



Intel® IXP2800 Network Processor

Hardware Reference Manual

November 2003



Revision History

Date	Revision	Description
March 2002	001	First release for IXP2800 Customer Information Book V 0.4
May 2002	002	Update for the IXA SDK 3.0 release.
August 2002	003	Update for the IXA SDK 3.0 Pre-Release 4.
November 2002	004	Update for the IXA SDK 3.0 Pre-Release 5.
May 2003	005	Update for the IXA SDK 3.1 Alpha Release
September 2003	006	Update for the IXA SDK 3.5 Pre-Release 1
November 2003	007	Added information about Receiver and Transmitter Interoperation with Framers and Switch Fabrics.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

The IXP2800 Network Processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com>.

Copyright © Intel Corporation, November

intel®

Contents

1	Introduction	21
1.1	About this Document	21
1.2	Related Documentation	21
1.3	Conventions	22
2	Technical Description	23
2.1	Overview	23
2.2	Intel XScale® Core Microarchitecture	26
2.2.1	ARM Compatibility	26
2.2.2	Features	26
2.2.2.1	Multiply/Accumulate (MAC)	26
2.2.2.2	Memory Management	26
2.2.2.3	Instruction Cache	26
2.2.2.4	Branch Target Buffer	27
2.2.2.5	Data Cache	27
2.2.2.6	Interrupt Controller	27
2.2.2.7	Address Map	27
2.3	Microengines	29
2.3.1	Microengine Bus Arrangement	31
2.3.2	Control Store	31
2.3.3	Contexts	31
2.3.4	Datapath Registers	33
2.3.4.1	General-Purpose Registers (GPRs)	33
2.3.4.2	Transfer Registers	33
2.3.4.3	Next Neighbor Registers	34
2.3.4.4	Local Memory	35
2.3.5	Addressing Modes	37
2.3.5.1	Context-Relative Addressing Mode	37
2.3.5.2	Absolute Addressing Mode	38
2.3.5.3	Indexed Addressing Mode	38
2.3.6	Local CSRs	39
2.3.7	Execution Datapath	40
2.3.7.1	Byte Align	40
2.3.7.2	CAM	42
2.3.8	CRC Unit	45
2.3.9	Event Signals	46
2.4	DRAM	47
2.4.1	Size Configuration	47
2.4.2	Read and Write Access	48
2.5	SRAM	48
2.5.1	QDR Clocking Scheme	49
2.5.2	SRAM Controller Configurations	50
2.5.3	SRAM Atomic Operations	51
2.5.4	Queue Data Structure Commands	51
2.5.5	Reference Ordering	52
2.5.5.1	Reference Order Tables	52
2.5.5.2	Microengine Software Restrictions to Maintain Ordering	53

2.6	Scratchpad Memory.....	54
2.6.1	Scratchpad Atomic Operations.....	54
2.6.2	Ring Commands.....	55
2.7	Media and Switch Fabric Interface.....	56
2.7.1	SPI-4.....	58
2.7.2	CSIX.....	59
2.7.3	Receive.....	59
2.7.3.1	RBUF.....	60
2.7.3.1.1	SPI-4 and the RBUF.....	60
2.7.3.1.2	CSIX and RBUF.....	61
2.7.3.2	Full Element List.....	62
2.7.3.3	RX_THREAD_FREELIST.....	63
2.7.3.4	Receive Operation Summary.....	63
2.7.4	Transmit.....	64
2.7.4.1	TBUF.....	65
2.7.4.1.1	SPI-4 and TBUF.....	65
2.7.4.1.2	CSIX and TBUF.....	66
2.7.4.2	Transmit Operation Summary.....	67
2.7.5	The Flow Control Interface.....	68
2.7.5.1	SPI-4.....	68
2.7.5.2	CSIX.....	68
2.8	Hash Unit.....	68
2.9	PCI Controller.....	71
2.9.1	Target Access.....	71
2.9.2	Master Access.....	71
2.9.3	DMA Channels.....	72
2.9.3.1	DMA Descriptor.....	72
2.9.3.2	DMA Channel Operation.....	73
2.9.3.3	DMA Channel End Operation.....	74
2.9.3.4	Adding Descriptors to an Unterminated Chain.....	74
2.9.4	Mailbox and Message Registers.....	74
2.9.5	PCI Arbiter.....	75
2.10	Control and Status Register Access Proxy.....	76
2.11	Intel XScale® Core Peripherals.....	76
2.11.1	Interrupt Controller.....	76
2.11.2	Timers.....	77
2.11.3	General Purpose I/O.....	77
2.11.4	Universal Asynchronous Receiver/Transmitter.....	77
2.11.5	Slow Port.....	77
2.12	I/O Latency.....	78
2.13	Performance Monitor.....	78
3	Intel XScale® Core.....	79
3.1	Introduction.....	79
3.2	Features.....	80
3.2.1	Multiply/ACcumulate (MAC).....	80
3.2.2	Memory Management.....	80
3.2.3	Instruction Cache.....	81
3.2.4	Branch Target Buffer.....	81
3.2.5	Data Cache.....	81
3.2.6	Performance Monitoring.....	81

3.2.7	Power Management.....	81
3.2.8	Debug	81
3.2.9	JTAG.....	82
3.3	Memory Management.....	82
3.3.1	Architecture Model.....	83
3.3.1.1	Version 4 vs. Version 5.....	83
3.3.1.2	Memory Attributes.....	83
3.3.1.2.1	Page (P) Attribute Bit	83
3.3.1.2.2	Instruction Cache	83
3.3.1.2.3	Data Cache and Write Buffer	83
3.3.1.2.4	Details on Data Cache and Write Buffer Behavior	84
3.3.1.2.5	Memory Operation Ordering	84
3.3.2	Exceptions	85
3.3.3	Interaction of the MMU, Instruction Cache, and Data Cache.....	85
3.3.4	Control	85
3.3.4.1	Invalidate (Flush) Operation.....	85
3.3.4.2	Enabling/Disabling	86
3.3.4.3	Locking Entries	86
3.3.4.4	Round-Robin Replacement Algorithm	88
3.4	Instruction Cache.....	89
3.4.1	Instruction Cache Operation	90
3.4.1.1	Operation When Instruction Cache is Enabled	90
3.4.1.2	Operation When The Instruction Cache Is Disabled.....	90
3.4.1.3	Fetch Policy	90
3.4.1.4	Round-Robin Replacement Algorithm	91
3.4.1.5	Parity Protection.....	91
3.4.1.6	Instruction Cache Coherency.....	92
3.4.2	Instruction Cache Control	92
3.4.2.1	Instruction Cache State at Reset	92
3.4.2.2	Enabling/Disabling	92
3.4.2.3	Invalidating the Instruction Cache.....	93
3.4.2.4	Locking Instructions in the Instruction Cache	93
3.4.2.5	Unlocking Instructions in the Instruction Cache	95
3.5	Branch Target Buffer	95
3.5.1	Branch Target Buffer (BTB) Operation	95
3.5.1.1	Reset.....	96
3.5.2	Update Policy.....	96
3.5.3	BTB Control	96
3.5.3.1	Disabling/Enabling	96
3.5.3.2	Invalidation	97
3.6	Data Cache.....	97
3.6.1	Overviews	97
3.6.1.1	Data Cache Overview	97
3.6.1.2	Mini-Data Cache Overview	98
3.6.1.3	Write Buffer and Fill Buffer Overview.....	99
3.6.2	Data Cache and Mini-Data Cache Operation	100
3.6.2.1	Operation When Caching is Enabled.....	100
3.6.2.2	Operation When Data Caching is Disabled	100
3.6.2.3	Cache Policies	100
3.6.2.3.1	Cacheability	100
3.6.2.3.2	Read Miss Policy	100

3.6.2.3.3	Write Miss Policy.....	101
3.6.2.3.4	Write-Back Versus Write-Through	101
3.6.2.4	Round-Robin Replacement Algorithm	102
3.6.2.5	Parity Protection.....	102
3.6.2.6	Atomic Accesses.....	102
3.6.3	Data Cache and Mini-Data Cache Control	103
3.6.3.1	Data Memory State After Reset	103
3.6.3.2	Enabling/Disabling	103
3.6.3.3	Invalidate and Clean Operations.....	103
3.6.3.3.1	Global Clean and Invalidate Operation	103
3.6.4	Re-configuring the Data Cache as Data RAM	105
3.6.5	Write Buffer/Fill Buffer Operation and Control	106
3.7	Configuration	106
3.8	Performance Monitoring	107
3.8.1	Performance Monitoring Events	107
3.8.1.1	Instruction Cache Efficiency Mode.....	108
3.8.1.2	Data Cache Efficiency Mode.....	109
3.8.1.3	Instruction Fetch Latency Mode.....	109
3.8.1.4	Data/Bus Request Buffer Full Mode	109
3.8.1.5	Stall/Writeback Statistics.....	110
3.8.1.6	Instruction TLB Efficiency Mode	110
3.8.1.7	Data TLB Efficiency Mode	111
3.8.2	Multiple Performance Monitoring Run Statistics	111
3.9	Performance Considerations	111
3.9.1	Interrupt Latency.....	111
3.9.2	Branch Prediction	112
3.9.3	Addressing Modes	112
3.9.4	Instruction Latencies.....	112
3.9.4.1	Performance Terms	113
3.9.4.2	Branch Instruction Timings	114
3.9.4.3	Data Processing Instruction Timings	115
3.9.4.4	Multiply Instruction Timings.....	115
3.9.4.5	Saturated Arithmetic Instructions	117
3.9.4.6	Status Register Access Instructions	117
3.9.4.7	Load/Store Instructions	118
3.9.4.8	Semaphore Instructions	118
3.9.4.9	Coprocessor Instructions	118
3.9.4.10	Miscellaneous Instruction Timing.....	119
3.9.4.11	Thumb Instructions	119
3.10	Test Features.....	119
3.10.1	IXP2800 Network Processor Endianness.....	119
3.10.1.1	Read and Write Transactions Initiated by the Intel XScale® Core	121
3.10.1.1.1	Reads Initiated by the Intel XScale® Core	121
3.10.1.1.2	The Intel XScale® Core Writing to the IXP2800	123
3.11	Intel XScale® Gasket Unit	126
3.11.1	Overview.....	126
3.11.2	Intel XScale® Gasket Functional Description	127
3.11.2.1	Command Memory Bus to Command Push/Pull Conversion	127
3.11.3	CAM Operation	128
3.11.4	Atomic Operations	128
3.11.4.1	Intel XScale® Core Access to SRAM Q-Array.....	130
3.11.5	I/O Transaction	130



3.11.6	Hash Access	131
3.11.7	Gasket Local CSR	131
3.11.8	Interrupt	132
3.12	Intel XScale® Core Peripheral Interface	135
3.12.1	XPI Overview	135
3.12.1.1	Data Transfers	136
3.12.1.2	Data Alignment	136
3.12.1.3	Address Spaces for XPI Internal Devices	137
3.12.2	UART Overview	138
3.12.3	UART Operation	139
3.12.3.1	UART FIFO OPERATION	139
3.12.3.1.1	UART FIFO Interrupt Mode Operation - Receiver Interrupt	139
3.12.3.1.2	FIFO Polled Mode Operation	140
3.12.4	Baud Rate Generator	141
3.12.5	General Purpose I/O (GPIO)	141
3.12.6	Timers	142
3.12.6.1	Timer Operation	142
3.12.7	SlowPort Unit	144
3.12.7.1	PROM Device Support	144
3.12.7.2	µP interface support for the Framer	145
3.12.7.3	SlowPort Unit Interfaces	146
3.12.7.4	Address Space	146
3.12.7.5	SlowPort Interfacing Topology	147
3.12.7.6	SlowPort 8-bit Device Bus Protocols	148
3.12.7.6.1	Mode 0 Single Write Transfer for Fixed-Timed Device	148
3.12.7.6.2	Mode 0 Single Write Transfer for a Self-timing Device	149
3.12.7.6.3	Mode 0 Single Read Transfer for Fixed-timed Device	150
3.12.7.6.4	Single Read Transfer for a Self-timing Device	151
3.12.7.7	SONET/SDH Microprocessor Access Support	151
3.12.7.7.1	Mode 1: 16-bit Microprocessor Interface Support with 16-bit Address Lines	152
3.12.7.7.2	Mode 2: Interface With 8 Data Bits and 11 Address Bits	156
3.12.7.7.3	Mode 3: Support for the Intel and AMCC 2488 Mbps SONET/SDH Microprocessor Interface	158
4	Microengines	167
4.1	Overview	167
4.1.1	Control Store	169
4.1.2	Contexts	169
4.1.3	Datapath Registers	171
4.1.3.1	General-Purpose Registers (GPRs)	171
4.1.3.2	Transfer Registers	171
4.1.3.3	Next Neighbor Registers	172
4.1.3.4	Local Memory	172
4.1.4	Addressing Modes	173
4.1.4.1	Context-Relative Addressing Mode	173
4.1.4.2	Absolute Addressing Mode	174
4.1.4.3	Indexed Addressing Mode	174
4.2	Local CSRs	174
4.3	Execution Datapath	174



4.3.1	Byte Align.....	174
4.3.2	CAM.....	177
4.4	CRC Unit.....	180
4.5	Event Signals.....	180
4.5.1	Microengine "Endianness"	181
4.5.1.1	Read from RBUF (64-bits)	181
4.5.1.2	Write to TBUF	182
4.5.1.3	Read/Write from/to SRAM	182
4.5.1.4	Read/Write from/to DRAM	182
4.5.1.5	Read/Write from/to SHAC and Other CSRs	182
4.5.1.6	Write to Hash Unit.....	183
4.5.2	Media Access	183
4.5.2.1	Read from RBUF	184
4.5.2.2	Write to TBUF	185
4.5.2.3	TBUF to SPI-4 Transfer	186
5	DRAM.....	187
5.1	Overview.....	187
5.2	Size Configuration	188
5.3	DRAM Clocking	189
5.4	Bank Policy	190
5.5	Interleaving	191
5.5.1	Three Channels Active (3-Way Interleave).....	191
5.5.2	Two Channels Active (2-Way Interleave)	193
5.5.3	One Channel Active (No Interleave)	193
5.5.4	Interleaving Across RDRAMs and Banks	194
5.6	Parity and ECC	194
5.6.1	Parity and ECC Disabled	194
5.6.2	Parity Enabled	195
5.6.3	ECC Enabled	195
5.6.4	ECC Calculation and Syndrome	196
5.7	Timing Configuration.....	196
5.8	Microengine Signals	197
5.9	Serial Port.....	197
5.10	RDRAM Controller Block Diagram.....	198
5.10.1	Commands	199
5.10.2	DRAM Write.....	199
5.10.2.1	Masked Write	199
5.10.3	DRAM Read.....	200
5.10.4	CSR Write.....	200
5.10.5	CSR Read.....	200
5.10.6	Arbitration	201
5.10.7	Reference Ordering	201
5.11	DRAM Push/Pull Arbiter	201
5.11.1	Arbiter Push/Pull Operation	202
5.11.2	DRAM Push Arbiter Description	203
5.12	DRAM Pull Arbiter Description.....	204



6	SRAM Interface	207
6.1	Overview	207
6.2	SRAM Interface Configurations	208
6.3	SRAM Interface Configurations	209
6.3.1	Internal Interface	209
6.3.2	Number of Channels	209
6.3.3	Coprocessor and/or SRAMs Attached to a Channel	209
6.4	SRAM Controller Configurations	209
6.5	Command Overview	211
6.5.1	Basic Read/Write Commands	211
6.5.2	Atomic Operations	212
6.5.3	Queue Data Structure Commands	213
6.5.3.1	Read_Q_Descriptor Commands	217
6.5.3.2	Write_Q_Descriptor Commands	217
6.5.3.3	ENQ and DEQ Commands	217
6.5.4	Ring Data Structure Commands	217
6.5.5	Journaling Commands	218
6.5.6	CSR Accesses	218
6.6	Parity	218
6.7	Address Map	219
6.8	Reference Ordering	220
6.8.1	Reference Order Tables	220
6.8.2	Microcode Restrictions to Maintain Ordering	221
6.9	Coprocessor Mode	222
7	SHaC—Unit Expansion	225
7.1	Overview	225
7.1.1	SHaC Unit Block Diagram	225
7.1.2	Scratchpad	227
7.1.2.1	Scratchpad Description	227
7.1.2.2	Scratchpad Interface	229
7.1.2.2.1	Command Interface	229
7.1.2.2.2	Push/Pull Interface	229
7.1.2.2.3	CSR Bus Interface	229
7.1.2.2.4	Advanced Peripherals Bus Interface (APB)	229
7.1.2.3	Scratchpad Block Level Diagram	229
7.1.2.3.1	Scratchpad Commands	230
7.1.2.3.2	Ring Commands	231
7.1.2.3.3	Clocks and Reset	235
7.1.2.3.4	Reset Registers	235
7.1.3	Hash Unit	236
7.1.3.1	Hashing Operation	237
7.1.3.2	Hash Algorithm	239
8	Media and Switch Fabric Interface	241
8.1	Overview	241
8.1.1	SPI-4	243
8.1.2	CSIX	246



8.1.3	CSIX/SPI-4 Interleave Mode.....	246
8.2	Receive.....	247
8.2.1	Receive Pins.....	248
8.2.2	RBUF	248
8.2.2.1	SPI-4.....	250
8.2.2.2	CSIX.....	252
8.2.3	Full Element List	255
8.2.4	Rx_Thread_Freelist_#	255
8.2.5	Rx_Thread_Freelist_Timeout_#	256
8.2.6	Receive Operation Summary.....	256
8.2.7	Receive Flow Control Status	258
8.2.7.1	SPI-4.....	258
8.2.7.2	CSIX.....	259
8.2.7.2.1	Link-level.....	259
8.2.7.2.2	Virtual Output Queue	260
8.2.8	Parity.....	260
8.2.8.1	SPI-4.....	260
8.2.8.2	CSIX.....	261
8.2.8.2.1	Horizontal Parity.....	261
8.2.8.2.2	Vertical Parity.....	261
8.2.9	Error Cases.....	261
8.3	Transmit.....	262
8.3.1	Transmit Pins.....	263
8.3.2	TBUF	263
8.3.2.1	SPI-4.....	266
8.3.2.2	CSIX.....	267
8.3.3	Transmit Operation Summary.....	268
8.3.3.1	SPI-4.....	269
8.3.3.2	CSIX.....	270
8.3.3.3	Transmit Summary.....	271
8.3.4	Transmit Flow Control Status	271
8.3.4.1	SPI-4.....	271
8.3.4.2	CSIX.....	274
8.3.4.2.1	Link-level.....	274
8.3.4.2.2	Virtual Output Queue	274
8.3.5	Parity.....	274
8.3.5.1	SPI-4.....	274
8.3.5.2	CSIX.....	275
8.3.5.2.1	Horizontal Parity.....	275
8.3.5.2.2	Vertical Parity.....	275
8.4	RBUF and TBUF Summary	275
8.5	CSIX Flow Control Interface	276
8.5.1	TXCSRB, RXCSRB	276
8.5.2	FCIFIFO, FCEFIFO	278
8.5.2.1	Full Duplex CSIX.....	278
8.5.2.2	Simplex CSIX.....	280
8.5.3	TXCDAT/RXCDAT, TXCSOF/RXCSOF, TXCPAR/RXCPAR, and TXCFC/RXCFC	281
8.6	Deskew and Training.....	282
8.6.1	Data Training Pattern.....	283
8.6.2	Flow Control Training Pattern	284

8.6.3	Use of Dynamic Training	285
8.7	CSIX Startup Sequence.....	289
8.7.1	CSIX Full Duplex	289
8.7.1.1	Ingress IXP2800	289
8.7.1.2	Egress IXP2800	289
8.7.1.3	Single IXP2800	290
8.7.2	CSIX Simplex.....	290
8.7.2.1	Ingress IXP2800	290
8.7.2.2	Egress IXP2800	290
8.7.2.3	Single IXP2800	291
8.8	Interface to Command and Push and Pull Busses	291
8.8.1	RBUF or MSF CSR to Microengine S Transfer In Register for instruction:	293
8.8.2	Microengine S Transfer Out Register to TBUF or MSF CSR for instruction:	293
8.8.3	Microengine to MSF CSR for instruction:.....	293
8.8.4	From RBUF to DRAM for instruction:.....	293
8.8.5	From DRAM to TBUF for instruction:	294
8.9	Receiver and Transmitter Interoperation with Framers and Switch Fabrics	294
8.9.1	Receiver and Transmitter Configurations	295
8.9.1.1	Simplex Configuration	295
8.9.1.2	Hybrid Simplex Configuration	296
8.9.1.3	Dual NPU Full Duplex Configuration.....	297
8.9.1.4	Single NPU Full Duplex Configuration (SPI-4.2).....	298
8.9.1.5	Single NPU, Full Duplex Configuration (SPI-4.2 and CSIX-L1)	299
8.9.2	System Configurations.....	300
8.9.2.1	Framer, Single NPU Ingress and Egress, and Fabric Interface Chip ..	300
8.9.2.2	Framer, Dual NPU Ingress, Single NPU Egress, and Fabric Interface Chip.....	301
8.9.2.3	Framer, Single NPU Ingress and Egress, and CSIX-L1 Chips for Translation and Fabric Interface	301
8.9.2.4	CPU Complex, NPU, and Fabric Interface Chip	302
8.9.2.5	Framer, Single NPU, Co-Processor, and Fabric Interface Chip	303
8.9.3	SPI-4.2 Support	304
8.9.3.1	SPI-4.2 Receiver	304
8.9.3.2	SPI-4.2 Transmitter	305
8.9.4	CSIX-L1 Protocol Support	306
8.9.4.1	CSIX-L1 Interface Reference Model: Traffic Manager and Fabric Interface Chip.....	306
8.9.4.2	Intel® IXP2800 Support of the CSIX-L1 Protocol	307
8.9.4.2.1	Mapping to 16-Bit Wide DDR LVDS	307
8.9.4.2.2	Support for Dual Chip, Full-Duplex Operation	308
8.9.4.2.3	Support for Simplex Operation.....	309
8.9.4.2.4	Support for Hybrid Simplex Operation	310
8.9.4.2.5	Support for Dynamic De-Skew Training.....	311
8.9.4.3	CSIX-L1 Protocol Receiver Support	312
8.9.4.4	CSIX-L1 Protocol Transmitter Support	313
8.9.4.5	Implementation of a Bridge Chip to CSIX-L1	314
8.9.5	Dual Protocol (SPI and CSIX-L1) Support.....	315
8.9.5.1	Dual Protocol Receiver Support.....	315
8.9.5.2	Dual Protocol Transmitter Support.....	315
8.9.5.3	Implementation of a Bridge Chip to CSIX-L1 and SPI-4.2	315
8.9.6	Transmit State Machine	316



8.9.6.1	SPI-4.2 Transmitter State Machine.....	317
8.9.6.2	Training Transmitter State Machine.....	318
8.9.6.3	CSIX-L1 Transmitter State Machine	318
8.9.7	Dynamic De-Skew	319
8.9.8	Summary of Receiver and Transmitter Signals	320
9	PCI Unit.....	321
9.1	Overview.....	321
9.2	PCI Pin Protocol Interface Block.....	323
9.2.1	PCI Commands	324
9.2.2	IXP2800 Network Processor Initialization.....	325
9.2.2.1	Initialization by the Intel XScale® Core.....	325
9.2.2.2	Initialization by a PCI Host.....	326
9.2.3	PCI Type 0 Configuration Cycles.....	326
9.2.3.1	Configuration Write	327
9.2.3.2	Configuration Read.....	327
9.2.4	PCI 64-Bit Bus Extension	327
9.2.5	PCI Target Cycles.....	328
9.2.5.1	PCI Accesses to CSR	328
9.2.5.2	PCI Accesses to DRAM	328
9.2.5.3	PCI Accesses to SRAM	328
9.2.5.4	Target Write Accesses From PCI Bus	328
9.2.5.5	Target Read Accesses From PCI Bus	329
9.2.6	PCI Initiator Transactions	329
9.2.6.1	PCI Request Operation.....	330
9.2.6.2	PCI Commands.....	330
9.2.6.3	Initiator Write Transactions	330
9.2.6.4	Initiator Read Transactions	330
9.2.6.5	Initiator Latency Timer	331
9.2.6.6	Special Cycle	331
9.2.7	PCI Fast Back to Back Cycles	331
9.2.8	PCI Retry	331
9.2.9	PCI Disconnect.....	331
9.2.10	PCI Built In System Test.....	332
9.2.11	PCI Central Functions.....	332
9.2.11.1	PCI Interrupt Inputs.....	332
9.2.11.2	PCI Reset Output.....	333
9.2.11.3	PCI Internal Arbiter	333
9.3	Slave Interface Block.....	334
9.3.1	CSR Interface	334
9.3.2	SRAM Interface	335
9.3.2.1	SRAM Slave Writes	335
9.3.2.2	SRAM Slave Reads	336
9.3.3	DRAM Interface	336
9.3.3.1	DRAM Slave Writes	336
9.3.3.2	DRAM Slave Reads	338
9.3.4	Mailbox and Doorbell Registers.....	339
9.3.5	PCI Interrupt Pin	341
9.4	Master Interface Block.....	342
9.4.1	DMA Interface.....	342
9.4.1.1	Allocation of the DMA Channels	342
9.4.1.2	Special Registers for Microengine Channels	343

9.4.1.3	DMA Descriptor.....	343
9.4.1.4	DMA Channel Operation.....	345
9.4.1.5	DMA Channel End Operation	346
9.4.1.6	Adding Descriptor to an Unterminated Chain	346
9.4.1.7	DRAM to PCI Transfer	346
9.4.1.8	PCI to DRAM Transfer	347
9.4.2	Push/Pull Command Bus Target Interface.....	347
9.4.2.1	Command Bus Master Access to Local Configuration Registers	347
9.4.2.2	Command Bus Master Access to Local Control and Status Registers.....	348
9.4.2.3	Command Bus Master Direct Access to PCI Bus	348
9.4.2.3.1	PCI Address Generation for IO and MEM cycles.....	348
9.4.2.3.2	PCI Address Generation for Configuration Cycles.....	349
9.4.2.3.3	PCI Address Generation for Special and IACK Cycles.....	349
9.4.2.3.4	PCI Enables	349
9.4.2.3.5	PCI Command	349
9.5	PCI Unit Error Behavior	350
9.5.1	PCI Target Error Behavior	350
9.5.1.1	Target Access Has an Address Parity Error	350
9.5.1.2	Initiator Asserts PCI_PERR# in Response to One of Our Data Phases	350
9.5.1.3	Discard Timer Expires on a Target Read.....	350
9.5.1.4	Target Access to the PCI_CSR_BAR Space Has Illegal Byte Enables.....	350
9.5.1.5	Target Write Access Receives Bad Parity PCI_PAR with the Data.....	350
9.5.1.6	SRAM Responds With a Memory Error on One or More Data Phases on a Target Read	351
9.5.1.7	DRAM Responds With a Memory Error on One or More Data Phases on a Target Read	351
9.5.2	As a PCI Initiator During a DMA Transfer	351
9.5.2.1	DMA Read From DRAM (Memory-to-PCI Transaction) Gets a Memory Error	351
9.5.2.2	DMA Read From SRAM (Descriptor Read) Gets a Memory Error.....	351
9.5.2.3	DMA From DRAM Transfer (Write to PCI) Receives PCI_PERR# on PCI Bus.....	352
9.5.2.4	DMA To DRAM (Read from PCI) Has Bad Data Parity	352
9.5.2.5	DMA Transfer Experiences a Master Abort (Time-Out) on PCI.....	352
9.5.2.6	DMA Transfer Receives a Target Abort Response During a Data Phase	353
9.5.2.7	DMA Descriptor Has a 0x0 Word Count (Not an Error)	353
9.5.3	As a PCI Initiator During a Direct Access from the Intel XScale® Core or Microengine	353
9.5.3.1	Master Transfer Experiences a Master Abort (Time-Out) on PCI.....	353
9.5.3.2	Master Transfer Receives a Target Abort Response During a Data Phase	353
9.5.3.3	Master from the Intel XScale® Core or Microengine Transfer (Write to PCI) Receives PCI_PERR# on PCI Bus	353
9.5.3.4	Master Read From PCI (Read from PCI) Has Bad Data Parity	354
9.5.3.5	Master Transfer Receives PCI_SERR# from the PCI Bus	354
9.5.3.6	Intel XScale® Core Microengine Requests Direct Transfer when the PCI Bus is in Reset	354
9.6	PCI Data Byte Lane Alignment	354
9.6.1	Endian for Byte Enable	357



10	Clocks, Reset, and Initialization	361
10.1	Clocks	361
10.2	Synchronization Between Frequency Domains	365
10.3	Reset	366
10.3.1	Hardware Reset Using nRESET or PCI_RST#	366
10.3.2	PCI Initiated Reset	368
10.3.3	Watchdog Timer Initiated Reset	368
10.3.3.1	Slave IXP (Non-Central Function)	369
10.3.3.2	Master IXP (PCI Host, Central Function)	369
10.3.3.3	Master IXP (Central Function)	369
10.3.4	Software Initiated Reset	369
10.3.5	Reset Removal operation based on CFG_PROM_BOOT	370
10.3.5.1	When CFG_PROM_BOOT is 1 (BOOT_PROM is Present)	370
10.3.5.2	When CFG_PROM_BOOT is 0 (BOOT_PROM is Not Present)	370
10.3.6	Strap Pins	370
10.3.7	Powerup Reset Sequence	372
10.4	Boot Mode	372
10.4.1	Flash ROM	374
10.4.2	PCI Host Download	374
10.5	Initialization	375



Figures

1	IXP2800 Network Processor Functional Block Diagram.....	24
2	IXP2800 Network Processor Detailed Diagram.....	25
3	Intel XScale® 4GB (32-bit) Address Space	28
4	Microengine Block Diagram	30
5	Context State Transition Diagram.....	32
6	Byte Align Block Diagram	40
7	CAM Block Diagram	43
8	Echo Clock Configuration	49
9	Logical View of Rings	55
10	Example System Block Diagram	57
11	Full-Duplex Block Diagram	58
12	Simplified MSF Receive Section Block Diagram	59
13	Simplified Transmit Section Block Diagram.....	64
14	Hash Unit Block Diagram.....	70
15	DMA Descriptor Reads	72
16	Intel XScale® Core Architecture Features	80
17	Example of Locked Entries in TLB.....	88
18	Instruction Cache Organization	89
19	Locked Line Effect on Round Robin Replacement.....	94
20	BTB Entry	95
21	Branch History	96
22	Data Cache Organization	98
23	Mini-Data Cache Organization.....	99
24	Byte Steering for Read and Byte Enable Generation by the Intel XScale® Core	122
25	Intel XScale® Core Initiated Write to the IXP2800 Network Processor.....	124
26	Intel XScale® Core Initiated Write to the IXP2800 Network Processor (Continued).....	125
27	Global Buses Connection to the Intel XScale® Gasket.....	127
28	Flow Through the Intel XScale® Core Interrupt Controller	133
29	Interrupt Mask Block Diagram	134
30	XPI Interfaces (B0) for 2400/2800	136
31	Example UART Data Frame	139
32	GPIO Functional Diagram.....	141
33	Timer Control Unit Interfacing Diagram	142
34	Timer Internal Logic Diagram	143
35	SlowPort Unit Interface Diagram	146
36	An Example of Address Space Hole Diagram	146
37	SlowPort Example Application Topology	147
38	Mode 0 Single Write Transfer for a Fixed-Timed Device.....	148
39	Mode 0 Single Write Transfer for a Self-timing Device.....	149
40	Mode 0 Single Read Transfer for a Fixed-timed Device.....	150
41	Mode 0 Single Read Transfer for a Self-timing Device.....	151
42	An Interface Topology with Lucent TDAT042G5 SONET/SDH	153
43	Mode 1 Single Write Transfer for Lucent TDAT042G5 Device (B0)	154
44	Mode 1 Single Read Transfer for Lucent TDAT042G5 Device (B0).....	155
45	An Interface Topology with PMC-Sierra PM5351 S/UNI-TETRA	156
46	Mode 2 Single Write Transfer for PMC-Sierra PM5351 Device (B0).....	157
47	Mode 2 Single Read Transfer for PMC-Sierra PM5351 Device (B0).....	158



48	An Interface Topology with Intel / AMCC SONET/SDH Device.....	159
49	Mode 3 Second Interface Topology with Intel / AMCC SONET/SDH Device.....	160
50	Mode 3 Single Write Transfer Followed by Read (B0)	161
51	Mode 3 Single Read Transfer Followed by Write (B0)	162
52	An Interface Topology with Intel / AMCC SONET/SDH Device in Motorola Mode	163
53	Second Interface Topology with Intel / AMCC SONET/SDH Device	164
54	Mode 4 Single Write Transfer (B0)	165
55	Mode 4 Single Read Transfer (B0)	166
56	Microengine Block Diagram.....	168
57	Context State Transition Diagram	170
58	Byte Align Block Diagram	175
59	CAM Block Diagram	178
60	Read from RBUF (64-bits).....	181
61	Write to TBUF (64-bits).....	182
62	48-bit, 64-bit, and 128-bit Hash Operand Transfers	183
63	Bit, Byte and Long-Word Organization in One RBUF Element	184
64	Write to TBUF.....	185
65	MSF Interface	186
66	Clock Configuration	189
67	IXP2800 Clocking for RDRAM at 400 MHz	190
68	IXP2800 Clocking for RDRAM at 508 MHz	190
69	Address Mapping Flow	191
70	RDRAM Controller Block Diagram	198
71	DRAM Push/Pull Arbiter Functional Blocks	202
72	DRAM Push Arbiter Functional Blocks	204
73	DRAM Pull Arbiter Functional Blocks	205
74	SRAM Controller/Chassis Block Diagram	208
75	SRAM Clock Connection on a Channel.....	210
76	External Pipeline Registers Block Diagram	211
77	Queue Descriptor with Four Links	214
78	Enqueueing One Buffer at a Time	214
79	Previously Linked String of Buffers.....	214
80	First Step to Enqueue a String of Buffers to a Queue (ENQ_Tail_and_Link).....	215
81	Second Step to Enqueue a String of Buffers to a Queue (ENQ_Tail)	215
82	Connection to a Coprocessor Though Standard QDR Interface	222
83	Coprocessor with Memory Mapped FIFO Ports	223
84	SHaC Top Level Diagram.....	226
85	Scratchpad Block Diagram	228
86	Ring Communication Logic Diagram	231
87	Hash Unit Block Diagram	236
88	Example System Block Diagram	242
89	Full-Duplex Block Diagram	243
90	Receive and Transmit Clock Generation.....	245
91	Simplified Receive Section Block Diagram.....	247
92	RBUF Element State Diagram.....	257
93	DIP-4 Codes' Extent	260
94	Simplified of Transmit Section Block Diagram.....	262
95	TBUF State Diagram	271
96	Tx Calendar Block Diagram.....	272
97	CSIX Flow Control Interface — TXCSRB and RXCSRB.....	277



98	CSIX Flow Control Interface — FCIFIFO and FCEFIFO in Full Duplex Mode	279
99	CSIX Flow Control Interface — FCIFIFO and FCEFIFO in Simplex Mode.....	280
100	MSF to Command and Push and Pull Busses Interface Block Diagram	292
101	Basic I/O Capability of the Intel® IXP2800	294
102	Simplex Configuration.....	295
103	Hybrid Simplex Configuration	296
104	Dual NPU, Full Duplex Configuration	297
105	Single NPU, Full Duplex Configuration (SPI-4.2 Protocol)	298
106	Single NPU, Full Duplex Configuration (SPI-4.2 and CSIX-L1 Protocols).....	299
107	Framer, Single NPU Ingress, Single NPU Egress, and Fabric Interface Chip.....	300
108	Framer, Dual NPU Ingress, Single NPU Egress, and Fabric Interface Chip	301
109	Framer, Single NPU Ingress, Single NPU Egress, CSIX-L1 Translation Chip and CSIX-L1 Fabric Interface Chip	301
110	CPU Complex, NPU, and Fabric Interface Chips	302
111	Framer, Single NPU, Co-Processor, and Fabric Interface Chip	303
112	SPI-4.2 Interface Reference Model with Receiver and Transmitter Labels Corresponding to Link Layer Device Functions	304
113	CSIX-L1 Interface Reference Model with Receiver and Transmitter Labels Corresponding to Fabric Interface Chip Functions	306
114	Reference Model for Intel® IXP2800 Support of the Simplex Configuration Using Independent Ingress and Egress Interfaces	309
115	Reference Model for Hybrid Simplex Operation	310
116	Block Diagram of Dual Protocol (SPI-4.2 and CSIX-L1) Bridge Chip	316
117	Summary of Receiver and Transmitter Signaling	320
118	PCI Functional Blocks.....	322
119	Data Access Paths	323
120	PCI Arbiter Configuration Using CFG_PCI_ARB(GPIO[2])	333
121	Example of Target Write to SRAM of 68 Bytes.....	335
122	Example of Target Write to DRAM of 68 Bytes	337
123	Example of Target Read from DRAM Using 64-Byte Burst	338
124	Generation of the Doorbell Interrupts to PCI	340
125	Generation of the Doorbell Interrupts to the Intel XScale® Core.....	340
126	PCI Interrupts.....	341
127	DMA Descriptor Reads	344
128	PCI Address Generation for Command Bus Master to PCI.....	348
129	PCI Address Generation for Command Bus Master to PCI Configuration Cycle	349
130	Overall Clock Generation and Distribution.....	362
131	IXP2800 Network Processor Clock Generation.....	365
132	Synchronization Between Frequency Domains	366
133	Reset Out Behavior	367
134	Reset Generation.....	368
135	Boot Process	373

Tables

1	Data Terminology	22
2	IXP2800 Network Processor Microengine Bus Arrangement.....	31
3	Next Neighbor Write as a Function of CTX_ENABLE[NN_MODE]	35
4	Registers Used By Contexts in Context-Relative Addressing Mode	38
5	Align Value and Shift Amount	40
6	Register Contents for Example 10.....	41
7	Register Contents for Example 11.....	41
8	Algorithm for Debug Software to Find out the Contents of the CAM	45
9	RDRAM Sizes.....	47
10	SRAM Controller Configurations	50
11	Total Memory per Channel	50
12	Address Reference Order.....	52
13	Q_array Entry Reference Order	53
14	Ring Full Signal Use -- Number of Contexts and Length vs Ring Size.....	56
15	RBUF SPI-4 Status Definition.....	61
16	RBUF CSIX Status Definition	62
17	TBUF SPI-4 Control Definition.....	66
18	TBUF CSIX Control Definition	66
19	DMA Descriptor Format.....	73
20	Doorbell Interrupt Registers.....	75
21	I/O Latency	78
22	Data Cache and Buffer Behavior when X = 0.....	83
23	Data Cache and Buffer Behavior when X = 1	84
24	Memory Operations that Impose a Fence	85
25	Valid MMU & Data/mini-data Cache Combinations.....	85
26	Performance Monitoring Events	107
27	Some Common Uses of the PMU	108
28	Branch Latency Penalty.....	112
29	Latency Example	114
30	Branch Instruction Timings (Those predicted by the BTB)	114
31	Branch Instruction Timings (Those not predicted by the BTB)	114
32	Data Processing Instruction Timings	115
33	Multiply Instruction Timings	115
35	Implicit Accumulator Access Instruction Timings.....	117
36	Saturated Data Processing Instruction Timings	117
37	Status Register Access Instruction Timings	117
34	Multiply Implicit Accumulate Instruction Timings	117
38	Load and Store Instruction Timings.....	118
39	Load and Store Multiple Instruction Timings	118
40	Semaphore Instruction Timings.....	118
41	CP15 Register Access Instruction Timings	118
42	CP14 Register Access Instruction Timings	119
43	SWI Instruction Timings.....	119
44	Count Leading Zeros Instruction Timings.....	119
45	Little Endian Encoding.....	120
46	Big Endian Encoding	120
47	Byte Enable Generation by the Intel XScale® Core for Byte Transfers in Little and Big Endian Systems	121



48	Byte Enable Generation by the Intel XScale® Core for 16-bit Data Transfers in Little and Big Endian Systems.....	123
49	Byte Enable Generation by the Intel XScale® Core for Byte Writes in Little and Big Endian Systems.....	123
50	Byte Enable Generation by the Intel XScale® Core for Word Writes in Little and Big-Endian Systems	124
51	CMB Write Command to CPP Command Conversion.....	128
52	IXP2800 Network Processor SRAM Q-Array Access Alias Addresses	130
53	GCSR Address Map(0xd700 0000)	132
54	Data Transaction Alignment	137
55	Address spaces for XPI Internal Devices	137
56	8-bit Flash Memory Device Density	144
57	SONET/SDH Devices	145
58	Next Neighbor Write as a Function of CTX_Enable[NN_Mode]	172
59	Registers Used By Contexts in Context-Relative Addressing Mode	173
60	Align Value and Shift Amount	175
61	Register Contents for Example 23.....	175
62	Register Contents for Example 24.....	176
63	RDRAM Loading.....	188
64	RDRAM Sizes.....	188
65	Address Rearrangement for 3-Way Interleave	192
66	Address Rearrangement for 3-Way Interleave (Rev B)	193
67	Address Bank Interleaving.....	194
68	RDRAM Timing Parameter Settings	196
69	Ordering of Reads and Writes to the Same Address for DRAM.....	201
70	DRAM Push Arbiter Operation.....	203
71	DPLA Description	204
72	SRAM Controller Configurations.....	209
73	Total Memory per Channel	210
74	Atomic Operations	212
75	Queue Format.....	216
76	Ring/Journal Format	216
77	Ring Size Encoding	217
78	Address Map.....	219
79	Address Reference Order.....	220
80	Q_array Entry Reference Order.....	221
81	Ring Full Signal Use -- Number of Contexts and Length Versus Ring Size	232
82	Head/Tail, Base and Full by Ring Size	233
83	Intel XScale® Core and Microengine Instructions	235
84	S Transfer Registers Hash Operands.....	237
85	SPI-4 Control Word Format	244
86	Order of Bytes within the SPI-4 Data Burst	245
87	CFrame Types.....	246
88	Receive Pins Usage by Protocol	248
89	Order in Which Received Data Is Stored in RBUF	248
90	Mapping of Received Data to RBUF Partitions.....	249
91	Number of Elements per RBUF Partition.....	249
92	RBUF SPIF-4 Status Definition.....	252
93	RBUF CSIX Status Definition	254
94	Rx_Thread_Freelist Use.....	255

95	Summary of SPI-4 and CSIX RBUF Operations	258
96	Transmit Pins Usage by Protocol	263
97	Order in Which Data is Transmitted from TBUF	263
98	Mapping of TBUF Partitions to Transmit Protocol	264
99	Number of Elements per TBUF Partition	264
100	TBUF SPI-4 Control Definition	266
101	TBUF CSIX Control Definition	268
102	Transmit SPI-4 Control Word	269
103	Transmit CSIX Header	270
104	Summary of RBUF and TBUF Operations	275
105	SRB Definition by Clock Phase Number	277
106	Data Deskew Functions.....	282
107	Calendar Deskew Functions.....	283
108	Flow Control Deskew Functions.....	283
109	Data Training Sequence.....	284
110	Flow Control Training Sequence	284
111	Calendar Training Sequence	285
112	IXP2800 Network Processor Requires Data Training	286
113	Switch Fabric or SPI-4 Framer Requires Data Training	287
114	IXP2800 Network Processor Requires Flow Control Training.....	288
115	Switch Fabric Requires Flow Control Training	288
116	SPI-4.2 Transmitter State Machine Transitions on 16-Bit Bus Transfers	317
117	Training Transmitter State Machine Transitions on 16-Bit Bus Transfers	318
118	CSIX-L1 Transmitter State Machine Transitions on CWord Boundaries.....	318
119	PCI Block FIFO Sizes	324
120	Maximum Loading	324
121	PCI Commands	324
122	PCI BAR Programmable Sizes.....	326
123	PCI BAR Sizes with PCI host Initialization	326
124	Legal Combinations of the Strap Pin Options.....	332
125	Slave Interface Buffer Sizes	334
126	Doorbell Interrupt Registers.....	339
127	IRQ Interrupt Options by Stepping	341
128	DMA Descriptor Format.....	344
129	PCI Maximum Burst Size.....	347
130	Command Bus Master Configuration Transactions.....	349
131	Command Bus Master Address Space Map to PCI	349
132	Byte Lane Alignment for 64-Bit PCI Data In (64 Bits PCI Little Endian to Big Endian with Swap)	354
135	Byte Lane Alignment for 32-bit PCI Data In (32 Bits PCI Big Endian to Big Endian without Swap)	355
136	Byte Lane Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Little Endian with Swap)	355
133	Byte Lane Alignment for 64-bit PCI Data In (64 Bits PCI Big Endian to Big Endian without Swap)	355
134	Byte Lane Alignment for 32-bit PCI Data In (32 Bits PCI Little Endian to Big Endian with Swap)	355
137	Byte Lane Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Big Endian without Swap)	356
138	Byte Lane Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Little Endian with Swap)	356



139	Byte Lane Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Big Endian without Swap)	356
140	Byte Enable Alignment for 64-bit PCI Data In (64 Bits PCI Little Endian to Big Endian with Swap)	357
141	Byte Enable Alignment for 64-bit PCI Data In (64 Bits PCI Big Endian to Big Endian without Swap)	357
142	Byte Enable Alignment for 32-bit PCI Data In (32 bits PCI Little Endian to Big Endian with Swap)	357
143	Byte Enable Alignment for 32-bit PCI Data In (32 Bits PCI Big Endian to Big Endian without Swap)	358
144	Byte Enable Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Little Endian with Swap)	358
145	Byte Enable Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Big Endian without Swap)	358
146	Byte Enable Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Little Endian with Swap)	358
147	Byte Enable Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Big Endian without Swap)	359
148	PCI I/O Cycles with Data Swap Enable	360
149	Clock Usage Summary	362
150	Clock Rates Examples	364
151	IXP2800 Network Processor Strap Pins	371
152	Supported Strap Combinations	372



intel® Introduction

1

1.1 About this Document

This document is the hardware reference manual for the Intel® IXP2800 Network Processor. This information is intended for use by developers and is organized as follows:

Section 2, “Technical Description” contains a hardware overview.

Section 3, “Intel XScale® Core” describes the embedded Intel XScale® core.

Section 4, “Microengines” describes Microengine operation.

Section 5, “DRAM” describes the DRAM Unit.

Section 6, “SRAM Interface” describes the SRAM Unit.

Section 7, “SHaC—Unit Expansion” describes the Scratchpad, Hash Unit, and CSRs (SHaC).

Section 8, “Media and Switch Fabric Interface” describes the Media and Switch Fabric (MSF) Interface used to connect the network processor to a physical layer device.

Section 9, “PCI Unit” describes the PCI Unit.

Section 10, “Clocks, Reset, and Initialization” describes the clocks, reset and initialization sequence.

1.2 Related Documentation

Further information on the IXP2800 is available in the following documents:

IXP2800 Network Processor Datasheet - Contains summary information on the IXP2800 Network Processor including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

IXP2400/IXP2800 Network Processor Programmer's Reference Manual - Contains detailed programming information for designers.

IXP2400/IXP2800 Network Processor Development Tools User's Guide - Describes the Workbench and the development tools you can access through the use of the Workbench.



1.3 Conventions

Table 1 lists the terminology used in this manual.

Table 1. Data Terminology

Term	Words	Bytes	Bits
Byte	$\frac{1}{2}$	1	8
Word	1	2	16
Longword	2	4	32
Quadword	4	8	64



Technical Description

2

2.1 Overview

This section provides a brief overview of the IXP2800 Network Processor internal hardware. This section is intended as an overall hardware introduction to the network processor.

The major blocks are:

- Intel XScale® core— General purpose 32-bit RISC processor (ARM® Version 5 Architecture compliant) used to initialize and manage the network processor, and can be used for higher layer network processing tasks.
- Intel XScale® technology Peripherals (XPI)—Interrupt Controller, Timers, UART, General Purpose I/O (GPIO) and interface to low-speed off chip peripherals (such as maintenance port of network devices) and Flash ROM.
- Microengines (MEs) —Sixteen 32-bit programmable engines specialized for Network Processing. Microengines do the main data plane processing per packet.
- DRAM Controllers— Three independent controllers for Rambus® DRAM. Typically DRAM is used for data buffer storage.
- SRAM Controllers —Four independent controllers for QDR SRAM. Typically SRAM is used for control information storage.
- Scratchpad Memory—16 KBytes storage for general purpose use.
- Hash Unit—Polynomial hash accelerator. The Intel XScale® core and Microengines can use it to offload hash calculations.
- Control and Status Register Access Proxy— CAP. These provide special inter-processor communication features to allow flexible and efficient inter-Microengine and Microengine to Intel XScale® core communication.
- Media and Switch Fabric Interface (MSF)—Interface for network framers and/or Switch Fabric. Contains receive and transmit buffers.
- PCI Controller—64-bit PCI Rev 2.2 compliant I/O bus. PCI can be used to either connect to a Host processor, or to attach PCI compliant peripheral devices.
- Performance Monitor—Counters which can be programmed to count selected internal chip hardware events, which can be used to analyze and tune performance.

Figure 1 is a simple block diagram of the network processor showing the major internal hardware blocks. Figure 2 is a detailed diagram of the network processor units and busses.

Figure 1. IXP2800 Network Processor Functional Block Diagram

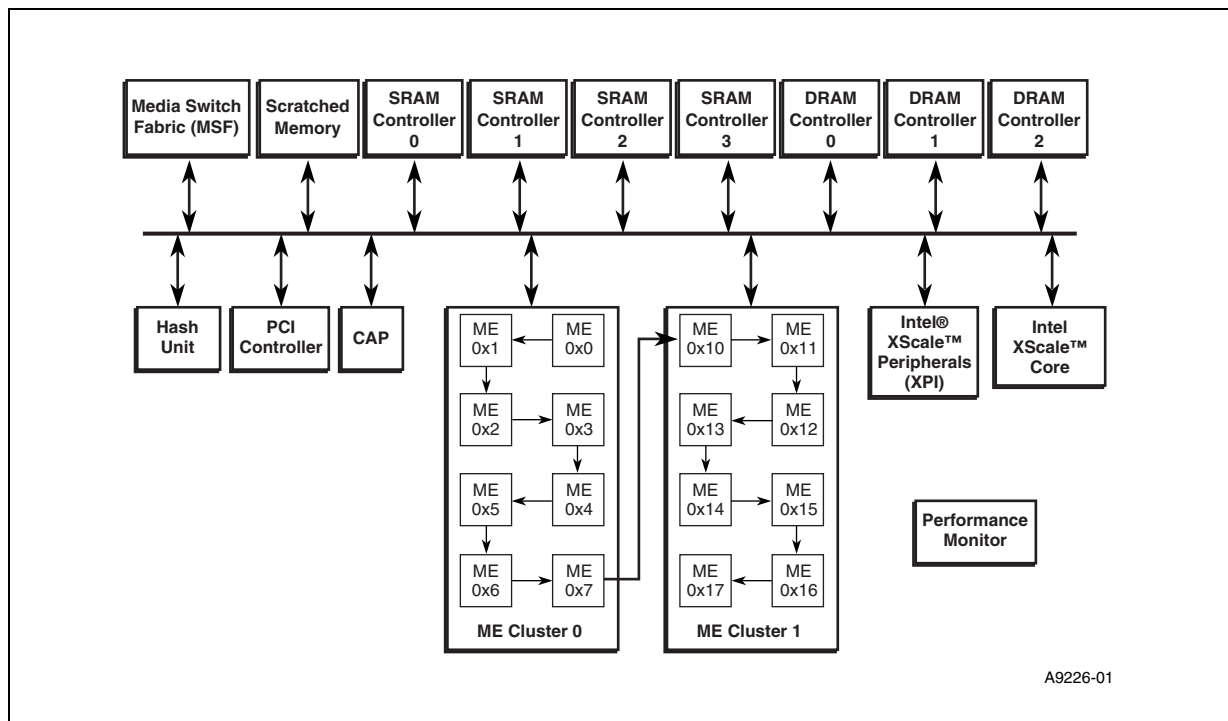
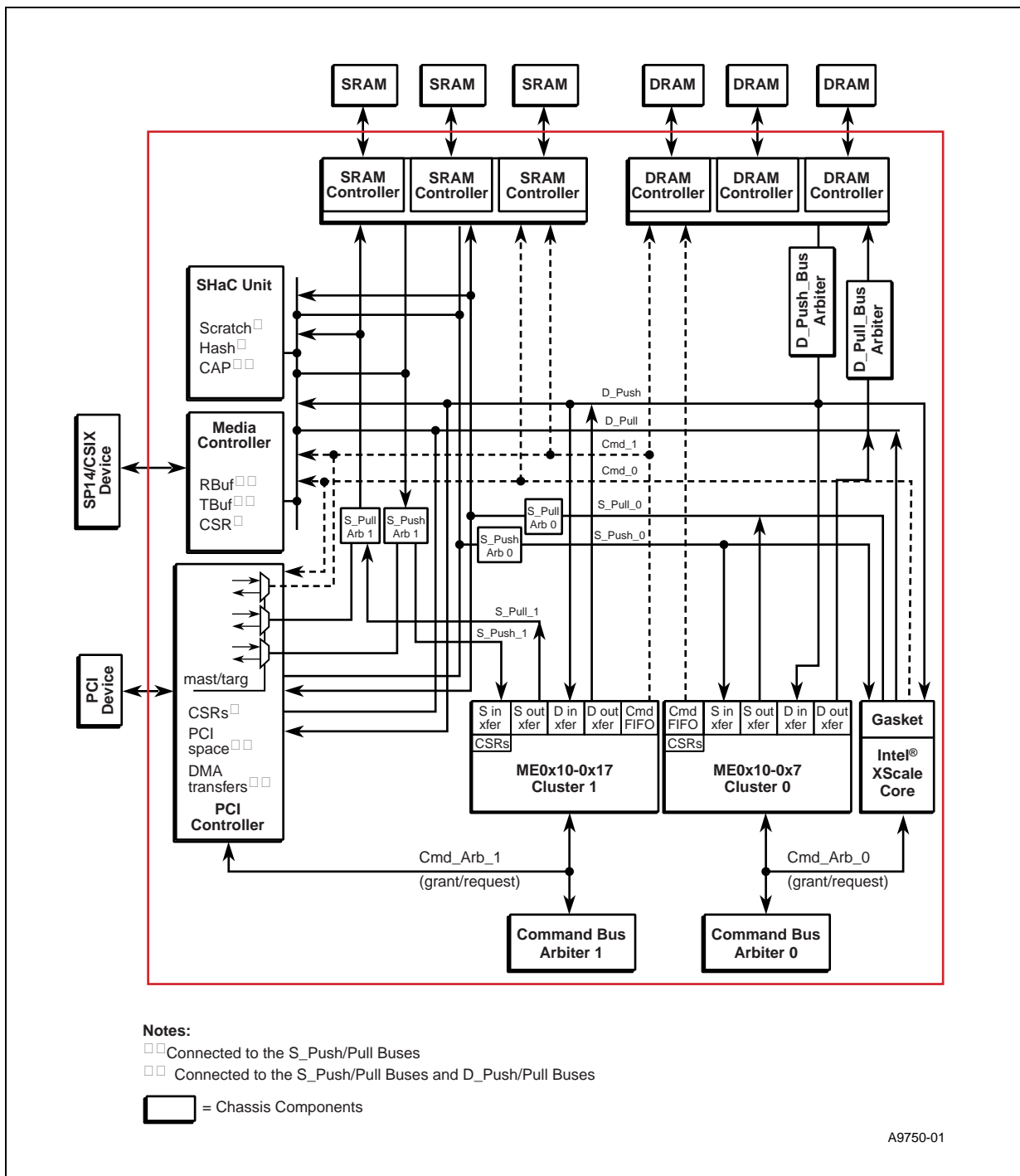


Figure 2. IXP2800 Network Processor Detailed Diagram





2.2 Intel XScale® Core Microarchitecture

The Intel XScale® microarchitecture consists of a 32-bit general purpose RISC processor that incorporates an extensive list of architecture features that allows it to achieve high performance.

2.2.1 ARM Compatibility

The Intel XScale® microarchitecture is ARM® Version 5 (V5) Architecture compliant. It implements the integer instruction set of ARM® V5, but does not provide hardware support of the floating point instructions.

The Intel XScale® microarchitecture provides the Thumb instruction set (ARM V5T) and the ARM V5E DSP extensions.

Backward compatibility with the first generation of StrongARM® products is maintained for user-mode applications. Operating systems may require modifications to match the specific hardware features of the Intel XScale® microarchitecture and to take advantage of the performance enhancements added to the Intel XScale® core.

2.2.2 Features

2.2.2.1 Multiply/Accumulate (MAC)

The MAC unit supports early termination of multiplies/accumulates in two cycles and can sustain a throughput of a MAC operation every cycle. Several architectural enhancements were made to the MAC to support audio coding algorithms, which include a 40-bit accumulator and support for 16-bit packed values.

2.2.2.2 Memory Management

The Intel XScale® microarchitecture implements the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Reference Manual*. The MMU provides access protection and virtual to physical address translation.

The MMU Architecture also specifies the caching policies for the instruction cache and data memory. These policies are specified as page attributes and include:

- identifying code as cacheable or non-cacheable
- selecting between the mini-data cache or data cache
- write-back or write-through data caching
- enabling data write allocation policy
- and enabling the write buffer to coalesce stores to external memory

2.2.2.3 Instruction Cache

The Intel XScale® microarchitecture implements a 32-Kbyte, 32-way set associative instruction cache with a line size of 32 bytes. All requests that “miss” the instruction cache generate a 32-byte read request to external memory. A mechanism to lock critical code within the cache is also provided.

2.2.2.4 Branch Target Buffer

The Intel XScale® microarchitecture provides a Branch Target Buffer (BTB) to predict the outcome of branch type instructions. It provides storage for the target address of branch type instructions and predicts the next address to present to the instruction cache when the current instruction address is that of a branch.

The BTB holds 128 entries.

2.2.2.5 Data Cache

The Intel XScale® microarchitecture implements a 32-Kbyte, 32-way set associative data cache and a 2-Kbyte, 2-way set associative mini-data cache. Each cache has a line size of 32 bytes, and supports write-through or write-back caching.

The data/mini-data cache is controlled by page attributes defined in the MMU Architecture and by coprocessor 15.

The Intel XScale® microarchitecture allows applications to re-configure a portion of the data cache as data RAM. Software may place special tables or frequently used variables in this RAM.

2.2.2.6 Interrupt Controller

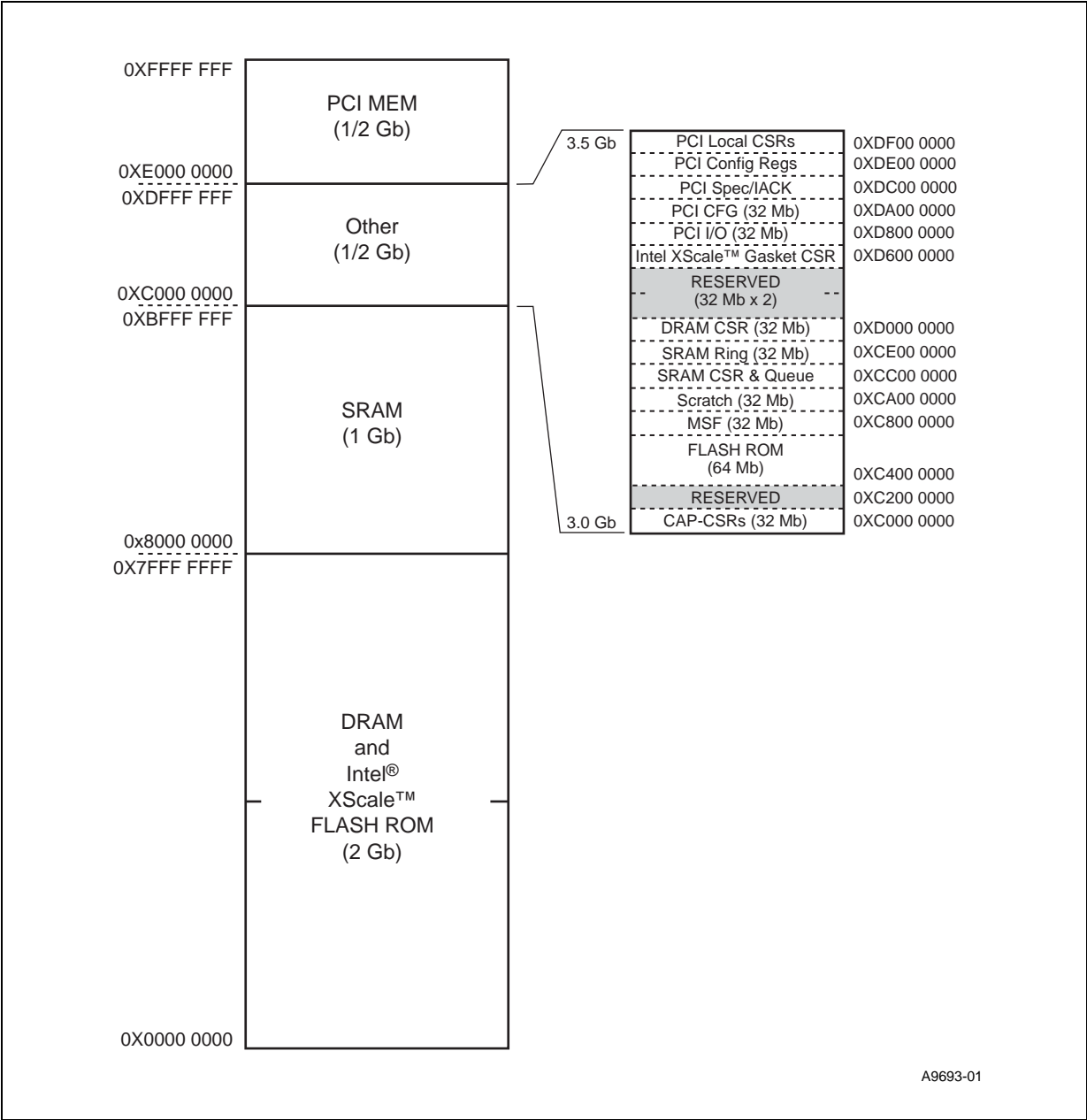
The Intel XScale® microarchitecture provides two levels of interrupt, IRQ and FIQ. They can be masked via coprocessor 13. Note that there is also a memory mapped interrupt controller described with the Intel XScale® Peripherals (see [Section 3.12](#)), which is used to mask and steer many chip-wide interrupt sources.

2.2.2.7 Address Map

[Figure 3](#) shows the partitioning of the Intel XScale® microarchitecture 4 GB address space.



Figure 3. Intel XScale® 4GB (32-bit) Address Space



A9693-01



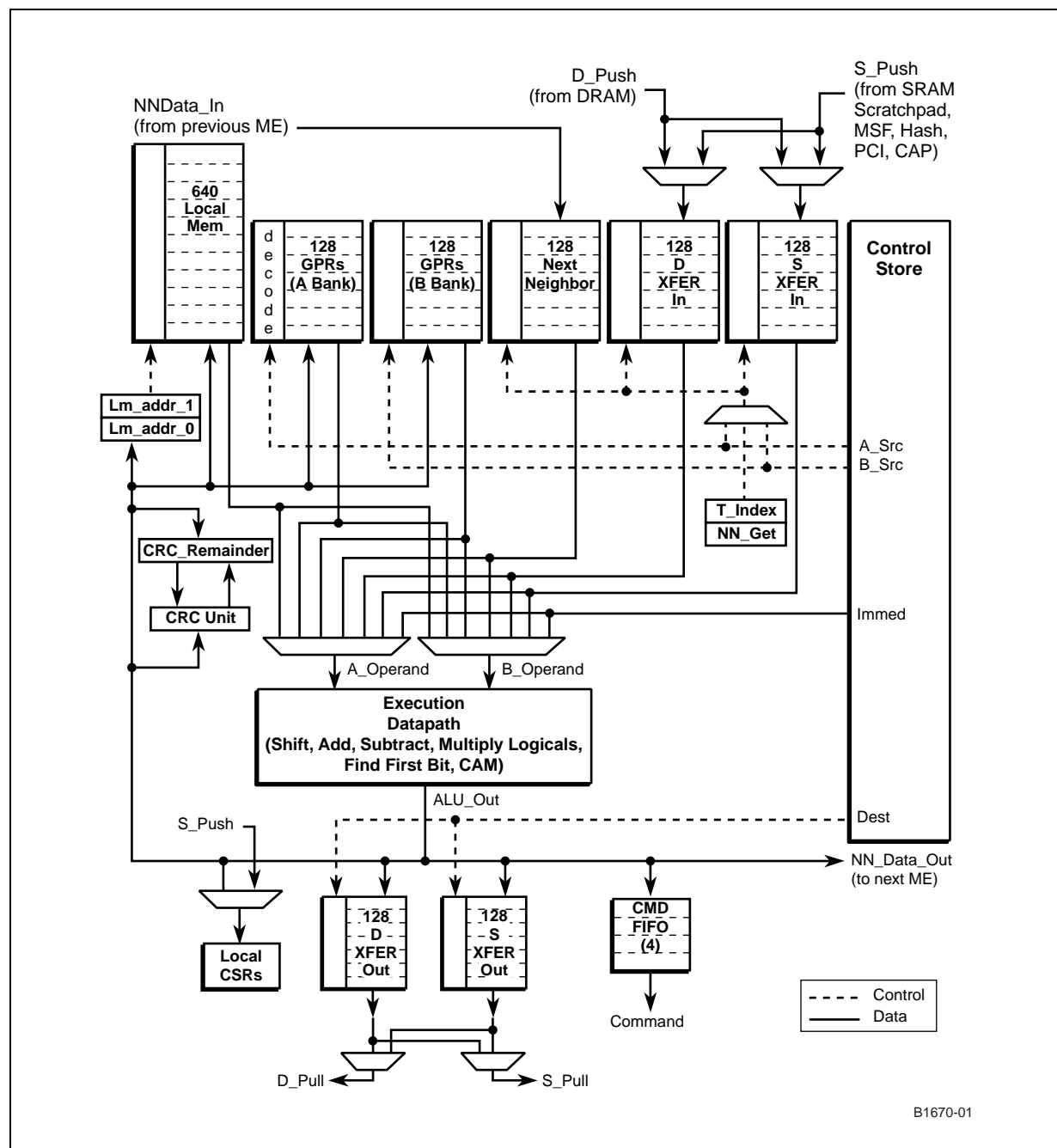
2.3 Microengines

The Microengines do most of the programmable pre-packet processing in the IXP2800 Network Processor. There are 16 Microengines, connected as shown in [Figure 1](#). The Microengines have access to all shared resources (SRAM, DRAM, MSF, etc.) as well as private connections between adjacent Microengines (referred to as “next neighbors”).

The block diagram in [Figure 4](#) is used in the Microengine description. Note that this block diagram is simplified for clarity; some blocks and connectivity have been omitted to make the diagram more readable. Also, this block diagram does not show any pipeline stages, rather it shows the logical flow of information.

The Microengine provides support for software controlled multi-threaded operation. Given the disparity in processor cycle times vs external memory times, a single thread of execution will often block waiting for external memory operations to complete. Having multiple threads available allows for threads to interleave operation—there is often at least one thread ready to run while others are blocked.

Figure 4. Microengine Block Diagram



2.3.1 Microengine Bus Arrangement

The IXP2800 Network Processor supports a single D-Push/Pull Bus and both Microengine clusters interface to the same bus. the IXP2800 Network Processor supports two command buses and two sets of S-Push/Pull Buses and are connected as shown in Table 2. Table 2 also shows the next neighbor relationship between the Microengine.

Table 2. IXP2800 Network Processor Microengine Bus Arrangement

Microengine Cluster	Microengine Number	Next Neighbor	Previous Neighbor	Command Bus	S Push and Pull Bus
0	0x00	0x01	NA	0	0
	0x01	0x02	0x00		
	0x02	0x03	0x01		
	0x03	0x04	0x02		
	0x04	0x05	0x03		
	0x05	0x06	0x04		
	0x06	0x07	0x05		
	0x07	0x10	0x06		
1	0x10	0x11	0x07	1	1
	0x11	0x12	0x10		
	0x12	0x13	0x11		
	0x13	0x14	0x12		
	0x14	0x15	0x13		
	0x15	0x16	0x14		
	0x16	0x17	0x15		
	0x17	NA	0x16		

2.3.2 Control Store

The Control Store is a RAM, which holds the program that the Microengine executes. It holds 8192 instructions, each of which is 40-bits wide. It is initialized by the Intel XScale® core, which writes to USTORE_ADDR and USTORE_DATA Local CSRs.

The Control Store is protected by parity against soft errors. Parity checking is enabled by CTX_ENABLE[CONTROL STORE PARITY ENABLE]. A parity error on an instruction read will halt the Microengine and assert an interrupt to the Intel XScale® core.

2.3.3 Contexts

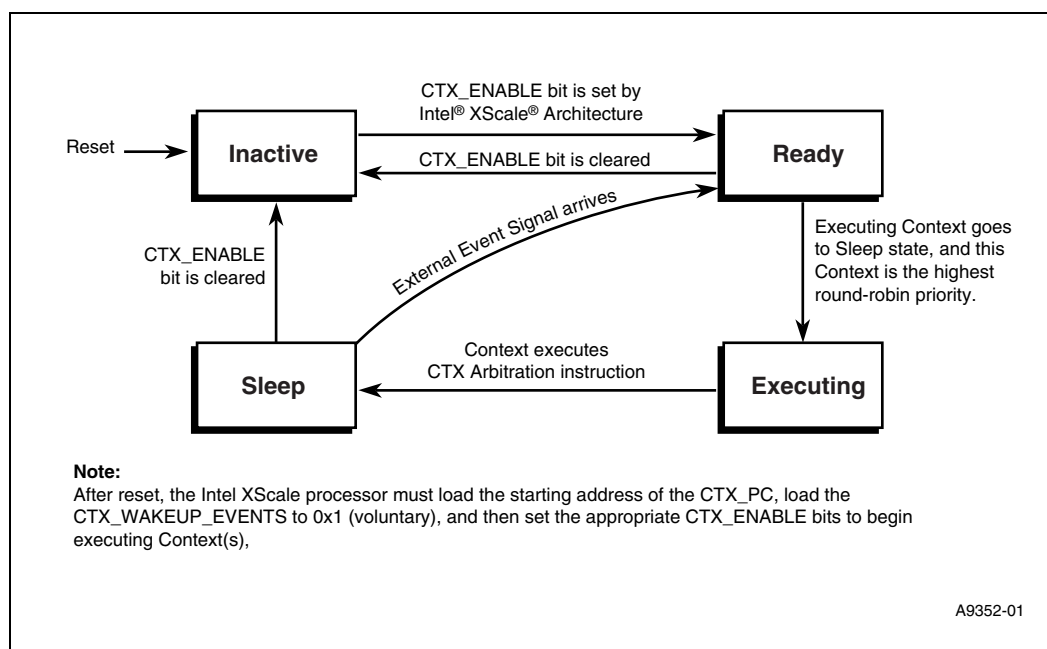
There are eight hardware Contexts available in the Microengine. To allow for efficient context swapping, each Context has its own register set, Program Counter, and Context specific Local Registers. Having a copy per Context eliminates the need to move Context specific information to/from shared memory and Microengine registers for each Context swap. Fast context swapping allows a Context to do computation while other Contexts wait for I/O (typically external memory accesses) to complete or for a signal from another Context or hardware unit. [Note that a context swap is similar to a taken branch in timing.]

Each of the eight Contexts is in one of four states.

1. Inactive—Some applications may not require all eight contexts. A Context is in the Inactive state when its CTX_ENABLE CSR enable bit is a '0'.
2. Executing—A Context is in Executing state when its context number is in ACTIVE_CTX_STS CSR. The executing Context's PC is used to fetch instructions from the Control Store. A Context will stay in this state until it executes an instruction that causes it to go to Sleep state (there is no hardware interrupt or preemption; Context swapping is completely under software control). At most one Context can be in Executing state at any time.
3. Ready—In this state, a Context is ready to execute, but is not because a different Context is executing. When the Executing Context goes to Sleep state, the Microengine's context arbiter selects the next Context to go to the Executing state from among all the Contexts in the Ready state. The arbitration is round robin.
4. Sleep—Context is waiting for external event(s) specified in the INDIRECT_WAKEUP_EVENTS CSR to occur (typically, but not limited to, an I/O access). In this state the Context does not arbitrate to enter the Executing state.

The state diagram in Figure 5 illustrates the Context state transitions. Each of the eight Contexts will be in one of these states. At most one Context can be in Executing state at a time; any number of Contexts can be in any of the other states.

Figure 5. Context State Transition Diagram



The Microengine is in Idle state whenever no Context is running (all Contexts are in either Inactive or Sleep states). This state is entered:

1. After reset (because CTX_ENABLE Local CSR is clear, putting all Contexts into Inactive states).
2. When a context swap is executed, but no context is ready to wakeup.

3. When a `ctx_arb[bpt]` instruction is executed by the Microengine (this is a special case of #2 above, since the `ctx_arb[bpt]` clears `CTX_ENABLE`, putting all Contexts into Inactive states).

The Microengine provides the following functionality during Idle state:

1. The Microengine continuously checks if a Context is in Ready state. If so, a new Context begins to execute. If no Context is Ready, the Microengine remains in the Idle state.
2. Only the ALU instructions are supported. They are used for debug via special hardware defined in number 3 below.
3. A write to the `USTORE_ADDR` Local CSR with the `USTORE_ADDR[ECS]` bit set, causing the Microengine to repeatedly execute the instruction pointed by the address specified in the `USTORE_ADDR` CSR. Only the ALU instructions are supported in this mode. Also, the result of the execution is written to the `ALU_OUT` Local CSR rather than a destination register.
4. A write to the `USTORE_ADDR` Local CSR with the `USTORE_ADDR[ECS]` bit set, followed by a write to the `USTORE_DATA` Local CSR loads an instruction into the Control Store. After the Control Store is loaded, execution proceeds as described in number 3 above.

2.3.4 Datapath Registers

As shown in the block diagram in [Figure 4](#) each Microengine contains four types of 32-bit datapath registers:

1. 256 General Purpose Registers
2. 512 Transfer Registers
3. 128 Next Neighbor Registers
4. 640 32-bit words of Local Memory

2.3.4.1 General-Purpose Registers (GPRs)

GPRs are used for general programming purposes. They are read and written exclusively under program control. GPRs, when used as a source in an instruction, supply operands to the execution datapath. When used as a destination in an instruction, they are written with the result of the execution datapath. The specific GPRs selected are encoded in the instruction.

The GPRs are physically and logically contained in two banks, GPR A, and GPR B, defined in [Table 3](#).

2.3.4.2 Transfer Registers

Transfer Registers (abbreviated Xfer Registers) are used for transferring data to and from the Microengine and locations external to the Microengine, (for example DRAMs, SRAMs etc.). There are four types of transfer registers.

- `S_TRANSFER_IN`
- `S_TRANSFER_OUT`
- `D_TRANSFER_IN`
- `D_TRANSFER_OUT`



TRANSFER_IN Registers, when used as a source in an instruction, supply operands to the execution datapath. The specific register selected is either encoded in the instruction, or selected indirectly via T_INDEX. TRANSFER_IN Registers are written by external units (A typical case is when the external unit returns data in response to read instructions. However, there are other methods to write TRANSFER_IN Registers, for example a read instruction executed by one Microengine may cause the data to be returned to a different Microengine. Details are covered in the instruction set descriptions).

TRANSFER_OUT Registers, when used as a destination in an instruction, are written with the result from the execution datapath. The specific register selected is encoded in the instruction, or selected indirectly via T_INDEX. TRANSFER_OUT Registers supply data to external units (for example, write data for an SRAM write).

The S_TRANSFER_IN and S_TRANSFER_OUT Registers connect to the S_PUSH and S_PULL busses, respectively.

The D_TRANSFER_IN and D_TRANSFER_OUT Transfer Registers connect to the D_PUSH and D_PULL busses, respectively.

Typically, the external units access the Transfer Registers in response to instructions executed by the Microengines. However, it is possible for an external unit to access a given Microengine's Transfer Registers either autonomously, or under control of a different Microengine, or the Intel XScale® core, etc. The Microengine interface signals controlling writing/reading of the Transfer_IN/TRANSFER_OUT registers are independent of the operation of the rest of the Microengine, therefore the data movement does not stall or impact other instruction processing (it is the responsibility of software to synchronize usage of read data).

2.3.4.3 Next Neighbor Registers

Next Neighbor Registers, when used as a source in an instruction, supply operands to the execution datapath. They are written in two different ways:

1. by an adjacent Microengine (the "Previous Neighbor").
2. by the same Microengine they are in, as controlled by CTX_ENABLE[NN_MODE].

The specific register is selected in one of two ways:

1. Context-relative, the register number is encoded in the instruction.
2. As a Ring, selected via NN_GET and NN_PUT CSR Registers.

The usage is configured in CTX_ENABLE[NN_MODE].

- When CTX_ENABLE[NN_MODE] is '0' -- When Next Neighbor is used as a destination in an instruction, the instruction result data is sent out of the Microengine, to the Next Neighbor Microengine.
- When CTX_ENABLE[NN_MODE] is '1' -- When Next Neighbor is used as a destination in an instruction, the instruction result data is written to the selected Next Neighbor Register in the same Microengine. Note that there is a 5 instruction latency until the newly written data may be read. The data is not sent out of the Microengine as it would be when CTX_ENABLE[NN_MODE] is '0'.

Table 3. Next Neighbor Write as a Function of CTX_ENABLE[NN_MODE]

NN_MODE	Where Does Write Go?	
	External	NN Register in This Microengine
0	Yes	No
1	No	Yes

2.3.4.4 Local Memory

Local Memory is addressable storage located in the Microengine. Local Memory is read and written exclusively under program control. Local Memory supplies operands to the execution datapath as a source, and receives results as a destination. The specific Local Memory location selected is based on the value in one of the LM_ADDR Registers, which are written by local_csr_wr instructions. There are two LM_ADDR Registers per Context and a working copy of each. When a Context goes to Sleep state, the value of the working copies is put into the Context's copy of LM_ADDR. When the Context goes to Executing state, the value in its copy of LM_ADDR are put into the working copies. The choice of LM_ADDR_0 or LM_ADDR_1 is selected in the instruction.

It is also possible to make use of both or one LM_ADDRs as global by setting CTX_ENABLE[LM_ADDR_0_GLOBAL] and/or CTX_ENABLE[LM_ADDR_1_GLOBAL]. When used globally, all Contexts use the working copy of LM_ADDR in place of their own Context specific one; the Context specific ones are unused.

There is a three-instruction latency when writing a new value to the LM_ADDR, as shown in [Example 1](#).

Example 1. Three-Cycle Latency when Writing a New Value to LM_ADDR

```
;some instruction to compute the address into gpr_m
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_m]; put gpr_m into lm_addr
;unrelated instruction 1
;unrelated instruction 2
;unrelated instruction 3
alu[dest_reg, *l$index0, op, src_reg]
;dest_reg can be used as a source in next instruction
```

LM_ADDR can also be incremented or decremented in parallel with use as a source and/or destination (using the notation *l\$index#++ and *l\$index#--), as shown in [Example 2](#), where three consecutive Local Memory locations are used in three consecutive instructions.

Example 2. Using LM_ADDR in Consecutive Instructions

```
alu[dest_reg1, src_reg1, op, *l$index0++]
alu[dest_reg2, src_reg2, op, *l$index0++]
alu[dest_reg3, src_reg3, op, *l$index0++]
```

Local Memory is written by selecting it as a destination. [Example 3](#) shows copying a section of Local Memory to another section. Each instruction accesses the next sequential Local Memory location from the previous instruction.

Example 3. Copying One Section of Local Memory to Another Section

```
alu[*l$index1++, --, B, *l$index0++]
alu[*l$index1++, --, B, *l$index0++]
alu[*l$index1++, --, B, *l$index0++]
```

[Example 4](#) shows loading and using both Local Memory addresses.

Example 4. Loading and Using both Local Memory Addresses

```
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_m]
local_csr_wr[INDIRECT_LM_ADDR_1, gpr_n]
;unrelated instruction 1
;unrelated instruction 2
alu[dest_reg1, *l$index0, op, src_reg1]
alu[dest_reg2, *l$index1, op, src_reg2]
```

As shown in [Example 1](#) there is a latency in loading LM_ADDR. Until the new value is loaded the old value is still usable. [Example 5](#) shows the maximum pipelined usage of LM_ADDR.

Example 5. Maximum Pipelined Usage of LM_ADDR

```
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_m]
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_n]
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_o]
local_csr_wr[INDIRECT_LM_ADDR_0, gpr_p]
alu[dest_reg1, *l$index0, op, src_reg1] ; uses address from gpr_m
alu[dest_reg2, *l$index0, op, src_reg2] ; uses address from gpr_n
alu[dest_reg3, *l$index0, op, src_reg3] ; uses address from gpr_o
alu[dest_reg4, *l$index0, op, src_reg4] ; uses address from gpr_p
```

LM_ADDR can also be used as the base of a 16 32-bit word region of memory, with the instruction specifying the offset from that base, as shown in [Example 6](#). The source and destination can use different offsets.

Example 6. LM_ADDR Used as Base of a 16 32-bit word Region of Local Memory

```
alu[*l$index0[3], *l$index0[4], +, 1]
```

Note: Local Memory has 640 32-bit words. The local memory pointers (LM_ADDR) have an addressing range of up to 1K longwords. However, only 640 longwords are currently populated with RAM. Therefore:

0 - 639 (0x0 - 0x27F) are addressable as local memory.

640 - 1023 (0x280 - 0x3FF) are addressable, but not populated with RAM.

To the programmer, all instructions using Local Memory act as follows, including read/modify/write instructions like `immed_w0`, `ld_field`, etc.

1. Read LM_ADDR location (if LM_ADDR is specified as source).
2. Execute logic function.
3. Write LM_ADDR location (if LM_ADDR is specified as destination).

4. If specified, increment or decrement LM_ADDR.
5. Proceed to next instruction.

Example 7 is legal because `lm_addr_0[2]` does not post-modify LM_ADDR.

Example 7. LM_ADDR Use as Source and Destination

```
alu[*l$index0[2], --, ~B, *l$index0]
```

In Example 7, the programmer sees:

1. Read Local Memory memory location pointed to by LM_ADDR.
2. Invert the data.
3. Write the data into the address pointed to by LM_ADDR with the value of “2” OR’ed into the lower bits.
4. Increment LM_ADDR.
5. Proceed to next instruction.

In Example 8, the second instruction will access the Local Memory location one past the source/destination of the first.

Example 8. LM_ADDR Post-increment

```
alu[*l$index0++, --, ~B, gpr_n]
alu[gpr_m, --, ~B, *l$index0]
```

2.3.5 Addressing Modes

GPRs can be accessed in two different addressing modes:

- Context-Relative
- Absolute

Some instructions can specify either mode, other instructions can specify only Context-Relative mode.

Transfer and Next Neighbor registers can be accessed in Context-Relative and Indexed modes.

Local Memory is accessed in Indexed mode.

The addressing mode in use is encoded directly into each instruction, for each source and destination specifier.

2.3.5.1 Context-Relative Addressing Mode

The GPRs are logically subdivided into equal regions such that each Context has relative access to one of the regions. The number of regions is configured in the CTX_ENABLE CSR, and can be either 4 or 8. Thus a Context-Relative register number is actually associated with multiple different physical registers. The actual register to be accessed is determined by the Context making the access request (the Context number is concatenated with the register number specified in the instruction). Context-Relative addressing is a powerful feature that enables eight (or four) different contexts to share the same code image, yet maintain separate data.

Table 4 shows how the Context number is used in selecting the register number in relative mode. The register number in Table 4 is the Absolute GPR address, or Transfer or Next Neighbor Index number to use to access the specific Context-Relative register. For example, with 8 active Contexts, Context-Relative Register 0 for Context 2 is Absolute Register Number 32.

Table 4. Registers Used By Contexts in Context-Relative Addressing Mode

Number of Active Contexts	Active Context Number	GPR Absolute Register Numbers		S Transfer or Neighbor Index Number	D Transfer Index Number
		A Port	B Port		
8 (Instruction always specifies Registers in range 0-15)	0	0-15	0-15	0-15	0-15
	1	16-31	16-31	16-31	16-31
	2	32-47	32-47	32-47	32-47
	3	48-63	48-63	48-63	48-63
	4	64-79	64-79	64-79	64-79
	5	80-95	80-95	80-95	80-95
	6	96-111	96-111	96-111	96-111
	7	112-127	112-127	112-127	112-127
4 (Instruction always specifies Registers in range 0-31)	0	0-31	0-31	0-31	0-31
	2	32-63	32-63	32-63	32-63
	4	64-95	64-95	64-95	64-95
	6	96-127	96-127	96-127	96-127

2.3.5.2 Absolute Addressing Mode

With Absolute addressing, any GPR can be read or written by any one of the eight Contexts in a Microengine. Absolute addressing enables register data to be shared among all of the Contexts, for example for global variables, or for parameter passing. All 256 GPRs can be read by Absolute address.

2.3.5.3 Indexed Addressing Mode

With Indexed addressing, any Transfer or Next Neighbor register can be read or written by any one of the eight Contexts in a Microengine. Indexed addressing enables register data to be shared among all of the Contexts. For indexed addressing the register number comes from the T_INDEX register for Transfer Registers or NN_PUT and NN_GET registers (for Next Neighbor Registers).

Example 9 shows an example of using the Index Mode. Assume that the numbered bytes have been moved into the S_Transfer_In registers as shown.

Example 9. Use of Indexed Addressing Mode (Sheet 1 of 2)

Transfer Register #	Data			
	31:24	23:16	15:8	7:0
0	0x00	0x01	0x02	0x03
1	0x04	0x05	0x06	0x07
2	0x08	0x09	0x0a	0x0b

Example 9. Use of Indexed Addressing Mode (Sheet 2 of 2)

Transfer Register #	Data			
	31:24	23:16	15:8	7:0
3	0x0c	0x0d	0x0e	0x0f
4	0x10	0x11	0x12	0x013
5	0x14	0x15	0x16	0x17
6	0x18	0x19	0x1a	0x1b
7	0x1c	0x1d	0x1e	0x1f

If the software wants to access a specific byte that is known at compile-time, it will normally use context-relative addressing. For example to access the word in transfer register 3:

```
alu[dest, --, B, $xfer3] ; move the data from s_transfer 3 to gpr dest
```

If the location of the data is found at run-time, indexed mode can be used. For example, the case where the start of an encapsulated header is dependent on a value in an outer header (the outer header byte is in a fixed location).

```
; Check byte 2 of transfer 0
; If value==5 header starts on byte 0x9, else byte 0x14
br=byte[$0, 2, 0x5, L1#], defer_[1]
local_csr_wr[t_index_byte_index, 0x09]
local_csr_wr[t_index_byte_index, 0x14]
nop ; wait for index registers to be loaded

L1#:
; Move bytes right justified into destination registers
nop ; wait for index registers to be loaded
nop ;
byte_align_be[dest1, *$index++]
byte_align_be[dest2, *$index++]
; etc
```

Note that the `t_index` and `byte_index` registers are loaded by the same instruction.

2.3.6 Local CSRs

Local Control and Status Registers (CSRs) are external to the Execution Datapath, and hold specific purpose information. They can be read and written by special instructions (`local_csr_rd` and `local_csr_wr`) and are typically accessed less frequently than datapath registers.

Because Local CSRs are not built in the datapath, there is a write to use delay of three instructions, and a read to consume penalty of two instructions.

2.3.7 Execution Datapath

The Execution Datapath can take one or two operands, perform an operation, and optionally write back a result. The sources and destinations can be GPRs, Transfer Registers, Next Neighbor Registers, and Local Memory. The operations are shifts, add/subtract, logicals, multiply, byte align, and find first one bit.

2.3.7.1 Byte Align

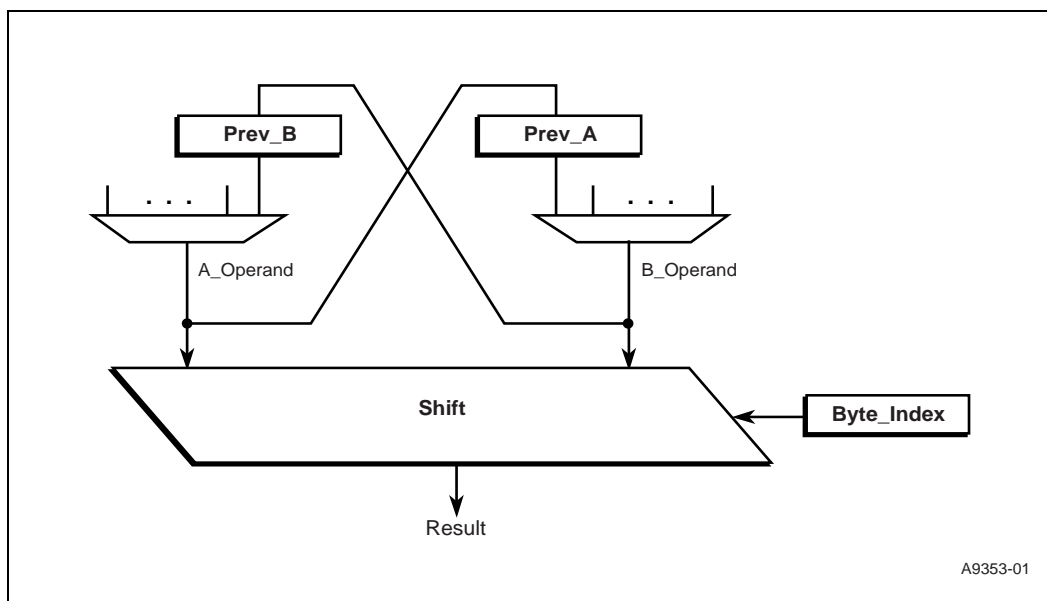
The datapath provides a mechanism to move data from source register(s) to any destination register(s) with byte aligning. Byte aligning takes four consecutive bytes from two concatenated values (8 bytes), starting at any of four byte boundaries (0, 1, 2, 3), and based on the endian-type (which is defined in the instruction opcode), as shown in [Example 4](#). The four bytes are taken from two concatenated values. Four bytes are always supplied from a temporary register that always holds the A or B operand from the previous cycle, and the other four bytes from the B or A operand of the Byte Align instruction.

The operation is described below low using the block diagram in [Figure 6](#). The alignment is controlled by the 2 LSBs of the BYTE_INDEX Local CSR.

Table 5. Align Value and Shift Amount

Align Value (in Byte_Index[1:0])	Right Shift Amount (# of Bits) (Decimal)	
	Little Endian	Big Endian
0	0	32
1	8	24
2	16	16
3	24	8

Figure 6. Byte Align Block Diagram



Example 11 shows an align sequence of instructions and the value of the various operands. Table 6 shows the data in the registers for this example. The value in BYTE_INDEX[1:0] CSR (which controls the shift amount) for this example is 2.

Table 6. Register Contents for Example 10

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	0	1	2	3
1	4	5	6	7
2	8	9	A	B
3	C	D	E	F

Example 10. Big Endian Align

Instruction	Prev B	A Operand	B Operand	Result
Byte_align_be[--, r0]	--	--	0123	--
Byte_align_be[dest1, r1]	0123	0123	4567	2345
Byte_align_be[dest2, r2]	4567	4567	89AB	6789
Byte_align_be[dest3, r3]	89AB	89AB	CDEF	ABCD
NOTE: A Operand comes from Prev_B register during byte_align_be instructions.				

Example 11 shows another sequence of instructions and the value of the various operands. Table 7 shows the data in the registers for this example.

The value in BYTE_INDEX[1:0] CSR (which controls the shift amount) for this example is 2.

Table 7. Register Contents for Example 11

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	3	2	1	0
1	7	6	5	4
2	B	A	9	8
3	F	E	D	C

Example 11. Little Endian Align

Instruction	A Operand	B Operand	Prev A	Result
Byte_align_le[--, r0]	3210	--	--	--
Byte_align_le[dest1, r1]	7654	3210	3210	5432
Byte_align_le[dest2, r2]	BA98	7654	7654	9876
Byte_align_le[dest3, r3]	FEDC	BA98	BA98	DCBA
NOTE: B Operand comes from Prev_A register during byte_align_le instructions.				



As the examples show, byte aligning “n” words takes “n+1” cycles due to the first instruction needed to start the operation.

Another mode of operation is to use the T_INDEX register with post-increment, to select the source registers. T_INDEX operation is described later in this chapter.

2.3.7.2 CAM

The block diagram in [Figure 7](#) is used to explain the CAM operation.

The CAM has 16 entries. Each entry stores a 32 bit value, which can be compared against a source operand by instruction:

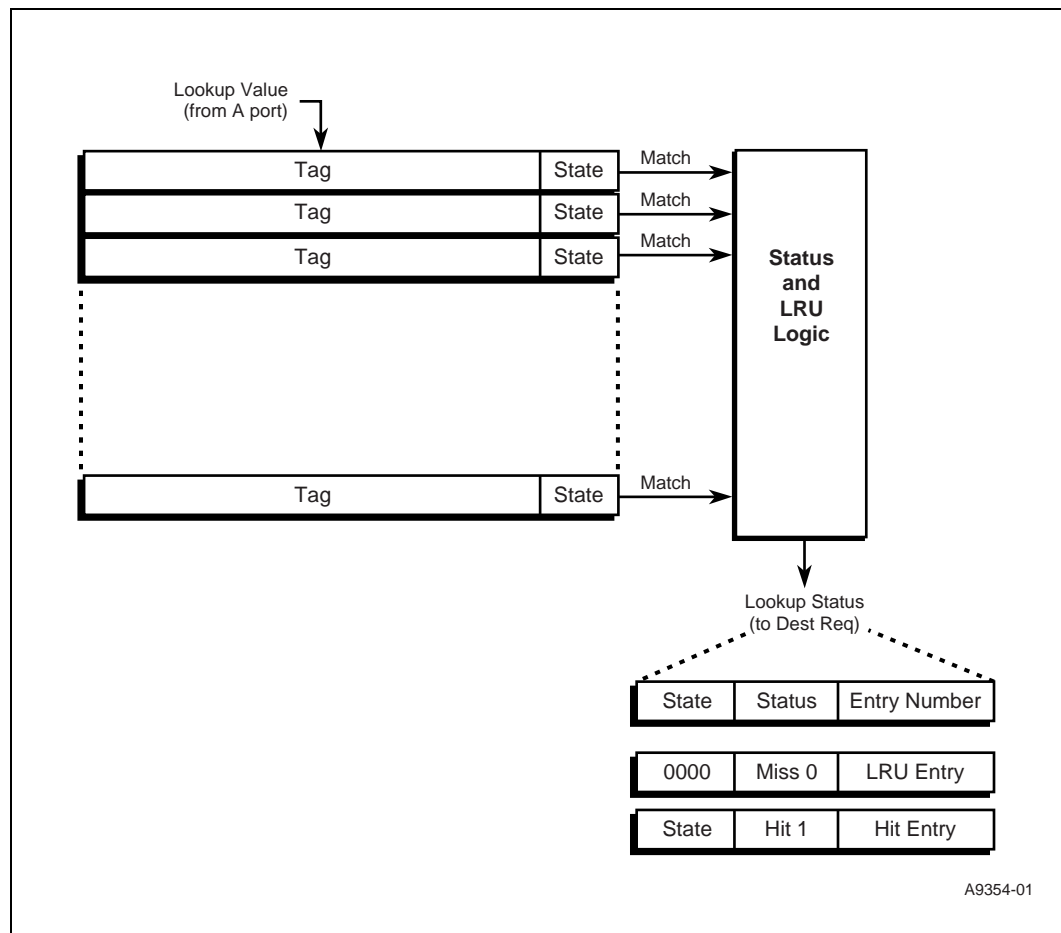
```
CAM_Lookup[dest_reg, source_reg]
```

All entries are compared in parallel, and the result of the lookup is a 9 bit value which is written into the specified destination register in bits 11:3, with all other bits of the register zero (the choice of bits 11:3 is explained below). The result can also optionally be written into either of the LM_Addr registers (see below in this section for details).

The 9-bit result consists of 4 State bits (dest_reg[11:8]), concatenated with a 1-bit Hit/Miss indication (dest_reg[7]), concatenated with 4-bit entry number (dest_reg[6:3]). All other bits of dest_reg are written with 0. Possible results of the lookup are:

- miss (0)—lookup value is not in CAM, entry number is Least Recently Used entry (which can be used as a suggested entry to replace), and State bits are 0000.
- hit (1)—lookup value is in CAM, entry number is entry which has matched, State bits are the value from the entry which has matched.

Figure 7. CAM Block Diagram



Note: The State bits are data associated with the entry. The use is only by software. There is no implication of ownership of the entry by any Context. The State bits hardware function is:

- the value is set by software (at the time the entry is loaded, or changed in an already loaded entry).
- its value is read out on a lookup that hits, and used as part of the status written into the destination register.
- its value can be read out separately (normally only used for diagnostic or debug).

