Work mode and task ID filters for these events (per partition) are defined in IECTLS register.

Address 0x3E0 (IEUTR) Ownership: Both:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R		INQ3								INQ2						
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		INQ1								INQ0						
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-37. Indirect Event Unconditional Transition Configuration Register (IEUTR)

The fields of this register are described in this table.

Table 8-49. Field description of IEUTR

Field	Description
6-0 INQ0	Specifies an input event that causes an unconditional setting of Q0 (if the filters in IECTLS qualify it). The list of events is specified in Table 8-48.
14-8 INQ1	Specifies an input event that causes an unconditional setting of Q1 (if the filters in IECTLS qualify it). The list of events is specified in Table 8-48.
22-16 INQ2	Specifies an input event that causes an unconditional setting of Q2 (if the filters in IECTLS qualify it). The list of events is specified in Table 8-48.
30-24 INQ3	Specifies an input event that causes an unconditional setting of Q3 (if the filters in IECTLS qualify it). The list of events is specified in Table 8-48.

8.7.2.4 IEU register configuration example

The requested scenario: The outcome Y is to be generated after identifying that events P and R occurred, in any order, in a pipelined fashion. For example, when the sequence P-R-P occurs, two Y outcomes should be reported. The state diagram that implements this scenario is illustrated in this figure.

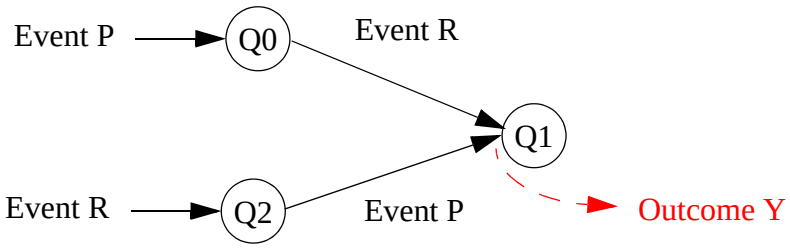

Figure 8-38. State transition diagram of the IEU configuration example

The registers of the IEU and the target unit should be configured as follows:

- IECTLS: Q0..Q3=0000
- IECTR0: IN0=event R, SD0=Q1
- IECTR1: no configuration
- IECTR2: IN0=event P, SD0=Q1
- IECTR3: no configuration
- IEUTR: INQ0=event P; INQ1=event R
- The configuration register at the target subunit generating outcome Y should be configured to generate outcome Y upon a transition to Q1.

8.8 Event counting

Event counting in the SC3900 subsystem is performed in the profiling unit. This unit includes 7 counters, which could be configured to count a wide array of core and subsystem events. The counters are organized in the following subgroups:

- Two counter triads (6 total), which are configured to count a pre-defined set of related events
- A general reloadable counter, which is used for control, periodic event generation and to monitor real time intervals

The triad counters always count together with a shared configuration, counting a fixed set of events out of a pre-defined selection of event triplets. Both triads must be assigned to the same partition.

A counter can be configured with the following parameters:

- The event that is being counted
- The initial value of the counter, which determines after how many events the counter will overflow.
- Start events: events that dynamically activate the counter
- Stop event: events that dynamically stop the counter
- Sampling event: events that sample the value of the counter to a shadow register, for later reading by software or directing to the trace stream
- Reset event: events that dynamically reset the counter
- Counting filters: conditions that determine, per event, if the event will be counted or not
- For the reloadable counter, there are additional features that are supported specifically for it, described in Section 8.8.3, “Reloadable counter registers.”

In a typical profiling session, the user defines the events to be counted and the other counter settings. In particular, the user defines the set of events in which the counters are sampled into the snapshot registers, so that they could be reported to the user. The PU supports two schemes of off-loading the counter values:

- Automatic, using the trace unit. When properly configured, a sampling event causes the trace unit to generate profiling messages which dumps portions of the snapshot registers into the trace stream. The trace unit has a timestamp counter, which adds the timestamp to the messages, hence the events counted by the profiling counters normally do not include raw cycles.

- By software: When properly configured, upon a sampling event, a debug exception is generated, and pre-defined code reads the snapshot registers and uploads them to the user-defined location. The reloadable counter could be configured to count raw cycles, which are also sampled into a snapshot register upon the sampling event, to be read by the driver code with the counter snapshot registers.

Although the programming model of the two triads enables independent activation, the two must be activated with shared conditions by setting the PCCSR.SDTP bit. This setting defines that the counting mode, filtering, start, stop, reset and sampling conditions for both triads is as selected for triad A.

8.8.1 Registers of the profiling unit

The PU has the following registers:

- Profiling Control and Status Register
- Triad control registers A and B
- Counter A0, A1, A2, B0, B1, B2 value registers
- Counter A0, A1, A2, B0, B1, B2 snapshot registers
- Reloadable Counter 0 control register
- Reloadable Counter 0 value register
- Reloadable Counter 0 reload register
- Reloadable counter 0 snapshot register

8.8.2 Triad registers

This section describes the registers of the two triads. The structure of these registers is nearly identical for the two triads. They differ in the list of events that they count. The PCCSR enables to configure that the control settings of Triad A apply also to Triad B, and that the overflow event of triad A will also reflect the overflow condition from triad B.

8.8.2.1 Profiling Counters Control and Status Register (PCCSR)

A 32-bit read-write register, accessible for both the host and monitor partitions, which holds general configuration bits for the profiling triads, and the activation status of each. The register is writable by both debug session managers (the monitor and the external host), however fields belonging to a specific triad can only be written to by a debug master if the triad is allocated to its partition.

Address 0x400 (PCCSR)

Ownership: Both:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R								OVFB2	OVFB1	OVFB0	OVFB	ACTB		CMB		ENB
W								w1c	w1c	w1c	w1c	R				
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								OVFA2	OVFA1	OVFA0	OVFA	ACTA	SDTP	CMA		ENA
W								w1c	w1c	w1c	w1c	R				
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-39. Profiling Counters Control and Status Register (PCCSR)

The fields of this register are described in this table.

Table 8-50. Field Description of PCCSR

Field	Description
0 ENA	Enable Triad A 0: Disabled 1: Enabled
2-1 CMA	Counting Mode of Triad A 00: 32-bit counting, Upon overflow of any counter of the triad the respective OVFAn bit is set, the triad OVFA bit is set, a triad and counter event is generated, and the counter wraps around and continues to count 01, 10: reserved 11: Any input event to a counter sets the respective OVFA*n* bit, the triad OVFA bit is set, a triad and counter event is generated. The counter increments normally. In this mode, the triad counters function as a mux of the input events to the IEU, via the counter output events.
3 SDTP	Synchronous Dual Triad Profiling 0: Start, stop, sampling, clearing, filtering and counting mode settings for triad B are defined in its control register (TCRB). The overflow state of Triad A (OFVA) reflects only the events of the counters of Triad A. 1: Start, stop, sampling, clearing, filtering and counting mode settings for triad B are defined by the configuration of Triad A (TCRA). The overflow state of Triad A (OFVA) reflects all 6 counters. ENA and ENB must both be set in this mode. This bit must be set in this revision.
4 ACTA	Active status of Triad A 0: Not active 1: Active
5 OVFA	Combined Overflow status of Triad A. This sticky bit combines the overflow status from the 3 counters of triad A (if SDTP is clear) or from all the 6 counters (if SDTP is set). The bit remains set until cleared by writing 1 to it. 0: An overflow event did not occur since the last time this bit was cleared. 1: An overflow event occurred in one of the counters (A counters or A+B counters, depending on SDTP) since this bit was last cleared
6 OVFA0	Overflow status of counter A0 0: An overflow event did not occur in A0 since the last time this bit was cleared. 1: An overflow event occurred in A0 since the last time this bit was cleared

Table 8-50. Field Description of PCCSR (continued)

Field	Description
7 OVFA1	Overflow status of counter A1 0: An overflow event did not occur in A1 since the last time this bit was cleared. 1: An overflow event occurred in A1 since the last time this bit was cleared
8 OVFA2	Overflow status of counter A2 0: An overflow event did not occur in A2 since the last time this bit was cleared. 1: An overflow event occurred in A2 since the last time this bit was cleared
16 ENB	Enable Triad B 0: Disabled 1: Enabled The setting of this field should be the same as for Triad A (ENA)
18-17 CMB	Counting Mode of Triad B - to be configured when activating Triad B if SDTP bit is clear 00: 32-bit counting, Upon overflow of any counter of the triad the respective OVFBn bit is set, the triad OVFB bit is set, a triad and counter event is generated, and the counter wraps around and continues to count 01, 10: reserved 11: Any input event to a counter sets the respective OVFB*n* bit, the triad OVFB bit is set, a triad and counter event is generated. The counter increments normally. In this mode, the triad counters function as a mux of the input events to the IEU, via the counter output events.
20 ACTB	Active status of Triad B 0: Not active 1: Active
21 OVFB	Combined Overflow status of Triad B. This sticky bit combines the overflow status from the 3 counters of triad B. If the SDTP bit is set, this bit remains zero. The bit remains set until cleared by writing 1 to it. 0: An overflow event did not occur in triad B counters (if SDTP is set) since the last time this bit was cleared. 1: An overflow event occurred in one of the counters of triad B since this bit was last cleared
22 OVFB0	Overflow status of counter B0 0: An overflow event did not occur in B0 since the last time this bit was cleared. 1: An overflow event occurred in B0 since the last time this bit was cleared
23 OVFB1	Overflow status of counter B1 0: An overflow event did not occur in B1 since the last time this bit was cleared. 1: An overflow event occurred in B1 since the last time this bit was cleared
24 OVFB2	Overflow status of counter B2 0: An overflow event did not occur in B2 since the last time this bit was cleared. 1: An overflow event occurred in B2 since the last time this bit was cleared

8.8.2.2 Profiling Triad Control Register A, B (PTCRA/B)

Two identical 32-bit read-write registers, allocatable either to the host or monitor partition, each including control and status bits for one triad (A or B). In PTCRB, only the CEV field can be configured, the other fields have no effect because triad B inherits these settings from Triad A due to the setting of the PCCSR.SDTP bit, see Section 8.8.2.1, “Profiling Counters Control and Status Register (PCCSR)”.

Address 0x410 (PTCRA)
0x480 (PTCRB)

Ownership: Config:
Access: S: RW

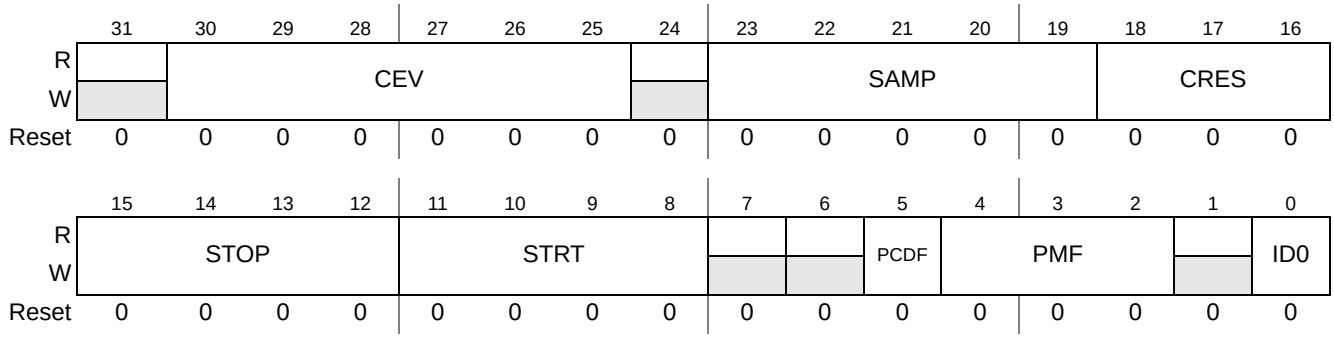

Figure 8-40. Profiling Triad Control Register A, B (PTCRA/B)

The fields of this register are described in this table.

Table 8-51. Field description of PTCRA/B

Field	Description
0 ID0	If set, consider events only for task ID comparison of ID comparator 0.
4-2 PMF	Privilege & Mode Filter: Consider events only if the working mode of the core matches, see Table 8-6.
5 PCDF	PC detector filter - consider events only if the PCDA detector signaled a hit for this cycle (for asynchronous events) or this VLES (for VLES-based events). PCDA can be configured only for PC range detection, see Table 8-52. 0 - no PC detector filtering 1 - with PC detector filtering
11-8 STRT	Events for starting the count. See Table 8-52. Any enabled event will start the count.
15-12 STOP	Events for stopping the count. See Table 8-52. Any enabled event will stop the count.
18-16 CRES	Events for resetting the counters of the triad. See Table 8-53. Upon any of the enabled reset events, the value of the triad counters is set to 0.
23-19 SAMP	Events for sampling the counters of the triad to the shadow registers. See Table 8-53. Sampling occurs upon any of the enabled samplings events.
30-25 CEV	Counted events: each value of the field defines the group of 3 events counted in this triad. The list of events is described in Table 8-55.

The list of units and events that are allocated to filter, start and stop counting of the profiling counters is listed in this table.

Table 8-52. Allocation of events for filtering, starting and stopping profiling counters

Triad counters	PC filter	Start events				Stop events			
		STRT[3]	STRT[2]	STRT[1]	STRT[0]	STOP[3]	STOP[2]	STOP[1]	STOP[0]
A0, A1, A2	PCDA 0	IEU Q2	IEU Q0	DEBUG EV0	reserved	IEU Q3	IEU Q1	DEBUG EV1	reserved
B0, B1, B2	PCDA 4	IEU Q2	IEU Q0	DEBUG EV4	reserved	IEU Q3	IEU Q1	DEBUG EV5	reserved

The conditions for sampling the triad registers into the shadow registers and for clearing the triad registers are listed in this table.

Table 8-53. Allocation of events for sampling and resetting profiling counters

Triad	Sample events					Reset events		
	SAMP[4]	SAMP[3]	SAMP[2]	SAMP[1]	SAMP[0]	CRES[2]	CRES[1]	CRES[0]
A	IEU Q0	reserved	triad overflow	INT/SBR	DEBUG EV.2	IEU Q1	sample event	start event

The sampling events, which sample the triad counters to the snapshot registers, are defined by bits that enable an individual event. When more than one bit is set, sampling occurs when any of the enabled events occurs. The events that may cause sampling are:

- A DEBUGEV.*n* instruction
- Identification of a routine boundary: jump to an exception, RTE* instruction, JSR/BSR and RTS* instruction
- An overflow of any of the counters in the triad.
- An IEU state transition event

If trace and profiling messages are enabled, the reason for sampling is reported in the message, see Table 8-85. In case more than one event occurred on the same cycle, the reported sampling reason is according to the list above (DEBUGEV.*n* has the highest priority).

The following priority applies to events affecting the activity status of the profiling counters:

1. Enable
2. Stop
3. Start
4. Disable

For example that if a start and a stop event occur in the same cycle, the stop event has precedence. These events affect the ACT status bit on the next cycle.

When the counter is in the Active state, the following priority applies between events that occur on the same cycle:

1. Reset
2. Count
3. Sample

For example: if a count event and a reset event occur together, the counter will be reset (have a value of 0 and not 1). In case the counter is near overflowing, overflow will not occur (except for CMA/B=11 which generates overflow for every input event). If a count and sample events occur together, the sampled value will not include the new count event (but the counter will count).

8.8.2.3 Summary of events counted in the profiling counters

This section lists the events that could be counted by the profiling counters. The events are organized into groups of 3, each counted by either Triad A or B. In many cases, two events groups were defined for concurrent profiling, as they belong to the same event category. In many cases, if the events counted in a triad or in two triads are consistent, the user may calculate additional event categories as sum or differences of counted events.