The LRU (Least Recently Used) Logic maintains a time-ordered list of CAM entry usage. When an entry is loaded, or matches on a lookup, it is marked as MRU (Most Recently Used). Note that a lookup that misses does **not** modify the LRU list.

The CAM is loaded by instruction:

```
CAM_Write[entry_reg, source_reg, state_value]
```

The value in the register specified by `source_reg` is put into the Tag field of the entry specified by `entry_reg`. The value for the State bits of the entry is specified in the instruction as `state_value`.

The value in the State bits for an entry can be written, without modifying the Tag, by instruction:

```
CAM_Write_State[entry_reg, state_value]
```

Note: CAM_Write_State does **not** modify the LRU list.

One possible way to use the result of a lookup is to dispatch to the proper code using instruction:

```
jump[register, label#],defer [3]
```

where the register holds the result of the lookup. The State bits can be used to differentiate cases where the data associated with the CAM entry is in flight, or is pending a change, etc. Because the lookup result was loaded into bits[11:3] of the destination register, the jump destinations are spaced 8 instructions apart. This is a balance between giving enough space for many applications to complete their task without having to jump to another region, vs consuming too much Control Store. Another way to use the lookup result is to branch on just the hit miss bit, and use the entry number as a base pointer into a block of Local Memory.

When enabled, the CAM lookup result is loaded into Local_Addr as follows:

```
LM_Addr[5:0] = 0 ([1:0] are read-only bits)
LM_Addr[9:6] = lookup result [6:3] (entry number)
LM_Addr[11:10] = constant specified in instruction
```

This function is useful when the CAM is used as a cache, and each entry is associated with a block of data in Local Memory. Note that the latency from when CAM_Lookup executes until the LM_Addr is loaded is the same as when LM_Addr is written by a Local_CSR_Wr instruction.

The Tag and State bits for a given entry can be read by instructions:

```
CAM_Read_Tag[dest_reg, entry_reg]
CAM_Read_State[dest_reg, entry_reg]
```

The Tag value and State bits value for the specified entry is written into the destination register, respectively for the two instructions (the State bits are placed into bits [11:8] of dest_reg, with all other bits 0). Reading the tag is useful in the case where an entry needs to be evicted to make room for a new value—the lookup of the new value results in a miss, with the LRU entry number returned as a result of the miss. The CAM_Read_Tag instruction can then be used to find the value that was stored in that entry. An alternative would be to keep the tag value in a GPR. These two instructions can also be used by debug and diagnostic software. Neither of these modify the state of the LRU pointer.

Note: The following rules must be adhered to when using the CAM.

- CAM is not reset by Microengine reset. Software must either do a CAM_clear prior to using the CAM to initialize the LRU and clear the tags to zero, or explicitly write all entries with CAM_write.
- No two tags can be written to have same value. If this rule is violated, the result of a lookup that matches that value will be unpredictable, and LRU state is unpredictable.

The value 0x00000000 can be used as a valid lookup value. However, note that CAM_clear instruction puts 0x00000000 into all tags. So in order to not violate rule 2 after doing CAM_clear, it is necessary to write all entries to unique values prior to doing a lookup of 0x00000000.

An algorithm for debug software to find out the contents of the CAM is shown in [Table 8](#).

Table 8. Algorithm for Debug Software to Find out the Contents of the CAM

```

; First read each of the tag entries. Note that these reads
; don't modify the LRU list or any other CAM state.
tag[0] = CAM_Read_Tag(entry_0);
.....
tag[15] = CAM_Read_Tag(entry_15);
; Now read each of the state bits
state[0] = CAM_Read_State(entry_0);
...
state[15] = CAM_Read_State(entry_15);
; Knowing what tags are in the CAM makes it possible to
; create a value that is not in any tag, and will therefore
; miss on a lookup.

; Next loop through a sequence of 16 lookups, each of which will
; miss, to obtain the LRU values of the CAM.
for (i = 0; i < 16; i++)
    BEGIN_LOOP
        ; Do a lookup with a tag not present in the CAM. On a
        ; miss, the LRU entry will be returned. Since this lookup
        ; missed the LRU state is not modified.
        LRU[i] = CAM_Lookup(some_tag_not_in_cam);
        ; Now do a lookup using the tag of the LRU entry. This
        ; lookup will hit, which makes that entry MRU.
        ; This is necessary to allow the next lookup miss to
        ; see the next LRU entry.
        junk = CAM_Lookup(tag[LRU[i]]);
    END_LOOP
; Because all entries were hit in the same order as they were
; LRU, the LRU list is now back to where it started before the
; loop executed.
; LRU[0] through LRU[15] holds the LRU list.

```

The CAM can be cleared with CAM_Clear instruction. This instruction writes 0x00000000 simultaneously to all entries tag, clears all the state bits, and puts the LRU into an initial state (where entry 0 is LRU, ..., entry 15 is MRU).

2.3.8 CRC Unit

The CRC Unit operates in parallel with the Execution Datapath. It takes two operands, performs a CRC operation, and writes back a result. CRC-CCITT, CRC-32, CRC-10, CRC-5, and iSCSI polynomials are supported. One of the operands is the CRC_Remainder Local CSR, and the other is a GPR, Transfer In Register, Next Neighbor, or Local Memory, specified in the instruction and passed through the Execution Datapath to the CRC Unit. The instruction specifies the CRC operation type, whether to swap bytes and or bits, and which bytes of the operand to include in the operation. The result of the CRC operation is written back into CRC_Remainder. The source operand can also be written into a destination register (however the byte/bit swapping and masking do not affect the destination register; they only affect the CRC computation). This allows moving data, for example, from S Transfer In registers to S Transfer Out registers at the same time as computing the CRC.

2.3.9 Event Signals

Event Signals are used to coordinate a program with completion of external events. For example, when a Microengine executes an instruction to an external unit to read data (which will be written into a TRANSFER_IN register), the program must insure that it does not try to use the data until the external unit has written it. This time is not deterministic due to queuing delays and other uncertainty in the external units (for example, DRAM refresh). There is no hardware mechanism to flag that a register write is pending, and then prevent the program from using it. Instead the coordination is under software control, with hardware support.

In the instructions that use external units (i.e., SRAM, DRAM, etc.) there are fields that direct the external unit to supply an indication (called an Event Signal) that the command has been completed. There are 15 Event Signals per Context that can be used, and Local CSRs per Context to track which Event Signals are pending and which have been returned. The Event Signals can be used to move a Context from Sleep state to Ready state, or alternatively, the program can test and branch on the status of Event Signals.

Event Signals can be set in nine different ways.

1. When data is written into S_TRANSFER_IN registers
2. When data is written into D_TRANSFER_IN registers
3. When data is taken from S_TRANSFER_OUT registers
4. When data is taken from D_TRANSFER_OUT registers
5. By a write to INTERTHREAD_SIGNAL register
6. By a write from Previous Neighbor Microengine to NEXT_NEIGHBOR_SIGNAL
7. By a write from Next Neighbor Microengine to PREVIOUS_NEIGHBOR_SIGNAL
8. By a write to SAME_ME_SIGNAL Local CSR
9. By Internal Timer

Any or all Event Signals can be set by any of the above sources.

When a Context goes to Sleep state (executes a `ctx_arb` instruction, or an instruction with `ctx_swap` token), it specifies which Event Signal(s) it requires to be put in Ready state. `ctx_arb` instruction also specifies if the logical AND or logical OR of the Event Signal(s) is needed to put the Context into Ready state.

When all the Context's Event Signals arrive, the Context goes to Ready state, and then eventually to Executing state. In the case where the Event Signal is linked to moving data into or out of Transfer registers (numbers 1 through 4 in the list above), the code can safely use the Transfer register as the first instruction (for example, using a Transfer_In register as a source operand will get the new read data). The same is true when the Event Signal is tested for branches (`br_=signal` or `br_!signal` instructions).

The `ctx_arb` instruction, CTX_SIG_EVENTS, and ACTIVE_CTX_WAKEUP_#_EVENTS Local CSR descriptions provide details.



2.4 DRAM

The IXP2800 Network Processor has controllers for three Rambus® DRAM (RDRAM) channels. Each of the controllers independently accesses its own RDRAMs, and can operate concurrently with the other controllers (i.e. they are not operating as a single, wider memory). DRAM provides high density, high bandwidth storage and is typically used for data buffers.

RDRAM sizes of 64Mb, 128Mb, 256Mb, 512 Mb, and 1 Gb are supported, however, each of the channels must have the same number, size, and speed of RDRAMs populated. Refer to [Section 5.2](#) for supported size and loading configurations.

Up to 2 GB of DRAM is supported. If less than 2 GB of memory is present, the upper part of the address space is not used. It is also possible, for system cost and area savings, to have Channels 0 and 1 populated with Channels 2 empty, or Channel 0 populated with Channels 1 and 2 empty.

Reads and writes to RDRAM are generated by Microengines, The Intel XScale® core, and PCI (external Bus Masters and DMA Channels). The controllers also do refresh and calibration cycles to the RDRAMs, transparently to software.

RDRAM Powerdown and Nap modes are not supported.

Hardware interleaving (also known as striping) of addresses is done to provide balanced access to all populated channels. The interleave size is 128 bytes. Interleaving helps to maintain utilization of available bandwidth by spreading consecutive accesses to multiple channels. The interleaving is done in the hardware in such a way that the three channels appear to software as a single contiguous memory space.

ECC (Error Correcting Code) is supported, but can be disabled. Enabling ECC requires that x18 RDRAMs be used. If ECC is disabled x16 RDRAMs can be used. ECC can detect and correct all single-bit errors, and detect all double-bit errors. When ECC is enabled, partial writes (writes of less than 8 bytes) must be done as read-modify-writes.

2.4.1 Size Configuration

Each channel can be populated with anywhere from one-to-four RDRAMs (Short Channel Mode). Refer to [Section 5.2](#) for supported size and loading configurations. The RAM technology used will determine the increment size and maximum memory per channel as shown in [Table 9](#).

Table 9. RDRAM Sizes

RDRAM Technology ¹	Increment Size	Maximum per Channel
64/72 Mb	8 MB	256 MB
128/144 Mb	16 MB	512 MB
256/288 Mb	32 MB	1 GB ²
512/576 Mb	64 MB	2 GB ²
NOTES: 1. The two numbers shown for each technology indicate x16 parts and x18 parts. 2. The maximum memory that can be addressed across all channels is 2GB. This limitation is based on the partitioning of the 4GB address space (32-bit addresses). Therefore if all three channels are used, each can be populated up to a maximum of 768MB. Two channels can be populated to a maximum of 1 GB each. A single channel could be populated to a maximum of 2 GB.		



RDRAMs with 1 x 16 dependent banks, 2 x 16 dependent banks, and 4 independent banks are supported.

2.4.2 Read and Write Access

The minimum DRAM physical access length is 16 bytes. Software (and PCI) can read or write as little as a single byte, however the time (and bandwidth) taken at the DRAMs is the same as for an access of 16 bytes. Therefore, the best utilization of DRAM bandwidth will be for accesses that are multiples of 16 bytes.

If ECC is enabled, writes of less than 8 bytes must do read-modify-writes, which take two 16-byte time accesses (one for the read and one for the write).

2.5 SRAM

The IXP2800 Network Processor has four independent SRAM controllers, which each support pipelined QDR synchronous static RAM (SRAM) and/or a coprocessor that adheres to QDR signaling. Any or all controllers can be left unpopulated if the application does not need to use them. SRAM are accessible by the Microengines, the Intel XScale® core, and the PCI Unit (external bus masters and DMA).

The memory is logically four bytes (32-bits) wide; physically the data pins are two bytes wide and are double clocked. Byte parity is supported. Each of the four bytes has a parity bit, which is written when the byte is written and checked when the data is read. There are byte enables which select which bytes to write for writes of less than 32-bits.

Each of the 4 QDR ports are QDR and QDRII compatible. Each port implements the “_K” and “_C” output clocks and “_CQ” as an input and their inversions. (Note: the “_C” and “_CQ” clocks are optional). Extensive work has been performed providing impedance controls within the IXP2800 Network Processor for network processor initiated signals driving to QDR parts. Providing a clean signaling environment is critical to achieving 200 to 250 MHz QDRII data transfers.

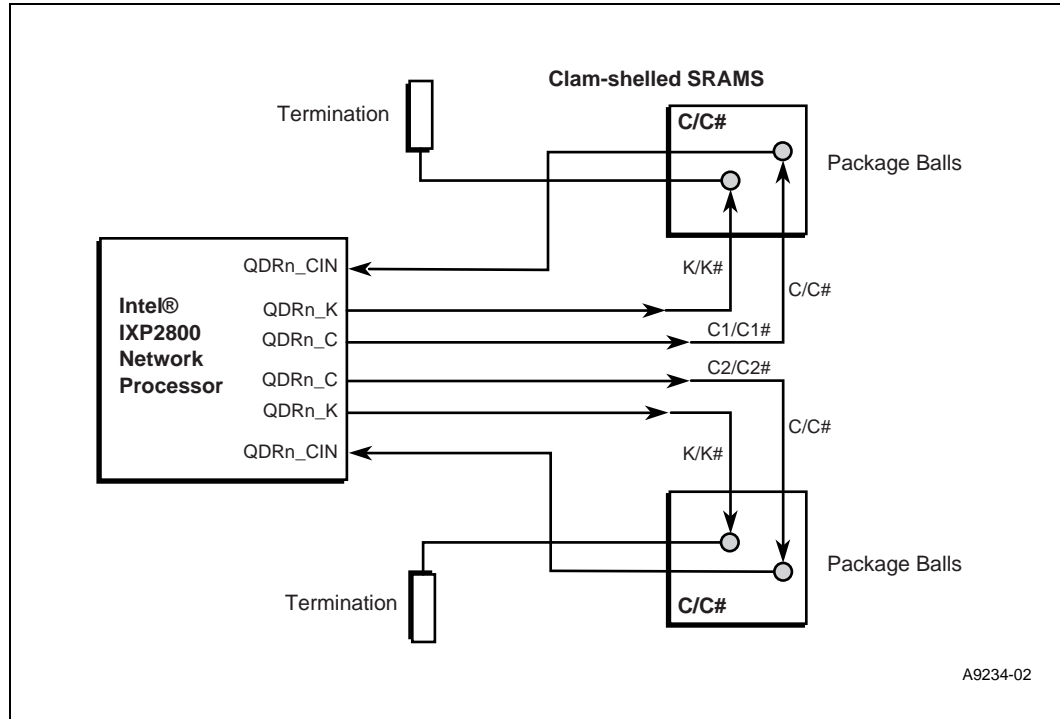
The configuration assumptions for the IXP2800 Network Processor I/O driver/receiver development includes four QDR loads and the IXP2800 Network Processor. The IXP2800 Network Processor supports bursts of 2 SRAMs, but does not support bursts of 4 SRAMs.

The SRAM controller can also be configured to interface to an external coprocessor that adheres to the QDR electricals and protocol. Each SRAM controller may also interface to an external coprocessor through its standard QDR interface. This interface will allow for the cohabitation of both SRAM devices and coprocessors to operate on the same bus. The coprocessor will behave as a memory mapped device on the SRAM bus.

2.5.1 QDR Clocking Scheme

The controller drives out two pairs of K clock (K and K#). It also drives out two pairs of C clock (C and C#). Both C/C# clocks externally return to the controller for reading data. Figure 8 shows the clock diagram if the clocking scheme for QDR interface driving four SRAM chips.

Figure 8. Echo Clock Configuration



2.5.2 SRAM Controller Configurations

Each channel has enough address pins (24) to support up to 64 MB of SRAM. The SRAM controllers can directly generate multiple port enables (up to 4 pairs) to allow for depth expansion. Two pairs of pins are dedicated for port enables. Smaller RAMs use fewer address signals than the number provided to accommodate the largest RAMs, so some address pins (23:20) are configurable as either address or port enable based on CSR setting as shown in [Table 10](#). Note that all of the SRAMs on a given channel must be the same size.

Table 10. SRAM Controller Configurations

SRAM Configuration	SRAM Size	Addresses Needed to Index SRAM	Addresses Used as Port Enables	Total Number of Port Select Pairs Available
512K x 18	1 MB	17:0	23:22, 21:20	4
1M x 18	2 MB	18:0	23:22, 21:20	4
2M x 18	4 MB	19:0	23:22, 21:20	4
4M x 18	8 MB	20:0	23:22	3
8M x 18	16 MB	21:0	23:22	3
16M x 18	32 MB	22:0	None	2
32M x 18	64 MB	23:0	None	2

Each channel can be expanded by depth according to the number of port enables available. If external decoding is used, then the number of SRAMs used is not limited by the number of port enables generated by the SRAM controller.

Note: Doing external decoding may require external pipeline registers to account for the decode time, depending on the desired frequency.

Maximum SRAM system sizes are shown in [Table 11](#). Shaded entries require external decoding, because they use more port enables than the SRAM controller can supply directly.

Table 11. Total Memory per Channel

SRAM Size	Number of SRAMs on Channel							
	1	2	3	4	5	6	7	8
512K x 18	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB
1M x 18	2 MB	4 MB	6 MB	8 MB	10 MB	12 MB	14 MB	16 MB
2M x 18	4 MB	8 MB	12 MB	16 MB	20 MB	24 MB	28 MB	32 MB
4M x 18	8 MB	16 MB	24 MB	32 MB	64 MB	NA	NA	NA
8M x 18	16 MB	32 MB	48 MB	64 MB	NA	NA	NA	NA
16M x 18	32 MB	64 MB	NA	NA	NA	NA	NA	NA
32M x 18	64 MB	NA	NA	NA	NA	NA	NA	NA



2.5.3 SRAM Atomic Operations

In addition to normal reads and writes, SRAM supports the following atomic operations. Microengines have specific instructions to do each atomic operation; Intel XScale® microarchitecture uses aliased address regions to do atomic operations.

- bit set
- bit clear
- increment
- decrement
- add
- swap

The SRAM does read-modify-writes for the atomic operations, the pre-modified data can also be returned if desired. The atomic operations operate on a single 32-bit word.

2.5.4 Queue Data Structure Commands

The ability to enqueue and dequeue data buffers at a fast rate is key to meeting line-rate performance. This is a difficult problem as it involves dependent memory references that must be turned around very quickly. The SRAM controller includes a data structure (called the Q_array) and associated control logic in order to perform efficient enqueue and dequeue operations. The Q_array has 64 entries, each of which can be used in one of four ways.

- Linked-list queue descriptor (resident queues)
- Cache of recently used linked-list queue descriptors (the backing store for the cache is in SRAM)
- Ring descriptor
- Journal

The commands provided are:

For Linked-list queues or Cache of recently used linked-list queue descriptors

- Read_Q_Descriptor_Head(address, length, entry, xfer_addr)
- Read_Q_Descriptor_Tail(address, length, entry)
- Read_Q_Descriptor_Other(address, entry)
- Write_Q_Descriptor(address, entry)
- Write_Q_Descriptor_Count(address, entry)
- ENQ(buff_desc_adr, cell_count, EOP, entry)
- ENQ_tail(buff_desc_adr, entry)
- DEQ(entry, xfer_addr)

For Rings

- Get(entry, length, xfer_addr)
- Put(entry, length, xfer_addr)

For Journals

- Journal(entry, length, xfer_addr)
- Fast_journal(entry)



Note: The Read_Q_Descriptor_Head, Read_Q_Descriptor_Tail, etc.) are used to initialize the rings and journals but not used to perform the ring and journal function.

2.5.5 Reference Ordering

This section covers the ordering between accesses to any one SRAM controller.

2.5.5.1 Reference Order Tables

Table 12 shows the architectural guarantees of order to access to the SAME SRAM address between a reference of any given type (shown in the column labels) and a subsequent reference of any given type (shown in the row labels). The definition of first and second is defined by the order they are received by the SRAM controller. (Note: A given IXP version may implement a superset of these order guarantees. However, that superset is not promised to be supported in future implementations. Verification is required to test only the order rules shown in Table 12 and Table 13).

Note: Note that a blank entry means no order is enforced.

Table 12. Address Reference Order

<div> <div>1st ref →</div> <div>2nd ref ↓</div> </div>	Memory Read	CSR Read	Memory Write	CSR Write	Memory RMW	Queue / Ring / Q_Descr Commands
Memory Read			Order			
CSR Read				Order		
Memory Write			Order			
CSR Write				Order		
Memory RMW					Order	
Queue / Ring / Q_Descr Commands						See Table 13.

Table 13 shows the architectural guarantees of order to access to the SAME SRAM Q_array entry between a reference of any given type (shown in the column labels) and a subsequent reference of any given type (shown in the row labels). The definition of first and second is defined by the order they are received by the SRAM controller. The same caveats apply as for Table 12.

Table 13. Q_array Entry Reference Order

1 st ref → 2 nd ref ↓	Read_Q_Descr head, tail	Read_Q_Descr other	Write_Q_Descr	Enqueue	Dequeue	Put	Get	Journal
Read_Q_Descr head,tail			Order					
Read_Q_Descr other	Order							
Write_Q_Descr								
Enqueue	Order	Order		Order				
Dequeue	Order	Order			Order			
Put						Order		
Get							Order	
Journal								Order

2.5.5.2 Microengine Software Restrictions to Maintain Ordering

It is the Microengine programmer's job to insure order where the program flow finds order to be necessary and where the architecture does not guarantee that order. The signaling mechanism can be used to do this. For example, say that ucode needs to update several locations in a table. A location in SRAM is used to "lock" access to the table. [Example 12](#) is the code for the table update.

Example 12. Table Update Code

```

IMMED [$xfer0, 1]
SRAM [write, $xfer0, flag_address, 0, 1], ctx_swap [SIG_DONE_2]
; At this point, the write to flag_address has passed the point of coherency. Do
the table updates.
SRAM [write, $xfer1, table_base, offset1, 2] , sig_done [SIG_DONE_3]
SRAM [write, $xfer3, table_base, offset2, 2] , sig_done [SIG_DONE_4]
CTX_ARB [SIG_DONE_3, SIG_DONE_4]
; At this point, the table writes have passed the point of coherency. Clear the
flag to allow access by other threads.
IMMED [$xfer0, 0]
SRAM [write, $xfer0, flag_address, 0, 1, ctx_swap [SIG_DONE_2]

```

Other rules:

- All accesses to atomic variables should be via read-modify-write instructions.
- If the flow must know that a write is completed (actually in the SRAM itself), follow the write with a read to the same address. The write is guaranteed to be complete when the read data has been returned to the Microengine.



- With the exception of initialization, never do WRITE commands to the first 3 longwords of a queue_descriptor data structure (these are the longwords that hold head, tail, and count, etc.). All accesses to this data must be via the Q commands.
- To initialize the Q_array registers, perform a memory write of at least 3 longwords, followed by a memory read to the same address (to guarantee that the write completed). Then, for each entry in the Q_array, perform a read_q_descriptor_head followed by a read_q_descriptor_other using the address of the same 3 longwords.

2.6 Scratchpad Memory

The IXP2800 Network Processor contains a 16KB Scratchpad Memory, organized as 4K 32-bit words, that is accessible by Microengines and the Intel XScale® core.

The Scratchpad Memory provides the following operations:

- Normal reads and writes. From one to sixteen 32-bit words can be read/written with a single Microengine instruction. Note that Scratchpad is not byte-writeable (each write must write all four bytes).
- Atomic read-modify-write operations, bit-set, bit-clear, increment, decrement, add, subtract, and swap. The RMW operations can also optionally return the pre-modified data.
- Sixteen Hardware Assisted Rings for interprocess communication. [A ring is a FIFO that uses a head and tail pointer to store/read information in Scratchpad memory.]

Scratchpad Memory is provided as a third memory resource (in addition to SRAM and DRAM) that is shared by the Microengines and the Intel XScale® core. The Microengines and the Intel XScale® core can distribute memory accesses between these three types of memory resources to provide a greater number of memory accesses occurring in parallel.

2.6.1 Scratchpad Atomic Operations

In addition to normal reads and writes, the Scratchpad Memory supports the following atomic operations. Microengines have specific instructions to do each atomic operation; the Intel XScale® microarchitecture uses aliased address regions to do atomic operations.

- bit set
- bit clear
- increment
- decrement
- add
- subtract
- swap

The Scratchpad Memory does read-modify-writes for the atomic operations, the pre-modified data can also be returned if desired. The atomic operations operate on a single 32-bit word.



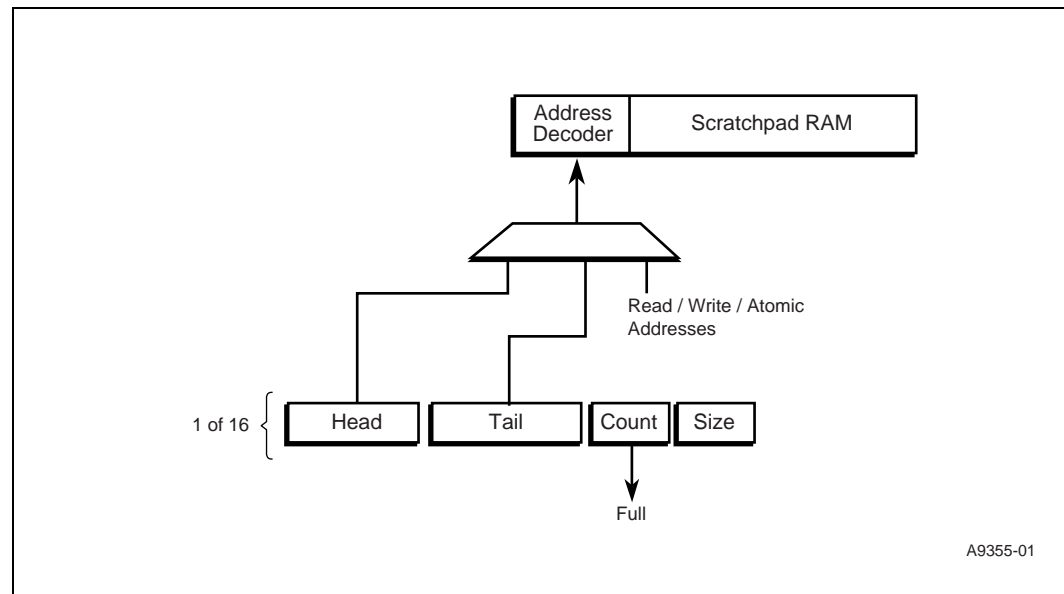
2.6.2 Ring Commands

The Scratchpad Memory provides sixteen Rings used for interprocess communication. The rings provide two operations.

- Get (ring, length)
- Put (ring, length)

Ring is the number of the ring (0 through 15) to get or put from, and length specifies the number of 32-bit words to transfer. A logical view of one of the rings is shown in Table 9.

Figure 9. Logical View of Rings



Head, Tail, and Size are registers in the Scratchpad Unit. Head and Tail point to the actual ring data, which is stored in the Scratchpad RAM. The count of how many entries are on the Ring is determined by hardware using the Head and Tail. For each Ring in use, a region of Scratchpad RAM must be reserved for the ring data.

Note: The reservation is by software convention. The hardware does not prevent other accesses to the region of Scratchpad Memory used by the Ring. Also the regions of Scratchpad Memory allocated to different Rings must not overlap.

Head points to the next address to be read on a get, and Tail points to the next address to be written on a put. The size of each Ring is selectable from the following choices: 128, 256, 512 or 1024 32-bit words.

Note: The region of Scratchpad used for a Ring is naturally aligned to its size.

When the Ring is near full, it asserts an output signal, which is used as a state input to the Microengines. They must use that signal to test (by doing Branch on Input State) for room on the Ring before putting data onto it. There is a lag in time from a put instruction executing to the Full signal being updated to reflect that put. To guarantee that a put will not overfill the ring there is a bound on the number of Contexts and the number of 32-bit words per write based on the size of the

ring, as shown in Table 14. Each Context should test the Full signal, then do the put if not Full, and then wait until the Context has been signaled that the data has been pulled before testing the Full signal again.

An alternate usage method is to have Contexts allocate and deallocate entries from a shared count variable, using the atomic subtract to allocate and atomic add to deallocate. In this case the Full signal is not used.

Table 14. Ring Full Signal Use -- Number of Contexts and Length vs Ring Size

Number of Contexts	Ring Size			
	128	256	512	1024
1	16	16	16	16
2	16	16	16	16
4	8	16	16	16
8	4	12	16	16
16	2	6	14	16
24	1	4	9	16
32	1	3	7	15
40	Illegal	2	5	12
48	Illegal	2	4	10
64	Illegal	1	3	7
128	Illegal	Illegal	1	3
NOTES: 1. Number in each table entry is the largest length that should be put. 16 is the largest length that a single put instruction can generate. 2. Illegal -- With that number of Contexts, even a length of one could cause the Ring to overflow.				

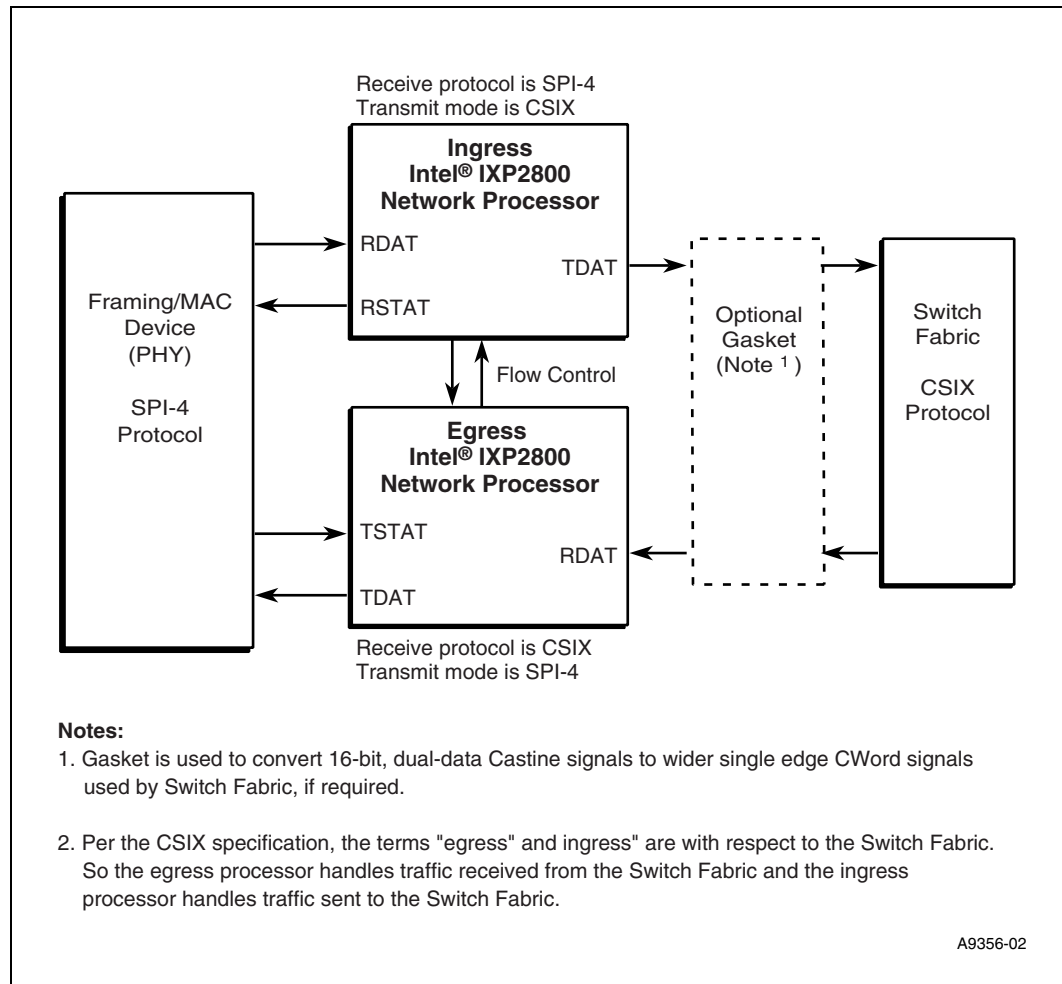
2.7 Media and Switch Fabric Interface

The Media and Switch Fabric (MSF) Interface is used to connect the IXP2800 Network Processor to a physical layer device (PHY) and/or to a Switch Fabric. the MSF consists of separate receive and transmit interfaces. Each of the receive and transmit interfaces can be separately configured for either SPI-4 Phase 2 (System Packet Interface) for PHY devices or CSIX-L1 protocol for Switch Fabric Interfaces.

The receive and transmit ports are unidirectional and independent of each other. Each port has 16 data signals, a clock, a control signal, and a parity signal, all of which use LVDS (differential) signaling, and are sampled on both edges of the clock. There is also a flow control port consisting of a clock, data, and ready status bits, and used to communicate between two IXP2800 Network Processors, or a the IXP2800 Network Processor chip a Switch Fabric Interface. These are also LVDS, dual-edge data transfer. All of the high speed LVDS interfaces support dynamic deskew training.

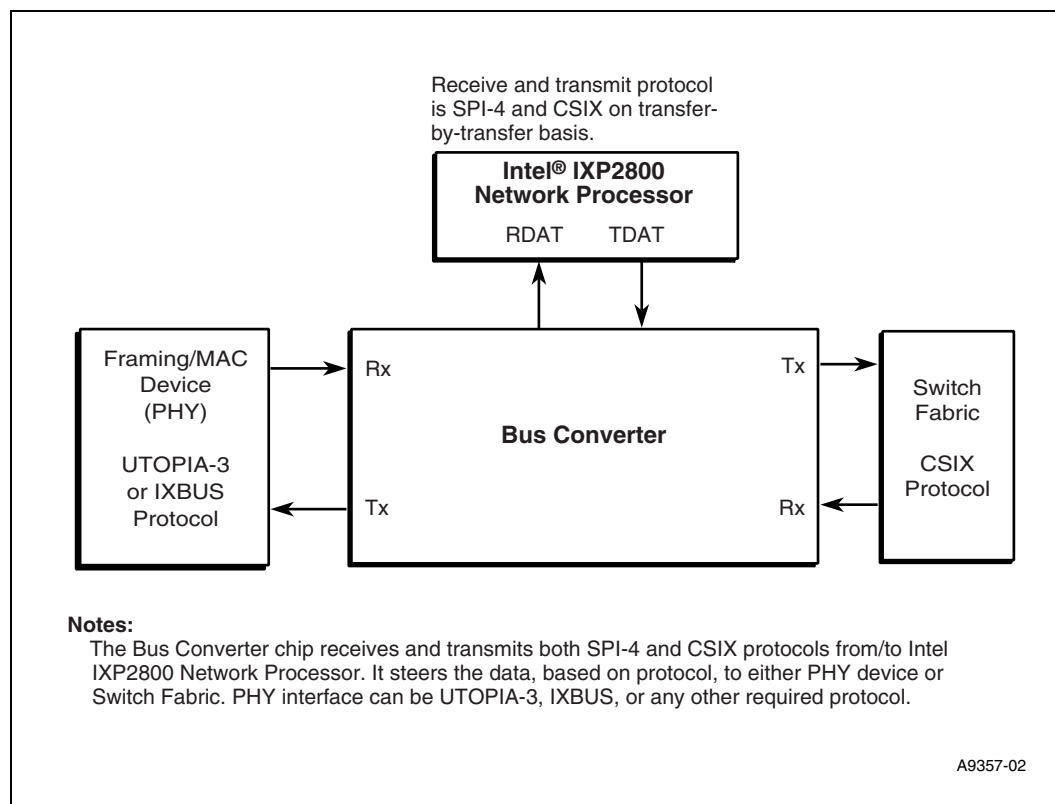
The block diagram in Figure 10 shows a typical configuration.

Figure 10. Example System Block Diagram



An alternate system configuration is shown in the block diagram in [Figure 11](#). In this case a single IXP2800 Network Processor is used for both Ingress and Egress. The bit rate supported would be less than in [Figure 10](#). A hypothetical Bus Converter chip, external to the IXP2800 Network Processor is used. The block diagram in [Figure 11](#) is only an illustrative example.

Figure 11. Full-Duplex Block Diagram



2.7.1 SPI-4

SPI-4 is an interface for packet and cell transfer between a physical layer (PHY) device and a link layer device (the IXP2800 Network Processor), for aggregate bandwidths of OC-192 ATM and Packet over SONET/SDH (POS), as well as 10 Gb/s Ethernet applications.

The Optical Internetworking Forum (OIF), www.oiforum.com, controls the SPI-4 Implementation Agreement document.

SPI-4 protocol transfers data in variable length bursts. Associated with each burst is information such as Port number (for a multi-port device such as a 10 x 1 GbE), SOP, EOP. This information is collected by MSF and passed to Microengines.

2.7.2 CSIX

CSIX-L1 (Common Switch Interface) defines an interface between a Traffic Manager (TM) and a Switch Fabric (SF) for ATM, IP, MPLS, Ethernet, and similar data communications applications.

The Network Processor Forum (NPF) www.npforum.org, controls the CSIX-L1 specification.

The basic unit of information transferred between Traffic Managers and Switch Fabrics is called a CFrame. There are three categories of Cframes:

- Data
- Control
- Flow Control

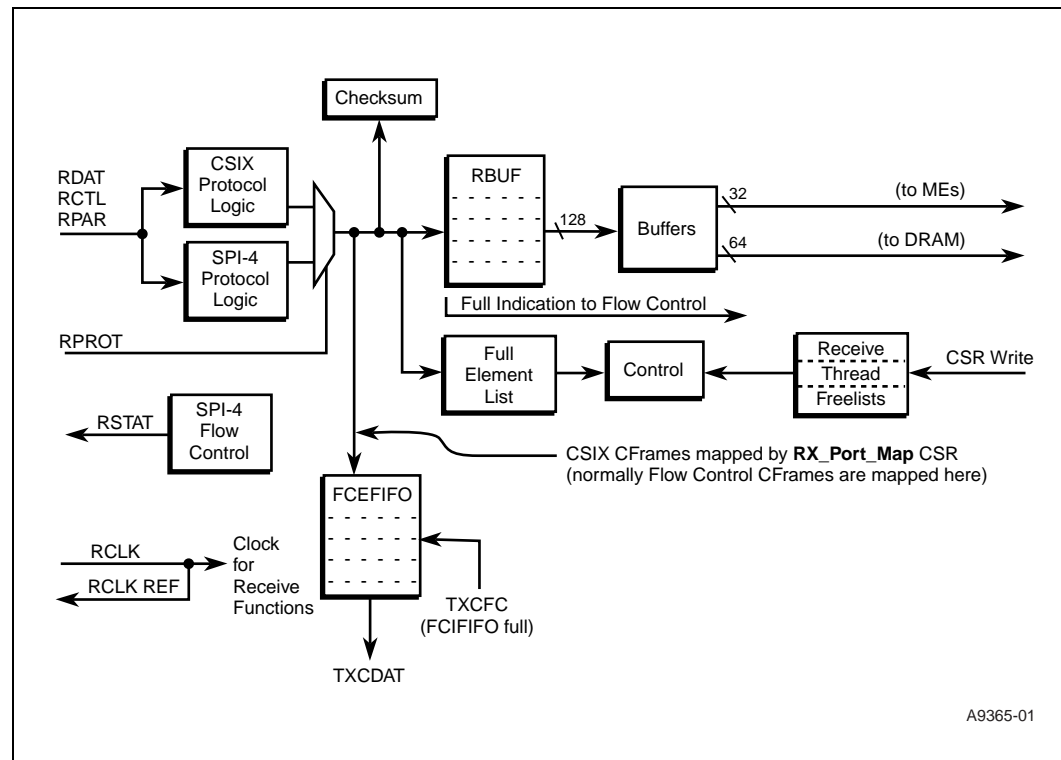
Associated with each CFrame is information such as length, type, address. This information is collected by MSF and passed to Microengines.

MSF also contains a number of hardware features related to flow control.

2.7.3 Receive

Figure 12 is a simplified block diagram of the MSF receive section.

Figure 12. Simplified MSF Receive Section Block Diagram



2.7.3.1 RBUF

RBUF is a RAM that holds received data. It stores received data in sub-blocks (referred to as elements), and is accessed by Microengines or the Intel XScale® core reading the received information. Details of how RBUF elements are allocated and filled is based on the receive data protocol. When data is received the associated status is put into the FULL_ELEMENT_LIST FIFO and subsequently sent to Microengines to process. FULL_ELEMENT_LIST insures that received elements are sent to Microengines in the order that the data was received.

RBUF contains a total of 8KB of data. The element size is programmable as either 64 bytes, 128 bytes, or 256 bytes per element. In addition, RBUF can be programmed to be split into one, two, or three partitions depending on application. For receiving SPI-4, one partition would be used. For receiving CSIX, two partitions are used (Control CFrames and Data CFrames). When both protocols are being used, the RBUF can be split into three partitions. For both SPI-4 and CSIX, three partitions are used.

Microengines can read data from the RBUF to Microengine S_Transfer_In registers using the `msf[read]` instruction where they specify the starting byte number (which must be aligned to 4 bytes), and number of 32-bit words to read. The number in the instruction can be either the number of 32-bit words, or number of 32-bit word pairs, using the single and double instruction modifiers, respectively.

Microengines can move data from RBUF to DRAM using the `dram` instruction where they specify the starting byte number (which must be aligned to 4 bytes), the number of 32-bit words to read, and the address in DRAM to write the data.

For both types of RBUF read, reading an element does not modify any RBUF data, and does not free the element, so buffered data can be read as many times as desired. This allows, for example, a processing pipeline to have different Microengines handle different protocol layers, with each Microengine reading only the specific header information it requires.

2.7.3.1.1 SPI-4 and the RBUF

SPI-4 data is placed into RBUF with each SPI-4 burst allocating an element. If a SPI-4 burst is larger than the element size, another element is allocated. The status information for the element contains the following information:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
RPROT	Element							Byte Count							SOP	EOP	Err	Len Err	Par Err	Abort Err	Null	Type	ADR								

6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
Reserved																Checksum															

The definitions of the fields are shown in [Table 15](#).

Table 15. RBUF SPI-4 Status Definition

Field	Definition
RPROT	This bit is a 0 indicating that the Status is for SPI-4. It is derived from the RPROT input signal.
Null	Null receive. If this bit is set, it means that the Rx_Thread_Freelist timeout expired before any more data was received, and that a null Receive Status Word is being pushed in order to keep the receive pipeline flowing. The rest of the fields in the Receive Status Word must be ignored; there is no data or RBUF entry associated with a null Receive Status Word.
ADR	The port number to which the data is directed. This field is taken from the ADR field of the Control Word that most recently preceded the data transfer.
Type	This field is taken from the Type field of the Control Word that most recently preceded the data transfer.
SOP	Indicates if the element is the start of a packet. This field is taken from the SOP field of the Control Word that most recently preceded the data transfer for the first element allocated after a Control Word. For subsequent elements (i.e. if more than one element worth of data follow the Control Word) this value is 0.
EOP	Indicates if the element is the end of a packet. This field is taken from the EOPS field of the Control Word that most recently succeeded the data transfer.
Byte_Count	Indicates the number of Data bytes, from 1 to 256, in the element (value 0x00 means 256). This field is derived from the number of data transfers that fill the element, and also the EOPS field of the Control Word that most recently succeeded the data transfer.
Element	The element number in the RBUF that holds the data. This is equal to the offset in RBUF of the first byte in the element, shifted right by 6 places
Par Err	Parity Error was detected in the DIP-4 parity field.
Length Err	A non-EOP burst occurred that was not a multiple of 16 bytes.
Abort Err	An EOP with Abort was received on bits[14:13] of the Control Word that most recently succeeded the data transfer.
Err	Error. This is the logical OR of Par Err, Length Err, and Abort Err.
Checksum	Checksum calculated over the Data Words in the element. This can be used for TCP.

2.7.3.1.2 CSIX and RBUF

CSIX CFrames are placed into either RBUF with each CFrame allocating an element. Unlike SPI-4, a single CFrame must not spill over into another element. Since CSIX spec specifies a maximum CFrame size of 256 bytes, this can be done by programming the element size to 256 bytes. However, if the Switch Fabric uses a smaller CFrame size, then a smaller RBUF element size can be used.

Flow Control CFrames are put into the FCEFIFO, to be sent to the Ingress IXP2800 Network Processor where a Microengine will read them to manage flow control information to the Switch Fabric.

The status information for the element contains the following information:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
RPROT	Element							Payload Length							CR	P	Err	Len Err	HP Err	VP Err	Null	Reserved					Type				
6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
Extension Header																															

The definitions of the fields are shown in [Table 16](#).

Table 16. RBUF CSIX Status Definition

Field	Definition
RPROT	This bit is a 1 indicating that the Status is for CSIX-L1. It is derived from the RPROT input signal.
Null	Null receive. If this bit is set, it means that the RX_THREAD_FREELIST timeout expired before any more data was received, and that a null Receive Status Word is being pushed in order to keep the receive pipeline flowing. The rest of the fields in the Receive Status Word must be ignored; there is no data or RBUF entry associated with a null Receive Status Word.
Type	Type Field from the CSIX Base Header
Payload Length	Payload Length Field from the CSIX Base Header. A value of 0x0 indicates 256 bytes.
VP Err	Vertical Parity Error was detected on the CFrame.
HP Err	Horizontal Parity Error was detected on the CFrame.
Length Err	Length Error; either amount of Payload received (before receipt of next Base Header) did not match value indicated in Base Header Payload Length field) or Payload Length field was greater than size of RBUF element.
Err	Error. This is the logical OR of VP Err, HP Err, and Length Err.
Element	The element number in the RBUF that holds the data. This is equal to the offset in RBUF of the first byte in the element, shifted right by 6 places.
CR	CR (CSIX Reserved) bit from the CSIX Base Header.
P	P (Private) bit from the CSIX Base Header.
Extension Header	The Extension Header from the CFrame. The bytes are received in big-endian order; byte 0 is in bits 63:56, byte 1 is in bits 55:48, byte 2 is in bits 47:40, and byte 3 is in bits 39:32.

2.7.3.2 Full Element List

Receive control hardware maintains the **FULL_ELEMENT_LIST** to hold the status of valid RBUF elements, in the order in which they were received. When an RBUF element is filled its status is added to the tail of the **FULL_ELEMENT_LIST**. When a Microengine is notified of element arrival (by having the status written to its **S_Transfer** register, it is removed from the head of the **FULL_ELEMENT_LIST**.

2.7.3.3 RX_THREAD_FREELIST

RX_THREAD_FREELIST is a FIFO that indicates Microengine Contexts that are awaiting an RBUF element to process. This allows the Contexts to indicate their ready status prior to the reception of the data, as a way to eliminate latency. Each entry added to a Freelist also has an associated S_TRANSFER register and signal number. There are three RX_THREAD_FREELISTS which correspond to the RBUF partitions.

To be added as ready to receive an element, a Microengine does a `msf[write]` or `msf[fast_write]` to the RX_THREAD_FREELIST address; the write data is the Microengine/CONTEXT/S_TRANSFER Register number to add to the Freelist.

When there is valid status at the head of the Full Element List it will be pushed to a Microengine. The receive control logic pushes the status information (which includes the element number) to the Microengine in the head entry of RX_THREAD_FREELIST, and sends an Event Signal to the Microengine. It then removes that entry from the RX_THREAD_FREELIST, and removes the status from Full Element List.

Each RX_THREAD_FREELIST has an associated countdown timer. If the timer expires and no new receive data is available yet, the receive logic will autopush a Null Receive Status Word to the next thread on the RX_THREAD_FREELIST. A Null Receive Status Word has the “Null” bit set, and does not have any data or RBUF entry associated with it.

The RX_THREAD_FREELIST timer is useful for certain applications. Its primary purpose is to keep the receive processing pipeline (implemented as code running on the Microengines) moving even when the line has gone idle.

It is especially useful if the pipeline is structured to handle mpackets in groups, i.e. eight mpackets at a time. If seven mpackets are received, then the line goes idle, then the timeout will trigger the autopush of a null Receive Status Word, filling the eighth slot and allowing the pipeline to advance. Another example is if one valid mpacket is received before the line goes idle for a long period; seven null Receive Status Words will be autopushed, allowing the pipeline to proceed. Typically the timeout interval is programmed to be slightly larger than the minimum arrival time of the incoming cells or packets.

The timer is controlled using the RX_THREAD_FREELIST_TIMEOUT_# CSR. The timer may be enabled or disabled, and the timeout value specified using this CSR.

2.7.3.4 Receive Operation Summary

During receive processing received Cframes, and SPI-4 cells and packets (which in this context are all called mpackets) are placed into the RBUF, and then handed off to a Microengine to process. Normally, by application design, some number of Microengine Contexts will be assigned to receive processing. Those Contexts will have their number added to the proper RX_THREAD_FREELIST (via `msf[write]` or `msf[fast_write]`), and then will go to sleep to wait for arrival of an mpacket (or alternatively poll waiting for arrival of an mpacket).

When an mpacket arrives, MSF receive control logic will autopush 8 bytes of information for the element to the Microengine/CONTEXT/S_TRANSFER Registers at the head of RX_THREAD_FREELIST. The information pushed is (see [Table 15](#) and [Table 16](#) for detailed definitions):

- Status Word (SPI-4) or Header Status (CSIX)
- Checksum (SPI-4) or Extension Header (CSIX)

To handle the case where the receive Contexts temporarily fall behind and RX_THREAD_FREELIST is empty, all received element numbers are held in the FULL_ELEMENT_LIST. In that case, as soon as an RX_THREAD_FREELIST entry is entered, the status of the head element of FULL_ELEMENT_LIST will be pushed to it.

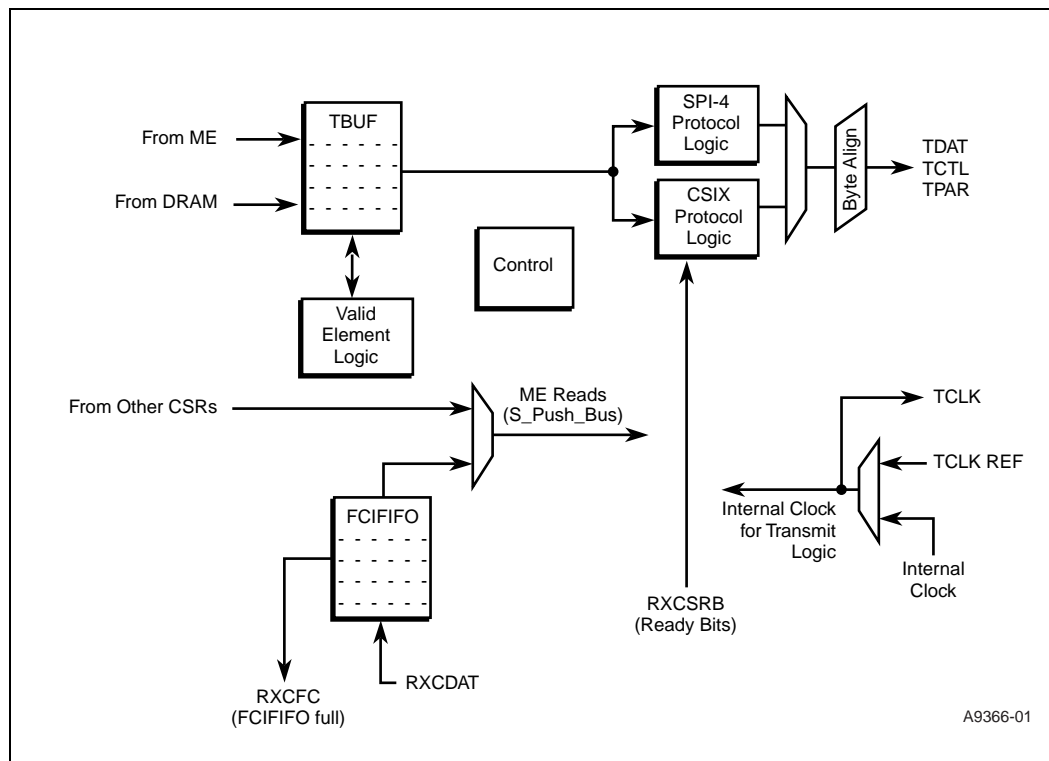
The Microengines may read part of (or the entire) RBUF element to their S_TRANSFER registers (via `msf[read]` instruction) for header processing, etc., and may also move the element data to DRAM (via `dram[rbuf_rd]` instruction).

When a Context is done with an element it does a `msf[write]` or `msf[fast_write]` to RBUF_ELEMENT_DONE address; the write data is the element number. This marks the element as free and available to be re-used. There is no restriction on the order in which elements are freed; Contexts can do different amounts of processing per element based on the contents of the element—therefore elements can be returned in a different order than they were handed to Contexts.

2.7.4 Transmit

Figure 13 is a simplified Block Diagram of the MSF transmit section.

Figure 13. Simplified Transmit Section Block Diagram





2.7.4.1 TBUF

TBUF is a RAM that holds data and status to be transmitted. The data is written into sub-blocks referred to as elements, by Microengines or the Intel XScale® core.

TBUF contains a total of 8KB of data. The element size is programmable as either 64 bytes, 128 bytes, or 256 bytes per element. In addition, TBUF can be programmed to be split into one, two, or three partitions depending on application. For transmitting SPI-4, one partition would be used. For transmitting CSIX, two partitions are used (Control CFrames and Data CFrames). For both SPI-4 and CSIX, three partitions are used.

Microengines can write data from Microengine S_TRANSFER_OUT registers to the TBUF using the `msf[write]` instruction where they specify the starting byte number (which must be aligned to 4 bytes), and number of 32-bit words to write. The number in the instruction can be either the number of 32-bit words, or number of 32-bit word pairs, using the single and double instruction modifiers, respectively.

Microengines can move data from DRAM to TBUF using the `dram` instruction where they specify the starting byte number (which must be aligned to 4 bytes), the number of 32-bit words to write, and the address in DRAM of the data.

All elements within a TBUF partition are transmitted in the order. Control information associated with the element defines which bytes are valid. The data from the TBUF will be shifted and byte aligned as required to be transmitted.

2.7.4.1.1 SPI-4 and TBUF

For SPI-4, data is put into the data portion of the element, and information for the SPI-4 Control Word that will precede the data is put into the Element Control Word.

When the Element Control Word is written the information is:

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0	
Payload Length									Prepend Offset		Prepend Length					Payload Offset		Res	SKIP	Res	SOP	EOP	ADR							

6	6	6	6	5	5	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	2
3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2
Res																															

The definitions of the fields are shown in [Table 17](#).

Table 17. TBUF SPI-4 Control Definition

Field	Definition
ADR	The port number to which the data is directed. This field will be sent in the ADR field of the Control Word that will precede the data transfer.
SOP	Indicates if the element is the start of a packet. This field will be sent in the SOPC field of the Control Word that will precede the data transfer.
EOP	Indicates if the element is the end of a packet. This field will be sent in the EOPS field of the Control Word that will succeed the data transfer. Note 1.
Prepend Offset	Indicates the first valid byte of Prepend, from 0 to 7
Prepend Length	Indicates the number of bytes in Prepend, from 0 to 31.
Payload Offset	Indicates the first valid byte of Payload, from 0 to 7.
Payload Length	Indicates the number of Payload bytes, from 1 to 256, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent. That value will also control the EOPS field (1 or 2 bytes valid indicated) of the Control Word that will succeed the data transfer. Note 1.
Skip	Allows software to allocate a TBUF element and then not transmit any data from it. 0—transmit data according to other fields of Control Word. 1—free the element without transmitting any data.
NOTE: 1. Normally EOPS is sent on the next Control Word (along with ADR and SOP) to start the next element. If there is no valid element pending at the end of sending the data, the transmit logic will insert an Idle Control Word with the EOPS information.	

2.7.4.1.2 CSIX and TBUF

For CSIX, payload information is put into the data area of the element, and Base and Extension Header information is put into the Element Control Word.

When the Element Control Word is written the information is:

3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	0	9	8	7	6	5	4	3	2	1	0
Payload Length								Prepend Offset		Prepend Length				Payload Offset		Res	Skip	Res	CR	P	Res				Type					

6	6	6	6	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	2
Extension Header																															

The definitions of the fields are shown in Table 18.

Table 18. TBUF CSIX Control Definition (Sheet 1 of 2)

Field	Definition
Type	Type Field to put into the CSIX Base Header. Idle type is <i>not</i> legal here.
CR	CR (CSIX Reserved) bit to put into the CSIX Base Header.
P	P (Private) bit to put into the CSIX Base Header.

Table 18. TBUF CSIX Control Definition (Sheet 2 of 2)

Field	Definition
Extension Header	The Extension Header to be sent with the CFrame. The bytes are sent in big-endian order; byte 0 is in bits 63:56, byte 1 is in bits 55:48, byte 2 is in bits 47:40, and byte 3 is in bits 39:32.
Prepend Offset	Indicates the first valid byte of Prepend, from 0 to 7.
Prepend Length	Indicates the number of bytes in Prepend, from 0 to 31.
Payload Offset	Indicates the first valid byte of Payload, from 0 to 7.
Payload Length	Indicates the number of Payload bytes, from 1 to 256, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent, and also put into the CSIX Base Header Payload Length field. Note that this length does not include any padding which may be required. Padding is inserted by transmit hardware as needed.
Skip	Allows software to allocate a TBUF element and then not transmit any data from it. 0—transmit data according to other fields of Control Word. 1—free the element without transmitting any data.

2.7.4.2 Transmit Operation Summary

During transmit processing data to be transmitted is placed into the TBUF under Microengine control. The Microengine allocates an element in software; the transmit hardware processes TBUF elements within a partition in strict sequential order so the software can track which element to allocate next.

Microengines may write directly into an element by `msf[write]` instruction, or have data from DRAM written into the element by `dram[tbuf_wr]` instruction. Data can be merged into the element by doing both.

There is a Transmit Valid bits per element, which marks the element as ready to be transmitted. Microengines move all data into the element, by either or both of `msf[write]` and `dram[tbuf_wr]` instructions to the TBUF. Microengines also write the element Transmit Control Word with information about the element. When all the data movement is complete the Microengine sets the element valid bit.

1. Move data into TBUF by either or both of `msf[write]` and `dram[tbuf_wr]` instructions to the TBUF.
2. Wait for 1 to complete.
3. Write Transmit Control Word at `TBUF_ELEMENT_CONTROL_#` address. Using this address sets the Transmit Valid bit.

2.7.5 The Flow Control Interface

The MSF provides flow control support for SPI-4 and CSIX.

2.7.5.1 SPI-4

SPI-4 uses a FIFO Status Channel to provide flow control information. MSF receives the information from the PHY device and stores it so that Microengines can read the information on a per-port basis. It can then use that information to determine when to transmit data to a given port.

The MSF also sends status to the PHY based on the amount of available space in the RBUF; that is done by hardware without Microengines.

2.7.5.2 CSIX

CSIX provides two types of flow control -- link level and per queue.

The link level control is handled by hardware. MSF will stop transmission in response to link level flow control received from the Switch Fabric. MSF will assert link level flow control based on the amount of available space in the RBUF.

Per queue flow control information is put into the FCIFIFO and handled by Microengine software. Also, if required, Microengines can send Flow Control CFrames to the Switch Fabric under software control.

In both cases, for a full duplex configuration, information is passed from the Switch Fabric to the Egress IXP2800 Network Processor, which then passes it to the Ingress IXP2800 Network Processor over a proprietary flow control interface.

2.8 Hash Unit

The IXP2800 Network Processor contains a Hash Unit that can take 48-bit, 64-bit, or 128-bit data and produces a 48-bit, a 64-bit, or a 128-bit hash index, respectively. The Hash Unit is accessible by the Microengines and the Intel XScale® core, and is useful in doing table searches with large keys, for example L2 addresses. [Figure 14](#) is a block diagram of the Hash Unit.

Up to three hash indexes can be created using a single Microengine instruction. This helps to minimize command overhead. The Intel XScale® core can only do a single hash at a time.

A Microengine initiates a hash operation by writing the hash operands into a contiguous set of S TRANSFER OUT Registers and then executing the hash instruction. The Intel XScale® core initiates a hash operation by writing a set of memory-mapped HASH_OP Registers, which are built in the Intel XScale® core gasket, with the data to be used to generate the hash index. There are separate registers for 48-bit, 64-bit, and 128-bit hashes. The data is written from MSB to LSB, with the write to LSB triggering the Hash Operation. In both cases, the Hash Unit reads the operand into an input buffer, performs the hash operation, and returns the result.

The Hash Unit uses a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indexes. Three separate multipliers are supported, one for 48-bit hash operations, one for 64-bit hash operations and one for 128-bit hash operations. The multiplier is programmed through Control registers in the Hash Unit.

The multiplicand is shifted into the hash array sixteen bits at a time. The hash array performs a ones-complement multiply and polynomial divide, calculated using the multiplier and 16 bits of the multiplicand. The result is placed into an output buffer register and also feeds back into the array. This process is repeated three times for a 48-bit hash (16 bits x 3 = 48), four times for a 64-bit hash (16 bits x 4 = 64), and eight times for a 128-bit hash (16 x 8 = 128). After an entire multiplicand has been passed through the hash array, the resulting hash index is placed into a two-stage output buffer.

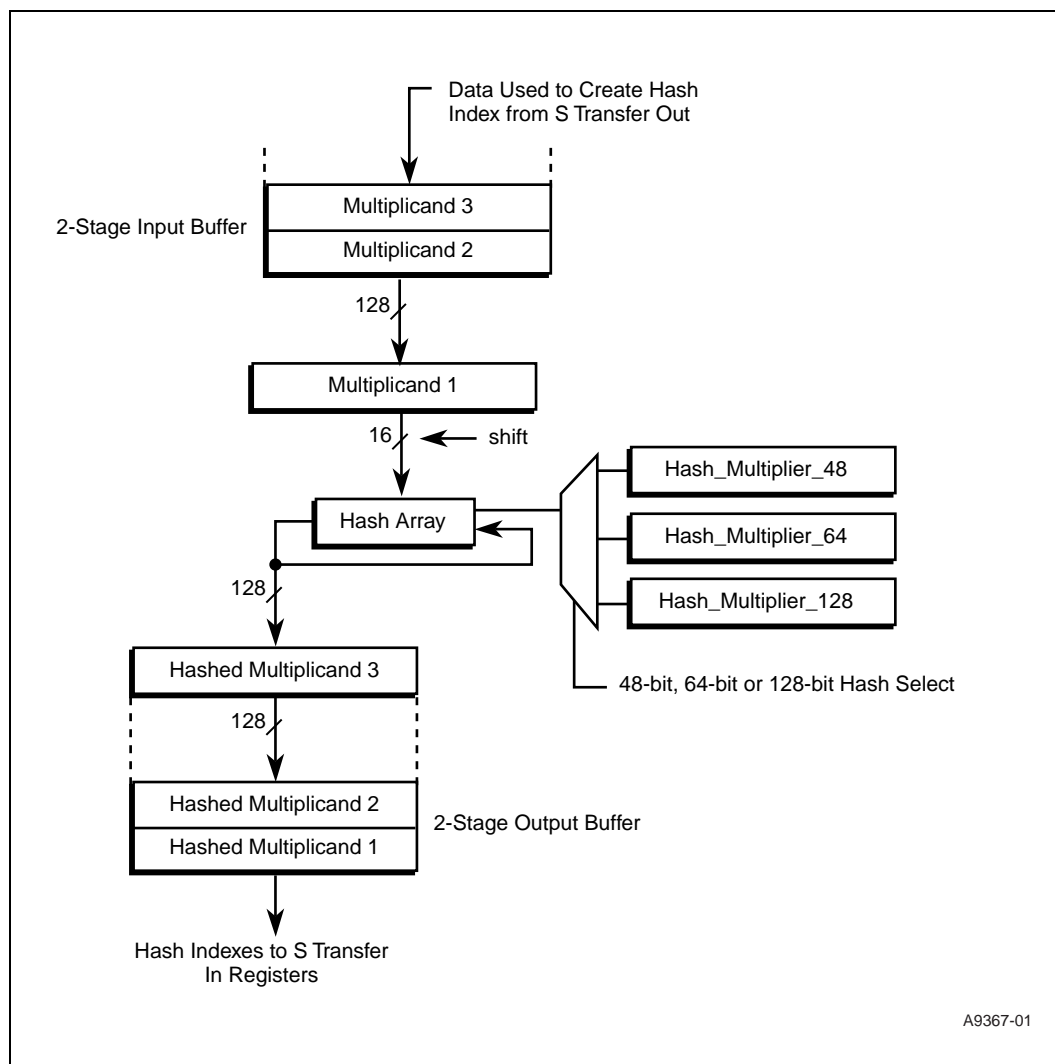
After each hash index is completed, the Hash Unit returns the hash index to the Microengines S Transfer In Registers, or the Intel XScale® core HASH_OP Registers. For Microengine initiated hash operations, the Microengine is signaled after all the hashes specified in the instruction have been completed.

For the Intel XScale® core initiated hash operations, the Intel XScale® core reads the results from the memory-mapped HASH_OP Registers. The addresses of Hash Results are the same as the HASH_OP Registers. Because of queuing delays at the Hash Unit, the time to complete an operation is not fixed. The Intel XScale® core can do one of two operations to get the hash results.

- Poll the HASH_DONE Register. This register is cleared when the HASH_OP Registers are written. Bit [0] of HASH_DONE Register is set when the HASH_OP Registers get the return result from the Hash Unit (when the last word of the result is returned). The Intel XScale® core software can poll on HASH_DONE, and read HASH_OP when HASH_DONE is equal to 0x00000001.
- Read HASH_OP directly. The interface hardware will acknowledge the read only when the result is valid. This method will result in the Intel XScale® core stalling if the result is not valid when the read happens.

The number of clock cycles required to perform a single hash operation equals: two or four cycles through the input buffers, three, four or eight cycles through the hash array, and two or four cycles through the output buffers. Because of the pipeline characteristics of the Hash Unit, performance is improved if multiple hash operations are initiated with a single instruction rather than separate hash instructions for each hash operation.

Figure 14. Hash Unit Block Diagram





2.9 PCI Controller

The PCI Controller provides a 64-bit, 66 MHz capable *PCI Local Bus Revision 2.2* interface. It is also compatible to 32-bit and/or 33 MHz PCI devices. The PCI controller provides the following functions:

- Target Access (external Bus Master access to SRAM, DRAM, and CSRs)
- Master Access (the Intel XScale® core access to PCI Target devices)
- Two DMA Channels
- Mailbox and Doorbell Registers for the Intel XScale® core to Host communication
- PCI arbiter

The IXP2800 Network Processor can be configured to act as PCI central function (for use in a stand-alone system), where it provides the PCI reset signal, or as an add-in device, where it uses the PCI reset signal as the chip reset input. The choice is made by connecting the `cfg_rst_dir` input pin low or high.

2.9.1 Target Access

There are three Base Address Registers (BARs) to allow PCI Bus Masters to access SRAM, DRAM, and CSRs, respectively. Examples of PCI Bus Masters include a Host Processor (for example a Pentium® processor), or an I/O device such as an Ethernet controller, SCSI controller, or encryption coprocessor.

The SRAM BAR can be programmed to sizes of 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, or no access.

The DRAM BAR can be programmed to sizes of 128 MB, 256 MB, 512 MB, or 1 GB, or no access.

The CSR BAR is 8 KB.

PCI Boot Mode is supported, in which the Host downloads the Intel XScale® core boot image into DRAM, while holding the Intel XScale® core in reset. Once the boot image has been loaded, the Intel XScale® core reset is deasserted. The alternative is to provide the boot image in a Flash ROM attached to the Slow Port.

2.9.2 Master Access

The Intel XScale® core and Microengines can directly access the PCI bus. The Intel XScale® core can do loads and stores to specific address regions to generate all PCI command types. Microengines use PCI instruction, and also use address regions to generate different PCI commands.



2.9.3 DMA Channels

There are two DMA Channels, each of which can move blocks of data from DRAM to the PCI or from the PCI to DRAM. The DMA channels read parameters from a list of descriptors in SRAM, perform the data movement to or from DRAM, and stop when the list is exhausted. The descriptors are loaded from predefined SRAM entries or may be set directly by CSR writes to DMA Channel registers. There is no restriction on byte alignment of the source address or the destination address. For PCI to DRAM transfers, the PCI command is Memory Read, Memory Read line, or Memory Read Multiple. For DRAM to PCI transfers, the PCI command is Memory Write. Memory Write Invalidate is not supported.

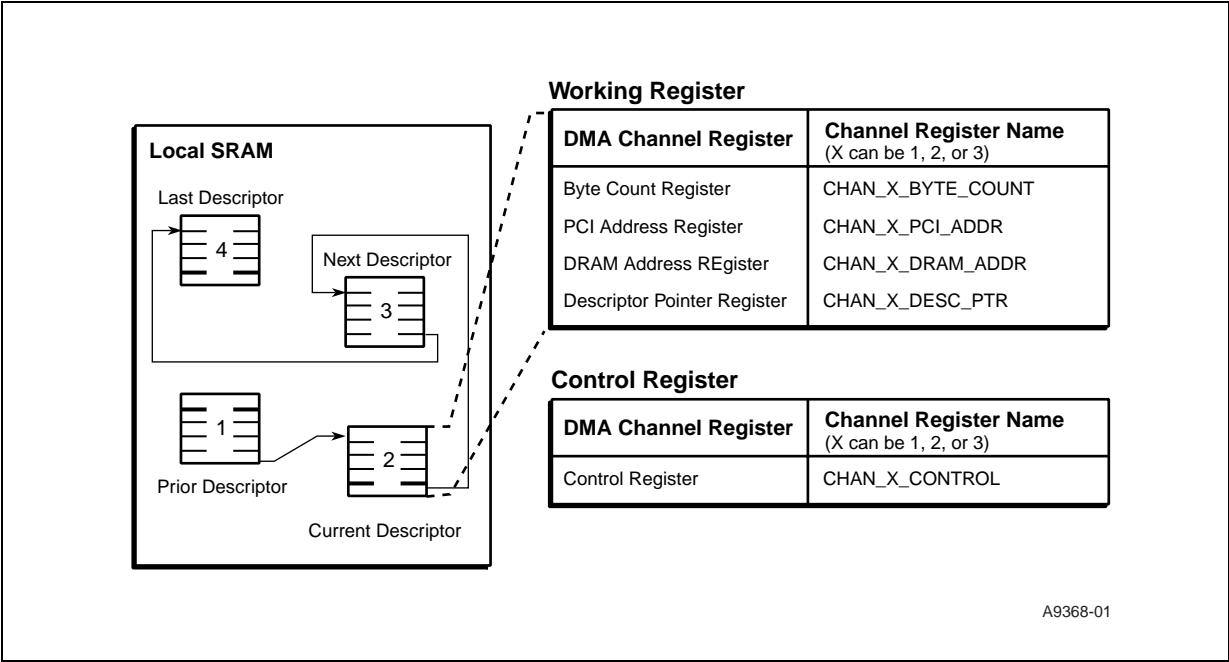
Up to two DMA channels are running at a time with three descriptors outstanding. Effectively, the active channels interleave bursts to or from the PCI Bus.

Interrupts are generated at the end of DMA operation for the Intel XScale® core. However, Microengines do not provide an interrupt mechanism. The DMA Channel will instead use an Event Signal to notify the particular Microengine on completion of DMA.

2.9.3.1 DMA Descriptor

Each descriptor occupies four 32-bit words in SRAM, aligned on a 16 byte boundary. The DMA channels read the descriptors from SRAM into working registers once the control register has been set to initiate the transaction. This control must be set explicitly. This starts the DMA transfer. The register names for the DMA channels are listed in Figure 15 and Table 19 lists the contents of the descriptor.

Figure 15. DMA Descriptor Reads



After a descriptor is processed, the next descriptor is loaded in the working registers. This process repeats until the chain of descriptors is terminated (i.e., the End of Chain bit is set).

Table 19. DMA Descriptor Format

Offset from Descriptor Pointer	Description
0x0	Byte Count
0x4	PCI Address
0x8	DRAM Address
0xC	Next Descriptor Address

2.9.3.2 DMA Channel Operation

The DMA channel can be set up to read the first descriptor in SRAM, or with the first descriptor written directly to the DMA channel registers.

When descriptors and the descriptor list are in SRAM, the procedure is as follows:

1. The DMA channel owner writes the address of the first descriptor into the DMA Channel Descriptor Pointer register (DESC_PTR).
2. The DMA channel owner writes the DMA Channel Control register (CONTROL) with miscellaneous control information and also sets the channel enable bit (bit 0). The channel initial descriptor bit (bit 4) in the CONTROL register must also be cleared to indicate that the first descriptor is in SRAM.
3. Depending on the DMA channel number, the DMA channel reads the descriptor block into the corresponding DMA registers, BYTE_COUNT, PCI_ADDR, DRAM_ADDR, and DESC_PTR.
4. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit in the CONTROL register.
5. If the end of chain bit (bit 31) in the BYTE_COUNT register is clear, the channel checks the Chain Pointer value. If the Chain Pointer value is not equal to 0, it reads the next descriptor and transfers the data (step 3 and 4 above). If the Chain Pointer value is equal to 0, it waits for the Descriptor Added bit of the Channel Control Register to be set before reading the next descriptor and transfers the data (step 3 and 4 above). If bit 31 is set, the channel sets the channel chain done bit in the CONTROL register and then stops.
6. Proceed to the Channel End Operation.

When single descriptors are written directly into the DMA channel registers, the procedure is as follows:

1. The DMA channel owner writes the descriptor values directly into the DMA channel registers. The end of chain bit (bit 31) in the BYTE_COUNT register must be set, and the value in the DESC_PTR register is not used.
2. The DMA channel owner writes the base address of the DMA transfer into the PCI_ADDR to specify the PCI starting address.
3. When the first descriptor is in the BYTE_COUNT register, the DRAM_ADDR register must be written with the address of the data to be moved.
4. The DMA channel owner writes the CONTROL register with miscellaneous control information, along with setting the channel enable bit (bit 0). The channel initial descriptor in register bit (bit 4) in the CONTROL register must also be set to indicate that the first descriptor is already in the channel descriptor registers.

5. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit (bit 2) in the CONTROL register.
6. Since the end of the chain bit (bit 31) in the BYTE_CONT register is set, the channel sets the channel chain done bit (bit 7) in the CONTROL register and then stops.
7. Proceed to the Channel End Operation.

2.9.3.3 DMA Channel End Operation

1. Channel owned by PCI
If not masked via the PCI Outbound Interrupt Mask register, the DMA channel interrupts the PCI host after the setting of the DMA done bit in the CHAN_X_CONTROL register, which is readable in the PCI Outbound Interrupt Status register.
2. Channel owned by the Intel XScale® core
If enabled via the Intel XScale® core Interrupt Enable registers, the DMA channel interrupts the Intel XScale® core by setting the DMA channel done bit in the CHAN_X_CONTROL register, which is readable in the Intel XScale® core Interrupt Status register.
3. Channel owned by Microengine
If enabled via the Microengine Auto-Push Enable registers, the DMA channel signals the Microengine after setting the DMA channel done bit in the CHAN_X_CONTROL register, which is readable in the Microengine Auto-Push Status register.

2.9.3.4 Adding Descriptors to an Unterminated Chain

It is possible to add a descriptor to a chain while a channel is running. To do so the chain should be left unterminated, that is the last descriptor should have End of Chain clear, and the Chain Pointer value equal to 0. A new descriptor (or linked list of descriptors) can be added to the chain by overwriting the Chain Pointer value of the unterminated descriptor (in SRAM) with the Local Memory address of the (first) added descriptor (Note that the added descriptor must actually be valid in Local Memory prior to that). After updating the Chain Pointer field, the software must write a 1 to the Descriptor Added bit of the Channel Control Register. This is necessary for the case where the channel was paused in order to re-activate the channel. However, software need not check the state of the channel before writing that bit; there is no side-effect of writing that bit in the case where the channel had not yet read the unlinked descriptor.

If the channel was paused or had read an unlinked Pointer, it will re-read the last descriptor processed (i.e. the one that originally had the zero value for Chain Pointer) to get the address of the newly added descriptor.

A descriptor can not be added to a descriptor which has End of Chain set.

2.9.4 Mailbox and Message Registers

Mailbox and Doorbell registers provide hardware support for communication between the Intel XScale® core and a device on the PCI Bus.

Four 32-bit mailbox registers are provided so that messages can be passed between the Intel XScale® core and a PCI device. All four registers can be read and written with byte resolution from both the Intel XScale® core and PCI. How the registers are used is application dependent and the messages are not used internally by the PCI Unit in any way. The mailbox registers are often used with the Doorbell interrupts.

Doorbell interrupts provide an efficient method of generating an interrupt as well as encoding the purpose of the interrupt. The PCI Unit supports a 32-bit the Intel XScale® core DOORBELL register that is used by a PCI device to generate an the Intel XScale® core interrupt, and a separate 32-bit PCI DOORBELL register that is used by the Intel XScale® core to generate a PCI interrupt. A source generating the Doorbell interrupt can write a software defined bitmap to the register to indicate a specific purpose. This bitmap is translated into a single interrupt signal to the destination (either a PCI interrupt or an the Intel XScale® core interrupt). When an interrupt is received, the DOORBELL registers can be read and the bit mask can be interpreted. If a larger bit mask is required than that is provided by the DOORBELL register, the MAILBOX registers can be used to pass up to 16 bytes of data.

The doorbell interrupts are controlled through the registers shown in [Table 20](#).

Table 20. Doorbell Interrupt Registers

Register Name	Description
XSCALE DOORBELL	Used to generate the Intel XScale® core Doorbell interrupts.
XSCALE DOORBELL SETUP	Used to initialize the Intel XScale® core Doorbell register and for diagnostics.
PCI DOORBELL	Used to generate the PCI Doorbell interrupts.
PCI DOORBELL SETUP	Used to initialize the PCI Doorbell register and for diagnostics.

2.9.5 PCI Arbiter

The PCI unit contains a PCI bus arbiter that supports two external masters in addition to the PCI Unit's initiator interface. If more than two external masters are used in the system, the arbiter can be disabled and an external (to the IXP2800 Network Processor) used. In that case, the IXP2800 Network Processor will provide its PCI request signal to the external arbiter, and use that arbiter's grant signal.

The arbiter uses a simple round-robin priority algorithm; it asserts the grant signal corresponding to the next request in the round-robin during the current executing transaction on the PCI bus (this is also called hidden arbitration). If the arbiter detects that an initiator has failed to assert frame `_1` after 16 cycles of both grant assertion and PCI bus idle condition, the arbiter deasserts the grant. That master does not receive any more grants until it deasserts its request for at least one PCI clock cycle. Bus parking is implemented in that the last bus grant will stay asserted if no request is pending.

To prevent bus contention, if the PCI bus is idle, the arbiter never asserts one grant signal in the same PCI cycle in which it deasserts another. It deasserts one grant, and then asserts the next grant after one full PCI clock cycle has elapsed to provide for bus driver turnaround.

2.10 Control and Status Register Access Proxy

The Control and Status Register Access Proxy (CAP) contains a number of chip-wide control and status registers. Some provide miscellaneous control and status, while others are used for inter-Microengine or Microengine to the Intel XScale® core communication (note that rings in Scratchpad Memory and SRAM can also be used for interprocess communication). These include:

- **INTERTHREAD SIGNAL**—Each thread (or context) on a Microengine can send a signal to any other thread by writing to InterThread_Signal register. This allows a thread to go to sleep waiting completion of a task by a different thread.
- **THREAD MESSAGE** —Each thread has a message register where it can post a software-specific message. Other Microengine threads, or the Intel XScale® core, can poll for availability of messages by reading THREAD_MESSAGE_SUMMARY register. Both the THREAD_MESSAGE and corresponding THREAD_MESSAGE_SUMMARY clear upon a read of the message; this eliminates a race condition when there are multiple message readers. Only one reader will get the message.
- **SELF DESTRUCT** —This register provides another type of communication. Microengine software can atomically set individual bits in the SELF DESTRUCT registers; the registers clear upon read. The meaning of each bit is software-specific. Clearing the register upon read eliminates a race condition when there are multiple readers.
- **THREAD INTERRUPT**—Each thread can interrupt the Intel XScale® core on two different interrupts; the usage is software-specific. Having two interrupts allows for flexibility, for example one can be assigned to normal service requests and one can be assigned to error conditions. If more information needs to be associated with the interrupt, mailboxes or Rings in Scratchpad Memory or SRAM could be used.
- **REFLECTOR**—CAP provides a function (called “reflector”) where any Microengine thread can move data between its registers and those of any other thread. In response to a single write or read instruction (with the address in the specific reflector range) CAP will get data from the source Microengine and put it into the destination Microengine. Both the sending and receiving threads can optionally be signalled upon completion of the data movement.

2.11 Intel XScale® Core Peripherals

2.11.1 Interrupt Controller

The Interrupt Controller provides the ability to enable or mask interrupts from a number of chip wide sources, for example:

- Timers (normally used by Real-Time Operating System).
- Interrupts generated by Microengine software to request services from the Intel XScale® core.
- External agents such as PCI devices.
- Error conditions, such as DRAM ECC error, or SPI-4 parity error.

Interrupt status is read as memory mapped registers; the state of an interrupt signal can be read even if it is masked from interrupting. Enabling and masking of interrupts is done as writes to memory mapped registers.



2.11.2 Timers

The IXP2800 Network Processor contains four programmable 32-bit timers, which can be used for software support. Each timer can be clocked by the internal clock, by a divided version of the clock, or by a signal on an external GPIO pin. Each timer can be programmed to generate a periodic interrupt after a programmed number of clocks. The range is from several ns to several minutes depending on the clock frequency.

In addition, timer 4 can be used as a watchdog timer. In this use, software must periodically reload the timer value; if it fails to do so and the timer counts to zero, it will reset the chip. This can be used to detect if software “hangs” or for some other reason fails to reload the timer.

2.11.3 General Purpose I/O

The IXP2800 Network Processor contains eight General Purpose I/O (GPIO) pins. These can be programmed as either input or output and can be used for slow speed I/O such as LEDs or input switches. They can also be used as interrupts to the Intel XScale® core, or to clock the programmable timers.

2.11.4 Universal Asynchronous Receiver/Transmitter

The IXP2800 Network Processor contains a standard RS-232 compatible Universal Asynchronous Receiver/Transmitter (UART), which can be used for communication with a debugger or maintenance console. Modem controls are not supported; if they are needed, GPIO pins can be used for that purpose.

The UART performs serial-to-parallel conversion on data characters received from a peripheral device and parallel-to-serial conversion on data characters received from the processor. The processor can read the complete status of the UART at any time during operation. Available status information includes the type and condition of the transfer operations being performed by the UART and any error conditions (parity, overrun, framing or break interrupt).

The serial ports can operate in either FIFO or non-FIFO mode. In FIFO mode, a 64-byte transmit FIFO holds data from the processor to be transmitted on the serial link and a 64-byte receive FIFO buffers data from the serial link until read by the processor.

The UART includes a programmable baud rate generator which is capable of dividing the internal clock input by divisors of 1 to $2^{16} - 1$ and produces a 16X clock to drive the internal transmitter logic. It also drives the receive logic. The UART can be operated in polled or in interrupt driven mode as selected by software.

2.11.5 Slow Port

The SlowPort is an external interface to the IXP2800 Network Processor, used for Flash ROM access and 8, 16, or 32-bit asynchronous device access. It allows the Intel XScale® core do read/write data transfers to these slave devices.

The address bus and data bus are multiplexed to reduce the pin count. In addition, 24 bits of address are shifted out on three clock cycles. Therefore, an external set of buffers is needed to latch the address. Two chip selects are provided.

The access is asynchronous. Insertion of delay cycles for both data setup and hold time is programmable via internal Control registers. The transfer can also wait for a handshake acknowledge signal from the external device.

2.12 I/O Latency

Table 21 shows the latencies for transferring data between the Microengine and the other sub-system components. The latency is measured in 1.4 GHz cycles.

Table 21. I/O Latency

	Sub-system			
	DRAM (RDR)	SRAM (QDR)	Scratch	MSF
Transfer size	8 bytes - 16 bytes (note 2)	4-bytes	4-bytes	8-bytes
average read latency	~295 cycles (note 3)	100(light load)-160(heavy load)	~100 cycles (range 53-152)	range 53-120 (RBUF)
average write latency	~53 cycles	~53 cycles	~40 cycles	~48 cycles (TBUF)
Note1: RDR, QDR, MSF, and Scratch values are extracted from a simulation model. Note 2: Minimum DRAM burst size on pins is 16 bytes. Transfers less than 16bytes incur the same as a 16 byte transfer. Note 3: At 1016 MHz, read latency should be ~ 240 cycles.				

2.13 Performance Monitor

The Intel® XScale™ core hardware provides two 32-bit performance counters that allow two unique events to be monitored simultaneously. In addition, the Intel® XScale™ core implements a 32-bit clock counter that can be used in conjunction with the performance counters; its sole purpose is to count the number of core clock cycles which is useful in measuring total execution time.

This section contains information describing the Intel XScale® core, Intel XScale® core gasket, and Intel XScale® core Peripherals (XPI).

For additional information about the Intel XScale® architecture refer to the *Intel XScale® Core Developers Manual* available on Intel's Developers website (<http://www.developer.intel.com>).

3.1 Introduction

The Intel XScale® core is an ARM® V5TE compliant microprocessor. It has been designed for high performance and low-power; leading the industry in mW/MIPs. The Intel XScale® core incorporates an extensive list of architecture features that allows it to achieve high performance. Many of the architectural features added to the Intel XScale® core help hide memory latency which often is a serious impediment to high performance processors.

This includes:

- the ability to continue instruction execution even while the data cache is retrieving data from external memory.
- a write buffer.
- write-back caching.
- various data cache allocation policies which can be configured different for each application.
- and cache locking.

All these features improve the efficiency of the memory bus external to the core.

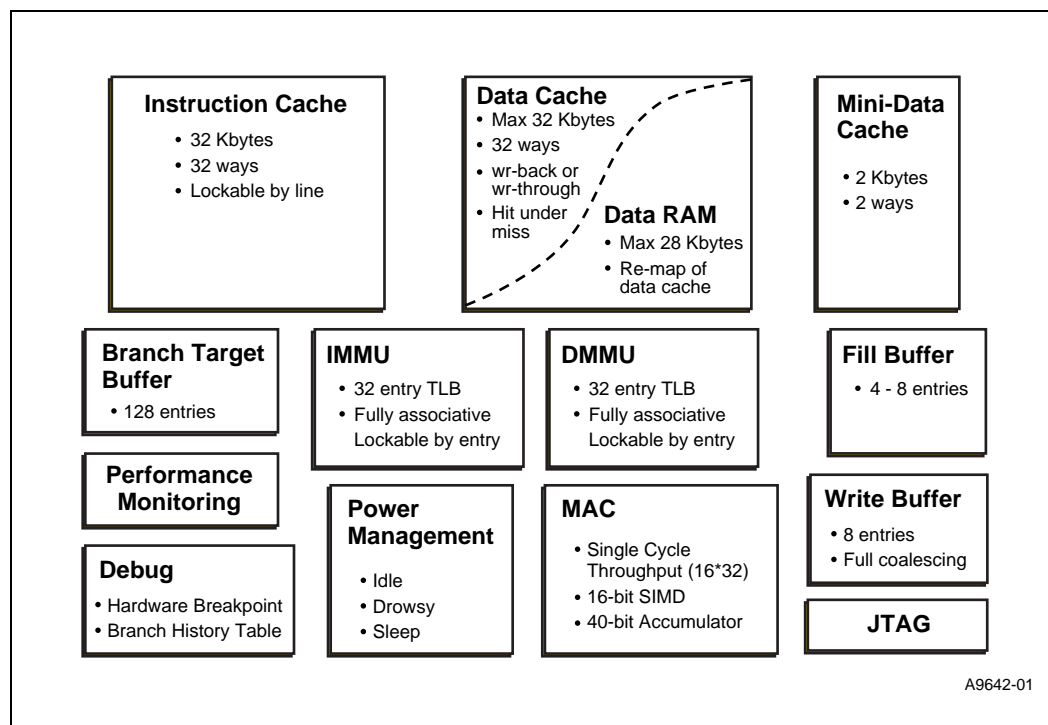
ARM® Version 5 (V5) Architecture added floating point instructions to ARM® Version 4. The Intel XScale® core implements the integer instruction set architecture of ARM® V5, but does not provide hardware support of the floating point instructions.

The Intel XScale® core provides the Thumb instruction set (ARM® V5T) and the ARM® V5E DSP extensions.

3.2 Features

Figure 16 shows the major functional blocks of the Intel XScale® core. The following sections give a brief, high-level overview of these blocks.

Figure 16. Intel XScale® Core Architecture Features



3.2.1 Multiply/ACcumulate (MAC)

The MAC unit supports early termination of multiplies/accumulates in two cycles and can sustain a throughput of a MAC operation every cycle. Several architectural enhancements were made to the MAC to support audio coding algorithms, which include a 40-bit accumulator and support for 16-bit packed data.

3.2.2 Memory Management

The Intel XScale® core implements the Memory Management Unit (MMU) Architecture specified in the *ARM® Architecture Reference Manual*. The MMU provides access protection and virtual to physical address translation.

The MMU Architecture also specifies the caching policies for the instruction cache and data memory. These policies are specified as page attributes and include:

- identifying code as cacheable or non-cacheable
- selecting between the mini-data cache or data cache
- write-back or write-through data caching

- enabling data write allocation policy
- and enabling the write buffer to coalesce stores to external memory

3.2.3 Instruction Cache

The Intel XScale® core implements a 32-Kbyte, 32-way set associative instruction cache with a line size of 32 bytes. All requests that “miss” the instruction cache generate a 32-byte read request to external memory. A mechanism to lock critical code within the cache is also provided.

3.2.4 Branch Target Buffer

The Intel XScale® core provides a Branch Target Buffer (BTB) to predict the outcome of branch type instructions. It provides storage for the target address of branch type instructions and predicts the next address to present to the instruction cache when the current instruction address is that of a branch.

The BTB holds 128 entries.

3.2.5 Data Cache

The Intel XScale® core implements a 32-Kbyte, a 32-way set associative data cache and a 2-Kbyte, 2-way set associative mini-data cache. Each cache has a line size of 32 bytes, and supports write-through or write-back caching.

The data/mini-data cache is controlled by page attributes defined in the MMU Architecture and by coprocessor 15.

The Intel XScale® core allows applications to re-configure a portion of the data cache as data RAM. Software may place special tables or frequently used variables in this RAM.

3.2.6 Performance Monitoring

Two performance monitoring counters have been added to the Intel XScale® core that can be configured to monitor various events. These events allow a software developer to measure cache efficiency, detect system bottlenecks, and reduce the overall latency of programs.

3.2.7 Power Management

The Intel XScale® core incorporates a power and clock management unit that can assist in controlling clocking and managing power.

3.2.8 Debug

The Intel XScale® core supports software debugging through two instruction address breakpoint registers, one data-address breakpoint register, one data-address/mask breakpoint register, and a trace buffer.



3.2.9 JTAG

Testability is supported on the Intel XScale® core through the Test Access Port (TAP) Controller implementation, which is based on IEEE 1149.1 (JTAG) Standard Test Access Port and Boundary-Scan Architecture. The purpose of the TAP controller is to support test logic internal and external to the Intel XScale® core such as built-in self-test, boundary-scan, and scan.

3.3 Memory Management

The Intel XScale® core implements the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Reference Manual*. To accelerate virtual to physical address translation, the Intel XScale® core uses both an instruction Translation Look-aside Buffer (TLB) and a data TLB to cache the latest translations. Each TLB holds 32 entries and is fully-associative. Not only do the TLBs contain the translated addresses, but also the access rights for memory references.

If an instruction or data TLB miss occurs, a hardware translation-table-walking mechanism is invoked to translate the virtual address to a physical address. Once translated, the physical address is placed in the TLB along with the access rights and attributes of the page or section. These translations can also be locked down in either TLB to guarantee the performance of critical routines.

The Intel XScale® core allows system software to associate various attributes with regions of memory:

- cacheable
- bufferable
- line allocate policy
- write policy
- I/O
- mini Data Cache
- Coalescing
- P bit

Note: The virtual address with which the TLBs are accessed may be remapped by the PID register.



3.3.1 Architecture Model

3.3.1.1 Version 4 vs. Version 5

ARM* MMU Version 5 Architecture introduces the support of tiny pages, which are 1 KByte in size. The reserved field in the first-level descriptor (encoding 0b11) is used as the fine page table base address.

3.3.1.2 Memory Attributes

The attributes associated with a particular region of memory are configured in the memory management page table and control the behavior of accesses to the instruction cache, data cache, mini-data cache and the write buffer. These attributes are ignored when the MMU is disabled.

To allow compatibility with older system software, the new Intel XScale® core attributes take advantage of encoding space in the descriptors that was formerly reserved.

3.3.1.2.1 Page (P) Attribute Bit

The P bit assigns a page attribute to a memory region. Refer to the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for details about the P bit.

3.3.1.2.2 Instruction Cache

When examining these bits in a descriptor, the Instruction Cache only utilizes the C bit. If the C bit is clear, the Instruction Cache considers a code fetch from that memory to be non-cacheable, and will not fill a cache entry. If the C bit is set, then fetches from the associated memory region will be cached.

3.3.1.2.3 Data Cache and Write Buffer

All of these descriptor bits affect the behavior of the Data Cache and the Write Buffer.

If the X bit for a descriptor is zero (see [Table 22](#)), the C and B bits operate as mandated by the ARM* architecture. If the X bit for a descriptor is one, the C and B bits' meaning is extended, as detailed in [Table 23](#).

Table 22. Data Cache and Buffer Behavior when X = 0

C B	Cacheable?	Bufferable?	Write Policy	Line Allocation Policy	Notes
0 0	N	N	-	-	Stall until complete ¹
0 1	N	Y	-	-	
1 0	Y	Y	Write Through	Read Allocate	
1 1	Y	Y	Write Back	Read Allocate	

1. Normally, the processor will continue executing after a data access if no dependency on that access is encountered. With this setting, the processor will stall execution until the data access completes. This guarantees to software that the data access has taken effect by the time execution of the data access instruction completes. External data aborts from such accesses will be imprecise.

Table 23. Data Cache and Buffer Behavior when X = 1

C B	Cacheable?	Bufferable?	Write Policy	Line Allocation Policy	Notes
0 0	-	-	-	-	Unpredictable -- do not use
0 1	N	Y	-	-	Writes will not coalesce into buffers ¹
1 0	(Mini Data Cache)	-	-	-	Cache policy is determined by MD field of Auxiliary Control register
1 1	Y	Y	Write Back	Read/Write Allocate	

1. Normally, bufferable writes can coalesce with previously buffered data in the same address range

3.3.1.2.4 Details on Data Cache and Write Buffer Behavior

If the MMU is disabled all data accesses will be non-cacheable and non-bufferable. This is the same behavior as when the MMU is enabled, and a data access uses a descriptor with X, C, and B all set to 0.

The X, C, and B bits determine when the processor should place new data into the Data Cache. The cache places data into the cache in lines (also called blocks). Thus, the basis for making a decision about placing new data into the cache is a called a “Line Allocation Policy.”

If the Line Allocation Policy is read-allocate, all load operations that miss the cache request a 32-byte cache line from external memory and allocate it into either the data cache or mini-data cache (this is assuming the cache is enabled). Store operations that miss the cache will not cause a line to be allocated.

If read/write-allocate is in effect, load or store operations that miss the cache will request a 32-byte cache line from external memory if the cache is enabled.

The other policy determined by the X, C, and B bits is the Write Policy. A write-through policy instructs the Data Cache to keep external memory coherent by performing stores to both external memory and the cache. A write-back policy only updates external memory when a line in the cache is cleaned or needs to be replaced with a new line. Generally, write-back provides higher performance because it generates less data traffic to external memory.

3.3.1.2.5 Memory Operation Ordering

A *fence* memory operation (memop) is one that guarantees all memops issued prior to the fence will execute before any memop issued after the fence. Thus software may issue a fence to impose a partial ordering on memory accesses.

Table 24 shows the circumstances in which memops act as fences.

Any swap (**SWP** or **SWPB**) to a page that would create a fence on a load or store is a fence.

Table 24. Memory Operations that Impose a Fence

operation	X	C	B
load	-	0	-
store	1	0	1
load or store	0	0	0

3.3.2 Exceptions

The MMU may generate prefetch aborts for instruction accesses and data aborts for data memory accesses.

Data address alignment checking is enabled by setting bit 1 of the Control Register (CP15, register 1). Alignment faults are still reported even if the MMU is disabled. All other MMU exceptions are disabled when the MMU is disabled.

3.3.3 Interaction of the MMU, Instruction Cache, and Data Cache

The MMU, instruction cache, and data/mini-data cache may be enabled/disabled independently. The instruction cache can be enabled with the MMU enabled or disabled. However, the data cache can only be enabled when the MMU is enabled. Therefore only three of the four combinations of the MMU and data/mini-data cache enables are valid (see [Table 25](#)). The invalid combination will cause undefined results.