The following table summarizes the different counting groups.

Table 8-54. Summary of event groups for profiling

Event category	Event group	Triad	Tag	Counter 0	Counter 1	Counter 2	Calculated entities	Comments
Core events - interrupts	Exceptions 1	A	CE1	Trap instructions	Critical interrupts	Non-critical interrupts	—	—
	Exceptions 2	B	CE2	MMU exceptions	Debug exceptions	Other exceptions	—	—
Core events - branch prediction	Branch predication 1: High level	A	CBP1	Total VLES	COF VLES (w/o hardware loops)	COF correctly predicted (w/o hardware loops)	Sequential VLES (including loops) = A0-A1 Wrongly predicted COF = A2-A1	Includes both taken and not-taken COFs.
	Branch prediction 2: BTB	B	CBP2	All BTB-able COF	BTB-able COFs, wrongly predicted, not in the BTB	BTB-able COFs, wrongly predicted, in the BTB	Non-BTB-able COF (incl. TFETCH) = A1-B0	Includes both taken and not-taken COFs.
	Hardware loop prediction	B	CLP	Total hardware end-of-loop COF	Incorrectly predicted buffered end-of-loop COF	Other incorrectly predicted end-of-loop COF (not in the BTB, in the BTB wrongly predicted, loop re-learn)	Total incorrect hardware loop prediction = B1+B2 Total correct prediction = B0-B1-B2	Buffered loops: those sequential loops which fit in the buffer
Core cycles	Cycle high level	A	CCH	Application cycles	—	Bubbles	Execution cycles = A0-A2	App. cycles: not in debug and low power modes
	Stall cycles 1	A	CCS1	No bubbles (once per VLES)	COF (pipe flush cycles)	interlocks (RSU) w/o holds	—	—
	Stall cycles 2	B	CCS2	Data memory holds and freezes	Program starvation	Rewind cycles (prog & data)	Application cycles = sum of A0 to B2	—
	Rewind events	B	CRW	Data rewind events	Program rewind events	DTU rewind events	—	—
Debug events	Debug Events 1	A	ED1	IEU Q0	IEU Q1	Reload counter 0 RZ	—	—
	Watchpoints 1	A	EW1	—	PCDA0	PCDA2	—	—
	DEBUGEV 1	A	EIN1	DEBUGEV.0	DEBUGEV.2	DEBUGEV.3	—	—

Table 8-54. Summary of event groups for profiling (continued)

Event category	Event group	Triad	Tag	Counter 0	Counter 1	Counter 2	Calculated entities	Comments
Instruction fetching	L1 Icache access sorting	A	IASR	Program access L1 hits	Program access L1 pre-fetch hits	Program access L1 miss	Total program access: A0+A1+A2	—
	Program L1→L2 cacheable access sorting	B	IL12	L2 Program access hits (shared/exclusive)	L2 Program access hits (modified)	L2 Program access miss	Total L1→L2 cacheable program accesses = B0+B1+B2	Only cacheable accesses, including CME accesses
	L1 Program pre-fetch accesses	A	IPF	CME/granular PF accesses - L1 hit (including pre-fetch hit)	—	CME granular - L1 miss	Total CME granular accesses: A0+A1+A2	Only program CME and cache granular accesses. ctr1 does not count
	Program access holds	B	IAH	Program cacheable hold + contention cycles	Program non-cacheable holds + miscellaneous	Program rewind events	Program holds = B0+B1	The core may continue to execute during program holds
Data channel - general	Data access holds - HL	B	DGAH	Data cache holds	Data store freeze	Data rewind events	—	B0 and B1 events may overlap
	Coherency - invalidation events	A	DGCI	L1 data invalidate from L2	CME/granular invalidation accesses	L1 data invalidation from internal conflicts	—	Only actual invalidation is counted

Table 8-54. Summary of event groups for profiling (continued)

Event category	Event group	Triad	Tag	Counter 0	Counter 1	Counter 2	Calculated entities	Comments
Data Loads	L1 Dcache load access sorting	A	DLSR	Data access L1 hits	Data access L1 pre-fetch hits	Data access L1 miss	Total data access: A0+A1+A2	—
	Data L1→L2 cacheable load access sorting	B	DL12	L2 Data access hits (shared/exclusive)	L2 Data access hits (modified)	L2 Data access miss	Total L1→L2 cacheable data accesses = B0+B1+B2	Only cacheable accesses, including CME accesses
	L1 Data pre-fetch accesses	A	DLPF	CME/granular PF accesses - L1 hit (including pre-fetch hit)	CME/granular cache pre-fetch accesses that were dropped	CME granular - L1 miss	Total CME granular accesses: A0+A1+A2	Only data CME and cache granular accesses
	Data load holds 1	A	DLH1	Data load cacheable hold cycles	Data loads non-cacheable hold cycles	Data cache contention cycles	—	Holds for parallel accesses counted once
	Data load holds 2	B	DLH2	Full fetch queue hold cycles	miscellaneous hold cycles (barriers, special commands)	Address queue load after store hazard cycles	Total data cache holds = A0+A1+A2+B0+B1	Address queue holds may overlap the data cache holds
	Data load special cases	A	DLSP	Total memory loads	Non-aligned 4K loads	load rewind events	—	—
Data stores	Store access sorting	A	DSAS	Total store accesses	SGB merges	SGB write out hot (DLINK to L1)	SGB output accesses: A0-A1 (w/o pending entries)	No cache commands & barriers, after non-alignment split, merges may be 4 at a time
	SGB events	B	DSGE	SGB freeze cycles	SGB read-modify-write accesses to L2	SGB accepter rewind events	—	For parallel accesses, a freeze is counted once

Table 8-54. Summary of event groups for profiling (continued)

Event category	Event group	Triad	Tag	Counter 0	Counter 1	Counter 2	Calculated entities	Comments
Bus load	L1-L2 bus load	A	BL12	DLINK read beats	DLINK write beats	ILINK beats	—	A single load access (A0,A2) can be 2 bus cycles (beats) if cacheable. Cache commands (no-tag access) counted as writes
L2 cache statistics	L2 demand (Elink) accesses - 1	A	KED1	Total L2 demand accesses	L2 program accesses	L2 data accesses	Barriers+CI=A.0-A.1-A.2 L2D hit = A.2-B.2 L2I hit = A.1=B.1 L2 miss = B.1+B.2	Each counter counts up to 4 events per cycle
	L2 demand (Elink) accesses - 2	B	KED2	Total L2 hit	L2 instruction miss	L2 data miss		
	L2 AOUT master requests to CoreNet	A	KAOT	Total L2 AOUT requests	L2 AOUT requests sent as global (M=1)	L2 AOUT data side requests (including barriers)	Incoher. AOUT = A.0-A.1 Instr. AOUT = A0-A2	—
	L2 total traffic	B	KTOT	Total L2 demand (ELINK) accesses	L2 Snoop requests	Tot. L2 accesses from all sources	RLT (rld etc.) = B.2-B1-B0	Counters 0 and 2 may count up to 4 events per cycle
	L2 external accesses	A	KEXT	L2 Total stashes	L2 Snoop requests	L2 Stash requests degraded to snoop	Coher. snoop = A.1-A.0 Snoop miss = A.1-B(b).0 Stash success = A.0-A2 Stash w/o INT = B(a).0-B(b).1-B(b).2 Snoop pushes = B(b)b.1+B(b).2	KEXT should be counted with either KDIN or KSNP
	L2 DIN	B(a)	KDIN	L2 reload requests from CoreNet	L2 Store allocate	—		
	L2 snooping	B(b)	KSNP	L2 Snoop hit	L2 snoops causing MINT	L2 snoops causing SINT		

Table 8-55. Triad Counted Events (CEV) field values

CEV value	Triad A event group		Triad B event group		Comments
	Tag	Event group (see Table 8-54)	Tag	Event group (see Table 8-54)	
0	CE1	Exceptions 1	CE2	Exceptions 2	Coherent groups
2	CBP1	Branch prediction 1	CBP2	Branch prediction 2	Coherent groups
3	—	—	CLP	Hardware loop prediction	—
4	—		CRW	Rewind events	—
5	CCH	Cycles - high level	—	—	—
6	CCS1	Stall cycles 1	CCS2	Stall cycles 2	Coherent groups
10	ED1	Debug events 1	—	—	—
11	EW1	Watchpoints 1	—	—	—
12	EIN1	DEBUGEV instructions 1	—	—	—
20	IAS	L1 Instruction cache access sorting	IL12	Instruction L1→L2 cacheable access sorting	—
21	IPF	L1 instruction cache pre-fetch	IAH	Instruction access holds	—
25	DGCI	Data Coherency - invalidation events	DGAH	Data accesses - general hold/freeze	—
26	DLSR	L1 data cache load access sorting	DL12	Data L1→L2 cacheable access sorting	—
27	DLPF	L1 data cache pre-fetch	—	—	—
28	DLH1	Data load holds 1	DLH2	Data load holds 2	Coherent groups
29	DLSP	Data load special cases	—	—	—
35	DSAS	Data Store Accesses Sorting	DSGE	SGB events	—
43	BL12	L1-L2 bus load	—	—	—
50	KED1	L2 demand (Elink) accesses - 1	KED2	L2 demand (Elink) accesses–2	Coherent groups
51	KAOT	L2 AOUT traffic	KTOT	L2 total traffic	—
52	KEXT	L2 external accesses	KDIN	L2 DIN	Triad A coherent with either KDIN or KNSP
53	—	—	KNSP	L2 snooping	—
All others	—	Reserved	—	Reserved	—

8.8.2.4 Profiling Counter Value Registers A*n*/B*n* (PCVRA*n*/B*n*)

Six 32-bit read-write registers, allocatable either to the host or monitor partition, each holding the counted value of a triad counter. The counters are counting up, and generate a counter event when overflowing from 0xFFFFFFFF to 0. The counters also generate overflow signals from intermediate locations in the counter for generating trace messages, see Section 8.9.5.2, “Profiling trace.”

Address 0x420 (PCVRA<*n*>)
offset: <4*n>
range: *n*=0..2
0x490 (PCVRB<*n*>)
offset: <4*n>
range: *n*=0..2

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PCV<*n*>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PCV<*n*>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-41. Profiling Counter Value A*n*/B*n* (PCVRA*n*/B*n*)

The fields of this register are described in this table.

Table 8-56. Field description of PCVRA*n*/B*n*

Field	Description
31-0 PCV<*n*>	The counter value of the respective profiling counter.

8.8.2.5 Profiling Counter Snapshot Registers A*n*/B*n* (PCSRA*n*/B*n*)

Six 32-bit read-only registers, allocatable to either the host or monitor partition, which hold a synchronized snapshot of the counted values of each respective counter in the triad. The value from the respective Counter Value register (PTCVRAn/Bn) is copied to the respective snapshot register upon a sampling event as defined in Table 8-53. The snapshot values could be read by software later on, or automatically sent to the trace stream if tracing the profiling messages is enabled.

Address 0x440 (PCSRA<*n*>)
offset: <4*n>
range: *n*=0..2
0x4B0 (PCSRB<*n*>)
offset: <4*n>
range: *n*=0..2

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PCS<*n*>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PCS<*n*>															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-42. Profiling Counter Snapshot A*n*/B*n* (PCSRA*n*/B*n*)

The fields of this register are described in this table.

Table 8-57. Field description of PCSRA*n*/B*n*

Field	Description
31-0 PCS<*n*>	The snapshot of the respective profiling counter.

8.8.3 Reloadable counter registers

The PU includes a reloadable counter, designed to help controlling other debug events that require interval or event counting. The main differences between the reloadable counter and the profiling counters are the following:

- A reloadable counter is not part of a triad. When enabled, it counts individually.
- As its name implies, a reloadable counter has a reload register. Upon a reload event or when reaching zero, it can be reloaded with the initial count value.
- The register values cannot be traced. However the counter event can generate a trace message.
- The counter counts down, unlike the profiling counters which count up

A useful use case of these counters could be, for example, generating periodic profile trace messages. To help support software-managed profiling schemes, the reloadable counter also has a snapshot register, which copies the count value upon a sampling event of Triad A.

8.8.3.1 Reloadable Counter Control Register 0 (RCCR0)

A 32-bit read-write register, allocatable to either the host or monitor partition, used to configure the reloadable counter.

Address 0x500 (RCCR<*n*>)
offset: <20**n*>
range: *n*=0

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R		CEV							RLD				STOP			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	STRT				PCDF					ID0	CMOD			RZ	ACT	EN
W														w1c	R	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-43. Reloadable Counter Control Register 0/1 (RCCR0)

The fields of this register are described in this table.

Table 8-58. Field description of RCCR0

Field	Description
0 EN	Enable 0 - Disabled 1 - Enabled
1 ACT	Active 0 - Not active 1 - Active
2 RZ	Reached Zero - the general output event generated by the counter. The bit is cleared when the counter is enabled, and set when the counter reached zero. Once set, it could be cleared by software by writing “1” to it.
5-3 CMOD	Counting mode **000**: One shot counting: When active, counts until the counter reaches 0, and disables itself (EN and ACT are cleared). When transferring from 1 to 0, generates an event and sets the RZ bit. The counter does not wrap around. **001**: Continuos counting: When active, counts continuously. After reaching 1, in the following event reloads the value from the reload register, generates an event, sets RZ, and continues to count. A reload event reloads the counter without modifying RZ nor generating an event. RZ is not cleared once set. Reloading when the counter is not active is ignored.
6 ID0	If set, consider events only for task ID comparison of ID comparator 0.
11 PCDF	PC detector filter - Consider events only if the PCDA detector signaled a hit in this cycle (for asynchronous events) for this VLES (for VLES-based events). PCDA can be configured only for PC detection. See Table 8-59. 0 - no PC detector filtering 1 - with PC detector filtering
15-12 STRT	Events for starting the count. See Table 8-59.
19-16 STOP	Events for stopping the count. See Table 8-59.
23-21 RLD	Events for reloading the counter. See Table 8-59.
30-25 CEV	Counted events: each value of the field defines the event that is counted in this counter. The list of events is described in Table 8-60.

The list of events that are used to filter, start, stop and reload the counter are described in this table.

Table 8-59. Event allocation for filtering, start/stop and reloading a reloadable counter

Reloadable counter	PC filter	Start events				Stop events				Reload events		
		STRT[3]	STRT[2]	STRT[1]	STRT[0]	STOP[3]	STOP[2]	STOP[1]	STOP[0]	RLD[2]	RLD[1]	RLD[0]
0	PCDA 0	IEU Q2	IEU Q0	DEBUGEV0	Reserved	IEU Q3	IEU Q1	DEBUGEV1	Reserved	IEU Q3	IEU Q1	Start event

The following priority applies to events affecting the activity status of the profiling counters:

1. Enable

2. Stop
3. Start
4. Disable

For example, if a start and a stop event occur in the same cycle, the stop event has precedence. These events affect the ACT status bit on the next cycle.

When the counter is in the Active state, and the counter is in continuous mode, the following priority applies between events that occur on the same cycle:

1. Reload
2. Count
3. Sample

For example, if a count event and a reload event occur together, the counter will be reloaded. Note that in the other counting modes, these events may also interact with the activity or enable status of the counter, as described in the CMOD field of RCCR0, see Table 8-58.

The list of events that can be counted by the reloadable counters is described in this table.

Table 8-60. Events counted by the reloadable counters

Value	Counted events	Comments	Value	Counted events	Comments
0	Core cycles	Frozen during debug mode	36	PCDA 0 detection	—
1	Application cycles	Frozen during debug and low power modes	37	PCDA 1 detection	—
2	Non-stall cycles	Cycles where the core is not stalled, frozen, or in debug/low power	38	PCDA 2 detection	—
3	VLES execution	—	39	PCDA 3 detection	—
4-5	Reserved	—	40	PCDA 4 detection	—
6	Reserved	—	41	PCDA 5 detection	—
7	Reserved	—	42	PCDA 6 detection	—
8	Reserved	—	43	PCDA 7 detection	—
9	Reserved	—	44	Reserved	—
10	Reserved	—	45	Reserved	—
11	Reserved	—	46	Reserved	—
13	Triad A overflow	—	47	reserved	—
14	Triad B overflow	—	48	External input 0	—
15	Reserved	—	49	External input 1	—
16-19	Reserved	—	50	IEU event Q0	—
20	DEBUGEV.0	—	51	IEU event Q1	—
21	DEBUGEV.1	—	52	IEU event Q2	—
22	DEBUGEV.2	—	53	IEU event Q3	—

Table 8-60. Events counted by the reloadable counters (continued)

Value	Counted events	Comments	Value	Counted events	Comments
23	DEBUGEV.3	—	54-63	Reserved	—
24	DEBUGEV.4	—	—	—	—
25	DEBUGEV.5	—	—	—	—
26	DEBUGEV.6	—	—	—	—
27	DEBUGEV.7	—	—	—	—
28	Reserved	—	—	—	—
29	Reserved	—	—	—	—
30	Reserved	—	—	—	—
31	Reserved	—	—	—	—
32	Reserved	—	—	—	—
33	Reserved	—	—	—	—
34	Reserved	—	—	—	—
35	Reserved	—	—	—	—

8.8.3.2 Reloadable Counter Value Registers 0 (RCVR0)

A 32-bit read-write register, allocatable to either the host or monitor partition, which holds the counted value of the reloadable counter. The counter counts down, and depending on the counting mode, generates a counter event when reaching zero. The counter value is reloaded from the Reload register upon the following events:

- Enabling the counter
- Upon a counted event when the counter has the value of “1” (in the continuous counting mode only)
- A reload event as configured in Table 8-59.

Address 0x508 (RCVR<*n*>)
offset: <20**n*>
range: *n*=0

Ownership: Config:
Access: S: RW

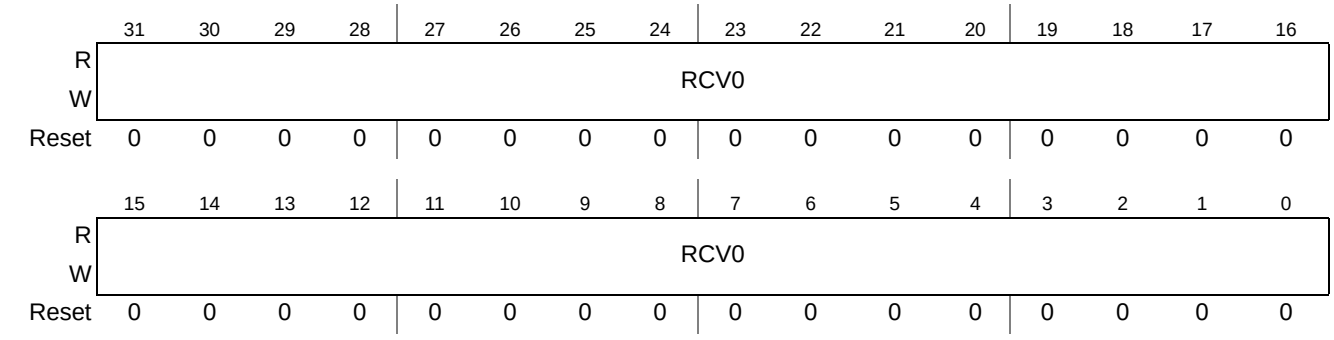

Figure 8-44. Reloadable Counter Value Register 0 (RCVR0)

The fields of this register are described in this table.

Table 8-61. Field description of RCVR0

Field	Description
31-0 RCV0	The counter value of the reloadable counter.

8.8.3.3 Reloadable Counter Reload Registers 0 (RCRR0)

A 32-bit read-write register, allocatable to either the host or the monitor partition, which holds the value to be reloaded to the reloadable counter upon the occurrence of a reload event. For correct operation, the reload value must be configured to a non-zero value.

Address 0x50C (RCRR<*n*>)
offset: <20**n*>
range: *n*=0

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	RCR0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	RCR0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-45. Reloadable Counter Reload Register 0 (RCRR0)

The fields of this register are described in this table.

Table 8-62. Field description of RCRR0

Field	Description
31-0 RCR0	The reload value of the reloadable counter

8.8.3.4 Reloadable Counter Snapshot Register 0 (RCSR0)

A 32-bit read-only register, allocatable to either the host or the monitor partition, which samples the value of the reloadable counter upon a sampling event of the triad A. For sampling to occur, the reloadable counter and triad A must belong to the same partition, and triad A must be enabled.

Address 0x510 (RCSR<*n*>)
offset: <20**n*>
range: *n*=0

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	RCS0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	RCS0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-46. Reloadable Counter Snapshot Register 0 (RCSR0)

The fields of this register are described in this table.

Table 8-63. Field description of RCSR0

Field	Description
31-0 RCS0	The snapshot of the reloadable counter.

8.9 Understanding tracing support

The tracing support in the SC3900 subsystem is based on the Nexus (IEEE ISTO 5001) standard.Tracing support is performed by the Trace Unit, also named the NSC (Nexus StarCore Client) in the DTU. The trace unit as a whole could be allocated to either the host or monitor partitions - no sub-portions of it could be independently allocated.

8.9.1 Supported trace configurations

Trace message types belong to functional groups that are generally independent. Some limitations apply to the supported trace group combinations that could be activated together (see Table 8-64), however, the programming model assumes that each tracing group could be individually enabled regardless of the other active tracing groups. The supported tracing groups are:

- Task/Process ID: reporting task/process ID changes
- Program trace: reporting change of flow events, allowing to reconstruct the program execution flow. There are two variants of program trace:
  - Light program trace: reporting only the following COF events: Jump to an interrupt, return from an interrupt (RTE* instructions), jump to a subroutine (SBR and JSR instructions), return from subroutine (RTS* instructions)
  - Full program trace: every change of flow is reported, including hardware loop iterations. During program trace it is possible to enable additional trace messages which reference the program trace stream, such as correlation messages, and others.

- Watchpoint messages: Light-weight messages that are reported upon occurrences of various debug events (outputs of Debug Event Detectors)
- Data Acquisition messages: messages that are generated when the user writes data to dedicated core registers, enabling to send user-defined data into the trace stream
- Profiling messages: Sending the profiling counters (one or two triads) into the trace stream
- In addition there are several standard Nexus messages that are mandatory: debug status, error messages, and so on.

This table lists the supported trace message combinations. The output for other trace combinations is not guaranteed.

Table 8-64. Supported trace combinations

Profile	Description	Full prog trace	Light prog trace		Profile trace		Ownrshp. trace	Watchpt. trace	Data aqust. trace
Debug 1	Full debug flow—most visibility	x	—	—	—		x	x	x
Debug 2	Light debug with reduced bandwidth	—	x	—	—		x	x	x
Debug 4	printf debug (instrumented trace)	—	—	—	—	—	—	x	x
Profile 1	Full code profiling	x	—	—	x	—	x	x	x
Profile2	Profiling upon events	—	—	—	x	—	x	x	x

8.9.2 Messages and beats

The Nexus standard defines the way that trace messages are reported on the trace bus. Messages are reported in data units called “beats.” The size of the beat, and the number of beats that can be reported per cycle are not set in the standard and are implementation specific. In SC3900, the beat size is 30 bits, and two beats can be reported per SoC cycle. The following rules apply to how messages are split into beats:

- A message is reported in whole beats. If the message length is less that an integer multiple of the beat size, the message is zero padded to the nearest beat. This means that different messages always occupy different beats.
- A variable field has to be the last in its beat.

In SC3900, the size of the beat affects the definition of the message, in order to minimize cases of zero padding.

8.9.3 NCS message buffers

The NCS has inside it two types of message buffers:

- A main message queue, that holds all messages waiting to be reported to the SoC. This queue buffers local peaks in message generation and averages the transmission rate to a value based on the core to SoC frequency ratio, and the availability of the NPC block at the SoC level to accept the messages
- Several input buffers, per message generator (for example, program trace, profiling trace, etc.). These buffers are designed to absorb some contention situations where too many messages are generated together and contend on entry to the main message queue, which has a limited input capacity.

The main message queue can accept at most four messages at the same cycle. In case more than 4 messages are generated concurrently, the priority between them is as described in Table 8-65.

Table 8-65. Message priority at the entrance to the main message queue

Message	Priority	Comments
Error	0 - highest	—
Data acquisition	1	—
Ownership	2	—
Watchpoint	3	—
Program trace: indirect branch, sync, RFM, light program trace	4	—
Program trace: correlation	5	—
Debug status	6	—
Timestamp correlation	7	—
Profiling	8	—

8.9.4 Trace unit registers

The Trace unit includes the following registers:

- Trace Control Register 1
- Trace Status Register
- Trace Control Register 3
- Trace Control Register 4
- Trace Profile Messages Control Register
- Trace Watchpoint Mask Register
- Overrun Control Register
- TMDAT Image Register

8.9.4.1 Trace Control Register 1 (TC1)

A 32-bit read-write register, for configuring some of the general trace options.

Address 0x600 (TC1) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	RTTS		PTT				TSCE									
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		POTD	TSEN						TM							
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-47. Trace Control Register 1 (TC1)

The functions of the bits in this register are described in Table 8-66. Some values in standard fields are vendor specific; they are marked with (V).

Table 8-66. Field description of TC1

Field	Description
7-0 TM	Trace Mode 00000000: Trace disabled XXXXXXX1: Ownership trace enabled XXXXXX1X: Reserved XXXXX1XX: Program trace enabled XXXX1XXX: Watchpoint trace enabled XXX1XXXX: Reserved XX1XXXXX: Data acquisition trace enabled X1XXXXXX: (V) Profiling trace enabled 1XXXXXXX: Reserved
13-12 TSEN	Time Stamp Enable 00: Timestamp disabled 01: Timestamp is sent only with timestamp correlation messages 10: Timestamp enabled for all messages 11: Reserved
14 POTD	Periodic Ownership Trace Disabled 0: Reserved 1: Disabled - must be configured this way
25-24 TSCE	Time Stamp Correlation Event (V) - relevant only if TSEN > 0 01: Send a timestamp correlation message upon IEU event Q0 Other combinations are reserved If a profiling message is enabled, for all cases of TSEN >0, the profiling message following the timestamp correlation message will include a timestamp

Table 8-66. Field description of TC1 (continued)

Field	Description
29-28 PTT	Program Trace type (if enabled in the TM field) (V) 00: Full program trace, RTS treated as an indirect COF 01: Full program trace, RTS treated as a direct COF when possible 10: Light program trace 11: Reserved
31 RTTS	Retransmit the trace buffer (V) Setting this bit re-transmits the contents of the trace buffer. When set, the TU does not accept any trace inputs. When the re-transmission finishes, the TU is turned inactive, as reflected in TRSR.ACTTM field. Following this procedure, the TU has to be disabled (TM filed set to 0) before enabling again. This procedure may be of use for post-mortem analysis, when the data previously transmitted to the SoC was either filtered or sent to a circular buffer and trampled over.

8.9.4.2 Trace Status Register (TRSR)

A 32-bit read only register that records status information about the trace unit.

Address 0x604 (TRSR) Ownership: Config:
Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R			NCSID													
W			R													
Reset	0	0	U	U	U	U	U	U	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R				USMS				ACTA	ACTTM							
W				R				R	R							
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-48. Trace Status Register (TRSR)

The functions of the bits in this register are described in this table.

Table 8-67. Field description of TRSR

Field	Description
7-0 ACTTM	Active Trace Modes: bit per the activation status of each trace mode, enabled through TC1.TM An “active” state (1) for a trace mode means that this mode is enabled in DC1.TM, and in addition, either: - A start condition for this trace mode is not defined in TC3 - A start condition is defined for this trace mode, a start condition had occurred, and a stop condition did not occur yet. A “not active” state (0) for a trace mode means that either - This mode is not enabled in TC1.TM - Enabled, but a start condition (defined for it in TC3) did not occur yet - Enabled, and a stop condition occurred but a start condition did not follow it ACTTM[0]: Ownership trace messages are active ACTTM[1]: Reserved ACTTM[2]: Program trace messages are active ACTTM[3]: Watchpoint trace messages are active ACTTM[4]: Reserved ACTTM[5]: Data acquisition trace messages are active ACTTM[6]: Profiling trace messages are active ACTTM[7]: Reserved
8 ACTA	Activity Status of Trace Configuration A (defined in TC3) 0: Trace configuration A is not active (not enabled, or stop event received but start not yet) 1: Trace configuration A is active (start event received and stop event not yet) Reflects the OR-ed status of the respective ACTTM bits, if individually enabled in TC3.ACTA
12 USMS	Unsent Messages 0: There are no unsent messages in the TU 1: There are unsent messages in the TU
29-24 NCSID	The number used for this core in the SRC field of Nexus messages, hard-wired

8.9.4.3 Trace Control Register 3 (TC3)

A 32-bit read-write register, controlling the general tracing filters, and dynamic start/stop events.

Address 0x60C (TC3) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R											TCA		STOPA			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	STRTA									PCDF						
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-49. Trace Control Register 3 (TC3)

The functions of the bits in this register are described in this table.

Table 8-68. Field description of TC3

Field	Description
6-5 PCDF	PC detector filter - trace only if any of the assigned PCDA detector (see Table 8-69) signaled a match for the VLES relevant for this message. This PCDA can be configured only for PC range detection. This filter affects only the following messages: - Data acquisition message (TCODE 7) - Profiling trace (TCODE 58, 59, 60) Field settings: x1: Enable PC filtering by PCDA0 1x: Enable PC filtering by PCDA4 Note that full program trace messages are not affected by PCDF. To filter those, dynamic start/stop conditions should be used.
15-12 STRTA	Upon an enabled event (see Table 8-69), start tracing the trace messages specified in TCA
19-16 STOPA	Upon an enabled event (see Table 8-69), stop tracing the trace messages specified in TCA
21-20 TCA	Tracing Configuration A: 00: none x1: Program trace 1x:Reserved

The filters, start and stop events are listed in this table.

Table 8-69. Allocation of events for filtering, starting and stopping trace messages

Trace messages	PC filter	Start events				Stop events			
		STRT[3]	STRT[2]	STRT[1]	STRT[0]	STOP[3]	STOP[2]	STOP[1]	STOP[0]
Affecting only the messages listed in TC3.PCDF	PCDA 0, 4	—	—	—	—	—	—	—	—
Selected in TCA	—	IEU Q2	IEU Q0	DEBUGEV0	PCDA2	IEU Q3	IEU Q1	DEBUGEV1	PCDA3

8.9.4.4 Trace Control Register 4 (TC4)

A 32-bit read-write register, which defines what events trigger Program Correlation Messages (PCM) when full Program tracing is enabled.

Address 0x610 (TC4) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	EVCDM															
W																
Reset	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0

Figure 8-50. Trace Control Register 4 (TC4)

The functions of the bits in this register are described in this table.

Table 8-70. Field description of TC4

Field	Description
15-0 EVCDM	Event Code (EVCODE) Mask. Reset value of “1” (disabled by default) for EVCODEs #9 and above, see Table 8-84. 0000000000000000: no EVCODEs masked for PCM XXXXXXXXXXXXXXX1: EVCODE #1 masked for PCM XXXXXXXXXXXXXX1X: EVCODE #2 masked for PCM ... 1XXXXXXXXXXXXXXX: EVCODE #16 masked for PCM

The list of events for which there is an EVCODE is described in Table 8-84.