Table 25. Valid MMU & Data/mini-data Cache Combinations

MMU	Data/mini-data Cache
Off	Off
On	Off
On	On

3.3.4 Control

3.3.4.1 Invalidate (Flush) Operation

The entire instruction and data TLB can be invalidated at the same time with one command or they can be invalidated separately. An individual entry in the data or instruction TLB can also be invalidated.

Globally invalidating a TLB will not affect locked TLB entries. However, the invalidate-entry operations can invalidate individual locked entries. In this case, the locked remains in the TLB, but will never “hit” on an address translation. Effectively, a hole is in the TLB. This situation may be rectified by unlocking the TLB.



3.3.4.2 Enabling/Disabling

The MMU is enabled by setting bit 0 in coprocessor 15, register 1 (Control Register).

When the MMU is disabled, accesses to the instruction cache default to cacheable and all accesses to data memory are made non-cacheable.

A recommended code sequence for enabling the MMU is shown in [Example 13](#).

Example 13. Enabling the MMU

```
; This routine provides software with a predictable way of enabling the MMU.
; After the CPWAIT, the MMU is guaranteed to be enabled. Be aware
; that the MMU will be enabled sometime after MCR and before the instruction
; that executes after the CPWAIT.
; Programming Note: This code sequence requires a one-to-one virtual to
; physical address mapping on this code since
; the MMU may be enabled part way through. This would allow the instructions
; after MCR to execute properly regardless the state of the MMU.

MRC P15,0,R0,C1,C0,0; Read CP15, register 1
ORR R0, R0, #0x1; Turn on the MMU
MCR P15,0,R0,C1,C0,0; Write to CP15, register 1

; The MMU is guaranteed to be enabled at this point; the next instruction or
; data address will be translated.
```

3.3.4.3 Locking Entries

Individual entries can be locked into the instruction and data TLBs. If a lock operation finds the virtual address translation already resident in the TLB, the results are unpredictable. An invalidate by entry command before the lock command will ensure proper operation. Software can also accomplish this by invalidating all entries, as shown in [Example 14](#).

Locking entries into either the instruction TLB or data TLB reduces the available number of entries (by the number that was locked down) for hardware to cache other virtual to physical address translations.

A procedure for locking entries into the instruction TLB is shown in [Example 14](#).

If a MMU abort is generated during an instruction or data TLB lock operation, the Fault Status Register is updated to indicate a Lock Abort, and the exception is reported as a data abort.

Example 14. Locking Entries into the Instruction TLB

```
; R1, R2 and R3 contain the virtual addresses to translate and lock into
; the instruction TLB.

; The value in R0 is ignored in the following instruction.
; Hardware guarantees that accesses to CP15 occur in program order

MCR P15,0,R0,C8,C5,0 ; Invalidate the entire instruction TLB

MCR P15,0,R1,C10,C4,0 ; Translate virtual address (R1) and lock into
; instruction TLB
MCR P15,0,R2,C10,C4,0 ; Translate
; virtual address (R2) and lock into instruction TLB
MCR P15,0,R3,C10,C4,0 ; Translate virtual address (R3) and lock into
; instruction TLB

CPWAIT

; The MMU is guaranteed to be updated at this point; the next instruction will
; see the locked instruction TLB entries.
```

Note: If exceptions are allowed to occur in the middle of this routine, the TLB may end up caching a translation that is about to be locked. For example, if R1 is the virtual address of an interrupt service routine and that interrupt occurs immediately after the TLB has been invalidated, the lock operation will be ignored when the interrupt service routine returns back to this code sequence. Software should disable interrupts (FIQ or IRQ) in this case.

As a general rule, software should avoid locking in all other exception types.

The proper procedure for locking entries into the data TLB is shown in [Example 15](#).

Example 15. Locking Entries into the Data TLB

```
; R1, and R2 contain the virtual addresses to translate and lock into the data TLB

MCR P15,0,R1,C8,C6,1 ; Invalidate the data TLB entry specified by the
; virtual address in R1
MCR P15,0,R1,C10,C8,0 ; Translate virtual address (R1) and lock into
; data TLB

; Repeat sequence for virtual address in R2
MCR P15,0,R2,C8,C6,1 ; Invalidate the data TLB entry specified by the
; virtual address in R2
MCR P15,0,R2,C10,C8,0 ; Translate virtual address (R2) and lock into
; data TLB

CPWAIT ; wait for locks to complete

; The MMU is guaranteed to be updated at this point; the next instruction will
; see the locked data TLB entries.
```

Note: Care must be exercised here when allowing exceptions to occur during this routine whose handlers may have data that lies in a page that is trying to be locked into the TLB.

3.3.4.4 Round-Robin Replacement Algorithm

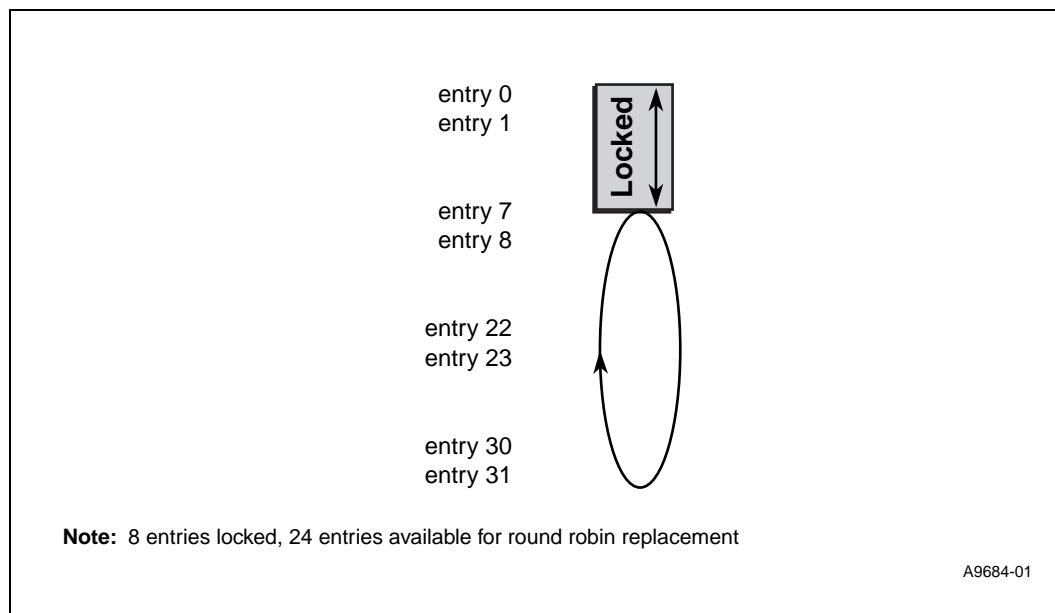
The line replacement algorithm for the TLBs is round-robin; there is a round-robin pointer that keeps track of the next entry to replace. The next entry to replace is the one sequentially after the last entry that was written. For example, if the last virtual to physical address translation was written into entry 5, the next entry to replace is entry 6.

At reset, the round-robin pointer is set to entry 31. Once a translation is written into entry 31, the round-robin pointer gets set to the next available entry, beginning with entry 0 if no entries have been locked down. Subsequent translations move the round-robin pointer to the next sequential entry until entry 31 is reached, where it will wrap back to entry 0 upon the next translation.

A lock pointer is used for locking entries into the TLB and is set to entry 0 at reset. A TLB lock operation places the specified translation at the entry designated by the lock pointer, moves the lock pointer to the next sequential entry, and resets the round-robin pointer to entry 31. Locking entries into either TLB effectively reduces the available entries for updating. For example, if the first three entries were locked down, the round-robin pointer would be entry 3 after it rolled over from entry 31.

Only entries 0 through 30 can be locked in either TLB; entry 31 can never be locked. If the lock pointer is at entry 31, a lock operation will update the TLB entry with the translation and ignore the lock. In this case, the round-robin pointer will stay at entry 31.

Figure 17. Example of Locked Entries in TLB





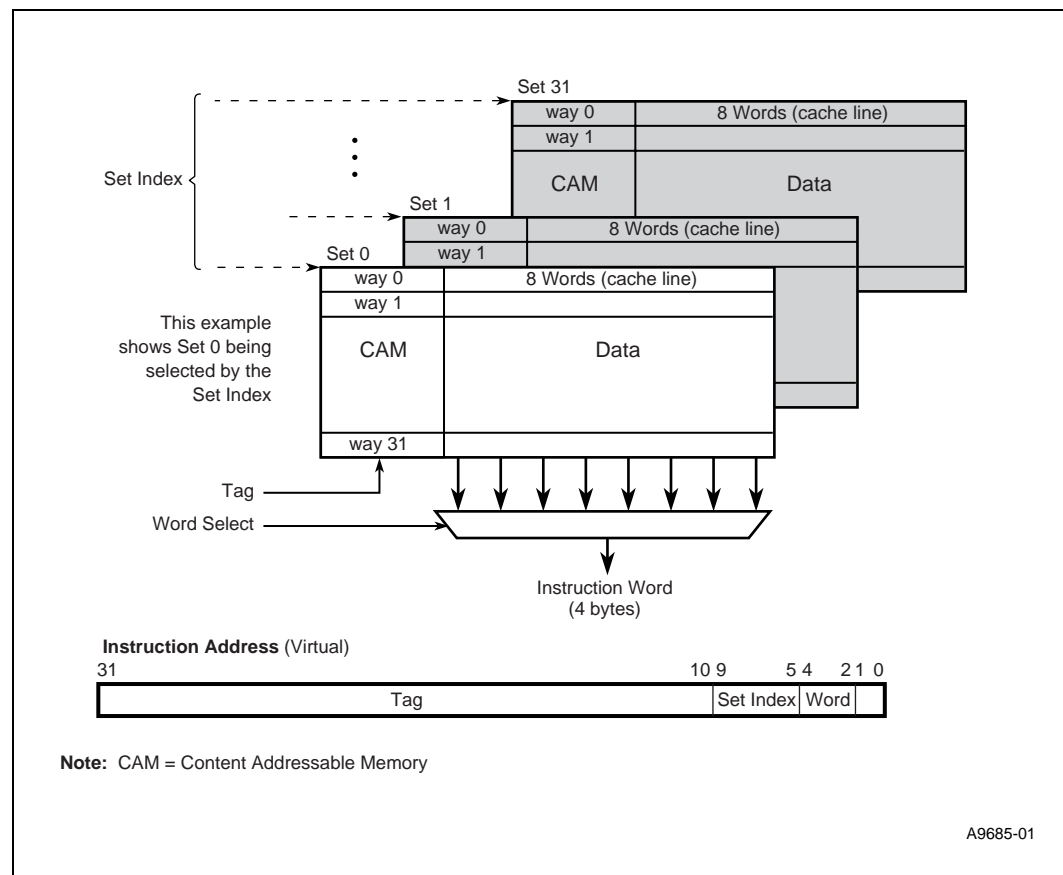
3.4 Instruction Cache

The Intel XScale® core instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code. Code can also be locked down when guaranteed or fast access time is required.

Figure 18 shows the cache organization and how the instruction address is used to access the cache.

The instruction cache is a 32-Kbyte, 32-way set associative cache; this means there are 32 sets with each set containing 32 ways. Each way of a set contains eight 32-bit words and one valid bit, which is referred to as a line. The replacement policy is a round-robin algorithm and the cache also supports the ability to lock code in at a line granularity.

Figure 18. Instruction Cache Organization



The instruction cache is virtually addressed and virtually tagged.

Note: The virtual address presented to the instruction cache may be remapped by the PID register.



3.4.1 Instruction Cache Operation

3.4.1.1 Operation When Instruction Cache is Enabled

When the cache is enabled, it compares every instruction request address against the addresses of instructions that it is currently holding. If the cache contains the requested instruction, the access “hits” the cache, and the cache returns the requested instruction. If the cache does not contain the requested instruction, the access “misses” the cache, and the cache requests a fetch from external memory of the 8-word line (32 bytes) that contains the requested instruction using the fetch policy. As the fetch returns instructions to the cache, they are placed in one of two fetch buffers and the requested instruction is delivered to the instruction decoder.

A fetched line will be written into the cache if it is cacheable. Code is designated as cacheable when the Memory Management Unit (MMU) is disabled or when the MMU is enable and the cacheable (C) bit is set to 1 in its corresponding page.

Note that an instruction fetch may “miss” the cache but “hit” one of the fetch buffers. When this happens, the requested instruction will be delivered to the instruction decoder in the same manner as a cache “hit.”

3.4.1.2 Operation When The Instruction Cache Is Disabled

Disabling the cache prevents any lines from being written into the instruction cache. Although the cache is disabled, it is still accessed and may generate a “hit” if the data is already in the cache.

Disabling the instruction cache *does not* disable instruction buffering that may occur within the instruction fetch buffers. Two 8-word instruction fetch buffers will always be enabled in the cache disabled mode. So long as instruction fetches continue to “hit” within either buffer (even in the presence of forward and backward branches), no external fetches for instructions are generated. A miss causes one or the other buffer to be filled from external memory using the fill policy.

3.4.1.3 Fetch Policy

An instruction-cache “miss” occurs when the requested instruction is not found in the instruction fetch buffers or instruction cache; a fetch request is then made to external memory. The instruction cache can handle up to two “misses.” Each external fetch request uses a fetch buffer that holds 32-bytes and eight valid bits, one for each word. A miss causes the following:

1. A fetch buffer is allocated.
2. The instruction cache sends a fetch request to the external bus. This request is for a 32-byte line.
3. Instructions words are returned back from the external bus, at a maximum rate of 1 word per core cycle. As each word returns, the corresponding valid bit is set for the word in the fetch buffer.
4. As soon as the fetch buffer receives the requested instruction, it forwards the instruction to the instruction decoder for execution.
5. When all words have returned, the fetched line will be written into the instruction cache if it's cacheable and if the instruction cache is enabled. The line chosen for update in the cache is controlled by the round-robin replacement algorithm. This update may evict a valid line at that location.
6. Once the cache is updated, the eight valid bits of the fetch buffer are invalidated.



3.4.1.4 Round-Robin Replacement Algorithm

The line replacement algorithm for the instruction cache is round-robin. Each set in the instruction cache has a round-robin pointer that keeps track of the next line (in that set) to replace. The next line to replace in a set is the one after the last line that was written. For example, if the line for the last external instruction fetch was written into way 5-set 2, the next line to replace for that set would be way 6. None of the other round-robin pointers for the other sets are affected in this case.

After reset, way 31 is pointed to by the round-robin pointer for all the sets. Once a line is written into way 31, the round-robin pointer points to the first available way of a set, beginning with way 0 if no lines have been locked into that particular set. Locking lines into the instruction cache effectively reduces the available lines for cache updating. For example, if the first three lines of a set were locked down, the round-robin pointer would point to the line at way 3 after it rolled over from way 31.

3.4.1.5 Parity Protection

The instruction cache is protected by parity to ensure data integrity. Each instruction cache word has 1 parity bit. (The instruction cache tag is NOT parity protected.) When a parity error is detected on an instruction cache access, a prefetch abort exception occurs if the Intel XScale® core attempts to execute the instruction. Before servicing the exception, hardware places a notification of the error in the Fault Status Register (Coprocessor 15, register 5).

A software exception handler can recover from an instruction cache parity error. This can be accomplished by invalidating the instruction cache and the branch target buffer and then returning to the instruction that caused the prefetch abort exception. A simplified code example is shown in [Example 16](#). A more complex handler might choose to invalidate the specific line that caused the exception and then invalidate the BTB.

Example 16. Recovering from an Instruction Cache Parity Error

```
; Prefetch abort handler
MCR P15,0,R0,C7,C5,0 ; Invalidate the instruction cache and branch target
                        ; buffer

CPWAIT                ; wait for effect
                        ;

SUBS PC,R14,#4         ; Returns to the instruction that generated the
                        ; parity error

; The Instruction Cache is guaranteed to be invalidated at this point
```

If a parity error occurs on an instruction that is locked in the cache, the software exception handler needs to unlock the instruction cache, invalidate the cache and then re-lock the code in before it returns to the faulting instruction.



3.4.1.6 Instruction Cache Coherency

The instruction cache does not detect modification to program memory by loads, stores or actions of other bus masters. Several situations may require program memory modification, such as uploading code from disk.

The application program is responsible for synchronizing code modification and invalidating the cache. In general, software must ensure that modified code space is not accessed until modification and invalidating are completed.

To achieve cache coherence, instruction cache contents can be invalidated after code modification in external memory is complete.

If the instruction cache is not enabled, or code is being written to a non-cacheable region, software must still invalidate the instruction cache before using the newly-written code. This precaution ensures that state associated with the new code is not buffered elsewhere in the processor, such as the fetch buffers or the BTB.

Naturally, when writing code as data, care must be taken to force it completely out of the processor into external memory before attempting to execute it. If writing into a non-cacheable region, flushing the write buffers is sufficient precaution. If writing to a cacheable region, then the data cache should be submitted to a Clean/Invalidate operation to ensure coherency.

3.4.2 Instruction Cache Control

3.4.2.1 Instruction Cache State at Reset

After reset, the instruction cache is always disabled, unlocked, and invalidated (flushed).

3.4.2.2 Enabling/Disabling

The instruction cache is enabled by setting bit 12 in coprocessor 15, register 1 (Control Register). This process is illustrated in [Example 17](#).

Example 17. Enabling the Instruction Cache

```
; Enable the ICache
MRC P15, 0, R0, C1, C0, 0      ; Get the control register
ORR R0, R0, #0x1000            ; set bit 12 -- the I bit
MCR P15, 0, R0, C1, C0, 0      ; Set the control register

CPWAIT
```



3.4.2.3 Invalidating the Instruction Cache

The entire instruction cache along with the fetch buffers are invalidated by writing to coprocessor 15, register 7. This command does not unlock any lines that were locked in the instruction cache nor does it invalidate those locked lines. To invalidate the entire cache including locked lines, the unlock instruction cache command needs to be executed before the invalidate command.

There is an inherent delay from the execution of the instruction cache invalidate command to where the next instruction will see the result of the invalidate. The routine in [Example 18](#) can be used to guarantee proper synchronization.

Example 18. Invalidating the Instruction Cache

```
MCR P15,0,R1,C7,C5,0 ; Invalidate the instruction cache and branch
                        ; target buffer

CPWAIT

; The instruction cache is guaranteed to be invalidated at this point; the next
; instruction sees the result of the invalidate command.
```

The Intel XScale® core also supports invalidating an individual line from the instruction cache.

3.4.2.4 Locking Instructions in the Instruction Cache

Software has the ability to lock performance critical routines into the instruction cache. Up to 28 lines in each set can be locked; hardware will ignore the lock command if software is trying to lock all the lines in a particular set (i.e., ways 28-31 can never be locked). When this happens, the line will still be allocated into the cache but the lock will be ignored. The round-robin pointer will stay at way 31 for that set.

Lines can be locked into the instruction cache by initiating a write to coprocessor 15. Register *Rd* contains the virtual address of the line to be locked into the cache.

There are several requirements for locking down code:

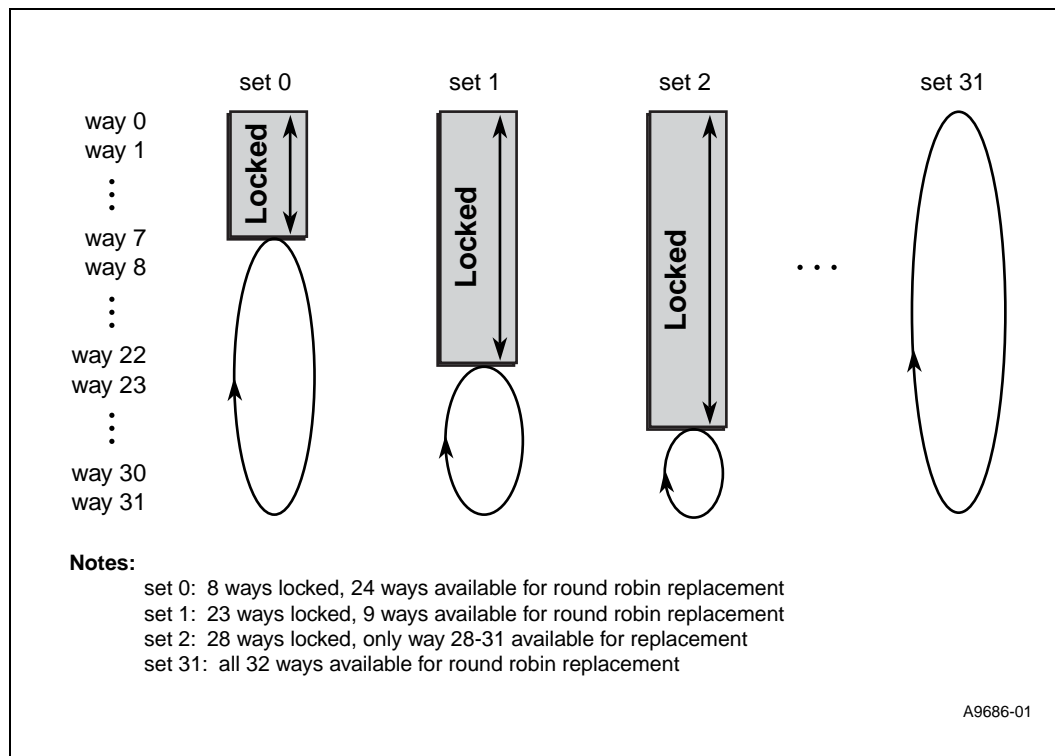
1. the routine used to lock lines down in the cache must be placed in non-cacheable memory, which means the MMU is enabled. As a corollary: no fetches of cacheable code should occur while locking instructions into the cache.
2. the code being locked into the cache must be cacheable
3. the instruction cache must be enabled and invalidated prior to locking down lines

Failure to follow these requirements will produce unpredictable results when accessing the instruction cache.

System programmers should ensure that the code to lock instructions into the cache does not reside closer than 128 bytes to a non-cacheable/cacheable page boundary. If the processor fetches ahead into a cacheable page, then the first requirement noted above could be violated.

Lines are locked into a set starting at way 0 and may progress up to way 27; which set a line gets locked into depends on the set index of the virtual address. [Figure 19](#) is an example of where lines of code may be locked into the cache along with how the round-robin pointer is affected.

Figure 19. Locked Line Effect on Round Robin Replacement



Software can lock down several different routines located at different memory locations. This may cause some sets to have more locked lines than others as shown in Figure 19.

Example 19 shows how a routine, called “lockMe” in this example, might be locked into the instruction cache. Note that it is possible to receive an exception while locking code.

Example 19. Locking Code into the Cache

```
lockMe:                                ; This is the code that will be locked into the cache
    mov r0, #5
    add r5, r1, r2
    . . .
lockMeEnd:
    . . .

codeLock:                              ; here is the code to lock the "lockMe" routine
    ldr r0, =(lockMe AND NOT 31); r0 gets a pointer to the first line we
    should lock
    ldr r1, =(lockMeEnd AND NOT 31); r1 contains a pointer to the last line we
    should lock

lockLoop:
    mcr p15, 0, r0, c9, c1, 0; lock next line of code into ICache
    cmp r0, r1                ; are we done yet?
    add r0, r0, #32           ; advance pointer to next line
    bne lockLoop              ; if not done, do the next line
```




3.4.2.5 Unlocking Instructions in the Instruction Cache

The Intel XScale® core provides a global unlock command for the instruction cache. Writing to coprocessor 15, register 9 unlocks all the locked lines in the instruction cache and leaves them valid. These lines then become available for the round-robin replacement algorithm.

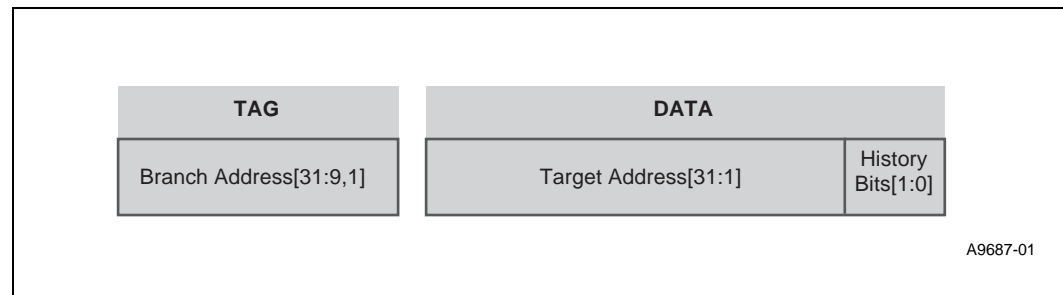
3.5 Branch Target Buffer

The Intel XScale® core uses dynamic branch prediction to reduce the penalties associated with changing the flow of program execution. The Intel XScale® core features a branch target buffer that provides the instruction cache with the target address of branch type instructions. The branch target buffer is implemented as a 128-entry, direct mapped cache.

3.5.1 Branch Target Buffer (BTB) Operation

The BTB stores the history of branches that have executed along with their targets. [Figure 20](#) shows an entry in the BTB, where the tag is the instruction address of a previously executed branch and the data contains the target address of the previously executed branch along with two bits of history information.

Figure 20. BTB Entry



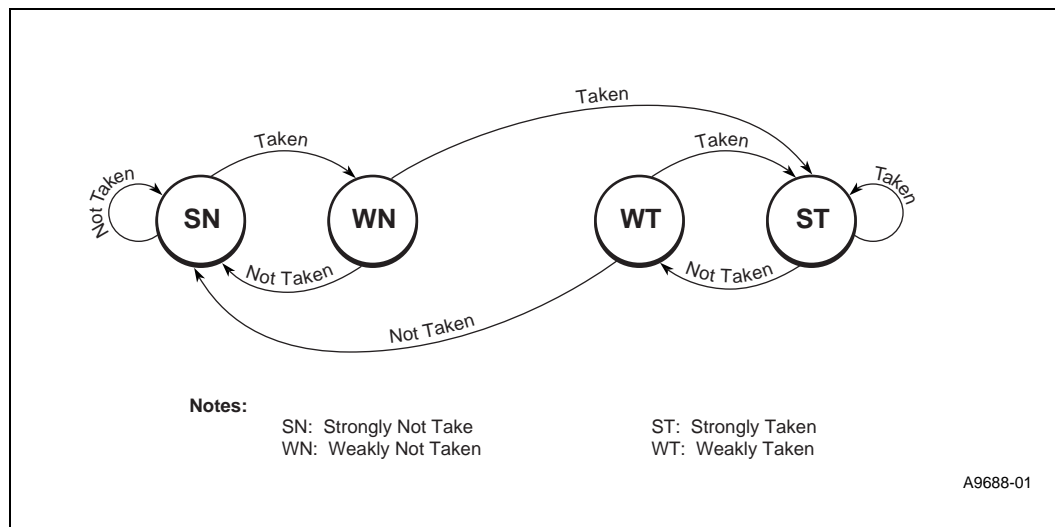
The BTB takes the current instruction address and checks to see if this address is a branch that was previously seen. It uses bits [8:2] of the current address to read out the tag and then compares this tag to bits [31:9,1] of the current instruction address. If the current instruction address matches the tag in the cache and the history bits indicate that this branch is usually taken in the past, the BTB uses the data (target address) as the next instruction address to send to the instruction cache.

Bit[1] of the instruction address is included in the tag comparison in order to support Thumb execution. This organization means that two consecutive Thumb branch (B) instructions, with instruction address bits[8:2] the same, will contend for the same BTB entry. Thumb also requires 31 bits for the branch target address. In ARM* mode, bit[1] is zero.

The history bits represent four possible prediction states for a branch entry in the BTB. [Figure 21](#) shows these states along with the possible transitions. The initial state for branches stored in the BTB is Weakly-Taken (WT). Every time a branch that exists in the BTB is executed, the history bits are updated to reflect the latest outcome of the branch, either taken or not-taken.

The BTB does not have to be managed explicitly by software; it is disabled by default after reset and is invalidated when the instruction cache is invalidated.

Figure 21. Branch History



3.5.1.1 Reset

After Processor Reset, the BTB is disabled and all entries are invalidated.

3.5.2 Update Policy

A new entry is stored into the BTB when the following conditions are met:

- the branch instruction has executed
- the branch was taken
- the branch is not currently in the BTB

The entry is then marked valid and the history bits are set to WT. If another valid branch exists at the same entry in the BTB, it will be evicted by the new branch.

Once a branch is stored in the BTB, the history bits are updated upon every execution of the branch as shown in Figure 21.

3.5.3 BTB Control

3.5.3.1 Disabling/Enabling

The BTB is always disabled with Reset. Software can enable the BTB through a bit in a coprocessor register.

Before enabling or disabling the BTB, software must invalidate it (described in the following section). This action will ensure correct operation in case stale data is in the BTB. Software should not place any branch instruction between the code that invalidates the BTB and the code that enables/disables it.



3.5.3.2 Invalidation

There are four ways the contents of the BTB can be invalidated.

1. Reset.
2. Software can directly invalidate the BTB via a CP15, register 7 function.
3. The BTB is invalidated when the Process ID Register is written.
4. The BTB is invalidated when the instruction cache is invalidated via CP15, register 7 functions.

3.6 Data Cache

The Intel XScale® core data cache enhances performance by reducing the number of data accesses to and from external memory. There are two data cache structures in the Intel XScale® core, a 32 Kbyte data cache and a 2 Kbyte mini-data cache. An eight entry write buffer and a four entry fill buffer are also implemented to decouple the Intel XScale® core instruction execution from external memory accesses, which increases overall system performance.

3.6.1 Overviews

3.6.1.1 Data Cache Overview

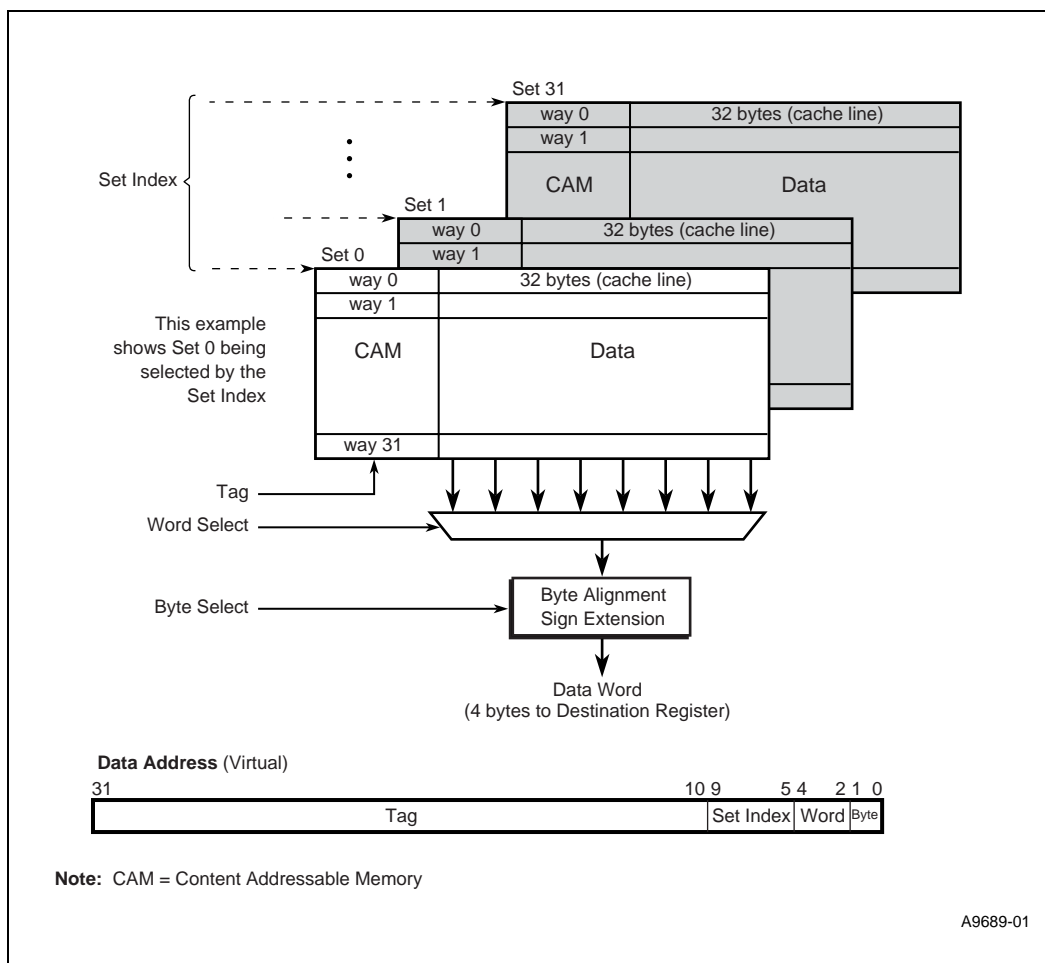
The data cache is a 32-Kbyte, 32-way set associative cache; this means there are 32 sets with each set containing 32 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There also exist two dirty bits for every line, one for the lower 16 bytes and the other one for the upper 16 bytes. When a store hits the cache the dirty bit associated with it is set. The replacement policy is a round-robin algorithm and the cache also supports the ability to reconfigure each line as data RAM.

Figure 22 shows the cache organization and how the data address is used to access the cache.

Cache policies may be adjusted for particular regions of memory by altering page attribute bits in the MMU descriptor that controls that memory.

The data cache is virtually addressed and virtually tagged. It supports write-back and write-through caching policies. The data cache always allocates a line in the cache when a cacheable read miss occurs and will allocate a line into the cache on a cacheable write miss when write allocate is specified by its page attribute. Page attribute bits determine whether a line gets allocated into the data cache or mini-data cache.

Figure 22. Data Cache Organization



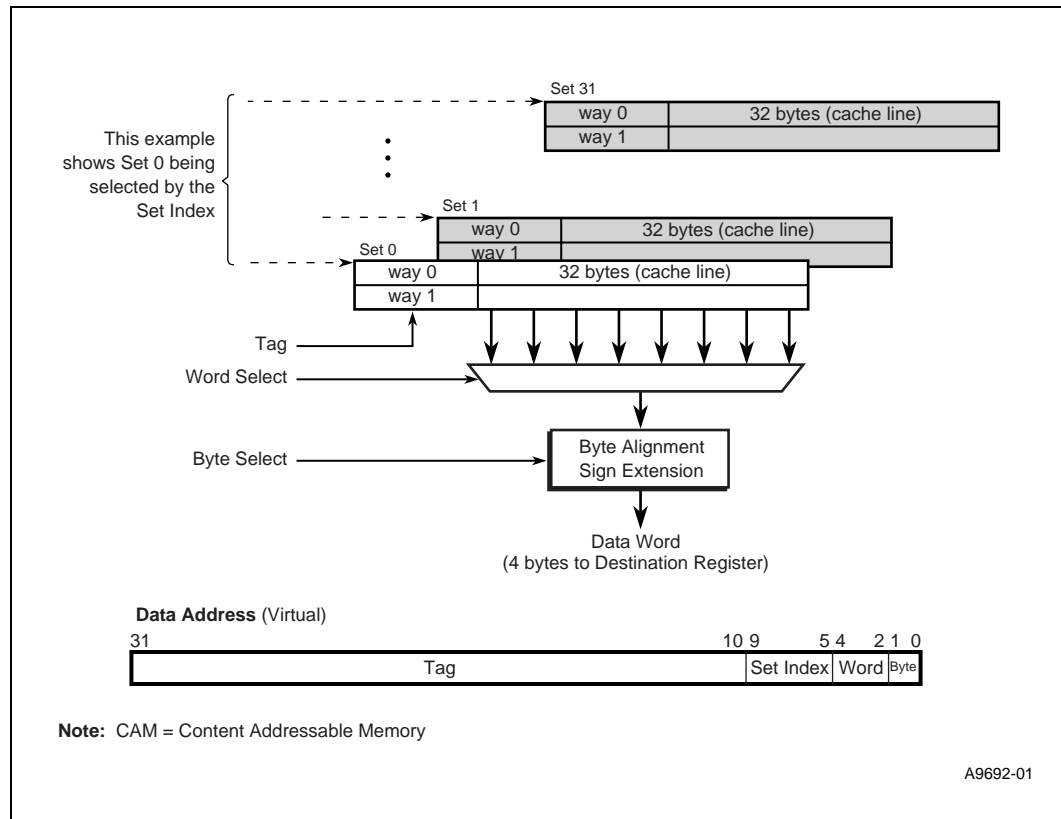
3.6.1.2 Mini-Data Cache Overview

The mini-data cache is a 2-Kbyte, 2-way set associative cache; this means there are 32 sets with each set containing 2 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There also exist 2 dirty bits for every line, one for the lower 16 bytes and the other one for the upper 16 bytes. When a store hits the cache the dirty bit associated with it is set. The replacement policy is a round-robin algorithm.

Figure 23 shows the cache organization and how the data address is used to access the cache.

The mini-data cache is virtually addressed and virtually tagged and supports the same caching policies as the data cache. However, lines can't be locked into the mini-data cache.

Figure 23. Mini-Data Cache Organization



3.6.1.3 Write Buffer and Fill Buffer Overview

The Intel XScale® core employs an eight entry write buffer, each entry containing 16 bytes. Stores to external memory are first placed in the write buffer and subsequently taken out when the bus is available.

The write buffer supports the coalescing of multiple store requests to external memory. An incoming store may coalesce with any of the eight entries.

The fill buffer holds the external memory request information for a data cache or mini-data cache fill or non-cacheable read request. Up to four 32-byte read request operations can be outstanding in the fill buffer before the Intel XScale® core needs to stall.

The fill buffer has been augmented with a four entry pend buffer that captures data memory requests to outstanding fill operations. Each entry in the pend buffer contains enough data storage to hold one 32-bit word, specifically for store operations. Cacheable load or store operations that hit an entry in the fill buffer get placed in the pend buffer and are completed when the associated fill completes. Any entry in the pend buffer can be pended against any of the entries in the fill buffer; multiple entries in the pend buffer can be pended against a single entry in the fill buffer.

Pended operations complete in program order.



3.6.2 Data Cache and Mini-Data Cache Operation

The following discussions refer to the data cache and mini-data cache as one cache (data/mini-data) since their behavior is the same when accessed.

3.6.2.1 Operation When Caching is Enabled

When the data/mini-data cache is enabled for an access, the data/mini-data cache compares the address of the request against the addresses of data that it is currently holding. If the line containing the address of the request is resident in the cache, the access “hits” the cache. For a load operation the cache returns the requested data to the destination register and for a store operation the data is stored into the cache. The data associated with the store may also be written to external memory if write-through caching is specified for that area of memory. If the cache does not contain the requested data, the access “misses” the cache, and the sequence of events that follows depends on the configuration of the cache, the configuration of the MMU and the page attributes.

3.6.2.2 Operation When Data Caching is Disabled

The data/mini-data cache is still accessed even though it is disabled. If a load hits the cache it will return the requested data to the destination register. If a store hits the cache, the data is written into the cache. Any access that misses the cache will not allocate a line in the cache when it’s disabled, even if the MMU is enabled and the memory region’s cacheability attribute is set.

3.6.2.3 Cache Policies

3.6.2.3.1 Cacheability

Data at a specified address is cacheable given the following:

- the MMU is enabled
- the cacheable attribute is set in the descriptor for the accessed address
- and the data/mini-data cache is enabled

3.6.2.3.2 Read Miss Policy

The following sequence of events occurs when a cacheable load operation misses the cache:

1. The fill buffer is checked to see if an outstanding fill request already exists for that line.
If so, the current request is placed in the pending buffer and waits until the previously requested fill completes, after which it accesses the cache again, to obtain the request data and returns it to the destination register.
If there is no outstanding fill request for that line, the current load request is placed in the fill buffer and a 32-byte external memory read request is made. If the pending buffer or fill buffer is full, the Intel XScale® core will stall until an entry is available.
2. A line is allocated in the cache to receive the 32-bytes of fill data. The line selected is determined by the round-robin pointer (see [Section 3.6.2.4](#)). The line chosen may contain a valid line previously allocated in the cache. In this case both dirty bits are examined and if set, the four words associated with a dirty bit that’s asserted will be written back to external memory as a four word burst operation.

3. When the data requested by the load is returned from external memory, it is immediately sent to the destination register specified by the load. A system that returns the requested data back first, with respect to the other bytes of the line, will obtain the best performance.
4. As data returns from external memory it is written into the cache in the previously allocated line.

A load operation that misses the cache and is NOT cacheable makes a request from external memory for the exact data size of the original load request. For example, **LDRH** requests exactly two bytes from external memory, **LDR** requests 4 bytes from external memory, etc. This request is placed in the fill buffer until, the data is returned from external memory, which is then forwarded back to the destination register(s).

3.6.2.3.3 Write Miss Policy

A write operation that misses the cache will request a 32-byte cache line from external memory if the access is cacheable and write allocation is specified in the page. In this case the following sequence of events occur:

1. The fill buffer is checked to see if an outstanding fill request already exists for that line.
If so, the current request is placed in the pending buffer and waits until the previously requested fill completes, after which it writes its data into the recently allocated cache line.
If there is no outstanding fill request for that line, the current store request is placed in the fill buffer and a 32-byte external memory read request is made. If the pending buffer or fill buffer is full, the Intel XScale® core will stall until an entry is available.
2. The 32-bytes of data can be returned back to the Intel XScale® core in any word order, i.e., the eight words in the line can be returned in any order. Note that it does not matter, for performance reasons, which order the data is returned to the Intel XScale® core since the store operation has to wait until the entire line is written into the cache before it can complete.
3. When the entire 32-byte line has returned from external memory, a line is allocated in the cache, selected by the round-robin pointer (see [Section 3.6.2.4](#)). The line to be written into the cache may replace a valid line previously allocated in the cache. In this case both dirty bits are examined and if any are set, the four words associated with a dirty bit that's asserted will be written back to external memory as a 4 word burst operation. This write operation will be placed in the write buffer.
4. The line is written into the cache along with the data associated with the store operation.

If the above condition for requesting a 32-byte cache line is not met, a write miss will cause a write request to external memory for the exact data size specified by the store operation, assuming the write request doesn't coalesce with another write operation in the write buffer.

3.6.2.3.4 Write-Back Versus Write-Through

The Intel XScale® core supports write-back caching or write-through caching, controlled through the MMU page attributes. When write-through caching is specified, all store operations are written to external memory even if the access hits the cache. This feature keeps the external memory coherent with the cache, i.e., no dirty bits are set for this region of memory in the data/mini-data cache. This however does not guarantee that the data/mini-data cache is coherent with external memory, which is dependent on the system level configuration, specifically if the external memory is shared by another master.

When write-back caching is specified, a store operation that hits the cache will not generate a write to external memory, thus reducing external memory traffic.



3.6.2.4 Round-Robin Replacement Algorithm

The line replacement algorithm for the data cache is round-robin. Each set in the data cache has a round-robin pointer that keeps track of the next line (in that set) to replace. The next line to replace in a set is the next sequential line after the last one that was just filled. For example, if the line for the last fill was written into way 5-set 2, the next line to replace for that set would be way 6. None of the other round-robin pointers for the other sets are affected in this case.

After reset, way 31 is pointed to by the round-robin pointer for all the sets. Once a line is written into way 31, the round-robin pointer points to the first available way of a set, beginning with way 0 if no lines have been re-configured as data RAM in that particular set. Re-configuring lines as data RAM effectively reduces the available lines for cache updating. For example, if the first three lines of a set were re-configured, the round-robin pointer would point to the line at way 3 after it rolled over from way 31. Refer to [Section 3.6.4](#) for more details on data RAM.

The mini-data cache follows the same round-robin replacement algorithm as the data cache except that there are only two lines the round-robin pointer can point to such that the round-robin pointer always points to the least recently filled line. A least recently used replacement algorithm is not supported because the purpose of the mini-data cache is to cache data that exhibits low temporal locality, i.e., data that is placed into the mini-data cache is typically modified once and then written back out to external memory.

3.6.2.5 Parity Protection

The data cache and mini-data cache are protected by parity to ensure data integrity; there is one parity bit per byte of data. (The tags are NOT parity protected.) When a parity error is detected on a data/mini-data cache access, a data abort exception occurs. Before servicing the exception, hardware will set bit 10 of the Fault Status Register register.

A data/mini-data cache parity error is an imprecise data abort, meaning R14_ABORT (+8) may not point to the instruction that caused the parity error. If the parity error occurred during a load, the targeted register may be updated with incorrect data.

A data abort due to a data/mini-data cache parity error may not be recoverable if the data address that caused the abort occurred on a line in the cache that has a write-back caching policy. Prior updates to this line may be lost; in this case the software exception handler should perform a “clean and clear” operation on the data cache, ignoring subsequent parity errors, and restart the offending process. This operation is shown in [Section 3.6.3.3.1](#).

3.6.2.6 Atomic Accesses

The **SWP** and **SWPB** instructions generate an atomic load and store operation allowing a memory semaphore to be loaded and altered without interruption. These accesses may hit or miss the data/mini-data cache depending on configuration of the cache, configuration of the MMU, and the page attributes. Refer to [Section 3.11.4](#) for more information.

3.6.3 Data Cache and Mini-Data Cache Control

3.6.3.1 Data Memory State After Reset

After processor reset, both the data cache and mini-data cache are disabled, all valid bits are set to zero (invalid), and the round-robin bit points to way 31. Any lines in the data cache that were configured as data RAM before reset are changed back to cacheable lines after reset, i.e., there are 32 KBytes of data cache and zero bytes of data RAM.

3.6.3.2 Enabling/Disabling

The data cache and mini-data cache are enabled by setting bit 2 in coprocessor 15, register 1 (Control Register).

[Example 20](#) shows code that enables the data and mini-data caches. Note that the MMU must be enabled to use the data cache.

Example 20. Enabling the Data Cache

```
enableDCache:

    MCR p15, 0, r0, c7, c10, 4; Drain pending data operations...
    ;
    MRC p15, 0, r0, c1, c0, 0; Get current control register
    ORR r0, r0, #4          ; Enable DCache by setting 'C' (bit 2)
    MCR p15, 0, r0, c1, c0, 0; And update the Control register
```

3.6.3.3 Invalidate and Clean Operations

Individual entries can be invalidated and cleaned in the data cache and mini-data cache via coprocessor 15, register 7. Note that a line locked into the data cache remains locked even after it has been subjected to an invalidate-entry operation. This will leave an unusable line in the cache until a global unlock has occurred. For this reason, do not use these commands on locked lines.

This same register also provides the command to invalidate the entire data cache and mini-data cache. These global invalidate commands have no effect on lines locked in the data cache. Locked lines must be unlocked before they can be invalidated. This is accomplished by the Unlock Data Cache command.

3.6.3.3.1 Global Clean and Invalidate Operation

A simple software routine is used to globally clean the data cache. It takes advantage of the line-allocate data cache operation, which allocates a line into the data cache. This allocation evicts any cache dirty data back to external memory. [Example 21](#) shows how data cache can be cleaned.



Example 21. Global Clean Operation

```
; Global Clean/Invalidate THE DATA CACHE
; R1 contains the virtual address of a region of cacheable memory reserved for
; this clean operation
; R0 is the loop count; Iterate 1024 times which is the number of lines in the
; data cache

;; Macro ALLOCATE performs the line-allocation cache operation on the
;; address specified in register Rx.
;;
MACRO ALLOCATE Rx
    MCR P15, 0, Rx, C7, C2, 5
ENDM

MOV R0, #1024
LOOP1:
ALLOCATE R1          ; Allocate a line at the virtual address
                    ; specified by R1.
ADD R1, R1, #32      ; Increment the address in R1 to the next cache line
SUBS R0, R0, #1      ; Decrement loop count
BNE LOOP1
;
; Clean the Mini-data Cache
; Can't use line-allocate command, so cycle 2KB of unused data through.
; R2 contains the virtual address of a region of cacheable memory reserved for
; cleaning the Mini-data Cache
; R0 is the loop count; Iterate 64 times which is the number of lines in the
; Mini-data Cache.

MOV R0, #64
LOOP2:
LDR R3, [R2], #32 ; Load and increment to next cache line
SUBS R0, R0, #1   ; Decrement loop count
BNE LOOP2
;
; Invalidate the data cache and mini-data cache
MCR P15, 0, R0, C7, C6, 0
;
```

The line-allocate operation does not require physical memory to exist at the virtual address specified by the instruction, since it does not generate a load/fill request to external memory. Also, the line-allocate operation does not set the 32 bytes of data associated with the line to any known value. Reading this data will produce unpredictable results.

The line-allocate command will not operate on the mini Data Cache, so system software must clean this cache by reading 2KByte of contiguous unused data into it. This data must be unused and reserved for this purpose so that it will not already be in the cache. It must reside in a page that is marked as mini Data Cache cacheable.

The time it takes to execute a global clean operation depends on the number of dirty lines in cache.



3.6.4 Re-configuring the Data Cache as Data RAM

Software has the ability to lock tags associated with 32-byte lines in the data cache, thus creating the appearance of data RAM. Any subsequent access to this line will always hit the cache unless it is invalidated. Once a line is locked into the data cache it is no longer available for cache allocation on a line fill. Up to 28 lines in each set can be reconfigured as data RAM, such that the maximum data RAM size is 28 Kbytes.

Hardware does not support locking lines into the mini-data cache; any attempt to do this will produce unpredictable results.

There are two methods for locking tags into the data cache; the method of choice depends on the application. One method is used to lock data that resides in external memory into the data cache and the other method is used to re-configure lines in the data cache as data RAM. Locking data from external memory into the data cache is useful for lookup tables, constants, and any other data that is frequently accessed. Re-configuring a portion of the data cache as data RAM is useful when an application needs scratch memory (bigger than the register file can provide) for frequently used variables. These variables may be strewn across memory, making it advantageous for software to pack them into data RAM memory.

Refer to the *Intel XScale® Core Developers Manual* for code examples.

Tags can be locked into the data cache by enabling the data cache lock mode bit located in coprocessor 15, register 9. Once enabled, any new lines allocated into the data cache will be locked down.

Note that the **PLD** instruction will not affect the cache contents if it encounters an error while executing. For this reason, system software should ensure the memory address used in the **PLD** is correct. If this cannot be ascertained, replace the **PLD** with a **LDR** instruction that targets a scratch register.

Lines are locked into a set starting at way 0 and may progress up to way 27; which set a line gets locked into depends on the set index of the virtual address of the request. [Figure 19](#) is an example of where lines of code may be locked into the cache along with how the round-robin pointer is affected.

Software can lock down data located at different memory locations. This may cause some sets to have more locked lines than others as shown in [Figure 19](#).

Lines are unlocked in the data cache by performing an unlock operation.

Before locking, the programmer must ensure that no part of the target data range is already resident in the cache. The Intel XScale® core will not refetch such data, which will result in it not being locked into the cache. If there is any doubt as to the location of the targeted memory data, the cache should be cleaned and invalidated to prevent this scenario. If the cache contains a locked region which the programmer wishes to lock again, then the cache must be unlocked before being cleaned and invalidated.



3.6.5 Write Buffer/Fill Buffer Operation and Control

The write buffer is always enabled which means stores to external memory will be buffered. The K bit in the Auxiliary Control Register (CP15, register 1) is a global enable/disable for allowing coalescing in the write buffer. When this bit disables coalescing, no coalescing will occur regardless the value of the page attributes. If this bit enables coalescing, the page attributes X, C, and B are examined to see if coalescing is enabled for each region of memory.

All reads and writes to external memory occur in program order when coalescing is disabled in the write buffer. If coalescing is enabled in the write buffer, writes may occur out of program order to external memory. Program correctness is maintained in this case by comparing all store requests with all the valid entries in the fill buffer.

The write buffer and fill buffer support a drain operation, such that before the next instruction executes, all the Intel XScale® core data requests to external memory have completed.

Writes to a region marked non-cacheable/non-bufferable (page attributes C, B, and X all 0) will cause execution to stall until the write completes.

If software is running in a privileged mode, it can explicitly drain all buffered writes.

3.7 Configuration

The System Control Coprocessor (CP15) configures the MMU, caches, buffers and other system attributes. Where possible, the definition of CP15 follows the definition of the StrongARM* products. Coprocessor 14 (CP14) contains the performance monitor registers and the trace buffer registers.

CP15 is accessed through MRC and MCR coprocessor instructions and allowed only in privileged mode. Any access to CP15 in user mode or with LDC or STC coprocessor instructions will cause an undefined instruction exception.

CP14 registers can be accessed through MRC, MCR, LDC, and STC coprocessor instructions and allowed only in privileged mode. Any access to CP14 in user mode will cause an undefined instruction exception.

The Intel XScale® core Coprocessors, CP15 and CP14, do not support access via CDP, MRRC, or MCRR instructions. An attempt to access these coprocessors with these instructions will result in an Undefined Instruction exception.

Many of the MCR commands available in CP15 modify hardware state sometime after execution. A software sequence is available for those wishing to determine when this update occurs.

Like certain other ARM* architecture products, the Intel XScale® core includes an extra level of virtual address translation in the form of a PID (Process ID) register and associated logic. Privileged code needs to be aware of this facility because, when interacting with CP15, some addresses are modified by the PID and others are not.

An address that has yet to be modified by the PID (“PIDified”) is known as a *virtual address* (VA). An address that has been through the PID logic, but not translated into a physical address, is a *modified virtual address* (MVA). Non-privileged code always deals with VAs, while privileged code that programs CP15 occasionally needs to use MVAs. For details refer to the *Intel XScale® Core Developers Manual*.



3.8 Performance Monitoring

The Intel XScale® core hardware provides two 32-bit performance counters that allow two unique events to be monitored simultaneously. In addition, the Intel XScale® core implements a 32-bit clock counter that can be used in conjunction with the performance counters; its sole purpose is to count the number of core clock cycles which is useful in measuring total execution time.

The Intel XScale® core can monitor either occurrence events or duration events. When counting occurrence events, a counter is incremented each time a specified event takes place and when measuring duration, a counter counts the number of processor clocks that occur while a specified condition is true. If any of the 3 counters overflow, an IRQ or FIQ will be generated if it's enabled. Each counter has its own interrupt enable. The counters continue to monitor events even after an overflow occurs, until disabled by software. Refer to the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for more detail.

Each of these counters can be programmed to monitor any one of various events.

To further augment performance monitoring, the Intel XScale® core clock counter can be used to measure the executing time of an application. This information combined with a duration event can feedback a percentage of time the event occurred with respect to overall execution time.

Each of the three counters and the performance monitoring control register are accessible through Coprocessor 14 (CP14), registers 0-3. Access is allowed in privileged mode only.

The following are a few notes about controlling the performance monitoring mechanism:

- An interrupt will be reported when a counter's overflow flag is set and its associated interrupt enable bit is set in the PMNC register. The interrupt will remain asserted until software clears the overflow flag by writing a one to the flag that is set. *Note:* the product specific interrupt unit and the CPSR must have enabled the interrupt in order for software to receive it.
- The counters continue to record events even after they overflow.

3.8.1 Performance Monitoring Events

Table 26 lists events that may be monitored by the PMU. Each of the Performance Monitor Count Registers (PMN0 and PMN1) can count any listed event. Software selects which event is counted by each PMNx register by programming the evtCountx fields of the PMNC register.

Table 26. Performance Monitoring Events (Sheet 1 of 2)

Event Number (evtCount0 or evtCount1)	Event Definition
0x0	Instruction cache miss requires fetch from external memory.
0x1	Instruction cache cannot deliver an instruction. This could indicate an ICache miss or an ITLB miss. This event will occur every cycle in which the condition is present.
0x2	Stall due to a data dependency. This event will occur every cycle in which the condition is present.
0x3	Instruction TLB miss.
0x4	Data TLB miss.
0x5	Branch instruction executed, branch may or may not have changed program flow.
0x6	Branch mispredicted. (B and BL instructions only.)

Table 26. Performance Monitoring Events (Sheet 2 of 2)

Event Number (evtCount0 or evtCount1)	Event Definition
0x7	Instruction executed.
0x8	Stall because the data cache buffers are full. This event will occur every cycle in which the condition is present.
0x9	Stall because the data cache buffers are full. This event will occur once for each contiguous sequence of this type of stall.
0xA	Data cache access, not including Cache Operations
0xB	Data cache miss, not including Cache Operations
0xC	Data cache write-back. This event occurs once for each 1/2 line (four words) that are written back from the cache.
0xD	Software changed the PC. This event occurs any time the PC is changed by software and there is not a mode change. For example, a mov instruction with PC as the destination will trigger this event. Executing a swi from User mode will not trigger this event, because it will incur a mode change.
0x10 through 0x17	Refer to the <i>Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for more details.
all others	Reserved, unpredictable results

Some typical combination of counted events are listed in this section and summarized in [Table 27](#). In this section, we call such an event combination a *mode*.

Table 27. Some Common Uses of the PMU

Mode	PMNC.evtCount0	PMNC.evtCount1
Instruction Cache Efficiency	0x7 (instruction count)	0x0 (ICache miss)
Data Cache Efficiency	0xA (Dcache access)	0xB (DCache miss)
Instruction Fetch Latency	0x1 (ICache cannot deliver)	0x0 (ICache miss)
Data/Bus Request Buffer Full	0x8 (DBuffer stall duration)	0x9 (DBuffer stall)
Stall/Writeback Statistics	0x2 (data stall)	0xC (DCache writeback)
Instruction TLB Efficiency	0x7 (instruction count)	0x3 (ITLB miss)
Data TLB Efficiency	0xA (Dcache access)	0x4 (DTLB miss)

3.8.1.1 Instruction Cache Efficiency Mode

PMN0 totals the number of instructions that were executed, which does not include instructions fetched from the instruction cache that were never executed. This can happen if a branch instruction changes the program flow; the instruction cache may retrieve the next sequential instructions after the branch, before it receives the target address of the branch.

PMN1 counts the number of instruction fetch requests to external memory. Each of these requests loads 32 bytes at a time.

Statistics derived from these two events:

- Instruction cache miss-rate. This is derived by dividing PMN1 by PMN0.
- *The average number of cycles it took to execute an instruction or commonly referred to as cycles-per-instruction (CPI)*. CPI can be derived by dividing CCNT by PMN0, where CCNT was used to measure total execution time.

3.8.1.2 Data Cache Efficiency Mode

PMN0 totals the number of data cache accesses, which includes cacheable and non-cacheable accesses, mini-data cache access and accesses made to locations configured as data RAM.

Note that **STM** and **LDM** will each count as several accesses to the data cache depending on the number of registers specified in the register list. **LDRD** will register two accesses.

PMN1 counts the number of data cache and mini-data cache misses. Cache operations do not contribute to this count.

The statistic derived from these two events is:

- Data cache miss-rate. This is derived by dividing PMN1 by PMN0.

3.8.1.3 Instruction Fetch Latency Mode

PMN0 accumulates the number of cycles when the instruction-cache is not able to deliver an instruction to the Intel XScale® core due to an instruction-cache miss or instruction-TLB miss. This event means that the processor core is stalled.

PMN1 counts the number of instruction fetch requests to external memory. Each of these requests loads 32 bytes at a time. This is the same event as measured in instruction cache efficiency mode and is included in this mode for convenience so that only one performance monitoring run is need.

Statistics derived from these two events:

- *The average number of cycles the processor stalled waiting for an instruction fetch from external memory to return.* This is calculated by dividing PMN0 by PMN1. If the average is high then the Intel XScale® core may be starved of the bus external to the Intel XScale® core.
- *The percentage of total execution cycles the processor stalled waiting on an instruction fetch from external memory to return.* This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time.

3.8.1.4 Data/Bus Request Buffer Full Mode

The Data Cache has buffers available to service cache misses or uncacheable accesses. For every memory request that the Data Cache receives from the processor core a buffer is speculatively allocated in case an external memory request is required or temporary storage is needed for an unaligned access. If no buffers are available, the Data Cache will stall the processor core. How often the Data Cache stalls depends on the performance of the bus external to the Intel XScale® core and what the memory access latency is for Data Cache miss requests to external memory. If the Intel XScale® core memory access latency is high, possibly due to starvation, these Data Cache buffers will become full. This performance monitoring mode is provided to see if the Intel XScale® core is being starved of the bus external to the Intel XScale® core, which will effect the performance of the application running on the Intel XScale® core.

PMN0 accumulates the number of clock cycles the processor is being stalled due to this condition and PMN1 monitors the number of times this condition occurs.

Statistics derived from these two events:

- *The average number of cycles the processor stalled on a data-cache access that may overflow the data-cache buffers.* This is calculated by dividing PMN0 by PMN1. This statistic lets you know if the duration event cycles are due to many requests or are attributed to just a few requests. If the average is high then the Intel XScale® core may be starved of the bus external to the Intel XScale® core.



- *The percentage of total execution cycles the processor stalled because a Data Cache request buffer was not available.* This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time.

3.8.1.5 Stall/Writeback Statistics

When an instruction requires the result of a previous instruction and that result is not yet available, the Intel XScale® core stalls in order to preserve the correct data dependencies. PMN0 counts the number of stall cycles due to data-dependencies. Not all data-dependencies cause a stall; only the following dependencies cause such a stall penalty:

- Load-use penalty: attempting to use the result of a load before the load completes. To avoid the penalty, software should delay using the result of a load until it's available. This penalty shows the latency effect of data-cache access.
- Multiply/Accumulate-use penalty: attempting to use the result of a multiply or multiply-accumulate operation before the operation completes. Again, to avoid the penalty, software should delay using the result until it's available.
- ALU use penalty: there are a few isolated cases where back to back ALU operations may result in one cycle delay in the execution.

PMN1 counts the number of writeback operations emitted by the data cache. These writebacks occur when the data cache evicts a dirty line of data to make room for a newly requested line or as the result of clean operation (CP15, register 7).

Statistics derived from these two events:

- *The percentage of total execution cycles the processor stalled because of a data dependency.* This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time. Often a compiler can reschedule code to avoid these penalties when given the right optimization switches.
- Total number of data writeback requests to external memory can be derived solely with PMN1.

3.8.1.6 Instruction TLB Efficiency Mode

PMN0 totals the number of instructions that were executed, which does not include instructions that were translated by the instruction TLB and never executed. This can happen if a branch instruction changes the program flow; the instruction TLB may translate the next sequential instructions after the branch, before it receives the target address of the branch.

PMN1 counts the number of instruction TLB table-walks, which occurs when there is a TLB miss. If the instruction TLB is disabled PMN1 will not increment.

Statistics derived from these two events:

- Instruction TLB miss-rate. This is derived by dividing PMN1 by PMN0.
- *The average number of cycles it took to execute an instruction or commonly referred to as cycles-per-instruction (CPI).* CPI can be derived by dividing CCNT by PMN0, where CCNT was used to measure total execution time.



3.8.1.7 Data TLB Efficiency Mode

PMN0 totals the number of data cache accesses, which includes cacheable and non-cacheable accesses, mini-data cache access and accesses made to locations configured as data RAM.

Note that **STM** and **LDM** will each count as several accesses to the data TLB depending on the number of registers specified in the register list. **LDRD** will register two accesses.

PMN1 counts the number of data TLB table-walks, which occurs when there is a TLB miss. If the data TLB is disabled PMN1 will not increment.

The statistic derived from these two events is:

- Data TLB miss-rate. This is derived by dividing PMN1 by PMN0.

3.8.2 Multiple Performance Monitoring Run Statistics

Even though only two events can be monitored at any given time, multiple performance monitoring runs can be done, capturing different events from different modes. For example, the first run could monitor the number of writeback operations (PMN1 of mode, Stall/Writeback) and the second run could monitor the total number of data cache accesses (PMN0 of mode, Data Cache Efficiency). From the results, a percentage of writeback operations to the total number of data accesses can be derived.

3.9 Performance Considerations

This section describes relevant performance considerations that compiler writers, application programmers and system designers need to be aware of to efficiently use the Intel XScale® core. Performance numbers discussed here include interrupt latency, branch prediction, and instruction latencies.

3.9.1 Interrupt Latency

Minimum Interrupt Latency is defined as the minimum number of cycles from the assertion of any interrupt signal (IRQ or FIQ) to the execution of the instruction at the vector for that interrupt. The point at which the assertion begins is TBD. This number assumes best case conditions exist when the interrupt is asserted, e.g., the system isn't waiting on the completion of some other operation.

A sometimes more useful number to work with is the *Maximum Interrupt Latency*. This is typically a complex calculation that depends on what else is going on in the system at the time the interrupt is asserted. Some examples that can adversely affect interrupt latency are:

- the instruction currently executing could be a 16-register LDM,
- the processor could fault just when the interrupt arrives,
- the processor could be waiting for data from a load, doing a page table walk, etc., and
- high core to system (bus) clock ratios.

Maximum Interrupt Latency can be reduced by:

- ensuring that the interrupt vector and interrupt service routine are resident in the instruction cache. This can be accomplished by locking them down into the cache.



- removing or reducing the occurrences of hardware page table walks. This also can be accomplished by locking down the application's page table entries into the TLBs, along with the page table entry for the interrupt service routine.

3.9.2 Branch Prediction

The Intel XScale® core implements dynamic branch prediction for the ARM* instructions **B** and **BL** and for the Thumb instruction **B**. Any instruction that specifies the PC as the destination is predicted as not taken. For example, an **LDR** or a **MOV** that loads or moves directly to the PC will be predicted not taken and incur a branch latency penalty.

These instructions -- ARM **B**, ARM **BL** and Thumb **B** -- enter into the branch target buffer when they are "taken" for the first time. (A "taken" branch refers to when they are evaluated to be true.) Once in the branch target buffer, the Intel XScale® core dynamically predicts the outcome of these instructions based on previous outcomes. Table 28 shows the branch latency penalty when these instructions are correctly predicted and when they are not. A penalty of zero for correct prediction means that the Intel XScale® core can execute the next instruction in the program flow in the cycle following the branch.

Table 28. Branch Latency Penalty

Core Clock Cycles		Description
ARM*	Thumb	
+0	+ 0	Predicted Correctly. The instruction is in the branch target cache and is correctly predicted.
+4	+ 5	Mispredicted. There are three occurrences of branch misprediction, all of which incur a 4-cycle branch delay penalty. 1. The instruction is in the branch target buffer and is predicted not-taken, but is actually taken. 2. The instruction is not in the branch target buffer and is a taken branch. 3. The instruction is in the branch target buffer and is predicted taken, but is actually not-taken

3.9.3 Addressing Modes

All load and store addressing modes implemented in the Intel XScale® core do not add to the instruction latencies numbers.

3.9.4 Instruction Latencies

The latencies for all the instructions are shown in the following sections with respect to their functional groups: branch, data processing, multiply, status register access, load/store, semaphore, and coprocessor.

The following section explains how to read these tables.



3.9.4.1 Performance Terms

- **Issue Clock (cycle 0)**
The first cycle when an instruction is decoded *and* allowed to proceed to further stages in the execution pipeline (i.e., when the instruction is actually issued).
- **Cycle Distance from A to B**
The cycle distance from cycle *A* to cycle *B* is $(B-A)$ -- that is, the number of cycles from the start of cycle *A* to the start of cycle *B*. Example: the cycle distance from cycle 3 to cycle 4 is one cycle.
- **Issue Latency**
The cycle distance *from* the first issue clock of the current instruction *to* the issue clock of the next instruction. The actual number of cycles can be influenced by cache-misses, resource-dependency stalls, and resource availability conflicts.
- **Result Latency**
The cycle distance *from* the first issue clock of the current instruction *to* the issue clock of the first instruction that can use the result without incurring a resource dependency stall. The actual number of cycles can be influenced by cache-misses, resource-dependency stalls, and resource availability conflicts.
- **Minimum Issue Latency (without Branch Misprediction)**
The minimum cycle distance *from* the issue clock of the current instruction *to* the first possible issue clock of the next instruction assuming best case conditions (i.e., that the issuing of the next instruction is not stalled due to a resource dependency stall; the next instruction is immediately available from the cache or memory interface; the current instruction does not incur resource dependency stalls during execution that can not be detected at issue time; and if the instruction uses dynamic branch prediction, correct prediction is assumed).
- **Minimum Result Latency**
The required minimum cycle distance *from* the issue clock of the current instruction *to* the issue clock of the first instruction that can use the result without incurring a resource dependency stall assuming best case conditions (i.e., that the issuing of the next instruction is not stalled due to a resource dependency stall; the next instruction is immediately available from the cache or memory interface; and the current instruction does not incur resource dependency stalls during execution that can not be detected at issue time).
- **Minimum Issue Latency (with Branch Misprediction)**
The minimum cycle distance *from* the issue clock of the current branching instruction *to* the first possible issue clock of the next instruction. This definition is identical to *Minimum Issue Latency* except that the branching instruction has been mispredicted. It is calculated by adding *Minimum Issue Latency (without Branch Misprediction)* to the minimum branch latency penalty number from [Table 28](#), which is four cycles.
- **Minimum Resource Latency**
The minimum cycle distance from the issue clock of the current multiply instruction to the issue clock of the next multiply instruction assuming the second multiply does not incur a data dependency and is immediately available from the instruction cache or memory interface.
[Example 22](#) contains a code fragment and an example of computing latencies.

Example 22. Computing Latencies

```
UMLALr6, r8, r0, r1
ADD r9, r10, r11
SUB r2, r8, r9
MOV r0, r1
```

Table 29 shows how to calculate Issue Latency and Result Latency for each instruction. Looking at the issue column, the **UMLAL** instruction starts to issue on cycle 0 and the next instruction, **ADD**, issues on cycle 2, so the Issue Latency for **UMLAL** is two. From the code fragment, there is a result dependency between the **UMLAL** instruction and the **SUB** instruction. In Table 29, **UMLAL** starts to issue at cycle 0 and the **SUB** issues at cycle 5, thus the Result Latency is five.

Table 29. Latency Example

Cycle	Issue	Executing
0	umlal (1st cycle)	--
1	umlal (2nd cycle)	umlal
2	add	umlal
3	sub (stalled)	umlal & add
4	sub (stalled)	umlal
5	sub	umlal
6	mov	sub
7	--	mov

3.9.4.2 Branch Instruction Timings

Table 30. Branch Instruction Timings (Those predicted by the BTB)

Mnemonic	Minimum Issue Latency when Correctly Predicted by the BTB	Minimum Issue Latency with Branch Misprediction
B	1	5
BL	1	5

Table 31. Branch Instruction Timings (Those not predicted by the BTB)

Mnemonic	Minimum Issue Latency when the branch is not taken	Minimum Issue Latency when the branch is taken
BLX(1)	N/A	5
BLX(2)	1	5
BX	1	5
Data Processing Instruction with PC as the destination	Same as Table 32	4 + numbers in Table 32
LDR PC,<>	2	8
LDM with PC in register list	3 + numreg ¹	10 + max (0, numreg-3)

1. numreg is the number of registers in the register list including the PC.



3.9.4.3 Data Processing Instruction Timings

Table 32. Data Processing Instruction Timings

Mnemonic	<shifter operand> is NOT a Shift/Rotate by Register		<shifter operand> is a Shift/Rotate by Register OR <shifter operand> is RRX	
	Minimum Issue Latency	Minimum Result Latency ¹	Minimum Issue Latency	Minimum Result Latency ¹
ADC	1	1	2	2
ADD	1	1	2	2
AND	1	1	2	2
BIC	1	1	2	2
CMN	1	1	2	2
CMP	1	1	2	2
EOR	1	1	2	2
MOV	1	1	2	2
MVN	1	1	2	2
ORR	1	1	2	2
RSB	1	1	2	2
RSC	1	1	2	2
SBC	1	1	2	2
SUB	1	1	2	2
TEQ	1	1	2	2
TST	1	1	2	2

1. If the next instruction needs to use the result of the data processing for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

3.9.4.4 Multiply Instruction Timings

Table 33. Multiply Instruction Timings (Sheet 1 of 2)

Mnemonic	Rs Value (Early Termination)	S-Bit Value	Minimum Issue Latency	Minimum Result Latency ¹	Minimum Resource Latency (Throughput)
MLA	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	3	2
		1	3	3	3
	all others	0	1	4	3
		1	4	4	4
MUL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	3	2
		1	3	3	3
	all others	0	1	4	3
		1	4	4	4



Table 33. Multiply Instruction Timings (Sheet 2 of 2)

Mnemonic	Rs Value (Early Termination)	S-Bit Value	Minimum Issue Latency	Minimum Result Latency ¹	Minimum Resource Latency (Throughput)
SMLAL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	2	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	2	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	2	RdLo = 4; RdHi = 5	4
		1	5	5	5
SMLALxy	N/A	N/A	2	RdLo = 2; RdHi = 3	2
SMLAWy	N/A	N/A	1	3	2
SMLAxy	N/A	N/A	1	2	1
SMULL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	1	RdLo = 4; RdHi = 5	4
		1	5	5	5
SMULWy	N/A	N/A	1	3	2
SMULxy	N/A	N/A	1	2	1
UMLAL	Rs[31:15] = 0x00000	0	2	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00	0	2	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	2	RdLo = 4; RdHi = 5	4
		1	5	5	5
UMULL	Rs[31:15] = 0x00000	0	1	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00	0	1	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	1	RdLo = 4; RdHi = 5	4
		1	5	5	5

1. If the next instruction needs to use the result of the multiply for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

Table 34. Multiply Implicit Accumulate Instruction Timings

Mnemonic	Rs Value (Early Termination)	Minimum Issue Latency	Minimum Result Latency	Minimum Resource Latency (Throughput)
MIA	Rs[31:16] = 0x0000 or Rs[31:16] = 0xFFFF	1	1	1
	Rs[31:28] = 0x0 or Rs[31:28] = 0xF	1	2	2
	all others	1	3	3
MIAXy	N/A	1	1	1
MIAPH	N/A	1	2	2

Table 35. Implicit Accumulator Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency	Minimum Resource Latency (Throughput)
MAR	2	2	2
MRA	1	(RdLo = 2; RdHi = 3) ¹	2

1. If the next instruction needs to use the result of the MRA for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

3.9.4.5 Saturated Arithmetic Instructions

Table 36. Saturated Data Processing Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
QADD	1	2
QSUB	1	2
QDADD	1	2
QDSUB	1	2

3.9.4.6 Status Register Access Instructions

Table 37. Status Register Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
MRS	1	2
MSR	2 (6 if updating mode bits)	1

3.9.4.7 Load/Store Instructions

Table 38. Load and Store Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
LDR	1	3 for load data; 1 for writeback of base
LDRB	1	3 for load data; 1 for writeback of base
LDRBT	1	3 for load data; 1 for writeback of base
LDRD	1 (+1 if Rd is R12)	3 for Rd; 4 for Rd+1; 2 for writeback of base
LDRH	1	3 for load data; 1 for writeback of base
LDRSB	1	3 for load data; 1 for writeback of base
LDRSH	1	3 for load data; 1 for writeback of base
LDRT	1	3 for load data; 1 for writeback of base
PLD	1	N/A
STR	1	1 for writeback of base
STRB	1	1 for writeback of base
STRBT	1	1 for writeback of base
STRD	2	1 for writeback of base
STRH	1	1 for writeback of base
STRT	1	1 for writeback of base

Table 39. Load and Store Multiple Instruction Timings

Mnemonic	Minimum Issue Latency ¹	Minimum Result Latency
LDM	3 - 23	1-3 for load data; 1 for writeback of base
STM	3 - 18	1 for writeback of base

1. LDM issue latency is 7 + N if R15 is in the register list and 2 + N if it is not. STM issue latency is calculated as 2 + N. N is the number of registers to load or store.

3.9.4.8 Semaphore Instructions

Table 40. Semaphore Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
SWP	5	5
SWPB	5	5

3.9.4.9 Coprocessor Instructions

Table 41. CP15 Register Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
MRC	4	4
MCR	2	N/A

Table 42. CP14 Register Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
MRC	7	7
MCR	7	N/A
LDC	10	N/A
STC	7	N/A

3.9.4.10 Miscellaneous Instruction Timing

Table 43. SWI Instruction Timings

Mnemonic	Minimum latency to first instruction of SWI exception handler
SWI	6

Table 44. Count Leading Zeros Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
CLZ	1	1

3.9.4.11 Thumb Instructions

The timing of Thumb instructions are the same as their equivalent ARM* instructions. This mapping can be found in the *ARM* Architecture Reference Manual*. The only exception is the Thumb BL instruction when H = 0; the timing in this case would be the same as an ARM* data processing instruction.

3.10 Test Features

This section gives a brief overview of the Intel XScale® core JTAG features. The Intel XScale® core provides test features compatible with the IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std. 1149.1). These features include a TAP controller, a 5-bit instruction register, and test data registers to support software debug. The Intel XScale® core also provides support for a boundary-scan register, device ID register, and other data test register. A full description of these features can be found in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

3.10.1 IXP2800 Network Processor Endianness

Endianness defines the way bytes are addressed within a word. A little endian system is one in which byte zero is the least significant byte (LSB) in the word and byte three is the most significant byte. A big endian system is one in which byte zero is the most significant byte (MSB) and byte 3 is the LSB. For example the value of 0x12345678 at address 0x0 in a 32 bit little endian system looks like this:



Table 45. Little Endian Encoding

Address/Byte Lane	0x0/ByteLane 3	0x0/ByteLane 2	0x0/ByteLane 1	0x0/ByteLane 0
Byte Value	0x12	0x34	0x56	0x78

The same value is stored in Big Endian system looks like this:

Table 46. Big Endian Encoding

Address/Byte Lane	0x0/ByteLane 3	0x0/ByteLane 2	0x0/ByteLane 1	0x0/ByteLane 0
Byte Value	0x78	0x56	0x34	0x12

Bits within a byte are always in Little Endian order. The least significant bit resides at bit location 0 and the most significant bit resides at bit location 7 (7:0).

The following conventions are used in this document:

- 1 Byte: 8-bit data
- 1 Word: 16-bit data
- 1 Long-word: 32-bit data
- Long Word Little Endian Format (LWLE) 32-bit data (0x12345678) arranged as {12 34 56 78}
- Long Word Little Endian Format (LWLE) 64-bit data 0x12345678 9ABCDE56 arranged as {12 34 56 78 9A BC DE 56}
- Long Word-Big Endian format (LWBE) 32-bit data (0x12345678) arranged as {78 56 34 12}
- Long Word-Big Endian format (LWBE) 64-bit data 0x12345678 9ABCDE56 arranged as {78 56 34 12, 56 DE BC 9A}

Endianness for the IXP2800 processor can be divided into three major categories:

- Read and write transactions initiated by the Intel XScale® core:
 - Reads initiated by Intel XScale® core
 - Writes initiated by Intel XScale® core
- SRAM and DRAM access:
 - 64-bit Data transfer between DRAM and the Intel XScale® core
 - Byte, word or long-word transfer between SRAM/DRAM and Intel XScale® core
 - Data transfer between SRAM/DRAM and PCI
 - Microengine initiated access to SRAM and DRAM
- PCI Accesses
 - the Intel XScale® core generated reads/writes to PCI in memory space
 - the Intel XScale® core generated read/write of external/internal PCI config registers



3.10.1.1 Read and Write Transactions Initiated by the Intel XScale® Core

The Intel XScale® core may be used in either a little endian or big endian configuration. The configuration affects the entire system in which the Intel XScale® microarchitecture exists. Software and hardware must agree on the byte ordering to be used. In software, a system's byte order is configured with CP15 register 1, the control register. Bit 7 of this register, the B bit, informs the processor of the byte order in use by the system. Note that this bit takes effect even if the MMU is not otherwise in use or enabled.

Though it is the responsibility of system hardware to assign correct byte lanes to each byte field in the data bus, in the IXP2800 it is left to the software to interpret byte lanes in accordance with the endianness of the system. As shown in Figure 24, system byte lanes 0–3 are connected directly to the Intel XScale® core byte lanes 0–3. What this means is that byte lane 0 (M[7:0]) of the system is connected to byte lane 0 (X[7:0]) of the Intel XScale® core, byte lane 1 (M[15:8]) of the system is connected to byte lane 1 (X[15:8]) of the Intel XScale® core and so on.

Interface operation of the Intel XScale® core and the rest of the IXP2800 can be divided into two parts:

- Intel XScale® core reading from the IXP2800
- Intel XScale® core writing to the IXP2800

3.10.1.1.1 Reads Initiated by the Intel XScale® Core

Intel XScale® core reads can be one of the following three types:

- Byte read
- 16-bits (word) read
- 32-bits (Long Word) read

Byte Read

When reading a byte, the Intel XScale® core generates the byte_enable that corresponds to the proper byte lane as defined by the endianness setting. Table 47 summarizes byte enable generation for this mode.

Table 47. Byte Enable Generation by the Intel XScale® Core for Byte Transfers in Little and Big Endian Systems

Byte# to be read	Byte Enables When System is Little Endian				Byte Enables When System is Big Endian			
	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]
Byte0	1	0	0	0	0	0	0	1
Byte1	0	1	0	0	0	0	1	0
Byte2	0	0	1	0	0	1	0	0
Byte3	0	0	0	1	1	0	0	0

The 4-to-1 mux steers the byte read into byte lane 0 location of the read register inside the Intel XScale® core. Select signals for the mux are generated based on endian setting and ByteEnable generated by the Intel XScale® core as defined in Figure 24.

16-bit (Word) Read

When reading a word, the Intel XScale® core generates the byte_enable that corresponds to the proper byte lane as defined by the endianness setting. Figure 25 summarizes byte enable generation for this mode.

The 4-to-1 mux steers Byte lane 0 or Byte lane 2 into Byte0 location of the read register inside the Intel XScale® core. The 2-to-1 mux steers Byte lane 1 or Byte lane 3 into Byte1 location of the read register inside the Intel XScale® core. The Intel XScale® core does not allow word access to an odd byte address. Select signals for the mux are generated based on endian setting and ByteEnable generated by the Intel XScale® core as defined in Figure 24. Table 48 summarizes byte enable generation for this mode.

Figure 24. Byte Steering for Read and Byte Enable Generation by the Intel XScale® Core

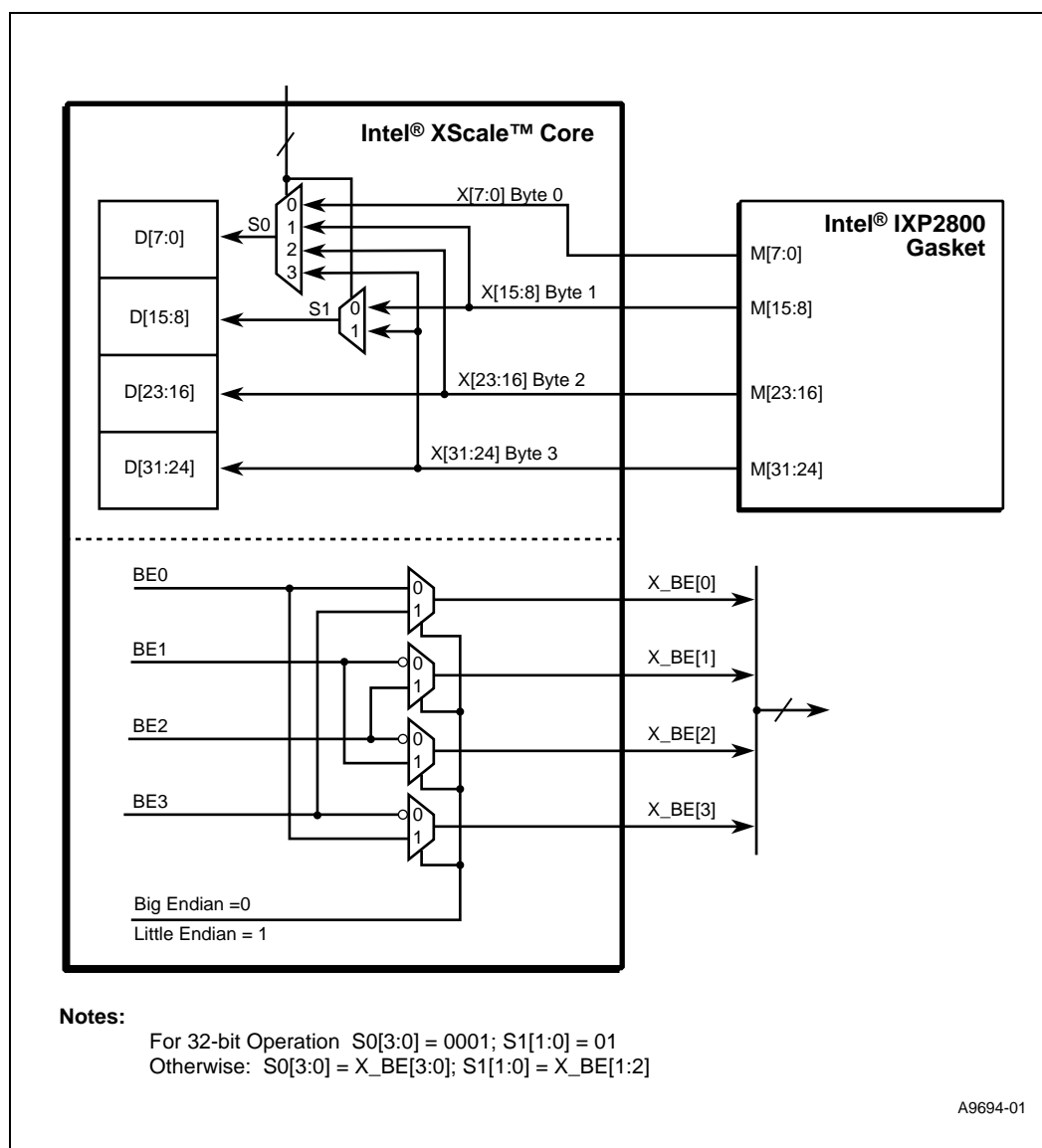


Table 48. Byte Enable Generation by the Intel XScale® Core for 16-bit Data Transfers in Little and Big Endian Systems

Word to be read	Byte Enables When System is Little Endian				Byte Enables When System is Big Endian			
	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]
Byte0 & Byte1	1	1	0	0	0	0	1	1
Byte2 & Byte3	0	0	1	1	1	1	0	0

32-bits (Long Word) Read

32-bits (long Word) reads are independent of endianness setting and byte lane 0 from the Intel XScale® core's data bus gets into Byte 0 location of the read register inside Intel XScale® core, byte lane 1 from Intel XScale® core's data bus gets into Byte1 location of the read register inside Intel XScale® core and so on. It is up to the software to deal with byte location properly based on the endian setting.

3.10.1.1.2 The Intel XScale® Core Writing to the IXP2800

Similar to reads, writes by Intel XScale® core can also be divided in following three categories:

- Byte Write
- Word Write (16-bits)
- Long Word write (32-bits)

Byte Write

When Intel XScale® core writes single byte to external memory, it puts the byte in the byte lane where it intends to write it along with the byte enable for that byte turned ON based on endian setting of the system. Intel XScale® core register bits [7:0] always contain the byte to be written regardless of the B-bit setting. For example if the Intel XScale® core wants to write to byte 0 in little endian system, it puts the byte in byte lane0 and turns X_BE[0] ON. If the system is big endian, in that case the Intel XScale® core puts byte0 in byte lane 3 and turns X_BE[3] ON. Other possible combinations of byte lanes and byte enables are shown in the Table 49. Other byte lanes besides the one currently driven by the Intel XScale® core contain undefined data.

Table 49. Byte Enable Generation by the Intel XScale® Core for Byte Writes in Little and Big Endian Systems

Byte# to be written	Byte Enables when system is Little Endian				Byte Enables when system is Big Endian			
	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]
Byte0	1	0	0	0	0	0	0	1
Byte1	0	1	0	0	0	0	1	0
Byte2	0	0	1	0	0	1	0	0
Byte3	0	0	0	1	1	0	0	0

Word Write (16-bits Write)

When the Intel XScale® core writes a 16-bit word to external memory, it puts the bytes in the byte lanes where it intends to write them along with the byte enables for those bytes turned ON based on the endian setting of the system. The Intel XScale® core does not allow a word write on an odd byte address. The Intel XScale® core register bits [15:0] always contain the word to be written regardless of the B-bit setting. For example if the Intel XScale® core wants to write one word to a little endian system at address 0x0002, it will copy byte0 to byte lane 2 and byte1 to byte lane 3 along with X_BE[2] and X_BE[3] turned ON. If the Intel XScale® core wants to write one word to a big endian system at address 0x0002, it will copy byte0 to byte lane 0 and byte1 to byte lane 1 along with X_BE[0] and X_BE[1] turned ON. Other possible combinations of byte lanes and byte enables are shown in Table 50. Other byte lanes besides the ones currently driven by the Intel XScale® core contain undefined data.

Table 50. Byte Enable Generation by the Intel XScale® Core for Word Writes in Little and Big-Endian Systems

Word to be written	Byte Enables When System is Little Endian				Byte Enables When System is Big Endian			
	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]	X_BE[0]	X_BE[1]	X_BE[2]	X_BE[3]
Byte0 & Byte1	1	1	0	0	0	0	1	1
Byte2 & Byte3	0	0	1	1	1	1	0	0

Long Word (32-bits) Write

The long word to be written is put on the Intel XScale® core's data bus with byte0 on X[7:0], byte1 on X[15:8], byte2 on X[23:16] and byte4 on X[31:24] (see Figure 25). All the byte enables are turned ON. A 32-bit long word write (0x12345678) by the Intel XScale® core to address 0x0000 irrespective of the endianness of the system causes byte0 (0x78) to be written to address 0x0000, byte1 (0x56) to address 0x0001, byte2 (0x34) to address 0x0002 and byte3 (0x12) to address 0x0003.

Figure 25. Intel XScale® Core Initiated Write to the IXP2800 Network Processor

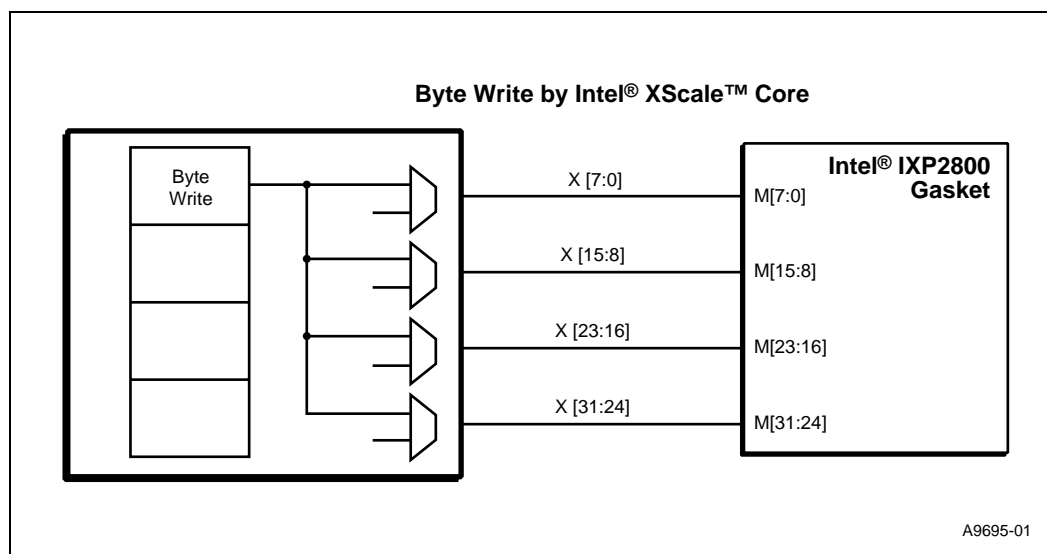
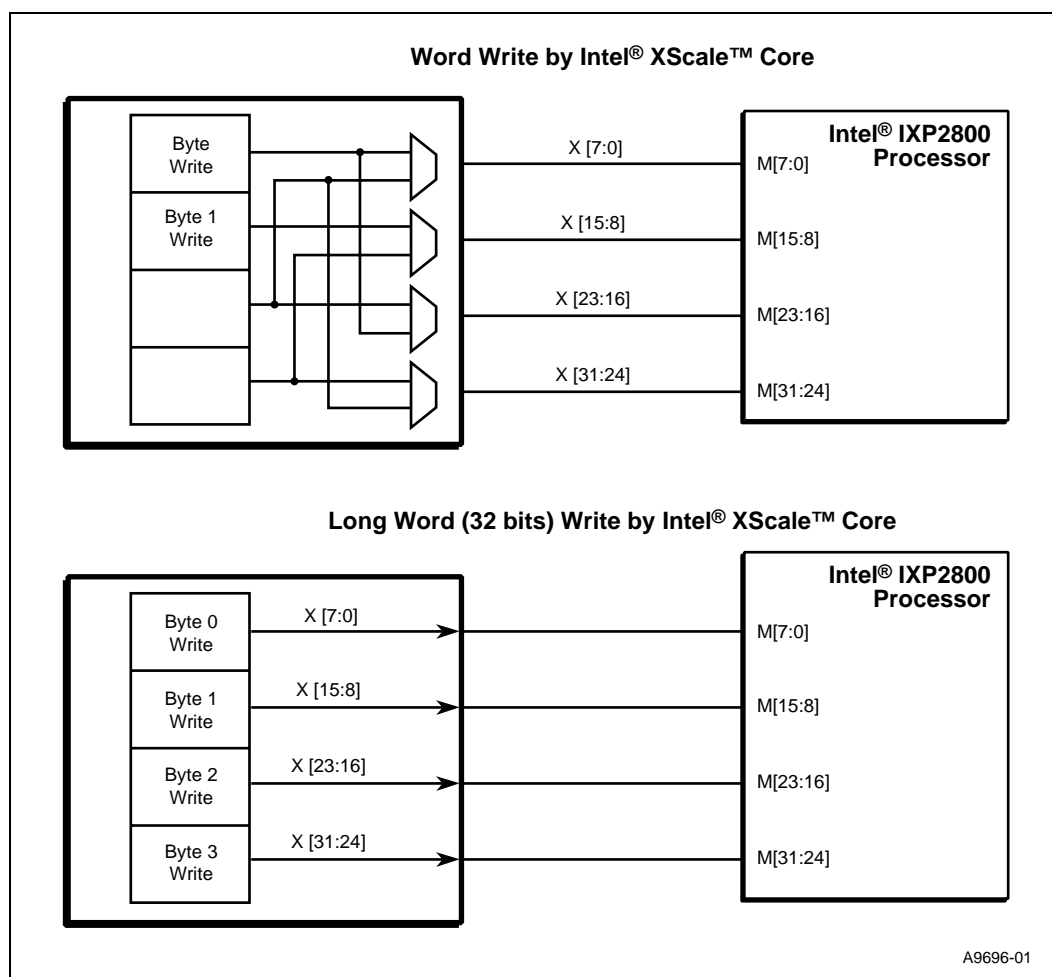


Figure 26. Intel XScale® Core Initiated Write to the IXP2800 Network Processor (Continued)





3.11 Intel XScale® Gasket Unit

3.11.1 Overview

The Intel XScale® core uses the Core Memory Bus (CMB) to communicate with the functional blocks. The rest of the IXP2800 Network Processor functional blocks use the Command Push Pull (CPP) as the global bus to pass data. Therefore the gasket is needed to translate Core Memory Bus commands to Command Push Pull commands.

This gasket has a set of local CSRs, including interrupt registers. These registers can be accessed by the Intel XScale® core via the gasket internal bus. The CSR Access Proxy (CAP) is allowed to only do a set on these interrupt registers.

The Intel XScale® core coprocessor bus is not used in the IXP2800 Network Processors, all accesses are only through the Command Memory Bus.

Figure 27 shows the block diagram of the global bus connections to the gasket.

The gasket unit has the following features:

- Interrupts are sent to the Intel XScale® core via the gasket, with the interrupt controller registers used for masking the interrupts.
- The gasket converts CMB reads and writes to CPP format.
- All the atomic operations are applied on SRAM and SCRATCH only, not DRAM.
- There is a stepping-stone sitting between the Intel XScale® core and the gasket. The Intel XScale® core runs at 600MHz to 700MHz. The gasket currently supports a 1:1 (IXP2800 Network Processor and 2:1 (IXP2400 Network Processor) clock ratio. For a 2:1 ratio, the Command Push Pull bus will be running at half of the frequency of the Intel XScale® core.
- In IXP2800 memory controllers, read after write ordering is enforced. There is no write after read enforcement for the Intel XScale® core. The gasket will perform enforcement by employing Content Addressable Memory (CAM) to detect a write to an address with read pending. This only applies for writes to SRAM.
- The gasket CPP interface contains one command bus, one D_Push bus, one D_Pull bus, one S_Push bus, one S_Pull bus, each with a 32-bit data width.

A maximum four outstanding reads and four outstanding writes from the Intel XScale® core are allowed.

[illegible]

3.11.2.1 Command Memory Bus to Command Push/Pull Conversion

Table 51 shows how many CPP commands are generated by the gasket from each CMB command. Write data is guaranteed to be 32 bit (long word) aligned. **Table 51** shows only the Store command. In the Load case, the gasket simply converts it to the CPP format. No command splitting is required. A Load can only be a byte (8 bits), a word (16 bits), long word (32 bits), or eight long words (8x32).