8.9.4.5 Trace Profile Message Control Register (TPMCR)

A 32-bit read-write register. The fields in this register configure the behavior of profiling trace messages. Enabling and trigger configuration of this message is performed in Section 8.9.4.1, “Trace Control Register 1 (TC1),” and Section 8.9.4.3, “Trace Control Register 3 (TC3).”

Address 0x614 (TPMCR) Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R																
W																

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R															PTCS	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-51. Trace Profile Message Control Register (TPMCR)

The functions of the bits in this register are described in this table.

Table 8-71. Field description of TPMCR

Field	Description
1-0 PTCS	Profile Counter Trace Selection, if enabled in TC1.TM. 00: reserved 01: Trace triad A 10: Trace Triad B 11: Trace both Triads A and B

8.9.4.6 Trace Watchpoint Mask Register (TWMSK)

A 32-bit read-write register, which controls which events will generate a watchpoint message. Bits 18 and 28 to 31 must be masked in this revision when watchpoint tracing is enabled.

Address 0x618 (TWMSK) Ownership: Config:
Access: S: RW

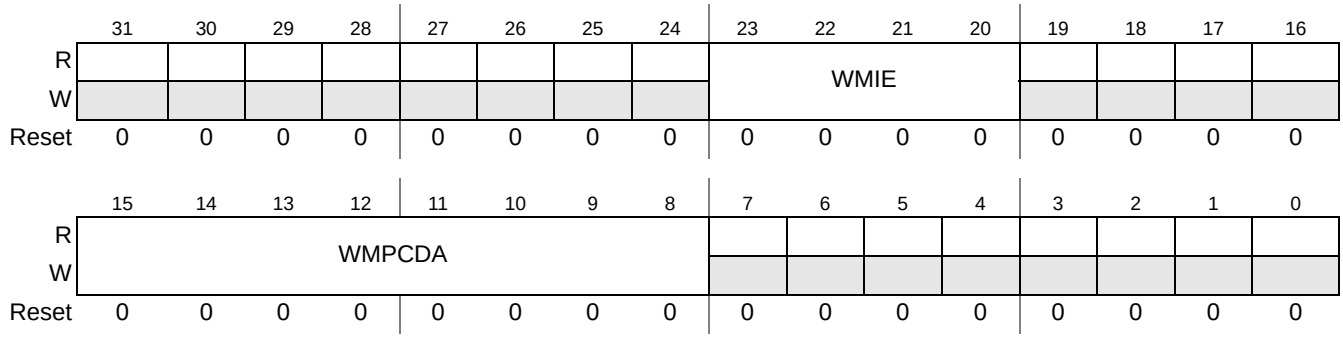

Figure 8-52. Trace Watchpoint Mask Register (TWMSK)

The functions of the bits in this register are described in this table.

Table 8-72. Field description of TWMSK

Field	Description
15-8 WMPCDA	Watchpoint Message enable of PC and Data Address detector events WMPCDA[*n*]: set for enabling watchpoint messages from PCDA*n* event
23-20 WMIE	Watchpoint Message enable of IEU events WMIE[*n*]: set for enabling watchpoint messages from Q*n* event

8.9.4.7 Trace Overrun Control Register (TOCR)

A 32-bit read-write register, used to control the actions and priority when message density approaches overrun.

Address 0x61C (TOCR)

Ownership: Config:
Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R														FTSC		CFEN
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-53. Trace Overrun Control Register (TOCR)

The functions of the bits in this register are described in this table.

Table 8-73. Field description of TOCR

Field	Description
0 CFEN	Enable Core Freezing when trace buffers are in danger of loosing messages. For messages that reflect code execution activity (program trace, profiling trace messages, address messages) it is ensured no messages are lost, at the expense of affecting performance and perhaps loosing the real-time behavior. It is not guaranteed that messages that are generated from asynchronous reasons will not be lost. In case a message type can be triggered by both program execution activity and non-core reasons, the non-core triggers must be disabled to ensure no message loss. Examples for messages which also have non-core triggers: - Program correlation for IEU events or external triggers - Profiling messages triggered by events which have external causes - Watchpoint messages which may have external causes Note that when using this mode for profiling messages, the profiled events should be selected to be those that are only core-related otherwise the count may be distorted by events occurring during the freeze. 0: Core is not frozen when there is danger of loosing messages 1: Core is frozen when there is danger of loosing messages When this bit is set, the configuration of message suppression in SPEN is ignored.
2 FTSC	Enable freezing the Timestamp counter when the core is frozen by the trace unit when CFEN = 1 Freezing the timestamp will allow closer to reality timestamps for core-generated messages. In any case the timestamps will not be fully accurate because the caches continue to operate while the core is frozen 0: The timestamp is not frozen when the core freezes due to a CFEN condition 1: The timestamp is frozen when the core freezes due to a CFEN condition

8.9.4.8 TMDAT Image Register (TMDATI)

The TMDAT Image (TMDATI) register is a read-only 32-bit register, which reflects the value of the TMDAT core register. The value of this register is updated whenever the core performs a write to the TMDAT register (TFRA or MOVE.L to TMDAT). When data acquisition tracing is enabled, a core write to TMDAT in the core generates a data acquisition message. See the *SC3900 FVP Core Reference Manual* for a description of the MDAT register. Reflection in TMDATI occurs when the core is in debug mode, also when tracing is disabled, and also if the trace unit is allocated to the monitor partition. The register is

intended to support communication in debug mode between the core and the external debugger. In particular, the host debugger can transfer the value of an address register (making it observable in the memory map) so that it could be saved, after which this register could be initialized by another core command to hold the memory address for writing information from the core with additional core commands.

Address 0x630 (TMDATI) Ownership: Config:
Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	TMDATV															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	TMDATV															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8-54. TMDAT Image Register (TMDATI)

The fields of this register are described in this table.

Table 8-74. Field description of TMDATI

Field	Description
31-0 TMDATV	The value written to the TMDAT core register.

8.9.5 General message categories

This section describes in high level some general categories that involve several types of messages.

8.9.5.1 Full program trace

This group of messages includes messages that carry information on change of flow operations, and on other information that is correlated with the program execution flow. Program trace can be activated right when tracing is enabled, or it could be dynamically activated depending on another event. The whole group of messages is activated or de-activated together. Program trace may be activated in two detail levels: full program trace, and light program trace. Light program trace only reports subroutine and interrupt calls and returns, always using the full PC. No additional correlation messages are generated in this mode. Full program trace reports any COF, including hardware loop iterations. Whenever the COF uses an explicit address (direct COF), only a taken/not taken indication is reported for it, allowing very high compression. Full program tracing employs a 35-bit COF history buffer, which holds a bit (taken/not taken) per direct COF. Not-taken indirect COFs are also added a bit in the buffer. The buffer is reported and cleared upon a taken indirect COF event, when the buffer is full or when processing or program tracing is suspended.

In addition, full program trace could be supplemented with correlation messages which allow to report additional information categories with information correlating them to the PC stream.

Full program trace involves the following messages:

- Program synchronization (TCODE 9): a message that occurs when program trace is started, resumed or after recovering from an error condition (see Table 8-81).
- Indirect branch history with sync (TCODE 28): The message holds the destination PC of the indirect COF, and the history buffer accumulated up until that point. In case of an interrupt, the message also includes the ICNT field, marking the PC where the code was interrupted in the execution stream.
- Resource full—ICNT (TCODE 27, RCODE 0): The ICNT field counts serial VLES, and resets upon events such as a taken COF or synchronization event (see Table 8-75). When there is a sequence of more than 256 sequential VLES, a message marking a counter wrap around is generated.
- Resource full—history buffer (TCODE 27, RCODE 1): In case the history buffer is filled without a message that dumps it, a resource full message dumping the full buffer is generated.
- Program correlation—history buffer (TCODE 33, EVCODE 1,5,6): upon an event when the core enters an idle state (debug or low power), and when program tracing is disabled or temporarily suspended (a STOP event or PC filter condition).

8.9.5.2 Profiling trace

The state of the profiling triads can be sampled and reported upon several configurable events (see Table 8-52). The most useful sampling event is upon a jump and return form a subroutine and interrupt. The profiling trace message can trace one or two triads. The message includes the full PC so these messages are independent from program tracing. In order to save bandwidth, only 15 bits of each counter are traced upon a trace event. A Reasource Full message is generated upon any overflow of any traced counter into the MSB of the reported portion. This message reports 4 bits of the counter - bit 15 of the window and the 3 following bits. This report allows the profiling tool to keep track of the counter values and reconstruct the most significant portion of the counters also in may cases where overflow or trace messages are lost. In the example of Figure 8-55, the 15-bit traced window of the counter is of bits 14:0. The resource full message is generated when the counter overflows into bit 14 (one bit earlier than absolutely necessary). Upon this event, bits 17:14 of the counter (termed the spill bits) are reported.

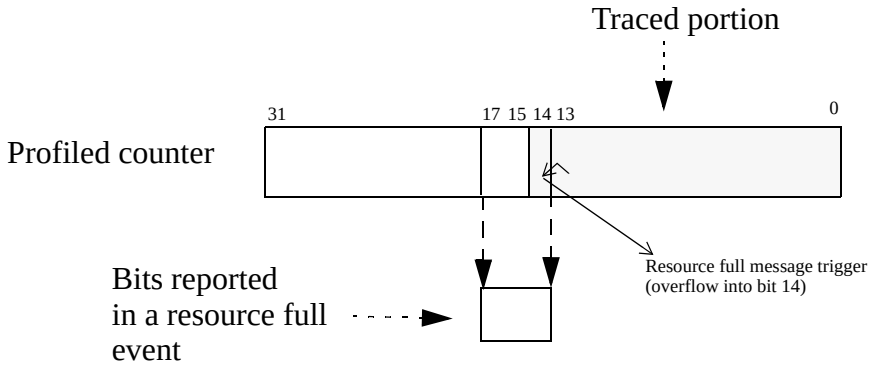

Figure 8-55. Bits traced in profiling messages and upon overflow

The following messages are involved in profiling trace:

- Profiling trace of one triad (TCODE 58,59): The two messages are identical, differing only in the sampling reasons.
- Profiling trace of two triads (TCODE 60)
- Resource full - profiling counter overflow (TCODE 27, RCODE 10): Reports which counter or counters overflowed (6 bits), and 4 bits per triad reporting spill bits of the overflowing counter. This message enables to report extra bits from two counters, one per triad. If more than one counter overflowed in the same triad in the same cycle, only one of them (lower index) will report the extra bits. However the fact that it overflowed is captured in the 6 status bits.

8.9.6 Supported trace messages

The supported trace messages are described in Table 8-75.

Table 8-75. Supported trace messages

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Debug Status	Standard, fields modified	—	6	TCODE	fixed	—	TCODE number = 0
		—	6	SRC	fixed	—	Source processor identifier
		—	2	STATE	fixed	—	Processing state the core entered to
		STATE = 00,01,10	16	STAUTS	fixed	1	Exception ID information, see Table 8-77
		0	28	TSTAMP	variable	+1	Timestamp (optional)
	Standard, fields modified	—	6	TCODE	fixed	—	TCODE number = 0
		—	6	SRC	fixed	—	Source processor identifier
		—	2	STATE	fixed	—	Processing state the core entered to
		STATE = 11	32	STATUS	fixed	2	Debug reason information, see Table 8-77
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Error	Standard	—	6	TCODE	fixed	—	TCODE number = 8
		—	6	SRC	fixed	—	Source processor identifier
		—	4	ETYPE	fixed	—	Error type, see Table 8-78
		—	10	ECODE	fixed	1	Error code, see Table 8-79
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Ownership	Standard	—	6	TCODE	fixed	—	TCODE number = 2
		—	6	SRC	fixed	—	Source processor identifier
		—	48	PROCESS	fixed	2	Task/process ID, see Table 8-80
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Data Acquisition	Standard		6	TCODE	fixed	—	TCODE number = 7
			6	SRC	fixed	—	Source processor identifier
			8	IDTAG	fixed	—	Taken from TMDTAG [7:0]
		10	32	DQDATA	variable	1-2	Taken from TMDAT [31:0]
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Program trace—synchronization	Standard, field added	—	6	TCODE	fixed	—	TCODE number = 9
		—	6	SRC	fixed	—	Source processor identifier
		—	6	PRED	fixed	—	The state of the 6 predicates right before the VLES pointed by the PC
		1	31	PC	variable	1-2	Full current PC. Leading zeros can be omitted
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Program trace - indirect branch history with sync	Standard	—	6	TCODE	fixed	—	TCODE number = 28
		—	6	SRC	fixed	—	Source processor identifier
		—	2	B-TYPE	fixed	—	Branch type, see Table 8-82.
		BTYPE = INT	8	I-CNT	fixed	—	# of VLES completed since the last time the instruction counter was cleared, see Section 8.9.10.7, “I-CNT field”
		—	31	DPC	fixed	—	Full indirect COF target PC
		7	36	HIST	variable	2-**3**	Direct branch/not taken COF history information
		0	28	TSTAMP	variable	+1	Timestamp (optional)
	Standard - ICNT removed	—	6	TCODE	fixed	—	TCODE number = 28
		—	6	SRC	fixed	—	Source processor identifier
		BTYPE not INT	2	B-TYPE	fixed	—	Branch type, see Table 8-82.
			31	DPC	fixed	—	Full indirect COF target PC
		15	36	HIST	variable	**2**-3	Direct branch/not taken COF history information
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Program trace - resource full	Standard	—	6	TCODE	fixed	—	TCODE number = 27
		—	6	SRC	fixed	—	Source processor identifier
		COF history	4	RCODE	fixed	—	Resource code identifying the full resource, see Table 8-83
		—	35	RDATA	fixed	2	Resource data, see Table 8-83
		0	28	TSTAMP	variable	+1	Timestamp (optional)
	Standard	—	6	TCODE	fixed	—	TCODE number = 27
		—	6	SRC	fixed	—	Source processor identifier
		all others	4	RCODE	fixed	—	Resource code identifying the full resource, see Table 8-83
		—	14	RDATA	fixed	1	Resource data, see Table 8-83
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Program trace - correlation	Standard	—	6	TCODE	fixed	—	TCODE number = 33
		—	6	SRC	fixed	—	Source processor identifier
		cases with COF dump	4	EVCODE	fixed	—	Event code identifying the event to correlate with program flow, see Table 8-84
		—	8	I-CNT	fixed	—	# of VLES completed since the last time the instruction counter was cleared, see Section 8.9.10.7, “I-CNT field”
		6	36	CDATA	variable	1-**2**	Correlation data
		0	28	TSTAMP	variable	+1	Timestamp (optional)
	Standard	—	6	TCODE	fixed	—	TCODE number = 33
		—	6	SRC	fixed	—	Source processor identifier
		all other cases	4	EVCODE	fixed	—	Event code identifying the event to correlate with program flow, see Table 8-84
		—	8	I-CNT	fixed	—	# of VLES completed since the last time the instruction counter was cleared, see Section 8.9.10.7, “I-CNT field”
		—	6	CDATA	variable	1	Correlation data (none, just zero padding the beat)
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Light Program Trace -(indirect branch with sync)	Standard, ICNT removed	—	6	TCODE	fixed	—	TCODE number = 4
		—	6	SRC	fixed	—	Source processor identifier
		—	3	SMP	fixed	—	Sampling reason, see Table 8-85
		15	31	DPC	variable	1-2	Full indirect COF target PC (zero truncation)
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Profiling trace - one triad	SC3900 specific	—	6	TCODE	fixed	—	TCODE number = 58, 59
		—	6	SRC	fixed	—	Source processor identifier
		—	2	SMP1 SMP2	fixed	—	For TCODE 58, using SMP1, For TCODE 59, using SMP2, see Table 8-85
		—	45	PDATA	fixed	—	Profiling data - 15 bits per counter
		—	31	PC	fixed	3	Full PC of the VLES with the profiling event
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Profiling trace - two triads	SC3900 specific	—	6	TCODE	fixed	—	TCODE number = 60
		—	6	SRC	fixed	—	Source processor identifier
		—	3	SMP	fixed	—	Sampling reason
		—	90	PDATA	fixed	—	Profiling data - 15 bits per counter
		15	31	PC	variable	4-5	Full PC of the VLES with the profiling event (leading zero truncation)
		0	28	TSTAMP	variable	+1	Timestamp (optional)