Table 51. CMB Write Command to CPP Command Conversion

Store Length	CPP SRAM Cmd Count	CPP DRAM Cmd Count	Remark
Byte, word, long word	1	1	SRAM uses 4-bit mask, DRAM uses an 8-bit mask.
2 long word	1 or 2	1 or 2	SRAM: If there is any mask bit detected as '0', two commands will be generated. DRAM: If it starts with odd word address, two commands will be generated.
3 long word	1 or 3	2	SRAM: If there is a mask bit of '0' detected, 3 SRAM commands will be generated. DRAM: always 2 DRAM commands.
4 long word	1 or 4	1 or 2	SRAM: If there is a mask bit of '0' detected, four commands will be generated. DRAM: If there is a mask bit of '0' detected, two commands will be generated.
8 long word			Not allowed in a write.

3.11.3 CAM Operation

In the SRAM controller, access ordering is guaranteed only for a read coming after a write. The gasket enforces order rules in the following two cases.

1. Write coming after a read.
2. Read-Modify-Write coming after read.

The address CAMing is on 8 word boundaries. The SRAM effective address is 28-bits. Deduct 5 bits (2 bits for the word address and 3 bits for 8 words), and the tag width for the CAM is 23-bits wide. The CAM only operates on SRAM accesses.

3.11.4 Atomic Operations

The Intel XScale® core has Swap (SWP) and Swap Byte (SWPB) instructions that generate an atomic read-write pair to a single address. These instructions are supported for the SRAM and Scratch space, and also to any other address space if it is done by a Read command followed by Write command.

cbiIO is asserted when a data cache request is initiated to a memory region with cacheable and bufferable bits in the translation table first-level descriptor set to zero. Also, if cbiIO is asserted during the CMB read portion of the SWP, then it also does a Read Command followed by Write Command, regardless of address. In those cases the SWP/SWPB is atomic with respect to processes running on the Intel XScale® core, but not with respect to the Microengines.



The following summarizes the Atomic operation.

Address Space	cbiIO	Operation
SRAM/Scratch	0	RMW Command
Not SRAM/Scratch	x	Read Command followed by Write Command
Any	1	Read Command followed by Write Command

When the Intel XScale® core presents the read command portion of the SWP it will assert the cbiLock signal. The gasket will ack the read and save the BufID in the push_ff. It will not arbitrate for the command bus at that time; rather it will wait for the corresponding write of the SWP (which will also have cbiLock asserted). At that time the gasket will arbitrate for the command bus to send a command with the atomic operation in the command field [the command is based on the address space chosen for the SRAM/Scratch, which has multiple aliased address ranges].

The SRAM or Scratch controller will pull the data, do the atomic read-modify-write, and then push the read data back. The gasket will use the saved BufID when returning the data to CMB.

Note: Unrelated reads, such as instruction and Page Table fetches, can come in the interval between the read-lock and write-unlock, and will be handled by the gasket. No other data reads or writes will come in that interval. Also, the Intel XScale® core will not wait for the SWP read data before presenting the write data.

The gasket uses address aliases to generate the following atomic operations.

- Bit Set
- Bit Clear
- Add
- Subtract
- Swap

For the alias address type of atomic operation, the Intel XScale® core will issue a SWP command with an alias address if it needs data return. The Intel XScale® core will use the write command with an alias address if it doesn't need data return.

Xscale_IF will not check the second address, as long as it detects one write after one read, both with cbiLock enabled. It will take the write address and put it in the command.

The summary of the rules for Atomic command in I/O space are.

- SWP to SRAM/Scratch and Not cbiIO, Xscale_IF generates an Atomic operation command.
- SWP to all other Addresses that are not SRAM/Scratch, will be treated as separate read and write commands. No Atomic command is generated.
- SWP to SRAM/Scratch and cbiIO, will be treated as separate read and write commands. No Atomic command is generated.

3.11.4.1 Intel XScale® Core Access to SRAM Q-Array

The Intel XScale® core can access the SRAM controllers queue function to do buffer allocation and freeing. Allocation does a SRAM dequeue (deq) operation, and freeing does a SRAM enqueue (enq) operation. Alias addresses are used as shown in Table 52 to access the freelist. Each SRAM channel supports up to 64 lists, so there are 64 addresses per channel.

Table 52. IXP2800 Network Processor SRAM Q-Array Access Alias Addresses

Channel	Address Range
0	0xCC00 0100 – 0xCC00 01FC
1	0xCC40 0100 – 0xCC40 01FC
2	0xCC80 0100 – 0xCC80 01FC
3	0xCCC0 0100 – 0xCCC0 01FC

Address 7:2 selects which Queue_Array entry within the SRAM channel is used.

Doing a load to an address in the table will do a deq, the SRAM controller returns the dequeued information (i.e. the buffer pointer) as the load data.

Doing a store to an address in the table will do an enq. The data to be enqueued is taken from the Intel XScale® core store data.

The gasket will generate command fields as follows, based on address and cbiLd:

```
Target_ID = SRAM (00 0010)
Command = deq (1011) if cbiLd, enq (1100) if ~cbiLd
Token[1:0] = 0x0
Byte_Mask = 0xFF
Length = 0x1
Address = {XScale_Address[23:22], XScale_Address[7:2], XScale_Write_Data[25:2]}
```

(Note: On command bus -- address[31:30] selects the SRAM channel, address[29:24] is Q_Array number; and address[23:0] is the SRAM longword address. For Dequeue, SRAM controller ignores address[23:0].)

3.11.5 I/O Transaction

Intel XScale® core can request an I/O transaction by asserting xsoCBI_IO concurrently with xsoCBI_Req. The value of xsoCBI_IO is undefined when xsoCBI_Req is not asserted. When the gasket sees an I/O request with xsoCBI_IO asserted, it will raise xsiCBR_Ack but will not acknowledge future requests until the IO transaction is complete. The gasket will check if all the command FIFOs and write data FIFOs are empty or not. It will also check if the command counters (SRAM and DRAM) are equal to zero. All these checks are to guarantee that:

- Writes are issued to the target, and targets have pulled the data.
- Pending reads have their data all back to the gasket.

When the gasket sees that all the conditions are satisfied, it will assert xsiCBR_SynchDone to the Intel XScale® core. XsiCBR_SynchDone is one cycle long and does not need to coincide with xsiCBR_DataValid.



3.11.6 Hash Access

Hash accesses are accomplished by the gasket Local_CSR accesses from the Intel XScale® core. There are two sets of registers in the gasket that are involved in Hash accesses.

- Four 32 bit XG_GCSR_Hash[3:0] registers for holding the data to be hashed and index returned as well.
- A XG_GCSR_CTR0(valid) register to hold the status of the Hash Access.

The procedure for the Intel XScale® core to setup a Hash access is as follows.

1. The Intel XScale® core writes data to XG_GCSR_Hash by Local_CSR access using address [X:yy:zz]. X selects Hash register set. yy selects hash_48, hash_64 or hash_128 mode. zz selects one of four Hash_Data registers.
2. Data write order is 3-2-1-0(for hash_128), 1-0(for hash_48 or hash_64). When the data write to Hash_Data[0] is performed, it triggers the Hash request to go out on the CPP bus. At the same time, XG_GCSR_Hash(valid) will be cleared by hardware.
3. The Intel XScale® core starts to poll Hash_Result_Valid periodically by Local_CSR read.
4. After some period of time, the Hash_Result is returned to XG_GCSR_Hash, and XG_GCSR_CTR0(valid) is set to indicate that Hash_Result is ready to be retrieved.
5. The Intel XScale® core issues a Local_CSR read to read back the Hash_Result.

Note, each Hash command requests only one index returned.

The Hash CSR is in the gasket local CSR space.

3.11.7 Gasket Local CSR

There are two sets of Control and Status registers residing in the gasket Local CSR space. ICSR refers to the Interrupt CSR. The ICSR address range is 0xd600_0000 - 0xd6ff_ffff. The Gasket CSR (GCSR) refers to the Hash CSRs and debug CSR. It has a range of 0xd700_0000 - 0xd7ff_ffff. GCSR is shown in [Table 53](#).

Note: The Gasket registers are defined in the *IXP2400/IXP2800 Network Processor Programmers Reference Manual*.

Table 53. GCSR Address Map(0xd700 0000)

Bits	Name	R/W	Description	Address Offset
[31:0]	XG_GCSR_HASH0	R/W	Hash word 0 Write from Intel XScale® core. Rd/Wr from CPP.	0x00 : for 48bit Hash 0x10 : for 64bit Hash 0x20 : for 128bit Hash
[31:0]	XG_GCSR_HASH1	R/W	Hash word 1 Write from Intel XScale® core. Rd/Wr from CPP.	0x04 : for 48bit Hash 0x14 : for 64bit Hash 0x24 : for 128bit Hash
[31:0]	XG_GCSR_HASH2	R/W	Hash word 2 Write from Intel XScale® core. Rd/Wr from CPP.	0x28 : for 128bit Hash
[31:0]	XG_GCSR_HASH3	R/W	Hash word 3 Write from Intel XScale® core. Rd/Wr from CPP.	0x2c : for 128bit Hash
[31:0]	XG_GCSR_CTR0	R	[31:1] reserved. [0] hash valid flag. Read from Intel XScale® core. Set by LCSR control.	0x30
[31:0]	XG_GCSR_CTR1	R/W	[31:1] reserved. [0] Break_Function When set to 1, the debug break signal is used to stop the clocks. When set to 0, the debug break signal is used to cause an Intel XScale® core debug breakpoint	0x3c

3.11.8 Interrupt

The Intel XScale® core CSR controller contains local CSR(s) and interrupts inputs from multiple sources. The diagram in [Figure 28](#) shows the flow through the controller.

Within the Interrupt/CSR Register block there are raw status registers, enable registers, and local CSR(s). The raw status registers are the un-masked interrupt status. These interrupt status are masked or steered to the Intel XScale® core's IRQ or FIQ inputs by multiple levels of enable registers.

Refer to [Figure 29](#).

- {IRQ,FIQ}Status = (RawStatus & {IRQ,FIQ}Enable)
- {IRQ,FIQ}ErrorStatus = (ErrorRawStatus & {IRQ,FIQ}ErrorEnable)
- {IRQ,FIQ}ThreadStatus_\$_# = ({IRQ,FIQ}ThreadRawStatus_\$_# & {IRQ,FIQ}ThreadEnable_\$_#)

Each interrupt input is visible in the RawStatusRegister and is masked or steered by two level of interrupt enable registers. The error and thread status are masked by one level of enable registers. Their combination along with other interrupt sources contributes to the RawStatusReg. The RawStatus is masked via IRQEnable/FIQEnable to trigger the IRQ and FIQ interrupt to the Intel XScale® core.

The enable register's bits are set and cleared through EnableSet and EnableClear registers. The Status, RawStatus, and Enable Registers are read-only, and EnableSet and EnableClear are write-only. Also, Enable and EnableSet share the same address for reads and writes respectively.

Note that software needs to take into account the delay between the clearing of an interrupt condition and having its status updated in the RawStatus registers. Also in the case of simultaneous writes to the same registers, the value of the last write is recorded.

Figure 28. Flow Through the Intel XScale® Core Interrupt Controller

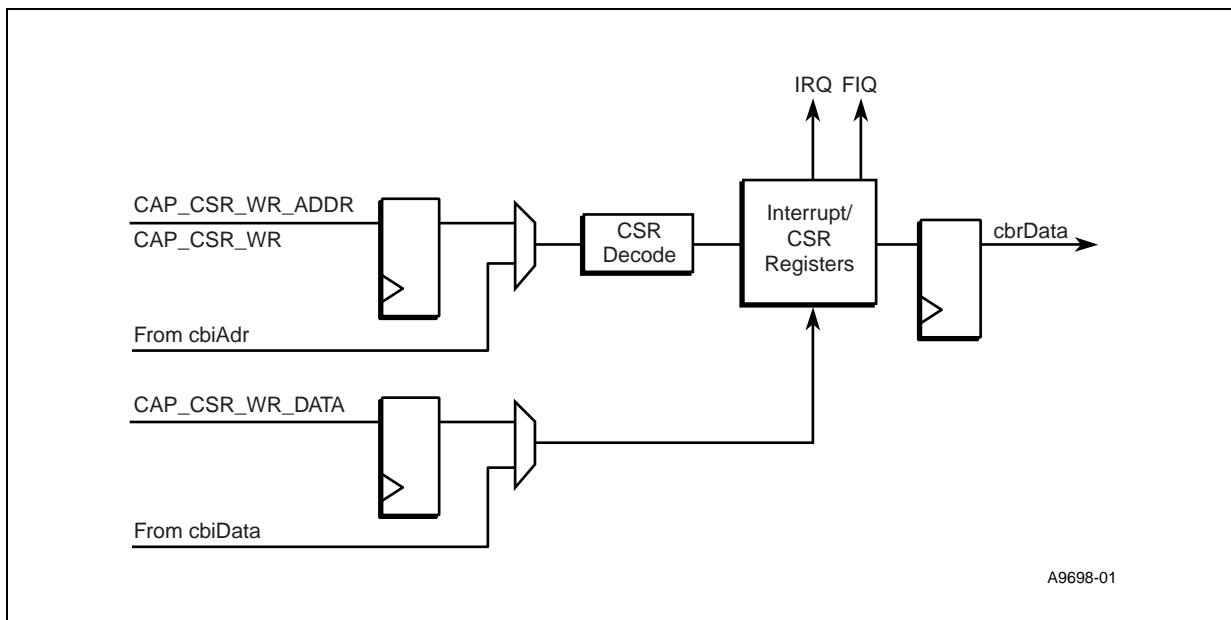
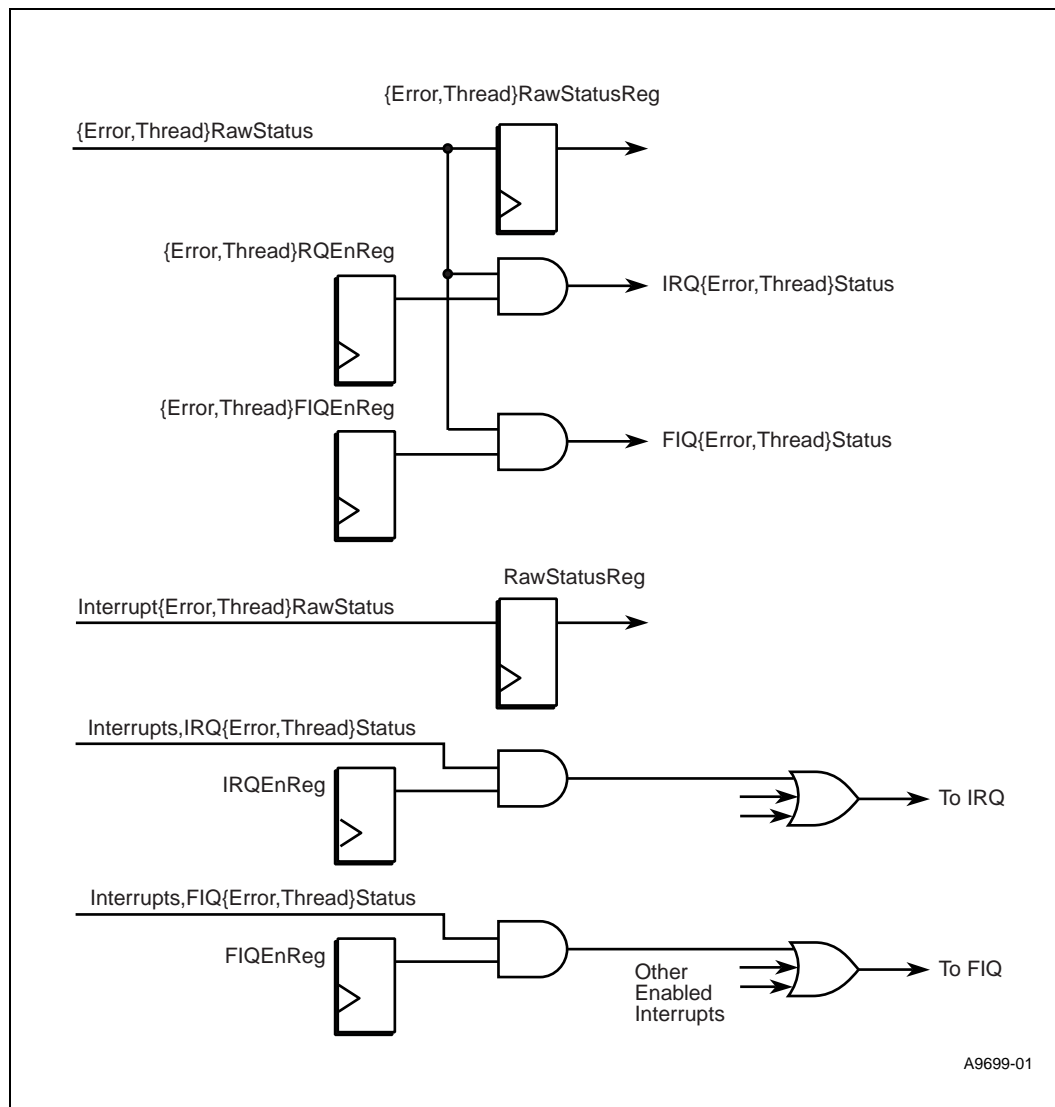


Figure 29. Interrupt Mask Block Diagram





3.12 Intel XScale® Core Peripheral Interface

This section describes the Intel XScale® core Peripheral Interface unit (XPI). The XPI is the block that connects to all the slow and serial interfaces that communicate with the Intel XScale® core through the APB bus. These can also be accessed by the Microengines and PCI unit.

This section does not describe the Intel XScale® core interface protocol, only how to interface with the peripheral devices connected to the core. The I/O units described are:

- UART
- Watchdog timers
- GPIO
- SlowPort

All the peripheral units are memory mapped from the Intel XScale® core point of view.

3.12.1 XPI Overview

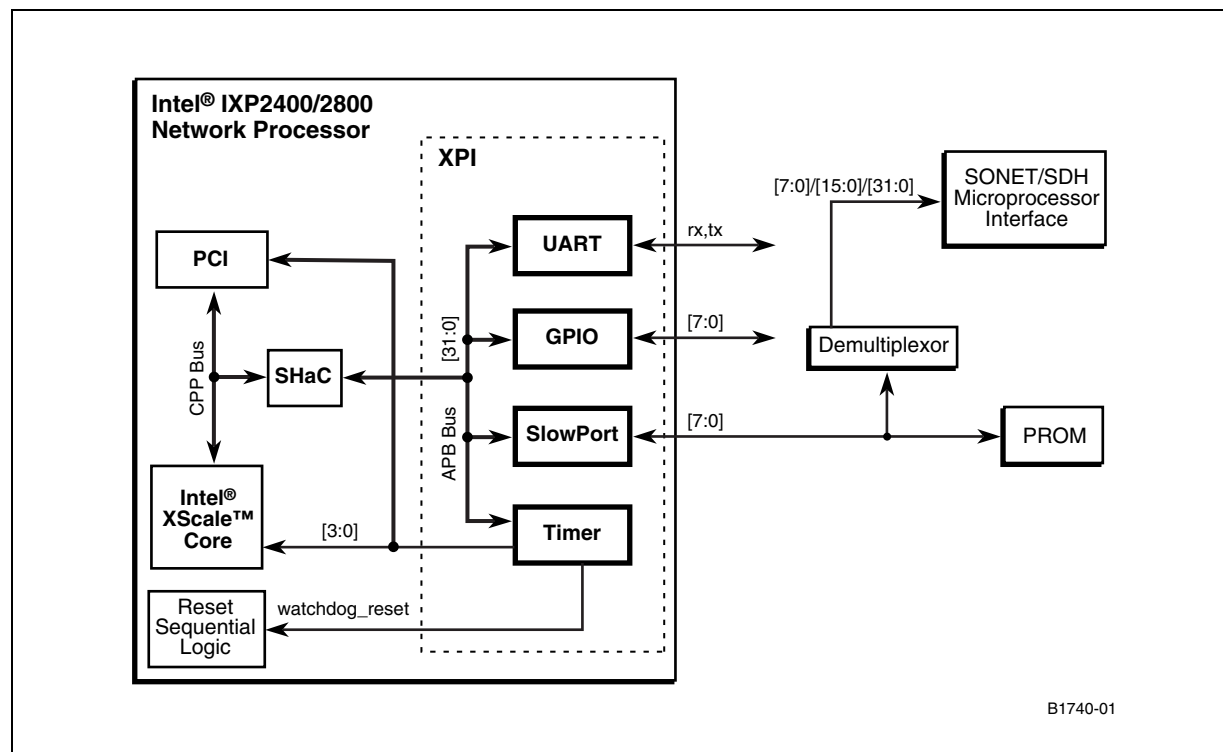
Figure 30 shows the XPI location in the IXP2800 Network Processor. The XPI receives read and write commands from the Command Push Pull bus to addresses the memory has mapped to I/O devices.

The SHaC (Scratchpad, Hash Unit, and CSRs) acts like a bridge to control the access from the Intel XScale® core or other host (like the PCI Unit). The extended APB bus is used to communicate between the XPI and the SHaC. The extended APB has only one signal, APB_RDY, added. This signal is used to tell the SHaC when the transaction should be terminated.

The XPI is responsible for passing the data between the extended APB bus and the internal fubs, like the UART, GPIO, Timer, and SlowPort, which will in turn pass these data to an external peripheral device with a corresponding bus format.

The XPI is always a master on the SlowPort bus and all the SlowPort devices act like slaves. On the other side, the SHaC is always the master and the XPI is the slave with respect to the APB.

Figure 30. XPI Interfaces (B0) for 2400/2800



3.12.1.1 Data Transfers

The current rate for data transfers is four bytes, except for the SlowPort. The 8-bit and 16-bit accesses are only available in the SlowPort bus. The devices connected to the SlowPort dictate this data width. The user has to configure the data width register resident in the SlowPort in order to run a different type of data transaction. There is no burst to SlowPort.

3.12.1.2 Data Alignment

For all the CSR accesses, a 32-bit data bus is assumed. Therefore, the lower two bits of the address bus are ignored.

However, for the SlowPort accesses, 8-bit, 16-bit, or 32-bit data access is dictated by the external device connected to the SlowPort. The APB Bus should be able to match the data width according to which devices it is talking to.

See [Table 54](#) for additional details on data alignment.

Table 54. Data Transaction Alignment

Interface Units	APB Bus	Read	Write
GRegs	32 bits	32 bits	32 bits
UART	32 bits	32 bits	32 bits
GPIO	32 bits	32 bits	32 bits
Timer	32 bits	32 bits	32 bits
SlowPort Microprocessor Access	8 bits	8 bits	8 bits
	16 bits	16 bits	16 bits
	32 bits	32 bits	32 bits
SlowPort Flash Memory Access ¹	32 bits for 32-bit read mode, 8 bits for register read mode; 8 bits for write;	Assemble 8 bits into 32-bit data for 32-bit read mode; 8 bits for register read mode (8-bit read mode).	8 bits
CSR Access	32 bits	32 bits	32 bits

1. The flash memory interface only supports 8-bit wide flash devices. APB write transactions are assumed to be 8-bits wide, and correspond to one write cycle at the flash interface. APB read transactions are assumed to be 32-bits wide, and correspond to four flash read cycles for the 32-bit read mode set in the SP_FRM register. However, for the flash register read mode (8-bit read mode), it only needs one flash read cycle of 8-bit data and passes it back to APB directly. By default, the 32-bit read mode is set. It is advisable to stay in this mode most of the time and not change them dynamically during accesses.

3.12.1.3 Address Spaces for XPI Internal Devices

Table 55 shows the address space assignment for XPI devices.

Table 55. Address spaces for XPI Internal Devices

Units	Starting Address	Ending Address
GPIO	0xC0010000	0xC0010040
TIMER	0xC0020000	0xC0020040
UART	0xC0030000	0xC003001C
PMU	0xC0050000	0xC0050E00
SlowPort CSR	0xC0080000	0xC0080028
SlowPort Device	0xC4000000	0xC7FFFFFFF



3.12.2 UART Overview

The UART performs serial-to-parallel conversion on data characters received from a peripheral device and parallel-to-serial conversion on data characters received from the network processor. The processor can read the complete status of UART at any time during the functional operation. Available status information includes the type and condition of the transfer operations being performed by the UART and any error conditions (parity, overrun, framing or break interrupt).

The serial ports can operate in either FIFO or non-FIFO mode. In FIFO mode, a 64-byte transmit FIFO holds data from the processor to be transmitted on the serial link and a 64-byte receive FIFO buffers data from the serial link until read by the processor.

The UART includes a programmable baud rate generator which is capable of dividing the clock input by divisors of 1 to $2^{16} - 1$ and produces a 16X clock to drive the internal transmitter logic. It also drives the receive logic. UART has a processor interrupt system. The UART can be operated in polled or in interrupt driven mode as selected by software.

The UART has the following features

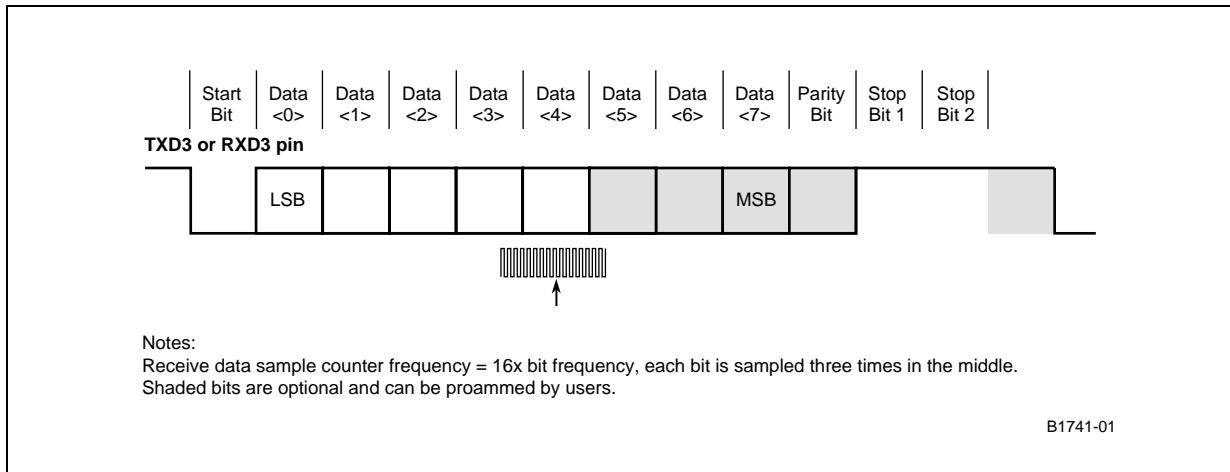
- Functionally compatible with National Semiconductor's PC16550D for basic receive and transmit.
- Adds or deletes standard asynchronous communications bits (start, stop, and parity) to or from the serial data
- Independently controlled transmit, receive, line status
- Programmable baud rate generator allows division of clock by 1 to $(2^{16} - 1)$ and generates an internal 16X clock
- 5, 6, 7 or 8-bit characters
- Even, odd, or no parity detection
- 1, 1-1/2, or 2 stop bit generation
- Baud rate generation
- False start bit detection
- 64-byte Transmit FIFO
- 64-byte Receive FIFO
- Complete status reporting capability
- Internal diagnostic capabilities include:
 - Break
 - Parity
 - Overrun
 - Framing error simulation
- Fully prioritized interrupt system controls



3.12.3 UART Operation

The format of a UART data frame is shown in Figure 31.

Figure 31. Example UART Data Frame



Each data frame is between 7 bits and 12 bits long depending on the size of data programmed, if parity is enabled and if two stop bits is selected. The frame begins with a start bit that is represented by a high to low transition. Next, either 5 to 8 bits of data are transmitted, beginning with the least significant bit. An optional parity bit follows, which is set if even parity is enabled and an odd number of ones exist within the data byte, or if odd parity is enabled and the data byte contains an even number of ones. The data frame ends with one, one and a half or two stop bits as programmed by the user, which is represented by one or two successive bit periods of a logic one.

3.12.3.1 UART FIFO OPERATION

The UART has one transmit FIFO and one receive FIFO. The transmit FIFO is 64-bytes deep and 8-bits wide. The receive FIFO is 64-bytes deep and 11-bits wide.

3.12.3.1.1 UART FIFO Interrupt Mode Operation - Receiver Interrupt

When the Receive FIFO and receiver interrupts are enabled (UART_FCR[0]=1 and UART_IER[0]=1), receiver interrupts occur as follows:

- The receive data available interrupt is invoked when the FIFO has reached its programmed trigger level. The interrupt is cleared when the FIFO drops below the programmed trigger level.
- The UART_IIR receive data available indication also occurs when the FIFO trigger level is reached, and like the interrupt, the bits are cleared when the FIFO drops below the trigger level.
- The receiver line status interrupt (UART_IIR = C6H), as before, has the highest priority. The receiver data available interrupt (UART_IIR=C4H) is lower. The line status interrupt occurs only when the character at the top of the FIFO has errors.
- The data ready bit (DR in UART_LSR register) is set to 1 as soon as a character is transferred from the shift register to the Receive FIFO. This bit is reset to 0 when the FIFO is empty.



Character Time-out Interrupt

When the receiver FIFO and receiver time out interrupt are enabled, a character time-out interrupt occurs when all of the following conditions exist:

- At least one character is in the FIFO.
- The last received character was longer than four continuous character times ago (if two stop bits are programmed the second one is included in this time delay).
- The most recent processor read of the FIFO was longer than four continuous character times ago.

The maximum time between a received character and a time-out interrupt is 160 ms at 300 baud with a 12-bit receive character (i.e., 1 start, 8 data, 1 parity, and 2 stop bits).

When a time out interrupt occurs, it is cleared and the timer is reset when the processor reads one character from the receiver FIFO. If a time-out interrupt has not occurred, the time-out timer is reset after a new character is received or after the processor reads the receiver FIFO.

Transmit Interrupt

When the transmitter FIFO and transmitter interrupt are enabled (UART_FCR[0]=1, UART_IER[1]=1), transmit interrupts occur as follows:

- The Transmit Data Request interrupt occurs when the transmit FIFO is half empty or more than half empty. The interrupt is cleared as soon as the Transmit Holding Register is written (1 to 64 characters may be written to the transmit FIFO while servicing the interrupt) or the IIR is read.

3.12.3.1.2 FIFO Polled Mode Operation

With the FIFOs enabled (TRFIFOE bit of UART_FCR set to 1), setting UART_IER[4:0] to all zeros puts the serial port in the FIFO polled mode of operation. Since the receiver and the transmitter are controlled separately, either one or both can be in the polled mode of operation. In this mode, software checks receiver and transmitter status via the UART_LSR. As stated in the register description:

- UART_LSR[0] is set as long as there is one byte in the receiver FIFO.
- UART_LSR[1] through UART_LSR[4] specify which error(s) has occurred for the character at the top of the FIFO. Character error status is handled the same way as interrupt mode. The UART_IIR is not affected since UART_IER[2] = 0.
- UART_LSR[5] indicates when the transmitter FIFO needs data.
- UART_LSR[6] indicates that both the transmitter FIFO and shift register are empty.
- UART_LSR[7] indicates whether there are any errors in the receiver FIFO.



3.12.4 Baud Rate Generator

The baud rate generator is a programmable block and generates a clock used in the transmit block. The output frequency of the baud rate generator is 16X the baud rate; baud rate is calculated as:

$$\text{BaudRate} = \text{APB Clock} / (16 \times \text{Divisor})$$

The Divisor ranges from 1 to $2^{16} - 1$. For example, for a APB clock of 1MHZ and baud rate of 300bps the divisor is 209.

3.12.5 General Purpose I/O (GPIO)

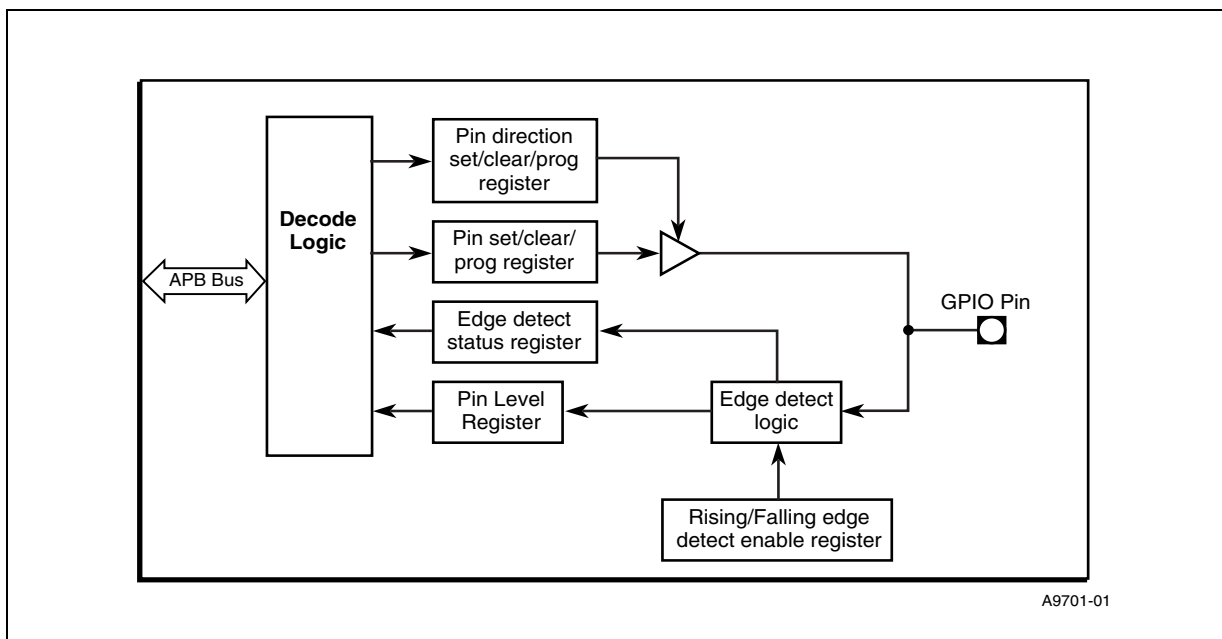
The IXP2800 Network Processor has eight General Purpose Input/Output (GPIO) port pins for use in generating and capturing application-specific input and output signals. Each pin is programmable as an input or output or as an interrupt signal sourcing from an external device. The GPIO can be used with appropriate software in I2C application.

Each GPIO pin can be configured as a input or an output by programming the corresponding GPIO pin direction register. When programmed as an input, the current state of the GPIO can be read through the corresponding GPIO pin level register. The register can be read at any time and can be used to confirm the state of the pin when it is configured as an output. In addition, each GPIO pin can be programmed to detect a rising or a falling edge by setting the corresponding GPIO rising/falling edge detect registers.

When configured as an output, the pin can be controlled by writing to the GPIO set register to write a 1 and by writing to the GPIO clear register to write a 0. These registers can be written regardless of whether the pin is configured as an input or a output.

Each of the GPIO pins is designed the same and instantiated to the number of GPIO port pins. [Figure 32](#) shows a GPIO functional diagram. The GPIO pin as seen can be programmed based on the configuration registers.

Figure 32. GPIO Functional Diagram



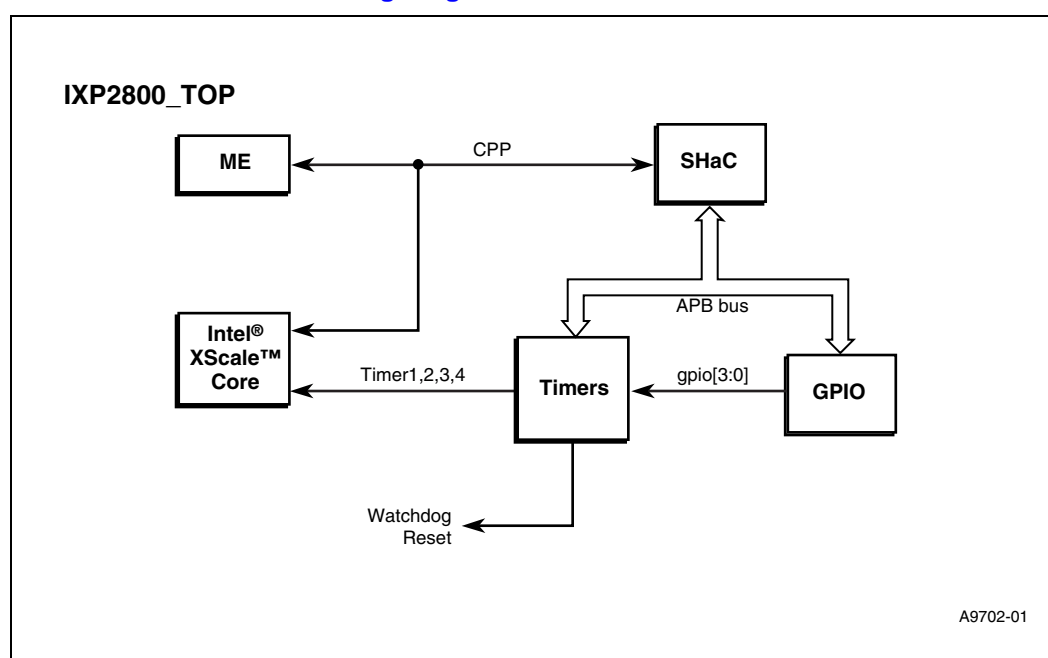
3.12.6 Timers

The IXP2800 Network Processor supports four timers. These timers are clocked by the Advanced Peripheral/Bus Clock (APB-CLK), which runs at 50 MHz. to produce the PLPL_APB_CLK, PLPL_APB_CLK/16 or PLPL_APB_CLK/256 signals. The counters are loaded with an initial value, count down to zero, and raise an interrupt (if interrupts are not masked).

In addition, timer 4 can be used as a watchdog timer when the watchdog enable bits are configured to one. When used as a watchdog timer, and when a count of zero is encountered, it will initiate the reset sequence.

Figure 33 shows the timer control unit interfacing with other functional blocks.

Figure 33. Timer Control Unit Interfacing Diagram



3.12.6.1 Timer Operation

Each timer consists of a 32-bit counter.

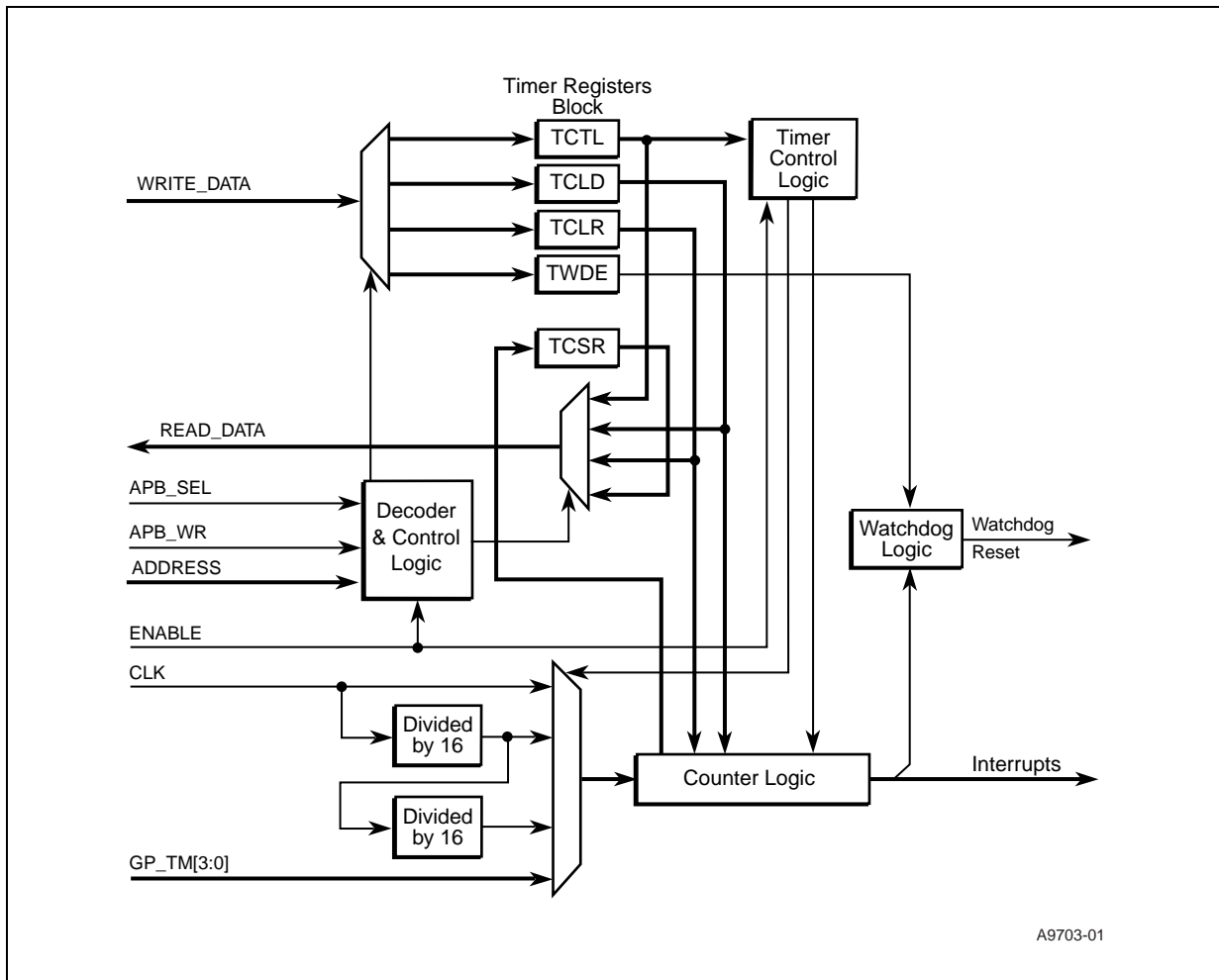
By default, the timer counter load register (TCLD) is set to 0xFFFFFFFF. The timer will count down from 0xFFFFFFFF to zero, then wrap back to 0xFFFFFFFF and continue to decrement if the TCLD is not programmed to any value. If a different value is programmed in the TCLD, then the counter will load this value every time it counts down to zero.

An interrupt is issued to the Intel XScale® core whenever the counter reaches zero. The interrupt signals can be enabled or disabled by the IRQEnable/FIQEnable registers. The interrupt remains asserted until it is cleared by writing a 1 to the corresponding timer clear register (TCLR).

The counter can be advanced by the clock, clock divided by 16, clock divided by 256, and the GPIO signals. The clock rate is controlled by the TCTL value programmed into the TCTL registers. There are four gpio signals, GPIO[3:0] which correspond to Timer 1, 2, 3, and 4, respectively. These signal are synchronized within the timer-clock domain before driving the counter.

Figure 34 shows the Timer Internal logic.

Figure 34. Timer Internal Logic Diagram





3.12.7 SlowPort Unit

The IXP2800 Network Processor SlowPort Unit supports basic PROM access and 8, 16, and 32-bit microprocessor device access. It allows a master, (Intel XScale® core or Microengine), to do a read/ write data transfer to these slave devices.

The address bus and data bus are multiplexed to reduce the pin count. In addition, the address bus is also compressed from A[25:0] down to A[7:0] and shifted out with three clock cycles. Therefore, an external set of buffers is needed for address storage and latch.

The access can be asynchronous. Insertion of delay cycles is possible for both setup and hold data. A programmable timing control mechanism is provided for this purpose.

There are two types of interfaces supported in the SlowPort Unit:

- Flash memory interface
- μ P interface.

The Flash memory interface is used for the PROM device. The μ P interface can be used for SONET/SDH Framer μ P access.

There are two ports in the SlowPort unit. The first is dedicated to the flash memory device while the second to the μ P device.

3.12.7.1 PROM Device Support

For all the Flash Memory access, only 8-bit devices are supported. APB write transactions are assumed to be 8-bits wide, and correspond to one write cycle at the flash interface. The extended APB read transactions are assumed to be 32-bits wide, and correspond to four read cycles at the flash memory interface for all the flash memory data read. However, for the flash register read inside the flash memory, like the flash status register, the returned data are one byte and placed in the lower order byte location. In this case, only one external transaction cycle is involved.

To accomplish this, a register (SP_FRM) is installed to allow to configure between 8-bit read mode and 32-bit read mode. By default, it goes to 32-bit read mode. For the 8-bit read mode, one read cycle is involved. No packing process is needed. The data will be directly placed onto the lower order byte, [7:0] and passed to APB bus. For the 32-bit read mode, it needs four read cycles. All 4 bytes are packed into a 32-bit data and passed to the APB bus. 16-bit mode is not supported for read.

Write always accesses the flash with one 8-bit cycle. Therefore, no unpacking process is needed.

The PROM device supported are listed in [Figure 56](#):

Table 56. 8-bit Flash Memory Device Density

Vendor	Part Number	Size
Intel	28F128J3A	16MB
Intel	28F640J3A	8MB
Intel	28F320J3A	4MB



3.12.7.2 μ P interface support for the Framer

The SlowPort Unit also supports a microprocessor interface with Framer components. Some supported devices are listed in [Table 57](#):

Table 57. SONET/SDH Devices

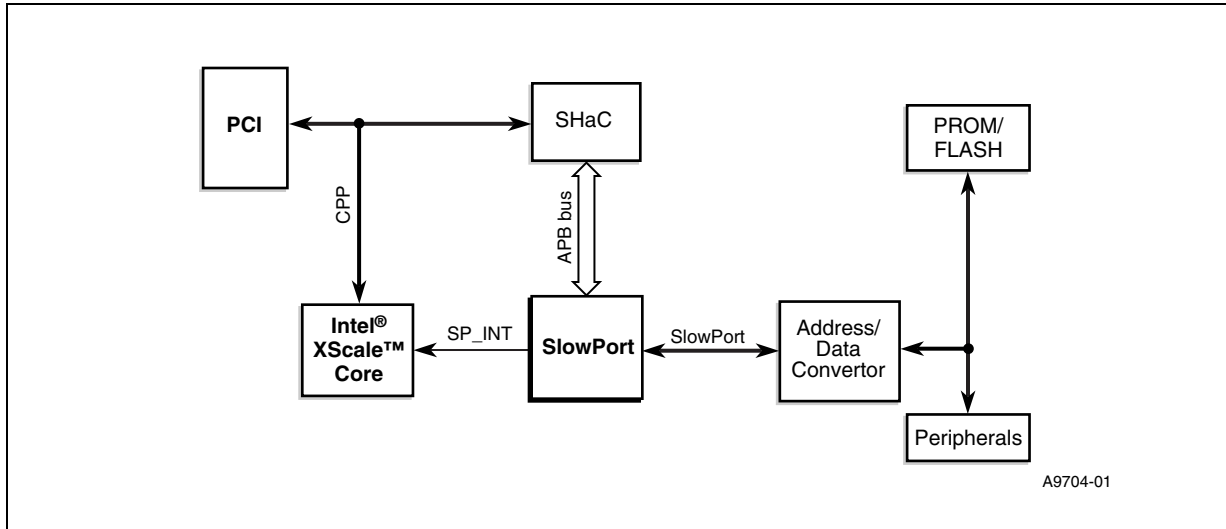
Vendor	Part Number	μ P Interface	SP_PCR register Setting	DW Setting in SP_ADC register
PMC-Sierra	PM3386	16 bits	0x3	0x1
PMC-Sierra	PM5345	8 bits	0x2	0x0
PMC-Sierra	PM5346	8 bits	0x2	0x0
PMC-Sierra	PM5347	8 bits	0x2	0x0
PMC-Sierra	PM5348	8 bits	0x2	0x0
PMC-Sierra	PM5349	8 bits	0x2	0x0
PMC-Sierra	PM5350	8 bits	0x2	0x0
PMC-Sierra	PM5351	8 bits	0x2	0x0
PMC-Sierra	PM5352	8 bits	0x2	0x0
PMC-Sierra	PM5355	8 bits	0x2	0x0
PMC-Sierra	PM5356	8 bits	0x2	0x0
PMC-Sierra	PM5357	8 bits	0x2	0x0
PMC-Sierra	PM5358	16 bits	0x2	0x1
PMC-Sierra	PM5381	16 bits	0x2	0x1
PMC-Sierra	PM5382	8 bits	0x2	0x0
PMC-Sierra	PM5386	16 bits	0x2	0x1
AMCC	S4801 (AMAZON)	8 bits	0x0	0x0
AMCC	S4803 (YUKON)	8 bits	0x0	0x0
AMCC	S4804 (RHINE)	8/16 bits	0x0/0x3	0x0/0x1
Intel	IXF6012 (Volga)	16 bits	0x3/0x4 ¹	0x1
Intel	IXF6048 (Amazon-A)	16 bits	0x3/0x4 ^a	0x1
Intel	Centaur		0x3/0x4 ^a	
Intel	IXF6501		0x3/0x4 ^a	
Intel	Ben Nevis	32 bits	0x3/0x4 ^a	0x2
Lucent	TDAT042G5	16 bits	0x1/	0x1
Lucent	TDAT04622	16 bits	0x1	0x1
Lucent	TDAT021G2	16 bits	0x1	0x1

1. Usually there are two different protocols, Intel or Motorola, of μ P interface in the Intel framer; the setting in the PCR should match with protocols activated in the framer.

3.12.7.3 SlowPort Unit Interfaces

Figure 35 shows the SlowPort Unit interface diagram.

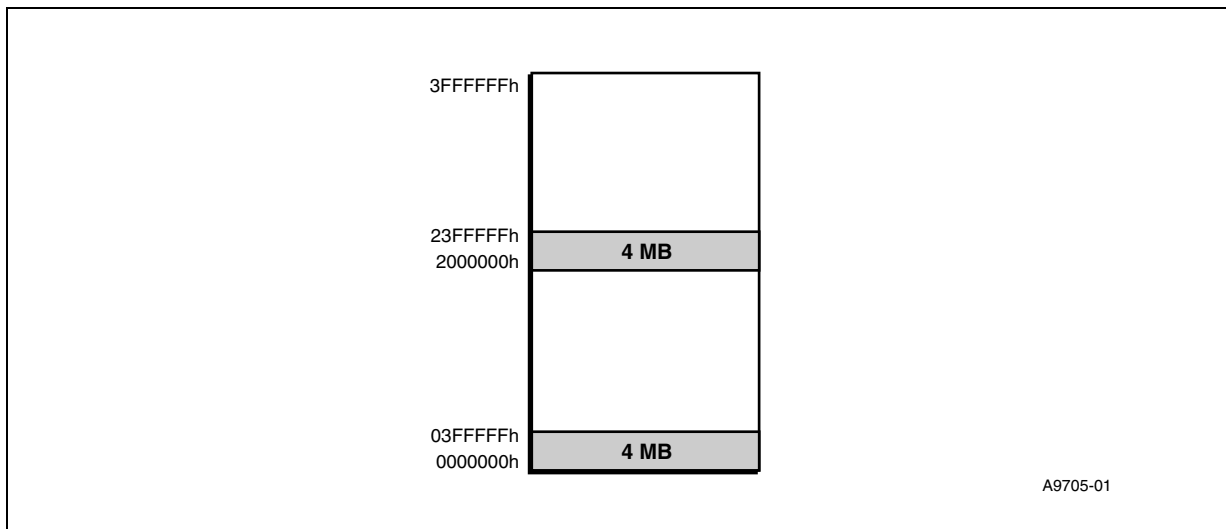
Figure 35. SlowPort Unit Interface Diagram



3.12.7.4 Address Space

The total address space is defined as 64 MB, which is further divided into two segments of 32 MB each. Two devices can connect to this bus. If these peripheral devices have a density of 256 Mbit (32 MB) each, all the address space is going to be filled like a contiguous address space. However, if a small capacity device is used (like a 4 MB, 8 MB, 16 MB), there will be a memory hole left in between these two devices. Figure 36 is a 4 MB memory example. Trying to read the space in between, you will get the repeating value for each 4 MB location

Figure 36. An Example of Address Space Hole Diagram





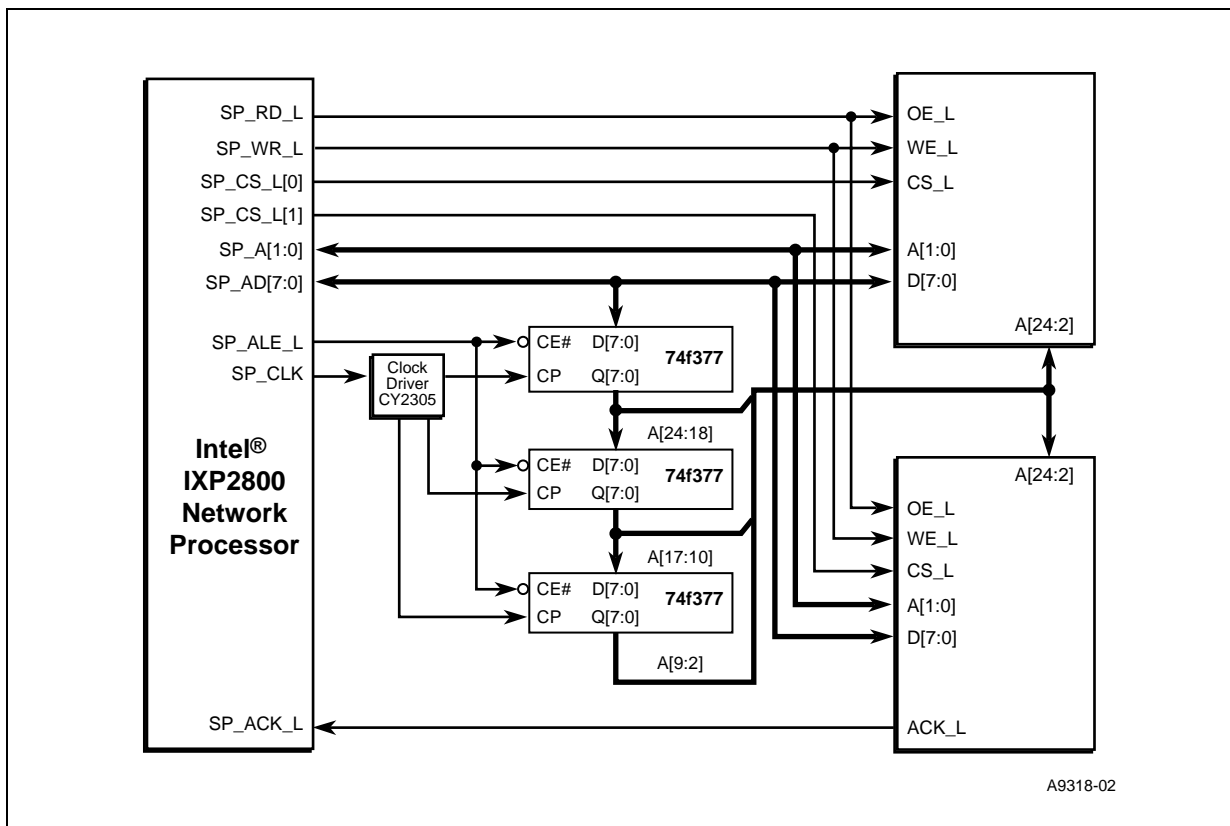
3.12.7.5 SlowPort Interfacing Topology

Figure 37 demonstrates one of the topologies used to connect to an 8-bit device. From the diagram, we can observe that address is shifted out 8 bits at a time and buffered into three 74F377 or equivalent tri-state buffer devices in three consecutive clock cycles. These buffers also output separately to form a 25-bit wide address bus to address the 8-bit devices. The data are expected to be driven out after the address has been placed into the buffers.

There are two devices shown in Figure 37. The top one is the fix-timed device, while the lower one, self-timing device. For the self-timing device, the access latency depends on the SP_ACK_L responded back from this device.

Three extra signals, SP_CP, SP_OE_L and SP_DIR, are added to pack and unpack the data when a 16-bit or 32-bit device is hooked up to SlowPort. They are used for special application only as described below.

Figure 37. SlowPort Example Application Topology



3.12.7.6 SlowPort 8-bit Device Bus Protocols

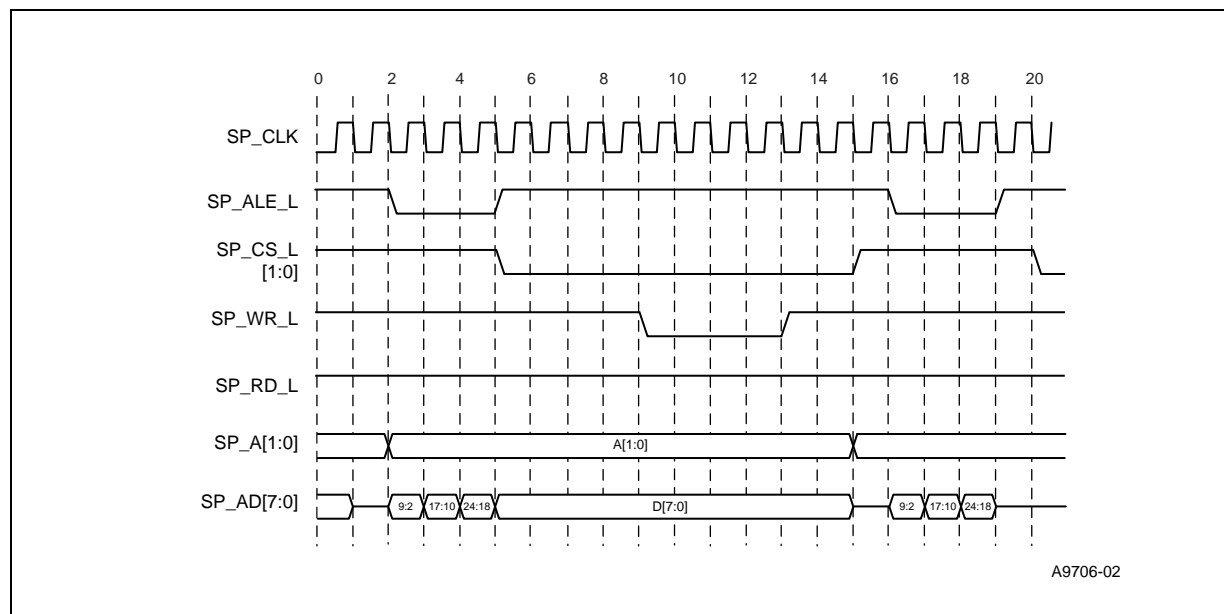
The write/read transfer protocols are discussed in the following sections. The burst transfers are going to be broken down into single mode transfer. For each single write/read transaction, it can be either fixed-timed transaction or self-timing transaction. The fixed-timed transaction has the response fixed in a certain period, which can be controlled by the timing control registers.

For the self-timing transaction, the response timing is dictated by the peripheral device. Hence, wait states can be inserted during the transaction. All the back-to-back transactions are intervened with one clock cycle. The SlowPort clock, SP_CLK, shown in the following waveform diagrams, is generated by dividing the PLPL_APB_CLK. The divisor used is specified in the clock control register, SP_CCR.

3.12.7.6.1 Mode 0 Single Write Transfer for Fixed-Timed Device

Figure 38, shows the single write transfer for a fixed-timed device with the CSR programmed to a value of setup=4, pulse width=0, and hold=1, followed by another read transfer.

Figure 38. Mode 0 Single Write Transfer for a Fixed-Timed Device



The transaction is initiated with SP_ALE_L asserted. It latches the address from the SP_AD[7:0] bus into the external buffer, using three clock cycles. After that, it should deassert the SP_ALE_L to disable latching the address into the buffers.

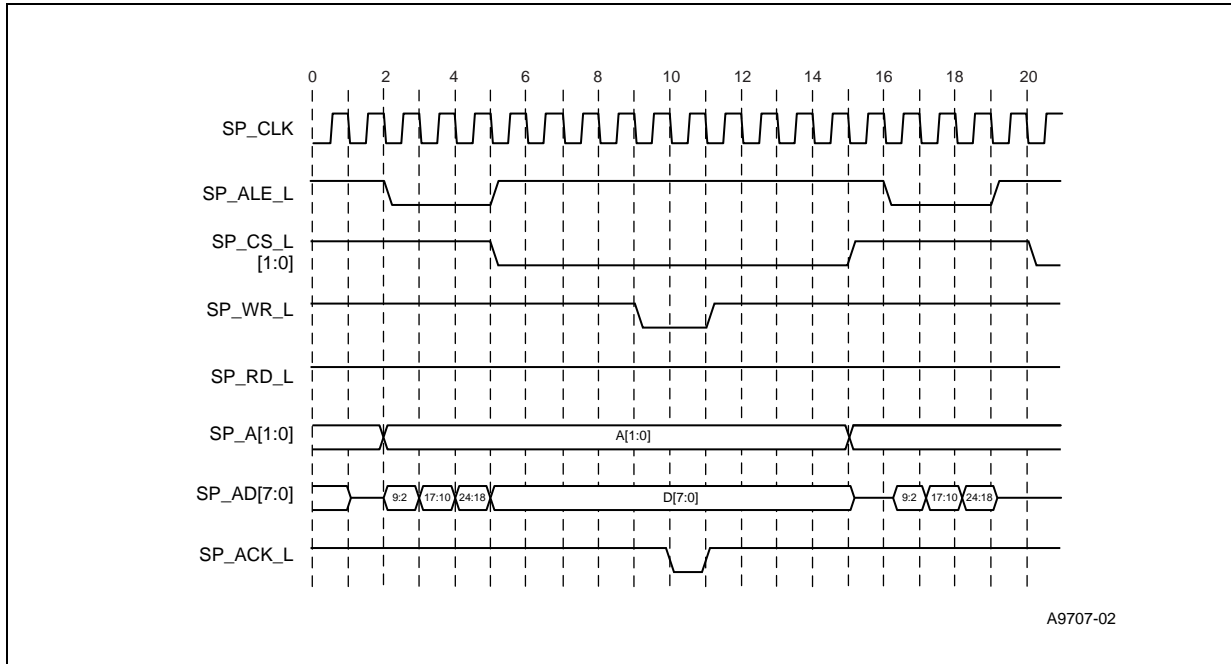
The SP_A[1:0] signals span the whole transaction cycle.

For the write, it drives the data onto the SP_AD[7:0]. Meanwhile, it asserts the SP_CS_L[1:0] signals. Depending on the timing control setup parameter, for this case, the SP_WR_L is not asserted until four clock cycles have elapsed. The SP_CS_L[1:0] signals are deasserted two clocks after the SP_WR_L is deasserted.

3.12.7.6.2 Mode 0 Single Write Transfer for a Self-timing Device

Figure 39 depicts the single write transfer for a self-timing device with the CSR programmed to setup=4, pulse width=0, and hold=4. Similarly, a read transaction is attached behind.

Figure 39. Mode 0 Single Write Transfer for a Self-timing Device



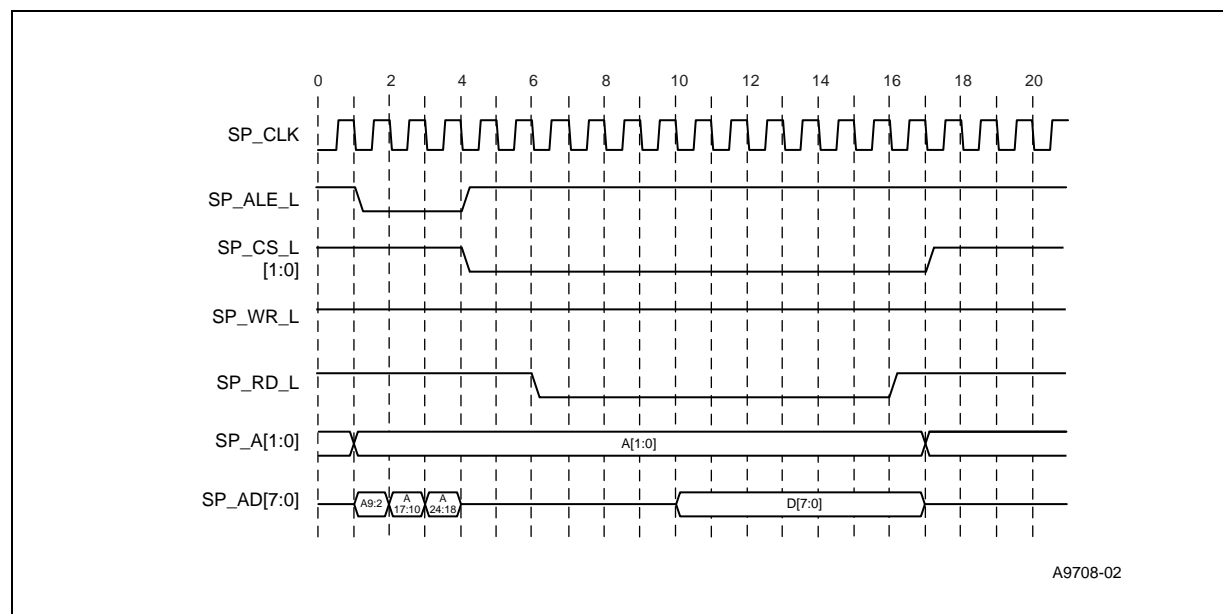
Similar to the single write for fixed-timed device, the ALE_L, CS_L[1:0], AD[7:0], and A[1:0] follow the same pattern, and the timing is controlled by the timing control register. Except for the WR_L which is terminated depending on the SP_ACK_L returned from the self-timing device.

The time-out counter will be set to 255. If no SP_ACK_L responds back when the time-out counter reaches zero, the transaction is terminated with a time-out. An interrupt signal is issued to the bus master simultaneously with the time-out register update.

3.12.7.6.3 Mode 0 Single Read Transfer for Fixed-timed Device

Figure 40 demonstrates the single read transfer issued to a fixed-timed PROM device followed by another write transaction. The CSR is assumed to be configured to the value setup=2, pulse width=10, and hold=1.

Figure 40. Mode 0 Single Read Transfer for a Fixed-timed Device



The address is loaded onto the external buffer in three clock cycles with the ALE_L asserted. Then, a clock cycle is inserted to tri-state all the AD[7:0] signals. The CS_L[1:0] signals come asserted on the fourth clock cycle. Now, the values stored in the timing control registers take effect. The RD_L becomes asserted after two clock cycles. It keeps asserted for ten clock cycles. The CS_L[1:0] should be de-asserted one clock cycle after RD_L is de-asserted. The data will be valid at clock cycle 16 as shown in the diagram. Since the hold delay has 2 cycles, transaction is terminated at clock cycle 16.

Figure 41 demonstrates the single read transfer issued to a self-timing PROM device followed by another write transaction. The CSR assumed to be programmed to the value of setup=4, pulse width=0, and hold=2.

The diagram shows the timing of various signals relative to a clock (SP_CLK) over 20 cycles. The signals are:

- SP_CLK**: A periodic clock signal.
- SP_ALE_L**: Active-low chip select signal, high for most of the time, with a pulse low from cycle 2 to 4 and another from cycle 16 to 18.
- SP_CS_L [1:0]**: Active-low chip select signals, high for most of the time, with a pulse low from cycle 5 to 14.
- SP_WR_L**: Active-low write strobe signal, high for most of the time, with a pulse low from cycle 8 to 12.
- SP_RD_L**: Active-low read strobe signal, high for most of the time, with a pulse low from cycle 10 to 14.
- SP_A[1:0]**: Address signals, high for most of the time, with a pulse low from cycle 8 to 12.
- SP_AD[7:0]**: Address data bus signals, high for most of the time, with a pulse low from cycle 10 to 14.
- SP_ACK_L**: Active-low acknowledgment signal, high for most of the time, with a pulse low from cycle 10 to 14.

3.12.7.7 SONET/SDH Microprocessor Access Support

However, because these microprocessor interfaces are not standardized, we treat them separately and a configuration register is installed to activate the bus to work with different interface protocol at a time. Extra pins are also added to accomplish this task.

A microprocessor interface type register is used to provide these kinds of solutions. The user is allowed to configure the interface to the following four different modes. The pin functionality and the interface protocol will be changed accordingly. By default, it activates the mode 0 with 8-bit generic PROM device support as mentioned above.



3.12.7.7.1 Mode 1: 16-bit Microprocessor Interface Support with 16-bit Address Lines

The address size control register is programmed to 16-bit address space for this case. This mode is designated for the devices with the similar protocol with the Lucent TDAT042G5 SONET/SDH device.

16-bit Microprocessor Interfacing Topology with 16-bit address lines

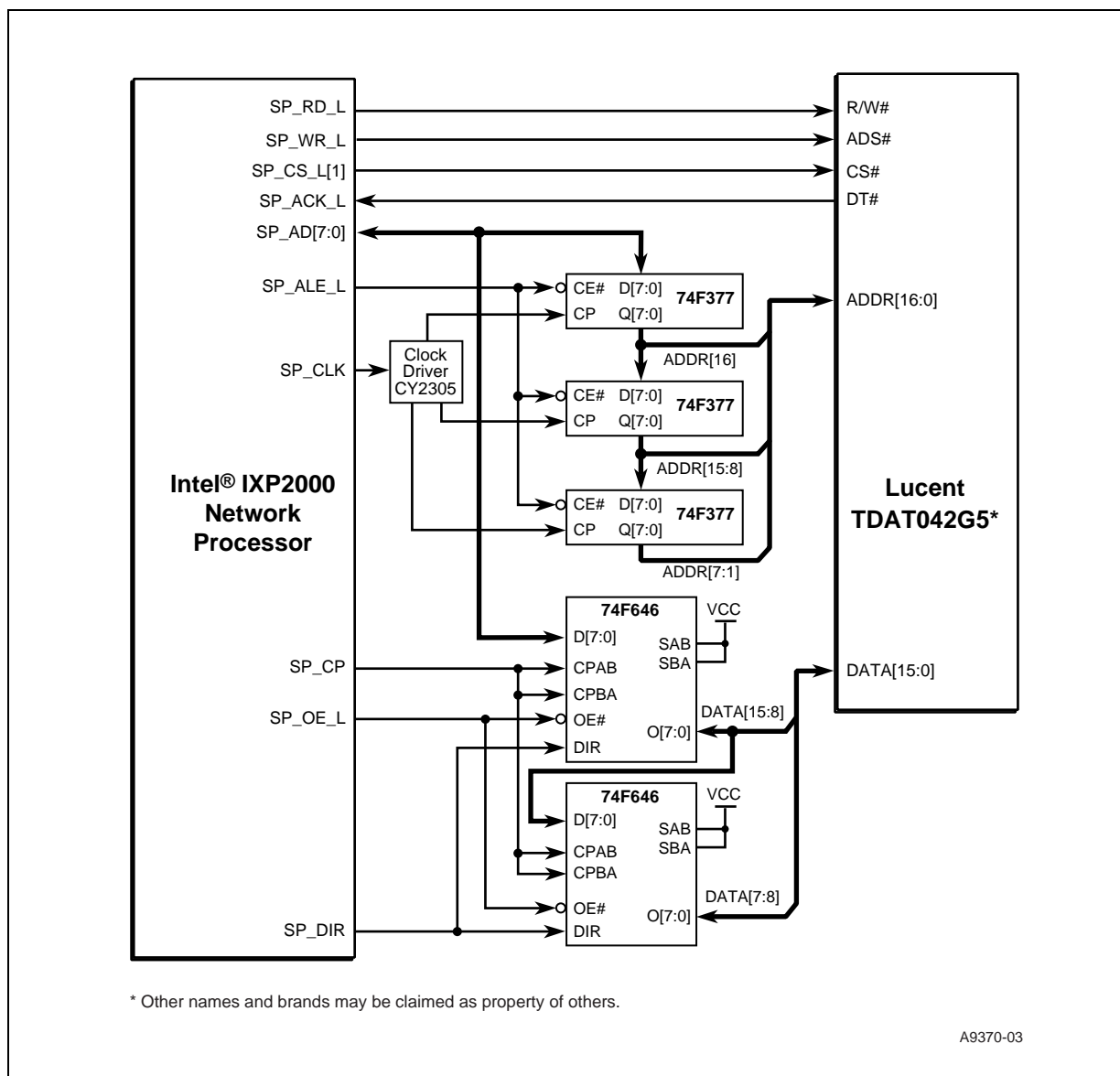
Figure 42 shows a solution for the 16-bit microprocessor interface. This solution bridges the Lucent TDAT042G5 SONET/SDH 16-bit interface. From Figure 42, we observe that the control pins SP_RD_L and SP_WR_L are converted to R/W and ADS. The CS and DT are still compactible with SP_CS_L[1] and SP_ACK_L protocol.

Extra pins are added to accomplish the task of multiplexing and demultiplexing the data bus. The total pin count is 18.

During the write cycle, 8-bit data are stacked into 16-bit data. They are first shifted into two tri-state buffers, 74F646 or equivalent by SP_CP, using two consecutive clock cycle. Then the SP_CS_L is used for output the 16-bit data, which is shared with the CS.

During the read cycle, the 16-bit data are unpacked into 8-bit data by SP_CP. Two 74F646 or equivalent tri-state buffers are used. First, the 16-bit data are stored into these buffers. Then they are shifted out by SP_DIR, using two consecutive clock cycle.

Figure 42. An Interface Topology with Lucent TDAT042G5 SSONET/SDH



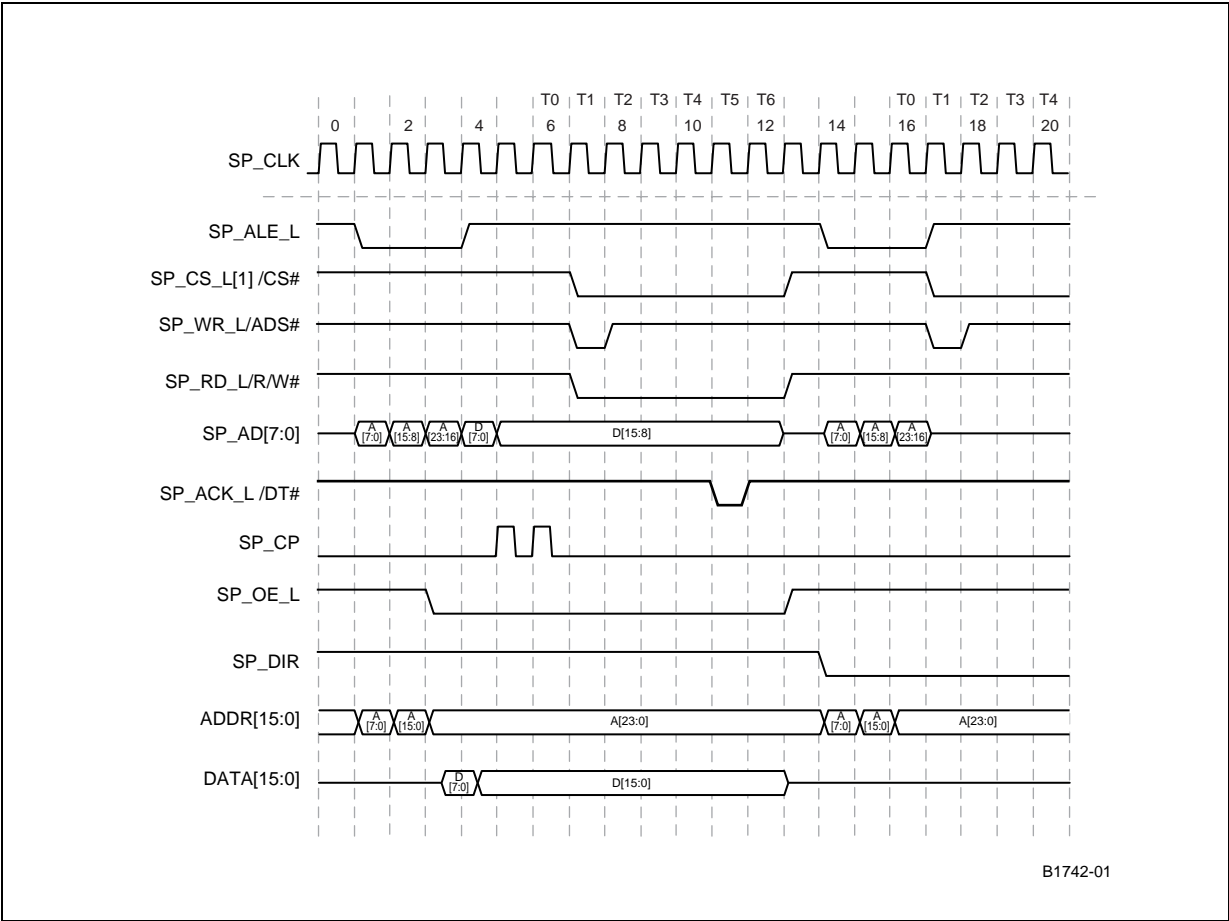


16-bit Microprocessor Write Interface Protocol

Figure 43 uses the Lucent TDAT042G5 device. In this case, the user should program the P_PCR register to mode 1 and also program the write timing control register to setup=7, pulse width=5, and hold=1, which represent 7 clock cycles for \overline{CS} , 5clock cycle for DT delay, and 1 clock cycle for \overline{ADS} . They are intervened with two idle cycles.

From Figure 43, we observe that there are a total of twelve clock cycles used for write access, (i.e., 240 ns), not including an intervened turnaround cycle after the write transaction. The throughput is 8.3 MB per second

Figure 43. Mode 1 Single Write Transfer for Lucent TDAT042G5 Device (B0)

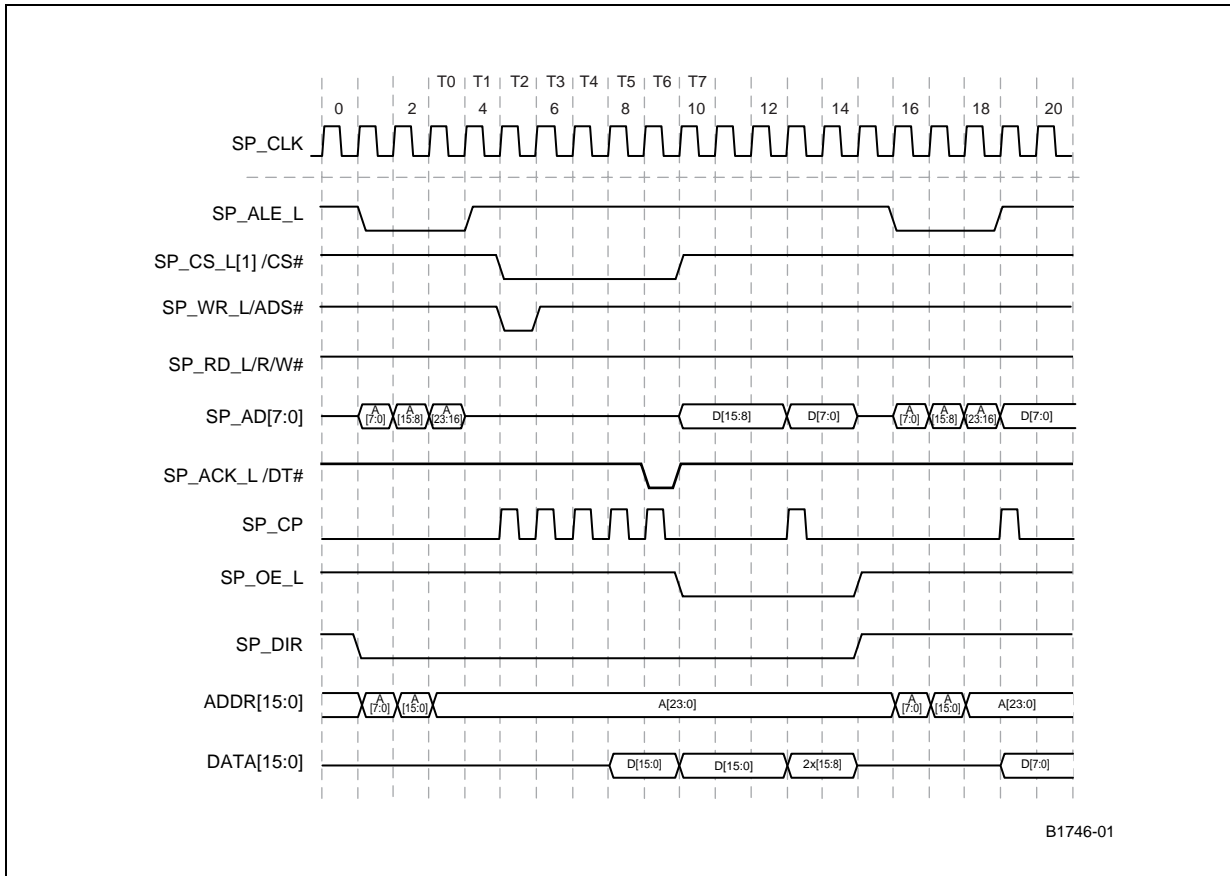


16-bit Microprocessor Read Interface Protocol

Figure 44, likewise depicts a single read transaction launched from the IXP2800 Network Processor to the Lucent TDAT042G5 device followed by a single read transaction. However, in this case the read timing control register has to be programmed to setup=0, pulse width=7, and hold =0.

In Figure 44, we can count twelve clock cycles used for the read transaction in total, (i.e., 240 ns) for a clock cycle of 10 ns, excluding a turnaround cycle after that. It has the throughput of 7.7 MB per second.

Figure 44. Mode 1 Single Read Transfer for Lucent TDAT042G5 Device (B0)



3.12.7.7.2 Mode 2: Interface With 8 Data Bits and 11 Address Bits

This application is designed for the PMC-Sierra PM5351 S/UNI-TETRA Device. For the PMC-Sierra PM5351, the address space is programmed to 11-bits; otherwise, other address space should be specified.

8-bit PMC-Sierra PM5351 S/UNI-TETRA Interfacing Topology

Figure 45 displays one of the topologies used to connect to the SlowPort with the PMC-Sierra PM5351 S/UNI-TETRA device.

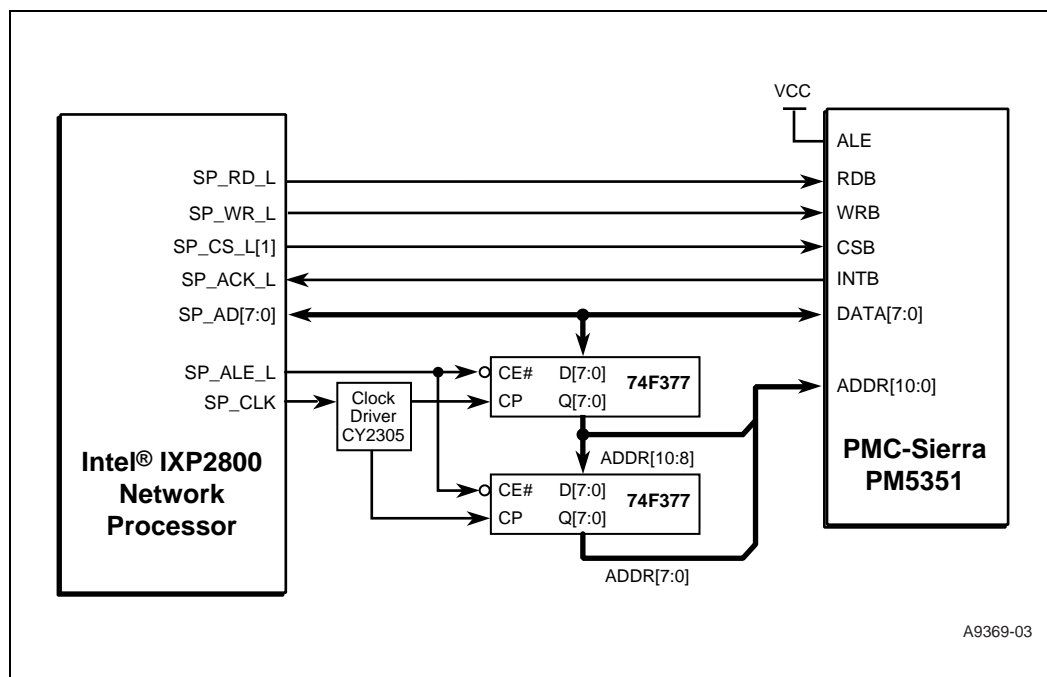
From Figure 45, because the protocols are very close to the generic SlowPort protocol, the pin counts and the functionality is quite compatible. We don't need to use any more pins in this case. The only difference is in the INTB signal, which will be connected to the SP_ACK_L. Therefore the SP_ACK_L needs to be converted to an interrupt signal.

Also because the address contains only 11bits, two 74F377 or equivalent buffers are needed.

The AS field in the SP_ADC register should be programmed to a 16-bit addressing space with the upper 5 address bits unconnected.

The timing controls are similar to the generic case.

Figure 45. An Interface Topology with PMC-Sierra PM5351 S/UNI-TETRA

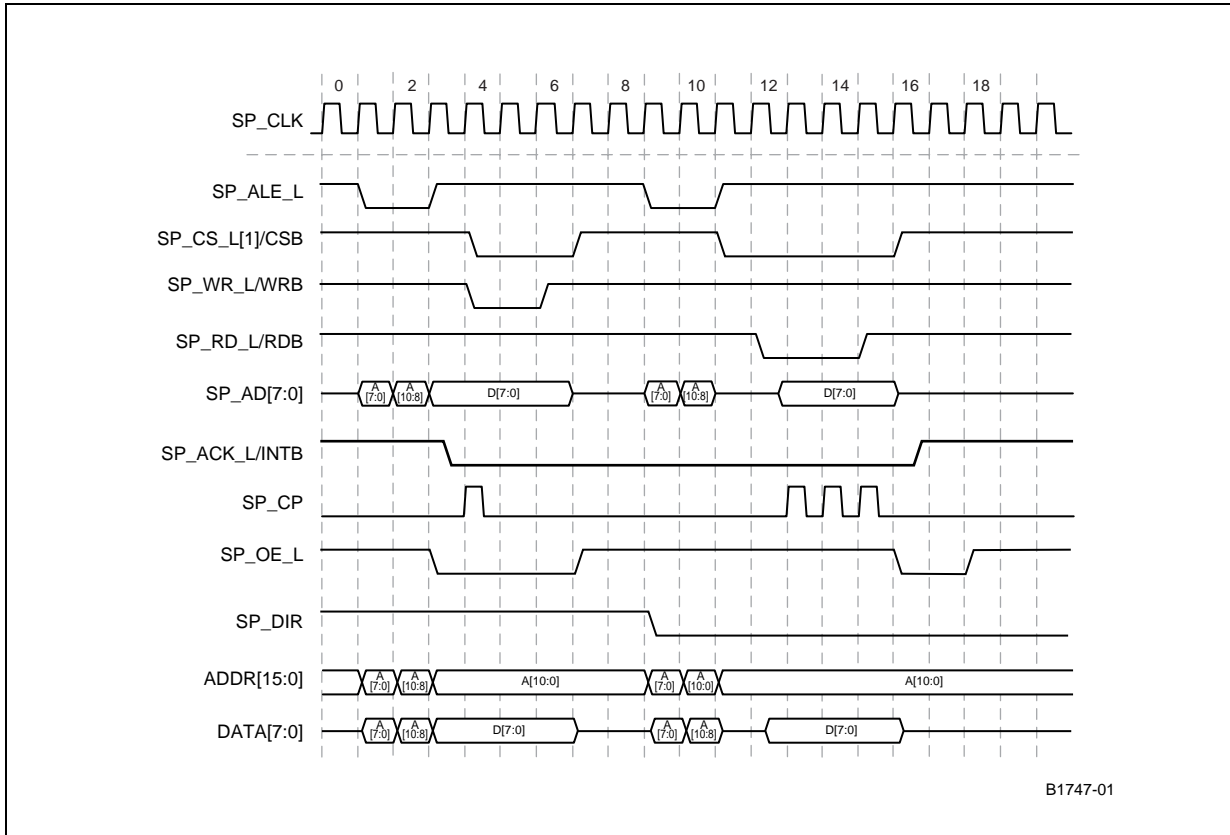


PMC-Sierra PM5351 S/UNI-TETRA Write Interface Protocol

Figure 46 depicts a single write transaction launched from the IXP2800 to the PMC-Sierra PM5351 device followed by single read transaction.

The write transaction for the PMC-Sierra component has 6 clock cycle or 120ns access time for a 50MHz SlowPort clock. In this case, no intervening cycle is added after the transaction. The I/O throughput is 8.3MB per second. The SP_PCR should be programmed to mode 2 and the fields of SU, PW, and HD in the SP_WTC2 should be set to 1, 2, 1 respectively. Here SU, PW, and HD represent the SP_CS_L[1] pulse width, SP_WR_L pulse width, and SP_CP pulse width respectively.

Figure 46. Mode 2 Single Write Transfer for PMC-Sierra PM5351 Device (B0)

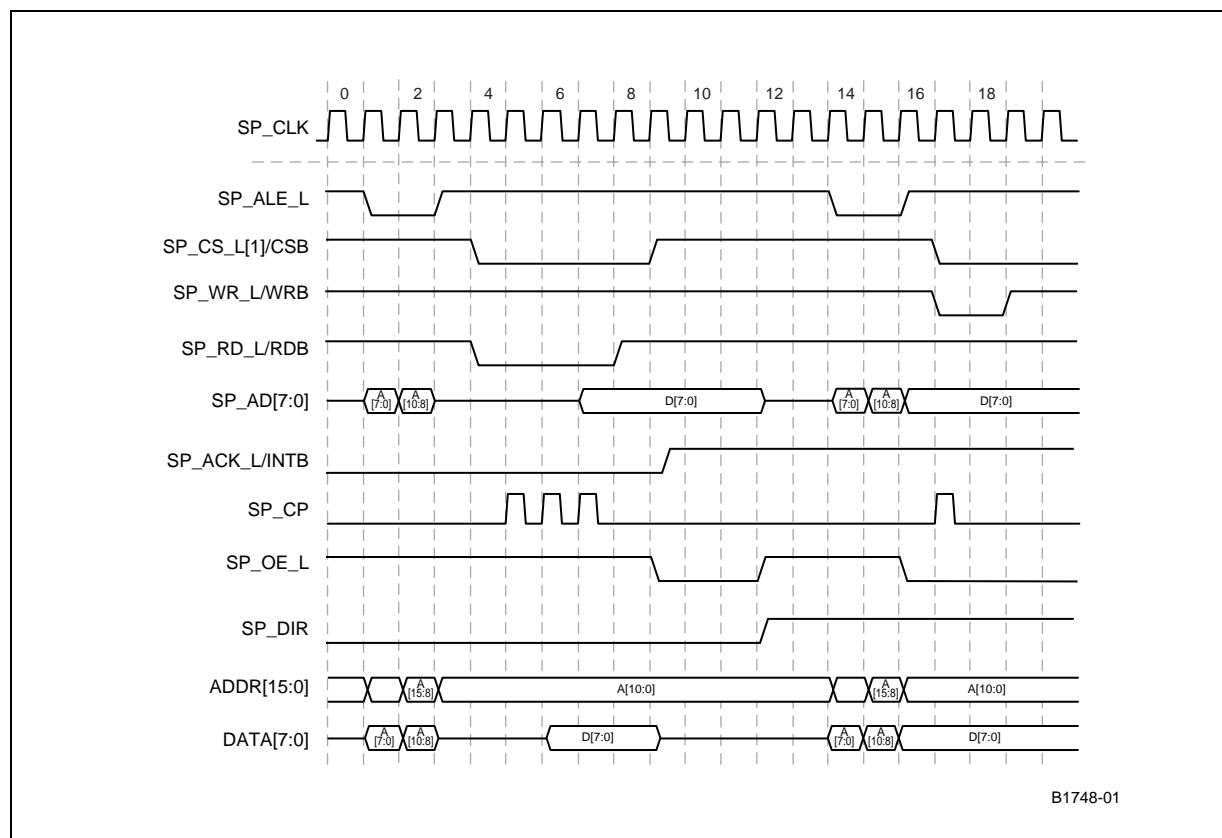


PMC-Sierra PM5351 S/UNI-TETRA Read Interface Protocol

Figure 47, depicts a single read transaction launched from the IXP2800 Network Processor to the PMC-Sierra PM5351 device followed by a single write transaction.

In this case, there are ten clock cycles of access time, or 200 ns in total, with three turnaround cycles attached at the back. The I/O throughput is 11.2 MB per second.

Figure 47. Mode 2 Single Read Transfer for PMC-Sierra PM5351 Device (B0)



3.12.7.7.3 Mode 3: Support for the Intel and AMCC 2488 Mbps SONET/SDH Microprocessor Interface

The user has to configure the address bus to 10 bits.

Mode 3 Interfacing Topology

Figure 48 demonstrates one of the topologies used to connect the SlowPort to the Intel and AMCC 2488Mbps SONET/SDH device. Similar to the Lucent TDAT042G5 interface, the address and the data need demultiplexing. Totally, it requires four buffers to accomplish this task.

The SP_RD_L, SP_WR_L, and SP_CS_L[1] entirely match the RDB, WRB, and CSB pins in the Intel and AMCC component. However, the INT has to be connected to the SP_ACK_L as the PMC-Sierra Interface does. The ALE pin shares the SP_CP signal. If the timing doesn't meet specification, ALE can be tied high as shown in Figure 49. It employs the same method as Lucent's TDAT042G5's topology to pack and unpack the data between the IXP2800 SlowPort interface and the Intel and AMCC microprocessor interface.

For a write, SP_CP loads the data onto the 74F646 or equivalent tri-state buffers, using two clock cycles. In order to reduce the pin count, the 16-bit data are latched with the same pin (SP_CS_L[1]), assuming that a turnaround cycle is inserted between the transaction cycles.

For a read, data are shifted out of two 74F646 or equivalent tri-state buffers by SP_CP, using two consecutive clock cycles.