Table 8-75. Supported trace messages (continued)

Message name	Message type	Size of field (bits)		Field name	Field type	Total Msg length (beats)	Field description
		case/ min.	size/ max				
Watchpoint	Standard - field added	—	6	TCODE	fixed	—	TCODE number = 15
		—	6	SRC	fixed	—	Source processor identifier
		—	2	GRP	fixed	—	Event group, see Table 8-86
		—	16	WPHIT	variable	1	Watchpoint source indicator (leading zeros can be omitted), see Table 8-87
		0	28	TSTAMP	variable	+1	Timestamp (optional)
Timestamp correlation	Specific, shared with PA	—	6	TCODE	fixed	—	TCODE number = 63
		—	6	SRC	fixed	—	Source processor identifier
		—	4	TCOR	fixed	—	Correlation value sampled from the interface
		—	6	T-TYPE	fixed	—	Type of the correlation request, see Table 8-88
		—	28	TSTAMP	variable	2	Timestamp: mandatory for this message

8.9.7 Timestamp counter

All trace messages can optionally be activated with a timestamp, by setting the TSEN field in TC1, see Section 8.9.4.1, “Trace Control Register 1 (TC1).” The timestamp is generated by a 24-bit counter that operates near the core at core frequency, and should provide accurate cycle measurements for architectural events. The timestamp is attached to the trace message when it enters the internal queue, which in some cases may be after a delay if several messages were attempting to enter the queue together. For this reason the TSTAMP field also has a 4-bit correction value which specifies the value that should be subtracted from the main 24-bit count in order to compensate for that delay. The structure of the TSTAMP field is described in Table 8-76.

Table 8-76. Timestamp field (TSTAMP) format

TSTAMP field bits	Name	Description
[23:0]	TS	The timestamp at the entry to the message queue
[27:24]	CORV	Correction value, to be subtracted from TS to get the accurate time stamp

When reported in messages other than timestamp correlation (TCODE = 63), the timestamp is always appended as an additional beat to the message. Hence no compression takes place even if the zero-compressed timestamp value could have fit in the unused portion of the previous beat.

The Timestamp counter is reset to zero when tracing is disabled. When the core enters debug mode, the time stamp freezes, from the time when the input buffers feeding the main queue are empty (hence were given the correct time stamp). The time stamp counter is frozen also when the trace unit freezes the core in core-freeze tracing mode (TOCR.CFEN is set; see Section 8.9.4.7, “Trace Overrun Control Register (TOCR)”). Note that a few message types may be generated when the core is in debug mode (for example, the debug status message). In this case, the time stamp of these messages may not reflect the correct time. The time stamp is not frozen when the core is in a low power mode, if it has clocks. Clock accuracy is of course also lost in low power modes where the clocks are disabled.

8.9.8 Mandatory trace messages

This section describes in more detail the mandatory trace messages.

8.9.8.1 Debug status message

The debug message carries information on entry and exit from processing states (Execution, NAP, DOZE and DEBUG mode). The STATE field indicates to what processing state the core transferred to. The length of the status information that is reported in the message depends on the STATE field. For the Nexus interface the message length is static because per a STATE value the length of STATUS is fixed. The debug status message is generated also while the core is in low power mode or debug mode, as long as the clocks are active.

In case of entry into debug mode, the STATUS field reports the value of the Debug Mode Reason Status Register (DMRSR). When returning to execution, 16 bits of the Exception ID register (EIDR) are reported. The format of the STATUS field in the Debug Status message is described in this table.

Table 8-77. Structure of the status field in the debug status message

Bits in STATE field	Value	Meaning	STATUS
[1:0]	00	Transfer into Execution	EIDR[15:0], see Section 2.1.6.5, “The Exception ID Register (EIDR)”
—	01	Transfer into low power mode	0x0000 - Reserved 0x4000 - Reserved 0x8000 - DOZE 0xC000 - NAP
—	10	Reserved	—
—	11	Transfer into DEBUG mode	DMRSR [31:0] see Figure 8-9

8.9.8.2 Error message

The error message carries information on the occurrence of a trace error or data loss, and information on the type of error. The message is reported after the error condition ended:

- In case of queue overrun, after the queue was flushed and tracing resumed
- In case of priority contention, after the input buffer servicing the discarded message was emptied

The format of the error type (ETYPE) field in the error message is described in Table 8-78. In case more than one condition occurred simultaneously, one reason is reported according to the order in the list.

Table 8-78. Error Type (ETYPE) format

Error Type value	Description
0000	Queue overrun, one message or more were lost
0001	Contention with higher priority message caused message(s) to be lost
0010 - 1111	Reserved

The format of the error code (ECODE) field in the error message is described in Table 8-79 below. The error message reports all types of messages that were lost, a bit per message type.

Table 8-79. Error Code (ECODE) format

Error code value	Description
XXXXXXXXX1	Watchpoint message(s) lost
XXXXXXXX1X	Program trace COF, resource full or program sync message(s) lost
XXXXXXX1XX	Program trace PCM
XXXXXX1XXX	Ownership message(s) lost
XXXXX1XXXX	Debug status message(s) lost
XXXX1XXXXX	Reserved
XXX1XXXXXX	Profile message(s) lost
XX1XXXXXXX	Data acquisition message(s) lost
X1XXXXXXXX	reserved
1XXXXXXXXX	reserved

8.9.9 Ownership trace message

The ownership message carries information on the current task ID (8 bit PID + 8-bit DID, in the TID core register), and the PROCID core register. The message is generated upon any of the following events:

- Return from exception (SR2.EXP changed from 1 to 0), when the hardware carried a state indication that there was a write to the TID or the PROCID register before the RTE*. This indication state is cleared upon every RTE*.

- Occurrence of a program synchronization event, see Section 8.9.10.2, “Program trace synchronization”

The value of the PROCESS field in the Ownership trace message is described in Table 8-80.

Table 8-80. Structure of the process field in the ownership message

PROCESS field bits	Field name	Values
[15:0]	TID	Taken from TID [15:0]
[47:16]	PROCID	Taken from PROCID [31:0]

8.9.10 Program trace messages

8.9.10.1 Program trace—program address reporting

The SC3900 trace definition reports a PC only as a full value. It does not employ compression with a previously reported PC. At most, leading zeros can be discarded from the PC value if it allows to fit the truncated value into a lower number of beats. For some messages, the PC value relates to the current PC at the time of reporting the message; in this case, the field name in Table 8-75 is “PC.” In other cases, the PC relates to the destination PC of a COF instruction. In such a case, the field name is DPC.

Full PC reporting is used in the following messages:

- Indirect Branch History Message with Sync (destination PC)—TCODE 28
- Indirect Branch with Sync (destination PC)—TCODE 4
- Program Synchronization (current PC)—TCODE 9
- Profiling message (current PC)—TCODE 58, 59, 60

In some cases, the PC field is defined as variable length, and the leading zeros of the PC value can be truncated if the field can fit in a smaller number of beats.

8.9.10.2 Program trace synchronization

Full program trace is designed to allow to reconstruct the full execution flow (VLES by VLES) by reporting every change of flow event. During full program tracing, in cases where PC tracking may have been lost, a special program synchronization message (TCODE 9) is issued, allowing the trace analyzing tool to re-synchronize on the PC trace. These events are either the loss of a program trace message (TCODE 9, 27, 28, 33), or re-starting program trace reporting after it was previously suspended. The event causing program synchronization are also termed “hard” synchronization events. The list of “hard” synchronization conditions is listed in this table.

Table 8-81. Hard program synchronization conditions

Condition	Description/comments
Exit from reset	If tracing was enabled while the core was in reset
Transition from a low power mode into execution	From DOZE

Table 8-81. Hard program synchronization conditions (continued)

Condition	Description/comments
Exit from Debug mode into execution	Single steps and core commands are not considered exit from debug mode
Activation of program trace	"Activation" is when full program trace turns active A synchronization message is reported whenever full program trace is activated/resumed. Note that the filters may change state only at interrupt boundaries. START events are more flexible, see Section 8.9.4.3, "Trace Control Register 3 (TC3)."
FIFO overrun	An overrun condition had previously occurred in which one or more trace occurrences were discarded by the debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) prior to a sync message. The error message contains an ECODE value indicating the type(s) of messages lost due to the overrun condition. The message is generated after the buffer is emptied, with the error message. A synchronization message of this type is generated only in case a program trace message was lost
Message contention	A message was lost due to contention in an input buffer. The target outputs an Error Message (TCODE = 8) prior to a sync message. The error message contains an ECODE value indicating the type of message lost due to the contention. The sync message is generated after the input buffer servicing the message that was discarded has turned to have at least one vacancy - with the reporting of the error message. A synchronization message of this type is generated only in case a program trace message was lost

The SC3900 does not have situations of "Soft" synchronization defined in the Nexus standard, because it does not employ PC compression.

8.9.10.3 Program trace—indirect branch history message

This message is issued upon an indirect COF (a COF whose target address is not known statically - R register, memory or an interrupt). An indirect COF requires reporting an explicit address, because the trace analyzing tool cannot deduce it from the source code. In addition, this message includes a "history buffer" of previous direct COF instructions - one bit per COF (1 for a taken COF, 0 for a not-taken COF). The identity, source and destination addresses of these direct COFs is deduced by the trace analyzing tool from the source code. The COF instructions that are reported as direct COFs in the history buffer:

- JT, BT, JSR, BSR, JMPRF instructions with an absolute destination address
- SKIP.*n*[.U] SKIP.SQ*n*[.U], LPSKIP.*n*[.U], LPSKIP.SQ*n*[.U], BREAK.*n*
- Hardware loop LA (LPEND.*n*, LPEND.SQ.*n*)
- A subset of RTS (RAS valid) cases, if treating them as direct is enabled in TC1.PTT (01 setting), see Section 8.9.10.4, "Treating RTS return address as a direct COF."
- Any not taken COF (in general false predication, including with an indirect address)

The following COF instructions and operations are reported as an indirect COF, when taken:

- Taken JT, JSR instructions to a destination in an R register

- If TC1.PTT is set to 00 - all RTS instructions are reported as indirect
  IF TC1.PTT is set to 01 - only a subset of RTS (RAS valid) instructions are reported as indirect
- RTS.STK instructions, and RTS (RAS no valid)
- Taken RTE, RTE.CIC
- CONT.*n*
- Taken TRAP, Illegal, DEBUGE.*n*
- Any jump to an exception

The types of indirect COF that are described in the B-TYPE field in the branch history message are described in Table 8-82.

Table 8-82. Indirect Branch Type (B-TYPE) format

B-TYPE value	Condition
00	Taken JT or JSR to an R register
01	Exception jump (including TRAP, Illegal, DEBUGE.*n*)
10	Return from subroutine (RTS, RTS.STK)
11	Return from exception (RTE, RTE.CIC)

8.9.10.4 Treating RTS return address as a direct COF

Technically, a return from subroutine is an indirect COF because the return address is restored from memory. However, in most cases, subroutines return to the VLES following the one from which they were called. The SC3900 core uses this fact to implement a fast subroutine return mechanism (RAS), which is based on an internal stack of four return PCs, to support fast return address prediction of up to four subroutine nesting levels. For a more detailed explanation of this mechanism, see the *SC3900 FVP Core Reference Manual*. The SC3900 fast access return is exposed in the architecture, meaning that in case the user manipulates the return address in memory but does not clear the RAS stack, the core will jump to the internally stored address and not to the address retrieved from memory. When the user wishes to enforce a return from a return address in memory, the RTS.STK instruction should be used.

This architectural exposure of the return stack enables the trace mechanism to use it safely - generally speaking, when an RTS is executing with a RAS valid indication, it means that the return address is the same as was saved by the previous JSR, assuming that previous JSR was reported earlier in the trace stream. An RTS (valid RAS) will be reported as indirect in only in cases where there is a possibility that the tool cannot deduce when the return address is. This includes all cases of RTS where RAS is not valid, and also some RAS valid cases—in case the matching JSR was not reported in the trace (due to message loss, for example).

Tracing RTS as a direct COF is enabled in the PTT field in TC1 - selecting if full program trace treats RTS always as indirect (00 setting), or reports some of the RTS instructions as direct (setting 01). The mechanism includes a saturating counter of values 0 to 4 (including), and works as follows:

- When full program trace turns active (upon enabling, a START event) - the counter is cleared
- Upon a taken JSR or BSR instruction - the counter is incremented (cannot be incremented above 4)
- Upon any taken RTS instruction - the counter is decremented (cannot be decremented below 0)

- Upon the following conditions the counter is cleared:
  - An indication from the core that the RAS stack was cleared (a taken RTS.STK instruction, CLRIC instruction, RTE.CIC instruction)
  - Execution of a taken RTS instruction for which the RAS was not valid
- When executing a taken RTS instruction with a valid RAS indication, if the counter value is greater than 0, the RTS is reported as a direct COF. If the value is 0, the RTS is reported as indirect.

8.9.10.5 Resource full message

A resource full message is generated when a trace resource that is used for trace administration and is implicitly accumulated was filled. The messages describes the resource type (via the RCODE field) and dumps the associated data, if relevant. According to the Nexus standard the resource full message is associated with program trace, however there are resource full conditions not related to program tracing where this message is generated (timestamp full and profiling counter overflow). The resources for which this message is generated are listed in this table.

Table 8-83. Resources that are reported by the resource full message

Resource	RCODE	Trace mode	RDATA	Description
Instruction counter	0000	Full program trace	0	The number of sequential VLES reported since the last reported ICNT value, last taken COF, SYNC message or last ICNT resource full massage (the latest event). No meaningful RDATA
Branch history buffer	0001	Full program trace	Buffer contents (35 bits)	The contents of the COF history buffer, reported only for contents not reported with branch indirect and program correlation messages
Timestamp counter	1000	All trace modes for which the time stamp is enabled (TC1.TSEN > 0)	0	The message is reported upon a timestamp counter overflow event. No meaningful RDATA
Profiling counter overflow	1010	Profiling trace	14 bits: [5:0] - overflow of counter B3..A0 [9:6] - bits [17:14] of the counter window of Triad A [13:10] - bits [17:14] of the counter window of Triad B	Upon an overflow event in a traced profiling counter. The overflow event is detected when the most significant trace bit in the window defined for this counter is set, per the definition in Section 8.9.5.2, “Profiling trace”. Per triad, if more than one counter overflows, the bits for the lower-indexed overflowing counter are reported

8.9.10.6 Program trace—correlation message

Program correlation messages are used to correlate processor events to instructions in the full program flow. The list of events that could be reported is listed in Table 8-84. Each EVCODE could be individually activated by setting a bit in TC4, see Section 8.9.4.4, “Trace Control Register 4 (TC4)”. Note that some of the EVCODEs are disabled by default in TC4.