Figure 48. An Interface Topology with Intel / AMCC SONET/SDH Device

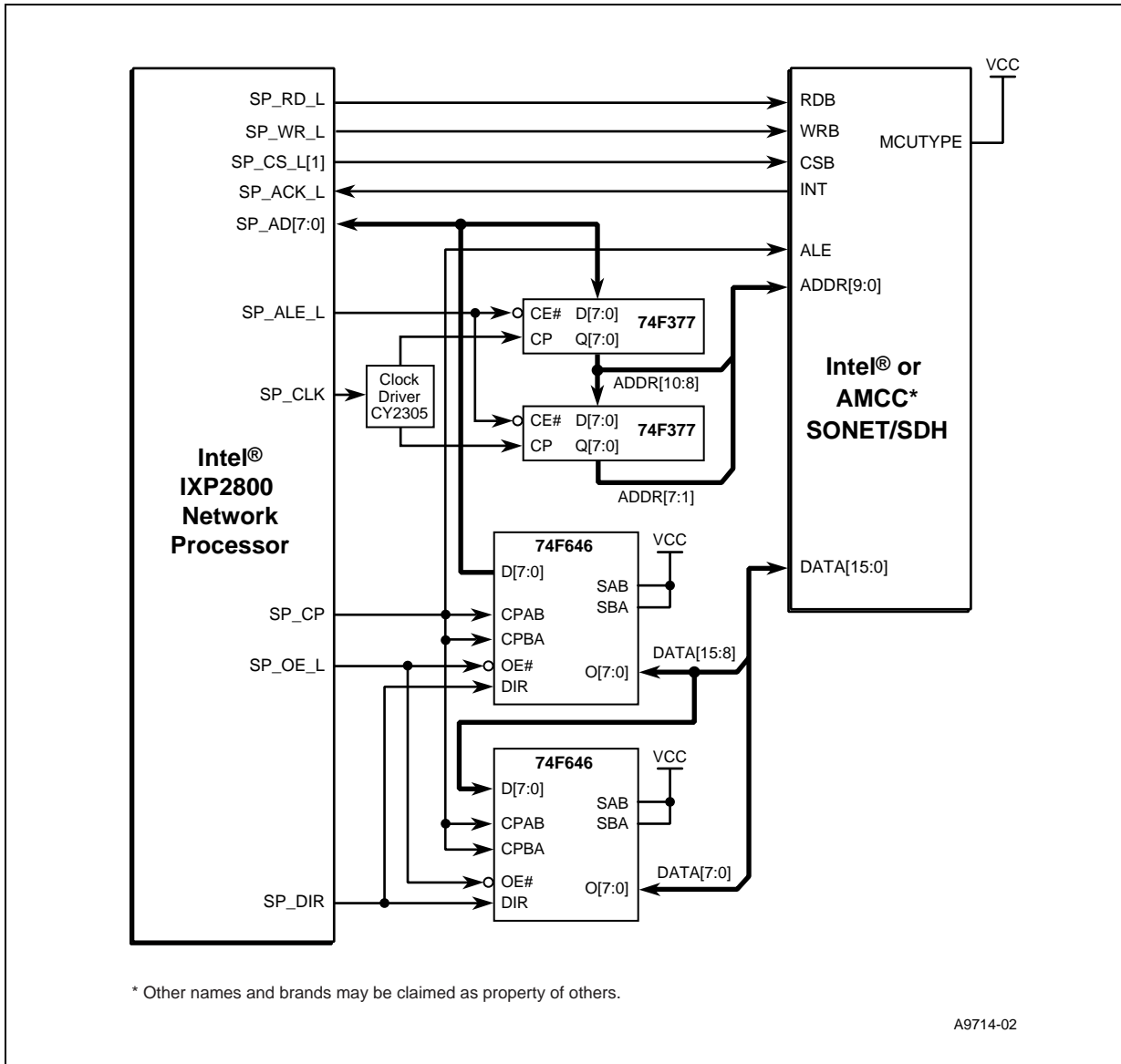
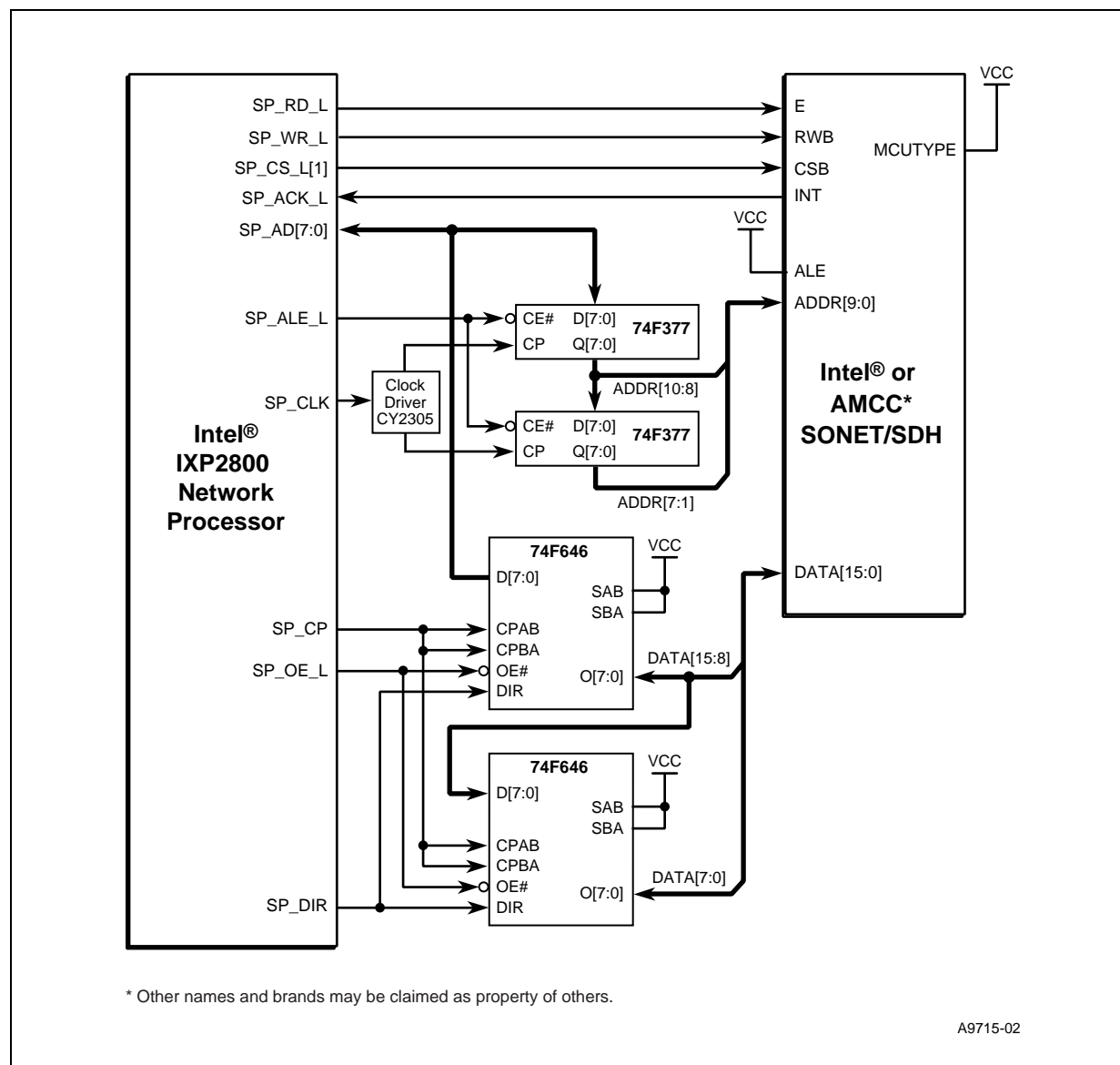


Figure 49. Mode 3 Second Interface Topology with Intel / AMCC SONET/SDH Device

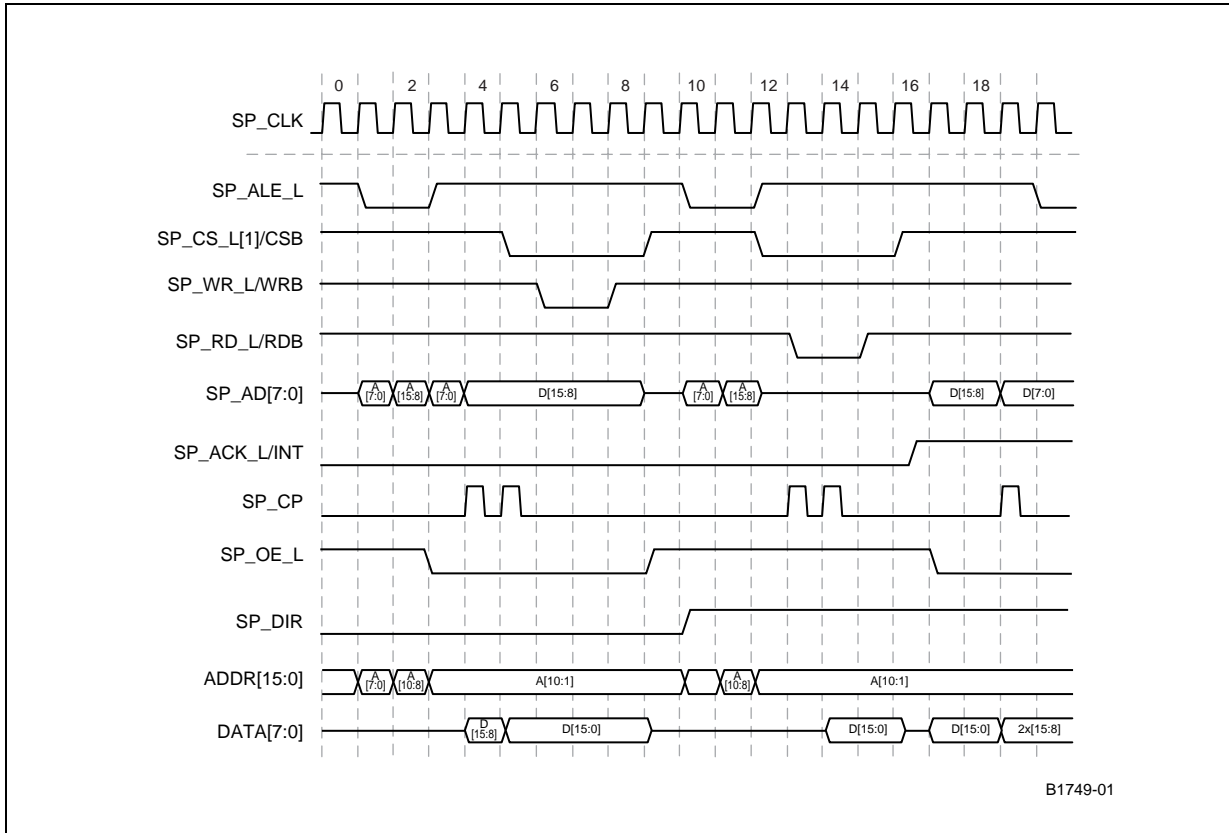


Mode 3 Write Interface Protocol

Figure 50 depicts a single write transaction launched from the IXP2800 Network Processor to the Intel and AMCC SONET/SDH device followed by two consecutive reads.

Compared with the Lucent TDAT042G5, this device has a shorter access time, about 8 clock cycles (i.e., 160 ns). In this case, an intervening cycle may not be needed for the write transactions. Therefore, the throughput is about 12.5 MB per second.

Figure 50. Mode 3 Single Write Transfer Followed by Read (B0)

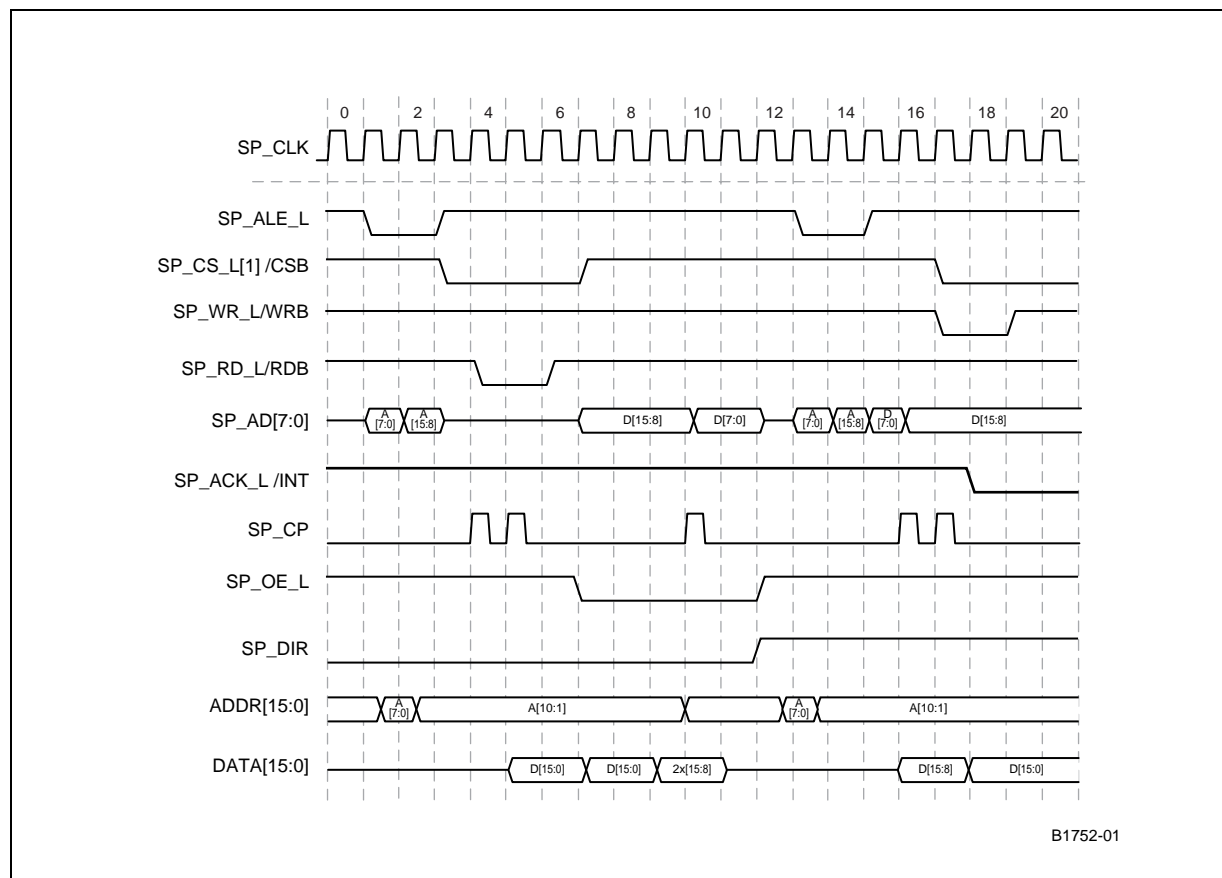


Mode 3 Read Interface Protocol

Figure 51 depicts a single read transaction launched from the IXP2800 to the Intel and AMCC SONET/SDH device followed by two consecutive writes.

Similarly, the access time is much better than the Lucent TDAT042G5. The access time is 8 clock cycles or 160ns for a 50 MHz SlowPort clock. Here, there are three intervening cycles between transactions. Therefore, the throughput is 11.1 MB per second.

Figure 51. Mode 3 Single Read Transfer Followed by Write (B0)



Mode 4 Interfacing Topology

Figure 52 demonstrates one of the topologies used to connect SlowPort to the Intel and AMCC SONET/SDH device.

Similar to the Lucent TDAT042G5 interface, the address and the data need demultiplexing. It requires a total of six buffers.

The RD_L, WR_L, and CS_L[1] entirely match the E, RWB, and CSB pins respectively in the Intel framer configured to Motorola mode. However, the INT has to be connected to the SP_ACK_L as the PMC-Sierra Interface does. The ALE pin can share the SP_CP. However, if it doesn't meet the timing, ALE pin can be tied high as shown in Figure 53.

It employs the same way to pack and unpack the data between the IXP2800 Network Processor SlowPort interface and the Intel and AMCC microprocessor interface.

For a write, W2B loads the data onto the 74F646 or equivalent tri-state buffers, using two clock cycles. In order to reduce the pin count, the 16-bit data are latched with the same pin (CS_L[1]), assuming that a turnaround cycle is inserted between the transaction cycles.

For a read, data are pipelined out of two 74F646 or equivalent tri-state buffers by B2S, using two consecutive clock cycles.

Figure 52. An Interface Topology with Intel / AMCC SONET/SDH Device in Motorola Mode

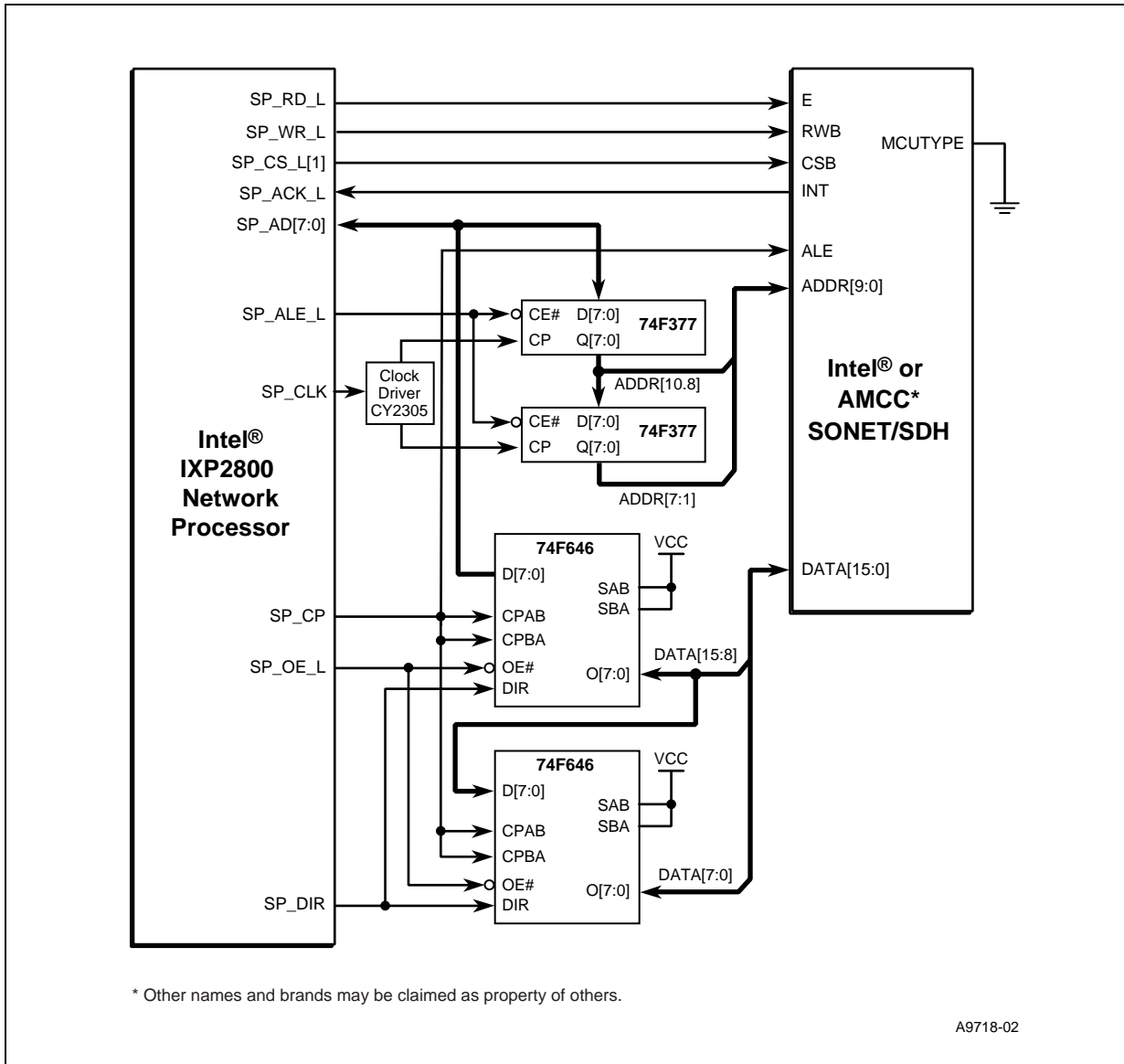
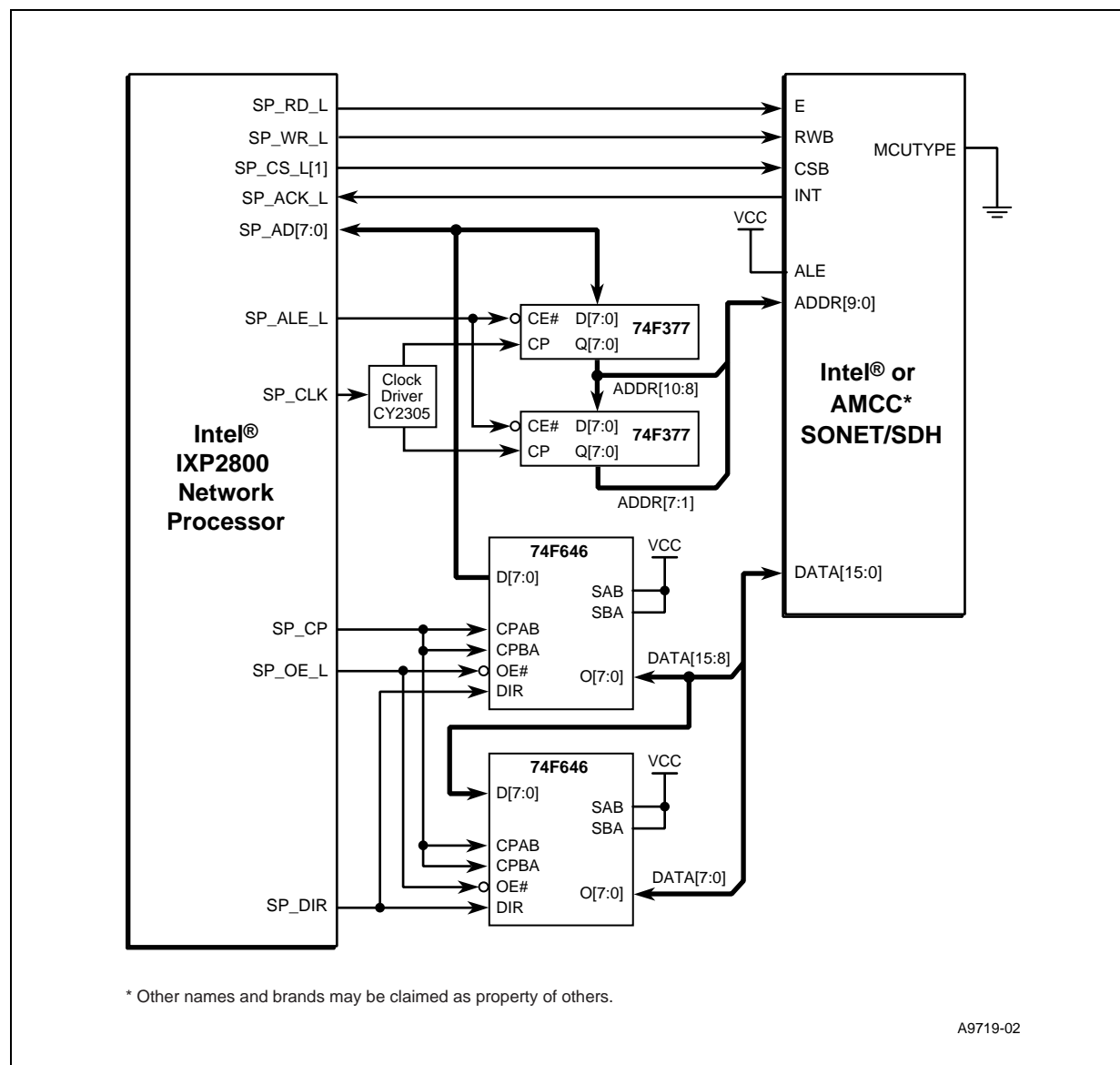


Figure 53. Second Interface Topology with Intel / AMCC SONET/SDH Device

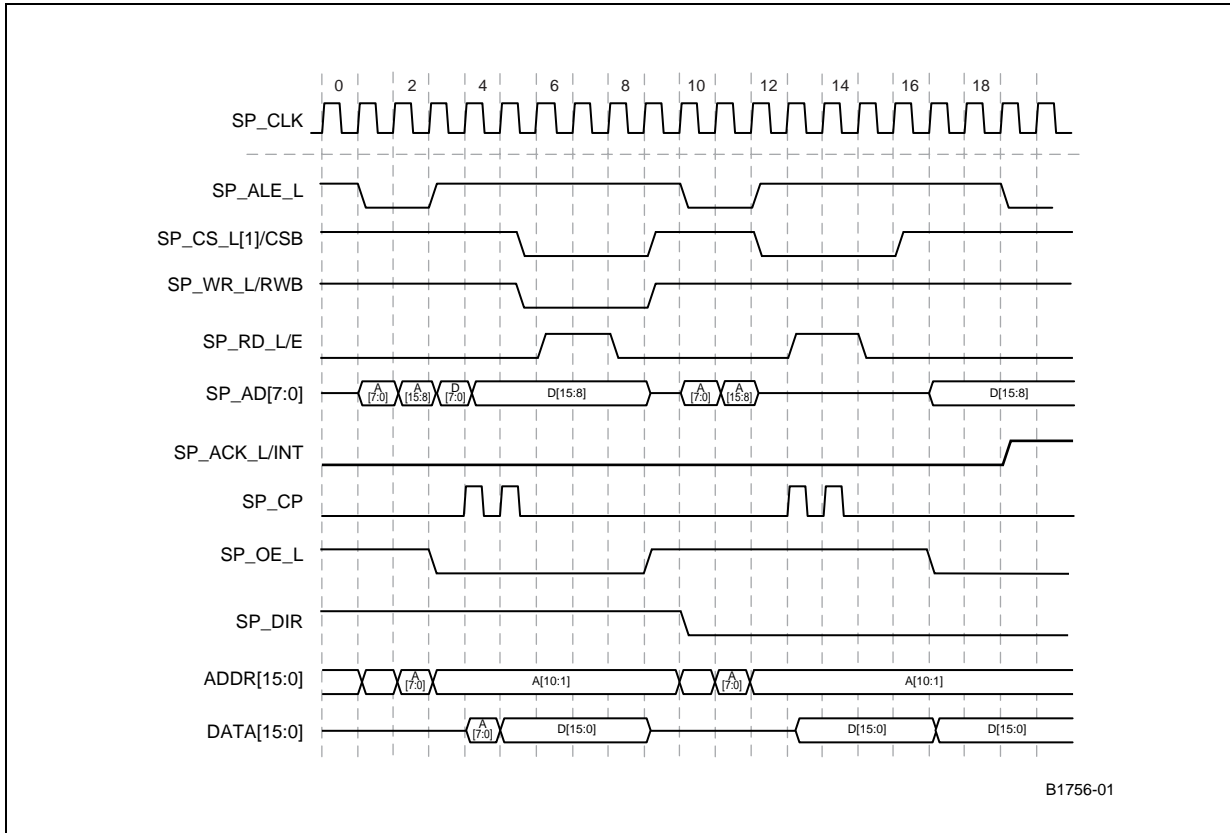


Mode 4 Write Interface Protocol

Figure 54 depicts a single write transaction launched from the IXP2800 Network Processor to the Intel and AMCC SONET/SDH device, followed by two consecutive reads.

Compared with the Lucent TDAT042G5, this device has a shorter access time, about 8 clock cycles (i.e., 120 ns). In this case, an intervened cycle may not be needed, therefore, the throughput is about 12.5 MB per second.

Figure 54. Mode 4 Single Write Transfer (B0)



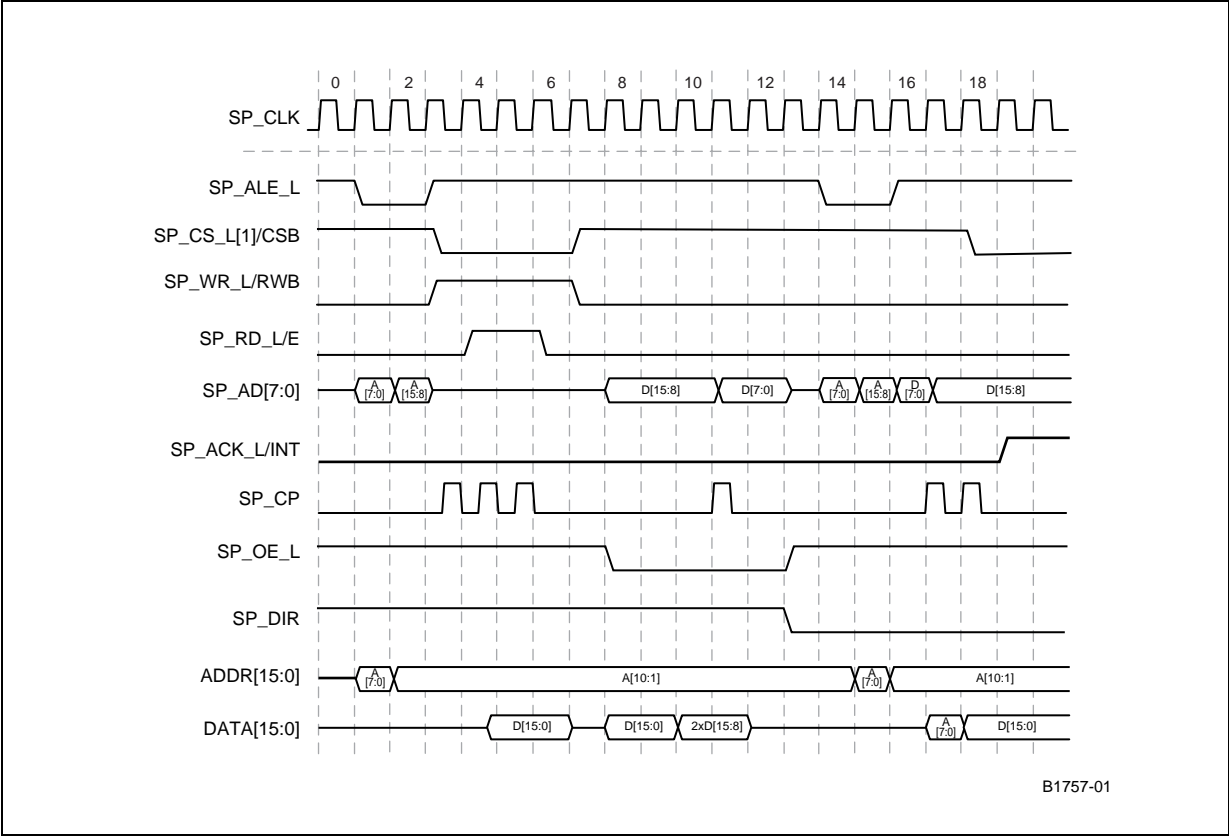


Mode 4 Read Interface Protocol

Figure 55, depicts a single read transaction launched from the IXP2800 Network Processor to the Intel and AMCC SONET/SDH device, followed by two consecutive writes.

Similarly, the access time is much better the Lucent TDAT042G5, the access time is about 8 clock cycles or 160ns. Here, we need an intervened cycle at the back. Therefore, the throughput is 11.2 MB per second.

Figure 55. Mode 4 Single Read Transfer (B0)



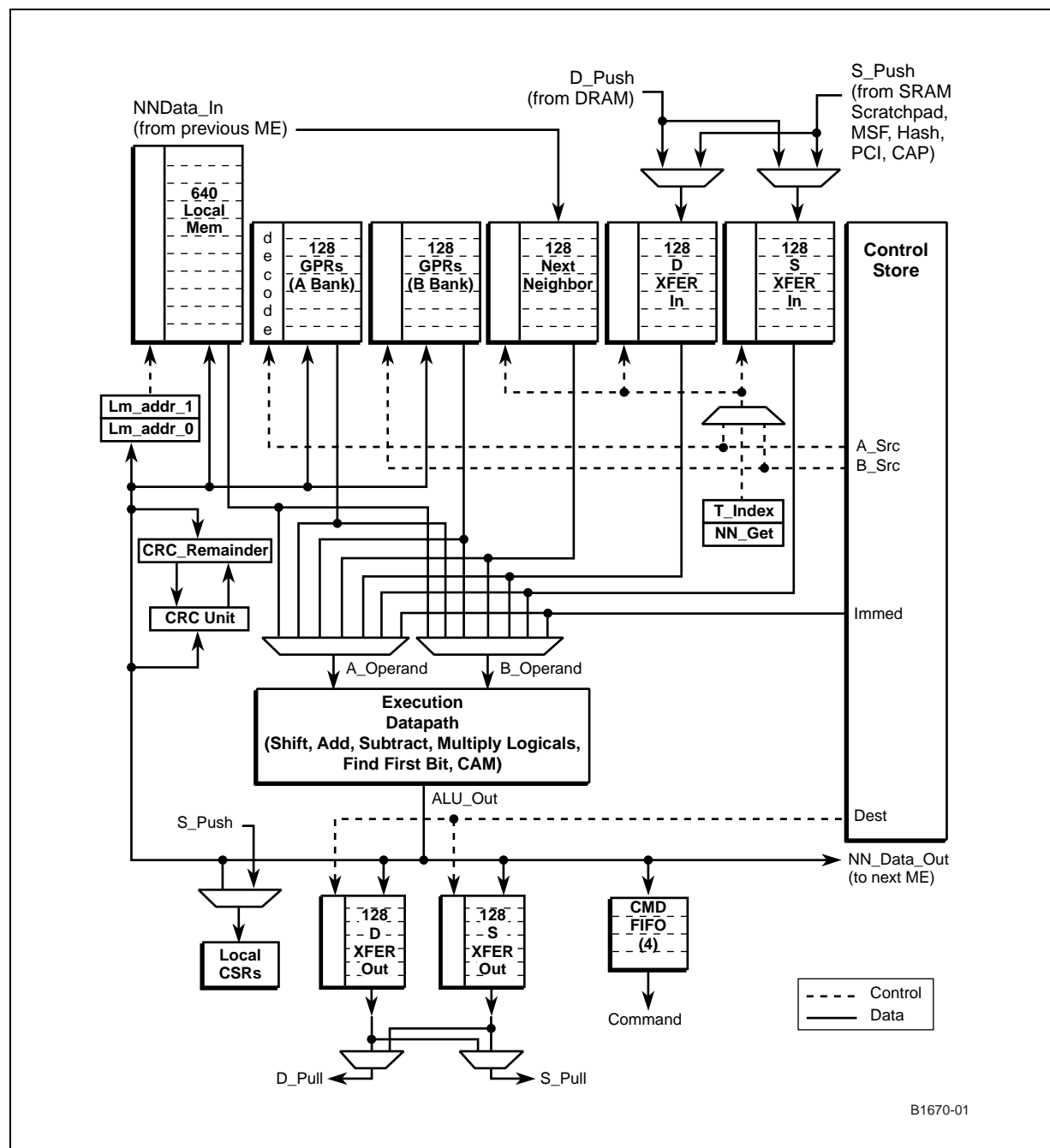
This section defines the Network Processor Microengine (ME). This is the second version of the Microengine, and is often referred to as the MEv2 (Microengine Version 2).

4.1 Overview

The following sections describe the programmer's view of the Microengine. The block diagram in [Figure 56](#) is used in the description. Note that this block diagram is simplified for clarity, not all interface signals are shown, and some blocks and connectivity have been omitted to make the diagram more readable. This block diagram does not show any pipeline stages, rather it shows the logical flow of information.

The Microengine provides support for software controlled multi-threaded operation. Given the disparity in processor cycle times versus external memory times, a single thread of execution will often block waiting for external memory operations to complete. Having multiple threads available allows for threads to interleave operation—there is often at least one thread ready to run while others are blocked.

Figure 56. Microengine Block Diagram





4.1.1 Control Store

The Control Store is a static RAM, which holds the program that the Microengine executes. It holds 8192 instructions, each of which is 40-bits wide. It is initialized by an external device which writes to Ustore_Addr and Ustore_Data Local CSRs.

The Control Store can optionally be protected by parity against soft errors. The parity protection is optional, so that it can be disabled for implementations that don't need or want to pay the cost for it. Parity checking is enabled by CTX_Enable[Control Store Parity Enable]. A parity error on an instruction read will halt the Microengine and assert an output signal that can be used as an interrupt.

4.1.2 Contexts

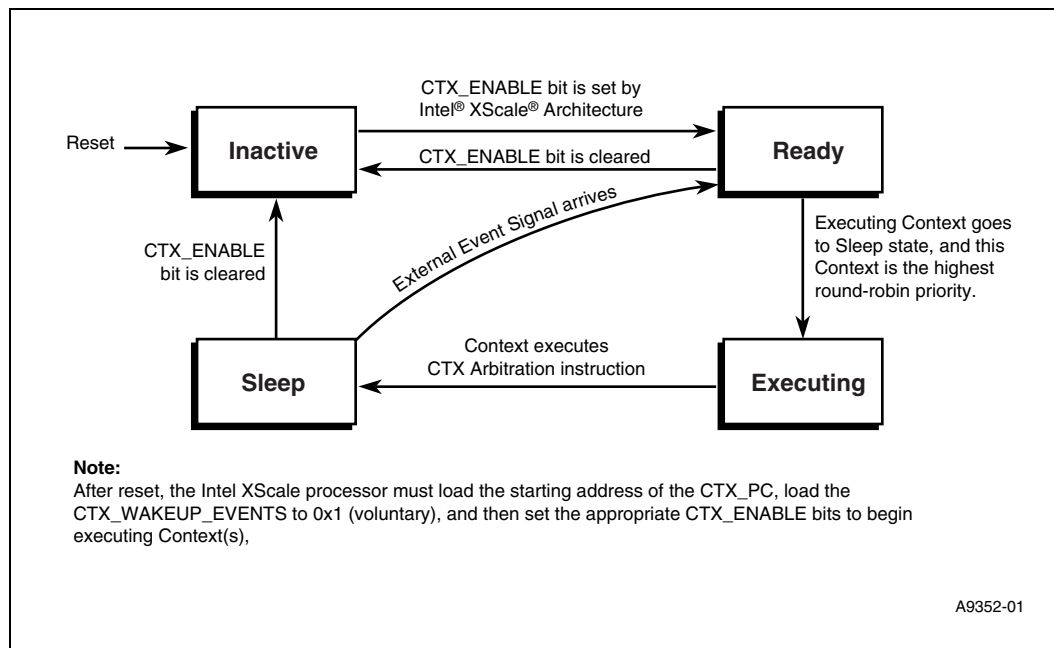
There are eight hardware Contexts available in the Microengine. To allow for efficient context swapping, each Context has its own register set, Program Counter, and Context specific Local Registers. Having a separate copy per Context eliminates the need to move Context specific information to/from shared memory and Microengine registers for each Context swap. Fast context swapping allows a Context to do computation while other Contexts wait for IO (typically external memory accesses) to complete or for a signal from another Context or hardware unit. *Note:* a context swap is similar to a taken branch in timing.

Each of the eight Contexts is always in one of four states.

1. Inactive — Some applications may not require all eight contexts. A Context is in the Inactive state when its CTX_Enable CSR enable bit is a '0'.
2. Executing — A Context is in Executing state when its context number is in Active_CTX_Status CSR. The executing Context's PC is used to fetch instructions from the Control Store. A Context will stay in this state until it executes an instruction that causes it to go to Sleep state (there is no hardware interrupt or preemption; Context swapping is completely under software control). At most one Context can be in Executing state at any time.
3. Ready — In this state, a Context is ready to execute, but is not because a different Context is executing. When the Executing Context goes to Sleep state, the Microengine's context arbiter selects the next Context to go to the Executing state from among all the Contexts in the Ready state. The arbitration is round robin.
4. Sleep — Context is waiting for external event(s) specified in the CTX_#_Wakeup_Events CSR to occur (typically, but not limited to, an IO access). In this state the Context does not arbitrate to enter the Executing state.

The state diagram in [Figure 57](#) illustrates the Context state transitions. Each of the eight Contexts will be in one of these states. At most one Context can be in Executing state at a time; any number of Contexts can be in any of the other states.

Figure 57. Context State Transition Diagram



The Microengine is in Idle state whenever no Context is running (all Contexts are in either Inactive or Sleep states). This state is entered:

1. After reset (because CTX_Enable Local CSR is clear, putting all Contexts into Inactive states).
2. When a context swap is executed, but no context is ready to wakeup.
3. When a `ctx_arb[bpt]` instruction is executed by the Microengine (this is a special case of #2 above, since the `ctx_arb[bpt]` clears CTX_Enable, putting all Contexts into Inactive states).

The Microengine provides the following functionality during Idle state:

1. The Microengine continuously checks if a Context is in Ready state. If so, a new Context begins to execute. If no Context is Ready, the Microengine remains in the Idle state.
2. Only the ALU instructions are supported. They are used for debug via special hardware defined in number 3 below.
3. A write to the Ustore_Addr Local CSR with the Ustore_Addr[ECS] bit set, causing the Microengine to repeatedly execute the instruction pointed by the address specified in the Ustore_Addr CSR. Only the ALU instructions are supported in this mode. Also, the result of the execution is written to the ALU_Out Local CSR rather than a destination register.
4. A write to the Ustore_Addr Local CSR with the Ustore_Addr[ECS] bit set, followed by a write to the Ustore_Data Local CSR loads an instruction into the Control Store. After the Control Store is loaded, execution proceeds as described in number 3 above. Note that the write to Ustore_Data causes Ustore_Addr to increment, so it must be written back to the address of the desired instruction.



4.1.3 Datapath Registers

As shown in the block diagram in [Figure 56](#), each Microengine contains four types of 32-bit datapath registers:

- 256 General Purpose Registers
- 512 Transfer Registers
- 128 Next Neighbor Registers
- 640 32-bit words of Local Memory

4.1.3.1 General-Purpose Registers (GPRs)

GPRs are used for general programming purposes. They are read and written exclusively under program control. GPRs, when used as a source in an instruction, supply operands to the execution datapath. When used as a destination in an instruction, they are written with the result of the execution datapath. The specific GPRs selected are encoded in the instruction.

The GPRs are physically and logically contained in two banks, GPR A, and GPR B, defined in [Table 59](#).

Note: The Microengine registers are defined in the *IXP2400/IXP2800 Network Processor Programmers Reference Manual*.

4.1.3.2 Transfer Registers

Transfer registers (abbreviated Xfer registers) are used for transferring data to and from the Microengine and locations external to the Microengine, (for example DRAMs, SRAMs etc.). There are four types of transfer registers.

- S_Transfer_In
- S_Transfer_Out
- D_Transfer_In
- D_Transfer_Out

Transfer_In registers, when used as a source in an instruction, supply operands to the execution datapath. The specific register selected is either encoded in the instruction, or selected indirectly using T_Index. Transfer_In registers are written by external units based on the Push_ID input to the Microengine.

Transfer_Out registers, when used as a destination in an instruction, are written with the result from the execution datapath. The specific register selected is encoded in the instruction, or selected indirectly via T_Index. Transfer_Out registers supply data to external units based on the Pull_ID input to the Microengine.

As shown in [Figure 56](#), the S_Transfer_In and D_Transfer_In registers connect to both the S_Push and D_Push busses via a multiplexor internal to the Microengine. Additionally, the S_Transfer_Out and D_Transfer_Out Transfer registers connect to both the S_Pull and D_Pull busses. This feature enables a programmer to use the either type of transfer register regardless of the source or destination of the transfer.



Typically, the external units access the Transfer registers in response to commands sent by the MEs; the commands are sent in response to instructions executed by the Microengine (for example, the command instructs a SRAM controller to read from external SRAM, and place the data into a S_Transfer_In register). However, it is possible for an external unit to access a given Microengine's Transfer registers either autonomously, or under control of a different Microengine, or the Intel Xscale® core, etc. The Microengine interface signals controlling writing/reading of the Transfer_In/Transfer_Out registers are independent of the operation of the rest of the Microengine.

4.1.3.3 Next Neighbor Registers

A new feature added for the Microengine Version 2 are 128 Next Neighbor registers that provide a dedicated datapath for transferring data from the previous/next neighbor Microengine.

Next Neighbor registers, when used as a source in an instruction, supply operands to the execution datapath. They are written in two different ways 1) by an external entity, typically, but not limited to, another, adjacent Microengine, or 2) by the same Microengine they are in, as controlled by CTX_Enable[NN_Mode].

The specific register is selected in one of two ways: (1) Context-relative, the register number is encoded in the instruction, or (2) as a Ring, selected via NN_Get and NN_Put CSR registers.

When CTX_Enable[NN_Mode] is '0' -- When Next Neighbor is used as a destination in an instruction, the instruction result data is sent out of the Microengine, typically to another, adjacent Microengine.

When CTX_Enable[NN_Mode] is '1' -- When Next Neighbor is used as a destination in an instruction, the instruction result data is written to the selected Next Neighbor register in the Microengine. Note that there is a 5 instruction latency until the newly written data may be read. The data is not sent out of the Microengine as it would be when CTX_Enable[NN_Mode] is '0'.

Table 58. Next Neighbor Write as a Function of CTX_Enable[NN_Mode]

NN_Mode	Where Does Write Go?	
	External	NN Register in This ME
0	Yes	No
1	No	Yes

4.1.3.4 Local Memory

Local Memory is addressable storage located in the Microengine, organized as 640 32-bit words. Local Memory is read and written exclusively under program control. Local Memory supplies operands to the execution datapath as a source, and receives results as a destination. The specific Local Memory location selected is based on the value in one of the Local Memory_Addr registers, which are written by local_CSR_wr instructions. There are two LM_Addr registers per Context and a working copy of each. When a Context goes to Sleep state, the value of the working copies is put into the Context's copy of LM_Addr. When the Context goes to Executing state, the value in its copy of LM_Addr is put into the working copies. The choice of LM_Addr_0 or LM_Addr_1 is selected in the instruction.



It is also possible to make use of both or one LM_Addrs as global by setting CTX_Enable[LM_Addr_0_Global] and/or CTX_Enable[LM_Addr_1_Global]. When used globally, all Contexts use the working copy of LM_Addr in place of their own Context specific one; the Context specific ones are unused.

4.1.4 Addressing Modes

GPRs can be accessed in two different addressing modes: Context-Relative and Absolute. Some instructions can specify either mode, other instructions can specify only Context-Relative mode.

- Transfer and Next Neighbor registers can be accessed in Context-Relative and Indexed modes.
- Local Memory is accessed in Indexed mode.
- The addressing mode in use is encoded directly into each instruction, for each source and destination specifier.

4.1.4.1 Context-Relative Addressing Mode

The GPRs are logically subdivided into equal regions such that each Context has exclusive access to one of the regions. The number of regions is configured in the CTX_Enable CSR, and can be either 4 or 8. Thus a Context-Relative register name is actually associated with multiple different physical registers. The actual register to be accessed is determined by the Context making the access request (the Context number is concatenated with the register number specified in the instruction—see Table 59). Context-Relative addressing is a powerful feature that enables eight different contexts to share the same microcode, yet maintain separate data.

Table 59 shows how the Context number is used in selecting the register number in relative mode. The register number in Table 59 is the Absolute GPR address, or Transfer or Next Neighbor Index number to use to access the specific Context-Relative register. For example, with 8 active Contexts, Context-Relative Register 0 for Context 2 is Absolute Register Number 32.

Table 59. Registers Used By Contexts in Context-Relative Addressing Mode

Number of Active Contexts	Active Context Number	GPR Absolute Register Numbers		S Transfer or Neighbor Index Number	D Transfer Index Number
		A Port	B Port		
8	0	0-15	0-15	0-15	0-15
	1	16-31	16-31	16-31	16-31
	2	32-47	32-47	32-47	32-47
	3	48-63	48-63	48-63	48-63
	4	64-79	64-79	64-79	64-79
	5	80-95	80-95	80-95	80-95
	6	96-111	96-111	96-111	96-111
	7	112-127	112-127	112-127	112-127
4	0	0-31	0-31	0-31	0-31
	2	32-63	32-63	32-63	32-63
	4	64-95	64-95	64-95	64-95
	6	96-127	96-127	96-127	96-127



4.1.4.2 Absolute Addressing Mode

With Absolute addressing, any GPR can be read or written by any one of the eight Contexts in an Microengine. Absolute addressing enables register data to be shared among all of the Contexts, for example for global variables, or for parameter passing. All 256 GPRs can be read by Absolute address.

4.1.4.3 Indexed Addressing Mode

With Indexed addressing, any Transfer or Next Neighbor register can be read or written by any one of the eight Contexts in an Microengine. Indexed addressing enables register data to be shared among all of the Contexts. For indexed addressing the register number comes from the T_Index register for Transfer Registers or NN_Put and NN_Get registers (for Next Neighbor Registers).

4.2 Local CSRs

Local Control and Status registers (CSRs) are external to the Execution Datapath, and hold specific purpose information. They can be read and written by special instructions (`local_csr_rd` and `local_csr_wr`) and are typically accessed less frequently than datapath registers. Because Local CSRs are not built in the datapath, there is a write to use delay of either three or four cycles, and a read to consume penalty of one cycle.

4.3 Execution Datapath

The Execution Datapath can take one or two operands, perform an operation, and optionally write back a result. The sources and destinations can be GPRs, Transfer registers, Next Neighbor registers, and Local Memory. The operations are shifts, add/subtract, logicals, multiply, byte align, and find first bit set.

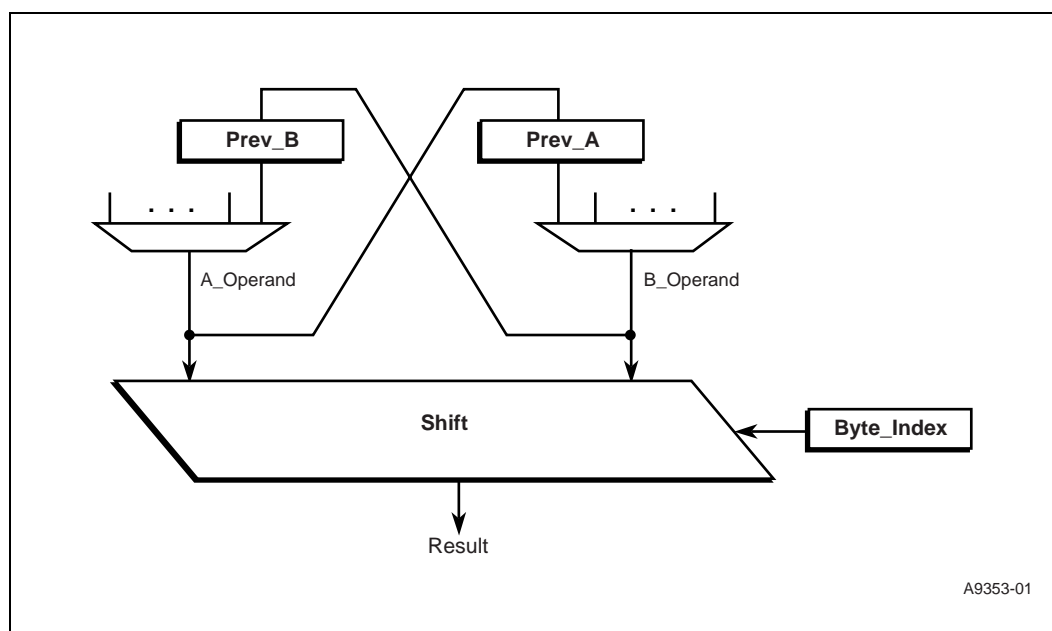
4.3.1 Byte Align

The datapath provides a mechanism to move data from source register(s) to any destination register(s) with byte aligning. Byte aligning takes four consecutive bytes from two concatenated values (8 bytes), starting at any of four byte boundaries (0, 1, 2, 3), and based on the endian-type (which is defined in the instruction opcode), as shown in [Table 60](#). The four bytes are taken from two concatenated values. Four bytes are always supplied from a temporary register that always holds the A or B operand from the previous cycle, and the other four bytes from the B or A operand of the Byte Align instruction. The operation is described below using the block diagram [Figure 58](#). The alignment is controlled by the 2 lsbs of the Byte_Index Local CSR.

Table 60. Align Value and Shift Amount

Align Value (in Byte_Index[1:0])	Right Shift Amount (# of Bits) (Decimal)	
	Little Endian	Big Endian
0	0	32
1	8	24
2	16	16
3	24	8

Figure 58. Byte Align Block Diagram



Example 23 shows an align sequence of instructions and the value of the various operands. **Table 61** shows the data in the registers for this example. The value in **Byte_Index[1:0]** CSR (which controls the shift amount) for this example is 2.

Table 61. Register Contents for Example 23

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	0	1	2	3
1	4	5	6	7
2	8	9	A	B
3	C	D	E	F

Example 23. Big Endian Align

Instruction	Prev B	A Operand	B Operand	Result
Byte_align_be[--, r0]	--	--	0123	--
Byte_align_be[dest1, r1]	0123	0123	4567	2345
Byte_align_be[dest2, r2]	4567	4567	89AB	6789
Byte_align_be[dest3, r3]	89AB	89AB	CDEF	ABCD
NOTE: A Operand comes from Prev_B register during byte_align_be instructions.				

Example 24 shows another sequence of instructions and the value of the various operands. Table 62, shows the data in the registers for this example.

The value in Byte_Index[1:0] CSR (which controls the shift amount) for this example is 2.

Table 62. Register Contents for Example 24

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	3	2	1	0
1	7	6	5	4
2	B	A	9	8
3	F	E	D	C

Example 24. Little Endian Align

Instruction	A Operand	B Operand	Prev A	Result
Byte_align_le[--, r0]	3210	--	--	--
Byte_align_le[dest1, r1]	7654	3210	3210	5432
Byte_align_le[dest2, r2]	BA98	7654	7654	9876
Byte_align_le[dest3, r3]	FEDC	BA98	BA98	DCBA
NOTE: B Operand comes from Prev_A register during byte_align_le instructions.				

As the examples show, byte aligning “n” words takes “n+1” cycles due to the first instruction needed to start the operation.

Another mode of operation is to use the T_Index register with post-increment, to select the source registers. T_Index operation is described later in this chapter.



4.3.2 CAM

The block diagram in [Figure 59](#) is used to explain the CAM operation.

The CAM has 16 entries. Each entry stores a 32-bit value, which can be compared against a source operand by instruction:

```
CAM_Lookup[dest_reg, source_reg]
```

All entries are compared in parallel, and the result of the lookup is a 9-bit value which is written into the specified destination register in bits 11:3, with all other bits of the register zero (the choice of bits 11:3 is explained below). The result can also optionally be written into either of the LM_Addr registers (see below in this section for details).

The 9-bit result consists of four State bits (dest_reg[11:8]), concatenated with a 1-bit Hit/Miss indication (dest_reg[7]), concatenated with 4-bit entry number (dest_reg[6:3]). All other bits of dest_reg are written with 0. Possible results of the lookup are:

- miss (0) — lookup value is not in CAM, entry number is Least Recently Used entry (which can be used as a suggested entry to replace), and State bits are 0000.
- hit (1) — lookup value is in CAM, entry number is entry which has matched; State bits are the value from the entry which has matched.

Note: The State bits are data associated with the entry. State bits are only used by software. There is no implication of ownership of the entry by any Context. The State bits hardware function is:

- the value is set by software (at the time the entry is loaded, or changed in an already loaded entry).
- its value is read out on a lookup that hits, and used as part of the status written into the destination register.
- its value can be read out separately (normally only used for diagnostic or debug).

The LRU (Least Recently Used) Logic maintains a time-ordered list of CAM entry usage. When an entry is loaded, or matches on a lookup, it is marked as MRU (Most Recently Used). Note that a lookup that misses does *not* modify the LRU list.

The CAM is loaded by instruction:

```
CAM_Write[entry_reg, source_reg, state_value]
```

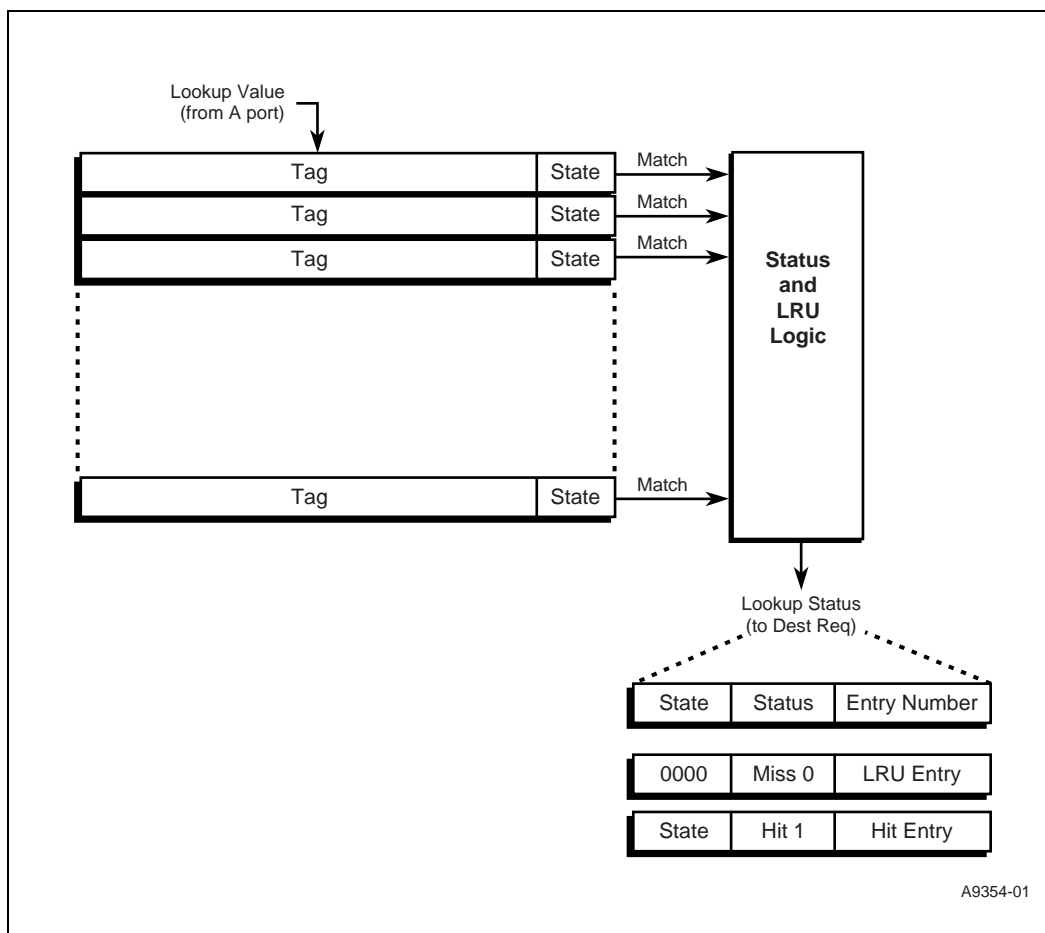
The value in the register specified by source_reg is put into the Tag field of the entry specified by entry_reg. The value for the State bits of the entry is specified in the instruction as state_value.

The value in the State bits for an entry can be written, without modifying the Tag, by instruction:

```
CAM_Write_State[entry_reg, state_value]
```

Note: CAM_Write_State does *not* modify the LRU list.

Figure 59. CAM Block Diagram



One possible way to use the result of a lookup is to dispatch to the proper code using instruction:

```
jump[register, label#],defer [3]
```

where the register holds the result of the lookup. The State bits can be used to differentiate cases where the data associated with the CAM entry is in flight, or is pending a change, etc. Because the lookup result was loaded into bits[11:3] of the destination register, the jump destinations are spaced 8 instructions apart. This is a balance between giving enough space for many applications to complete their task without having to jump to another region, vs consuming too much Control Store. Another way to use the lookup result is to branch on just the hit miss bit, and use the entry number as a base pointer into a block of Local Memory.

When enabled, the CAM lookup result is loaded into Local_Addr as follows:

LM_Addr[5:0] = 0 ([1:0] are read-only bits)

LM_Addr[9:6] = lookup result [6:3] (entry number)

LM_Addr[11:10] = constant specified in instruction

This function is useful when the CAM is used as a cache, and each entry is associated with a block of data in Local Memory. Note that the latency from when CAM_Lookup executes until the LM_Addr is loaded is the same as when LM_Addr is written by a Local_CSR_Wr instruction.

The Tag and State bits for a given entry can be read by instructions:

```
CAM_Read_Tag[dest_reg, entry_reg]
CAM_Read_State[dest_reg, entry_reg]
```

The Tag value and State bits value for the specified entry is written into the destination register, respectively for the two instructions (the State bits are placed into bits [11:8] of dest_reg, with all other bits 0). Reading the tag is useful in the case where an entry needs to be evicted to make room for a new value—the lookup of the new value results in a miss, with the LRU entry number returned as a result of the miss. The CAM_Read_Tag instruction can then be used to find the value that was stored in that entry. An alternative would be to keep the tag value in a GPR. These two instructions can also be used by debug and diagnostic software. Neither of these modify the state of the LRU pointer.

Note: The following rules must be adhered to when using the CAM.

- CAM is not reset by Microengine reset. Software must either do a CAM_clear prior to using the CAM to initialize the LRU and clear the tags to zero, or explicitly write all entries with CAM_write.
- No two tags can be written to have same value. If this rule is violated, the result of a lookup that matches that value will be unpredictable, and LRU state is unpredictable.

The value 0x00000000 can be used as a valid lookup value. However, note that CAM_clear instruction puts 0x00000000 into all tags. So in order to not violate rule 2 after doing CAM_clear, it is necessary to write all entries to unique values prior to doing a lookup of 0x00000000. An algorithm for debug software to find out the contents of the CAM is shown in [Example 25](#).

Example 25. Algorithm for Debug Software to Find out the Contents of the CAM

```
; First read each of the tag entries. Note that these reads
; don't modify the LRU list or any other CAM state.
tag[0] = CAM_Read_Tag(entry_0);
.....
tag[15] = CAM_Read_Tag(entry_15);
; Now read each of the state bits
state[0] = CAM_Read_State(entry_0);
...
state[15] = CAM_Read_State(entry_15);
; Knowing what tags are in the CAM makes it possible to
; create a value that is not in any tag, and will therefore
; miss on a lookup.

; Next loop through a sequence of 16 lookups, each of which will
; miss, to obtain the LRU values of the CAM.
for (i = 0; i < 16; i++)
    BEGIN_LOOP
        ; Do a lookup with a tag not present in the CAM. On a
        ; miss, the LRU entry will be returned. Since this lookup
        ; missed the LRU state is not modified.
        LRU[i] = CAM_Lookup(some_tag_not_in_cam);
        ; Now do a lookup using the tag of the LRU entry. This
        ; lookup will hit, which makes that entry MRU.
        ; This is necessary to allow the next lookup miss to
        ; see the next LRU entry.
        junk = CAM_Lookup(tag[LRU[i]]);
    END_LOOP
; Because all entries were hit in the same order as they were
; LRU, the LRU list is now back to where it started before the
; loop executed.
; LRU[0] through LRU[15] holds the LRU list.
```



The CAM can be cleared with CAM_Clear instruction. This instruction writes 0x00000000 simultaneously to all entries tag, clears all the state bits, and puts the LRU into an initial state (where entry 0 is LRU, ..., entry 15 is MRU).

4.4 CRC Unit

The CRC Unit operates in parallel with the Execution Datapath. It takes two operands, performs a CRC operation, and writes back a result. CRC-CCITT, CRC-32, CRC-10, CRC-5, and iSCSI polynomials are supported. One of the operands is the CRC_Remainder Local CSR, and the other is a GPR, Transfer In Register, Next Neighbor, or Local Memory, specified in the instruction and passed through the Execution Datapath to the CRC Unit. The instruction specifies the CRC operation type, whether to swap bytes and or bits, and which bytes of the operand to include in the operation. The result of the CRC operation is written back into CRC_Remainder. The source operand can also be written into a destination register (however the byte/bit swapping and masking do not affect the destination register; they only affect the CRC computation). This allows moving data, for example, from S Transfer In registers to S Transfer Out registers at the same time as computing the CRC.

4.5 Event Signals

Event Signals are used to coordinate a program with completion of external events. For example, when a Microengine issues a command to an external unit to read data (which will be written into a Transfer_In register), the program must insure that it does not try to use the data until the external unit has written it. There is no hardware mechanism to flag that a register write is pending, and then prevent the program from using it. Instead the coordination is under software control, with hardware support.

When the program issues the command to the external event, it can request that the external unit supply an indication (called an Event Signal) that the command has been completed. There are 15 Event Signals per Context that can be used, and Local CSRs per Context to track which Event Signals are pending and which have been returned. The Event Signals can be used to move a Context from Sleep state to Ready state, or alternatively, the program can test and branch on the status of Event Signals.

Event Signals can be set in nine different ways.

1. When data is written into S_Transfer_In Registers (part of S_Push_ID input)
2. When data is written into D_Transfer_In Registers (part of D_Push_ID input)
3. When data is taken from S_Transfer_Out Registers (part of S_Pull_ID input)
4. When data is taken from D_Transfer_Out Registers (part of D_Pull_ID input)
5. On InterThread_Sig_In input
6. On NN_Sig_In input
7. On Prev_Sig_In input
8. On write to Same_ME_Signal Local CSR
9. By Internal Timer

Any or all Event Signals can be set by any of the above sources.



When a Context goes to Sleep state (executes a `ctx_arb` instruction, or a Command instruction with `ctx_swap` token), it specifies which Event Signal(s) it requires to be put in Ready state. `Ctx_arb` instruction also specifies if the logical AND or logical OR of the Event Signal(s) is needed to put the Context into Ready state.

When a Context Event Signals arrive, it goes to Ready state, and then to Executing state. In the case where the Event Signal is linked to moving data into or out of Transfer registers (numbers 1 through 4 in the list above), the code can safely use the Transfer register as the first instruction (for example, using a `Transfer_In` register as a source operand will get the new read data). The same is true when the Event Signal is tested for branches (`br_=signal` or `br_!signal` instructions).

The `ctx_arb` instruction, `CTX_Sig_Events`, and `CTX_Wakeup_#_Events` Local CSR descriptions provide details.

4.5.1 Microengine “Endianness”

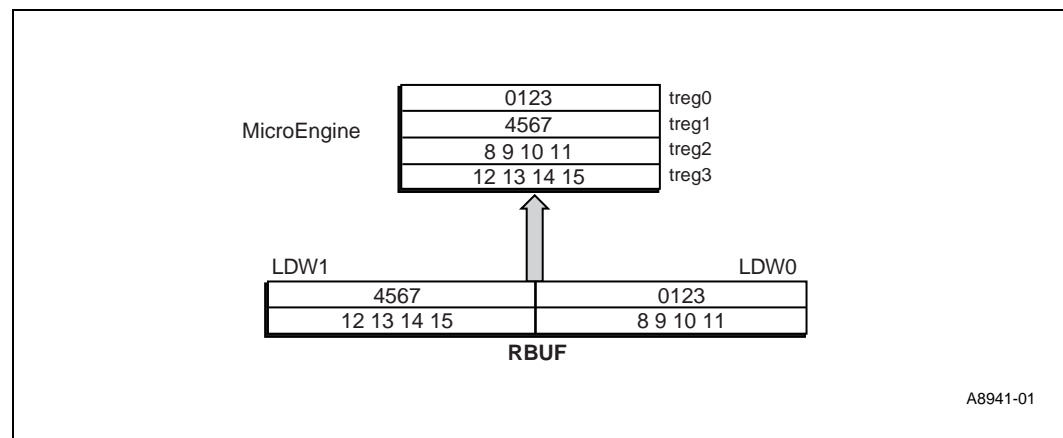
Microengine operation from an “endian” point of view can be divided into following categories:

- Read from RBUF (64-bits)
- Write to TBUF (64-bits)
- Read/write from/to SRAM
- Read/write from/to DRAM
- Read/write from/to SHAC and other CSRs
- Write to Hash

4.5.1.1 Read from RBUF (64-bits)

Data in RBUF is arranged in LWBE order. Whenever Microengine reads from RBUF, the low order long word (LDW0) is transferred into Microengine transfer register 0 (`treg0`), the high order long word (LDW1) is transferred into `treg1`, and so on. This is explained in [Figure 60](#).

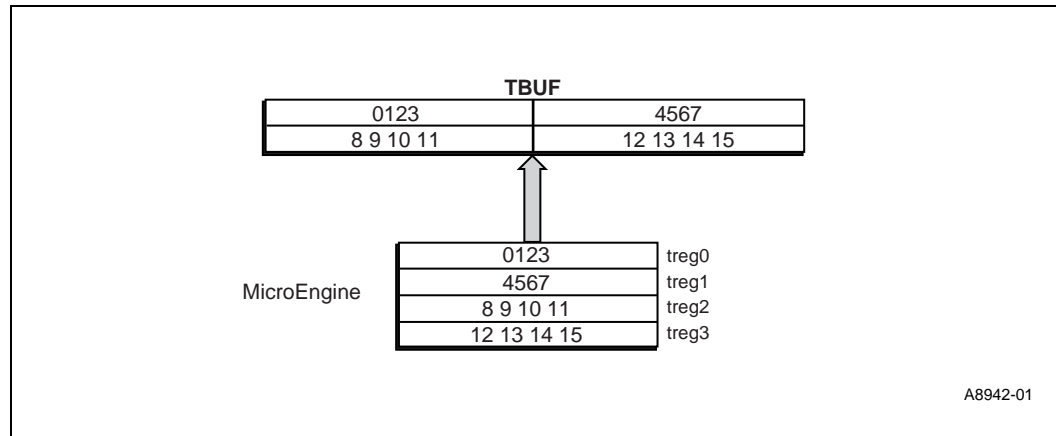
Figure 60. Read from RBUF (64-bits)



4.5.1.2 Write to TBUF

Data in TBUF is arranged in LWBE order. When writing from Microengine transfer registers to TBUF, treg0 goes into LDW0, treg1 goes into LDW1, and so on. See Figure 61.

Figure 61. Write to TBUF (64-bits)



4.5.1.3 Read/Write from/to SRAM

Data inside SRAM is in big-endian order. While transferring data from SRAM to a Microengine, no endianness is involved and first-read data goes into the first transfer register specified, the next read data into the second and so on.

4.5.1.4 Read/Write from/to DRAM

Data inside DRAM is in LWBE order. When a Microengine reads from DRAM, LDW0 goes into the first transfer register specified in the instruction, LDW1 goes into the next, and so on. While writing to DRAM, treg0 goes first followed by treg1 and both are combined in the DRAM controller as {LDW1, LDW0} and written as a 64-bit quantity into DRAM.

4.5.1.5 Read/Write from/to SHAC and Other CSRs

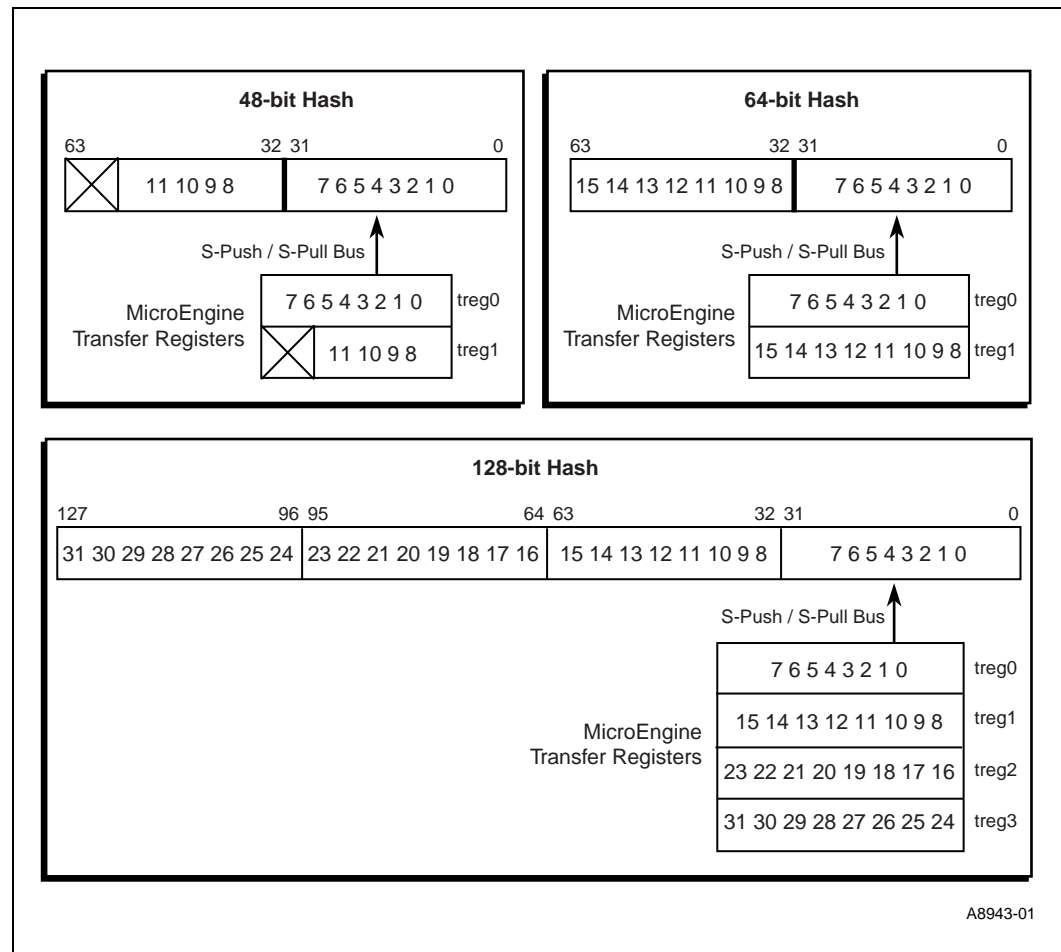
Read and write from SHAC and other CSRs happen as 32-bits operation only and are endian independent. Low byte goes into the low byte of transfer register and high byte goes into high byte of transfer register.



4.5.1.6 Write to Hash Unit

Figure 62 explains 48-bit, 64-bit, and 128-bit hash operations. When the Microengine transfers a 48-bit hash operand to the hash unit, the operand resides in two transfer registers and is transferred, as shown in Figure 62. In the second long word transfer, only the lower half is valid. Hash unit concatenates the two long words as shown in Figure 62. Similarly, 64-bit and 128-bit hash operand transfers from the Microengine to the hash unit happen as shown in Figure 62.

Figure 62. 48-bit, 64-bit, and 128-bit Hash Operand Transfers



4.5.2 Media Access

Media operation can be divided in two parts:

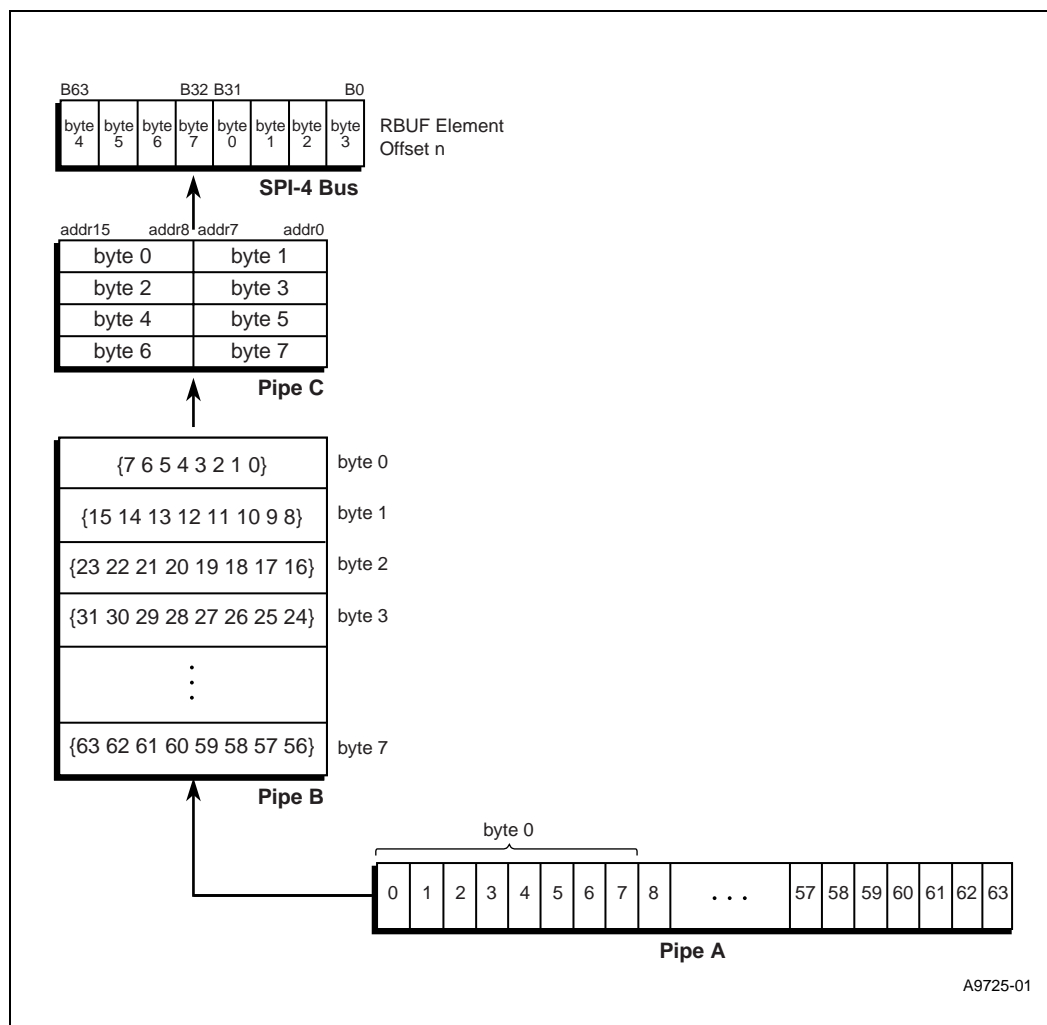
- Read from RBUF (Section 4.5.2.1)
- Write to TBUF (Section 4.5.2.2)

4.5.2.1 Read from RBUF

To analyze the endianness on the media receive interface and how bytes are arranged inside RBUF, a brief introduction of how bytes are generated from the serial interface is provided here. Pipe A denote the serial stream of data received at the serial interface (SERDES). Bit 0 of byte0 comes first followed by bit1 and so on. Pipe B converts this bit stream into byte stream byte0...byte7 and so on. So byte 0 currently is the least significant byte received. In PipeC before being transmitted to the SPI-4 interface, these bytes are organized in 16-bit words in big-endian order where byte0 is at B[15:8] and byte1 is at B[7:0].

When the SPI-4 interface inside the IXP2800 received these 16-bit words, they are put into RBUF in LWBE order where long words inside one RBUF entry are organized in little-endian order as shown in one RBUF element in Figure 63. In the least-significant-longword, byte0 is at higher address than byte3 (therefore big endian). Similarly in the most-significant-longword byte4 is at higher address than byte7 (therefore big endian). While transferring from RBUF to Microengine the least significant longword from one RBUF element is transferred first followed by the most significant longword into the Microengine transfer registers.

Figure 63. Bit, Byte and Long-Word Organization in One RBUF Element

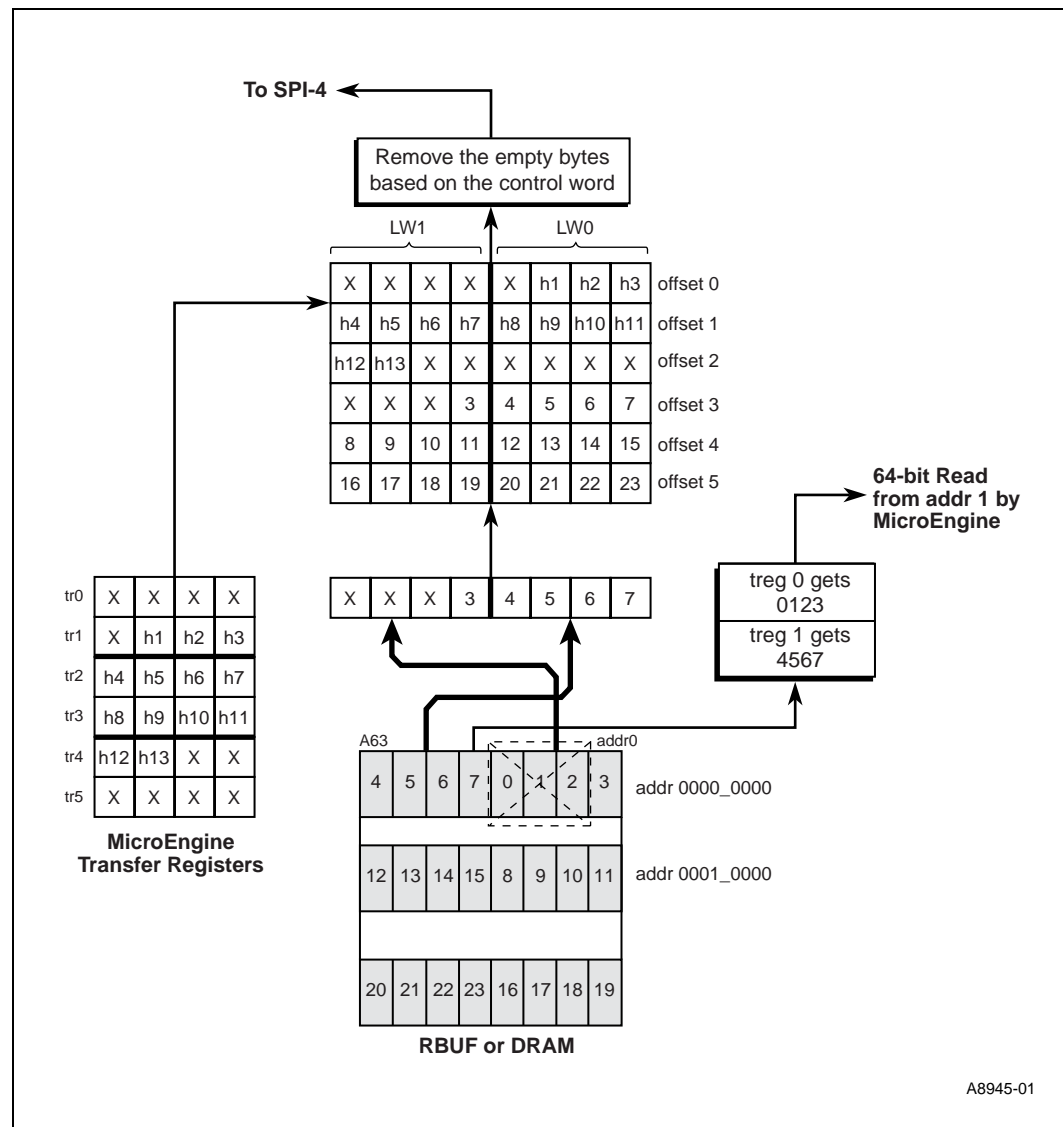




4.5.2.2 Write to TBUF

For writing to TBUF, the header comes from the Microengine and data comes from RBUF or DRAM. Since the Microengine to TBUF header transfer happened in 8-byte chunks, it is possible that the first long word that is inside tr0 may not contain any data if the valid header begins in transfer register tr1. Since data in tr0 goes to LW1 location at offset 0 and data in tr2 will go to LW0 location at offset 0, there will be some white spaces or invalid bytes at the beginning of the header at offset 0. These invalid bytes are removed by the aligner on the way out of TBUF based on the control word for this TBUF element. The data from tr2, tr3...tr6 is placed in TBUF as shown in Figure 64 in big-endian order.

Figure 64. Write to TBUF

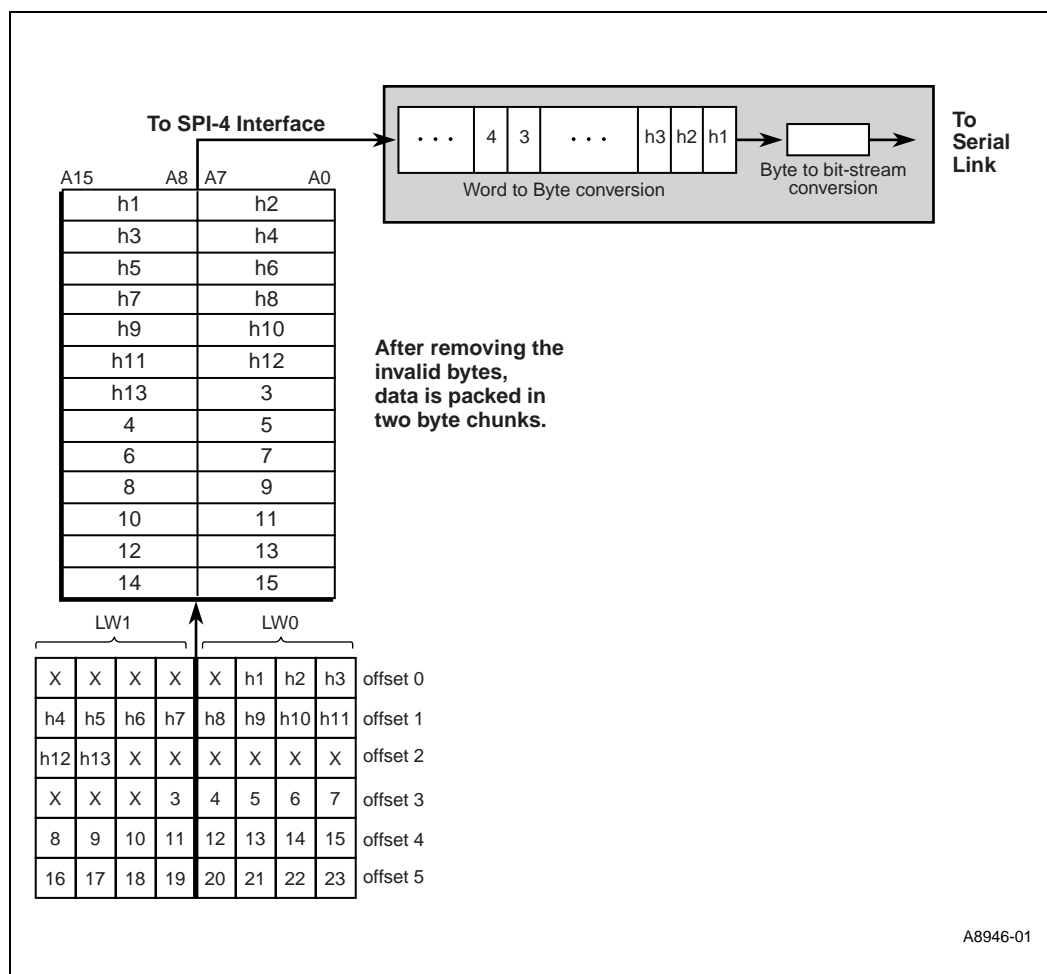


Since data in RBUF or DRAM is arranged in LWBE order, it is swapped on the way into the TBUF to make it truly big endian as shown in Figure 64. Again the white space at the beginning of payload that starts at offset 3 and at the end of header at offset 2 will be removed by the aligner on the way out of TBUF by the aligner.

4.5.2.3 TBUF to SPI-4 Transfer

Figure 65 shows how the MSF interface removes invalid bytes from TBUF data and transfers them onto the SPI-4 interface in 16-bit (2-byte) chunks.

Figure 65. MSF Interface



This section describes Rambus® DRAM operation.

5.1 Overview

The IXP2800 Network Processor has controllers for three Rambus DRAM (RDRAM) channels. Either one, two, or three channels can be enabled. When more than one channel is enabled, the channels are interleaved (also known as striping) on 128-byte boundaries to provide balanced access to all populated channels. Interleaving is performed in hardware and is transparent to the programmer. The programmer views the DRAM memory space as a contiguous block of memory.

The total address space of 2 GB is supported by the DRAM interface regardless of the number of channels that are enabled. The controllers support 64 Mb, 128 Mb, 256 Mb, 512 Mb, and 1 Gb devices, however because of the interleaving, each of the channels must have the same number, size, and speed RDRAMs populated. Each channel can be populated with up to 32 RDRAMs devices. While each channel must have the same size and speed RDRAMs, it is possible for each individual channel to have different size and speed RDRAMs, as long as the total amount of memory is the same for all the channels.

ECC (Error Correcting Code) is supported. Enabling ECC requires that x18 RDRAMs be used. If ECC is disabled x16 RDRAMs can be used.

The Microengines, Intel XScale® technology, and PCI (external Bus Masters and DMA Channels) have access to the DRAM memory space.

The controllers also automatically perform refresh as well as IO driver calibration to account for variations in operating conditions due to process, temperature, voltage and board layout.

RDRAM Powerdown and nap modes are not supported.

5.2 Size Configuration

Each channel can be populated with anywhere from one-to-four RDRAMs (Short Channel Mode). For supported loading configurations refer to [Table 63](#). The RAM technology used will determine the increment size and maximum memory per channel as shown in [Table 64](#).

Note: One or two channels can be left unpopulated if desired.

Table 63. RDRAM Loading

Bus Interface	Max # of Loads	Trace Length (inches)
Short Channel: 400 and 533 MHz	Four devices per channel.	20 ¹
Long Channel: 400 MHz	2 RIMMs per channel, a maximum of 32 devices in both RIMMs.	20 ¹
Long Channel: 533 MHz	1 RIMM and 1 C-RIMM per channel, a maximum of 16 devices.	20 ¹

1. For termination, the DRAM's should be located as close as possible to the IXP2800 Network Processor.

Table 64. RDRAM Sizes

RDRAM Technology ¹	Increment Size	Maximum per Channel
64/72 Mb	8 MB	256 MB
128/144 Mb	16 MB	512 MB
256/288 Mb	32 MB	1 GB ²
512/576 Mb	64 MB	2 GB ²
NOTES: 1. The two numbers shown for each technology indicate x16 parts and x18 parts. 2. The maximum memory that can be addressed across all channels is 2GB. This limitation is based on the partitioning of the 4GB address space (32-bit addresses). Therefore if all three channels are used, each can be populated up to a maximum of 768MB. Two channels can be populated to a maximum of 1 GB each. A single channel could be populated to a maximum of 1GB		

RDRAMs with 1 x 16 dependent banks, 2 x 16 dependent banks, and 4 independent banks are supported.

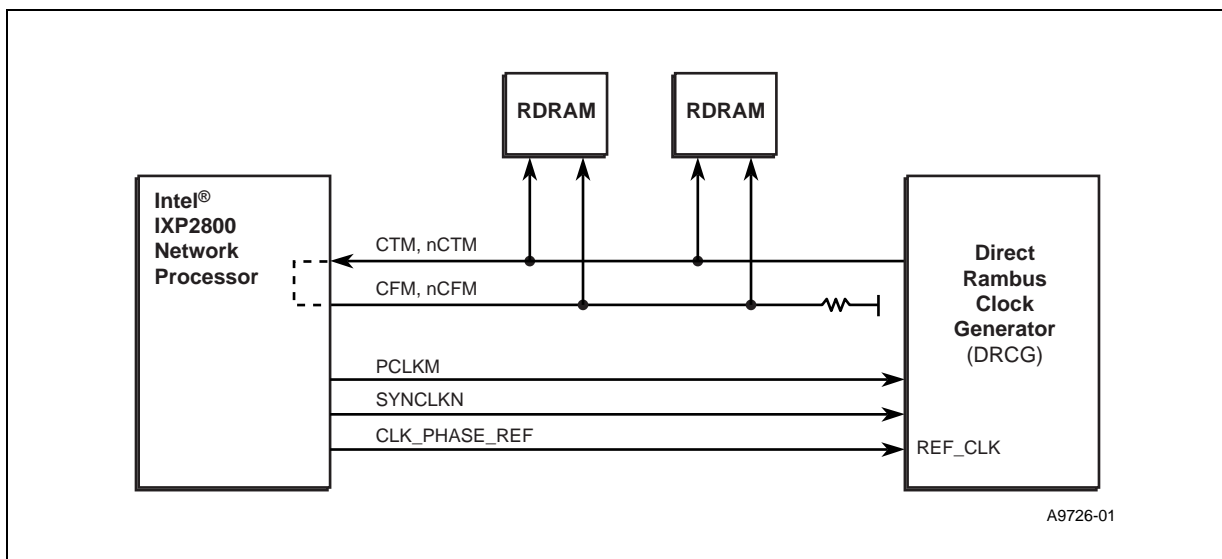


5.3 DRAM Clocking

Figure 66 shows the clock generation for one channel (this description is just for reference, for more details refer to Rambus design literature). The other channels use the same configuration.

Note: Refer to [Section 10](#) for additional information on clocking.

Figure 66. Clock Configuration



The RDRAM Controller receives two clocks, both generated internal to the IXP2800 Network Processor.

The internal clock, is used to control all logic associated with communication with other on-chip Units. This clock is $\frac{1}{2}$ of the Microengine frequency, and is in the range of 500 MHz to 700 MHz.

The other clock called, RMC clock is internally divided by two and brought out on the CLK_PHASE_REF pin, which is then used as the reference clock for the DRCG (see [Figure 67](#) and [Figure 68](#)). The reason for this is our internal RMC clock is derived from the Microengine clock (supported programmable divide range is from 8 to 15 for A stepping, 6 - 15 for B stepping) at a Microengine frequency of 1.4 GHz (the available RMC clock frequencies are 100, and 127 MHz). In the RMC implementation we have a fixed 1:1 clock relationship between the RMC clock and the SYNCLK ($\text{SYNCLK} = \text{Clock-to-Master(CTM)}/4$) therefore, in order to maintain this relationship we provide the clock to the DRCG. CTM is received by the DRAM controller which it drives back out as Clock-from-Master (CFM). Additionally the controller creates PCLKM and SYNCLKN which are also driven to the DRCG.

Figure 67. IXP2800 Clocking for RDRAM at 400 MHz

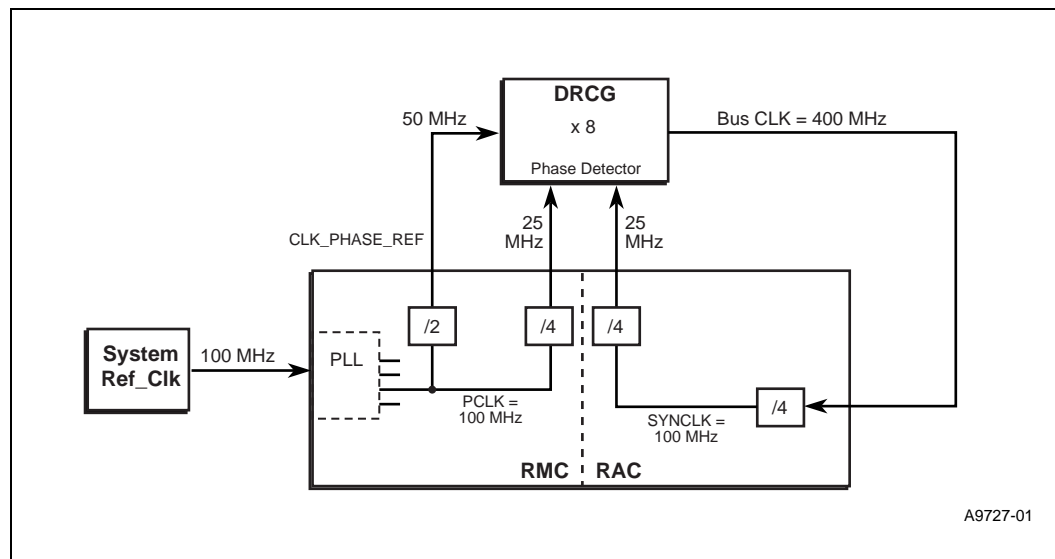
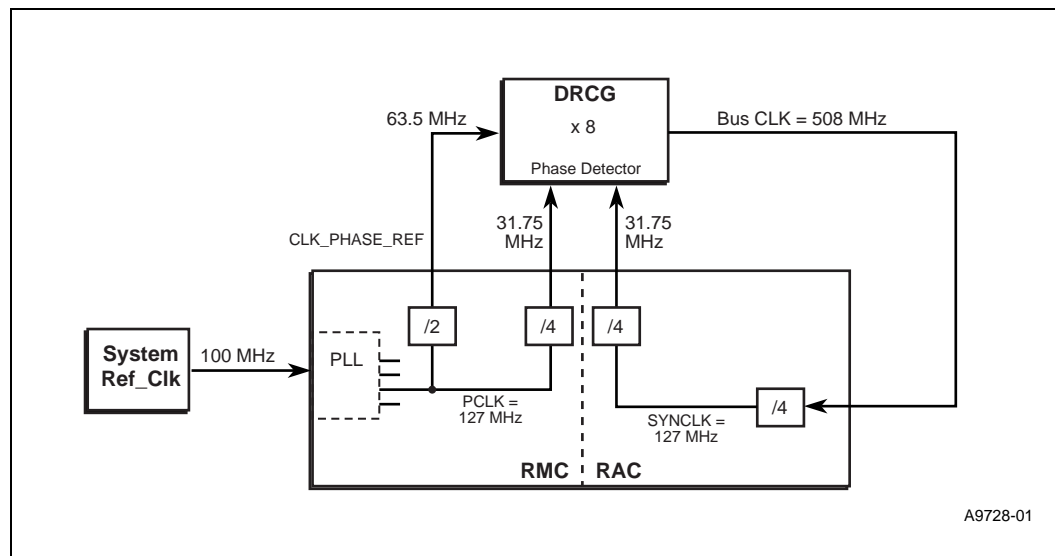


Figure 68. IXP2800 Clocking for RDRAM at 508 MHz



5.4 Bank Policy

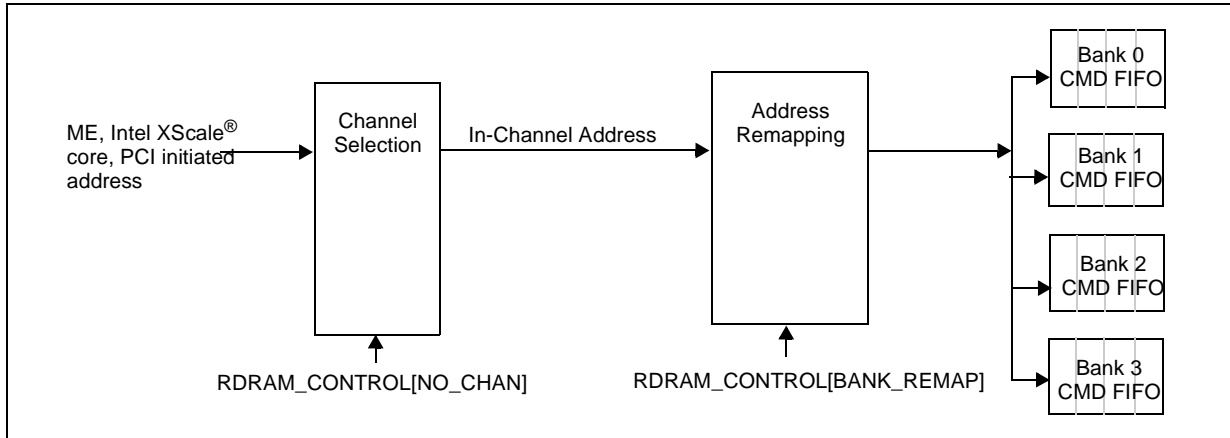
The RDRAM Controller uses a closed bank policy. Banks are activated long enough to do an access and then closed and precharged. They are not left open in anticipation of another access to the same page. This is unlike many CPU applications, where there is a high degree of locality. Since that locality does not exist in the typical applications that the IXP2800 Network Processor uses RDRAM, the bank closed policy is used.



5.5 Interleaving

The RDRAM channels are interleaved on 128 byte boundaries in hardware to improve concurrency and bandwidth utilization. Contiguous addresses are directed to different channels by rearranging the physical address bits in a programmable manner described in [Section 5.5.1](#) through [Section 5.5.3](#) and then remapped as described in [Section 5.5.4](#). [Figure 69](#) shows a block diagram of the flow.

Figure 69. Address Mapping Flow



Note that the mapping of addresses to channels is completely transparent to software. Software deals with physical addresses in RDRAM space; the mapping is done completely by hardware. Note that accessing an address above the amount of RDRAM populated will cause unpredictable results.

5.5.1 Three Channels Active (3-Way Interleave)

When all three channels are active, the interleave scheme selects the channel for each block using modulo-3 reduction (address bits [31:7] are summed as modulo-3, and the remainder is the selected channel number). The algorithm ensures that adjacent blocks are mapped to different channels.

For Rev A, the address within the DRAM is then selected by rearranging the received address, as shown in [Table 65](#). In this case the number of DRAMs on a channel must be either 1, 2, 4, 8, 16, or 32.

For Rev B, the address within the DRAM is selected by adding the received address to the contents of one of the CSRs K0 through K11, or zero, as shown in [Table 66](#). The values to load into K0 through K11 are a function of the amount of memory on the channel, and are specified in the Programmer's Reference Manual. Note that for memory size 32 MB, 64 MB, 128 MB, etc. the specified constants give the same remapping as was done in Rev A.