Table 8-84. Events reported in program correlation messages

Event	EVCODE	CDATA	Default	Description
Entry into idle mode	#1	1–36	Enabled	Dumps and clears COF history upon transition from execution into debug mode or a low power mode (DOZE)
De-activation of full program trace	#5	1–36	Enabled	Dumps and clears COF history upon de-activation of full program trace. “De-activation” is when full program trace turns not active.
Assertion of debug_in_event_0	#9	1	Disabled	—
IEU event Q0	#10	1	Disabled	The same event may cause counter sampling, for implementing event-driven profiling (see Table 8-53).
IEU event Q2	#11	1	Disabled	The same event may cause counter sampling, for implementing event-driven profiling (see Table 8-53).

In case several correlation events occurred in the same cycle, Only the event with the lowest EVCODE causes a correlation message.

8.9.10.7 I-CNT field

The trace unit holds an 8-bit instruction counter that counts the number of committed VLES since it was last cleared. The value of this counter is reported in several message types, all associated with full program tracing, in the I-CNT (Instruction Count) field. The following messages report the I-CNT:

- Indirect Branch History Message (TCODE: 28) reporting interrupts (BTYPE: 01—exception)
- Program Correlation Message (TCODE: 33)

The following events clear the instruction counter:

- Enabling of full program tracing
- A Program Synchronization Message (TCODE: 9). See Section 8.9.10.2, “Program trace synchronization.”
- A trace message with an I-CNT field
- A taken COF (direct or indirect)

The I-CNT value reflects the number of committed VLES since the last message clearing the counter, not including the VLES due to which the counter was cleared, and the message reporting the I-CNT. For example:

```
V1: taken COF              - clears counter, V1 not counted
V2:
V3: not taken COF          - does not clear the counter, V3 counted
V3:
V4: Program correlation message: reports I-CNT = 3, clears counter
```

and:

```
V1: program synchronization message- clears counter,
                                     V1 (VLES of reported PC) not counted
V2:
```

```
V3:
V*: Indirect branch message - exception - reports I-CNT = 2 (V* is not a VLES)
```

Note that some of the messages that involve reporting I-CNT or clearing the counter are not associated with a VLES (for example an indirect branch reporting an exception, a resource full - instruction counter message). However since all messages that clear or report the counter are not included in the count the interpretation of I-CNT remains consistent also for these cases. Note that the Resource Full message that reports an I-CNT full (TCODE: 27, RCODE:0) does not clear the I-CNT, the counter is cleared by itself when wrapping around.

8.9.11 Data acquisition trace messages

The core programming model has two registers to support data acquisition trace messages (DQM).

- TMTAG: Trace Message Tag: a 32-bit register, bits [7:0] hold the ID tag that will be issued as part of the DQM. Writing to this register does not trigger a DQM.
- TMDAT: a 32-bit write-only register. Upon writing to it, if trace is enabled and properly configured, a DQM will be generated, with IDTAG taken from TMTAG[7:0].

These two registers can always be written to by the program, in any working mode, without any affect on the executed program itself. Note that TMDAT has another function during debug mode that does not interfere with its functionality during trace (in debug mode tracing is implicitly disabled) - see Section 8.9.4.8, “TMDAT Image Register (TMDATI).” For more information on the TMTAG and TMDAT registers, see the *SC3900 FVP Core Reference Manual*.

8.9.12 Profiling trace messages

The profiling trace messages are used to dump snapshots of the profiling counters into the trace stream. Depending on the type of the message, either one or two triads are sampled. As described in Section 8.9.5.2, “Profiling trace”, only 15 bits per counter are sampled, and these are supplemented by resource full (counter overflow) messages, see Section 8.9.10.5, “Resource full message”. The events for which the counters are sampled are defined in the triad control registers in the profiling unit (PU), see Table 8-53. If profiling trace is enabled while these sampling events occur, the respective profiling trace messages are generated. The sampling reason triggering the message is reported in the message and described in Table 8-85. The two-triad message (TCODE 60) uses the 3-bit values defined in the SMP field. The single-triad message cannot accommodate a 3-bit SMP field, so it is split into two TCODE numbers (58 and 59), each using a different 2-bit sampling reason field (SMP1 and SMP2, respectively). Other than that, there is no difference between the TCODE 58 and 59 messages. In case more than one sampling event occurred in the same cycle, the reported event is done according to the priority defined at the end of Section 8.8.2.2, “Profiling Triad Control Register A, B (PTCRA/B).”

Table 8-85. Values of the SMP, SMP1 and SMP2 fields

	Value		Value	Meaning	Comments
SMP	000	SMP1	00	JSR, BSR	—
	001		01	RTS	—
	010		10	Jump to interrupt	—
	011		11	RTE, RTE.CIC	—
	100	SMP2	00	RTS.TK	—
	101		01	DEBUGEV.*n* instruction	—
	110		10	IEU event	—
	111		11	Reloadable counter RZ or Triad overflow	In general only a reloadable counter is of use for generating a trace event — the triad 31-bit overflow is intended for software-based profiling.

When a single triad is traced, only the sampling events defined for this triad will cause a trace message. When a two triads are traced, only the events that were defined for triad A will have an effect.The triad counters should be configured for shared control by the configuration bits of Triad A (PCCSR.SDTP = 1, see Section 8.8.2.1, “Profiling Counters Control and Status Register (PCCSR)”).

8.9.13 Watchpoint trace messages

The watchpoint message (TCODE 15) is used to report debug events into the trace stream. A total of 32 debug events can generate a watchpoint message, each could be individually enabled in the TWMSK register, see Section 8.9.4.6, “Trace Watchpoint Mask Register (TWMSK). The events are split into two groups, so that a watchpoint message includes at most 16 events. The GRP field in the watchpoint message defines which group is reported, as described in Table 8-86.

Table 8-86. Values of the GRP field in the watchpoint message

Value	Meaning
00	PCDA events
01	Imprecise events
1x	Reserved

The values for source indicators in the watchpoint message (WPHIT) are described in Table 8-87.

Table 8-87. Address detector Watchpoint Hit (WPHIT) field formats

WPHIT value	Meaning for GRP=0	Meaning for GRP=1
XXXXXXXXXXXXXXX1	Reserved	IEU Q0 event
XXXXXXXXXXXXXX1X	Reserved	IEU Q1 event
XXXXXXXXXXXXX1XX	Reserved	IEU Q2 event
XXXXXXXXXXXX1XXX	Reserved	IEU Q3 event

Table 8-87. Address detector Watchpoint Hit (WPHIT) field formats (continued)

WPHIT value	Meaning for GRP=0	Meaning for GRP=1
XXXXXXXXXX1XXXX	Reserved	Reserved
XXXXXXXXX1XXXXX	Reserved	Reserved
XXXXXXXX1XXXXXX	Reserved	Reserved
XXXXXXX1XXXXXXX	Reserved	Reserved
XXXXXX1XXXXXXXX	PCDA detector 0	Reserved
XXXXX1XXXXXXXXX	PCDA detector 1	Reserved
XXXX1XXXXXXXXXX	PCDA detector 2	Reserved
XXX1XXXXXXXXXXX	PCDA detector 3	Reserved
XX1XXXXXXXXXXXX	PCDA detector 4	Reserved
X1XXXXXXXXXXXXX	PCDA detector 5	Reserved
1XXXXXXXXXXXXXX	PCDA detector 6	Reserved
	PCDA detector 7	Reserved

When events from different groups occur at the same cycle, the message for the imprecise events is discarded and an error message is generated.

8.9.14 Timestamp correlation message

The timestamp correlation message is generated when the TC1.TSEN is configured to a non-zero value, and triggered in the following cases:

- In response to an external signal from the NPC. In this case, the NPC supplies a request tag and reason bits, which are sampled into the message, enabling the tool to interpret them and to correlate the different messages sent by the other clients.
- In response to an internal IEU event (enabled in TC1.TSCE field)—used to reflect a timestamp in use cases where the timestamp is not activated for every message. Note that in this case, the timestamp is added to the next profiling message as well.

The timestamp correlation type field T-TYPE supports both cases. Some values reflect the reason for external triggers, while others reflect internal reasons. Table 8-88 lists the values of the T-Type field. Note that for external triggers, the T-TYPE field is sampled from the interface, so the values in the table do not involve client logic.

Table 8-88. T-TYPE values in timestamp correlation messages

T-TYPE value	Trigger	Meaning
0xxxxx	External	Sampled from the interface, as defined in the NPC.

Table 8-88. T-TYPE values in timestamp correlation messages (continued)

T-TYPE value	Trigger	Meaning
1xxxxx	External	Reserved
11xxxx	Internal	110000 - Reserved 110001- IEU0 110010 - Reserved 110011 - Reserved 1100xx - Reserved

Chapter 9
Interfaces

9.1 Introduction

The SC3900 subsystem and cluster interfaces enable data traffic inside the cluster and to the rest of the system. They also provide a dedicated peripheral bus grid for accessing cluster and sub-system level peripherals by external masters. These interfaces use standard buses that enable easy integration with the different systems in which the FVP subsystem may be used.

9.2 Interfaces overview

The interface units are as follows:

- The SC3900 DSP subsystemCore/cache interface unit (CCIU) is used for L1 subsystem accesses to L2 (Kibo).
- The CoreNet bus interface unit (BIU) is the interface between the cluster and the CoreNet. The protocol is CoreNet.
- The peripheral bus is the interface for subsystem and cluster configuration registers. The protocol is a standard slave bus.
- AXI2ELINK block for MAPLE connections

9.3 Interfaces functional description

This figure shows the memory buses interconnect.

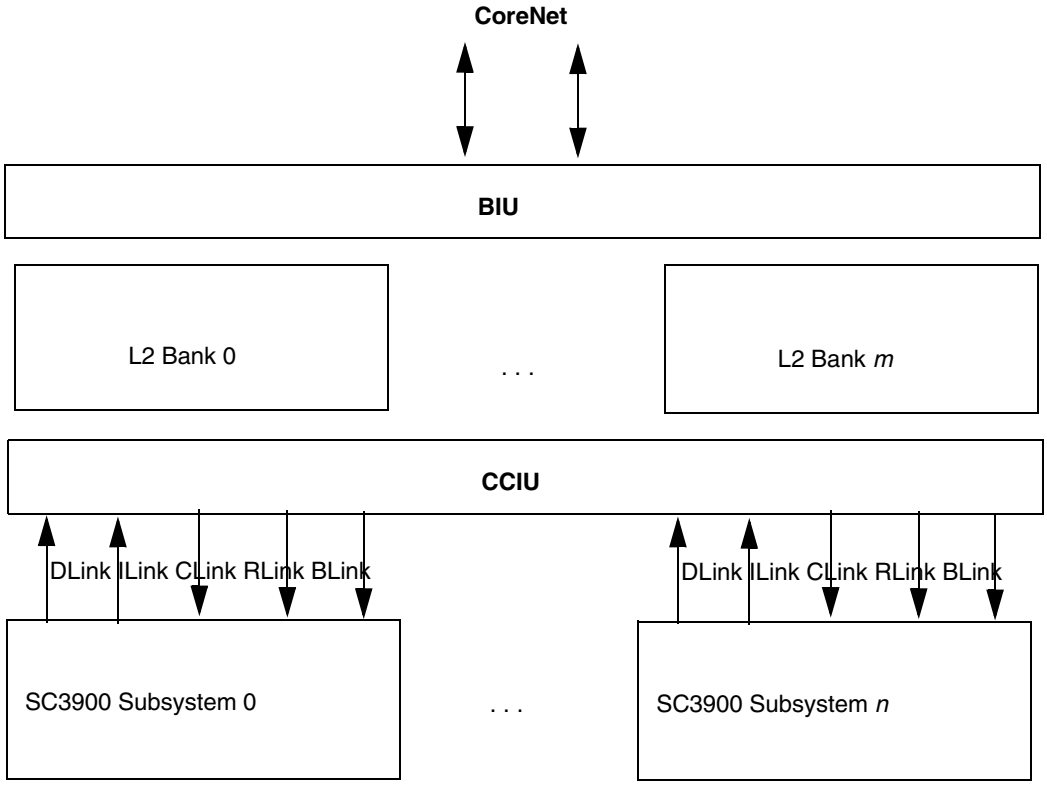

Figure 9-1. Memory buses interconnect

9.3.1 Core/Cache Interface Unit (CCIU)

CCIU is the interface unit between the L1 subsystems and the L2 cache banks. As the core-to-Kibo interface is synchronous, CCIU and L2 operate at the same clock as the core.

Each L1 subsystem is connected to CCIU via a logical interface called ELink, which consists of the following interfaces or “Links”:

- DLink—interface for L1 data requests towards L2 Cache
  - — Requests can be sent every clock cycle
  - — 512 bit wide write data bus
- ILink—interface for L1 instruction requests towards L2 Cache
  - — Requests can be sent every other clock cycle
- RLink—interface for read data to L1 subsystem
  - — 256 bit wide bus
  - — Data is sent in back-to-back manner, one CG (64 bytes) across two cycles

- CLink—interface for latency improvement for L2 cache miss data to be sent back to L1 cores
  - 256 bit wide bus
  - Data is sent directly from the BIU (SoC Fabric), generally one CG (64 bytes) across 4 cycles
- BLink—back invalidate interface towards L1 subsystem
  - Used for back invalidation of L1 subsystems.

All ELink data is assumed to be organized in big-endian byte order.

9.3.2 CoreNet Bus Interface Unit (BIU)

The SoC Fabric bus interface unit resides after the L2 cache and asynchronously reduces frequency from the L2 cache side to the CoreNet side by a ratio of 1.5 to 8. The CoreNet bus interface unit has the following features:

- Master port to the CoreNet:
  - 1 × 256 bit Data In used for reads and stashing
  - 1 × 256 bit Data out buses used gor writes and intervention snoops
  - 32 bit address buses (AIn, Aout)
  - Response buses to support snoops and stashing
- Pipelined address bus
- Support for tagged transactions to allow their out of order completion
- Coherency protocol—MESI+L, 64 bytes CG
- Data transactions of 1, 2, 4, 8, 16 and 64 byte
  - 1, 2, 4, and 8 byte accesses are single-beat and transfer the necessary number of valid bytes of data.
  - Addressing of the byte lanes is always big-endian (left to right) regardless of the endian mode of the core complex.

9.3.3 Peripheral bus

The peripheral bus is organized into a separate grid in the SC3900 cluster. This allows effective intracluster peripheral accesses and full SoC access to cluster peripherals. The bus is connected to the SoC's configuration control and status register (CCSR) tree root.

This table summarizes possible access paths.

Table 9-1. Peripheral access paths

Peripheral access	Path	Path properties
Same subsystem	Local slave bus	64 bit slave bus
Same cluster but different subsystem	Cluster slave bus	32 bit slave bus
Different cluster	To SoC via CCM bus From SoC via CCSR bus	32 bit slave bus

The main properties of the cluster peripheral bus grid are as follows:

- Cluster level interconnect using 32 bit slave bus
- Subsystem level interconnect using 64 bit slave bus
- Intracluster peripheral accesses close without exiting to SoC and changing frequency domains, providing best performance
- Asynchronous CCSR slave port from SoC
- Full error support
  - — Precise memory exception for read accesses
  - — Interrupt with address sample for write accesses
- Automatic transaction closure to non mapped peripheral space - core is not stalled due to wrong access
- Memory barriers support

The slave bus protocol has no pipeline, so it is important to use the full available bus width on the subsystem level (64 bit). Most of the registers are grouped together to allow a simultaneous 2long move.

It is also important to use both data buses when accessing registers on all levels. Using double 64 bit access for local subsystem peripherals is 3× times faster than four single 32-bit accesses. Double 32-bit access when accessing cluster level peripherals can win up to 1.5 time relative to one access per VLES.