Table 65. Address Rearrangement for 3-Way Interleave

When these bits of address are all "1"s...	Shift 30:7 right by this many bits	Add this amount to shifted 30:7 (based on amount of memory on the channel) Address within channel is {30:7+table_value}, 6:0}							
		8MB ³	16MB	32MB ³	64MB	128MB ³	256MB	512MB ³	1GB
30:7	26	N/A	N/A	N/A	N/A	N/A	N/A	N/A	8388607
28:7	24	N/A	N/A	N/A	N/A	N/A	2097151	4194303	8388606
26:7	22	N/A	N/A	N/A	524287	1048575	2097150	4194300	8388600
24:7	20	N/A	131071	262143	524286	1048572	2097144	4194288	8388576
22:7	18	65535	131070	262140	524280	1048560	2097120	4194240	8388480
20:7	16	65532	131064	262128	524256	1048512	2097024	4194048	8388096
18:7	14	65520	131040	262080	524160	1048320	2096640	4193280	8386560
16:7	12	65472	130944	261888	523776	1047552	2095104	4190208	8380416
14:7	10	65280	130560	261120	522240	1044480	2088960	4177920	8355840
12:7	8	64512	129024	258048	516096	1032192	2064384	4128768	8257536
10:7	6	61440	122880	245760	491520	983040	1966080	3932160	7864320
8:7	4	49152	98304	196608	393216	786432	1572864	3145728	6291456
None	2	0	0	0	0	0	0	0	0

NOTES:

1. This is a priority encoder; when multiple lines satisfy the condition, the line with the largest number of ones is used.
2. N/A means not applicable.
3. For these cases, the top 3 blocks (each block is 128 bytes) of the logical address space is not accessible. For example if each channel has 8 MB, only (24MB - 384) total bytes are usable. This is an artifact of the remapping method.
4. The numbers in the table are derived as follows.
For the first pair of ones (8:7) value is 3/4 the number of blocks. For each subsequent pair of ones, the value is the previous value, plus another 3/4 the remaining blocks.
 - [8:7]==11 - 3/4 * blocks
 - [10:7]==1111 - (3/4 + 3/16) * blocks
 - [12:7]==111111 - (3/4 + 3/16 + 3/64) * blocks
 - etc.



Table 66. Address Rearrangement for 3-Way Interleave (Continued)(Rev B)

When these bits of address are all "1" ¹	Add the value in this CSR to the address
30:7	K11
28:7	K10
26:7	K9
24:7	K8
22:7	K7
20:7	K6
18:7	K5
16:7	K4
14:7	K3
12:7	K2
10:7	K1
8:7	K0
None	Value 0 added.
NOTES: 1. This is a priority encoder; when multiple lines satisfy the condition, the line with the largest number of ones is used. 2. N/A means not applicable.	

5.5.2 Two Channels Active (2-Way Interleave)

It is possible to have only two channels populated for system cost and area savings. If only two channels are desired, than channels 0 and 1 should be populated and channel 2 should be left empty. In the Two Channel Mode, the address interleaving is designed with the goal of spreading adjacent accesses across the 2 channels.

When two channels are active, address bit 7 is used as the channel select. Addresses that have address 7 equal to 0 are mapped to channel 0 while those with address 7 equal to 1 are mapped to channel 1. The address within the channel is {[31:8], [6:0]}.

5.5.3 One Channel Active (No Interleave)

When only one channel is active, all accesses go to that channel. In this case it is possible for an access to split across two DRAM banks (which could be in different RDRAMs).

5.5.4 Interleaving Across RDRAMs and Banks

In addition to interleaving across the different RDRAM channels, addresses are also interleaved across RDRAM chips and internal banks. This improves utilization since certain operations to different banks can be performed concurrently. The interleaving is done based on rearranging the remapped address derived from [Section 5.5.1](#), [Section 5.5.2](#), and [Section 5.5.3](#) as a function of the memory size as shown in [Table 67](#). The two MSBs of the rearranged address are used to select which Bank Command FIFO the command is place in. The rearranged address is also partitioned to choose RDRAM chip, bank within RDRAM, and page within bank.

Table 67. Address Bank Interleaving

Memory Size on Channel (MB) ³	Remapped Address Based on RDRAM_Control[Bank_Remap]			
	00	01	10	11
8	7:14, 22:15	9:14, 7:8, 22:15	11:14, 7:10, 22:15	13:14, 7:12, 22:15
16	7:14, 23:15	9:14, 7:8, 23:15	11:14, 7:10, 23:15	13:14, 7:12, 23:15
32	7:14, 24:15	9:14, 7:8, 24:15	11:14, 7:10, 24:15	13:14, 7:12, 24:15
64	7:14, 25:15	9:14, 7:8, 25:15	11:14, 7:10, 25:15	13:14, 7:12, 25:15
128	7:14, 26:15	9:14, 7:8, 26:15	11:14, 7:10, 26:15	13:14, 7:12, 26:15
256	7:14, 27:15	9:14, 7:8, 27:15	11:14, 7:10, 27:15	13:14, 7:12, 27:15
512	7:14, 28:15	9:14, 7:8, 28:15	11:14, 7:10, 28:15	13:14, 7:12, 28:15
1024	7:14, 29:15	9:14, 7:8, 29:15	11:14, 7:10, 29:15	13:14, 7:12, 29:15
Bits used to select Bank Command FIFO	7:8	9:10	11:12	13:14
NOTES: 1. Table shows device/bank sorting of the channel remapped block address, which is in address 31:7. LSBs of the address are always 6:0 (byte within the block), which are not remapped 2. Unused MSBs of address have value of 0. 3. Size is programmed in RDRAM_Control[Size].				

5.6 Parity and ECC

DRAM can be optionally protected by byte parity or by an 8-bit error detecting and correcting code (ECC). RDRAMn_Control[ECC] for each channel selects whether or not that channel should use Parity, ECC, or no protection. When parity or ECC is enabled x18 RDRAMs must be used with the extra bits connected to the dqa[8] and dqb[8] signals. Eight bits of ECC code cover eight bytes of data (aligned to an 8-byte boundary).

5.6.1 Parity and ECC Disabled

- On reads, the data is delivered to the originator of the read; no error is possible.
- Partial writes (writes of less than eight bytes) are done as masked writes.



5.6.2 Parity Enabled

On writes, odd byte parity is computed for each byte and written into the corresponding parity bit. Partial writes (writes of less than eight bytes) are done as masked writes.

On reads, odd byte parity is computed on each byte of data and compared to the corresponding parity bit. If there is an error RDRAMn_Error_Status_1[Uncorr_Err] bit is set, which can interrupt the Intel XScale® core if enabled. The Data Error signal will be asserted when the read data is delivered on D_Push_Data.

The address of the error, along with other information, is logged in RDRAMn_Error_Status_1[ADDR] and RDRAMn_Error_Status_2. Once the error bit is set those registers are locked. That is, the information relating to subsequent errors will not be loaded until the error status bit is cleared by the Intel XScale® core write.

5.6.3 ECC Enabled

On writes, eight ECC check bits are computed based on 64 bits of data, and are written into the check bits. Partial writes (writes of less than eight bytes) will cause the channel controller to do a read-modify-write. Any single bit error detected during the read portion will be corrected prior to merging with the write data. An uncorrectable error detected during the read will not modify the data. Either type of error will set the appropriate error status bit as described during the read case (next paragraph).

On reads, the correct value for the check bits is computed from the data and is compared to the ECC check bits. If there is no error, data is delivered to the originator of the read as it came from the RDRAMs. If there is a single bit error it is corrected before being delivered (the correction is done automatically, the reader is given the correct data). The error is also logged by setting the RDRAMn_Error_Status_1[Corr_Err] bit, which can interrupt the Intel XScale® core if enabled.

If there is an uncorrectable error the RDRAMn_Error_Status_1[Uncorr_Err] bit is set, which can interrupt the Intel XScale® core if enabled. The Data Error signal will be asserted when the read data is delivered on D Push Data, unless the token Ignore Data Error was asserted in the command. In that case the RDRAM controller will not assert Data Error and will not assert a Signal (it will use 0xF, which is a null signal, in place of the requested signal number).

In both correctable and uncorrectable cases, the address of the error, along with other information, is logged in RDRAMn_Error_Status_1[ADDR] and RDRAMn_Error_Status_2. Once either of the error bits is set those registers are locked. That is, the information relating to subsequent errors will not be loaded until both error status bits are clear. That does not prevent the correction of single bit errors, only the logging.

Note: When a single bit error is corrected, the corrected data is not written back into memory (scrubbed) by hardware; this can be done by software if desired since all of the information pertaining to the error is logged.

To avoid the detection of false ECC errors, the RDRAM ECC mode must be initialized using the procedure described below:

- Ensure that parity/ECC is not enabled: program DRAM_CTRL[15:14] = 00
- Write all zeros to all the memory locations. By default this will initialize the memory with odd parity and in this case (data all 0), it coincides with ECC and this does not require any read modify writes because ECC is not enabled.
- Ensure that all of the writes are completed prior to enabling ECC. This is done by performing a read operation to 1000 locations.
- Enable ECC mode: program DRAM_CTRL[15:14] accordingly.

5.6.4 ECC Calculation and Syndrome

The ECC check bits are calculated by forming parity checks on groups of data bits. The check bits are stored in memory during writes via the dqa[8] and dqb[8] signals. Note that memory initialization code must put good ECC into all of memory by writing each location before it can be read. Writing any arbitrary data into memory, for example 0, will accomplish this. This will take several ms per MB of memory.

On reads, the expected code is calculated from the data, and then compared to (XORed) the ECC which was read. The result of the comparison is called the syndrome. If the syndrome is equal to zero, then there was no error. There are eight syndromes that are calculated based on the read data and its corresponding ECC bit. When ECC is enabled, upon detecting a single bit error, the syndrome is used to determine which bit needs to be flipped to correct the error.

5.7 Timing Configuration

Table 68 shows the example timing settings for RDRAMs of various speeds. The parameters are programmed in the RDRAM_Config CSRs (refer to the PRM for register descriptions).

Table 68. RDRAM Timing Parameter Settings

Parameter Name	-40-800	-45-800	-50-800	-45-711	-50-711	-45-600	-53-600
CfgTrcd	7	9	11	7	9	5	7
CfgTrasSyn	5	5	6	5	5	4	5
CfgTrp	8	8	10	8	8	6	8
CfgToffpSyn	4	4	4	4	4	4	4
CfgTrasrefSyn	5	5	6	5	5	4	5
CfgTprefSyn	2	2	2	2	2	2	2



5.8 Microengine Signals

Upon completion of a read or write, the RDRAM controller can signal a Microengine context, when enabled. It does so using the `sig_done` token; see [Example 26](#).

Example 26. RDRAM Controller Signaling a Microengine Context

```
dram [read,$xfer6,addr_a,0,1], sig_done_4
dram [read,$xfer7,addr_b,0,1], sig_done_6
ctx_arb[4, 5, 6, 7]
```

Because the RDRAM address space is interleaved, consecutive accesses can go to different RDRAM channels. There is no ordering guaranteed among different channels, so, a separate signal is needed for each.

In addition, because accesses start at any address, and can specify up to 16 64-bit words (128 bytes), they can also split across two channels (refer to [Section 5.5](#)). The `ctx_arb` instruction must set two `Wakeup_Events` (an odd/even pair) per access. The RDRAM controllers coordinate as follows:

- If the access split across two channels, the channel handling the low part of the split delivers the even numbered Event Signal, and the channel handling the upper part of the split delivers the odd numbered Event Signal.
- If the access does not split, the channel delivers both Event Signals (by coordinating with the D Push or D Pull arbiter for read and writes respectively).
- In all cases the channel delivers the Event Signal with the last data Push or Pull of a burst.

Using the above rules, the Microengine will be put into the Ready State (ready to resume executing) only when all accesses have completed.

5.9 Serial Port

The RDRAM chips are configured through a serial port, which consists of signals `D_SIO`, `D_CMD`, `D_SCK`. Access to the serial port is via the `RDRAM_Serial_Command` and `RDRAM_Serial_Data` CSRs (refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the register descriptions).

All serial commands are initiated by a write to `RDRAM_Serial_Command`. Because the serial port is slow, `RDRAM_Serial_Command` has a Busy bit, which indicates that a serial port command is in progress. Software must test this bit before initiating a command. This insures that software will not lose a command, while eliminating the need for a hardware FIFO for serial commands.

Serial writes are done by the following steps:

1. Read `RDRAM_Serial_Command`; test Busy bit until its a 0.
2. Write `RDRAM_Serial_Data`.
3. Write `RDRAM_Serial_Command` to start the write.

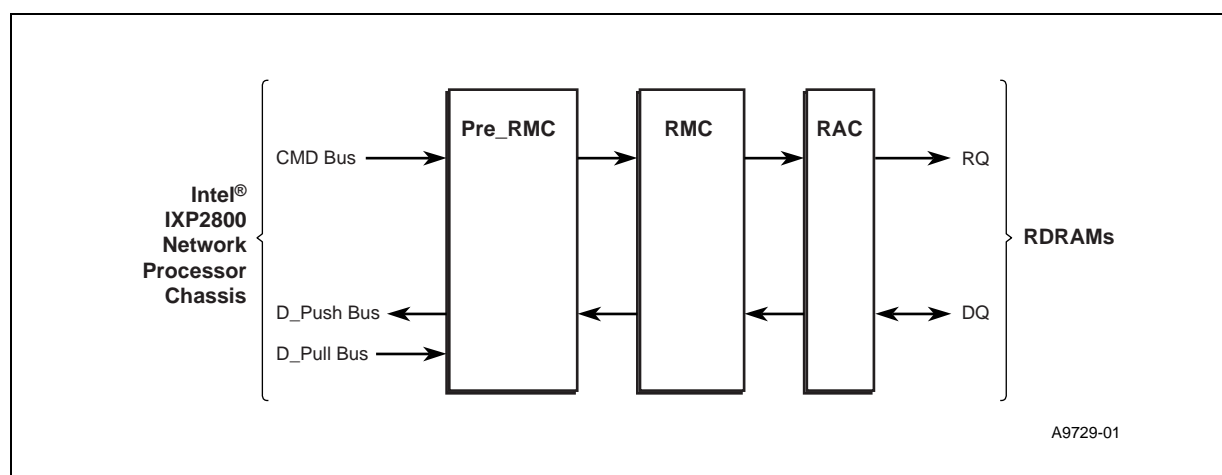
Serial reads are done by the following steps:

1. Read RDRAM_Serial_Command; test Busy bit until its a 0.
2. Write RDRAM_Serial_Command to start the read.
3. Read RDRAM_Serial_Command; test Busy bit until its a 0.
4. Read RDRAM_Serial_Data to collect the serial read data.

5.10 RDRAM Controller Block Diagram

The RDRAM controller consists of three pieces. Figure 70 is a simplified block diagram.

Figure 70. RDRAM Controller Block Diagram



Pre_RMC—has the queues for commands, data (both in and out) and interface to internal busses. It checks incoming commands and addresses to determine if they are targeted to the channel, and if so enqueues them [note that if a command splits across two channels, the channel must enqueue the portion of the command that it owns]. It sorts the enqueued commands to RDRAM banks, selects which command to execute based on policy to get good bank utilization, and then hands off that command to RMC. It also arbitrates for refresh and calibration, which it requests RMC to perform. Pre_RMC also contains the ECC logic, and the CSRs that set size, timing, ECC, etc.

RMC—controller that handles the pin protocol. It controls all timing dependencies, pin turnaround, RAS-CAS, RAS-RAS, etc., including bank interactions. RMC handles all commands in the order that it receives them. RMC is based on the Rambus RMC.

RAC—high speed parallel to serial and parallel to serial interface. This is a hard macro which contains the I/O pads and drivers, DLL, and associated pin interface logic.

Following is a brief explanation of command operation.

Pre_RMC enqueues commands and sends them to RMC. It is responsible for initiating Pull operations to get Microengine/RBUF/Intel XScale® core/PCI data into the Pull_Data FIFO. A write is not eligible to go to RMC until Pre_RMC has all the data in the Pull Data FIFO.

Pre_RMC provides the Full signal to the Command Arbiter to inform it stop allowing RDRAM commands.



5.10.1 Commands

When a valid command is placed on the command bus, the control logic checks to see if the address matches the channel's address range, based on interleaving as described in [Section 5.5](#). The command, address, length, etc. are enqueued into the command Inlet FIFO.

If the command Inlet FIFO becomes full, the channel sends a signal to the command arbiter which will prevent it from sending further DRAM commands. The full signal must be asserted while there is still enough room in the FIFOs to hold the worst case number of in-flight commands.

5.10.2 DRAM Write

When a write (or RBUF_RD, which does a DRAM write) command is at the head of the Command Inlet FIFO, it is moved to the proper Bank CMD FIFO, and the Pull_ID is sent to the Pull arbiter. This can only be done if there is room for the command in the Bank's CMD FIFO and for the pull data in the Bank's Pull Data FIFO (which must take into account all pull data in flight). If there is not room in the Bank's CMD FIFO, or the Bank's Pull Data FIFO, the write command will wait at the head of the Command Inlet FIFO. When the Pull_ID is sent to the Pull Arbiter, the Bank number is put into the PP (Pull in Progress) FIFO; this allows the channel to sort the Pull Data into the proper Bank Pull Data FIFO when it arrives.

The source of the Pull Data can be either RBUF, PCI, Microengine, or the Intel XScale® core, and is specified in the Pull_ID. When the source is RBUF or PCI, data will be supplied to the Pull Data FIFO 64 bits per cycle. When the source is Microengine or the Intel XScale® core, data will be supplied 32 bits per cycle, justified to the low 32 bits of Pull Data. It is up Pull Arbiter to merge and pack data as required. In addition, the data must be aligned according to the start address, which has longword resolution; this is done in Pre_RMC.

The Length field of the command at the head of the Bank CMD FIFO is compared to the number of 64-bit words in the Bank Pull_Data FIFO. When the number of 64-bit words in Pull_Data FIFO is greater or equal to the length, the write arbitrates for the RMC. When it wins arbitration it sends the address and command to RMC. RMC will request the write data from Pull_Data FIFO at the proper time to send it to the RDRAMs.

Note: The Microengine is signaled when the last data is pulled.

5.10.2.1 Masked Write

Masked writes (write of less than 8-bytes) are done as either Read-Modify-Writes when ECC is enabled, or as Rambus masked writes (using COLM packets), when ECC is not enabled. In both cases the masked write will modify 7 or fewer bytes; this is because the command bus limits a masked write to a ref_count of one.

If a RMW is used, no commands from that Bank's CMD FIFO will be started in between the read and the write; other Bank commands can be done during that time.



5.10.3 DRAM Read

When a read (or TBUF_WR, which does a DRAM read) command is at the head of the Command Inlet FIFO, it is moved to the proper Bank CMD FIFO if there is room. If there is not room in the Bank's CMD FIFO, the read command will wait at the head of the Command Inlet FIFO.

When a read command is at the head of the Bank CMD FIFO, and there is room for the read data in the Push Data FIFO (including all reads in flight at the RDRAM), it will arbitrate for RMC. When it wins arbitration it sends the address and command to RMC. The Push_ID is put into the RP FIFO (Read in Progress), to coordinate it with read data from RMC.

When read data is returned from RMC it is placed into the Push_Data FIFO. Each Push_Data is sent to the Push Arbiter with a Push_ID; the RDRAM controller increments the Push_ID for each data phase. If Push Arbiter asserts the full signal, Push Data is stopped and held in the Push Data skid FIFO. The Push Data is sent to the read destination under control of the Push Arbiter.

The destination of the Push Data can be either Intel XScale® core, PCI, TBUF or Microengine, and is specified in the Push_ID. When the destination is TBUF or PCI, data will be taken 64 bits per cycle. When the destination is Microengine or the Intel XScale® core, data will be taken 32 bits per cycle. The Push Arbiter justifies the data to the low 32 bits of Push Data. Note that the Microengine is signaled when the last data is pushed.

5.10.4 CSR Write

When a CSR write command is at the head of the Command Inlet FIFO, it is moved to the CSR CMD Register, and the Pull_ID is sent to the Pull arbiter. This can only be done if the CSR CMD Register is not currently occupied. If it is, the CSR write command will wait at the head of the Command Inlet FIFO. When the Pull_ID is sent to the Pull Arbiter, a tag put into the PP FIFO (Pull in Progress); this allows the channel to identify the Pull Data as CSR data when it arrives.

When the CSR pull data arrives it is put into the addressed CSR, and the CSR CMD Register is marked as empty.

5.10.5 CSR Read

When a CSR read command is at the head of the Command Inlet FIFO, it is moved to the CSR CMD Register. This can only be done if the CSR CMD Register is not currently occupied. If it is, the CSR read command will wait at the head of the Command Inlet FIFO.

On the first available cycle in which RDRAM data from RMC is not being put into the Push Data FIFO, the CSR data will be put into the Push Data FIFO. If it's convenient to guarantee a slot by putting a bubble on the RMC input, that's OK.



5.10.6 Arbitration

The channel needs to arbitrate among several different operations at RMC. Arbitration rules are given here for those cases. From highest to lowest priority:

- Refresh RDRAM
- Current calibrate RDRAM
- Bank operations. When there are multiple bank operations ready the rules are: (1) round robin among banks to avoid bank collisions, (2) skip a bank to avoid DQ bus turnarounds. No bank can be skipped more than twice.

Commands are given to RMC in the order in which they will be executed.

5.10.7 Reference Ordering

Table 69 lists the ordering of reads and writes to the same address for DRAM. The definition of first and second is defined by the time the command is valid on the command bus.

Table 69. Ordering of Reads and Writes to the Same Address for DRAM

First Access	Second Access	Ordering Rules
Read	Read	None. If there are no side effects on reads both readers will get the same data.
Read	Write	Reader must get the pre-modified data. This is <i>not</i> enforced in hardware. The write instruction must not be executed until after the ME receives the signal of read completion (i.e. program must use <code>sig_done</code> on the read).
Write	Read	Reader must get the post-modified data. This is <i>not</i> enforced in hardware. The read instruction must not be executed until after the ME receives the signal of write completion (i.e. program must use <code>sig_done</code> token on the write instruction and wait for the signal before executing the read instruction).
Write	Write	The hardware guarantees the writes will complete in the order they are issued.

5.11 DRAM Push/Pull Arbiter

The DRAM Push/Pull Arbiter contains the push and pull arbiters for the D-Cluster (DRAM Cluster). Both the PUSH and PULL data buses have multiple masters and multiple targets. The DRAM Push/Pull Arbiter determines which master gets to drive the data bus for a given transaction and to make sure the data is delivered correctly.

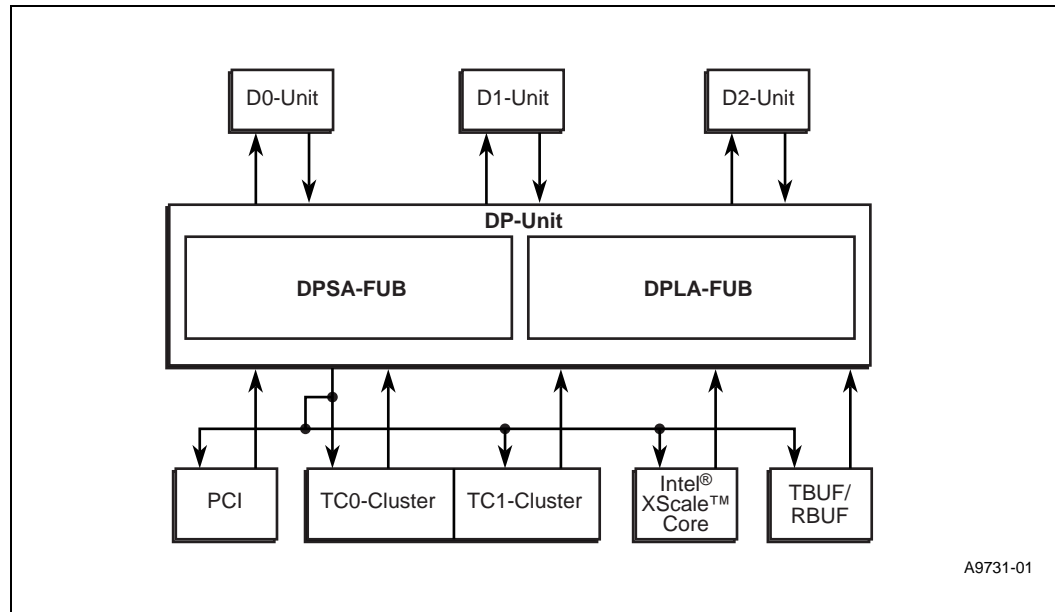
This unit has the following features:

- Up to three DX Unit (DRAM Unit) masters.
- 64-bit wide push and pull data buses.
- Round-robin arbitration scheme.
- Peak delivery of 64-bits per cycle.
- Supports third-party data transfers the Microengine's can command data movements between the MSF (Media) and either the DX Units or the CR Units.

- Supports chaining for burst DRAM push operations to tell the arbiter to grant consecutive push requests.
- Supports data error bit handling and delivery.

Figure 71 shows the functional blocks for the DRAM Push/Pull Arbiter.

Figure 71. DRAM Push/Pull Arbiter Functional Blocks



5.11.1 Arbiter Push/Pull Operation

Within the arbiter there are two functional units: the push arbiter and the pull arbiter. Push and pull always refer to the way data is flowing from the bus master, i.e., a Microengine makes a read request, the DRAM channel does the read, and then “pushes” the data back to the Microengine.

For a push transaction, a push master will drive command and data to the DRAM push arbiter (DPSA) and into a dedicated request FIFO. When that command is at the head of the FIFO and it is either the requesting unit’s turn to go based on the round-robin arbitration policy, or there are no other requesters, the arbiter will “grant” the request. This grant means that the arbiter will deliver the push data to the correct target with all the correct handshakes and retire the request. (Note that a data transaction is always eight bytes.)

The DRAM pull arbiter (DPLA) is slightly different because it functions on bursts of data transactions instead of single transactions. For a pull transaction, a pull master will drive a command to the pull arbiter and into a dedicated request FIFO. When the command gets to the head of the FIFO, it will be evaluated as was done for the push arbiter. The difference is that each command may reference bursts of data movements (always in multiples of 8 bytes). The arbiter will grant the command, and keep it granted until it increments through all of the data movements required by the command. As the data is read from its source, the command is modified to address the next data address, and a handshake to the requesting unit is driven when the data is valid.



5.11.2 DRAM Push Arbiter Description

The general data flow for a push operation is as shown in [Table 70](#). [Figure 72](#) shows the DRAM Push Arbiter functional blocks.

Table 70. DRAM Push Arbiter Operation

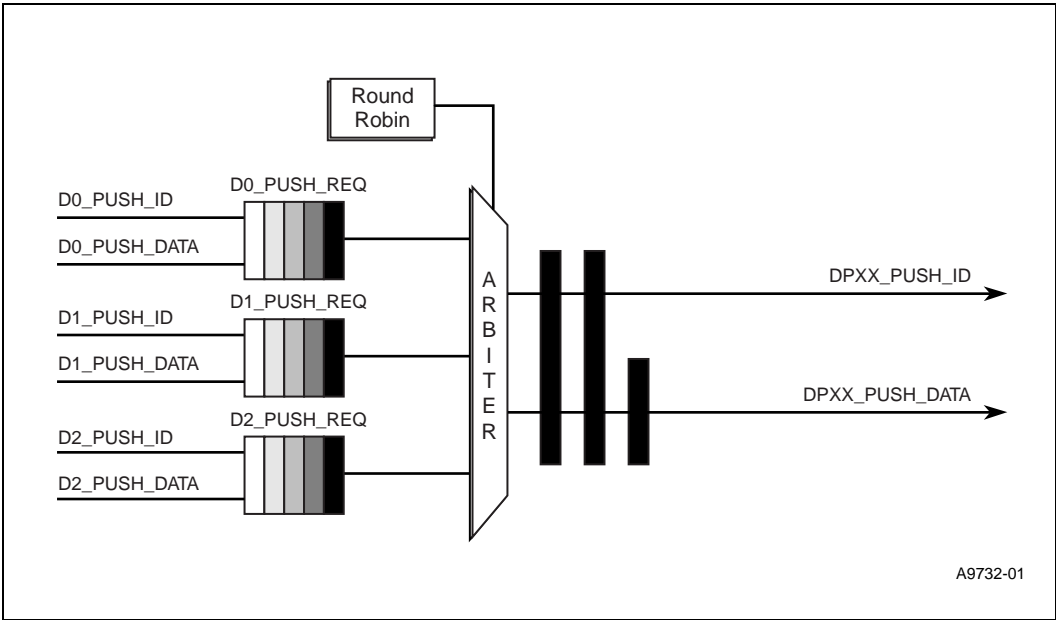
Push Bus Master/Requestor	Data Source	Data Destination
IXP2800 Network Processor		
D0 Unit D1 Unit D2 Unit	Current Master	TC0 Cluster (ME0–7) TC1 Cluster (ME10–17) Intel XScale® core PCI Unit MS Unit

The push arbiter takes push requests from any requestors. Each requestor has a dedicated request FIFO. A request comes in the form of a PUSH_ID, and is accompanied by the data to be pushed, a data error bit, and a chain bit. All of this information is enqueued in the correct FIFO for each request, that is, for each eight bytes of data. The push arbiter must drive a full signal to the requestor if the FIFO reaches a predefined “full” level to apply back pressure and stop requests from coming. The FIFO is 64 entries deep and goes full at 40 entries. The long skid allows for burst reads in flight to finish before stalling the DRAM controller. If the FIFO is not full, the push arbiter can enqueue a new request from each requestor on every cycle.

The push arbiter monitors the heads of each FIFO, and does a round robin arbitration between any available requestors. If the chain bit is asserted, it indicates that once the head request of a queue is granted, the arbiter should continue to grant that queue until the chain bit de-asserts. It is expected that the requestor will assert the chain bit for no longer than a full burst length. The push arbiter must also take special notice of requests destined for the receive buffer in the MSF (Media Switch Fabric). Finally, the push arbiter must manage the delivery of data at different rates depending on how wide the bus is going into a given target. The Microengines, PCI, and the Intel XScale® core all have 32-bit data buses. For these targets, the push arbiter takes 2 clock cycles to deliver 64-bits of data by first delivering bits 31:0 in the first cycle, and then putting bits 63:32 onto the low 32-bits of the PUSH_DATA in the second cycle.



Figure 72. DRAM Push Arbiter Functional Blocks



The DRM Push Arbiter boundary conditions are:

- Make sure each of the push_request queues assert the full signal and back pressure the requesting unit.
- Maintain 100% bus utilization, i.e., no holes.

5.12 DRAM Pull Arbiter Description

The general data flow for a push operation is as shown in Table 71. Figure 73 shows the DRAM Pull Arbiter functional blocks.

Table 71. DPLA Description

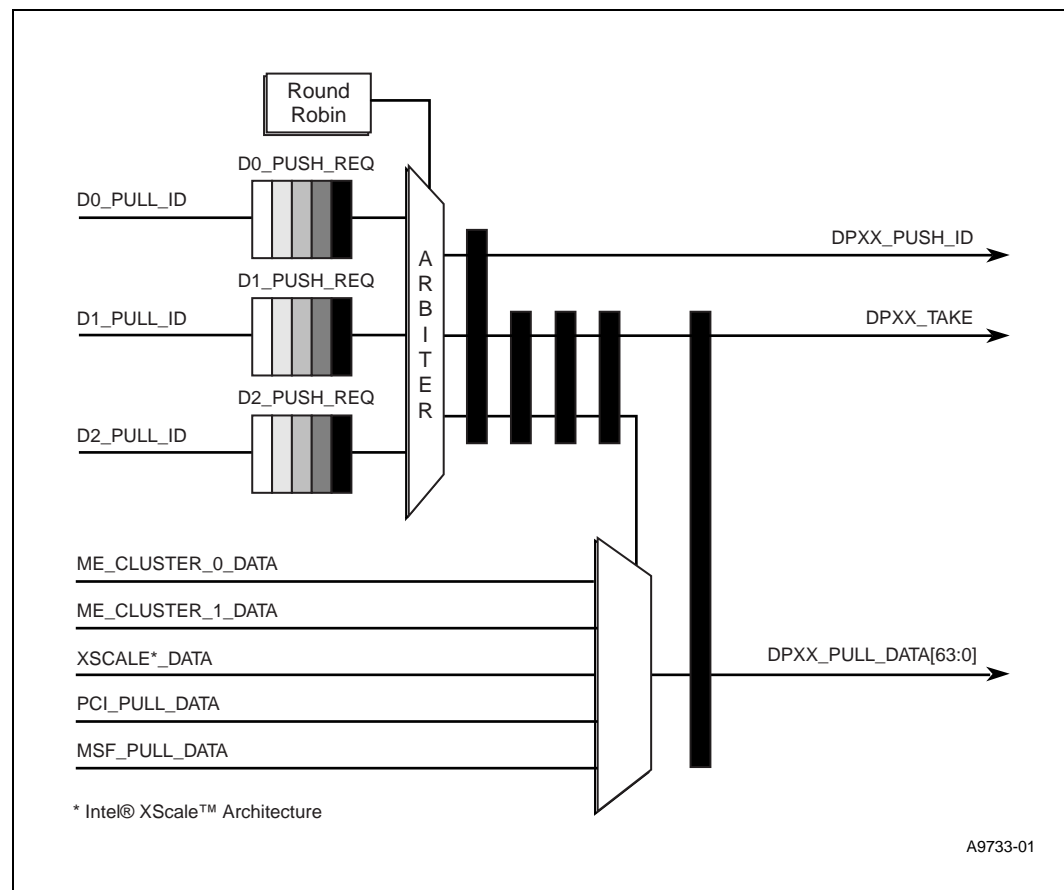
Pull Bus Master/Requestor	Data Source	Data Destination
IXP2800 Network Processor		
D0 Unit D1 Unit D2 Unit	TC0 Cluster (ME0–7) TC1 Cluster (ME8–15) Intel XScale® core PCI Unit MS Unit	Current Master

The pull arbiter is very similar to the push arbiter, except that it gathers the data from a data source ID and delivers it to the requesting unit where it is written to DRAM memory.

When a requestor gets a pull command on the CMD_BUS, the requestor sends the command to the pull arbiter. This is enqueued into a requestor-dedicated FIFO. The pull request FIFOs are much smaller than the push request FIFOs because pull requests can request up to 128 bytes of data. It is eight entries deep and asserts full when it has six entries to account for in-flight requests.

The pull arbiter monitors the heads of each of the three FIFOs. A round robin arbitration scheme is applied to all valid requests. When a request is granted, the request is completed no matter how many data transfers are required. Therefore, one request can take as many as 16–32 DRAMcycles. The push data bus can only use 32-bits when delivering data to the Microengine's, PCI, and the Intel XScale® core. For these data sources, it takes two cycles to pull every eight bytes requested; otherwise, it takes only one cycle per eight bytes. Note that on four byte cycles, data is delivered as pulled.

Figure 73. DRAM Pull Arbiter Functional Blocks







SRAM Interface

6

6.1 Overview

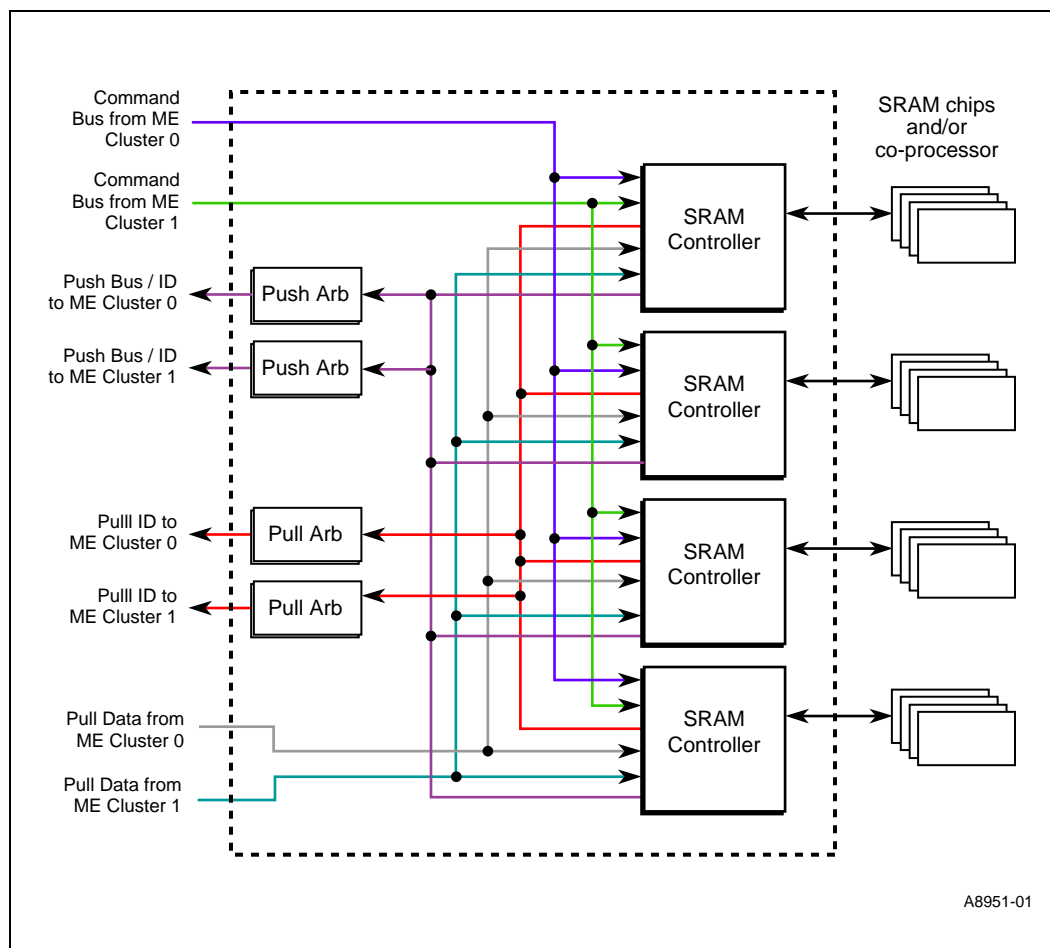
The IXP2800 Network Processor contains four independent SRAM controllers. SRAM controllers support pipelined QDR* synchronous static RAM (SRAM) and a coprocessor that adheres to QDR signaling. Any or all controllers can be left unpopulated if the application does not need to use them.

Reads and writes to SRAM are generated by MicroEngines (ME), the Intel XScale® core, and PCI Bus masters. They are connected to the controllers through Command Buses and Push and Pull Buses. Each of the SRAM controllers takes commands from the command bus and enqueues them. The commands are de-queued, according to priority, and successive access to the SRAMs are performed.

Each SRAM controller receives commands using two Command Buses, one of which may be tied off inactive, depending on the chip implementation. The SRAM Controller can enqueue a command from each Command Bus in each cycle. Data movement between the SRAM controllers and the MEs is through the S-Push bus and S-Pull bus.

The overall structure of the SRAM controllers is shown in [Figure 74](#).

Figure 74. SRAM Controller/Chassis Block Diagram



6.2 SRAM Interface Configurations

Memory is logically four bytes (one longword) wide while physically the data pins are two bytes wide and double-clocked. Byte parity is supported. Each of the four bytes has a parity bit, which is written when the byte is written and checked when the longword is read. There are byte enables that select which bytes to write for lengths of less than a longword.

The QDR controller implements a big-endian ordering scheme at the interface pins. For write operations bytes 0/1, (data bits [31:16]), and associated parity and byte enables are written on the rising edge of K clock while bytes 2/3, (data bits [15:0]), and associated parity and byte enables are written on the rising edge of K_n clock. For read operations bytes 0/1, (data bits [31:16]), and associated parity and byte enables are captured on the rising edge of CIN0 clock while bytes 2/3, (data bits [15:0]), and associated parity and byte enables are captured on the rising edge of CIN_n clock.



In general, QDR and QDR II burst of two SRAMs is supported at speeds up to 250 MHz. As other (larger) QDR SRAMs are introduced, they will also be supported.

The SRAM controller can also be configured to interface to an external coprocessor that adheres to the QDR or QDR II electrical and functional specification.

6.3 SRAM Interface Configurations

This section describes SRAM interface information.

6.3.1 Internal Interface

Each SRAM channel receives commands through the command bus mechanism and transfers data to and from the MEs, the Intel XScale® core, and PCI using SRAM push and SRAM pull buses.

6.3.2 Number of Channels

The IXP2800 supports four channels.

6.3.3 Coprocessor and/or SRAMs Attached to a Channel

Each channel will support the attachment of QDR SRAMs, a co-processor, or both depending on the module level signal integrity and loading.

6.4 SRAM Controller Configurations

There are enough address pins (24) to support up to 64 MB of SRAM. The SRAM controllers can directly generate multiple port enables (up to five pairs) to allow for depth expansion. Two pairs of pins are dedicated for port enables. Smaller RAMs use fewer address signals than the number provided to accommodate the largest RAMs, so some address pins (23:18) are configurable as either address or port enable based on CSR SRAM_Control[Port_Control] as shown in [Table 72](#).

Note: All of the SRAMs on a given channel must be the same size.

Note: [Table 72](#) shows the capability of the logic—1, 2, or 4 loads will be supported, and the table reflects that information, but is subject to change.

Table 72. SRAM Controller Configurations

SRAM Configuration	SRAM Size	Addresses Needed to Index SRAM	Addresses Used as Port Enables	Total Number of Port Select Pairs Available
512K x 18	1 MB	17:0	23:22, 21:20	4
1M x 18	2 MB	18:0	23:22, 21:20	4
2M x 18	4 MB	19:0	23:22, 21:20	4
4M x 18	8 MB	20:0	23:22	3



Table 72. SRAM Controller Configurations (Continued)

SRAM Configuration	SRAM Size	Addresses Needed to Index SRAM	Addresses Used as Port Enables	Total Number of Port Select Pairs Available
8M x 18	16 MB	21:0	23:22	3
16M x 18	32 MB	22:0	None	2
32M x 18	64 MB	23:0	None	1

Each channel can be expanded in depth according to the number of port enables available. If external decoding is used, then the number of SRAMs is not limited by the number of port enables generated by the SRAM controller.

Note: External decoding may require external pipeline registers to account for the decode time, depending on the desired frequency.

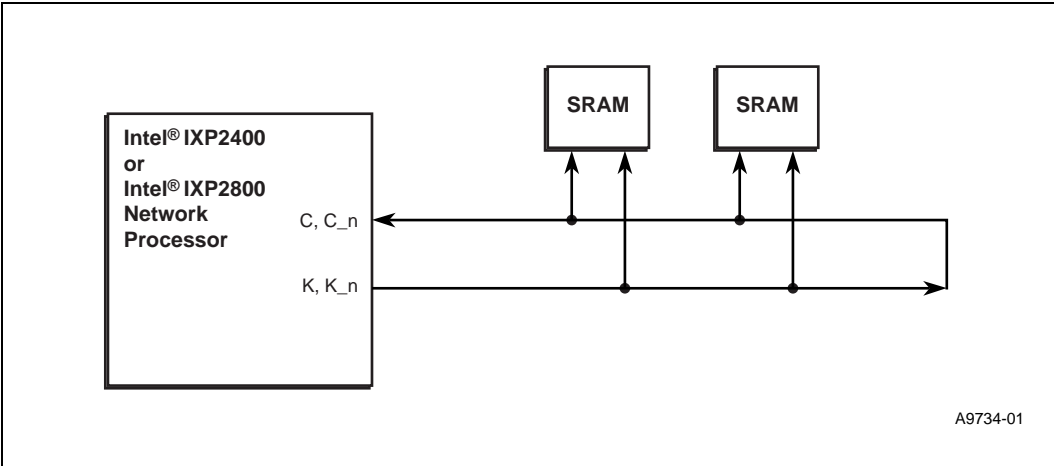
Maximum SRAM system sizes are shown in Table 73. Shaded entries require external decoding, because they use more port enables than the SRAM controller can directly supply.

Table 73. Total Memory per Channel

SRAM Size	Number of SRAMs on Channel							
	1	2	3	4	5	6	7	8
512K x 18	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB
1M x 18	2 MB	4 MB	6 MB	8 MB	10 MB	12 MB	14 MB	16 MB
2M x 18	4 MB	8 MB	12 MB	16 MB	20 MB	24 MB	28 MB	32 MB
4M x 18	8 MB	16 MB	24 MB	32 MB	64 MB	NA	NA	NA
8M x 18	16 MB	32 MB	48 MB	64 MB	NA	NA	NA	NA
16M x 18	32 MB	64 MB	NA	NA	NA	NA	NA	NA
32M x 18	64 MB	NA	NA	NA	NA	NA	NA	NA

Figure 75 shows how the SRAM clocks on a channel are connected. For receiving data from the SRAMs, clock path and data path are matched to meet hold time requirements.

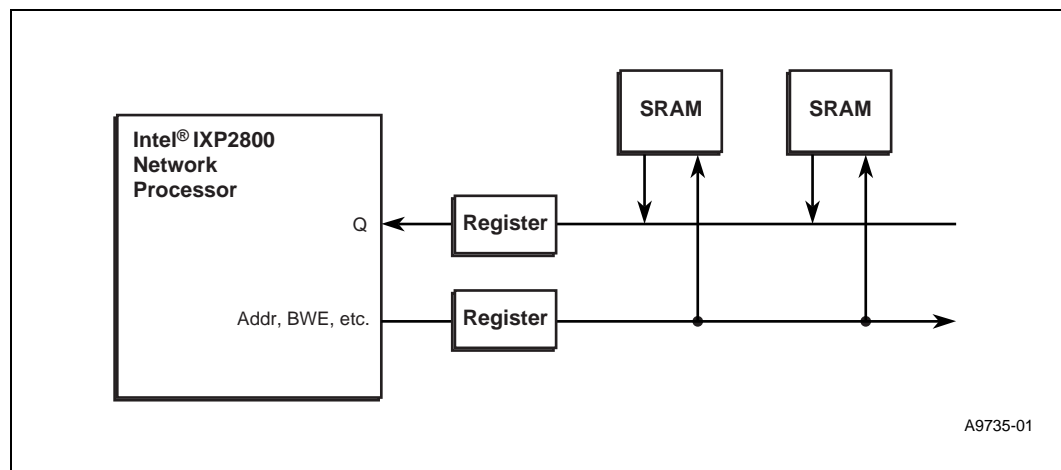
Figure 75. SRAM Clock Connection on a Channel



It is also possible to pipeline the SRAM signals with external registers. This is useful for the case when there is considerable loading on the address and data signals, which would slow down the cycle time. The pipeline stages make it possible to keep the cycle time fast by fanning out the address, byte write, and data signals. The RAM read data may also be put through a pipeline register, depending on configuration. External decoding of port selects can also be done to expand the number of SRAMs supported. Figure 76 is a block diagram that shows the concept of external pipelining.

A side effect of the pipeline registers is to add latency to reads, and the SRAM controller must account for that delay by waiting extra cycles (relative to no external pipeline registers) before it registers the read data. The number of extra pipeline delays is programmed in SRAM_Control[Pipeline].

Figure 76. External Pipeline Registers Block Diagram



6.5 Command Overview

This section will give an overview of the SRAM commands and their operation. The details will be given later in the document. Memory reference ordering will be specified along with the detailed command operation.

6.5.1 Basic Read/Write Commands

The basic read and write commands will transfer from 1 to 16 longwords of data to/from the QDR SRAM external to the IXP2000 series processor.

For a read command, the SRAM is read and the data placed on the Push bus one longword at a time. The command source (for example, the Microengine) is signaled that the command is complete during the last data phase of the push bus transfer.

For a write command, the data is first pulled from the source, then written to the SRAM in consecutive SRAM cycles. The command source is signaled that the command is complete during the last data phase of the pull bus transfer.



If a read operation stalls due to the pull-data FIFO filling, any concurrent write operation that is in progress to the same address will be temporarily stopped. This technique results in atomic data reads.

6.5.2 Atomic Operations

The SRAM Controller does `read-modify-writes` for the atomic operations, the pre-modified data can also be returned if desired. Other (non-atomic) readers and writers can access the addressed location in between the read and write portion of the read-modify-write. [Table 74](#) describes the atomic operations supported by the SRAM Controller.

Table 74. Atomic Operations

Instruction	Pull Operand	Value Written to SRAM
Set_bits	Optional ¹	SRAM_Read_Data OR Pull_Data
Clear_bits	Optional	SRAM_Read_Data AND NOT Pull_Data
Increment	No	SRAM_Read_Data + 0x00000001
Decrement	No	SRAM_Read_Data - 0x00000001
Add	Optional	SRAM_Read_Data + Pull_Data
Swap	Optional	Pull_Data

1. There are two versions of the Set, Clear, Add, and Swap instructions. One version pulls operand data from the Microengine transfer registers, while the second version passes the operand data to the SRAM Unit as part of the command.

Up to two Microengine signals will be assigned to each read-modify-write reference. Microcode should always tag the read-modify-write reference with an even numbered signal. If the operation requires a pull, then the requested signal will be sent on the pull. If the pre-modified data is to be returned to the Microengine, then the Microengine will be sent (requested signal OR 1) when that data is pushed.

In [Example 27](#), the version of `Test_and_Set` requires both a pull and a push:

Example 27. SRAM `Test_and_Set` with Pull Data

```
immed [$xfer0, 0x1]
SRAM[test_and_set, $xfer0, test_address, 0, 1], sig_done_2
// SIGNAL_2 is set when $xfer0 is pulled from this ME. SIGNAL_3 is
// set when $xfer0 is written with the test value. Sleep until both
// SIGNALS have arrived.
CTX_ARB[sigal_2, sigal_3]
```

In [Example 28](#) the version of Test_and_Set does not require a pull, but does issue a push. A signal is generated when the push is complete:

Example 28. SRAM Test_and_Set with Optional No-Pull Data

```
#define no_pull_mode_bit 24
#define byte_mask_override_bit 20
#define no_pull_data_bit 12
#define upper_part_bit 21

// This constant can be created once at init time
ALU[no_pull_constant, --, b, 0x3, <<no_pull_mode_bit]
ALU[no_pull_constant, no_pull_constant, or, 1, <<byte_mask_override_bit]

// Here is a no_pull test_and_add
ALU[temp, no_pull_constant, or, 0xfe, <<no_pull_data_bit]
ALU[temp, temp, or, 0x5, <<upper_part_bit]
SRAM[test_and_add, $x0, addra, 0], indirect_ref, sig_done[sig2]
CTX_ARB[sig2]
```

In [Example 29](#), an Increment operation does not require a pull:

Example 29. SRAM Increment without Pull Data

```
SRAM [incr, $xfer0, incr_address, 0, 1], signal_2
// SIGNAL_2 is set when $xfer0 is written with the pre-increment value.
CTX_ARB[signal_2]
```

6.5.3 Queue Data Structure Commands

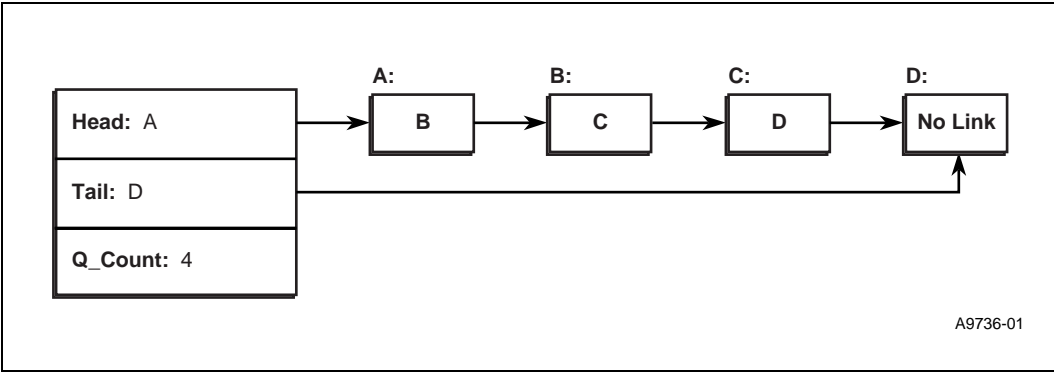
The ability to enqueue and dequeue data buffers at a fast rate is key to meeting chip performance goals. This is a difficult problem as it involves dependent memory references that must be turned around very quickly. The SRAM controller includes a data structure (called the Q_array) and associated control logic in order to perform efficient enqueue and dequeue operations. Optionally, this hardware or a portion of this hardware can be used to implement rings and journals.

A queue is an ordered list of data buffers stored at non-contiguous addresses. The first buffer added to the queue will be the first buffer removed from the queue. Queue entries are joined together by creating links from one data buffer to the next. This hardware implementation supports only a forward link. A queue is described by a pointer to its first entry (called the head) and a pointer to its last entry (the tail). In addition, there is a count of the number of items currently on the queue. This triplet (head, tail, and count) is referred to as the queue descriptor. In the IXP 2400 and IXP2800 chips, the queue descriptor is stored in that order—head first, then tail, then count. The longword alignment of the head addresses for all queue descriptors must be a power of two. For example, when there are no extra parameters on the queue descriptor, there will be one unused longword per queue descriptor.



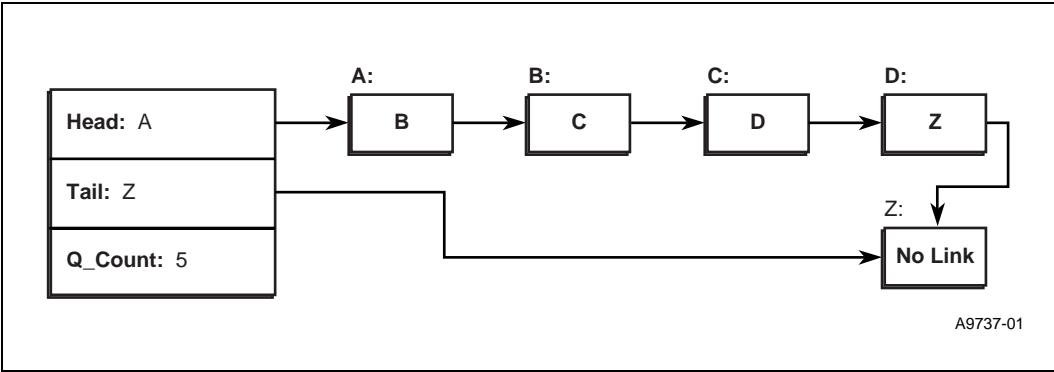
Figure 77 shows a queue descriptor and queue links for a queue containing four entries.

Figure 77. Queue Descriptor with Four Links



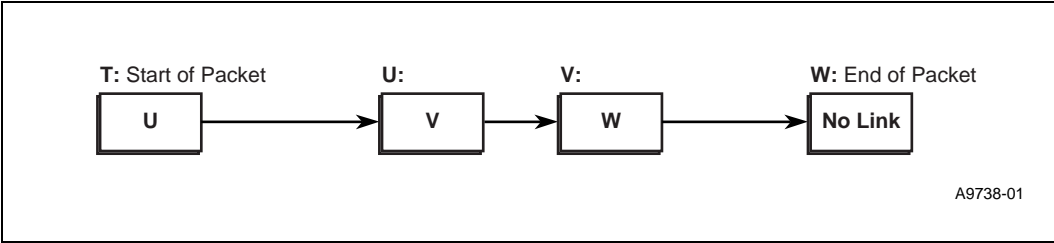
There are two different versions of the enqueue command, `ENQ_tail_and_link` and `ENQ_tail`. `ENQ_tail_and_link` is used to enqueue one buffer at a time. In Figure 77, issuing an `ENQ_tail_and_link` to buffer link address z will result in the queue shown in Figure 78.

Figure 78. Enqueueing One Buffer at a Time



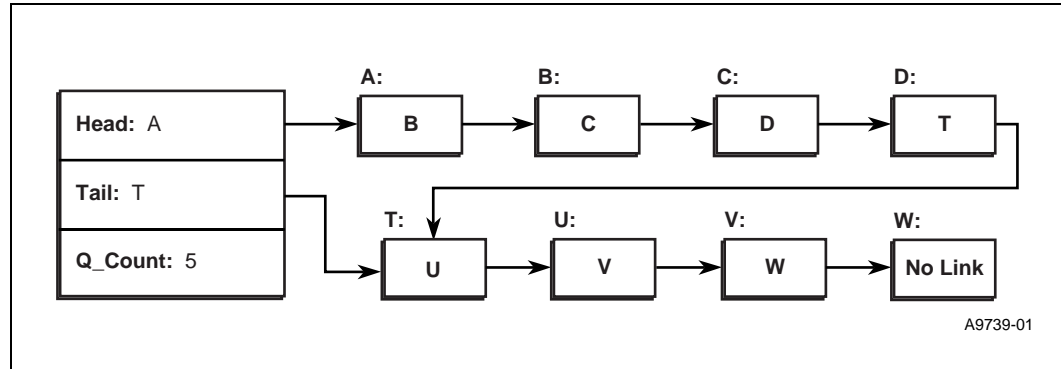
`ENQ_tail_and_link` followed by `ENQ_tail` are used to enqueue a *previously* linked string of buffers. The string of buffers is used in the case where one packet is too large to fit in one buffer. Instead, it is divided among multiple buffers. Figure 79 is an example of a string of buffers.

Figure 79. Previously Linked String of Buffers



To enqueue the string of buffers in Figure 79 to the example queue in Figure 77, first issue ENQ_tail_and_link to address T. Figure 80 is the result.

Figure 80. First Step to Enqueue a String of Buffers to a Queue (ENQ_Tail_and_Link)

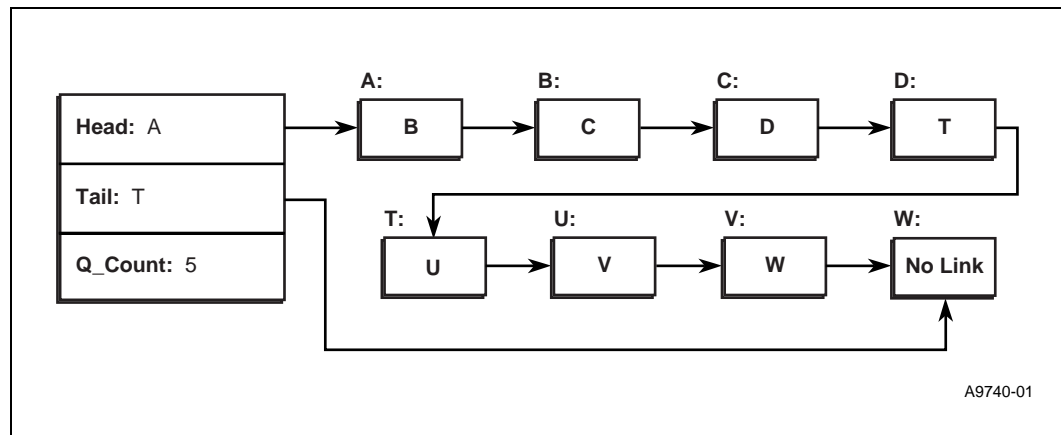


The second step is to issue and ENQ_tail to address W. This will fix the Tail to point to the last buffer of the string.

Note: Q_count is not changed by ENQ_tail because the string of buffers represents one packet.

Figure 81 is the final queue state.

Figure 81. Second Step to Enqueue a String of Buffers to a Queue (ENQ_Tail)



There are two different modes for the dequeue command. One mode removes an entire buffer from the queue. The second mode removes a piece of the buffer (referred to as a cell). The mode (cell dequeue or buffer dequeue) is selectable on a buffer-by-buffer basis by setting the cell_count bits (<30:24>) in the link longword.

A ring is an ordered list of data words stored in a fixed block of contiguous addresses. A ring is described by a head pointer and a tail pointer. Data is written, using the put command, to a ring at the address contained in the tail pointer and the tail pointer is incremented. Data is read, using the get command, from a ring at the address contained in the head pointer and the head pointer is incremented. Whenever either pointer reaches the end of the ring, the pointer is wrapped back to the address of the start of the ring.



A journal is similar to a ring. It is generally used for debugging. Journal commands only write to the data structure. New data overwrites the oldest data. Microcode can choose to tag the journal data with the Microengine number and CTX number of the journal writer.

The `Q_array` to support queuing, rings and journals contains 64 registers per SRAM channel. For a design with a large number of queues, the queue descriptors cannot all be stored on chip, and thus a subset of the queue descriptors (16) is cached in the `Q_array`. *To implement the cache, 16 contiguous `Q_array` registers must be allocated.* The cache tag (the mapping of queue number to `Q_array` registers) for the `Q_array` is maintained by microcode in the CAM of a Microengine. The writeback and load of the cached registers in the `Q_array` is under the control of that microcode.

Note: The size of the `Q_array` does not set a limit on the number of queues supported.

For other queues (free buffer pools, for example), rings, and journals, the information does not need to be subsetting and thus can be loaded into the `Q_array` at initialization time and left there to be updated solely by the SRAM controller.

The sum total of the cached queue descriptors plus the number of rings, journals and static queues must be less than or equal to 64 for a given SRAM channel.

The fields and sizes of the `Q_array` registers are shown in [Table 75](#) and [Table 76](#). All addresses are of type *longword*, and are 32 bits in length.

Table 75. Queue Format

Name	Longword #	Bit # ¹	Definition
EOP	0	31	End of Packet—decrement <code>Q_count</code> on dequeue
SOP	0	30	Start of Packet—used by the programmer
Cell Count	0	29:24	Number of cells in the buffer
Head	0	23:0	Head pointer
Tail	1	23:0	Tail pointer
<code>Q_count</code>	2	23:0	Number of packets on the queue or number of buffers on the queue
<code>SW_Private</code>	2	31:24	Ignored by hardware, returned to Microengine
Head Valid	N/A		Cached head pointer valid—maintained by hardware
Tail Valid	N/A		Cached tail pointer valid—maintained by hardware

1. Bits 31:24 of longword number 2 are available for use by ucode.

Table 76. Ring/Journal Format

Name	Longword #	Bit #	Definition
Ring Size	0	31:29	See Table 77 for size encoding.
Head	0	23:0	<i>Get pointer</i>
Tail	1	23:0	<i>Put pointer</i>
Ring Count	2	23:0	Number of longwords on the ring

Note: For a Ring or Journal, `Head` and `Tail` must be initialized to the same address.

Journals/Rings can be configured to be one of eight sizes, as shown in [Table 77](#).

Table 77. Ring Size Encoding

Ring Size Encoding	Size of Journal/Ring Area	Head/Tail Field Base	Head and Tail Field Increment
000	512 Longwords	23:9	8:0
001	1K	23:10	9:0
010	2K	23:11	10:0
011	4K	23:12	11:0
100	8K	23:13	12:0
101	16K	23:14	13:0
110	32K	23:15	14:0
111	64K	23:16	15:0

The following sections contain pseudo-code to describe the operation of the various queue and ring instructions.

Note: For these examples, NIL is the value 0.

6.5.3.1 Read_Q_Descriptor Commands

These commands are used to bring the queue descriptor data from QDR SRAM memory into the `Q_array`. Only portions of the `Q_descriptor` are read with each variant of the command in order to minimize QDR SRAM bandwidth utilization. It is assumed that microcode has previously evicted the `Q_descriptor` data for the entry prior to overwriting the entry data with the new `Q_descriptor` data. Refer to the *IXP2400/IXP2800 Programmer's Reference Manual*, Section 3.2.47, "SRAM (Read Queue Descriptor)" for more information.

6.5.3.2 Write_Q_Descriptor Commands

The write `Q_descriptor` commands are used to evict an entry in the `Q_array` and return its contents to QDR SRAM memory. Only the valid fields of the `Q_descriptor` are written in order to minimize QDR SRAM bandwidth utilization. Refer to the *IXP2400/IXP2800 Programmer's Reference Manual*, Section 3.2.48, "SRAM (Write Queue Descriptor)" for more information.

6.5.3.3 ENQ and DEQ Commands

These commands add or remove elements from the queue structure while updating the `Q_array` registers. Refer to the *IXP2400/IXP2800 Programmer's Reference Manual*, Section 3.2.49, "SRAM (Enqueue)" and Section 3.2.5, "SRAM (Dequeue)" for more information.

6.5.4 Ring Data Structure Commands

The ring structure commands use the `Q_array` registers to hold the head tail and count data for a ring data structure, which is a fixed size array of data with insert and remove pointers. Refer to the *IXP2400/IXP2800 Programmer's Reference Manual*, Section 3.2.53, "SRAM (Ring Operations)" for more information.



6.5.5 Journaling Commands

Journaling commands use the `Q_array` registers to index into an array of memory in the QDR SRAM that will be periodically written with information to help debug applications running on the IXP 2400 and IXP2800 processors. Once the array has been completely written once, subsequent journal writes will overwrite the previously written data—only the most recent data will be present in the data structure. Refer to the *IXP2400/IXP2800 Programmer's Reference Manual*, Section 3.2.52, “SRAM (Journal Operations)” for more information.

6.5.6 CSR Accesses

CSR accesses will write or read CSRs within each controller. The upper address bits will determine which channel will respond, while the CSR address within a channel are given in the lower address bits.

6.6 Parity

SRAM can be optionally protected by byte parity. Even parity is used—the combination of eight data bits and the corresponding parity bit will have an even number of ‘1’s. The SRAM controller generates parity on all SRAM writes. When parity is enabled (`SRAM_Control[Par_Enable]`) the SRAM controller checks for correct parity on all reads. Upon detection of a parity error on a read or the read portion of an atomic read-modify-write, the SRAM controller will record the address of the location with bad parity in `SRAM_Parity[Address]` and set the appropriate `SRAM_Parity[Error]` bit(s). Those bit(s) will interrupt the Intel XScale® core when enabled in `IRQ_Enable[SRAM_Parity]` or `FIQ_Enable[SRAM_Parity]`. The *Data Error* signal in the `Push_CMD` will be asserted when the data to be read is delivered (unless the token `Ignore Data Error` was asserted in the command; in that case the SRAM controller will not assert *Data Error*). When *Data Error* is asserted, the Push Arbiter will suppress the Microengine signal if the read was originated by a Microengine (it will use `0x0`, which is a null signal, in place of the requested signal number).

Note: If incorrect parity is detected on the read portion of an atomic read-modify-write, the incorrect parity will be preserved after the write (that is, the byte(s) with bad parity during the read will have incorrect parity written during the write).

When parity is used, Intel XScale® core software must initialize the SRAMs by:

1. Enable parity (write a 1 to `SRAM_Control[Par_Enable]`).
2. Writing to every SRAM address.

SRAM should not be read prior to doing the above initialization, otherwise parity errors are likely to be recorded.



6.7 Address Map

Each SRAM channel occupies a 1GB region of addresses. Channel 0 starts at 0, Channel 1 at 1GB, etc. Each SRAM controller receives commands from the command buses. It compares the target ID to the SRAM target ID, and address bits 31:30 to the channel number. If they both match, then the controller processes the command. See [Table 78](#).

Table 78. Address Map

Start Address	End Address	Responder
0x0000 0000	0x3FFF FFFF	Channel 0
0x4000 0000	0x7FFF FFFF	Channel 1
0x8000 0000	0xBFFF FFFF	Channel 2
0xc000 0000	0xFFFF FFFF	Channel 3

Note: If an access addresses a non-existent address within an SRAM controller's address space the results are unpredictable. For example the result of accessing address 0x0100 0000 when there is only 1 MB of SRAM populated on the channel will produce unpredictable results.

For SRAM (memory or CSR) references from the Intel XScale® core, the channel select is in address bits 29:28. The Gasket shifts those bits to 31:30 to match addresses generated by the MEs. Thus, the SRAM channel select logic is the same whether the command source is a Microengine or the Intel XScale® core.

The same channel start and end addresses are used both for SRAM memory and CSR references. CSR references are distinguished from memory references through the CSR encoding in the command field.

Note: Reads and writes to undefined CSR addresses will yield unpredictable results.

The IXP 2400 and IXP2800 addresses are byte addresses. As the fundamental unit of operation of the SRAM controller is a longword access, the SRAM controller will ignore the 2 low order address bits in *all* cases and utilize the byte mask field on memory address space writes to determine the bytes to write into the SRAM. Any combination of the four bytes can be masked. The operation of byte writes with a length other than 1 are unpredictable. That is, microcode should not use a ref_count greater than 1 longword when a byte_mask is active. CSRs are not byte writable.



6.8 Reference Ordering

This section discusses the ordering between accesses to any one SRAM controller. Various mechanisms are used to guarantee order—for example, references that always go to the same FIFOs remain in order. There is a CAM associated with *write addresses* that is used to order reads behind writes. Lastly, several counter pairs are used to implement “fences”. The input counter is tagged to a command and the command is not permitted to execute until the output counter matches the fence tag. All of this will be discussed in more detail in this section.

6.8.1 Reference Order Tables

Table 79 shows the architectural guarantees of order of accesses to the *same* SRAM address between a reference of any given type (shown in the column labels) and a subsequent reference of any given type (shown in the row labels). The definition of first and second is defined by the time the command is valid on the command bus. Verification requires testing only the order rules shown in Table 79 and Table 80. Note that a blank entry means no order is enforced.

Table 79. Address Reference Order

<div> <div>1st ref →</div> <div>2nd ref ↓</div> </div>	Memory Read	CSR Read	Memory Write	CSR Write	Atomics	Queue / Ring / Q_Descr Commands
Memory Read			Order		Order	
CSR Read				Order		
Memory Write			Order		Order	
CSR Write				Order		
Atomics			Order		Order	
Queue / Ring / Q_Descr Commands						See Table 80.

Table 80 shows the architectural guarantees of order to access to the *same* SRAM Q_array entry between a reference of any given type (shown in the column labels) and a subsequent reference of any given type (shown in the row labels). The terms first and second are defined with reference to the time the command is valid on the command bus. The same caveats that apply to Table 79 apply to Table 80.

Table 80. Q_array Entry Reference Order

1 st ref 2 nd ref	Read_Q_Descr head, tail	Read_Q_Descr other	Write_Q_Descr	Enqueue	Dequeue	Put	Get	Journal
Read_Q_Descr head,tail			Order ¹					
Read_Q_Descr other	Order							
Write_Q_Descr ²								
Enqueue	Order	Order		Order	Order ³			
Dequeue	Order	Order		Order ³	Order			
Put						Order		
Get							Order	
Journal								Order

1. The order of Read_Q_Descr_head/tail after Write_Q_Descr to the same element will be guaranteed only if it is to a different descriptor SRAM address. The order of Read_Q_Descr_head/tail after Write_Q_Descr to the same element with the same descriptor SRAM address is not guaranteed and should be handled by the Microengine code.
2. Write_Q_Descr reference order is not guaranteed after any of the other references. The Queue array hardware assumes that the Microengine managing the cached entries will flush an element ONLY when it becomes the LRU in the Microengine CAM. Using this scheme, the time between the last use of this element and the write reference is sufficient to guarantee the order.
3. Order between Enqueue references and Dequeue references are guaranteed only when the Queue is empty or near empty.

6.8.2 Microcode Restrictions to Maintain Ordering

It is the microcode programmer's job to ensure order where the program flow requires order and where the architecture does not guarantee that order.

One mechanism that can be used to do this is signaling. For example, say that microcode needs to update several locations in a table. A location in SRAM is used to *lock* access to the table.

Example 30 is the microcode for this table update.

Example 30. Table Update Microcode

```

IMMED [$xfer0, 1]
SRAM [write, $xfer0, flag_address, 0, 1, ctx_swap [SIG_DONE_2]
; At this point, the write to flag_address has passed the point of coherency. Do
the table updates.
SRAM [write, $xfer1, table_base, offset1, 2] , sig_done [SIG_DONE_3]
SRAM [write, $xfer3, table_base, offset2, 2] , sig_done [SIG_DONE_4]
CTX_ARB [SIG_DONE_3, SIG_DONE_4]
; At this point, the table writes have passed the point of coherency. Clear the
flag to allow access by other threads.
IMMED [$xfer0, 0]
SRAM [write, $xfer0, flag_address, 0, 1, ctx_swap [SIG_DONE_2]
```

Other microcode rules:

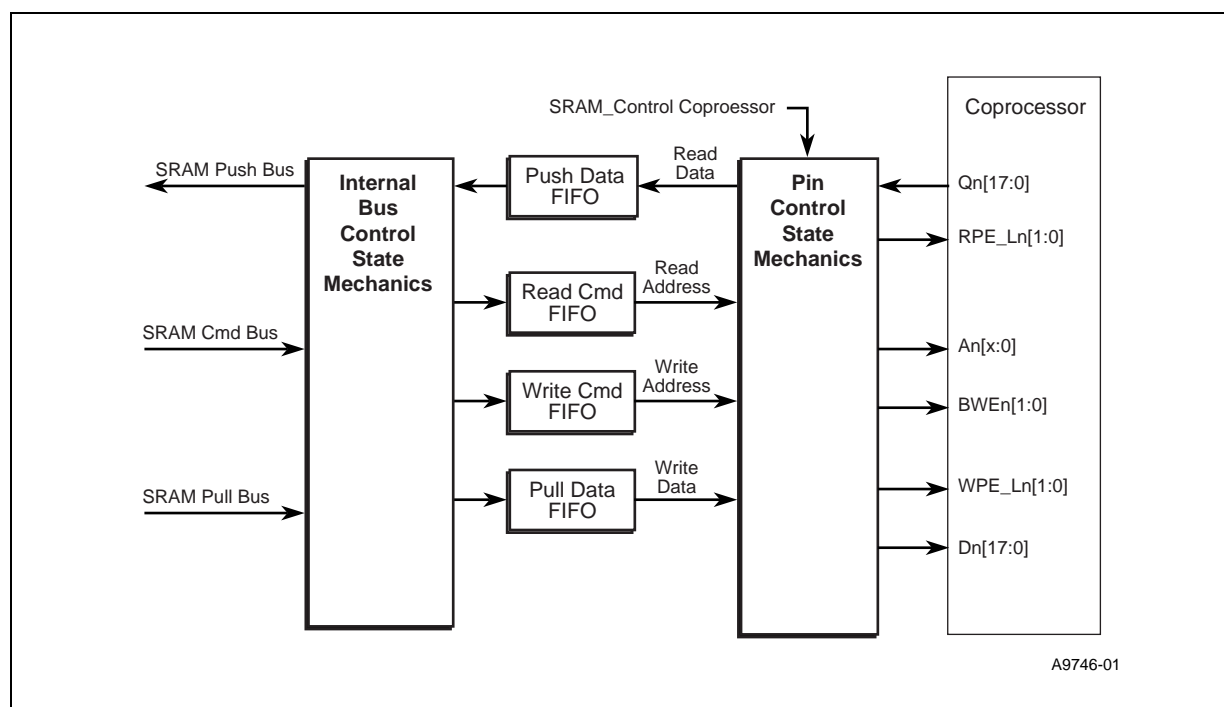
- All access to atomic variables should be through read-modify-write instructions.
- If the flow must know that a write is completed (actually in the SRAM itself), follow the write with a read to the same address. The write is guaranteed to be complete when the read data has been returned to the Microengine.
- With the exception of initialization, never do *write* commands to the first 3 longwords of a `queue_descriptor` data structure (these are the longwords that hold head, tail, and count). All accesses to this data must be through the Q commands.
- To initialize the `Q_array` registers, perform a memory write of at least 3 longwords, followed by a memory read to the same address (to guarantee that the write completed). Then, for each entry in the `Q_array`, perform a `read_q_descriptor_head` followed by a `read_q_descriptor_other` using the address of the same 3 longwords.

6.9 Coprocessor Mode

Each SRAM controller may interface to an external coprocessor through its standard QDR interface. This interface will allow for the cohabitation of both SRAM devices and coprocessors operating on the same bus. The coprocessor will behave as a memory mapped device on the SRAM bus. Figure 82 is a simplified block diagram of the SRAM controller. Figure 82 shows the connection to a coprocessor through a standard QDR interface.

Note: Most coprocessors will not need a large number of address bits—connect as many bits of A_n as required by the coprocessor.

Figure 82. Connection to a Coprocessor Through Standard QDR Interface



The external coprocessor interface is based on FIFO communication.

A thread can send parameters to the coprocessor by doing a normal SRAM write instruction:

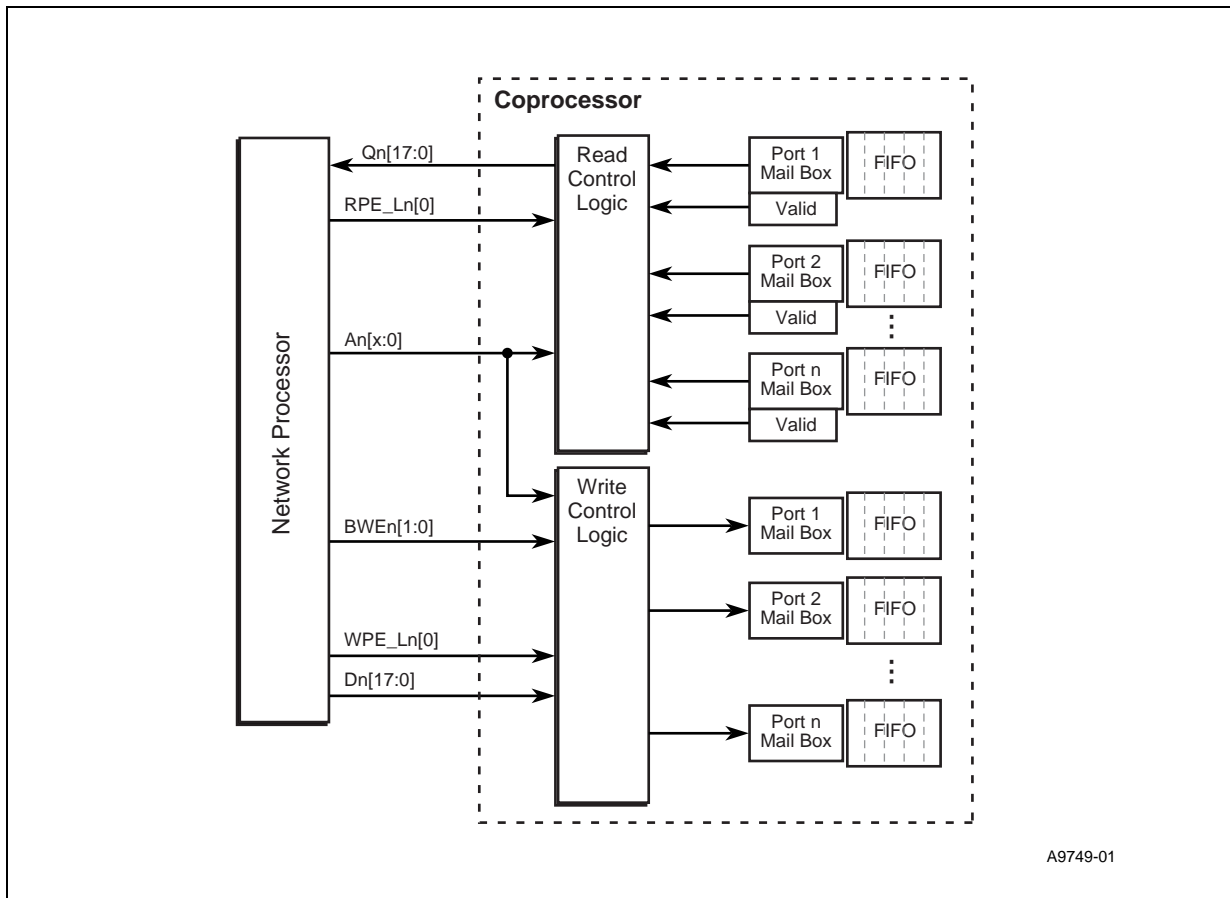
```
sram[write, $sram_xfer_reg, src1, src2, ref_count], optional_token
```

The number of parameters (longwords) passed is specified by *ref_count*. The address can be used to support multiple coprocessor FIFO ports. The coprocessor will perform some operation using the parameters, and then, sometime later it will pass back some number of results values (the number of parameters and results will be known by the coprocessor designers). The time between the input parameter and return values is not fixed; it is determined by the amount of time the coprocessor requires to do its processing and can be variable. When the coprocessor is ready to return the results it signals back to the SRAM controller through a mailbox valid bit that the data in the read FIFO is valid. A thread can get the return values by doing a normal SRAM read instruction:

```
sram[read, $sram_xfer_reg, src1, src2, ref_count], optional_token
```

Figure 83 shows the coprocessor with 1 to *n* memory-mapped FIFO ports.

Figure 83. Coprocessor with Memory Mapped FIFO Ports



If the read instruction executes before the return values are ready, the coprocessor will signal data invalid through the mailbox register on the read data bus ($Q_n[17:0]$). Signaling a thread upon pushing its read data works exactly as in a normal SRAM read.

There can be multiple operations in-progress in the coprocessor. The SRAM controller will send parameters to the coprocessor in response to each SRAM write instruction without waiting for return results of previous writes. If the coprocessor is capable of re-ordering operations—that is, returning the results for a given operation before returning the results of an earlier arriving operation—Microengine code must manage matching results to operations. Tagging the operation by putting a sequence value into the parameters, and having the coprocessor copy that value into the results is one way to accomplish this requirement.

Flow control will be under the Network Processor's Microengine control. An Microengine thread accessing a coprocessor port will maintain a count of the number of entries in that coprocessor 's write FIFO port. Each time an entry is written to that coprocessor port the count will be incremented. When a valid entry is read from that coprocessor read port the count will be decrement by the thread.



SHaC—Unit Expansion

7

This section covers the operation of the Scratchpad, Hash Unit and CSRs (SHaC).

7.1 Overview

The SHaC unit is a multifunction block containing Scratchpad memory and logic blocks to perform hashing operations and interface with Intel XScale® core peripherals and chip CSRs through the APB and CSR buses, respectively. The SHaC also houses the global registers, as well as chip Reset logic.

The SHaC unit has the following features:

- Communication to Intel XScale® core peripherals, such as GPIOs and timers, through the APB bus.
- Creation of hash indices of 48, 64, or 128-bit widths.
- Communication ring used by MicroEngines (MEs) for interprocess communication.
- Third-option memory storage usable by Intel XScale® core and MEs.
- CSR bus interface to permit fast writes to CSRs, as well as standard read and writes.
- Push/Pull Reflector to transfer data from the Pull bus to the Push bus.

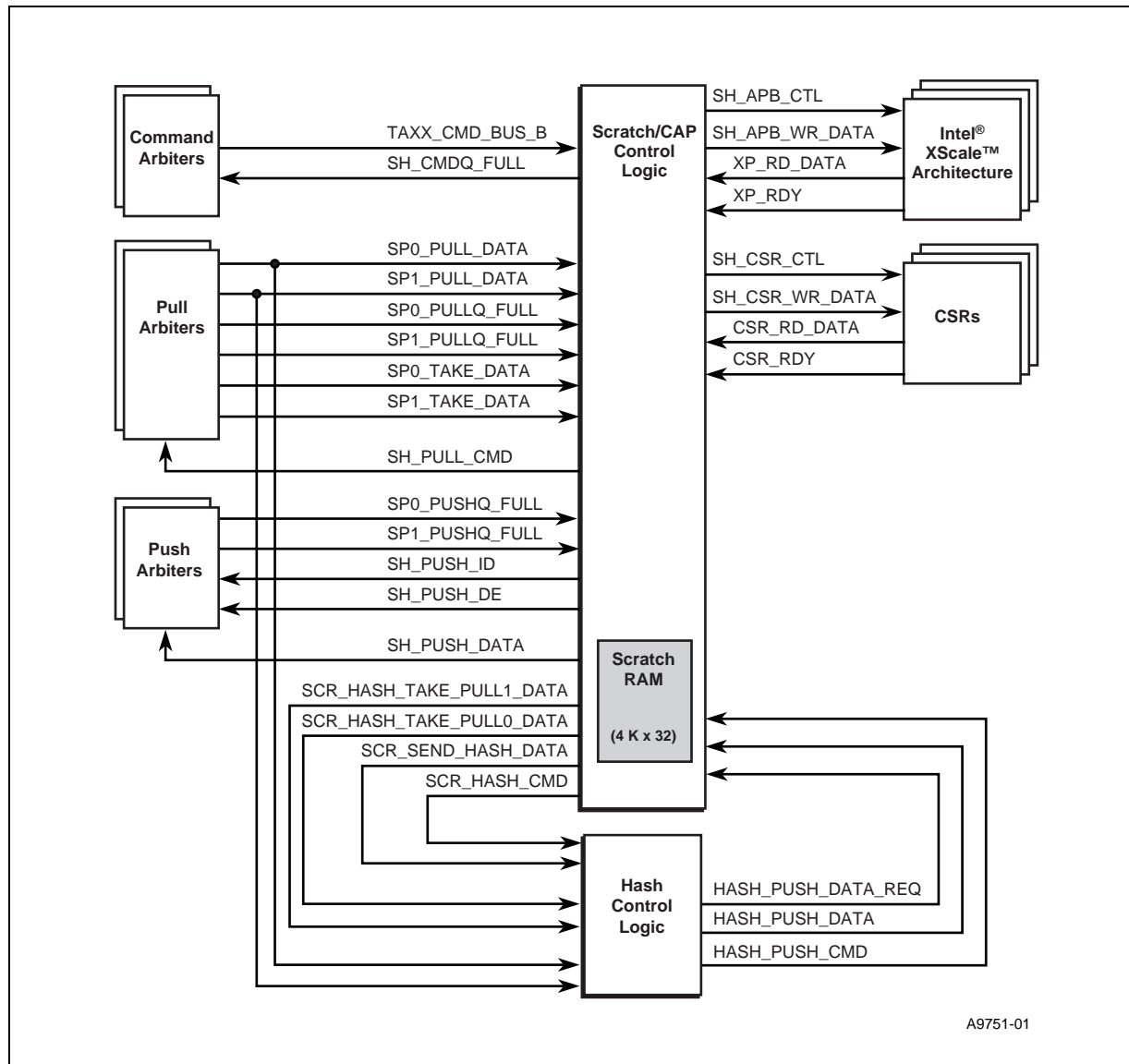
The CSR and ARM Advanced Peripheral Bus (APB) bus interfaces are controlled by the Scratchpad state machine and will be addressed in the Scratchpad design detail section. (See [Section 7.1.2.](#))

Note: Detailed information about CSRs is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

7.1.1 SHaC Unit Block Diagram

The SHaC unit contains two functional units: the Scratchpad and Hash Unit. Each will be described in greater detail in the following sections. The CAP and APB bus interfaces will be addressed as part of the Scratchpad description.

Figure 84. SHaC Top Level Diagram





7.1.2 Scratchpad

7.1.2.1 Scratchpad Description

The SHA-C Unit contains a 16 Kbyte Scratchpad memory, organized as 4K 32-bit words, that is accessible by the Intel XScale® core and MicroEngines (MEs). The Scratchpad connects to the internal Command, S_Push/S_Pull, CSR, and APB buses, as shown in [Figure 85](#).

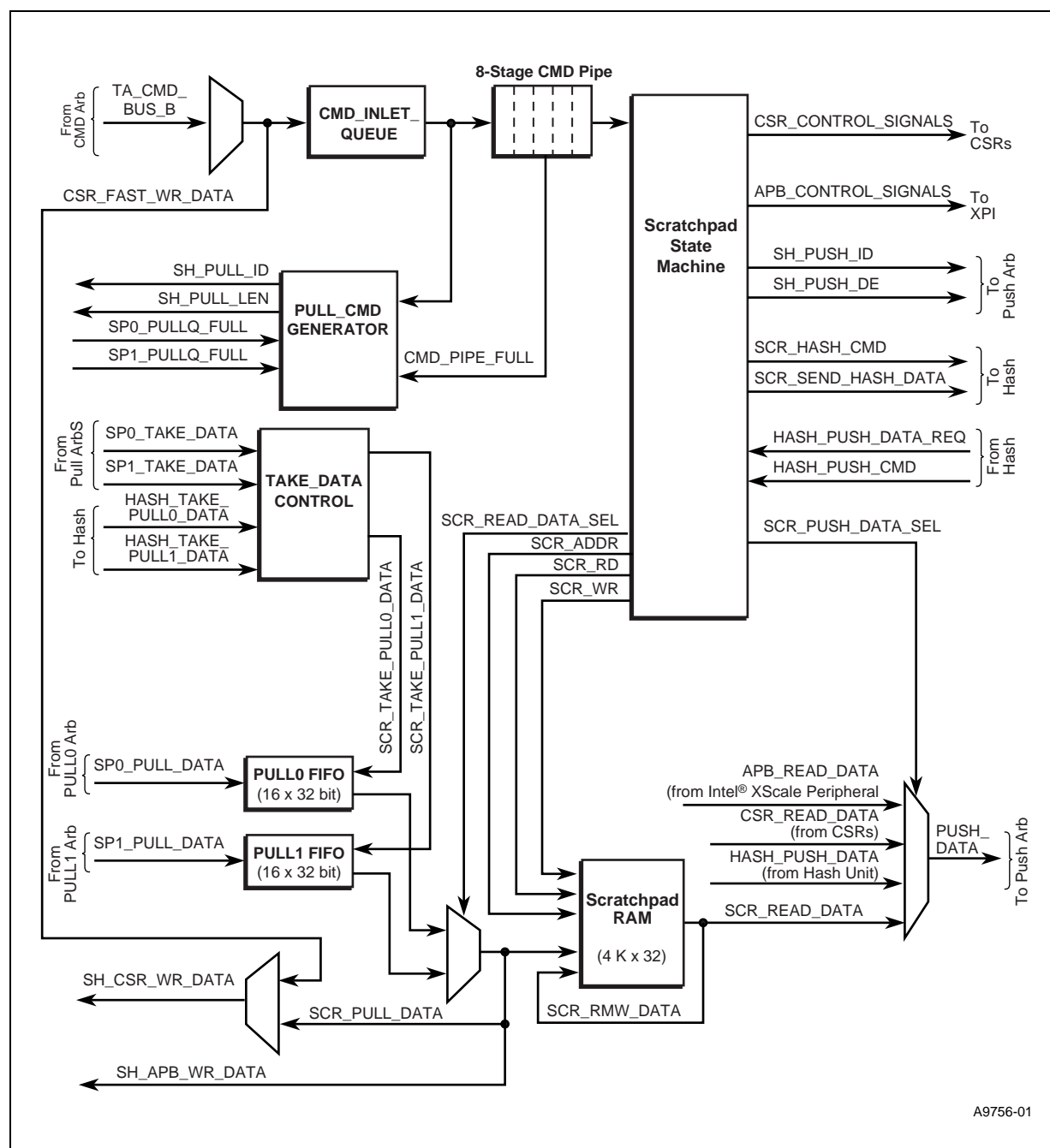
The Scratchpad memory provides the following operations:

- Normal reads and writes. From one to 16 longwords (32 bits) can be read/written with a single command. Note that Scratchpad is not byte-writable. Each write must write *all four bytes*.
- Atomic read-modify-write operations, bit-set, bit-clear, increment, decrement, add, subtract, and swap. The Read-Modify-Write (RMW) operations can also optionally return the premodified data.
- Sixteen Hardware Assisted Rings for interprocess communication.¹
- Standard support of APB peripherals such as UART, Timers, and GPIOs through the ARM Advanced Peripheral Bus (APB).
- Fast write and standard read and write operations to CSRs through the CSR Bus. A fast write is where the write data is supplied with the command, rather than pulling the data from the source.
- Push/Pull Reflector Mode that supports reading from a device on the pull bus and writing the data to a device on the push bus (reflecting the data from one bus to the other). A typical implementation of this mode is to allow a Microengine to read or write the transfer registers or CSRs in another Microengine. Note that the Push/Pull Reflector Mode only connects to a single Push/Pull bus. If a chassis implements more than one Push/Pull bus, it can only connect one specific bus to the CAP.

Scratchpad memory is provided as a third memory resource (in addition to SRAM and DRAM) that is shared by the MEs and Intel XScale® core. The MEs and Intel XScale® core can distribute memory accesses between these three types of memory resources to provide a greater number of memory accesses occurring in parallel.

1. A ring is a FIFO that uses a head and tail pointer to store/read information in Scratchpad memory.

Figure 85. Scratchpad Block Diagram



A9756-01

7.1.2.2 Scratchpad Interface

Note: The Scratchpad command and S_Push and S_Pull bus interfaces actually are shared with the Hash Unit. Only one command, to either of those units, can be accepted per cycle.

The CSR and APB buses will be described in detail in following sections.

7.1.2.2.1 Command Interface

The Scratchpad accepts commands from the Command Bus and can accept one command every cycle.

For Push/Pull reflector write and read commands, the command bus is rearranged before being sent to the Scratchpad state machine in order to allow a single state (REFLECT_PP) to be used to handle both commands.

7.1.2.2.2 Push/Pull Interface

The Scratchpad has the capability to interface to either one or two pairs of push/pull (PP) bus pairs. The interface from the Scratchpad to the PP bus pair is through the Push/Pull Arbiters. Each PP bus has a separate Push and Pull arbiter through which access to the Push bus and Pull bus, respectively, is regulated. Refer to the SRAM Push Arbiter and SRAM Pull Arbiter chapters for more information. When the Scratchpad is used in a chip that only utilizes one pair of PP buses, the other interface is unused.

7.1.2.2.3 CSR Bus Interface

The CSR Bus provides fast write and standard read and write operations from the Scratchpad to the CSRs in the CSR block.

7.1.2.2.4 Advanced Peripherals Bus Interface (APB)

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) hierarchy of buses that is optimized for minimal power consumption and reduced design complexity.

Note: The SHaC Unit uses a modified APB interface in which the APB peripheral is required to generate an acknowledge signal (APB_RDY_H) during read operations. This is done to indicate that valid data is on the bus. The addition of the acknowledge signal is an enhancement added specifically for the IXP Chassis. (For more details refer to the *ARM AMBA Specification 1.6.1.3*.)

7.1.2.3 Scratchpad Block Level Diagram

Scratchpad Command Overview

This section will detail the operations performed for each Scratchpad command. Command order is preserved because all commands go through a single command inlet FIFO.

When a valid command is placed on the command bus, the control logic checks the instruction field for the Scratchpad or CAP ID. The command, address, length, etc. are enqueued into the Command Inlet FIFO. If the command requires pull data, signals are generated and immediately sent to the Pull Arbiter. The command is pushed from the Inlet FIFO to the command pipe where it will be serviced according to the command type.



If the Command Inlet FIFO becomes full, the Scratchpad controller will send a full signal to the command arbiter which will prevent it from sending further Scratchpad commands.

7.1.2.3.1 Scratchpad Commands

The basic read and write commands will transfer from 1 to 16 longwords of data to/from the Scratchpad.

Reads

When a read command is at the head of the Command queue, the Push Arbiter is checked to see if it has enough room for the data. If so, the Scratchpad RAM is read, and the data is sent to the Push Arbiter one 32-bit word at a time (the Push_ID is updated for each word pushed). The Push Data is sent to the specified destination.

The read data is placed on the S_Push bus one 32-bit word at a time. If the master is a Microengine, it is signaled that the command is complete during the last phase of the push bus transfer. Other masters (Intel XScale® core and PCI) must count the number of data pushes to know when the transfer is complete.

Writes

When a write command is at the head of the Command Inlet FIFO, signals are sent to the Pull Arbiter. If there is room in the queue, the command is sent to the Command pipe.

When a write command is at the head of the Command pipe, the command waits for a signal from the Pull Data FIFO, indicating the data to be written is valid. Once the first longword is received, the data is written on consecutive cycles to the Scratchpad RAM until the burst (up to 16 longwords) is completed.

If the master is a Microengine, it is signaled that the command is complete during the last pull bus transfer. Other masters (Intel XScale® core and PCI) must count the number of data pulls to know when the transfer is complete.

Atomic Operations

The Scratchpad supports the following atomic operations.

- bit set
- bit clear
- increment
- decrement
- add
- subtract
- swap

The Scratchpad does read-modify-writes for the atomic operations, the pre-modified data also can be returned, if desired. The atomic operations operate on a single longword. There is one cycle between the read and write while the modification is done. In that cycle no operation is done, so an access cycle is lost.

When a read-modify-write command requiring pull data from a source is at the head of the Command Inlet FIFO, a signal is generated and sent to the Pull Arbiter (if there is room).

When the RMW command reaches the head of the Command pipe, the Scratchpad reads the memory location in the RAM. If the source requests the pre-modified data (Token[0] set), it is sent to the Push Arbiter at the time of the read. If the RMW requires pull data, the command waits for the data to be placed into the Pull Data FIFO before performing the operation; otherwise the operation is performed immediately. Once the operation has been performed, the modified data is written back to the Scratchpad RAM.

Up to two Microengine signals will be assigned to each read-modify-write reference. Microcode should always tag the read-modify-write reference with an even numbered signal. If the operation requires a pull, then the requested signal will be sent on the pull. If the read data is to be returned to the Microengine, then the Microengine will be sent (requested signal OR 1) when that data is pushed.

For all atomic operations, whether or not the read data is returned is determined by Command bus Token[0].

Note: Intel XScale® core can do atomic commands using aliased addresses in Scratchpad. An Intel XScale® core Store instruction to an atomic command address will do the RMW without returning the read data, an Intel XScale® core Swap instruction (SWP) to an atomic command address will do the RMW and return the read data to Intel XScale® core.

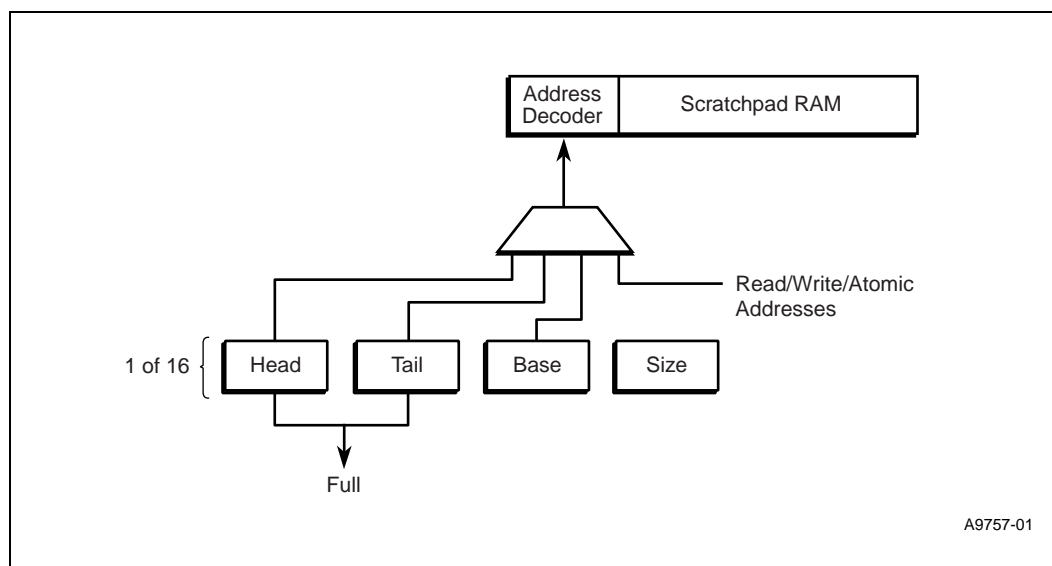
7.1.2.3.2 Ring Commands

The Scratchpad provides 16 Rings used for interprocess communication. The rings provide two operations.

- Get(ring, length)
- Put(ring, length)

Ring is the number of the ring (0 through 15) to get from or put to, and length specifies the number of longwords to transfer. A logical view of one of the rings is shown in Figure 86.

Figure 86. Ring Communication Logic Diagram





Head, Tail, Base, and Size are registers in the Scratchpad Unit. Head and Tail point to the actual ring data, which is stored in the Scratchpad RAM. For each ring in use, a region of Scratchpad RAM must be reserved for the ring data. The reservation is by software convention. The hardware does not prevent other accesses to the region of Scratchpad used by the ring. Also, the regions of Scratchpad memory allocated to different rings must not overlap.

Head points to the next address to be read on a get, and Tail points to the next address to be written on a put. The size of each ring is selectable from the following choices: 128, 256, 512, or 1,024 32-bit words. The size is specified in the Ring_Base register.

Note: The above rule stating that rings must not overlap implies that many configurations are not legal. For example, programming five rings to size of 1024 words would exceed the total size of Scratchpad memory, and therefore is not legal.

Note: Note that the region of Scratchpad used for a ring is naturally aligned to its size.

Each ring asserts an output signal which is used as a state input to the MEs. The software configures whether the Scratchpad asserts the signal if a ring becomes empty or if the ring is near full.

If configured to assert status when the rings are near full, MEs must test the input state (by doing Branch on Input Signal) before putting data onto a ring. There is a lag in time from a put instruction executing to the Full signal being updated to reflect that put. To be guaranteed that a put will not overfill the ring there is a bound on the number of Contexts and the number of 32-bit words per write based on the size of the ring, shown in Table 81. Each Context should test the Full signal, then do the put if not Full, and then wait until the Context has been signaled that the data has been pulled before testing the Full signal again.

Table 81. Ring Full Signal Use -- Number of Contexts and Length Versus Ring Size

Number of Contexts	Ring Size			
	128	256	512	1024
1	16	16	16	16
2	16	16	16	16
4	8	16	16	16
8	4	12	16	16
16	2	6	14	16
24	1	4	9	16
32	1	3	7	15
40	Illegal	2	5	12
48	Illegal	2	4	10
64	Illegal	1	3	7
128	Illegal	Illegal	1	3
NOTE: 1. Number in each table entry is the largest length that should be put. 16 is the largest length that a single put instruction can generate. 2. Illegal - With that number of Contexts, even a length of 1 could cause ring to overfill.				

The ring commands operate as outlined in the pseudo code in [Example 31](#). The operations are atomic meaning that multi-word gets and puts do all the reads and writes with no other intervening Scratchpad accesses.

Example 31. Ring Command Pseudo Code

GET Command

```
Get(ring, length)
If count[ring] >= length //enough data in the ring?
ME <-- Scratchpad[head[ring]] // each data phase
head[ring] += length % ringSize
count[ring] -= length
else ME <-- nil // 1 data phase signals read off empty list
```

NOTE: The Microengine signal is delivered with last data. In the case of nil, the signal is delivered with the 1 data phase.

PUT Command

Before issuing a PUT command, it is the responsibility of the Microengine thread issuing the command to make sure the Ring has enough room.

```
Put(ring, length)
SRAM[tail[ring]] <-- ME pull data // each data phase
tail[ring] += length % ringSize
Count[ring] += length
```

Table 82. Head/Tail, Base and Full by Ring Size

Size (# of 32-bit words)	Base Address	Head/Tail Offset	Full Threshold (Entries)
128	13:9	8:2	32
256	13:10	9:2	64
512	13:11	10:2	128
1024	13:12	11:2	256
NOTE: Note that bits [1:0] of the address are assumed to be 00.			

Prior to using the Scratchpad rings, software must initialize the Ring Registers (by CSR writes). The Base address of the ring must be written, and also the size field which determines the number of 32-bit words for the Ring.

Note: Detailed information about CSRs is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

Writes

For an APB or CAP CSR write, the Scratchpad arbitrates for the S_Pull_Bus, pulls the write data from the source identified in the instruction (either a Microengine transfer register or Intel XScale® core write buffer), and puts it into one of the Pull Data FIFOs. It then drives the address and writes data on to the appropriate bus. CAP CSRs locally decode the address to match their own. The Scratchpad generates a separate APB device select signal for each peripheral device (up to 15 devices). If the write is to an APB CSR, the control logic maintains valid signaling until the APB_RDY_H signal is returned (The APB RDY signal is an extension to the APB bus specification specifically added for the IXP Chassis). Upon receiving the APB_RDY_H signal, the APB select signal will be deasserted and the state machine returns to the idle state between commands. The CAP CSR bus does not support a similar acknowledge signal on writes since the Fast Write functionality requires that a write operation be retired each cycle.



For writes using the Reflector mode, Scratchpad arbitrates for the S_Pull_Bus, pulls the write data from the source identified in the instruction (either a Microengine transfer register or Intel XScale® core write buffer), and puts it into one of the Pull Data FIFOs (same as for APB and CAP CSR writes). The data is then removed from the Pull Data FIFO and sent to the Push Arbiter.

For CSR Fast Writes, the command bypasses the Inlet Command FIFO and is acted on at first opportunity. The CSR control logic has an arbiter that gives highest priority to fast writes. If an APB write is in progress when a fast write arrives, both write operations will complete simultaneously. For a CSR fast write, the Scratchpad extracts the write data from the command rather than pulling the data from a source over the Pull bus. It then drives the address and writes data to all CSRs on the CAP CSR bus, using the same method used for the CAP CSR write.

The Scratchpad unit supports CAP write operations with burst counts greater than 1, except for fast writes which only support a burst count of one. Burst support is required primarily for Reflector mode and software must ensure that burst is performed to a non-contiguous set of registers. CAP looks at the length field on the command bus and breaks each count into a separate APB write cycle, incrementing the CSR number for each bus access.

Reads

For an APB read, the Scratchpad drives the address, write, select, and enable signals, and then waits for the acknowledge signal (APB_RDY_H) from APB device. For a CAP CSR read, the address is driven, which controls a tree of multiplexors to select the appropriate CSR. CAP then waits for the acknowledge signal (CAP_CSR_RD_RDY). (Note that the CSR bus can support an acknowledge signal since the read operations occur on a separate read bus and will not interfere with Fast Write operations). In both cases, when the data is returned, the data is sent to the Push Arbiter and the Push Arbiter pushes the data to the destination.

For reads using the Reflector mode, the write data is pulled from the source identified in ADDRESS (either a Microengine transfer register or Intel XScale® core write buffer), and put into one of the Scratchpad Pull Data FIFOs. The data is then sent to the Push Arbiter. The arbiter then moves the data to the destination specified in the command. Note that this is the same as a Reflector mode write, except the source and destination are identified using opposite fields.

The Scratchpad performs one read operation at a time. In other words CAP will not begin a APB read until a CSR read has completed or vice versa. This simplifies the design by ensuring that, when lengths are greater than 1, the data is sent to the Push Arbiter in a contiguous order and not interleaved with data from a read on the other bus.

Signal Done

CAP can provide a signal to a Microengine upon completion of a command. For APB and CAP CSR operations, CAP signals the Microengine using the same method as any other target. For Reflector mode reads and writes, CAP uses the TOKEN field of the Command to determine whether to signal the command initiator, the Microengine that is the target of the reflection, both, or neither

7.1.2.3.3 Clocks and Reset

Clock generation and distribution is handled outside of CAP and is dependent on the specific chip implementation. Separate clock rates are required for CAP CSRs/Push/Pull Buses and ARB since APB devices tend to run slower. CAP provides reset signals for the CAP CSR block and APB devices. These resets are based on the system reset signal and synchronized to the appropriate bus clock.

Table 83 shows the Intel XScale® core and Microengine instructions used to access devices on these buses and it shows which buses are used during the operation. For example, to read an APB peripheral such as a UART CSR, a Microengine would execute a csr[read] instruction and Intel XScale® core would execute a Load (ld) instruction. Data is then moved between the CSR and the Intel XScale® core/Microengine by first reading the CSR via the APB bus and then writing the result to the Intel XScale® core/Microengine via the Push Bus.

Table 83. Intel XScale® Core and Microengine Instructions

Accessing	Read Operation	Write Operation
APB Peripheral	Access Method: Microengine: csr[read] Intel XScale® core: ld	Access Method: Microengine: csr[write] Intel XScale® core: st
	Bus Usages: Read source: APB bus Write dest: Push bus	Bus Usages: Read source: Pull Bus Write dest: APB bus
CAP CSR	Access Method: Microengine: csr[read] Intel XScale® core: ld	Access Method: Microengine: csr[write], fast_wr Intel XScale® core: st
	Bus Usages: Read source: CSR bus Write dest: Push bus	Bus Usages: csr[write] and st Read source: Pull Bus Write dest: CSR bus fast_wr Write dest: CSR bus
Microengine CSR or Xfer Register (Reflector Mode)	Access Method: Microengine: csr[read] Intel XScale® core: ld	Access Method: Microengine: csr[write] Intel XScale® core: st
	Bus Usages: Read source: Pull bus (Address) Write dest: Push bus (PP_ID)	Bus Usages: Reads: Pull Bus (PP_ID) Write dest: Push bus (Address)

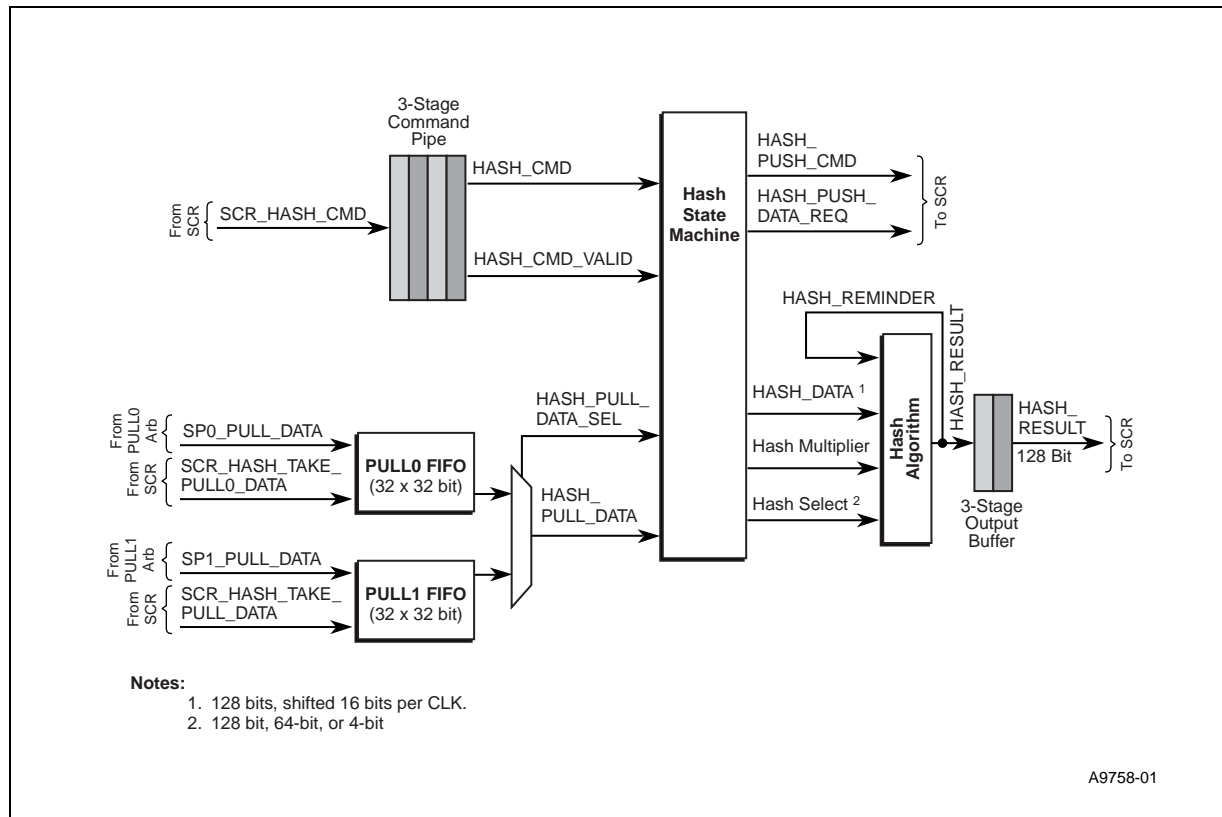
7.1.2.3.4 Reset Registers

The reset registers reside in the SHA-C. For more information on chip reset, refer [Section 10, “Clocks, Reset, and Initialization”](#). Strapping pins will be used to select the reset count (currently 140 cycles after deassert). Options for reset count will be 64 (default), 128, 512, and 2048.

7.1.3 Hash Unit

The SHaC unit contains a Hash Unit that can take 48-bit, 64-bit, or 128-bit data and produces a 48-bit, 64-bit, or a 128-bit hash index, respectively. The Hash Unit is accessible by the MEs and Intel XScale® core. Figure 87 shows a block diagram of the Hash Unit.

Figure 87. Hash Unit Block Diagram



7.1.3.1 Hashing Operation

Up to three hash indexes can be created using a single Microengine instruction. The Microengine hash instructions are shown in [Example 32](#).

Example 32. Microengine Hash Instructions

hash1_48[\$xfer], optional_token	
hash2_48[\$xfer], optional_token	
hash3_48[\$xfer], optional_token	
hash1_64[\$xfer], optional_token	
hash2_64[\$xfer], optional_token	
hash3_64[\$xfer], optional_token	
hash1_128[\$xfer], optional_token	
hash2_128[\$xfer], optional_token	
hash3_128[\$xfer], optional_token	
Where:	
\$xfer	The beginning of a contiguous set of registers that supply the data used to create the hash input and receive the hash index upon completion of the hash operation.
optional_token	sig_done, ctx_swap, defer [1]

A Microengine initiates a hash operation by writing a contiguous set of SRAM Transfer Registers and then executing the hash instruction. The SRAM Transfer Registers can be specified as either Context-Relative, or Indirect; Indirect will allow for any of the SRAM Transfer Register to be used. Two SRAM Transfer Registers are required to create hash indexes for 48-bit and 64-bit and four SRAM Transfer Registers to create 128-bit hash indexes, as shown in [Table 84](#). In the case of the 48-bit hash, the Hash Unit ignores the upper two bytes of the second Transfer Register.

Table 84. S Transfer Registers Hash Operands

Register		Address
48-Bit Hash Operations		
Don't care	hash 3[47:32]	\$xfer n+5
hash 3 [31:0]		\$xfer n+4
Don't care	hash 2[47:32]	\$xfer n+3
hash 2 [31:0]		\$xfer n+2
Don't care	hash 1[47:32]	\$xfer n+1
hash 1 [31:0]		\$xfer n
64-Bit Hash Operations		
hash 3 [63:32]		\$xfer n+5
hash 3 [31:0]		\$xfer n+4
hash 2 [63:32]		\$xfer n+3
hash 2 [31:0]		\$xfer n+2
hash 1 [63:32]		\$xfer n+1



Table 84. S Transfer Registers Hash Operands (Continued)

Register	Address
hash 1 [31:0]	\$xfer n
128-Bit Hash Operations	
hash 3 [127:96]	\$xfer n+11
hash 3 [95:64]	\$xfer n+10
hash 3 [63:32]	\$xfer n+9
hash 3 [31:0]	\$xfer n+8
hash 2 [127:96]	\$xfer n+7
hash 2 [95:64]	\$xfer n+6
hash 2 [63:32]	\$xfer n+5
hash 2 [31:0]	\$xfer n+4
hash 1 [127:96]	\$xfer n+3
hash 1 [64:95]	\$xfer n+2
hash 1 [63:32]	\$xfer n+1
hash 1 [31:0]	\$xfer n

Intel XScale® core initiates a hash operation by writing a set of memory-mapped Hash Operand Registers, which are built in the Intel XScale® core gasket, with the data to be used to generate the hash index. There are separate registers for 48-bit, 64-bit, and 128-bit hashes. Only one hash operation of each type can be done at a time. Writing to the last register in each group informs the gasket logic that it has all the operands for that operation, and it will then arbitrate for Command bus to send the command to the Hash Unit.

Note: Detailed information about CSRs is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

For both Microengine generated commands and Intel XScale® core generated commands, the command enters the Command Inlet FIFO. As with the Scratchpad write and RMW operations, signals are generated and sent to the Pull Arbiter. The Hash unit Pull Data FIFO allows the data for up to three hash operations to be read into the Hash Unit in a single burst. When the command is serviced, the first data to be hashed enters the hash array while the next two wait in the FIFO.

The Hash Unit uses a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indexes. Three separate multipliers are supported, one for 48-bit hash operations, one for 64-bit hash operations and one for 128-bit hash operations. The multiplier is programmed through registers (HASH_MULTIPLIER_64_1, HASH_MULTIPLIER_64_2, HASH_MULTIPLIER_48_1, HASH_MULTIPLIER_48_2, HASH_MULTIPLIER_128_1, HASH_MULTIPLIER_128_2, HASH_MULTIPLIER_128_3, HASH_MULTIPLIER_128_4).

The multiplicand is shifted into the hash array sixteen bits at a time. The hash array performs a ones complement multiply and polynomial divide, calculated using the multiplier and 16 bits of the multiplicand. The result is placed into an output register and also feeds back into the array. This process is repeated 3 times for a 48-bit hash (16 bits x 3 = 48), 4 times for a 64-bit hash (16 bits x 4 = 64) and 8 times for a 128-bit hash (16 x 8 = 128). After an entire multiplicand has been passed through the hash array, the resulting hash index is placed into a two-stage output pipeline and the next hash is immediately started.

The Hash Unit shares the Scratchpad's Push Data FIFO. After each hash index is completed, the index is placed into a three-stage output pipe and the Hash Unit sends a PUSH_DATA_REQ to the Scratchpad to indicate that it has a valid hash index to put into the Push Data FIFO for transfer. The Scratchpad will issue a SEND_HASH_DATA signal, transfers the hash index to the Push Data FIFO, and sends the data to the Arbiter.

For Intel XScale® core initiated hash operations, Intel XScale® core reads the results from its memory-mapped Hash Result Registers. The addresses of Hash Results are the same as the Hash Operand Registers. Because of queuing delays at the Hash Unit, the time to complete an operation is not fixed. Intel XScale® core can do one of two operations to get the hash results.

- Poll the Hash Done Register. This register is cleared when the Hash Operand Registers are written. Bit [0] of Hash Done Register is set when the Hash Result Registers get the return result from the Hash Unit (when the last word of the result is returned). Intel XScale® core software can poll on Hash Done, and read Hash Result when Hash Done is equal to 0x00000001.
- Read Hash Result directly. The gasket logic will acknowledge the read only when the result is valid. This method will result in Intel XScale® core stalling if the result is not valid when the read happens.

The number of clock cycles required to perform a single hash operation equals: two or four cycles through the input buffers, three, four, or eight cycles through the hash array, and two or four cycles through the output buffers. With the pipeline characteristics of the Hash Unit, performance is improved if multiple hash operations are initiated with a single instruction rather than separate hash instructions for each hash operation.

7.1.3.2 Hash Algorithm

The hashing algorithm used by allows flexibility and uniqueness since it can be programmed to provide different results for a given input. The algorithm uses binary polynomial multiplication and division under modulo-2 addition. The input to the algorithm is a 48-bit, 64-bit, or 128-bit value.

The data used to generate the hash index is considered to represent the coefficients of an order-47, order-63 or order-127 polynomial in x. The input polynomial (designated as A(x)) has the form:

Equation 1. $A_{48}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{46}x^{46} + a_{47}x^{47}$ (48-bit hash operation)

Equation 2. $A_{64}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{62}x^{62} + a_{63}x^{63}$ (64-bit hash operation)

Equation 3. $A_{128}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{126}x^{126} + a_{127}x^{127}$ (128-bit hash operation)

This polynomial is multiplied by a programmable hash multiplier using a modulo-2 addition. The hash multiplier, M(x) is stored in Hash Unit CSRs and represents the polynomial

Equation 4. $M_{48}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{46}x^{46} + m_{47}x^{47}$ (48-bit hash operation)

Equation 5. $M_{64}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{62}x^{62} + m_{63}x^{63}$ (64-bit hash operation)

Equation 6. $M_{128}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{126}x^{126} + m_{127}x^{127}$ (128-bit hash operation)

Since multiplication is performed using modulo-2 addition, the result is an order-94 polynomial, an order-126 polynomial or an order-254 polynomial with coefficients that are also 1 or 0. This product is divided by a fixed generator polynomial given by:



Equation 7. $G_{48}(x) = 1 + x^{10} + x^{25} + x^{36} + x^{48}$ (48-bit hash operation)

Equation 8. $G_{64}(x) = 1 + x^{17} + x^{35} + x^{54} + x^{64}$ (64-bit hash operation)

Equation 9. $G_{128}(x) = 1 + x^{33} + x^{69} + x^{98} + x^{128}$ (128-bit hash operation)

The division results in a quotient $Q(x)$, a polynomial of order-46, order-62 or order-126, and a remainder $R(x)$, a polynomial of order-47, order-63 or order-127. The operands are related by the equation:

Equation 10. $A(x)M(x) = Q(x)G(x) + R(x)$

The generator polynomial has the property of irreducibility. As a result, for a fixed multiplier $M(x)$, there is a unique remainder $R(x)$ for every input $A(x)$. The quotient $Q(x)$, can then be then discarded, since input $A(x)$ can be derived from its corresponding remainder $R(x)$. A given bounded set of input values $A(x)$ (for example, 8K or 16K table entries), with bit weights of an arbitrary density function can be mapped one-to-one into a set of remainders $R(x)$ such that the bit weights of the resulting Hashed Arguments (a subset of all values of $R(x)$ polynomials) are all about equal.

In other words, there is a high likelihood that the low order set of bits from the Hash Arguments are unique, so they can be used to build an index into the table. If the hash algorithm does not provide a uniform hash distribution for a given set of data, the programmable hash multiplier ($M(x)$) may be modified to provide better results.



Media and Switch Fabric Interface

8

8.1 Overview

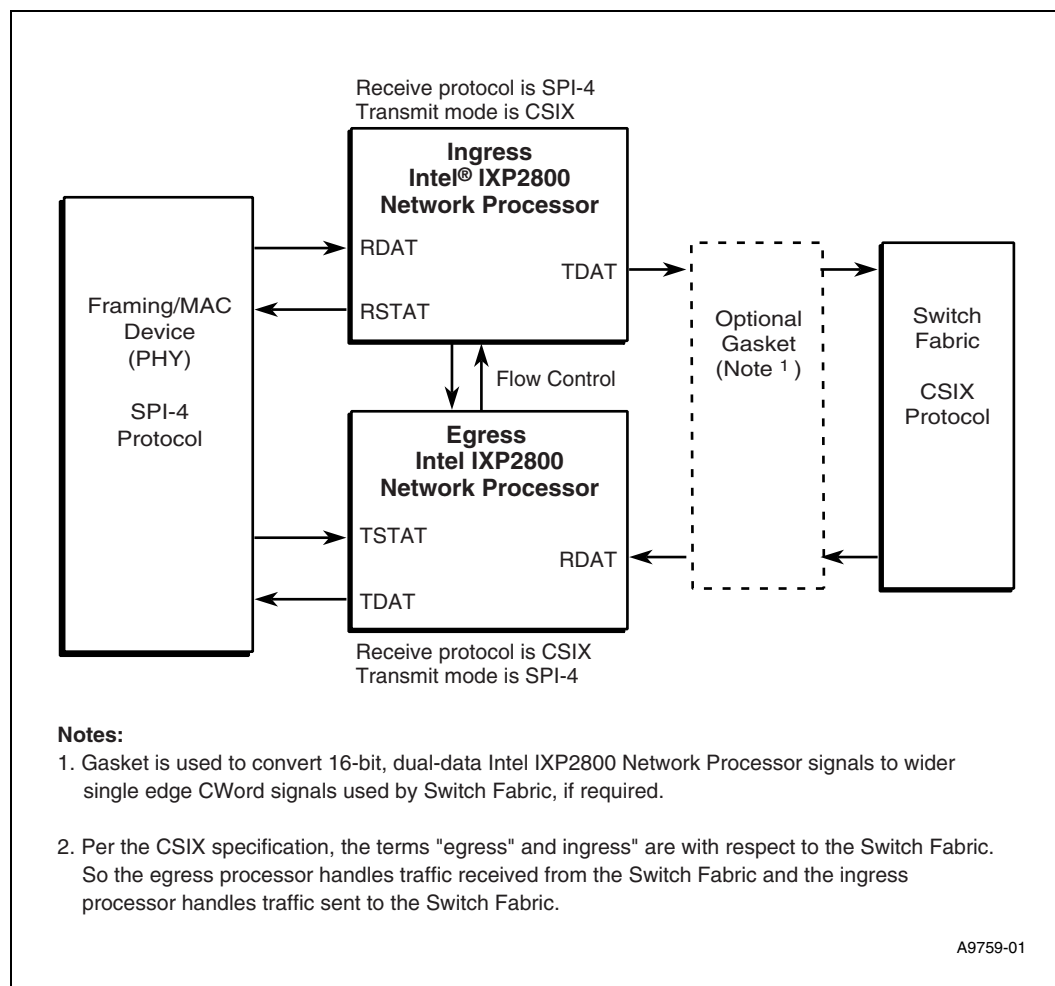
The Media and Switch Fabric (MSF) Interface connects the IXP2800 Network Processor to a physical layer device (PHY) and/or to a Switch Fabric. MSF consists of separate receive and transmit interfaces. Each of the receive and transmit interfaces can be separately configured for either SPI-4 Phase 2 (System Packet Interface) for PHY devices or CSIX-L1 protocol for Switch Fabric Interfaces.

The receive and transmit ports are unidirectional and independent of each other. Each port has 16 data signals, a clock, a control signal, and a parity signal, all of which use LVDS (differential) signaling, and are sampled on both edges of the clock. There is also a flow control port consisting of a clock, data, and ready status bits, and used to communicate between two IXP2800 Network Processors, or a IXP2800 Network Processor and a Switch Fabric Interface. These are also LVDS, dual-edge data transfer.

The usage of the signals, as well as the receive and transmit functions, are shown in the block diagram in [Figure 88](#), and described below.

Note: Detailed information about CSRs is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

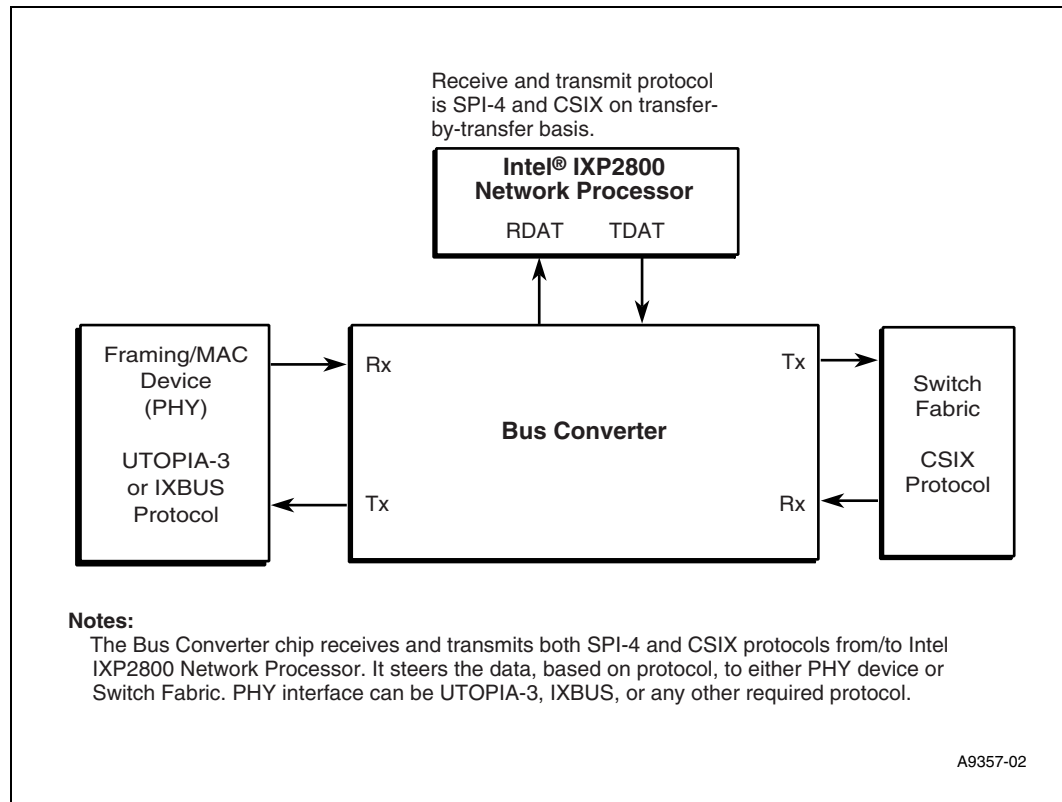
Figure 88. Example System Block Diagram



The use of some of the receive and transmit pins is based on protocol, SPI-4 or CSIX. For the LVDS pins, only the active high name is given (for LVDS there are two pins per signal). The definitions of the pins can be found in the SPI-4 and CSIX specs, referenced below.

An alternate system configuration is shown in the block diagram in [Figure 89](#). In this case a single IXP2800 Network Processor is used for both Ingress and Egress. The bit rate supported would be less than in [Figure 88](#). A hypothetical Bus Converter chip, external to the IXP2800 Network Processor is used. The block diagram in [Figure 89](#) is only an illustrative example.

Figure 89. Full-Duplex Block Diagram



8.1.1 SPI-4

SPI-4 is an interface for packet and cell transfer between a physical layer (PHY) device and a link layer device (the IXP2800 Network Processor), for aggregate bandwidths of OC-192 ATM and Packet over SONET/SDH (POS), as well as 10 Gb/s Ethernet applications.

The Optical Internetworking Forum (OIF), www.oiforum.com, controls the SPI-4 Implementation Agreement document.

SPI-4 has two types of transfers—Data when the **RCTL** signal is deasserted; Control when the **RCTL** signal is asserted. The Control Word format is shown in [Table 85](#) (this information is from SPI-4 specification, shown here for convenience).

Table 85. SPI-4 Control Word Format

Bit Position	Label	Description
15	Type	Control Word Type. <ul style="list-style-type: none"> 1: payload control word (payload transfer will immediately follow the control word). 0: idle or training control word.
14:13	EOPS	End-of-Packet (EOP) Status. Set to the following values below according to the status of the immediately preceding payload transfer. <ul style="list-style-type: none"> 00: Not an EOP. 01: EOP Abort (application-specific error condition). 10: EOP Normal termination, 2 bytes valid. 11: EOP Normal termination, 1 byte valid. EOPS is valid in the first Control Word following a burst transfer. It is ignored and set to "00" otherwise.
12	SOP	Start-of-Packet. Set to 1 if the payload transfer immediately following the Control Word corresponds to the start of a packet. Set to 0 otherwise. Set to 0 in all idle and training control words.
11:4	ADR	Port Address. 8-bit port address of the payload data transfer immediately following the Control Word. None of the addresses are reserved (all are available for payload transfer). Set to all zeroes in all idle Control Words. Set to all ones in all training Control Words.
3:0	DIP-4	4-bit Diagonal Interleaved Parity. 4-bit odd parity computed over the current Control Word and the immediately preceding data words (if any) following the last Control Word.

Control words are inserted only between burst transfers; once a transfer has begun, data words are sent uninterrupted until either End of Packet or a multiple of 16 bytes is reached.

The order of bytes within the SPI-4 data burst is shown in [Table 86](#). The most significant bits of the bytes correspond to bits 15 and 7. On data transfers that do not end on an even byte boundary, the unused byte on bits [7:0] is set to all zeros.

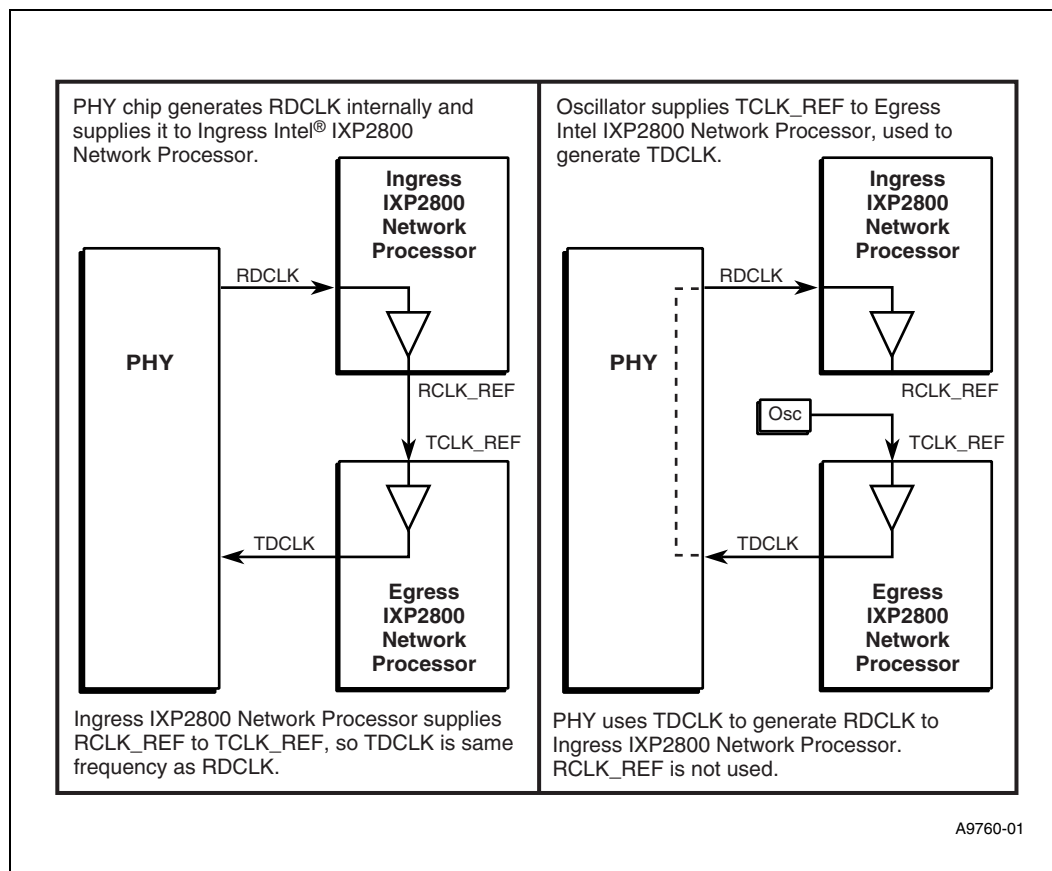
Table 86 shows the order of bytes on SPI-4; this example shows a 43 byte packet.

Table 86. Order of Bytes within the SPI-4 Data Burst

	Bit 15	Bit 8	Bit 7	Bit 0
Data Word 1	Byte 1		Byte 2	
Data Word 2	Byte 3		Byte 4	
Data Word 3	Byte 5		Byte 5	
Data Word 4	Byte 7		Byte 6	
...	
...	
...	
Data Word 21	Byte 41		Byte 42	
Data Word 22	Byte 43		00	

Figure 90 shows two ways in which the SPI-4 clocking can be done. Note that it is also possible to use an internally supplied clock and leave **TCLK_REF** unused.

Figure 90. Receive and Transmit Clock Generation



8.1.2 CSIX

CSIX_L1 (Common Switch Interface) defines an interface between a Traffic Manager (TM) and a Switch Fabric (SF) for ATM, IP, MPLS, Ethernet, and similar data communications applications.

The Network Processor Forum (NPF) www.npforum.org, controls the CSIX_L1 specification.

The basic unit of information transferred between TMs and SFs is called a CFrame. There are a number of CFrame types defined as shown in [Table 87](#).

Table 87. CFrame Types

Type Encoding	CFrame Type
0	Idle
1	Unicast
2	Multicast Mask
3	Multicast ID
4	Multicast Binary Copy
5	Broadcast
6	Flow Control
7	Command and Status
8-F	CSIX Reserved

For transmission from the IXP2800 Network Processor, CFrames are constructed for transmit under Microengine software control, and written into the Transmit Buffer (TBUF).

On receive to the IXP2800 Network Processor CFrames are either discarded, placed into Receive Buffer (RBUF), or placed into Flow Control Egress FIFO (FCEFIFO), according to mapping defined in **CSIX_Type_Map** CSR. CFrames put into RBUF are passed to a Microengine to be parsed by software. CFrames put into FCEFIFO are sent to the Ingress IXP2800 Network Processor over the Flow Control bus. Link-level Flow Control information (CSIX Ready field) in the Base Header of all CFrames (including Idle) is handled by hardware.

8.1.3 CSIX/SPI-4 Interleave Mode

SPI4 packets and CSIX cframes are interleaved when the RBUF and TBUF are configured in 3-partition mode. When the protocol signal RPROT or TPROT is high, the data bus is transferring CSIX CFRAMEs or IDLE cycles. When protocol is low, the data bus is transferring SPI-4 packets or idle cycles. When operating in interleave mode, RPROT must be driven high (logic 1) for the entire CSIX CFRAME or low (logic 0) for the entire SPI4 burst. When in 3-partition mode, the SPI-4 interval should be padded using SPI-4 idle cycles so that it ends on a 32 bit boundary or a complete RCLK or TCLK clock cycle. The actual SPI-4 data length can be any size. However, the SPI-4 interval which includes the SPI-4 control words and payload data must end on a 32-bit boundary.

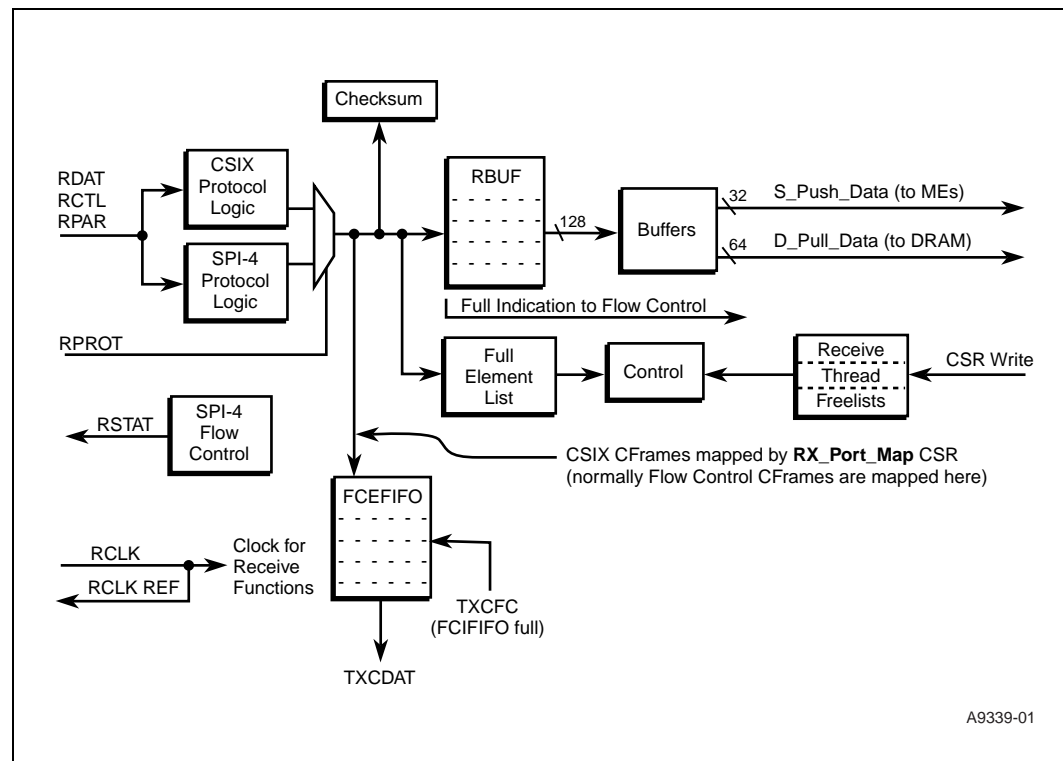
8.2 Receive

The receive section consists of:

- Receive Pins (Section 8.2.1)
- Checksum (Section 8.2.2)
- Receive Buffer (RBUF) (Section 8.2.2)
- Full Element List (Section 8.2.3)
- Rx_Thread_Freelist (Section 8.2.4)
- Flow Control Status (Section 8.2.7)

Figure 91 is a simplified block diagram of the receive section.

Figure 91. Simplified Receive Section Block Diagram



8.2.1 Receive Pins

The use of the receive pins is a function of **RPROT** input, as shown in [Table 88](#).

Table 88. Receive Pins Usage by Protocol

Name	Direction	SPI-4 Use	CSIX Use
RCLK	Input	RDCLK	TxCk
RDAT[15:0]	Input	RDAT[15:0]	TxData[15:0]
RCTL	Input	RCTL	TxSOF
RPAR	Input	Not Used	TxPar
RSCLK	Output	RSCLK	Not Used
RSTAT[1:0]	Output	RSTAT[1:0]	Not Used

In general, hardware does framing, parity checking, and flow control message handling. Interpretation of frame header and payload data is done by Microengine software.

The internal clock used is taken from **RCLK** pin. **RCLK_Ref** output is a buffered version of the clock. It can be used to supply **TCLK_Ref** of the Egress IXP2800 Network Processor if desired.

The receive pins **RDAT[15:0]**, **RCTL**, **RPAR** are sampled relative to **RCLK**. In order to work at high frequencies, each of those pins has de-skewing logic as described in [Section 8.6](#).

8.2.2 RBUF

RBUF is a RAM that holds received data. It stores received data in sub-blocks (referred to as elements), and is accessed by Microengine or the Intel XScale® core reading the received information. Details of how RBUF elements are allocated and filled is based on the receive data protocol, and is described in [Section 8.2.2.1–Section 8.2.2.2](#). When data is received the associated status is put into the **Full_Element_List** FIFO and subsequently sent to Microengine to process. **Full_Element_List** insures that received elements are sent to Microengine in the order that the data was received.

RBUF contains a total of 8 Kbyte of data. [Table 89](#) shows the order in which received data is stored in RBUF. Each number represents a byte, in order of arrival from the receiver interface.

Table 89. Order in Which Received Data Is Stored in RBUF

Data/Payload								Address Offset (Hex)
4	5	6	7	0	1	2	3	0
C	D	E	F	8	9	A	B	8
14	15	16	17	10	11	12	13	10

The mapping of elements to address offset in RBUF is based on the RBUF partition and element size, as programmed in **MSF_Rx_Control** CSR. RBUF can be partitioned into one, two, or three partitions based on **MSF_Rx_Control[RBUF_Partition]**. The mapping of received data to partitions is shown in [Table 90](#).

Table 90. Mapping of Received Data to RBUF Partitions

Number of Partitions in Use	Receive Data Protocol	Data Use by Partition, Fraction of RBUF Used, Start Byte Offset (Hex)		
		Partition Number		
		0	1	2
1	SPI-4 only	SPI-4 All Byte 0	n/a	n/a
2	CSIX only	CSIX Data ¾ of RBUF Byte 0	CSIX Control ¼ of RBUF Byte 0x1800	n/a
3	Both SPI-4 and CSIX	CSIX Data ½ of RBUF Byte 0	SPI-4 3/8 of RBUF Byte 0x1000	CSIX Control 1/8 of RBUF Byte 0x1C00

The data in each partition is further broken up into elements, based on **MSF_Rx_Control[RBUF_Element_Size_#]** (n = 0,1,2). There are three choices of element size, 64, 128, or 256 bytes.

Table 91 shows the RBUF partition options. Note that the choice of element size is independent for each partition.

Table 91. Number of Elements per RBUF Partition

RBUF_Partition Field	RBUF_Element_Size_# Field	Partition Number		
		0	1	2
00 (1 partition)	00 (64 byte)	128	Unused	Unused
	01 (128 byte)	64		
	10 (256 byte)	32		
01 (2 partitions)	00 (64 byte)	96	32	Unused
	01 (128 byte)	48	16	
	10 (256 byte)	24	8	
10 (3 partitions)	00 (64 byte)	64	48	16
	01 (128 byte)	32	24	8
	10 (256 byte)	16	12	4

Microengine can read data from the RBUF to Microengine S_Transfer_In registers using the `msf[read]` instruction, where they specify the starting byte number (which must be aligned to 4-byte units), and number of 32-bit words to read. The number in the instruction can be either the number of 32-bit words, or number of 32-bit word pairs, using the single and double instruction modifiers, respectively. The data is pushed to the Microengine on the S Push Bus by RBUF control logic.

`msf[read, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token`



The `src_op_1` and `src_op_2` operands are added together to form the address in RBUF (note that the base address of the RBUF is 0x2000). `ref_cnt` is the number of 32-bit words or word pairs, which are pushed into two sequential `S_Transfer_In` registers, starting with `$s_xfer_reg`.

Using the data in RBUF in [Table 89](#) above, reading 8 bytes from offset 0 into transfer registers 0 and 1 would yield the result in [Example 33](#).

Example 33. Data from RBUF Moved to Microengine Transfer Registers

Transfer Register Number	Bit Number within Transfer Register							
	31	24	23	16	15	8	7	0
0	0		1		2		3	
1	4		5		6		7	

Microengine can move data from RBUF to DRAM using the instruction:

```
dram[rbuf_rd, --, src_op1, src_op2, ref_cnt], indirect_ref
```

The `src_op_1` and `src_op_2` operands are added together to form the address in DRAM, so the `dram` instruction must use `indirect_ref` modifier to specify the RBUF address (refer to the IXP2800 Network Processor Chassis chapter for details). `ref_cnt` is number of 64-bit words which are read from RBUF.

Using the data in RBUF in [Table 89](#) above, reading 16 bytes from offset 0 in RBUF into DRAM would yield the result in [Example 34](#) in DRAM. [Note that DRAM addresses must be aligned to 8-byte units. The data from lower offset RBUF offsets goes into lower addresses in DRAM.]

Example 34. Data from RBUF Moved to DRAM

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0
4		5		6		7		0		1		2		3	
C		D		E		F		8		9		A		B	

For both types of RBUF read, reading an element does not modify any RBUF data, and does not free the element, so buffered data can be read as many times as desired.

8.2.2.1 SPI-4

SPI-4 data is placed into RBUF as follows:

At chip reset all elements are marked invalid (available).

When a SPI-4 Control Word is received (i.e., when `RCTL` is asserted) it is placed in a temporary holding register. The Checksum accumulator is cleared. The subsequent action is based on the Type field.

If Type is Idle or Training the Control Word is discarded.

If Type is not Idle or Training:

An available RBUF element is allocated by receive control logic. [If there is not an available element the data is discarded and **MSF Interrupt Status[RBUF_Overflow]** is set. Note that this normally should not happen because when number of RBUF elements falls below a programmed high water mark, flow control status is sent back to the PHY device. Refer to

[Section 8.2.7.1.](#)] The SPI-4 Control Word Type, EOPS, SOP, and ADR fields are placed into a temporary status register. The Byte_Count field of the element status is set to 0x0. As each Data Word is received the data is written into the element, starting at offset 0x0 in the element and Byte_Count is updated. Subsequent Data transfers are placed at higher offsets (i.e., 0x2, 0x4, etc.). The 16-bit Checksum Accumulator is also updated with the ones complement addition of each byte pair. [Note if the data transfer has an odd number of bytes, a byte of zeroes is appended as the more significant byte before the checksum addition is done.]

If a Control Word is received before the element is full — the element is marked valid. EOP for the element is taken from the value of the EOPS field (see [Table 85](#)) from the just received Control Word. If the EOPS field from the just received Control Word indicates that EOP is asserted, Byte_Count for the element is decremented by 0 or 1 according to the EOPS field (i.e., decrement by 0 if 2 bytes valid, by 1 if 1 byte valid). If the EOPS field indicates Abort, Byte_Count is rounded up to the next multiple of 4. The temporary status register value is put into Full_Element_List.

If the element becomes full before receipt of another Control Word — the element is marked as pre-valid. The eventual status is based on the next SPI-4 transfer(s).

If the next transfer is a Data Word — the previous element is changed from pre-valid to valid. The EOP for the element is 0. The temporary status register value is put into Full_Element_List. Another available RBUF element is allocated, and the new data is written into it. The temporary status for the new element gets the same ADR field of the previous element, and SOP is set to 0. Status word Byte_Count field is set to 0x2, and will count up as more Data Words arrive. Checksum Accumulator is cleared.

If the next transfer is a Control Word — the previous element is changed from pre-valid to valid. EOP for the element is taken from the value of the EOPS field from the just received Control Word. If the EOPS field from the just received Control Word indicates that EOP is asserted, Byte_Count for the element is decremented by 0 or 1 according to the EOPS field (i.e., decrement by 0 if 2 bytes valid, by 1 if 1 byte valid). The temporary status register value is put into Full_Element_List.

Data received from the bus is placed into the element lowest offset first in big-endian order (that is, with the first byte received in the most significant byte of the 32-bit word, etc.).

The status contains the following information:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
RPROT	Element							Byte Count							SOP	EOP	Err	Len Err	Par Err	Abort Err	Null	Type	ADR								

6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
Reserved																Checksum															

The definitions of the fields are shown in [Table 92](#).

Table 92. RBUF SPIF-4 Status Definition

Field	Definition
RPROT	This bit is a 0 indicating that the Status is for SPI-4. It is derived from the RPROT input signal.
Null	Null receive. If this bit is set, it means that the Rx_Thread_Freelist timeout expired before any more data was received, and that a null Receive Status Word is being pushed in order to keep the receive pipeline flowing. The rest of the fields in the Receive Status Word must be ignored; there is no data or RBUF entry associated with a null Receive Status Word.
ADR	The port number to which the data is directed. This field is taken from the ADR field of the Control Word that most recently preceded the data transfer.
Type	This field is taken from the Type field of the Control Word that most recently preceded the data transfer.
SOP	Indicates if the element is the start of a packet. This field is taken from the SOP field of the Control Word that most recently preceded the data transfer for the first element allocated after a Control Word. For subsequent elements (i.e., if more than one element worth of data follow the Control Word) this value is 0.
EOP	Indicates if the element is the end of a packet. This field is taken from the EOPS field of the Control Word that most recently succeeded the data transfer.
Byte_Count	Indicates the number of Data bytes, from 1 to 256, in the element (value 0x00 means 256). This field is derived from the number of data transfers that fill the element, and also the EOPS field of the Control Word that most recently succeeded the data transfer.
Element	The element number in the RBUF that holds the data. This is equal to the offset in RBUF of the first byte in the element, shifted right by six places
Par Err	Parity Error was detected in the DIP-4 parity field. See description in Section 8.2.8.1 .
Length Err	A non-EOP burst occurred that was not a multiple of 16 bytes.
Abort Err	An EOP with Abort was received on bits[14:13] of the Control Word that most recently succeeded the data transfer.
Err	Error. This is the logical OR of Par Err, Length Err, and Abort Err.
Checksum	Checksum calculated over the Data Words in the element. This can be used for TCP.

8.2.2.2 CSIX

CSIX CFrames are placed into either RBUF or FCEFIFO as follows:

At chip reset all RBUF elements are marked invalid (available) and FCEFIFO is empty.

When a Base Header is sent (i.e., when **RxSof** is asserted) it is placed in a temporary holding register. The Ready Field is extracted and held to be put into **FC_Egress_Status** CSR when (and if) the entire CFrame is received without error. The Type field is extracted and used to index into **CSIX_Type_Map** CSR to determine one of four actions.

- Discard (except for the Ready Field as described in [Section 8.2.7.2.1](#)).
- Place into RBUF Control CFrame partition.
- Place into RBUF Data CFrame partition.
- Place into FCEFIFO.

Note: Normally Idle CFrames (Type 0x0) will be discarded, Command and Status CFrames (Type 0x7) will be placed into Control Partition, Flow Control CFrames (Type 0x6) will be placed into FCEFIFO, and all others will be placed into Data Partition (see [Table 89](#)). The remapping done through **CSIX_Type_Map** CSR allows for more flexibility in usage if desired.

If the action is Discard the CFrame is discarded (except for the Ready Field as described in [Section 8.2.7.2.1](#)). The Base Header, as well as Extension Header and Payload (if any) are discarded.

If the destination is FCEFIFO:

The Payload is placed into the FCEFIFO, to be sent to the Ingress IXP2800 Network Processor over the **TXCDAT** pins. If there is not enough room in FCEFIFO for the entire CFrame, based on the Payload Size in the Base Header, the entire CFrame is discarded and **MSF_Interrupt_Status[FCEFIFO_Overflow]** is set.

If the destination is RBUF (either Control or Data):

An available RBUF element of the corresponding type is allocated by receive control logic. If there is not an available element the CFrame is discarded and **MSF_Interrupt_Status[RBUF_Overflow]** is set. Note that this normally should not happen because when number of RBUF elements falls below a programmed high water mark, back pressure is sent to the Switch Fabric. Refer to [Section 8.2.7.2](#).] The Type, Payload Length, CR (CSIX Reserved) and P (Private) bits, and (subsequently arriving) Extension Header are placed into a temporary status register. As the Payload (including padding if any) is received, it is placed into the allocated RBUF element, starting at offset 0x0. [Note—it is more exact to state that the first four bytes after the Base Header are placed into the status register as Extension Header. For Flow Control CFrames, there is no Extension Header; the first four bytes are part of the Payload. They would be found in the Extension Header field of the Status—no bytes are lost.]

When all of the Payload data (including padding if any), as indicated by the Payload Length field, and Vertical Parity has been received, the element is marked valid. If another **RxSof** is received prior to receiving the entire Payload, the element is also marked valid, and the Length Error status bit is set. If the Payload Length field of the Base Header is greater than the element size (as configured in **MSF_Rx_Control[RBUF_Element_Size]**, then the Length Error bit in the status will be set, and all payload bytes above the element size will be discarded.] The temporary status register value is put into **Full_Element_List**.

Note: In CSIX protocol, an RBUF element is allocated only on **RxSof** assertion. Therefore the element size must be programmed based on the Switch Fabric usage. For example, if the switch never sends a payload greater than 128 bytes, 128-byte elements can be selected. Otherwise, 256-byte elements must be selected.



Data received from the bus is placed into the element lowest offset first in big-endian order (that is, with the first byte received in the most significant byte of the 32-bit word, etc.).

The status contains the following information:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
RPROT	Element							Payload Length							CR	P	Err	Len Err	HP Err	VP Err	Null	Reserved					Type				
6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
Extension Header																															

The definitions of the fields are shown in [Table 93](#).

Table 93. RBUF CSIX Status Definition

Field	Definition
RPROT	This bit is a 1 indicating that the Status is for CSIX-L1. It is derived from the RPROT input signal.
Null	Null receive. If this bit is set, it means that the Rx_Thread_Freelist timeout expired before any more data was received, and that a null Receive Status Word is being pushed in order to keep the receive pipeline flowing. The rest of the fields in the Receive Status Word must be ignored; there is no data or RBUF entry associated with a null Receive Status Word.
Type	Type Field from the CSIX Base Header
Payload Length	Payload Length Field from the CSIX Base Header. A value of 0x0 indicates 256 bytes.
VP Err	Vertical Parity Error was detected on the CFrame. See description in Section 8.2.8.2.2 .
HP Err	Horizontal Parity Error was detected on the CFrame. See description in Section 8.2.8.2.1 .
Length Err	Length Error; either amount of Payload received (before receipt of next Base Header) did not match value indicated in Base Header Payload Length field) or Payload Length field was greater than size of RBUF element.
Err	Error. This is the logical OR of VP Err, HP Err, and Length Err.
Element	The element number in the RBUF that holds the data. This is equal to the offset in RBUF of the first byte in the element, shifted right by 6 places.
CR	CR (CSIX Reserved) bit from the CSIX Base Header.
P	P (Private) bit from the CSIX Base Header.
Extension Header	The Extension Header from the CFrame. The bytes are received in big-endian order; byte 0 is in bits 63:56, byte 1 is in bits 55:48, byte 2 is in bits 47:40, and byte 3 is in bits 39:32.

8.2.3 Full Element List

Receive control hardware maintains the Full Element List to hold the status of valid RBUF elements, in the order in which they were received. When an element is marked valid (as described in [Section 8.2.2.1](#) for SPI-4 and [Section 8.2.2.2](#) for CSIX), its status is added to the tail of the Full Element List. When a Microengine is notified of element arrival (by having the status written to its S_Transfer register; see [Section 8.2.4](#)), it is removed from the head of the Full Element List.

8.2.4 Rx_Thread_Freelist_#

Each **Rx_Thread_Freelist_#** is a FIFO that indicates Microengine Contexts that are awaiting an RBUF element to process. This allows the Contexts to indicate their ready status prior to the reception of the data, as a way to eliminate latency. Each entry added to a Freelist also has an associated S_Transfer register and signal number. The receive logic maintains either one, two, or three separate lists based on **MSF_Rx_Control[RBUF_Partition]**, **MSF_Rx_Control[CSIX_Freelist]**, and **Rx_Port_Map** as shown in [Table 94](#).

Table 94. Rx_Thread_Freelist Use

Number of Partitions ¹	Use	CSIX_Freelist ²	Rx_Thread_Freelist_# Used		
			0	1	2
1	SPI-4 only	n/a	SPI-4 Ports equal to or below Rx_Port_Map	SPI-4 Ports above Rx_Port_Map	Not Used
2	CSIX only	0	CSIX Data	CSIX Control	Not Used
		1	CSIX Data and CSIX Control	Not Used	Not Used
3	Both SPI-4 and CSIX	0	CSIX Data	SPI-4	CSIX Control
		1	CSIX Data and CSIX Control	SPI-4	Not Used
1. Programmed in MSF_Rx_Control[RBUF_Partition] . 2. Programmed in MSF_Rx_Control[CSIX_Freelist] .					

To be added as ready to receive an element, an Microengine does a `msf[write]` or `msf[fast_write]` to the **Rx_Thread_Freelist_#** address; the write data is the Microengine/Context/S_Transfer Register number to add to the Freelist. Note that using the data (rather than the command bus ID) permits a Context to add either itself or other Contexts as ready.

When there is valid status at the head of the Full Element List it will be pushed to a Microengine. The receive control logic pushes the status information (which includes the element number) to the Microengine in the head entry of **Rx_Thread_Freelist_#**, and sends an Event Signal to the Microengine. It then removes that entry from the **Rx_Thread_Freelist_#**, and removes the status from Full Element List. [Note that this implies the restriction—a Context waiting on status must not read the S_Transfer register until it has been signaled.] See [Section 8.2.6](#) for more detail. In the event that **Rx_Thread_Freelist_#** is empty, valid status will be held in Full Element List until an entry is put into **Rx_Thread_Freelist_#**.

8.2.5 Rx_Thread_Freelist_Timeout_#

Each **Rx_Thread_Freelist_#** has an associated countdown timer. If the timer expires and no new receive data is available yet, the receive logic will autopush a Null Receive Status Word to the next thread on the **Rx_Thread_Freelist_#**. A Null Receive Status Word has the “Null” bit set, and does not have any data or RBUF entry associated with it.

The **Rx_Thread_Freelist_#** timer is useful for certain applications. Its primary purpose is to keep the receive processing pipeline (implemented as microcode running on the Microengine) moving even when the line has gone idle. It is especially useful if the pipeline is structured to handle mpackets in groups, i.e., eight mpackets at a time. If seven mpackets are received, then the line goes idle, then the timeout will trigger the autopush of a null Receive Status Word, filling the eighth slot and allowing the pipeline to advance. Another example is if one valid mpacket is received before the line goes idle for a long period; seven null Receive Status Words will be autopushed, allowing the pipeline to proceed. Typically the timeout interval is programmed to be slightly larger than the minimum arrival time of the incoming cells or packets.

The timer is controlled using the **Rx_Thread_Freelist_Timeout_#** CSR. The timer may be enabled or disabled, and the timeout value specified using this CSR.

The following rules define the operation of the **Rx_Thread_Freelist** timer.

1. Writing a non-zero value to the **Rx_Thread_Freelist_Timeout_#** CSR both resets the timer and enables it. Writing a zero value to this CSR resets the timer and disables it.
2. If the timer is disabled, then only valid (non-null) Receive Status Words are autopushed to the receive threads; null Receive Status Words are never pushed.
3. If the timer expires and the **Rx_Thread_Freelist_#** is non-empty, but there is no mpacket available, this will trigger the autopush of a null Receive Status Word.
4. If the timer expires and the **Rx_Thread_Freelist_#** is empty, the timer stays in the EXPIRED state and is not restarted. A null Receive Status Word cannot be autopushed, since the logic has no destination to push anything to.
5. An expired timer is reset and restarted if and only if an autopush, null or non-null, is performed.
6. Whenever there is a choice, autopush of a non-null Receive Status Word takes precedence over a null Receive Status Word.

8.2.6 Receive Operation Summary

During receive processing received Cframes, cells and packets (which in this context are all called mpackets) are placed into the RBUF, and then, when marked valid, are immediately handed off to a Microengine to process. Normally, by application design, some number of Microengine Contexts will be assigned to receive processing. Those Contexts will have their number added to the proper **Rx_Thread_Freelist_#** (via `msf[write]` or `msf[fast_write]`), and then will go to sleep to wait for arrival of an mpacket (or alternatively poll waiting for arrival of an mpacket).

When an mpacket becomes valid as described in [Section 8.2.2.1](#) for SPI-4 and [Section 8.2.2.2](#) for CSIX, receive control logic will autopush 8 bytes of information for the element to the Microengine/Context/S_Transfer Registers at the head of **Rx_Thread_Freelist_#**. The information pushed is (see [Table 92](#) and [Table 93](#) for detailed definitions):

- Status Word (SPI-4) or Header Status (CSIX) to Transfer Register n (n is the Transfer Register programmed to the **Rx_Thread_Freelist_#**)
- Checksum (SPI-4) or Extension Header (CSIX) to Transfer Register n+1

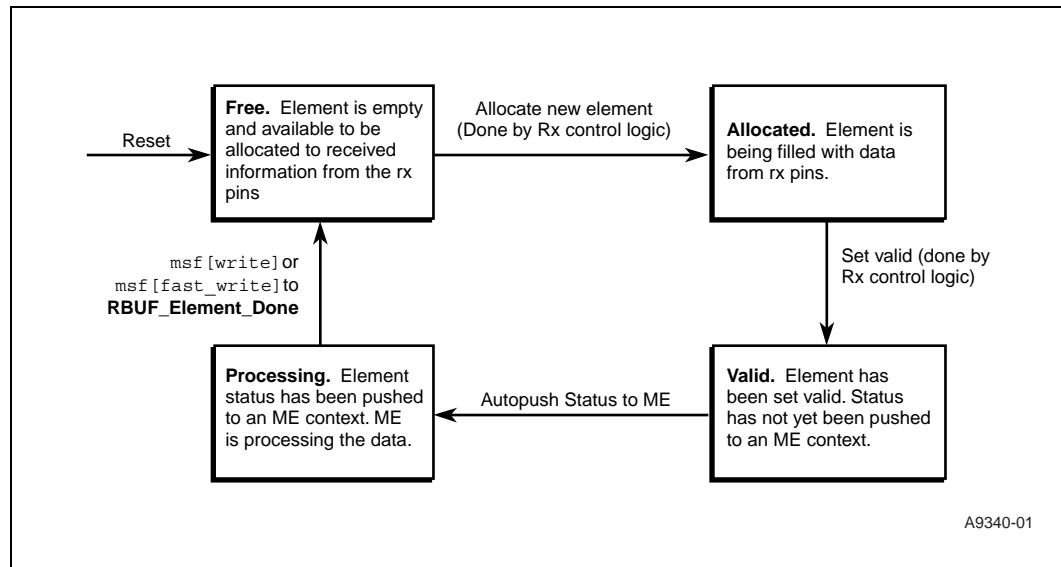
To handle the case where the receive Contexts temporarily fall behind and **Rx_Thread_Freelist_#** is empty, all received element numbers are held in the Full Element List. In that case, as soon as an **Rx_Thread_Freelist_#** entry is entered, the status of the head element of Full Element List will be pushed to it.

The Microengine may read part of (or the entire) RBUF element to their S_Transfer registers (via `msf[read]` instruction) for header processing, etc., and may also move the element data to DRAM (via `dram[rbuf_rd]` instruction).

When a Context is done with an element it does a `msf[write]` or `msf[fast_write]` to **RBUF_Element_Done** address; the write data is the element number. This marks the element as free and available to be re-used. There is no restriction on the order in which elements are freed; Contexts can do different amounts of processing per element based on the contents of the element—therefore elements can be returned in a different order than they were handed to Contexts.

The states that an RBUF element goes through are shown in [Figure 92](#).

Figure 92. RBUF Element State Diagram



[Table 95](#) summarizes the differences in RBUF operation between SPI-4 protocol and CSIX protocol.

Table 95. Summary of SPI-4 and CSIX RBUF Operations

Operation	SPI-4	CSIX
When is RBUF Element Allocated	Upon receipt of Payload Control Word or when Element data section fills and more Data Words arrive. The Payload Control Word allocates an element for data that will be received subsequent to it.	Start of Frame and Base Header Type is mapped to RBUF (in CSIX_Type_Map CSR).
How Much Data is Put into Element	All Data Words received between two Payload Control Words, or number of bytes in the element, whichever is less.	Number of bytes specified in Payload Length field of Base Header.
How is RBUF Element Set Valid	Upon receipt of Payload Control Word or when Element data section fills. The Payload Control Word validates the element holding data received prior to it.	All Payload is received (or if premature SOF, which will set an error bit in Element Status).
How is RBUF Element Handed to Microengine	Element Status is pushed to Microengine at the head of the appropriate Rx_Thread_Freelist_# (based on the protocol). Status is pushed to two consecutive Transfer Registers; bits[31:0] of Element Status to the first Transfer Register and bits[63:32] to the next higher numbered Transfer Register.	
How is RBUF Element returned to free list	CSR write to RBUF_Element_Done .	

8.2.7 Receive Flow Control Status

Flow control is handled in hardware. There are specific functions for SPI-4 and CSIX.

8.2.7.1 SPI-4

SPI-4, FIFO status information is sent periodically over the **RSTAT** signals from the Link Layer device (which is the IXP2800 Network Processor) to the PHY device. [Note that **TXCDAT** pins can act as **RSTAT** based on **MSF_Rx_Control[RSTAT_Select]** bit.] The information to be sent is based on the number of RBUF elements available to receive SPI-4.

The FIFO status of each port is encoded in a 2-bit data structure—code 0x3 is used for framing the data, and the other three codes are valid status values.

The FIFO status words are sent according to a repeating calendar sequence. Each sequence begins with the framing code to indicate the start of a sequence, followed by the status codes, followed by a parity code covering the preceding frame. The length of the calendar is defined in **Rx_Calendar_Length**, which is a CSR field that is initialized with the length of the calendar, since in many cases fewer than 256 ports are in use.

When **MSF_Rx_Control[RSTAT_En]** is disabled, **RSTAT** is held at 0x3.

The IXP2800 Network Processor transmits FIFO status only if **MSF_Rx_Control[RSTAT_En]** is set. The logic sends “Satisfied,” “Hungry,” or “Starving” based on either the high water mark of the RBUF, a global override value set in **MSF_Rx_Control[RSTAT_OV_VALUE]**, or a port-specific override value set in **RX_PORT_CALENDAR_STATUS_#**. The choice is controlled by **MSF_RX_CONTROL[RX_Calendar_Mode]**.

When set to **Conservative_Value**, the status value sent for each port is the most conservative of:

- The RBUF high water mark
- **MSF_RX_CONTROL[RSTAT_OV_VALUE]**
- **RX_PORT_CALENDAR_STATUS_#**

“Satisfied” is more conservative than “Hungry,” which is more conservative than “Starving.”

When **MSF_RX_CONTROL[RX_Calendar_Mode]** is set to **Force_Override**, the value of **RX_PORT_CALENDAR_STATUS_#** is used to determine which status value is sent. If **RX_PORT_CALENDAR_#** is set to 0x3, then the global status value set in **MSF_RX_CONTROL[RSTAT_OV_VALUE]** is sent, otherwise the port-specific status value set in **RX_PORT_CALENDAR_#** is sent.

The RBUF high water mark is based on the **MSF_Rx_Control** Register and is defined in [Table 91](#). The high water mark is programmed in **HWM_Control[RBUF_S_HWM]**. Note that either RBUF partition 0 or partition 1 will be used for SPI-4 ([Table 90](#)).

8.2.7.2 CSIX

There are two types of CSIX flow control:

- Link-level
- Virtual Output Queue (VOQ)

Information received from the Switch Fabric by the Egress IXP2800 Network Processor, must be communicated to the Ingress IXP2800 Network Processor, which is sending data to the Switch Fabric.

8.2.7.2.1 Link-level

Link-level flow control can be used to stop all transmission. Separate Link-level flow control is provided for Data CFrames and Control CFrames. CSIX protocol provides link-level flow control as follows. Every CFrame Base Header contains a Ready Field, which contains two bits; one for Control traffic (bit 6 of byte 1) and one for Data traffic (bit 7 of byte 1). The CSIX requirement for response is:

From the tick that the Ready Field leaves a component the maximum response time for a pause operation is defined as: $n \cdot T$, $n = C + L$ where:

- T is the clock period of the interface
- n is the maximum number of ticks for the response
- C is a constant for propagating the field within the "other" component (or chipset as the case may be) to the interface logic control the reverse direction data flow. C is defined to be 32 ticks.
- L is the maximum number of ticks to transport the maximum fabric CFrame size.

As each CFrame is received, the value of these bits is copied (by receive hardware) into the **FC_Egress_Status[SF_CReady]** and **FC_Egress_Status[SF_DReady]** respectively. The value of these two bits is sent from the Egress to the Ingress IXP2800 Network Processor on **TXCSRB** signal, and can be used to stop transmission to the Switch Fabric, as described in [Section 8.3.4.2](#). **TXCSRB** signal is described in [Section 8.5.1](#).

8.2.7.2.2 Virtual Output Queue

CSIX protocol provides Virtual Output Queue Flow Control via Flow Control CFrames.

CFrames that were mapped to FCEFIFO (via **CSIX_Type_Map** CSR) are parsed by the receive control logic and placed into FCEFIFO, which provides buffering while they are sent from the Egress the IXP2800 Network Processor to the Ingress IXP2800 Network Processor over the **TXCDAT** signals (normally Flow Control CFrames would be mapped to FCEFIFO).

The entire CFrame is sent over **TXCDAT**, including the Base Header and Vertical Parity field. The 32-bit CWord is sent four bits at a time, most significant bits first. The CFrames are forwarded in a “cut-through” manner, meaning the Egress IXP2800 Network Processor does not wait for the entire CFrame to be received before forwarding (each CWord can be forwarded as it is received).

If FCEFIFO gets full, as defined by **HWM_Control[FCEFIFO_HWM]**, then the **FC_Egress_Status[TM_CReady]** bit will be deasserted (to inform the Ingress IXP2800 Network Processor to deassert Control Ready in CFrames sent to the Switch Fabric).

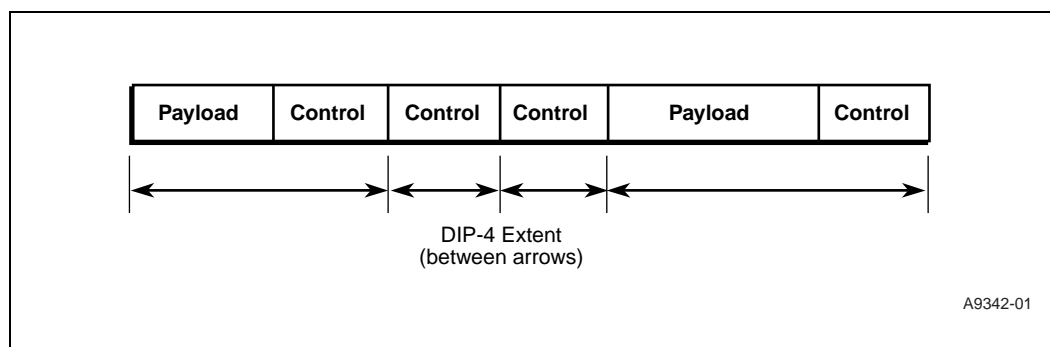
The usage of the Flow Control information in the Ingress IXP2800 Network Processor is described in [Section 8.3.4.2](#).

8.2.8 Parity

8.2.8.1 SPI-4

The receive logic computes 4-bit Diagonal Interleaved Parity (DIP-4) as specified in the SPI-4 specification. The DIP-4 field received in a control word contains odd parity computed over the current Control Word and the immediately preceding data words (if any) following the last Control Word. [Figure 93](#) shows the extent of the DIP-4 codes.

Figure 93. DIP-4 Codes' Extent



There is a DIP-4 Error Flag and a 4-bit DIP-4 Accumulator Register. After each Control Word is received the Flag is conditionally reset (see Note below this paragraph) and the Accumulator Register is cleared. As each Data Word (if any), and the first succeeding Control Word is received, DIP-4 parity is accumulated in the register as defined in the SPI-4 spec. The accumulated parity is compared to the value received in the DIP-4 field of that first Control Word. If it does not match the DIP-4 Error Flag is set. The value of the flag becomes the element status Par Err bit.

Note: An error in the DIP-4 code invalidates the transfer preceding the Control Word and also the transfer succeeding it, since the control information is assumed to be in error. Therefore the DIP-4 Error

Flag is not reset after a Control Word with bad DIP-4 parity. It is only reset after a Control Word with correct DIP-4 parity.

8.2.8.2 CSIX

8.2.8.2.1 Horizontal Parity

The receive logic computes Horizontal Parity on each 16-bits of each received Cword (separate parity for data received on rising and falling edge of the clock).

There is an internal HP Error Flag. At the end of each CFrame the flag is reset. As each 16-bits of each Cword is received, the expected odd parity value is computed from the data, and compared to the value received on **RxPar**. If there is a mismatch the flag is set. The value of the flag becomes the element status HP Err bit.

If the HP Error Flag is set, the **FC_Egress_Status[SF_CReady]** and **FC_Egress_Status[SF_DReady]** bits are cleared, and the **MSF_Interrupt_Status[HP_Error]** bit is set, which can interrupt the Intel XScale® core if enabled.

8.2.8.2.2 Vertical Parity

The receive logic computes Vertical Parity on CFrames.

There is a VP Error Flag and a 16-bit VP Accumulator Register. At the end of each CFrame the flag is reset and the register is cleared. As each Cword is received, odd parity is accumulated in the register as defined in the CSIX spec (16 bits of vertical parity are formed on 32 bits of received data by treating the data as words; i.e., bit 0 and bit 16 of the data are accumulated into parity bit 0, bit 1 and bit 17 of the data are accumulated into parity bit 1, etc.). After the entire CFrame has been received (including the Vertical Parity field; the two bytes following the Payload) the accumulated value should be 0xFFFF. If it is not the VP Error Flag is set. The value of the flag becomes the element status VP Err bit.

Note: The Vertical Parity always follows the Payload, which may include padding to the CWord width if the Payload Length field is not an integral number of CWords. The CWord width is programmed in **MSF_Rx_Control[Rx_CWord_Size]**.

If the VP Error Flag is set, the **FC_Egress_Status[SF_CReady]** and **FC_Egress_Status[SF_DReady]** bits are cleared, and the **MSF_Interrupt_Status[VP_Error]** bit is set, which can interrupt the Intel XScale® core.

8.2.9 Error Cases

Receive errors are specific to the protocol, SPI-4 or CSIX. The element status, described in [Table 92](#) and [Table 93](#), has appropriate error bits defined. Also, there are some IXP2800 Network Processor specific error cases, like when an mpacket arrives with no free elements, which are logged in the **MSF_Interrupt_Status** register, which can interrupt the Intel XScale® core if enabled.

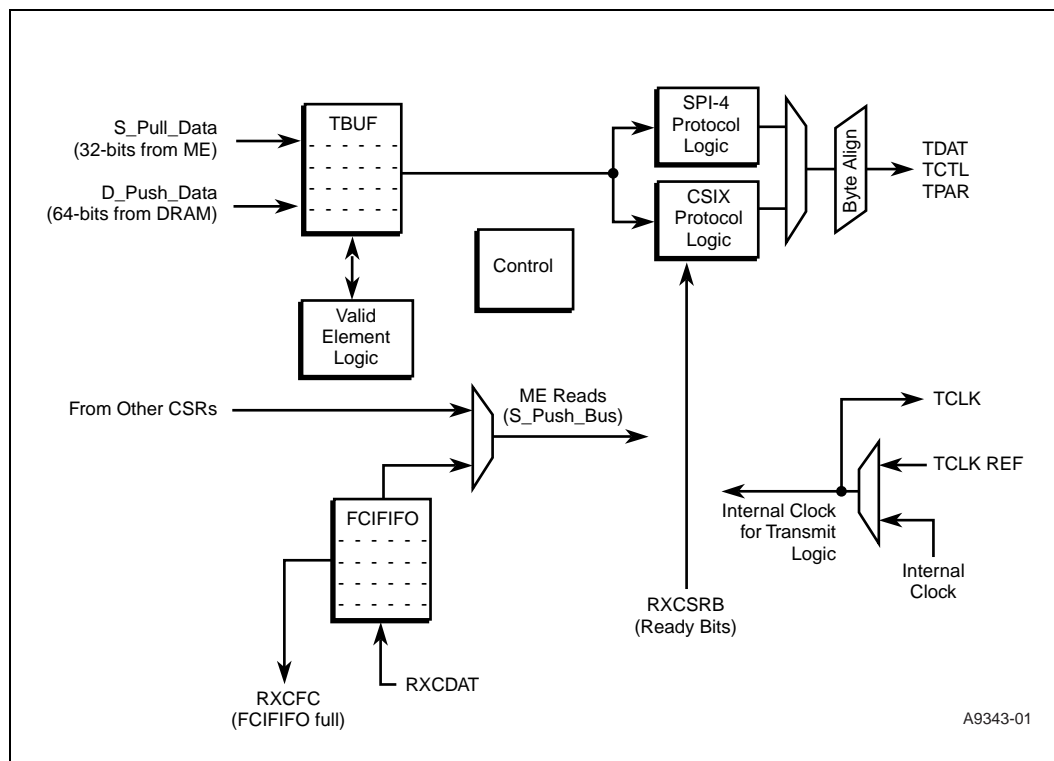
8.3 Transmit

The transmit section consists of:

- Transmit Pins (Section 8.3.1)
- Transmit Buffer (Section 8.3.2)
- Byte Aligner (Section 8.3.2)

Each of these is described below. Figure 94 is a simplified Block Diagram of Section 8.3.

Figure 94. Simplified of Transmit Section Block Diagram





8.3.1 Transmit Pins

The use of the transmit pins is a function of the protocol (which is determined by TBUF partition in **MSF_Tx_Control** CSR) as shown in [Table 96](#).

Table 96. Transmit Pins Usage by Protocol

Name	Direction	SPI-4 Use	CSIX Use
TCLK	Output	TDCLK	RxCk
TDAT[15:0]	Output	TDAT[15:0]	RxDat[15:0]
TCTL	Output	TCTL	RxSOF
TPAR	Output	Not Used	RTxPar
TSCLK	Input	TSCLK	Not Used
TSTAT[1:0]	Input	TSTAT[1:0]	Not Used

8.3.2 TBUF

The TBUF is a RAM that holds data and status to be transmitted. The data is written into sub-blocks referred to as elements, by Microengine or the Intel XScale® core. TBUF contains a total of 8 Kbyte of data, and associated control.

[Table 97](#) shows the order in which data is written into TBUF. Each number represents a byte, in order of transmission onto the tx interface. Note that this is reversed on a 32-bit basis relative to RBUF—the swap of 4 low bytes and 4 high bytes is done in hardware to facilitate the transmission of bytes.

Table 97. Order in Which Data is Transmitted from TBUF

Data/Payload								Address Offset (Hex)
0	1	2	3	4	5	6	7	0
8	9	A	B	C	D	E	F	8
10	11	12	13	14	15	16	17	10

The mapping of elements to address offset in TBUF is based on the TBUF partition and element size, as programmed in **MSF_Tx_Control** CSR. TBUF can be partitioned into one, two, or three partitions based on **MSF_Tx_Control[TBUF_Partition]**. The mapping of partitions to transmit data is shown in [Table 98](#).

Table 98. Mapping of TBUF Partitions to Transmit Protocol

Number of Partitions in Use	Transmit Data Protocol	Data Use by Partition, Fraction of TBUF Used, Start Byte Offset (Hex)		
		Partition Number		
		0	1	2
1	SPI-4 only	SPI-4 All Byte 0	n/a	n/a
2	CSIX only	CSIX Data ¾ of TBUF Byte 0	CSIX Control ¼ of TBUF Byte 0x1800	n/a
3	Both SPI-4 and CSIX	CSIX Data ½ of TBUF Byte 0	SPI-4 3/8 of TBUF Byte 0x1000	CSIX Control 1/8 of TBUF Byte 0x1C00

The data in each segment is further broken up into elements, based on **MSF_Tx_Control[TBUF_Element_Size_#]** (n = 0,1,2). There are three choices of element size, 64, 128, or 256 bytes.

Table 99 shows the TBUF partition options. Note that the choice of element size is independent for each partition.

Table 99. Number of Elements per TBUF Partition

TBUF_Partition Field	TBUF_Element_Size_# Field	Partition Number		
		0	1	2
00 (1 partition)	00 (64 byte)	128	Unused	Unused
	01 (128 byte)	64		
	10 (256 byte)	32		
01 (2 partitions)	00 (64 byte)	96	32	Unused
	01 (128 byte)	48	16	
	10 (256 byte)	24	8	
10 (3 partitions)	00 (64 byte)	64	48	16
	01 (128 byte)	32	24	8
	10 (256 byte)	16	12	4

Microengine can write data from Microengine S_Transfer_Out registers to the TBUF using the `msf[write]` instruction, where they specify the starting byte number (which must be aligned to 4 bytes), and number of 32-bit words to write. The number in the instruction can be either the number of 32-bit words, or number of 32-bit word pairs, using the single and double instruction modifiers, respectively. Data is pulled from Microengine to TBUF via S Pull Bus.

`msf[write, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token`

The `src_op_1` and `src_op_2` operands are added together to form the address in TBUF (note that the base address of the TBUF is 0x2000). `ref_cnt` is the number of 32-bit words or word pairs, which are pulled from sequential S_Transfer_Out registers, starting with `$s_xfer_reg`.

Microengine can move data from DRAM to TBUF using the instruction
`dram[tbuf_wr, --, src_op1, src_op2, ref_cnt], indirect_ref`

The `src_op_1` and `src_op_2` operands are added together to form the address in DRAM, so the `dram` instruction must use indirect mode to specify the TBUF address. `ref_cnt` is number of 64-bit words which are written into TBUF.

Data is stored in big-endian order. The most significant (lowest numbered) byte of each 32-bit word is transmitted first.

All elements within a TBUF partition are transmitted in the order. Control information associated with the element ([Section 100](#) and [Section 101](#)) defines which bytes are valid. The data from the TBUF will be shifted and byte aligned to the **TDAT** pins as required. Four parameters are defined.

Prepend Offset—Number of the first byte to send. This is information that is prepended onto the payload, for example as a header. It need not start at offset 0 in the element.

Prepend Length—Number of bytes of prepended information. This can be 0 to 31 bytes. If it is 0, then Prepend Offset must also be 0.

Payload Offset—Number of bytes to skip from the last 64-bit word of the Prepend to the start of Payload. The absolute byte number of the first byte of Payload in the element is:
 $((\text{Prepend Offset} + \text{Prepend Length} + 0x7) \& 0xF8) + \text{Payload Offset}$

Payload Length—Number of bytes of Payload.

The sum of Prepend Length, Payload length and any gaps in between them $(((\text{prepend_offset} + \text{prepend_length} + 7) \& 0xF8) + \text{payload_offset} + \text{payload_length})$ must be no greater than the number of bytes in the element. Typically the Prepend will be computed by a Microengine and written into the TBUF by `msf[write]` and the Payload will be written by `dram[tbuf_wr]`. These two operations can be done in either order; the microcode is responsible for making sure the element is not marked valid to transmit until all data is in the element (see [Section 8.3.3](#)).

[Example 35](#) illustrates the usage of the parameters. The element in [Example 35](#) is shown as 8 bytes wide because the smallest unit that can be moved into the element is 8 bytes. In [Example 35](#), bytes to be transmitted are shown in black (the offsets are byte numbers); bytes in gray are written into TBUF (because the writes always write 8 bytes), but are not transmitted.

Prepend Offset = 6 (bytes 0x0 through 0x5 are not transmitted).

Prepend Length = 16 (bytes 0x6 through 0x15 are transmitted).

Payload Offset = 7 (bytes 0x16 through 0x1E are not transmitted). The Payload starts in the next 8-byte row (i.e., the next “empty” row above where the Prepend stops), even if there is room in the last row containing Prepend information. This is done because the TBUF does not have byte write capability, and therefore would not merge the `msf[write]` and `dram[tbuf_wr]`. The software computing the Payload Offset only needs to know how many bytes of the payload that were put into DRAM need to be removed.

Payload Length = 33 (bytes 0x1F through 0x3F are transmitted).

0	1	2	3	4	5	6	7
8	9	A	B	C	D	E	F
10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27
28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37
38	39	3A	3B	3C	3D	3E	3F

8.3.2.1 SPI-4

When the Element Control Word is written the information is (note that the data comes from two consecutive Transfer Registers; bits [31:0] from the lower numbered and bits[63:32] from the higher numbered):

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
Payload Length								Prepend Offset			Prepend Length					Payload Offset			Res	Skip	Abort	SOP	EOP	ADR							

6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
Res																															

Table 100. TBUF SPI-4 Control Definition

Field	Definition
ADR	The port number to which the data is directed. This field will be sent in the ADR field of the Control Word that will precede the data transfer.
SOP	Indicates if the element is the start of a packet. This field will be sent in the SOPC field of the Control Word that will precede the data transfer.
EOP	Indicates if the element is the end of a packet. This field will be sent in the EOPS field of the Control Word that will succeed the data transfer. Note 1.

NOTE:

1. Normally EOPS is sent on the next Control Word (along with ADR and SOP) to start the next element. If there is no valid element pending at the end of sending the data, the transmit logic will insert an Idle Control Word with the EOPS information.

Table 100. TBUF SPI-4 Control Definition (Continued)

Field	Definition
Abort	Indicates if the element is the end of a packet that should be aborted. If this bit is set the status code of EOP Abort will be sent in the EOPS field of the Control Word that will succeed the data transfer. Note 1.
Prepend Offset	Indicates the first valid byte of Prepend, from 0 to 7, as defined in Section 8.3.2.
Prepend Length	Indicates the number of bytes in Prepend, from 0 to 31.
Payload Offset	Indicates the first valid byte of Payload, from 0 to 7, as defined in Section 8.3.2.
Payload Length	Indicates the number of Payload bytes, from 1 to 256, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent. That value will also control the EOPS field (1 or 2 bytes valid indicated) of the Control Word that will succeed the data transfer. Note 1.
Skip	Allows software to allocate a TBUF element and then not transmit any data from it. 0—transmit data according to other fields of Control Word. 1—free the element without transmitting any data.
NOTE: 1. Normally EOPS is sent on the next Control Word (along with ADR and SOP) to start the next element. If there is no valid element pending at the end of sending the data, the transmit logic will insert an Idle Control Word with the EOPS information.	

8.3.2.2 CSIX

For CSIX protocol, the TBUF should be set to two partitions in **MSF_Tx_Control[TBUF_Partition]**, one for Data traffic and one for Control traffic.

Payload information is put into the Payload area of the element, and Base and Extension Header information is put into the Element Control Word.

Data is stored in big-endian order. The most significant byte of each 32-bit word is transmitted first.

When the Element Control Word is written the information is (note that the data comes from two consecutive Transfer Registers; bits [31:0] from the lower numbered and bits[63:32] from the higher numbered):

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2
Payload Length								Prepend Offset		Prepend Length					Payload Offset		Res	SOP	Res	CR	u	Res					Type		

6	6	6	6	5	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3
3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4
Extension Header																													

The definitions of the fields are shown in Table 101.

Table 101. TBUF CSIX Control Definition

Field	Definition
Type	Type Field to put into the CSIX Base Header. Idle type is <i>not</i> legal here.
CR	CR (CSIX Reserved) bit to put into the CSIX Base Header.
P	P (Private) bit to put into the CSIX Base Header.
Extension Header	The Extension Header to be sent with the CFrame. The bytes are sent in big-endian order; byte 0 is in bits 63:56, byte 1 is in bits 55:48, byte 2 is in bits 47:40, and byte 3 is in bits 39:32.
Prepend Offset	Indicates the first valid byte of Prepend, from 0 to 7, as defined in Section 8.3.2 .
Prepend Length	Indicates the number of bytes in Prepend, from 0 to 31.
Payload Offset	Indicates the first valid byte of Payload, from 0 to 7, as defined in Section 8.3.2 .
Payload Length	Indicates the number of Payload bytes, from 1 to 256, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent, and also put into the CSIX Base Header Payload Length field. Note that this length does not include any padding which may be required. Padding is inserted by transmit hardware as needed.
Skip	Allows software to allocate a TBUF element and then not transmit any data from it. 0—transmit data according to other fields of Control Word 1—free the element without transmitting any data.

8.3.3 Transmit Operation Summary

During transmit processing data to be transmitted is placed into the TBUF under ME control. The Microengine allocates an element in software; the transmit hardware processes TBUF elements within a partition in strict sequential order so the software can track which element to allocate next.

Microengines may write directly into an element by `msf[write]` instruction, or have data from DRAM written into the element by `dram[tbuf_wr]` instruction. Data can be merged into the element by doing both.

There is a Transmit Valid bits per element, which marks the element as ready to be transmitted. Microengines move all data into the element, by either or both of `msf[write]` and `dram[tbuf_wr]` instructions to the TBUF. MEs also write the element Transmit Control Word with information about the element. The Microengines should use a single operation to perform the TCW write, i.e., a single `msf[write]` with a `ref_count` of 2. When all the data movement is complete the Microengine sets the element valid bit as shown in the following steps.

1. Move data into TBUF by either or both of `msf[write]` and `dram[tbuf_wr]` instructions to the TBUF.
2. Wait for 1 to complete.
3. Write Transmit Control Word at `TBUF_Element_Control_#` address. Using this address sets the Transmit Valid bit.

Note: When moving data from DRAM to TBUF using `dram[tbuf_wr]`, it is possible that there could be an uncorrectable error on the data read from DRAM (if ECC is enabled). In that case, the Microengine does not get an Event Signal, to prevent use of the corrupt data. The error is recorded in the DRAM controller (including the number of the Microengine that issued the `TBUF_Wr` command, refer to the DRAM chapter for details), and will interrupt the Intel XScale® core, if enabled, so that it can take appropriate action. Such action is beyond the scope of this document.

However it must include recovering the TBUF element by setting it valid with Skip bit set in the Control Word.

The transmit pipeline will be stalled since all TBUF elements must be transmitted in order; it will be un-stalled when the element is skipped.

8.3.3.1 SPI-4

Transmit control logic sends valid elements on the transmit pins in element order. First a Control Word is sent—it is formed as shown in Table 102. After the Control Word, the data is sent; the number of bytes to send is the total of Element Control Word Prepend Length field plus the Element Control Word Payload Length.

Table 102. Transmit SPI-4 Control Word

SPI-4 Control Word Field	Derived From
Type	Type Bit of Element Control Word
EOPS	EOP Bit, Prepend Length, Payload Length of previous element's Element Control Word
SOP	SOP Bit of Element Control Word
ADR	ADR field of Element Control Word
DIP-4	Parity accumulated on previous element's data and this Control Word

If the next sequential element is not valid when its turn comes up:

1. Send an idle Control Word with SOP set to 0, EOPS set to the values determined from the most recently sent element, ADR field 0x00, correct parity.
2. Until an element becomes valid, send idle Control Words with SOP set to 0, EOPS set to 00, ADR field 0x00, and correct parity.

Note: Sequential elements with same ADR are not “merged”, a Control Word is sent for each element.

Note: SPI-4 requires that all data transfers, except the last fragment (with EOP), be multiples of 16 bytes. It is up to the software loading the TBUF element to enforce this rule.

After an element has been sent on the transmit pins, the valid bit for that element is cleared. The **Tx_Sequence** register is incremented when the element has been transmitted; by also maintaining a sequence number of elements that have been allocated (in software), the microcode can determine how many elements are in-flight.



8.3.3.2 CSIX

Transmit control logic sends valid elements on the transmit pins in element order. Each element sends a single CFrame. First the Base Header is sent—it is formed as shown in [Table 103](#). Next the Extension Header is sent. Finally, the data is sent; the number of bytes to send is the total of Element Control Word Prepend Length field plus the Element Control Word Payload Length; plus padding to fill the final CWord if required (the CWord Size is programmed in MSF_Tx_Control[Tx_CWord_Size]). Both Horizontal Parity and Vertical Parity are transmitted, as described in [Section 8.3.5.2.1](#) and [Section 8.3.5.2.2](#).

Note: When transmitting a Flow Control CFrame, the entire payload must be written into the TBUF entry. The extension header field of the Transmit Control Word is not used for Flow Control CFrames.

Table 103. Transmit CSIX Header

CSIX Header Field	Derived From
Type	Type field of Element Control Word
Data Ready	FC_Ingress_Status[TM_DReady]
Control Ready	FC_Ingress_Status[TM_CReady]
Payload Length	Element Control Word Prepend Length + Element Control Word Payload Length
P	P Bit of Element Control Word
CR	CR Bit of Element Control Word
Extension Header	Extension Header field of Element Control Word

Control elements and Data elements share use of the transmit pins. Each will alternately transmit a valid element, if present.

If the next sequential element is not valid when its turn comes up, or if transmission is disabled by **FC_Ingress_Status[SF_CReady]** or **FC_Ingress_Status[SF_DReady]**, then transmit logic will alternate sending Idle CFrames with Dead Cycles; it will continue to do so until a valid element is ready. Idle CFrames get the value for the Ready Field from **FC_Ingress_Status[TM_CReady]** and **FC_Ingress_Status[TM_DReady]**, the Payload Length is set to 0.