9.3.3.1 Peripheral bus memory map

The SC3900 subsystem provides a 36-bit physical address bus or a 64-Gbyte address space. The SC3900 cluster dedicates 256 Kbytes to the CCSR peripheral bus. CCSR address space is visible from each SC3900 FVP in the cluster and from the CCSR slave port.

In order to access peripheral bus from the FVP it should have data MMU descriptor configured with bank 0 attribute. All virtual addresses that hit bank 0 descriptor will exit to peripheral bus. After reset data MMU has preconfigured the descriptor for peripheral accesses that have a start address of 0xFEC00000 and size of 256 Kbytes.

Each subsystem has dedicated 32 Kbytes of the address space to its internal peripherals. It is important to note that subsystem peripheral base address depends on the cluster peripheral base address and subsystem number. The SC3900 FVP has a dedicated register that contains the subsystem number in the cluster and cluster number in the SoC. Peripheral address space should be mapped using MMU translation to the same virtual address in every core to simplify peripherals management (including MMU itself).

This table shows the cluster peripheral address map.

Table 9-2. Cluster peripheral address map

Peripheral	Base address	Size
Subsystem 0	0x00000	32 Kbyte
Subsystem 1	0x08000	32 Kbyte
Reserved	0x10000–1FFFF	—

Table 9-2. Cluster peripheral address map (continued)

Peripheral	Base address	Size
L2 Registers	0x20000	4 Kbyte
Watchdog Timer 0	0x21000	1 Kbyte
Watchdog Timer 1	0x21400	1 Kbyte
Reserved	0x21800–3FFFF	—

Some registers have access restrictions, such as debug mode or the source of the access. Access to L1 peripherals is automatically closed with error indication when L1 clocks are stopped.

The SoC CCSR slave port can be used to access cluster and subsystem peripherals by external masters. SoC and local cluster accesses arbitrate using a round robin algorithm. SoC accesses are allowed even if the core is held due to an access on the main bus.

This table shows the subsystem peripheral address map. When there is a non-mapped address, an error is asserted by the MMU (precise exception for reads and dedicated interrupt for writes in the local cluster).

Table 9-3. Subsystem peripheral address map

Peripheral	Base address	Size
DU	0x4000	4 Kbyte
MMU	0x5000	4 Kbyte
CME	0x6000	1 Kbyte
EPIC	0x6400	1 Kbyte
TIMER	0x6800	1 Kbyte
Reserved	0x6C00–7FFF	—

Chapter 10
Timers

The SC3900 DSP subsystem has four independent timers (indexed from 0 to 3) with non-freeze clocks. Each timer is a 32-bit down-count counter that operates in either single- or multitrigger mode, as follows:

- Single-trigger mode

  The timer is either programmed to a predefined value, or it has 0xFFFF_FFFF as its reset value. When enabled through its control register, the timer counts down non-freeze, free-running clock cycles. After counting the predefined number of clock cycles, it stops counting, disables itself, and generates an interrupt to the EPIC.
- Multitrigger mode

  Each timer has a 32-bit preload register in which a predefined value must be programmed (its reset value is 0xFFFF_FFFF). An enabled timer loads the value from the preload register to the timer value register and starts to count down non-freeze, free-running clock cycles. After counting the predefined number of clock cycles, the timer generates an interrupt to the EPIC, reloads the value from the preload register, and continues to operate.

NOTE

Changing trigger modes is allowed only when the relevant counter is disabled.

The SC3900 DSP cluster has two watchdog timers (one for each SC3900 DSP subsystem). Each timer is a 64 -bit down count that operate in double trigger mode:

- The timer is either programmed to a predefined value, or it has 0xFFFF_FFFF as its reset value. When enabled through its control register, the timer counts down non-freeze, free-running clock cycles. After counting the predefined number of clock cycles, if interrupt is enabled it generates an interrupt to the EPIC and reload counter value and recount. Upon second expiration, pre-defined wrs bits are output. Interrupt is disable by any access to enable bit of the counter [ten]. WRS bits are cleared by writing “11”.

At reset, the timer value and the timer preload registers are written with ones. The timer control registers are written with zeros. The timers count the clock cycles when the SC3900 DSP subsystem core is in the wait processing state but not when it is in stop processing state or received a debug request to freeze the timers.

Additionally, a pair of timers can be configured to work in chaining mode, effectively creating a 64-bit timer. In this mode, one timer with an even index should be configured to work in multitriggered mode while a second timer with a corresponding odd index counts the number of times the first timer reaches zero. Watchdog timers are set to 64-bit counter only.

NOTE

Changing to/from chain mode is allowed only when the relevant counters are disabled.

The value of the chained timers can be accessed using a single 64 bit load instruction. If 32 bit accesses must be used (for example for external access to the timer registers), the shadow control register (TM_SC) must be written with the desired selection of the chained timers pair. This write causes automatic sampling of both timers' values into dedicated shadow registers (TM_S0 and TM_S1) that should be read afterwards to obtain a frozen timer value.

Each timer can be temporarily disabled by setting the FREEZE bit. Setting this bit stops the count, and resetting it resumes the count from the value at which it stopped, without reloading the preload value. However, disabling the counter and enabling it with the TEN bit reloads the preload value. Do not enable the timer while the FREEZE bit is set.

Re-starting WATCHDOG TIMER ONLY by writing 1 to the TEN bit will reload its value from the preload registers and the count will start once again. For TIMER in order to restart it from preload values TEN bit must be disabled and then enabled.

An external synchronization option exists for every timer. It allows easy synchronization of timers in different subsystems. Externally synchronized timers operate only if an external enable signal is detected. See the SoC Reference Manual for details about external enable signal connectivity.

The steps of programming a timer are as follows:

1. Disable the timer by writing to the timer control register.
2. Write the timer value/preload register.
3. Enable the timer by writing to the timer control register. This can be done together with updating the preload register using 64 bit register access.

Values 0 and 1 must not be written to the counter value or the preload registers. The chained timer pair should be programmed starting from the odd index timer.

Timer initial value must not be smaller than 0x000_000F.

The SC3900 core should be programmed to access its local (subsystem level) timer registers using 64 bit accesses (providing best performance), but the external access path to the timer can support only 32 bit accesses. To prevent ambiguity in the chained timer value read process, the local chained timer value must be accessed only using 64 bit accesses. Thus local timer read never causes a shadow value register change, and these registers should serve only external timer accesses.

The steps of the external read of chained timers value are as follows:

1. Write 0 or 1 to TM_SC register to sample the first or second chained timer pair.
2. Execute memory barrier to guarantee order between write and read.
3. Read from TM_S0 shadow value register.
4. Read from TM_S1 shadow value register.

Read of local timer has only one step: Read 2long (64-bit) from <i> and <i+1> value registers.

10.1 Register summary

All timer registers are memory-mapped and can be written or read by the core or external master through the peripheral bus. There is only one access per execution set for all timer registers. Reserved or unused bits in all registers should be written as zeros, and the read value should be masked. Writing to unimplemented or read-only registers generates a peripheral bus error. Reading from unimplemented registers generates a peripheral bus error and undefined data.

Table 10-1 defines the memory offset of the timer registers. This offset should be added to the base address of the SC3900 subsystem peripheral bus to obtain an actual address. All registers are 32 bits, but every two sequential registers could be accessed by a single aligned 64 bits access from the local core.

Table 10-1. Timer memory map

Address offset from timer base	Use	Section/page
000	Timer 0 Control Register (TM_T0C)	10.2.2/10-5
004	Timer 0 Pre-load Register (TM_T0P)	10.2.3/10-6
008	Timer 1 Control Register (TM_T1C)	10.2.1/10-4
00C	Timer 1 Pre-load Register (TM_T1P)	10.2.3/10-6
010	Timer 2 Control Register (TM_T2C)	10.2.2/10-5
014	Timer 2 Pre-load Register (TM_T2P)	10.2.3/10-6
018	Timer 3 Control Register (TM_T3C)	10.2.1/10-4
01C	Timer 3 Pre-load Register (TM_T3P)	10.2.3/10-6
020	Timer 0 Value Register (TM_T0V)	10.2.4/10-7
024	Timer 1 Value Register (TM_T1V)	10.2.4/10-7
028	Timer 2 Value Register (TM_T2V)	10.2.4/10-7
02C	Timer 3 Value Register (TM_T3V)	10.2.4/10-7
030	Shadow Register Control (TM_SC)	10.2.5/10-7
034	Shadow Value Register 0 (TM_S0)	10.2.6/10-8
038	Shadow Value Register 1 (TM_S1)	10.2.6/10-8
100	WD Timer 0 Control Register (WD_T0C)	10.2.7/10-9
104	WD Timer 0 Pre-load Register (WD_T0P)	10.2.8/10-10
10C	WD Timer 1 Pre-load Register (WD_T1P)	10.2.8/10-10
130	WD Shadow Register 0 Control (WD_SC0)	10.2.9/10-10
134	WD Shadow Value Register 0 (WD_S0)	10.2.10/10-11
138	WD Shadow Value Register 1 (WD_S1)	10.2.10/10-11
200	WD Timer 1 Control Register (WD_T1C)	10.2.7/10-9
204	WD Timer 2 Pre-load Register (WD_T2P)	10.2.8/10-10
20C	WD Timer 3 Pre-load Register (WD_T3P)	10.2.8/10-10

Table 10-1. Timer memory map (continued)

Address offset from timer base	Use	Section/page
230	WD Shadow Register 1 Control (WD_SC1)	10.2.9/10-10
234	WD Shadow Value Register 2 (WD_S2)	10.2.10/10-11
238	WD Shadow Value Register 3 (WD_S3)	10.2.10/10-11

10.2 Timer registers

The following sections describe the timer registers.

10.2.1 Timer 1 and Timer 3 Control Registers (TM_T1C and TM_T3C)

TM_T1C and TM_T3C, shown in the figure below, are responsible for the operation of Timer 1 and Timer 3 correspondingly. Access to TM_TiC register can be grouped into 2 long (64-bit) access to TM_TiP to improve performance.

Address 0x008 (TM_T1C)
0x018 (TM_T3C)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R												TCHAIN	TSYNC	TFREEZE	TMODE	TEN
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10-1. Timer 1 and Timer 3 Control Registers (TM_T1C and TM_T3C)

This table describes the TM_T1C and TM_T3C fields.

Table 10-2. Register TM_T1C TM_T3C bits description

Field	Description
4 TCHAIN	Chaining enable. 0 This timer counts clock cycles. 1 This timer counters a number of times the even timer reaches zero (for example, timer 1 counts the number of zeros of timer 0). If TCHAIN is set, then the rest of the timer controls are taken from the even timer control register.
3 TSYNC	External synchronization enable. 0 No external synchronization is needed. 1 External synchronization is enabled. Timer waits for high level external input signal to start counting.
2 TFREEZE	Timer Freeze. Freezes the operation of the timer. 0 Resetting this bit means that the timer continues its operation from the value it was stopped at. 1) Setting this bit freezes the operation of the timer. The timer must not be enabled while the FREEZE bit is set.

Table 10-2. Register TM_T1C TM_T3C bits description (continued)

Field	Description
1 TMODE	Timer Mode. Indicates the mode of the timer. 0 Timer operates in single-trigger mode. Timer generates an interrupt when reaching zero, stops counting and disables itself. 1 Timer operates in multi-trigger mode. The value from the preload register is loaded to the timer value register. The timer then starts to count down non-freeze, free-running clock cycles or zero events of the even timer companion. When reaching zero, it generates an interrupt to the EPIC, reloads the value from the preload register and continues to operate.
0 TEN	Timer Enable. Enables/disables the timer. 0 Disabled 1 Enabled. The timer waits for external synchronization if needed and then starts to count non-freeze, free-running clock cycles or zero events of the even timer companion.

10.2.2 Timer 0 and Timer 2 Control Registers (TM_T0C and TM_T2C)

TM_T0C and TM_T2C, shown in Figure 10-2, are responsible for the operation of Timer 0 and Timer 2 correspondingly. Access to TM_TiC register can be grouped into 2 long (64-bit) access to TM_TiP to improve performance.

Address 0x000 (TM_T0C)
0x010 (TM_T2C)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R													TSYNC	TFREEZE	TMODE	TEN
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10-2. Timer 0 and Timer 2 Control Registers (TM_T0C and TM_T2C)

Table 10-3 describes the TM_T0C and TM_T2C fields.

Table 10-3. Register TM_T0C TM_T2C bits description

Field	Description
3 TSYNC	External synchronization enable. 0 No external synchronization is needed. 1 External synchronization is enabled. Timer waits for high level external input signal to start counting.
2 TFREEZE	Timer Freeze. Freezes the operation of the timer. 0 Resetting this bit means that the timer continues its operation from the value it was stopped at. 1) Setting this bit freezes the operation of the timer. The timer must not be enabled while the FREEZE bit is set.

Table 10-3. Register TM_T0C TM_T2C bits description (continued)

Field	Description
1 TMODE	Timer Mode. Indicates the mode of the timer. 0 Timer operates in single-trigger mode. Timer generates an interrupt when reaching zero, stops counting and disables itself. 1 Timer operates in multi-trigger mode. The value from the preload register is loaded to the timer value register. The timer then starts to count down non-freeze, free-running clock cycles. When reaching zero, it generates an interrupt to the EPIC, reloads the value from the preload register and continues to operate.
0 TEN	Timer Enable. Enables/disables the timer. 0 Disabled 1 Enabled. The timer waits for external synchronization if needed and then starts to count non-freeze, free-running clock cycles.

10.2.3 Timer Preload Registers (TM_TiP, i = 0,1,2,3)

TM_TiP, shown in Figure 10-3, contain the preload value of corresponding timer. Access to TM_TiP register can be grouped into 2 long (64-bit) access to TM_TiC to improve performance.

Address 0x004 (TM_T0P)
0x00C (TM_T1P)
0x014 (TM_T2P)
0x01C (TM_T3P)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PV															
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PV															
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 10-3. Timer i Preload Register (TM_TiP). i = 0, 1, 2, 3

Table 10-4 describes the TM_TiP fields.

Table 10-4. Register TM_T0P TM_T1P TM_T2P TM_T3P bits description

Field	Description
31–0 PV	Preload Value. Timer initial value must not be smaller than 0x000_000F

10.2.4 Timer Value Registers (TM_TiV, i = 0,1,2,3)

The TM_TiV, shown in Figure 10-4, contain the value of the corresponding timer. The SC3900 core read from a chained pair of value registers (TM_T0V/TM_T1V or TM_2V/TM_3V) must be done as 64 bit access. An external read from a chained pair using 32 bit access must read the timer value through the corresponding shadow value register (TM_S0 or TM_S1).

Address 0x020 (TM_T0V)
0x024 (TM_T1V)
0x028 (TM_T2V)
0x02C (TM_T3V)

Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	TV															
W																
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	TV															
W																
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 10-4. Timer i Value Register (TM_TiV). i = 0, 1, 2, 3

Table 10-5 describes the TM_TiV fields.

Table 10-5. Register TM_T0V TM_T1V TM_T2V TM_T3V bits description

Field	Description
31–0 TV	Timer Value

10.2.5 Shadow Registers Control (TM_SC)

The TM_SC, shown in Figure 10-5, contains the control bits of shadow timer registers for multitrigger mode value read. An external read from a chained pair using 32 bit access must start from write to TM_SC and then read the desired timer pair from the shadow value registers (TM_S0 and TM_S1).

Address 0x030 (TM_SC)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																SSC
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10-5. Shadow Register Control (TM_SC)

Table 10-7 describes the TM_SC fields.

Table 10-6. Register TM_SC Bits Description

Field	Description
0 SSC	Shadow sampling control. 0 Timer value registers 0 and 1 are sampled into shadow value registers 0 and 1 correspondingly 1 Timer value registers 2 and 3 are sampled into shadow value registers 0 and 1 correspondingly

10.2.6 Timer Shadow Value Registers (TM_Si, i = 0,1)

The TM_Si, shown in Figure 10-6, contain the sampled shadow value of multitrigger mode timer as defined by the TM_SC register.

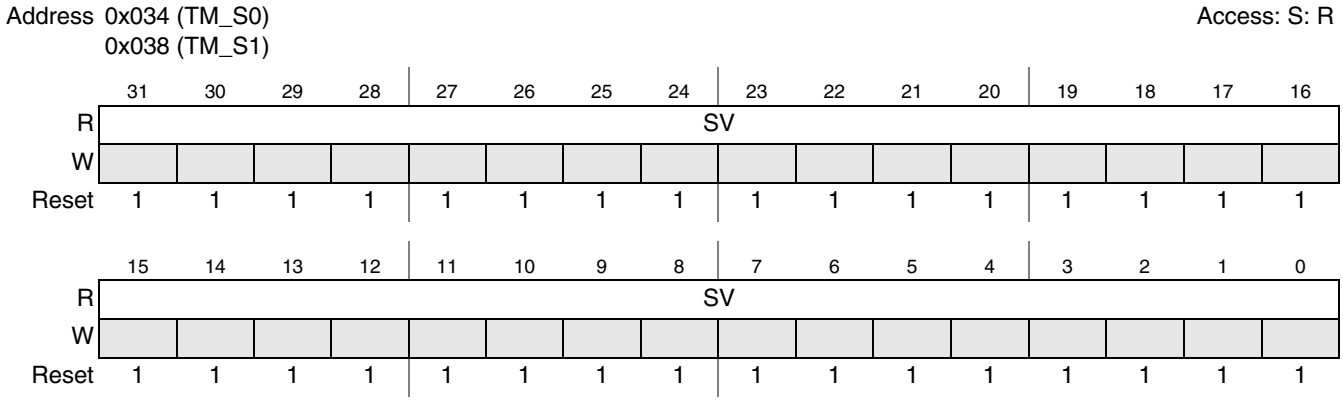

Figure 10-6. Timer i Shadow Value Register (TM_Si). i=0,1

Table 10-7 describes the TM_Si fields.

Table 10-7. Register TM_S0 TM_S1 bits description

Field	Description
31–0 SV	Shadow Timer Value Sampled from the corresponding value register during write to TM_SC register.

10.2.7 WD Timer 0 control Register (WD_T0/1C)

WD_T0C, shown in Figure 10-2, are responsible for the operation of WD Timer 0 and WD Timer 1.

Address 0x100 (WD_T0C) Access: S: RW

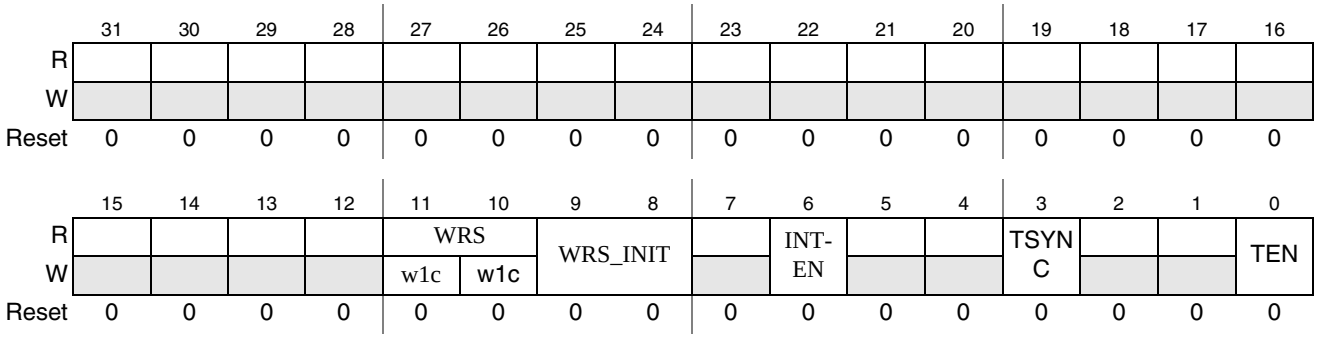

Figure 10-7. WD Timer 0 and WD Timer 1Control Registers (WD_T0C and WD_T1C)

Table 10-3 describes the WD_T0C and WD_T1C fields.

Table 10-8. Register WD_T0C WD_T1C bits description

Field	Description
11-10 WRS	In WD timer use only, WRS value. WRS value can be reset request from SoC logic. For exact definition see SoC reference manual
9-8 WRS_INIT	In WD timer use only, WRS value to be driven on second watchdog timer expiration.
6 INTEN	In WD timer use only, enable/disable interrupt. Once this bit was set it can not be cleared till reset.
3 TSYNC	External synchronization enable. 0 No external synchronization is needed. 1 External synchronization is enabled. Timer waits for high level external input signal to start counting.
0 TEN	Timer Enable. Enables/disables the timer. 0 Disabled 1 Enabled. The timer waits for external synchronization if needed and then starts to count non-freeze, free-running clock cycles.

10.2.8 Timer Preload Registers (WD_TiP, i = 0,1,2,3)

WD_TiP, shown in Figure 10-3, contain the preload value of WD timer.

Address 0x104 (WD_T0P)
0x10C (WD_T1P)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	PV															
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PV															
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 10-8. Timer i Preload Register (WD_TiP). i = 0, 1,2,3

Table 10-4 describes the WD_TiP fields.

Table 10-9. Register WD_T0P WD_T1P WD_T2P WD_T3P bits description

Field	Description
31–0 PV	Preload Value.Timer initial value must not be smaller than 0x000_000F

10.2.9 Shadow Registers Control (WD_SC0/1)

The WD_SC, shown in Figure 10-5, contains the control bits of shadow timer registers.

Address 0x130 (WD_SC)

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W																SSC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10-9. Shadow Register Control (WD_SC0/1)

Table 10-7 describes the WD_SC0/1 fields.

Table 10-10. Register WD_SC bits description

Field	Description
0 SSC	Shadow sampling control. Write 0 to sample

10.2.10 Timer Shadow Value Registers (WD_Si, i = 0,1,2,3)

The WD_Si, shown in Figure 10-6, contain the sampled shadow value of the timer as defined by the WD_SC register.

Address 0x134 (WD_S0)
0x138 (WD_S1)

Access: S: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	SV															
W																
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	SV															
W																
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 10-10. Timer i Shadow Value Register (WD_Si). i=0,1,2,3

Table 10-7 describes the WD_Si fields.

Table 10-11. Register WD_S0 WD_S1 bits description

Field	Description
31–0 SV	Shadow Timer Value Sampled from the corresponding value register during write to WD_SC register.

10.3 Timer restrictions

- Write to TM_TxC, or any register affecting an enabled timer should be always immediately followed with a DBAR.SCFG command.
- Do not write a zero or one to the timer counter value register when the counter is activated in normal mode.

Appendix A
Revision History

This appendix provides a list of the major differences between the *SC3900 FVP Cluster Reference Manual*, Revision C through Revision D.

A.1 Changes from Revision C to Revision D

Major changes to the *SC3900 FVP Cluster Reference Manual*, from Revision C to Revision D are as follows:

Section, page	Changes
xxv	Under the heading ‘Organization,’ added a reference to Chapter 8, “Debug and Trace Support.”
Chapter 3/3-1	Added MMU address programming limitations.
3.1/3-2	Updated text in bulleted item from “The MMU generates kill signals for the XA/XB/RP buses for errant accesses” to “The MMU generates kill signals for the core program and data buses for errant accesses.”
Table 3-5./3-19	In 0010 error ID row, added the following: “Bank 0 is always not-protected. Any master can access in Read or Write to Bank 0. The Permission Violation Error will not occur on the accesses to Bank 0.”
Table 3-5./3-19	In the 1000 error ID row (Illegal access gathering), added the following three items to the list of bulleted items: “Load with size 64B on bus B,” “Load_reserved with size 64B,” and “Destructive load with size 64B.”
3.2.2/3-5	In the third paragraph, added the following text: “This separation also applies at the effective address level in the core (bits 31:0 of the virtual address). The application has to allocate different effective addresses to memory used by tasks, and to memory used for system and OS code.”
3.2.2.4/3-13	Changed note text from “A non-overlapping memory regions programming, in two coupled descriptors, has no priority influence” to “If memory regions that are programmed in two coupled descriptors do not overlap, there is no priority relation between them.”
3.2.4.2.2/3-19	Changed references from “XA/XB” and “RP” to “core data bus” and “core program bus,” respectively.
3.3/3-28	Updated the text from “Register access width is restricted to the memory access width as shown in Table 3-9, which describes the MMU memory map” to “Table 3-9 describes the MMU memory map.
3.3.4/3-40	For Privilege level, updated the M_DSR[16] reset value to be “1.”

Table 3-12./3-39	Added the following text to the cell corresponding to the description column: “If the error occurred on a cross 4k boundary access (M_DSR[DAAL]=2’b01,2’b10), the captured address will be of the second part of the access.”
Table 3-33./3-57	In the 256K segment size row, updated the boundary restriction from 32M to 64M.
Table 3-41./3-62	Added the following note: “The M_PVA does not contain the address alignment information; this means M_PVA[0] is always zero.”
3.3.34/3-63	For Privilege level, updated the M_PSR[16] reset value to be “1.”
Table 3-40./3-61	Updated L2pid field explanation to the following: “This field defines bits [1:0] of the L2 Partitioning ID.(Bits [4:3] of the L2 Partitioning ID are the core number inside the cluster. Bit [2] of the L2 Partition ID is always zero).”
Table 3-55./3-76	Updated L2pid field explanation to the following: “This field defines bits [1:0] of the L2 Partitioning ID.(Bits [4:3] of the L2 Partitioning ID are the core number inside the cluster. Bit [2] of the L2 Partition ID is always zero).”
3.3/3-28	Removed tables from within bit descriptions tables. Added the following tables after their corresponding bit descriptions tables: Table 3-34, Table 3-35, Table 3-36, Table 3-38, Table 3-39, Table 3-52, and Table 3-53.
3.4/3-76	Changed section heading from “Application Information (Exit from Reset)” to “Default state of the MMU when exiting from reset.”
Table 4-1./4-3	In “Allocation in SGB” row, changed text from “SGB keeps order between noncacheable accesses and proceed them to the higher level cache with higher priority, bypassing cacheable entries” to “SGB keeps order between peripheral accesses and proceeds them to the higher level cache without merging.”
5.4/5-8	Added Section 5.4, “Memory map and register definitions,” and updated Figure 5-1.
Chapter 6/6-1	Updated block resolution to 128 byte.
6.1/6-1	Updated text from “The CME is divided into subunits, which are responsible for managing the instruction and data caches, and also an interface unit that handles the configuration requests” to “The CME is divided into three subunits, one of which is an interface unit. These subunits are responsible for managing the instruction and data caches; the interface unit also handles configuration requests.”
6.2.1/6-2	Changed “slave configuration bus” to “peripheral bus.”
Chapter 8/8-1	Added chapter.
Table 8-12./8-18	Changed “Monitor (core access marked in the MMU as “peripheral”)” to “Monitor (core access marked in the MMU as “bank0”).”
Table 8-31./8-44	Corrected header name from DERSR to “Field description of DRASR.”
Figure 8-18./8-44	Corrected the header name from DERSR to “Debug Resource Activity Status Register (DRASR).”
Table 8-34./8-48	Replaced reset values of bits that are set per implementation from C to U.
Table 8-67./8-92	Replaced reset values of bits that are set per implementation from C to U.

Table 10-2./10-4	In the description of TCHAIN, added the following sentence: “If TCHAIN is set, then the rest of the timer controls are taken from the even timer control register.”

Appendix B
Code Restrictions

This appendix describes the software restrictions for the SC3900 FVP subsystem. These restrictions are in addition to the programming rules of the SC3900 listed in the *SC3900 FVP Core Reference Manual*.

B.1 General restrictions

1. The register access width (long, 2long) is restricted to that specified in the corresponding block register table. Registers marked as ‘long, 2long’ in the Access Width column of the table can be accessed either with long or 2long access width. Registers marked only as long can be accessed with long access width only. 2long accesses two nearby registers. The register access address should be aligned to its size.
2. The programming sequence for all bank0 registers should be followed with a DBAR.SCFG instruction to ensure the writes are flushed out of the SGB and are written to the destination register.
3. Semaphore operations using a reservation mechanism should be performed on a dedicated memory space that is defined as peripheral and non cacheable. This semaphore memory should be read only using “load reserve” core commands (VLESes with LDRSTC attribute). No regular load accesses to semaphore memory should be performed. Regular (nonconditional) write to semaphore should be followed with a DBAR.SCFG instruction to ensure the writes are flushed out of the SGB and are written to the destination register.
4. Sequence of decorated stores (stores with DECOR instruction) and NOTIFY instructions must be followed with a DBAR.SCFG instruction to ensure that the following reads from the same address will read properly updated by decoration data.
5. No unaligned write access should be performed so that it hits simultaneously shared and non-shared MMU descriptors.

B.2 Operation system restrictions

1. SYNC.B should be grouped together with load after store to the same address as load but with different task ID when accessing shared data memory segment. Regular task ID change does not require SYNC.B as it is inserted implicitly. For more details please check *SC3900 FVP Core Reference Manual*.

B.3 MMU restrictions

1. Write to M_CR, M_DSDVAL, M_PSDVAL, M_WMCFG registers should be always immediately followed with a DBAR.SCFG command.

2. Write to M_WMCFG register from the core should be done using only one write in the VLES and next VLES should have DBAR.SCFG barrier (to make sure that SGB is empty). Writing to M_WMCFG from external master should be limited to boot sequence.
3. Program MATT registers should be programmed from a code section that is not affected by the changed control registers (for example, running from a different descriptor). The last write to the configuration register should be followed with DBAR.SCFG. Before the transfer to the area where the change takes effect, there should be a CLRIC #3 instruction. The registers requiring these sequences after they are programmed are as follows:
   — Any write to M_CR in the context of a program action (clearing program memory error, enabling protection/translation, and so on)
   — Any write to M_PSDVAL
   — Last MMU register write if there is a write in the MMU writing sequence that affects program memory (change of a program memory descriptor)

   For example:

```
<write to MMU register>
....
<write to last MMU register in sequence>
DBAR.SCFG
nop
CLRIC #3
```

   For more details please check SC3900 core documentation.
4. Changing physical attributes of the memory (coherent, guarded, cacheable and L2 cache write policy), both in descriptor and using M_CR global override, should be done only after all layer cache invalidation (subject to SoC limitations).

B.4 CME restrictions

1. When CME is programmed by its own core and polled by the same core afterwards, a next code techniques should be used to guarantee CME execution and finite CME polling:
   — Data channel CME polling loop should include two sequential nop instructions
   — Program channel CME polling loop should use hardware loop and have no more than 2 VLESes.

```
_loop_again:
        doen.0 #$ffff
        loopstart0
        ld.l (r1),d0 ; r1 is a polled CME register
        tstbm.c.l #$mask,d0,p1
        if.p1 break.0 _cme_done
        nop
        loopend0
       bra _loop_again
_cme_done
```

B.5 EPIC restrictions

1. Update of the EPIC interrupt disable bits by the core should be done while all EPIC interrupts are disabled (both DI and DCI bits in SR2 core register asserted; see *SC3900 FVP Core Reference Manual* for more details). The sequence software should comply to is:
   — Set SR2.DI and SR2.DCI bits
   — Update interrupt enable bit in the EPIC
   — Restore SR2.DI and SR2.DCI bits