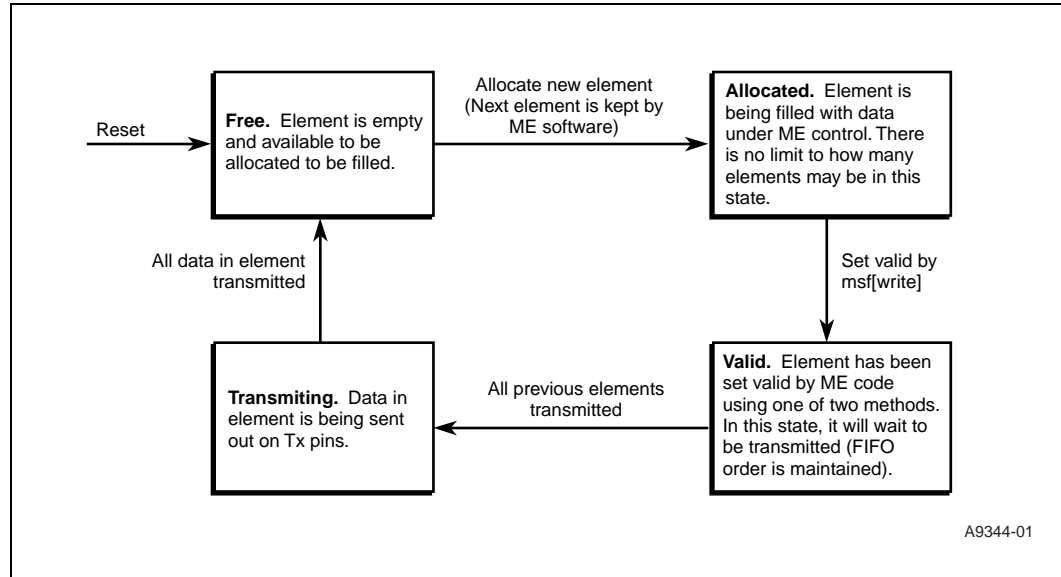
Note: A Dead Cycle is any cycle after the end of a CFrame, and prior to the start of another CFrame (i.e., SOF is not asserted). The end of a CFrame is defined as after the Vertical Parity has been transmitted. This in turn is found by counting the Payload Bytes specified in the Base Header and rounding up to CWord size.

After an element has been sent on the transmit pins, the valid bit for that element is cleared. The **Tx_Sequence** register is incremented when the element has been transmitted; by also maintaining a sequence number of elements that have been allocated (in software), the microcode can determine how many elements are in-flight.

8.3.3.3 Transmit Summary

The states that a TBUF element goes through are shown in Figure 95.

Figure 95. TBUF State Diagram



8.3.4 Transmit Flow Control Status

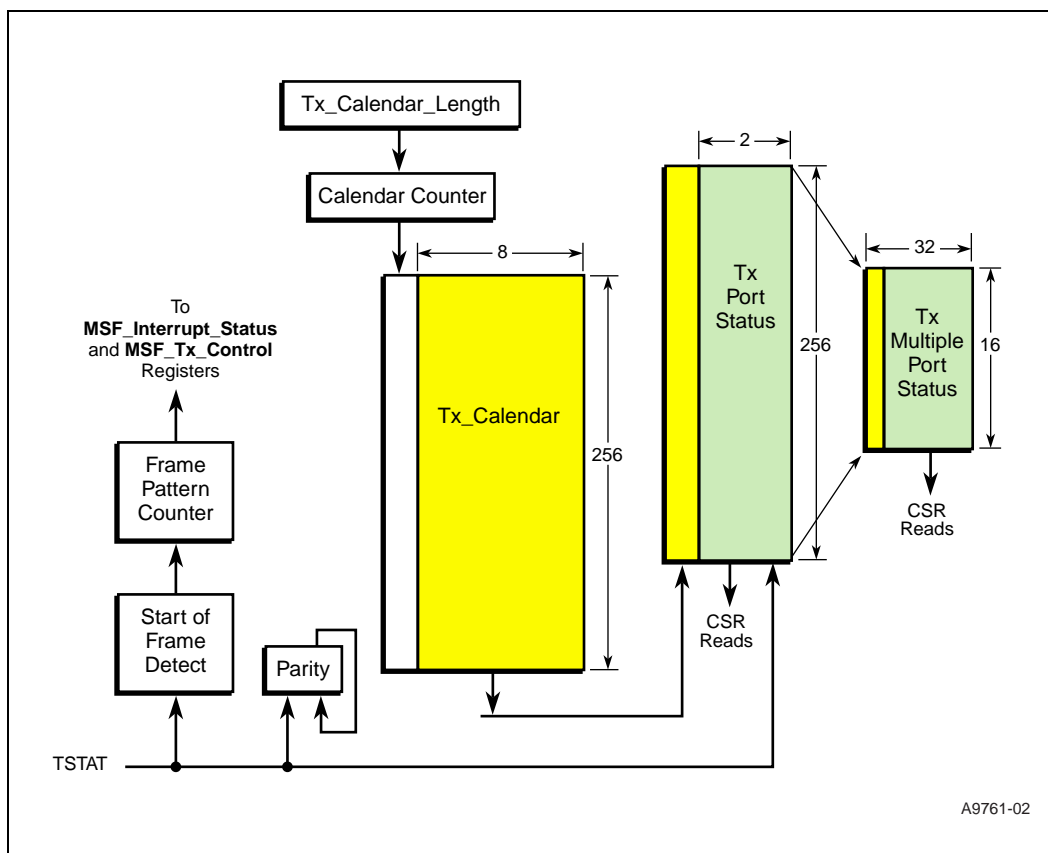
Transmit Flow Control is handled partly by hardware and partly by software. Information from the Egress IXP2800 Network Processor can be transmitted to the Ingress IXP2800 Network Processor (as described in Section 8.2.7 on Receive Flow Control); how it is used is described in the remainder of this section.

8.3.4.1 SPI-4

FIFO status information is sent periodically over the **TSTAT** signals from the PHY to the Link Layer device (which is the IXP2800 Network Processor). [Note that **RXC DAT** pins can act as **TSTAT** based on **MSF_Tx_Control[TSTAT_Select]** bit.] The FIFO status of each port is encoded in a 2-bit data structure—code 0x3 is used for framing the data, and the other three codes are valid status values, which are interpreted by Microengine software.

The FIFO status words are received according to a repeating calendar sequence. Each sequence begins with the framing code to indicate the start of a sequence, followed by the status codes, followed by a DIP-2 parity code covering the preceding frame. The length of the calendar, as well as the port values, are defined in this section, and shown in Figure 96.

Figure 96. Tx Calendar Block Diagram



Tx_Port_Status_# is a register file containing 256 registers, one for each of the SPI-4.2 ports. The port status is updated each time a new calendar status is received for each port, according to the mode programmed in **MSF_Tx_Control[Tx_Status_Update_Mode]**. **Tx_Port_Status_#** holds the latest received status for each port, and can be read by CSR reads.

There are 16 **Tx_Multiple_Port_Status_#** registers. Each aggregates the status for each group of 16 ports. These registers provide an alternative method for reading the FIFO status of multiple ports with a single CSR read. For example, **Tx_Multiple_Port_Status_0** contains the 2-bit status for ports 0 through 16, and provides the same status as reading the individual registers **Tx_Port_Status_0** through **Tx_Port_Status_15**.

The **TX_Port_Status_#** or the **TX_Multiple_Port_Status_#** registers must be read by the software in order to determine the status of each port and send data to them accordingly. The MSF hardware does not check these registers for port status before sending data out to a particular port.

The **MSF_Tx_Control[Tx_Status_Update_Mode]** field is used to select which of two methods should be used for updating the port status. The first method updates the port status with the new status value, regardless of the value received. The second method updates the port status only when a value is received that is equal to or less than the current value.

Note: Detailed information about the status update modes is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

Reading a port status causes its value to be changed. This provides a means to avoid reading stale status bits. The **MSF_Tx_Control[Tx_Status_Read_Mode]** field is used to select the method used to change the bits after they are read.

Tx_Calendar is a RAM with 256 entries of 8 bits each. It is initialized with the calendar information by software (the calendar is a list that indicates the sequence of port status that will be sent—the PHY and the IXP2800 Network Processor must be initialized with the same calendar). **Tx_Calendar_Length** is a CSR field that is initialized with the length of the calendar, since in many cases not all 256 entries of **Tx_Calendar** are used.

When the start of a Status frame pattern is detected (by a value of 0x3 on **TSTAT**) the Calendar Counter is initialized to zero. On each data cycle the Calendar Counter is used to index into **Tx_Calendar** to read a port number. The port number is used as an index to **Tx_Port_Status**, and the information received on **TSTAT** is put into that location in **Tx_Port_Status**. The count is incremented each cycle.

DIP-2 Parity is also accumulated on **TSTAT**. At the start of the frame, parity is cleared. When the count reaches **Tx_Calendar_Length** the next value on **TSTAT** is used to compare to the accumulated parity. The control logic then looks for the next frame start. If the received parity does not match the expected value **MSF_Interrupt_Status[TSTAT_Par_Err]** bit is set, which can interrupt the Intel XScale® core if enabled.

Note: An internal status flag records whether or not the most recently received DIP-2 was correct. When that flag is set (indicating bad DIP-2 parity) all reads to **Tx_Port_Status** return a status of “Satisfied” instead of the value in the **Tx_Port_Status** RAM. The flag is re-loaded at the next parity sample; so the implication is that all ports will return “Satisfied” status for at least one calendar.

SPI-4 protocol uses a continuous stream of repeated frame patterns to indicate a disabled status link. The IXP2800 Network Processor flow control status block has a Frame Pattern Counter that counts up each time a frame pattern is received on **TSTAT**, and is cleared when any other pattern is received. When the Frame Pattern Counter reaches 32

MSF_Interrupt_Status[Detect_No_Calendar] is set and **Train_Data[Detect_No_Calendar]** is asserted (**MSF_Interrupt_Status[Detect_No_Calendar]** must be cleared by a write to the **MSF_Interrupt_Status** register; **Train_Data[Detect_No_Calendar]** will reflect the current status and will deassert when the frame pattern stops). The transmit logic will generate training sequence on transmit pins while both **Train_Data[Detect_No_Calendar]** and **Train_Data[Train_Enable_TSTAT]** are asserted.

8.3.4.2 CSIX

There are two types of CSIX flow control:

- Link-level
- Virtual Output Queue (VOQ)

8.3.4.2.1 Link-level

The Link-level flow control function is done via hardware and consists of two parts:

1. Enable/disable transmission of valid TBUF elements.
2. Ready field to be sent in CFrames sent to the Switch Fabric.

As described in [Section 8.2.7](#), the Ready Field of received CFrames is placed into **FC_Egress_Status[SF_CReady]** and **FC_Egress_Status[SF_DReady]**. The value in those bits is sent to the Ingress IXP2800 Network Processor on **TXCSRB**. In Full Duplex Mode, the information is received on **RXCSRB** by the Ingress IXP2800 Network Processor and put into **FC_Ingress_Status[SF_CReady]** and **FC_Ingress_Status[SF_DReady]**. Those bits allow or stop transmission of Control and Data elements, respectively. When one of those bits transitions from allowing transmission to stopping transmission, the current CFrame in progress (if any) is completed, and the next CFrame of that type is prevented from starting.

Also described in [Section 8.2.7](#), if the Egress IXP2800 Network Processor RBUF gets near full, or if the Egress IXP2800 Network Processor FCEFIFO gets near full, it will send that information on **TXCSRB**. Those bits are put into **FC_Ingress_Status[TM_CReady]** and **FC_Ingress_Status[TM_DReady]**, and are used as the value in CFrame Base Header Control Ready and Data Ready, respectively.

8.3.4.2.2 Virtual Output Queue

The Virtual Output Queue flow control function is done by software, with hardware support.

As described in [Section 8.2.7](#), CSIX Flow Control CFrames received on the Egress IXP2800 Network Processor are passed to the Ingress IXP2800 Network Processor over **TXCDAT**. The information is received on **RXCDAT** and placed into the FCIFIFO. A Microengine reads that information by `msf[read]`, and uses it to maintain per-VOQ information. How that information is used is application dependent and is done in software. The hardware mechanism is described in [Section 8.5.3](#).

8.3.5 Parity

8.3.5.1 SPI-4

DIP-4 parity is computed by Transmit hardware placed into the Control Word sent at the beginning of transmission of a TBUF element, and also on Idle Control Words sent when no TBUF element is valid. The value to place into the DIP-4 field is computed on the preceding Data Words (if any), and the current Control Word.



8.3.5.2 CSIX

8.3.5.2.1 Horizontal Parity

The transmit logic computes odd Horizontal Parity for each transmitted 16-bits of each Cword, and transmits it on **TxPar**.

8.3.5.2.2 Vertical Parity

The transmit logic computes Vertical Parity on CFrames. There is a 16-bit VP Accumulator Register. At the beginning of each CFrame the register is cleared. As each Cword is transmitted, odd parity is accumulated in the register as defined in the CSIX spec (16 bits of vertical parity are formed on 32 bits of transmitted data by treating the data as words; i.e., bit 0 and bit 16 of the data are accumulated into parity bit 0, bit 1, and bit 17 of the data are accumulated into parity bit 1, etc.). The accumulated value is transmitted in the Cword along with the last byte of Payload and any padding, if required.

8.4 RBUF and TBUF Summary

Table 104 summarizes and contrasts the RBUF and TBUF operations.

Table 104. Summary of RBUF and TBUF Operations

Operation	RBUF	TBUF
Allocate element	<p>SPI-4</p> <p>Hardware allocates an element upon receipt of a non-idle Control Word, or when a previous element becomes full and another Data Word arrives with no intervening Control Word. Any available element in the SPI-4 partition may be allocated, however, elements are guaranteed to be handed to threads in the order they arrive.</p> <p>CSIX</p> <p>Hardware allocates an element upon receipt of RxSof asserted. Any available element in the CSIX Control or CSIX Data partition may be allocated (according to the type), however, elements are guaranteed to be handed to threads in the order they arrive.</p>	<p>Microengine allocates an element. Because the elements are transmitted in FIFO order (within each TBUF partition), the Microengine can keep the number of the next element in software.</p>
Fill element	<p>SPI-4</p> <p>Hardware fills the element with Data Words.</p> <p>CSIX</p> <p>Hardware fills the element with Payload.</p>	<p>Microcode fills the element from DRAM using <code>dram[tbuf_wr]</code> instruction and from Microengine registers using <code>msf[write]</code> instruction.</p>
Set element valid	<p>SPI-4</p> <p>Set valid by hardware when either it becomes full or when a Control Word is received.</p> <p>CSIX</p> <p>Set valid by hardware when the number of bytes in Payload Length have been received.</p>	<p>The element's Transmit Valid bit is set. This is done by a write to the TBUF_Element_Control_\$_# CSR (\$is A or B, # is the element number).</p>

Table 104. Summary of RBUF and TBUF Operations (Continued)

Operation	RBUF	TBUF
Remove data from element	Microcode moves data from the element to DRAM using <code>dram[rbuf_rd]</code> instruction and to Microengine registers using <code>msf[read]</code> instruction.	Hardware transmits information from the element to the Tx pins. Transmission of elements is in FIFO order within each partition; that is an element will be transmitted only when all preceding elements in that partition have been transmitted. Choice of element to transmit among partitions is round-robin.
Return element to Free List	Microcode writes to Rx_Element_Done with the number of the element to free.	Microengine software uses the <code>TX_Sequence_n</code> CSRs to track elements that have been transmitted.

8.5 CSIX Flow Control Interface

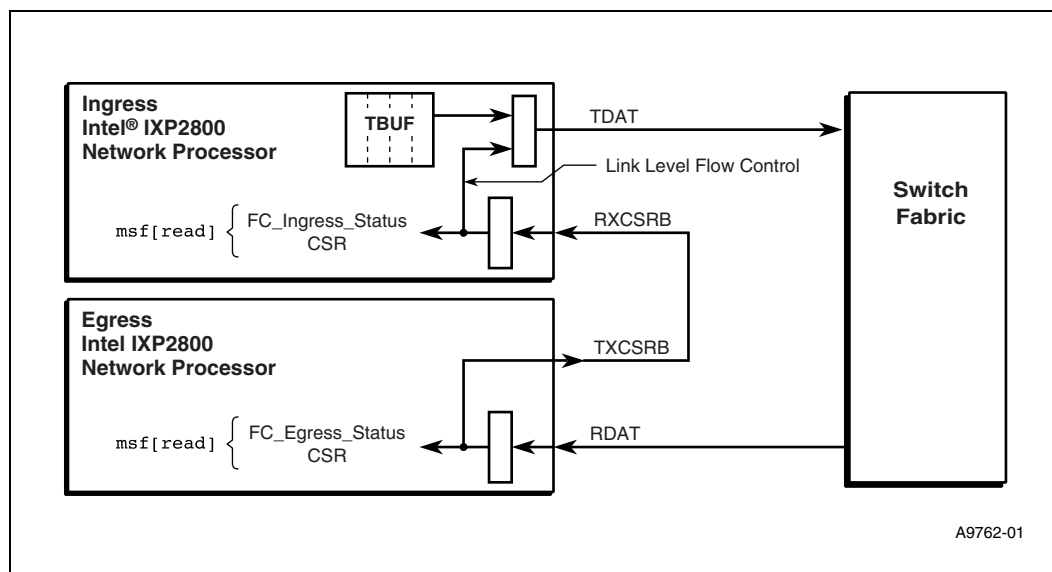
This section describes the Flow Control Interface. [Section 8.2](#) and [Section 8.3](#) of this chapter also contain descriptions of how those functions interact with Flow Control. There are two modes—Full Duplex, where flow control information goes from Egress IXP2800 Network Processor to the Ingress IXP2800 Network Processor, and Simplex mode, where the information from the Switch Fabric is sent directly to the Ingress IXP2800 Network Processor, and from the Egress IXP2800 Network Processor to the Switch Fabric.

8.5.1 TXCSRB, RXCSRB

TXCSRB and **RXCSRB** are used only in Full Duplex mode. (See [Figure 97](#).) They send information from the Egress IXP2800 Network Processor to the Ingress IXP2800 Network Processor for two reasons:

1. Pass the CSIX Ready Field (link-level flow control) from the Switch Fabric to the Ingress IXP2800 Network Processor. The information is used by the Ingress IXP2800 Network Processor's transmit control logic to stop transmission of CFrames to the Switch Fabric.
2. Set the value of the Ready field sent from the Ingress IXP2800 Network Processor to the Switch Fabric. This is to inform the Switch Fabric to stop transmitting CFrames to the Egress IXP2800 Network Processor, based on receive buffer resource availability in the Egress IXP2800 Network Processor.

Figure 97. CSIX Flow Control Interface — TXCSRB and RXCSRB



The information transmitted on **TXCSRB** can be read in **FC_Egress_Status** CSR, and the information received on **RXCSRB** can be read in **FC_Ingress_Status** CSR.

TXCSRB/RXCSRB signals carry the Ready information in a serial stream. Four bits of data are carried in 10 clock phases, LSB first, as shown in Table 105.

Table 105. SRB Definition by Clock Phase Number

Clock Cycle Number	Description	
	Source of bit on Egress IXP2800 Network Processor (TXCSRB)	Use of bit on Ingress IXP2800 Network Processor (RXCSRB)
0–5	Framing information. Data is 000001; this pattern allows the Ingress IXP2800 Network Processor to get synchronized to the serial stream regardless of the data values.	
6	Most recently received Control Ready from a CFrame Base Header. Also visible in FC_Egress_Status[SF_CReady] .	When 0—Stop sending Control CFrames to the Switch Fabric. When 1—OK to send Control CFrames to the Switch Fabric. Also visible in FC_Ingress_Status[SF_CReady] .
7	Most recently received Data Ready from a CFrame Base Header. Also visible in FC_Egress_Status[SF_DReady] .	When 0—Stop sending Data CFrames to the Switch Fabric. When 1—OK to send Data CFrames to the Switch Fabric. Also visible in FC_Ingress_Status[SF_DReady] .
8	RBUFF or FCEFIPO are above high water mark. Also visible in FC_Egress_Status[TM_CReady] .	Place this bit in the Control Ready bit of all outgoing CSIX Base Headers. Also visible in FC_Ingress_Status[TM_CReady] .
9	RBUFF is above high water mark. Also visible in FC_Egress_Status[TM_DReady] .	Place this bit in the Data Ready bit of all outgoing CSIX Base Headers. Also visible in FC_Ingress_Status[TM_DReady] .



The Transmit Data Ready bit sent from Egress to Ingress IXP2800 Network Processor will be deasserted if the following condition is met.

- RBUF CSIX Data partition is full, based on **HWM_Control[RBUF_D_HWM]**.

The Transmit Control Ready bit sent from Egress to Ingress IXP2800 Network Processor will be deasserted if either of the following conditions is met.

- RBUF CSIX Control partition is full, based on **HWM_Control[RBUF_C_HWM]**.
- FCEFIFO full, based on **HWM_Control[FCEFIFO_HWM]**.

8.5.2 FCIFIFO, FCEFIFO

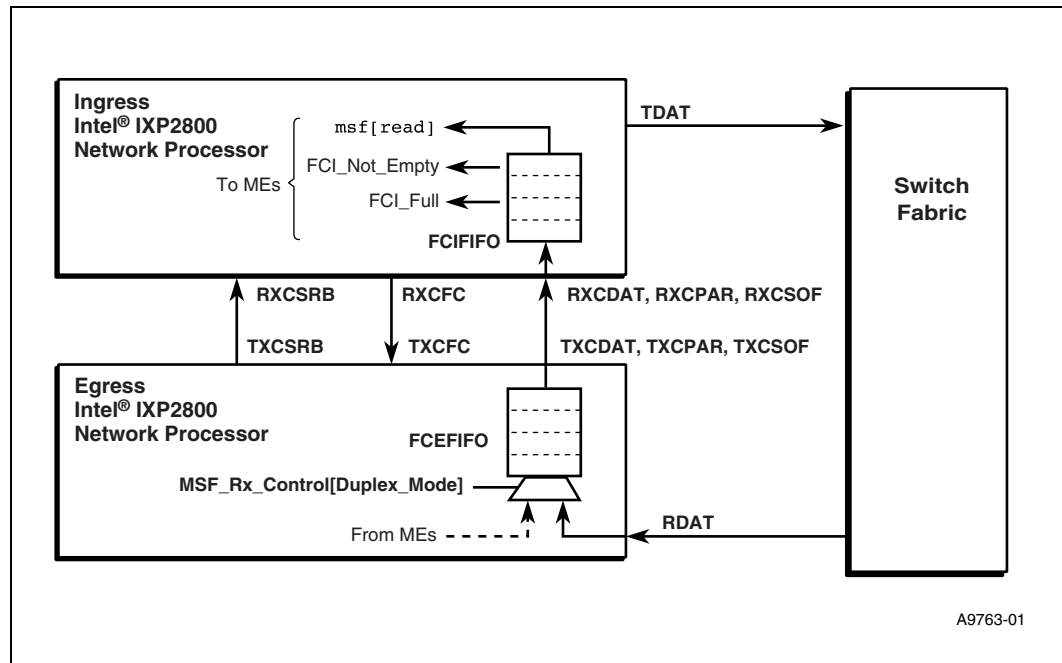
FCIFIFO and FCEFIFO are 1 Kbyte (256 entry x 32-bit) buffers for the flow control information. FCEFIFO holds data while it is being transmitted off of the Egress IXP2800 Network Processor. FCIFIFO holds data received into the Ingress IXP2800 Network Processor until Microengines can read it. There are two usage models for the FIFOs—Full Duplex Mode and Simplex Mode, selected by **MSF_Rx_Control[Duplex_Mode]**.

8.5.2.1 Full Duplex CSIX

In Full Duplex Mode, the information from the Switch Fabric is sent to the Egress IXP2800 Network Processor and must be communicated to the Ingress IXP2800 Network Processor via **TXCSRB/RXCSRB**. CSIX CFrames received from the Switch Fabric on the Egress IXP2800 Network Processor are put into FCEFIFO based on the mapping in **CSIX_Type_Map** CSR (normally they will be the Flow Control CFrames). The entire CFrame is put in, including the Base Header and Vertical Parity field.

The CFrames are forwarded in a “cut-through” manner, meaning the Egress IXP2800 Network Processor does not wait for the entire CFrame to be received before forwarding. The Egress processor will corrupt the Vertical Parity of the CFrame being forwarded if either a Horizontal or Vertical Parity is detected during receive to inform the Ingress processor that an error occurred. The Ingress IXP2800 Network Processor checks both Horizontal Parity and Vertical Parity and will discard the entire CFrame if bad parity is detected. The signal protocol details of how the information is sent from the Egress IXP2800 Network Processor to the Ingress IXP2800 Network Processor is described in [Section 8.5.3](#). (See [Figure 98](#).)

Figure 98. CSIX Flow Control Interface — FCIFIFO and FCEFIFO in Full Duplex Mode



The Ingress IXP2800 Network Processor puts the CFrames into the FCIFIFO, including the Base Header and Vertical Parity fields. It does not make a CFrame visible in the FCIFIFO until the entire CFrame has been received without errors. If there is an error the entire CFrame is discarded and **MSF_Interrupt_Status[FCIFIFO_Error]** is set.

CFrames in the FCIFIFO of the Ingress IXP2800 Network Processor are read by Microengines, which use them to keep current VOQ Flow Control information. [How and where that information is stored and used is a software function and is application dependent.] The FCIFIFO supplies two signals to Microengines, which can be tested using the BR_STATE instruction.

1. **FCI_Not_Empty**—indicates that there is at least one CWord in the FCIFIFO. This signal stays asserted until all CWords have been read. [Note that when FCIFIFO is empty, this signal will not assert until a full CFrame has been received into FCIFIFO; as that CFrame is removed by the Microengine this signal will stay asserted until all CWords have been removed, including any subsequently received CFrames.]
2. **FCI_Full**—indicates that FCIFIFO is above the high water mark defined in **HWM_Control[FCIFIFO_Int_HWM]**.

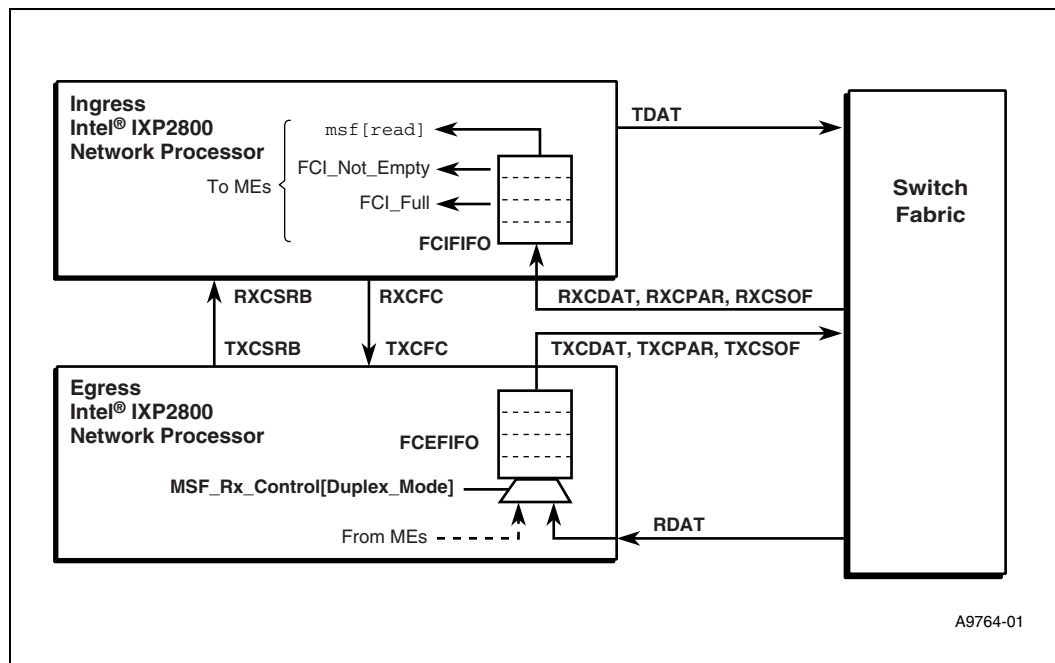
The Microengine that has been assigned to handle FCIFIFO must read the CFrame, 32 bits at a time, from the FCIFIFO using the **msf[read]** instruction to the FCIFIFO address; the length of the read can be anywhere from one to 16. The FCIFIFO handler thread must examine the Base Header to determine how long the CFrame is and perform the necessary number of reads from the FCIFIFO to dequeue the entire CFrame. If a read is issued to FCIFIFO when it is empty then an Idle CFrame will be read back (0x0000FFFF). Note that when FCIFIFO is receiving a CFrame, it does not make it visible until the entire CFrame has been received without errors.

The nearly-full signal is based on the high watermark programmed into **HWM_Control[FCIFIFO_Int_HWM]**. When asserted, this means that higher priority needs to be given to draining the FCIFIFO to prevent flow control from being asserted to the Egress IXP2800 Network Processor (by assertion of **RXCFC**).

8.5.2.2 Simplex CSIX

In Simplex Mode, the Flow Control signals are connected directly to the Switch Fabric; flow control information is sent directly from the Egress IXP2800 Network Processor to the Switch Fabric, and directly from the Switch Fabric to the Ingress IXP2800 Network Processor. (See [Figure 99](#).)

Figure 99. CSIX Flow Control Interface — FCIFIFO and FCEFIFO in Simplex Mode



The **TXCSR**/**RXCSR** pins are not used at all in Simplex Mode. The **RXCFC** and **TXCFC** pins are used for flow control in both Simplex and Duplex Modes.

The Egress IXP2800 Network Processor uses the **TXCSOF**, **TXCDAT**, and **TXCPAR** pins to send CFrames to the Switch Fabric.

The Ingress IXP2800 Network Processor uses the **RXCSOF**, **RXCDAT**, and **RXCPAR** pins to receive CFrames from the Switch Fabric (the Switch Fabric is expected to send Flow Control CFrames on these pins instead of the **RDAT** pins in Simplex Mode).

FC_Ingress_Status[SF_CReady] and **FC_Ingress_Status[SF_DReady]** bits are set are from the "Ready bits" received in all incoming CFrames received on this interface. Transmit hardware in the Ingress IXP2800 Network Processor uses the **FC_Ingress_Status[SF_CReady]** and **FC_Ingress_Status[SF_DReady]** bits to flow control the data and control transmit on **TDAT**.

CFrames in the FCIFIFO of the Ingress IXP2800 Network Processor are read by Microengines, which use them to keep current VOQ Flow Control information (this is the same as for Full Duplex Mode). The **FCI_Not_Empty** and **FCI_Full** status flags, as described in [Section 8.5.2.1](#) let the

Microengine know if the FCIFIFO has any CWords in it. When FCI_Full is asserted **FC_Ingress_Status[TM_CReady]** will be deasserted; that bit is put into the Ready field of CFrames going to the Switch Fabric, to inform it to stop sending Control CFrames.

Flow Control CFrames to the Switch Fabric are put into FCEFIFO, instead of into TBUF as in the Full Duplex Mode case. In this mode, the Microengines create CFrames and write them into FCEFIFO using `msf[write]` instruction to the FCEFIFO address; the length of the write can be anywhere from one to 16. The Microengine creating the CFrame must put a header that conforms to CSIX Base Header format in front of the message (in order to inform the hardware how many bytes to send).

The Microengine must first test if there is room in FCEFIFO by reading **FC_Egress_Status[FCEFIFO_Full]** status bit. After the CFrame has been written to FCEFIFO, the Microengine writes to **FCEFIFO_Validate** register to indicate that the CFrame should be sent out on **TXCDAT**. This is required to prevent underflow by insuring that the entire CFrame is in FCEFIFO before it can be transmitted. A validated CFrame at the head of FCEFIFO will be started on **TXCDAT** if **FC_Egress_Status[SF_CReady]** is asserted, and held off if it is deasserted. However, once started the entire CFrame is sent regardless of changes in **FC_Egress_Status[SF_CReady]**. **FC_Egress_Status[SF_DReady]** is ignored in controlling FCEFIFO.

FC_Egress_Status[TM_CReady] and **FC_Egress_Status[TM_DReady]** are placed by hardware into the Base Header of those outgoing CFrames. Horizontal and Vertical parity are created by hardware.

If there is no valid CFrame in FCEFIFO, or if **FC_Egress_Status[SF_CReady]** is deasserted, then idle CFrames are sent on **TXCDAT**. The idle CFrames also carry **FC_Egress_Status[TM_CReady]** and **FC_Egress_Status[TM_DReady]** in the Base Header Ready Field. In all cases the Switch Fabric must honor the "ready bits" to prevent overflowing RBUF.

8.5.3 TXCDAT/RXCDAT, TXCSOF/RXCSOF, TXCPAR/RXCPAR, and TXCFC/RXCFC

TXCDAT and **RXCDAT**, along with **TXCSOF/RXCSOF** and **TXCPAR/RXCPAR** are used to send CSIX Flow Control information from the Egress IXP2800 Network Processor to the Ingress IXP2800 Network Processor.

The protocol is basically the same as CSIX-LI, but with only four data signals.

TXCSOF is asserted to indicate start of a new CFrame. The format is the same as any normal CFrame—Base Header, followed by Payload and Vertical Parity, the only difference is that each CWord is sent on **TXCDAT** in four cycles, most significant bits first. **TXCPAR** carries odd parity for each four bits of data. The transmit logic also creates valid Vertical Parity at the end of the CFrame, with one exception. If the Egress IXP2800 Network Processor detected an error on the CFrame, it will create bad Vertical parity so that the Ingress IXP2800 Network Processor will detect that and discard it.

The Egress IXP2800 Network Processor sends CFrames from FCEFIFO in cut-through manner. If there is no data in FCEFIFO then the Egress IXP2800 Network Processor alternates sending Idle CFrames and Dead Cycles. [Note that FCIFIFO never enqueues Idle CFrames in either Full Duplex or Simplex Modes. The transmitted Idle CFrames are injected by the control state machine, not taken from the FCEFIFO.]



The Ingress IXP2800 Network Processor asserts **RXCFC** to indicate that FCIFIFO is full, as defined by **HWM_Control[FCIFIFO_Ext_HWM]**. The Egress IXP2800 Network Processor, upon receiving that signal asserted, will complete the current CFrame, and then transmit Idle CFrames until **RXCFC** deasserts. During that time the Egress IXP2800 Network Processor can continue to buffer Flow Control CFrames in FCEFIFO, however if that fills further CFrames mapped to FCEFIFO will be discarded.

Note: If there is no Switch Fabric present, this port could be used for interchip message communication. FC pins must connect between network processors as in Full Duplex Mode. Set **MSF_RX_CONTROL[DUPLEX_MODE] = 0** and **MSF_TX_CONTROL[DUPLEX_MODE] = 0** (Simplex) and **FC_STATUS_OVERRIDE=0x3ff**. MEs write CFrames to the FCEFIFO CSR as in Simplex Mode. The **RXCFC** and **TXCFC** pins must be connected between network processors to provide flow control.

8.6 Deskew and Training

This section describes the mechanisms used for deskewing and training.

There are three methods of operation that can be used, based on the application requirements.

1. Static Alignment — the receiver latches all data and control signals at a fixed point in time relative to clock.
2. Static Deskew — the receiver latches each data and control signal at a programmable point in time relative to clock. The programming value for each signal is characterized for a given system design and loaded into deskew control registers at system boot time.
3. Dynamic Deskew — the transmitter periodically sends a training pattern, which the receiver uses to automatically select the optimal timing point for each data and control signal. The timing values are loaded into the deskew control registers by the training hardware.

The IXP2800 Network Processor supports all three methods. There are three groups of high speed pins which this applies, as shown in [Table 106](#), [Table 107](#), and [Table 108](#). The groups are defined by which clock signal is used.

Table 106. Data Deskew Functions

Clock	Signals	IXP2800 Network Processor Operation
RCLK	RDAT	<ol style="list-style-type: none"> 1. Sample point for each pin is programmed in Rx_Deskew. 2. Deskew values set automatically when training pattern (Section 8.6.1) is received and is enabled in Train_Data[Ignore_Training].
	RCTL	
	RPAR	
	RPROT	
TCLK	TDAT	<ol style="list-style-type: none"> 1. Send training pattern <ul style="list-style-type: none"> • under software control (write to Train_Data[Continuous_Train] or Train_Data[Single_Train]) • when TSTAT input has framing pattern for more than 32 cycles and enabled in Train_Data[Train_Enable].
	TCTL	
	TPAR	
	TPROT	

Table 107. Calendar Deskew Functions

Clock	Signals	IXP2800 Network Processor Operation
RSCLK	RSTAT	<ol style="list-style-type: none"> Used to indicate need for data training on receive pins by forcing to continual framing pattern (write to Train_Data[RSTAT_En]). Send training pattern under software control (write to Train_Calendar[Continuous_Train] or Train_Calendar[Single_Train]).
TSCLK	TSTAT	<ol style="list-style-type: none"> Sample point for each pin is set in Rx_Deskew, either by manual programming or automatically. Deskew values set automatically when training pattern (Section) is received and is enabled in Train_Calendar[Ignore_Training]. Received continuous framing pattern can be used to initiate data training (Train_Data[Detect_No_Calendar]), and/or interrupt the Intel XScale® core.

Table 108. Flow Control Deskew Functions

Clock	Signals	IXP2800 Network Processor Operation
RXCLK	RXCFSOF	1. Sample point for each pin is programmed in Rx_Deskew . 2. Deskew values set automatically when training pattern (Section 8.6.2) is received and is enabled in Train_Flow_Control[Ignore_Training] . Note 1, 2
	RXCFSOF	
	RXCFSOF	
	RXCFSOF	
TXCLK	TXCFSOF	1. Send training pattern <ul style="list-style-type: none">under software control (write to Train_Flow_Control[Continuous_Train] or Train_Flow_Control[Single_Train])when TXCFC input has been asserted for more than 32 cycles and enabled in Train_Flow_Control[Train_Enable]. Note 1, 2
	TXCFSOF	
	TXCFSOF	
	TXCFSOF	
NOTES: 1. TXCFC is not trained. RXCFC is driven out relative to RXCLK ; TXCFC is received relative to TXCLK , but is treated as asynchronous. 2. RXCFC can be forced asserted by write to Train_Flow_Control[RXCFC_En] .		

8.6.1 Data Training Pattern

The data pin training sequence is shown in Table 109. This is a superset of SPI-4 training sequence, because it includes the **TPAR/RPAR** and **TPROT/RPOT** pins, which are not included in SPI-4.

Table 109. Data Training Sequence

Cycle (Note 4)	PROT	PAR	CTL	DATA															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 (Note 5)	0	x	1	0	x	x	0	0	0	0	0	0	0	0	0	a	b	c	d
2 to 11	0	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
12 to 21	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
20 α -18 to 20 α -9	0	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
20 α -8 to 20 α +1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

NOTES:

1. In cycle 1, x and abcd depend on the contents of the interval after the last preceding control word. This is an Idle Control Word.
2. α represents the number of repeats, as specified in SPI-4 specification. When the IXP2800 Network Processor is transmitting training sequences the value is in **Train_Data[Alpha]**.
3. On receive, the IXP2800 Network Processor will do dynamic deskew when **Train_Data[Ignore_Training]** is 0, and **RCTL** = 1 and **RDATA** = 0x0FFF for three consecutive samples. Note that **RPROT** and **RPAR** are ignored when recognizing the start of training sequence.
4. These are really phases (i.e., each edge of the clock is counted as one sample).
5. This cycle is valid for SPI4, it is not used in CSIX training.

8.6.2 Flow Control Training Pattern

This section defines training for the flow control pins (Table 110). These pins are normally used for CSIX flow control (Section 8.5), but can be programmed for use as SPI-4 Status Channel. The training pattern used is based on the usage.

The flow control pin training sequence when the pins are used for CSIX flow control is shown in Table 110.

Table 110. Flow Control Training Sequence

Cycle (Note 3)	XCSCF	XCDCAT				XCPAR	XCSCRB
		3	2	1	0		
1 to 10	1	1	1	0	0	0	0
11 to 20	0	0	0	1	1	1	1
20 α -19 to 20 α -10	1	1	1	0	0	0	0
20 α -9 to 20 α	0	0	0	1	1	1	1

NOTE:

1. α represents the number of repeats, as specified in SPI-4 specification. When the IXP2800 Network Processor is transmitting training sequences the value is in **Train_Flow_Control[Alpha]**.
2. On receive, the IXP2800 Network Processor will do dynamic deskew when **Train_Flow_Control[Ignore_Training]** is 0, and **RXCSCF** = 1, **RXCDCAT** = 0xC, **RXCPAR** = 0, and **RXCSCRB** = 0 for three consecutive samples.
3. These are really phases (i.e., each edge of the clock is counted as one sample).

The training sequence when the pins are used for SPI-4 Status Channel is shown in [Table 111](#). This is compatible to SPI-4 training sequence.

Table 111. Calendar Training Sequence

Cycle (Note 3)	XCDAT	
	1	0
1 to 10	0	0
11 to 20	1	1
20 α -19 to 20 α -10	0	0
20 α -9 to 20 α	1	1
NOTE: 1. α represents the number of repeats, as specified in SPI-4 specification. When the IXP2800 Network Processor is transmitting training sequences the value is in Train_Calendar[Alpha] . 2. On receive, the IXP2800 Network Processor will do dynamic deskew when Train_Calendar[Ignore_Training] is 0, and TCDAT = 0x0 for ten consecutive samples. 3. These are really phases (i.e., each edge of the clock is counted as one sample). 4. Only XCDAT[1:0] are included in training.		

8.6.3 Use of Dynamic Training

Dynamic training is done by cooperation of hardware and software as defined in this section.

The IXP2800 Network Processor will need training at reset or it loses training. Loss of training will typically be detected by parity errors on received data. [Table 112](#) lists the steps to initiate the training. SPI-4, CSIX Full Duplex, and CSIX Simplex cases follow similar but slightly different sequences. SPI-4 protocol uses the calendar status pins, **TSTAT/RSTAT** (or **RXCDAT/TXCDAT** if those are used for calendar status), as an indicator that data training is required. For CSIX use, the IXP2800 Network Processor uses a proprietary method of in-band signaling using Idle CFrames and Dead Cycles to indicate need for training.

Until the LVDS IOs are deskewed correctly, dip4 errors will occur. At startup, the receiver should request training followed by the transmitting device being sent training. The receiver should initially see received_training set and dip4 parity errors. The receiver should then clear the parity errors, wait for receive_training set and dip4_error cleared and check that all of the applicable RX_PHASEMON registers indicate no training errors. Then the LVDS IOs are properly trained.

Table 112. IXP2800 Network Processor Requires Data Training

Step	SPI-4 (IXP2800 Network Processor is Ingress Device)	CSIX (IXP2800 Network Processor is Egress Device)	
		Full Duplex	Simplex
1	Detect need for training (for example, reset or excessive parity errors)		
2	Force RSTAT (when using LVTTTL status channel) to continuous framing pattern (Write a 0 to Train_Data[RSTAT_En]), or force RXCDAT (when using LVDS status channel) to continuous training (Write a 1 to Train_Calendar[Continous_Train])	Force Transmission of Idle CFrames on Flow Control (Write a 1 to Train_Flow_Control[Force_FCIdle])	Force Transmission of Dead Cycles on Flow Control (Write a 1 to Train_Flow_Control[Force_FCDead])
3	Framer device detects RSTAT in continuous framing (when using LVTTTL status channel, or RXCDAT in continuous training (when using LVDS status channel))	Ingress IXP2800 Flow Control port detects Idle CFrames and sets Train_Flow_Control[Detect_FCIdle]	Switch Fabric detects Dead Cycles on Flow Control
4	Framer device transmits Training Sequence (IXP2800 receives on RDAT)	Ingress IXP2800 sends Dead Cycles on TDAT (if Train_Data[Dead_Enable_FCIdle] is set)	
5		Switch Fabric detects Dead Cycles on Data	
6		Switch Fabric transmits Training Sequence on Data	
7	When MSF_Interrupt_Status[Received_Training_Data] interrupt indicates training happened, MSF_Interrupt_Status[DIP4_ERR] write DIP4_ERR bit sent to clear previous errors and check that all of the applicable RX_PHASEMON registers indicate no training errors.		
	Write a 1 to Train_Data[RSTAT_En] or Write a 0 to Train_Calendar[Continous_Train]	Write a 0 to Train_Flow_Control[Force_FCIdle]	Write a 0 to Train_Flow_Control[Force_FCDead]

The second case is when the Switch Fabric or SPI-4 framing device indicates it needs Data training. [Table 113](#) lists that sequence.

Table 113. Switch Fabric or SPI-4 Framer Requires Data Training

Step	SPI-4	CSIX	
		Full Duplex	Simplex
1	Framer sends continuous framing code on IXP2800 calendar status pins TSTAT (when using LVTTTL status channel) or sends continuous training on IXP2800 calendar status pins RXCDAT (when using LVDS status channel).	Switch Fabric sends continuous Dead Cycles on Data.	Switch Fabric sends continuous Dead Cycles on Flow Control.
2	IXP2800 detects no calendar on TSTAT (when using LVTTTL status channel) or detects continuous training on RXCDAT (when using LVDS status channel), and sets Train_Data [Detect_No_Calendar]	Egress IXP2800 detects Dead Cycles and sets Train_Data [Detect_CDead]	Ingress IXP2800 detects Dead Cycles and sets Train_Flow_Control [Detect_FCDead]
3	IXP2800 transmits Training Pattern (if Train_Data [Train_Enable_TDAT] is set)	Egress IXP2800 Flow Control port sends continuous Dead Cycles if Train_Flow_Control [TD_Enable_CDead]	
4		Ingress IXP2800 Flow Control port detects continuous Dead Cycles and set Train_Flow_Control [Detect_FCDead]	
5		Ingress IXP2800 transmits continuous Training Sequence on data if Train_Data [Train_EN_FCDead]	
6	When Framer/Switch Fabric is trained it indicates that fact by reverting to normal operation.		
	Framer stops continuous framing code on calendar status pins.	Switch Fabric stops continuous Dead Cycles on Data.	Switch Fabric stops continuous Dead Cycles on Flow Control.

The IXP2800 Network Processor will need training at reset, or if it loses training. Loss of training will typically be detected by parity errors on received flow control information. [Table 114](#) lists the steps to initiate the training. CSIX Full Duplex, and CSIX Simplex cases follow similar but slightly different sequences.

Table 114. IXP2800 Network Processor Requires Flow Control Training

Step	CSIX (IXP2800 Network Processor is Ingress Device)	
	Full Duplex	Simplex
1	Force TXCFC pin asserted (Write a 0 to Train_Flow_Control [RXCFC_En])	Force Data pins to continuous Dead Cycles (Write a 1 to Train_Data[Force_CDead])
2	Egress IXP2800 Network Processor Flow Control port detects RXCFC sustained assertion and sets Train_Flow_Control [Detect_TXCFC_Sustained]	Switch Fabric detects Dead Cycles on Data
3	Ingress IXP2800 Network Processor transmits Training Sequence on Flow Control pins (if Train_Flow_Control [Train_Enable_CFC] is set)	Switch Fabric transmits Training Sequence on Flow Control pins.
4	When MSF_Interrupt_Status[Received_Training_FC] interrupt indicates training happened and all of the applicable RX_PHASEMON registers indicate no training errors, write CSR bits set in Step 1 to inactive value.	
	Write a 1 to Train_Flow_Control [RXCFC_En]	Write a 1 to Train_Data[Force_CDead]

The last case is when the Switch Fabric indicates it needs Flow Control training. [Table 115](#) lists that sequence.

Table 115. Switch Fabric Requires Flow Control Training

Step	Simplex (IXP2800 Network Processor is Egress Device)
1	Switch Fabric sends continuous Dead Cycles on Data.
2	Egress IXP2800 Network Processor detects Dead Cycles and sets Train_Data [Detect_CDead]
3	Egress IXP2800 Network Processor transmits Training Sequence on Flow Control pins (if Train_Flow_Control [Train_Enable_CDead] is set)
4	Switch Fabric, upon getting trained stops continuous Dead Cycles on Data.



8.7 CSIX Startup Sequence

This section defines the sequence required to startup the CSIX interface.

8.7.1 CSIX Full Duplex

8.7.1.1 Ingress IXP2800

1. On reset, FC_STATUS_OVERRIDE[Egress_Force_En] is set to force the Ingress IXP2800 to send Idle CFrames with low CReady and DReady bits to the Egress IXP2800 over TXCSRB.
2. The Microengine or the Intel XScale® core writes a 1 to MSF_Rx_Control[RX_En_C] so that Idle CFrames can be received.
3. The Microengine or the Intel XScale® core polls on MSF_Interrupt_Status[Detected_CSIX_Idle] to see when the first Idle CFrame is received. The Intel XScale® core may use the Detected_CSIX_Idle Interrupt if MSF_Interrupt_Enable[Detected_CSIX_Idle] is set.
4. When the first Idle CFrame is received, ME or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Egress_Force_En] to deactivate SRB Override or writes 2'b11 to FC_STATUS_OVERRIDE[7:6] ([TM_CReady] and [TM_DReady]). This will inform the Egress IXP2800 that the Switch Fabric has sent an Idle CFrame and the Ingress IXP2800 has detected it.

8.7.1.2 Egress IXP2800

1. On reset, FC_STATUS_OVERRIDE[Ingress_Force_En] is set.
2. The Microengine or the Intel XScale® core writes a 1 to MSF_Tx_Control[Transmit_Idle] and MSF_Tx_Control[Transmit_Enable] so that Idle CFrames with low CReady and Dready bits are sent over TDAT.
3. The Microengine or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Ingress_Force_En]. The Egress IXP2800 will then be sending Idle CFrames with CReady and DReady according to what is received on RXCSRB from the Ingress IXP2800. If the Egress IXP2800 has not detected an Idle CFrame, low TM_CReady and TM_DReady bits will be transmitted over its TXCSRB pin. If it has detected an Idle CFrame, the TM_CReady and TM_DReady bits are high. The TM_CReady and TM_DReady bits received on RXCSRB by the Ingress IXP2800 are used in the Base Headers of CFrames transmitted over TDAT.
4. The Microengine or the Intel XScale® core polls on FC_Ingress_Status[TM_CReady] and FC_Ingress_Status[TM_DReady]. When they are seen active, ME or the Intel XScale® core writes a 1 to MSF_Tx_Control[TX_En_CC] and MSF_Tx_Control[TX_En_CD]. Egress IXP2800 then resumes normal operation. Likewise, when the Switch Fabric recognizes Idle CFrames with "ready bits" high, it will assume normal operation.

8.7.1.3 Single IXP2800

1. The Microengine or the Intel XScale® core writes a 1 to MSF_Tx_Control[Transmit_Idle] and MSF_Tx_Control[Transmit_Enable] so that Idle CFrames with low CReady and DReady bits are sent over TDAT.
2. The Microengine or the Intel XScale® core writes a 1 to MSF_Rx_Control[RX_En_C] so that Idle CFrames can be received.
3. The Microengine or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Ingress_Force_En].
4. The Microengine or the Intel XScale® core polls on MSF_Interrupt_Status[Detected_CSIX_Idle] to see when the first Idle CFrame is received. The Intel XScale® core may use the Detected_CSIX_Idle Interrupt if MSF_Interrupt_Enable[Detected_CSIX_Idle] is set.
5. When the first Idle CFrame is received, the Microengine or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Egress_Force_En] to deactivate SRB Override or writes 2'b11 to FC_STATUS_OVERRIDE[7:6] ([TM_CReady and TM_DReady]).
6. The Microengine or the Intel XScale® core writes a 1 to MSF_Tx_Control[TX_En_CC] and MSF_Tx_Control[TX_En_CD]. IXP2800 resumes normal operation.

8.7.2 CSIX Simplex

8.7.2.1 Ingress IXP2800

1. On reset, FC_STATUS_OVERRIDE[Egress_Force_En] is set to force Ingress IXP2800 to send Idle CFrames with low CReady and DReady bits to Switch Fabric over TXCDAT.
2. The Microengine or the Intel XScale® core writes a 1 to MSF_Rx_Control[RX_En_C] so that Idle CFrames can be received.
3. The Microengine or the Intel XScale® core polls on MSF_Interrupt_Status[Detected_CSIX_Idle] to see when the first Idle CFrame is received. The Intel XScale® core may use the Detected_CSIX_Idle Interrupt if MSF_Interrupt_Enable[Detected_CSIX_Idle] is set.
4. When the first Idle CFrame is received, the Microengine or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Egress_Force_En]. Idle CFrames with "ready bits" high will be transmitted over TXCDAT. Ingress IXP2800 may resume normal operation.

8.7.2.2 Egress IXP2800

1. On reset, FC_STATUS_OVERRIDE[Ingress_Force_En] is set.
2. The Microengine or the Intel XScale® core writes a 1 to MSF_Tx_Control[Transmit_Idle] and MSF_Tx_Control[Transmit_Enable] so that Idle CFrames with low CReady and DReady bits are sent over TDAT.
3. The Microengine or the Intel XScale® core polls on MSF_Interrupt_Status[Detected_CSIX_FC_Idle] to see when the first Idle CFrame is received. The Intel XScale® core may use the Detected_CSIX_FC_Idle Interrupt if MSF_Interrupt_Enable[Detected_CSIX_FC_Idle] is set.
4. When the first Idle CFrame is received, ME or the Intel XScale® core writes a 0 to FC_STATUS_OVERRIDE[Ingress_Force_En] to deactivate SRB Override.

5. The Microengine or the Intel XScale® core polls on FC_Ingress_Status[TM_CReady] and FC_Ingress_Status[TM_DReady]. When they are seen active, the Microengine or the Intel XScale® core writes a 1 to MSF_Tx_Control[TX_En_CC] and MSF_Tx_Control[TX_En_CD]. Egress IXP2800 then resumes normal operation. Likewise, when the Switch Fabric recognizes Idle CFrames with "ready bits" high, it will assume normal operation.

8.7.2.3 Single IXP2800

Both CSIX startup routines described above will be needed to complete the CSIX startup sequence. Using Simplex mode on a single IXP2800 with RDAT, TDAT and RXCDAT, TXCDAT using CSIX, there are essentially two independent CSIX receive and transmit busses.

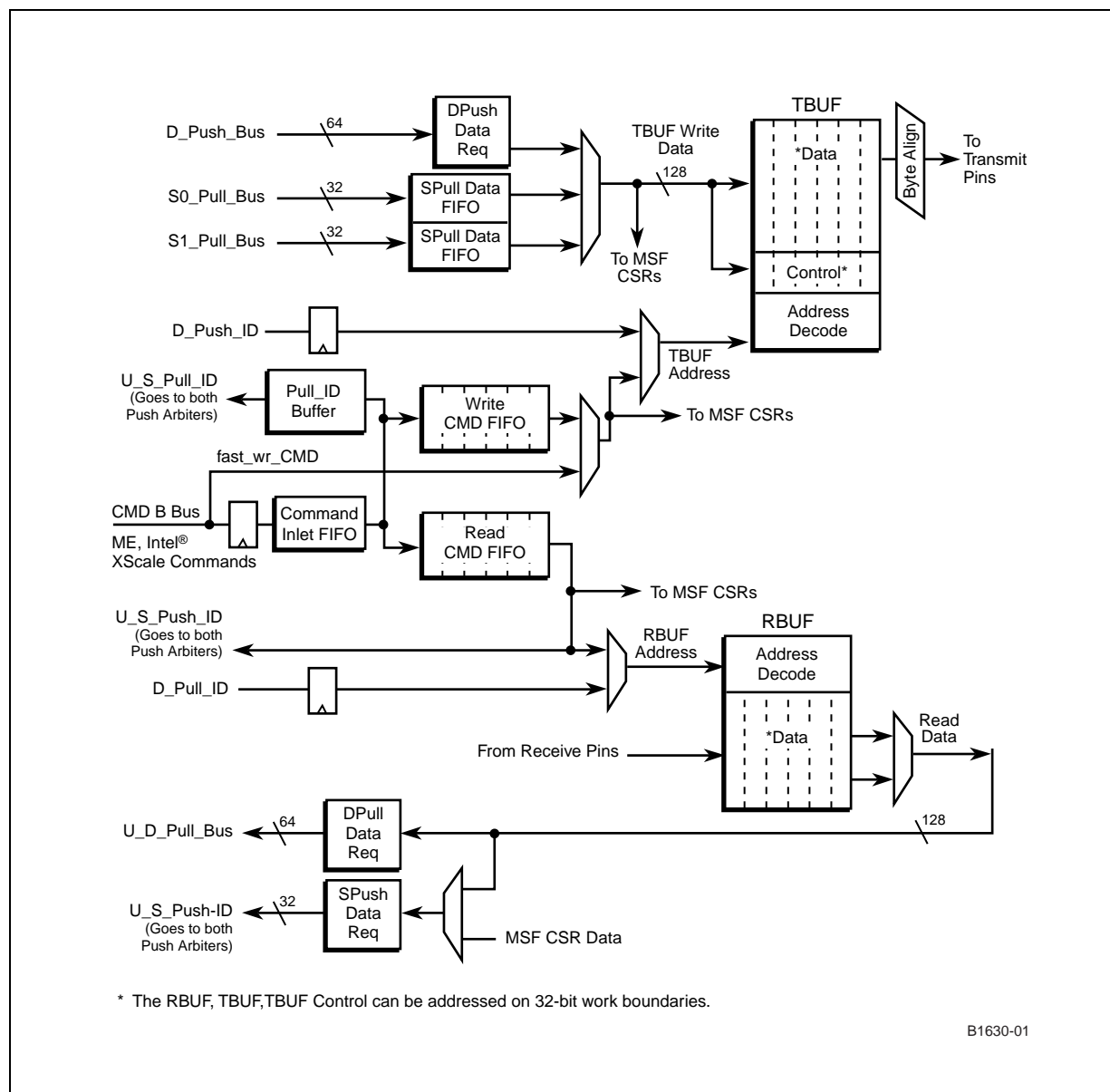
8.8 Interface to Command and Push and Pull Busses

The block diagram in [Figure 100](#) shows the interface of the MSF to the command and push and pull busses.

Data transfers to and from the TBUF/RBUF are done in the following cases (refer to section for details):

- [RBUF or MSF CSR to Microengine S Transfer In Register for instruction:](#)
- [Microengine S Transfer Out Register to TBUF or MSF CSR for instruction:](#)
- [Microengine to MSF CSR for instruction:](#)
- [From RBUF to DRAM for instruction:](#)
- [From RBUF to DRAM for instruction:](#)

Figure 100. MSF to Command and Push and Pull Busses Interface Block Diagram



8.8.1 RBUF or MSF CSR to Microengine S Transfer In Register for instruction:

```
msf[read, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token
```

For transfers to a Microengine, the MSF acts as a target. Commands from Microengines and the Intel XScale® core are received on the command bus. The commands are checked to see if they are targeted to the MSF. If so, they are enqueued into the Command Inlet FIFO, and then moved to the Read Cmd FIFO. When the Command Inlet FIFO is nearly full, it asserts a signal to the command arbiters. The command arbiters prevent further commands to the MSF until after the full signal is asserted. The RBUF element or CSR specified in the address field of the command is read and the data is registered in the SPUSH_DATA Register. The control logic then arbitrates for S_PUSH_BUS, and when granted, it drives the data.

8.8.2 Microengine S Transfer Out Register to TBUF or MSF CSR for instruction:

```
msf[write, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token
```

For transfers from Microengine, the MSF acts as a target. Commands from Microengines are received on the two command busses. The commands are checked to see if they are targeted to the MSF. If so, they are enqueued into the Command Inlet FIFO, and then moved to the Write Cmd FIFO. When the Command Inlet FIFO is nearly full, it asserts a signal to the command arbiters. The command arbiters prevent further commands to the MSF until after the full signal is asserted. The control logic then arbitrates for S_PULL_BUS, and when granted, it receives and registers the data from the Microengine into the S_PULL_DATA register. It then writes that data into the TBUF element or CSR specified in the address field of the command.

8.8.3 Microengine to MSF CSR for instruction:

```
msf[fast_write, src_op_1, src_op_2]
```

For fast write transfers from the Microengine, the MSF acts as a target. Commands from Microengines are received on the two command busses. The commands are checked to see if they are targeted to the MSF. If so, they are enqueued into the Command Inlet FIFO, and then moved to the Write Cmd FIFO. When the Command Inlet FIFO is nearly full, it asserts a signal to the command arbiters. The command arbiters prevent further commands to the MSF until after the full signal is asserted. The control logic uses the address and data, both found in the address field of the command. It then writes the data into the CSR specified.

8.8.4 From RBUF to DRAM for instruction:

```
dram[rbuf_rd, --, src_op1, src_op2, ref_cnt], indirect_ref
```

For the transfers to DRAM, the RBUF acts like a slave. The address of the data to be read is given in D_PULL_ID. The data is read from RBUF and registered in the D_PULL_DATA register. It is then multiplexed and driven to the DRAM channel on D_PULL_BUS.

8.8.5 From DRAM to TBUF for instruction:

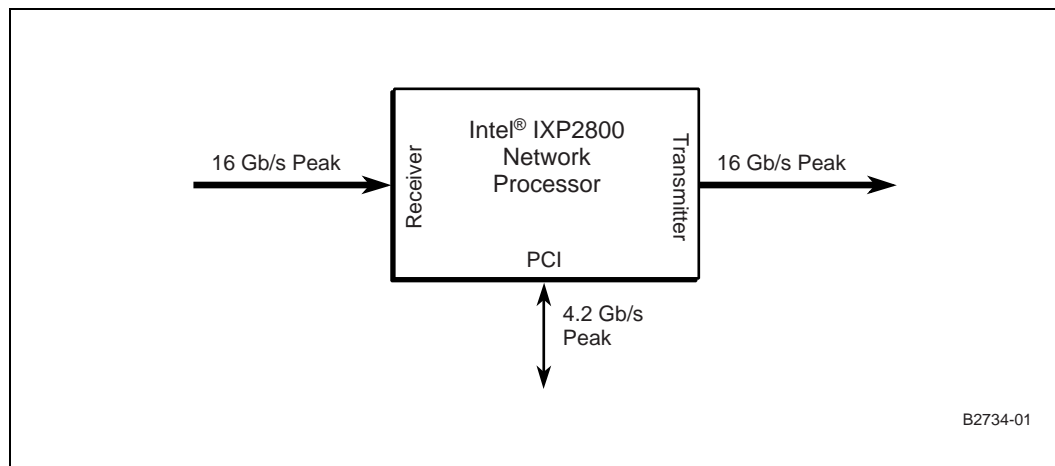
```
dram[tbuf_wr, --, src_op1, src_op2, ref_cnt], indirect_ref
```

For the transfers from DRAM, the TBUF acts like a slave. The address of the data to be written is given in D_PUSH_ID. The data is registered and assembled from D_PUSH_BUS, and then written into TBUF.

8.9 Receiver and Transmitter Interoperation with Framers and Switch Fabrics

The Intel® IXP2800 Network Processor can process data received at a peak rate of 16 Gb/s and transmit data at a peak rate of 16 Gb/s. In addition, data may be received and transmitted via the PCI bus at an aggregate peak rate of 4.2 Gb/s, as shown in Figure 101.

Figure 101. Basic I/O Capability of the Intel® IXP2800



The network processor's receiver and transmitter can be independently configured to support either an SPI-4.2 framer interface or a fabric interface consisting of DDR LVDS signaling and the CSIX-L1 protocol. The dynamic training sequence of SPI-4.2, used for de-skewing the signals, has been optionally incorporated into the fabric interface.

"SPI-4.2 is an interface for packet and cell transfer between a physical layer (PHY) device and a link layer device, for aggregate bandwidths of OC-192 ATM and Packet over SONET/SDH (POS), as well as 10 Gb/s Ethernet applications."¹ "CSIX-L1 is the Common Switch Interface. It defines a physical interface for transferring information between a traffic manager (Network Processor) and a switching fabric..."² The network processor adopts the protocol of CSIX-L1, but uses a DDR LVDS physical interface rather than an LVCMOS or HSTL physical interface.

SPI-4.2 supports up to 256 port addresses, with independent flow control for each. For data received by the PHY and passed to the link layer device, flow control is optional. The flow control mechanism is based upon independent pools of credits, corresponding to 16-byte blocks, for each port.

1. "System Packet Interface Level 4 (SPI-4) Phase 2: OC-192 System Interface for Physical and Link Layer Devices," Implementation Agreement: OIF-SPI4-02.0, Optical Internetworking Forum
2. "CSIX-L1: Common Switch Interface Specification-L1," CSIX

The CSIX-L1 protocol supports 4096 ports and 256 unicast classes of traffic. It supports various forms of multicast and 256 multicast queues of traffic. The protocol supports independent link-level flow control for data and control traffic and supports virtual output queue (VOQ) flow control for data traffic.

8.9.1 Receiver and Transmitter Configurations

The network processor receiver and transmitter *independently* support three different configurations:

- Simplex (SPI-4.2 or CSIX-L1 protocol), described in [Section 8.9.1.1](#).
- Hybrid simplex (transmitter only, SPI-4.2 data path, and CSIX-L1 protocol flow control), described in [Section 8.9.1.2](#).
- Dual NPU, full duplex (CSIX-L1 protocol), described in [Figure 8.9.1.3](#).

Additionally, the *combined* receiver and transmitter support a single NPU, full-duplex configuration using two different protocols:

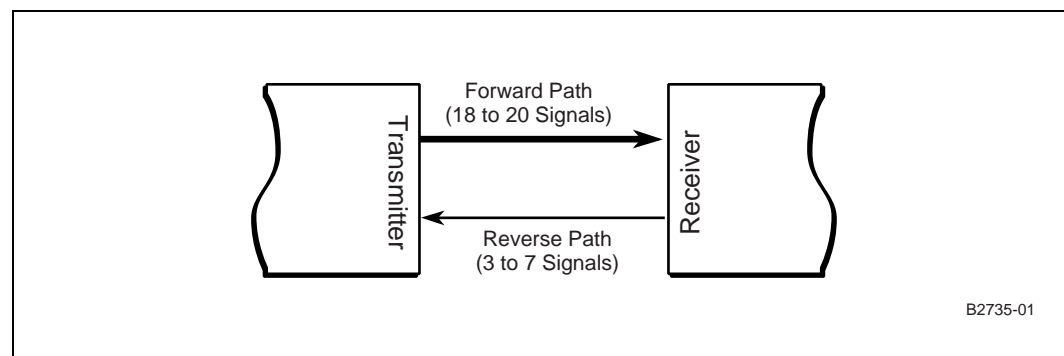
- Multiplexed SPI-4.2 protocol, described in [Section 8.9.1.4](#).
- CSIX-L1 protocol, described in [Section 8.9.1.5](#).

In both the simplex and hybrid simplex configurations, the path receiving from a framer, fabric, or NPU is independent of the path transmitting to a framer, fabric, or NPU. In a full duplex configuration, the receiving path forwards CSIX-L1 control information for the transmit path and vice versa.

8.9.1.1 Simplex Configuration

In the simplex configuration, as shown in [Figure 102](#), the reverse path provides control information to the transmitter. This control information may include flow control information and requests for dynamic training sequences.

Figure 102. Simplex Configuration



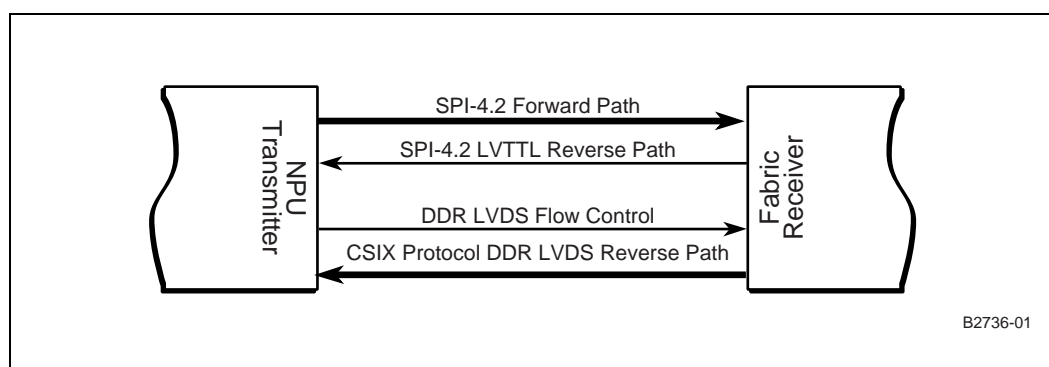
The SPI-4.2 mode of the simplex configuration supports an LVTTTL reverse path or status interface clocked at up to 125 MHz or a DDR LVDS reverse path or status interface clocked at up to 500 MHz. The SPI-4.2 mode status interface consists of a clock signal and two data signals.

The CSIX-L1 protocol mode of the simplex configuration supports a full-duplex implementation of the CSIX-L1 protocol, but no Data CFrames are transferred on the reverse path and the reverse path is a quarter of the width of the forward path. The CSIX-L1 protocol mode supports a DDR LVDS reverse path interface clocked at up to 500 MHz. The CSIX-L1 protocol mode reverse path control interface consists of a clock signal, four data signals, a parity signal, and a start-of-frame signal.

8.9.1.2 Hybrid Simplex Configuration

In the hybrid simplex configuration, data transfers and link-level flow control is supported via the SPI-4.2 modes of the receiver and transmitter, as shown in Figure 103. Only the LVTTL SPI-4.2 status interface is supported in this configuration.

Figure 103. Hybrid Simplex Configuration



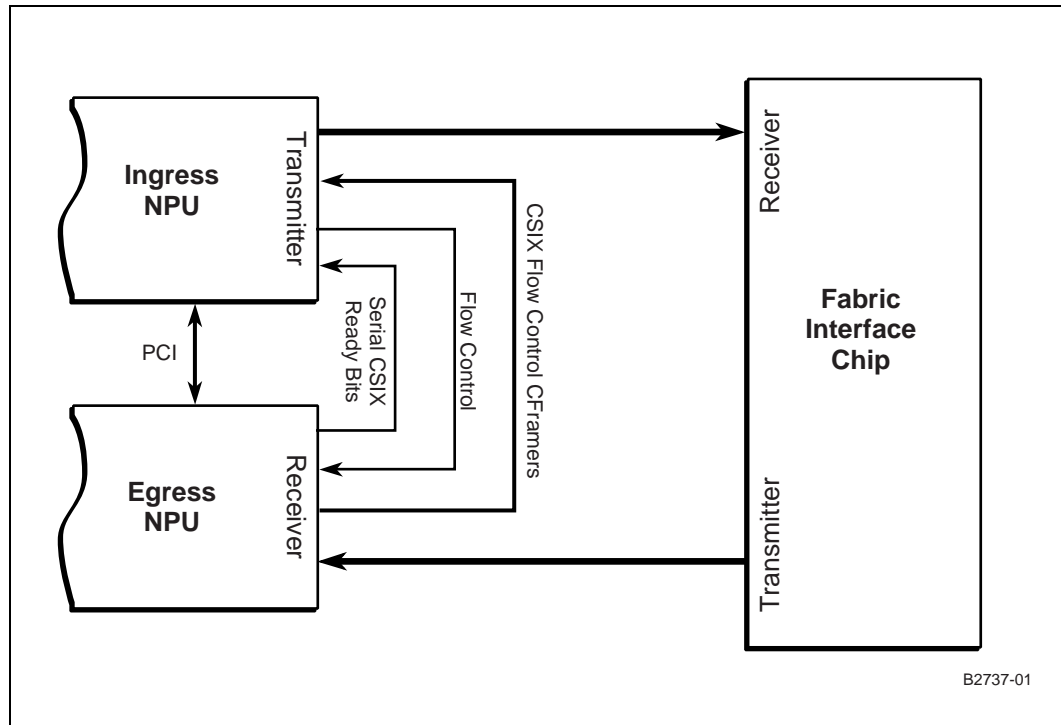
Virtual output queue flow control information (or other information) is delivered to the transmitter via the CSIX-L1 protocol via an interface similar to the reverse path of the CSIX-L1 protocol mode of the simplex configuration. Flow control for the CSIX-L1 CFrames is provided by an asynchronous LVDS signal back to the fabric and not by the "ready bits" of the CSIX-L1 protocol.

The hybrid simplex configuration for a fabric interface may be especially useful to implementers when an SPI-4.2 interface implementation is readily available. The CSIX-L1 protocol reverse path may not need to operate at a clock rate as aggressive as the SPI-4.2 interface and, as such, may be easier to implement than a full-rate data interface.

8.9.1.3 Dual NPU Full Duplex Configuration

In the dual NPU, full duplex configuration, an ingress NPU and an egress NPU are integrated to offer a single full duplex interface to a fabric, similar to the CSIX-L1 interface, as shown in Figure 104. This configuration provides an interface that is closest to the standard CSIX-L1 interface. It is easiest to bridge between this configuration and an actual CSIX-L1 interface.

Figure 104. Dual NPU, Full Duplex Configuration



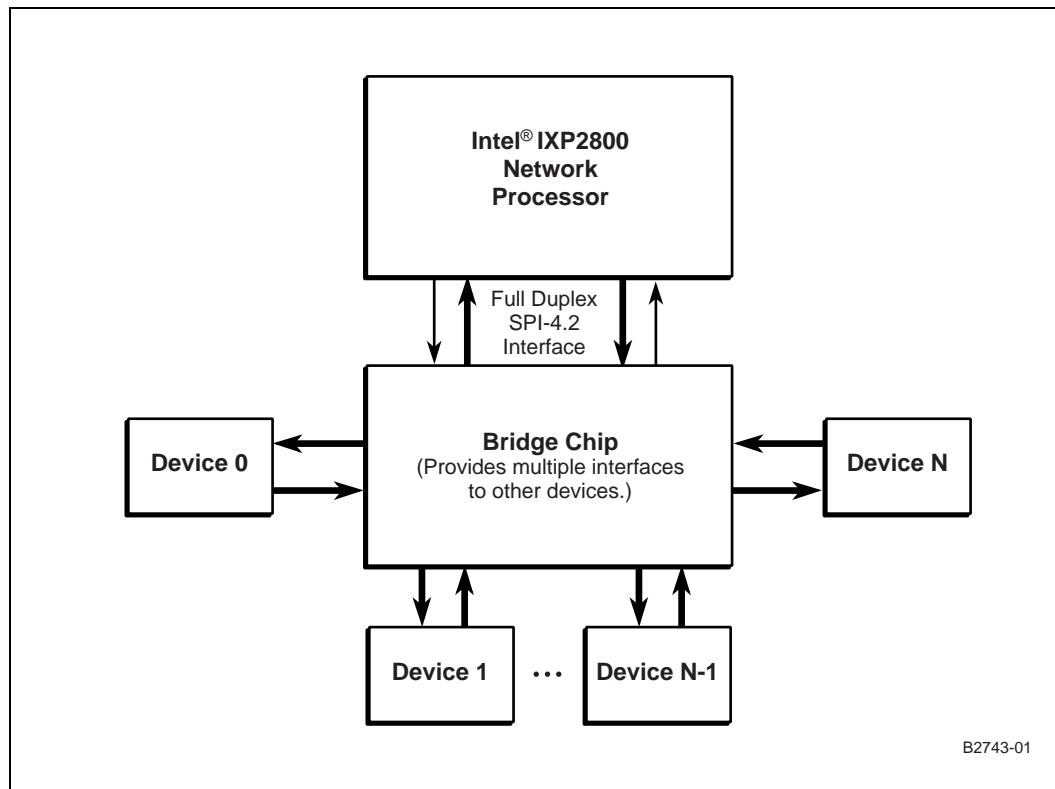
Flow control CFrames are forwarded by the egress NPU to the ingress NPU over a separate flow control interface. The bandwidth of this interface is a quarter of the primary interface offered to the fabric. A signal from ingress NPU to egress NPU provides flow control for this interface. (This interface is the same interface that was used in the hybrid simplex configuration.) A separate signal from egress NPU to ingress NPU provides the state of the CSIX-L1 "ready bits" that were received from the fabric, conveying the state of the fabric receiver, and those that should be sent to the fabric, conveying the state of the egress NPU receiver.

The PCI may be used to convey additional information between the egress NPU and ingress NPU.

8.9.1.4 Single NPU Full Duplex Configuration (SPI-4.2)

The single NPU, full duplex configuration (SPI-4.2 only) allows a single NPU to interface to multiple discrete devices, processing both the receiver and transmitter data for each, as shown in [Figure 105](#). Up to 256 devices can be addressed by the SPI-4.2 implementation. The bridge chip implements the specific interfaces for each of those devices.

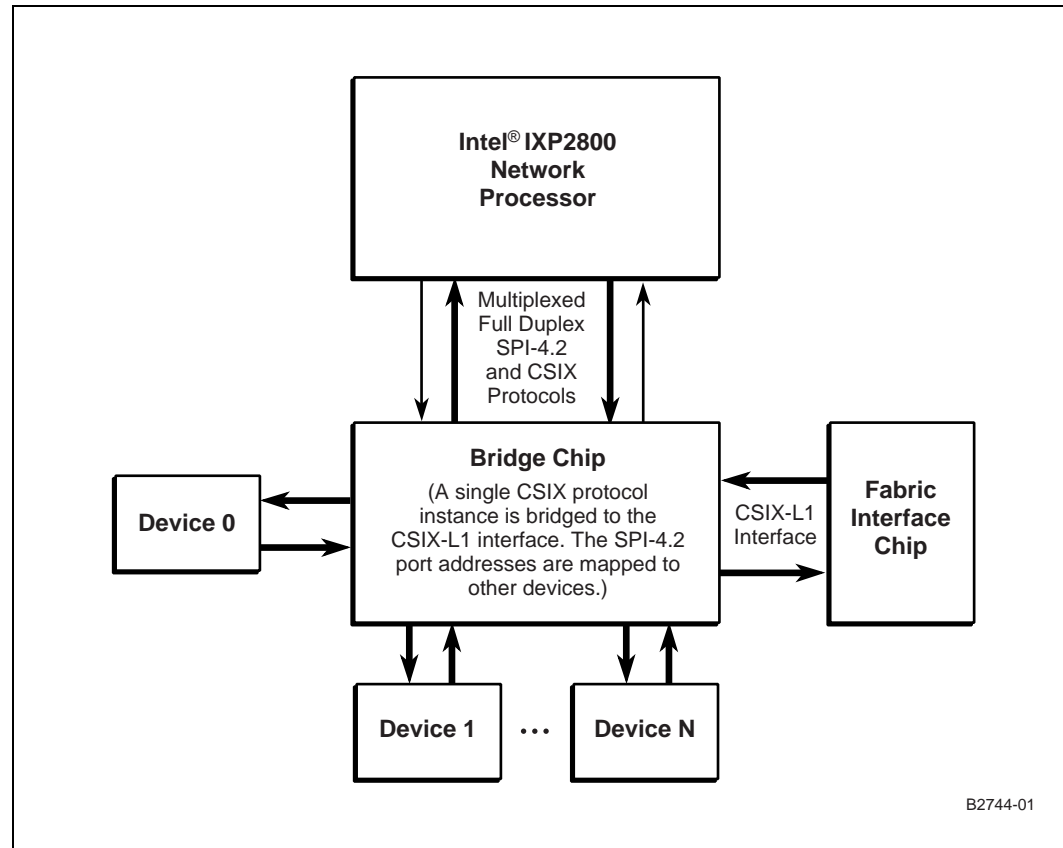
Figure 105. Single NPU, Full Duplex Configuration (SPI-4.2 Protocol)



8.9.1.5 Single NPU, Full Duplex Configuration (SPI-4.2 and CSIX-L1)

The Single NPU, Full Duplex Configuration (SPI-4.2 and CSIX-L1 Protocol) allows a single NPU to interface to a fabric via a CSIX-L1 interface and to multiple other discrete devices, as shown in Figure 106. The CSIX-L1 and SPI-4.2 protocols are multiplexed on the network processor receiver and transmitter interface. Independent processing and buffering resources are allocated to each protocol.

Figure 106. Single NPU, Full Duplex Configuration (SPI-4.2 and CSIX-L1 Protocols)



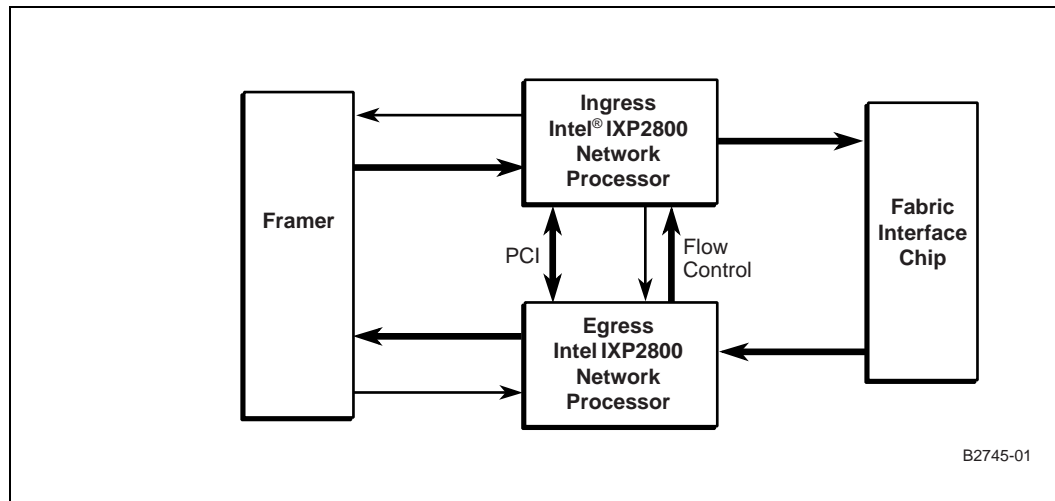
8.9.2 System Configurations

The receiver and transmitter configurations in the preceding [Section 8.9.1](#) enable several system designs, as shown in [Figure 107](#) through [Figure 111](#).

8.9.2.1 Framer, Single NPU Ingress and Egress, and Fabric Interface Chip

[Figure 107](#) illustrates the baseline system configuration consisting of the dual chip, full-duplex fabric configuration of network processors with a framer chip and a fabric interface chip

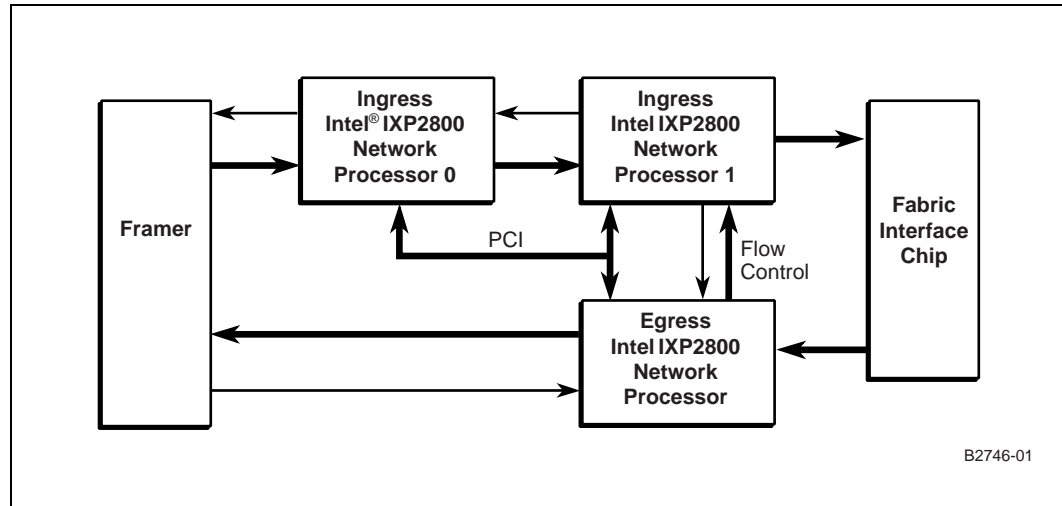
Figure 107. Framer, Single NPU Ingress, Single NPU Egress, and Fabric Interface Chip



8.9.2.2 Framer, Dual NPU Ingress, Single NPU Egress, and Fabric Interface Chip

If additional processing capacity is required in the ingress path, an additional network processor can be added to the configuration, as shown in Figure 108. The configuration of the interface between the two ingress network processors can use either the SPI-4.2 or CSIX-L1 protocol.

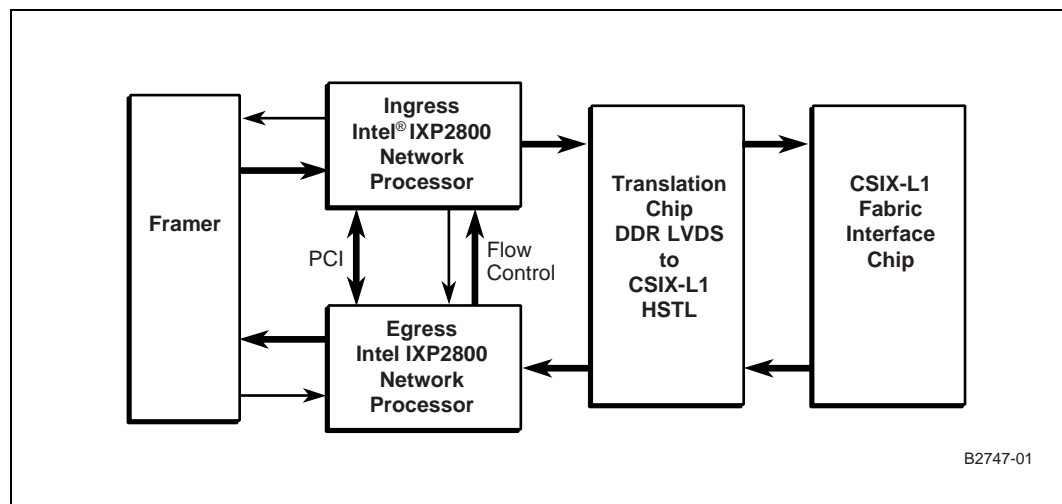
Figure 108. Framer, Dual NPU Ingress, Single NPU Egress, and Fabric Interface Chip



8.9.2.3 Framer, Single NPU Ingress and Egress, and CSIX-L1 Chips for Translation and Fabric Interface

To interface to existing standard CSIX-L1 fabric interface chips, a translation bridge can be employed, as shown in Figure 109. Translation between the network processor interface and standard CSIX-L1 is very simple by design.

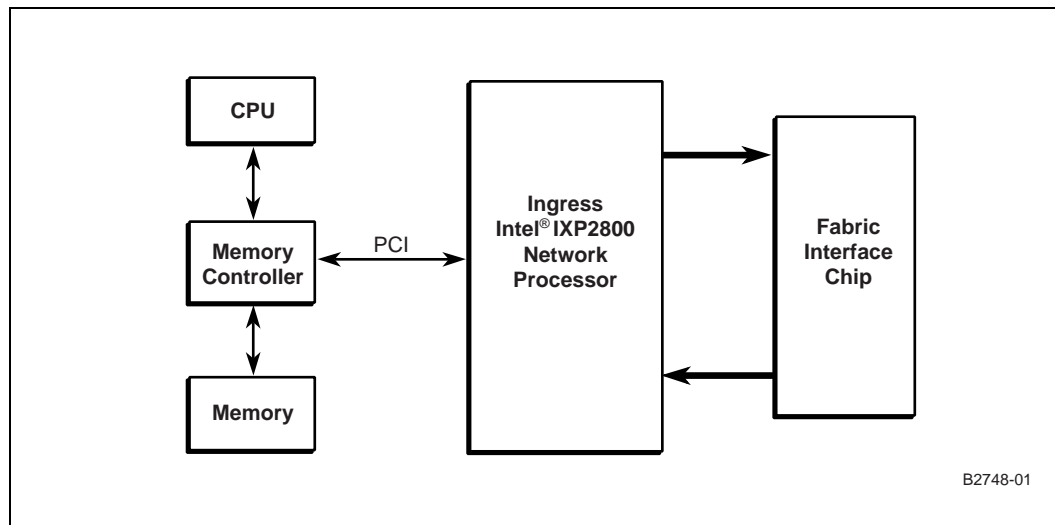
Figure 109. Framer, Single NPU Ingress, Single NPU Egress, CSIX-L1 Translation Chip and CSIX-L1 Fabric Interface Chip



8.9.2.4 CPU Complex, NPU, and Fabric Interface Chip

If a processor card requires access to the fabric, a single network processor can provide both ingress and egress access to the fabric for the processor via the PCI interface, as shown in Figure 110. In many cases the available aggregate peak bandwidth of 4.2 Gb/s is sufficient for the processor's capacity.

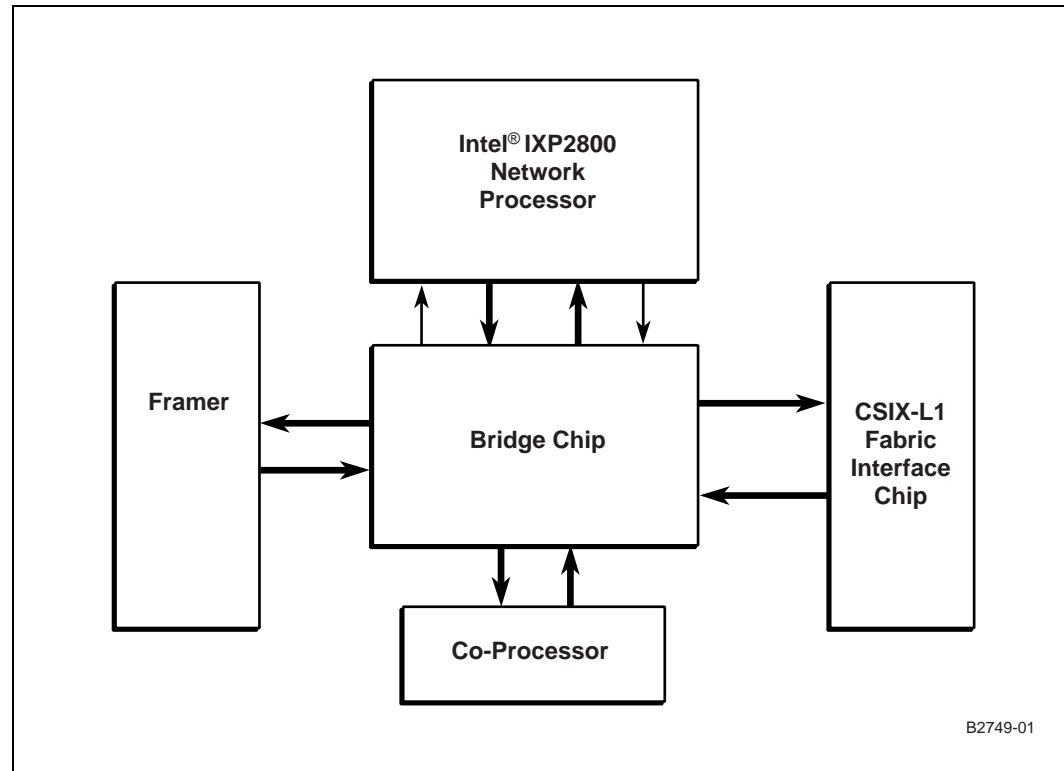
Figure 110. CPU Complex, NPU, and Fabric Interface Chips



8.9.2.5 Framer, Single NPU, Co-Processor, and Fabric Interface Chip

The network processor supports multiplexing the SPI-4.2 and CSIX-L1 protocols over its physical interface via a protocol signal. This capability enables using a bridge chip to allow a single network processor to support the ingress and egress paths between a framer and a fabric, provided the aggregate system bandwidth does not exceed the capabilities of that single network processor, as shown in [Figure 111](#).

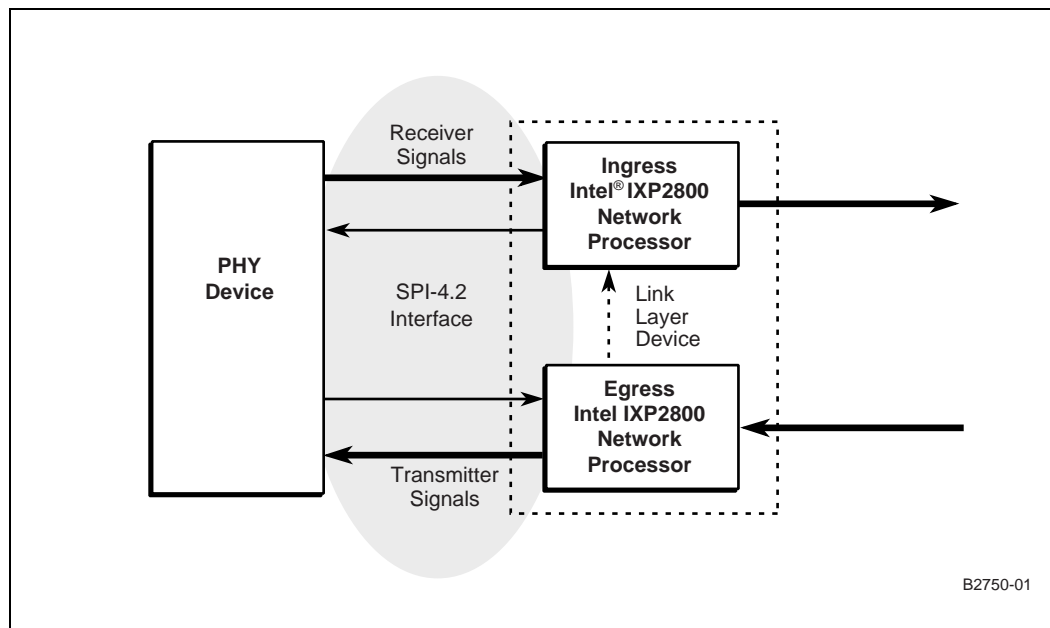
Figure 111. Framer, Single NPU, Co-Processor, and Fabric Interface Chip



8.9.3 SPI-4.2 Support

Data is transferred across the SPI-4.2 interface in variously-sized bursts and encapsulated with a leading and trailing control word. The control words provide annotation of the data with port address (0-255) information, start-of-packet and end-of-packet markers, and an error detection code (DIP-4). Data must be transferred in 16-byte integer multiples, except for the final burst of a packet.

Figure 112. SPI-4.2 Interface Reference Model with Receiver and Transmitter Labels Corresponding to Link Layer Device Functions



The status interface transfers state as an array of state or calendar, two bits per port, for all of the supported ports. The status information provides for reporting one of three status states for each port (satisfied, hungry, and starving) corresponding to credit availability for the port. The mapping of calendar offset to port is flexible. Individual ports may be repeated multiple times for greater frequency of update.

8.9.3.1 SPI-4.2 Receiver

The network processor receiver stores received SPI-4.2 bursts into receiver buffers. The buffers may be configured as 128 buffers of 64 bytes, 64 buffers of 128 bytes, or 32 buffers of 256 bytes. Information from the control words, the length of the burst, and the TCP checksum of the data are stored in an additional eight bytes of control storage. The buffers support storage of bursts containing an amount of data that is less than or equal to the buffer size. A burst that is greater than the configured size of the buffers is stored in multiple buffers. Each buffer is made available to software as it becomes filled.

As the filling of each buffer completes, the buffer is dispatched to a thread of a Microengine that has been registered in a free list of threads, and the eight bytes of control information are forwarded to the register context of the thread. If no thread is currently available, the receiver waits for a new thread to become available as other buffers are also filled (and then also have “waiting queues”).

As threads complete processing of the data in a buffer, the buffer is returned to a free list. Subsequently, the thread also returns to a separate free list. The return of buffers and threads to the free lists may occur in a different order than the order of their removal.

All SPI-4.2 ports sharing the interface have equal access to the buffering resources. Flow control can transition to a non-starving state when 25%, 50%, 75%, or 87.5% of the buffers are consumed, as configured by `HWM_Control[RBUF_S_HWM]`. At this point, the remaining buffers are available and, additionally, 2K bytes of packed FIFO (corresponding to 128 SPI-4.2 credits) are available for incoming data storage. If receiver flow control is expected to be asserted and for a sufficiently large number of ports and values of `MaxBurst1` or `MaxBurst2`, it may be necessary for the PHY device to discard credits already granted if a state of Satisfied is reported by the network processor to the device, treating the Satisfied state more as an XOFF state. Otherwise, excessive credits may be outstanding for the storage available and receiver overruns may occur.

For more information about the SPI-4.2 receiver, see [Section 8.2.7](#).

8.9.3.2 SPI-4.2 Transmitter

The network processor transmitter transfers SPI-4.2 bursts from transmitter buffers. The buffers may be configured as 128 buffers of 64 bytes, 64 buffers of 128 bytes, or 32 buffers of 256 bytes. The control word information and other control information for the burst are stored in additional control storage. The buffers are always transmitted in a fixed order. Software can determine the index of the last buffer transmitted, and keep track of the last buffer committed to the transmitter. The transmitter buffers are used as a ring, with the “get index” updated by the transmitter and the “put index” updated due to committing a buffer element to transmission.

Each transmit buffer supports a limited gather capability to stitch together a protocol header and a payload. The buffer supports independent prefix (or prepended) data and payload data. The prefix data can begin at any offset from 0 to 7 and have a length of from 0 to 31 bytes. The payload begins at an offset of 0 to 7 bytes from the next octal-byte boundary following the prefix and can fill out the remainder of the buffer. For more complicated merging or shifting of data within a burst, the data should be passed through a Microengine to perform any arbitrary merging and/or shifting.

Buffers may be statically allocated to different ports in an inter-leaved fashion so that bandwidth availability is balanced for each of the ports. Transmit buffers may be flagged to be skipped if no data is available for a particular port.

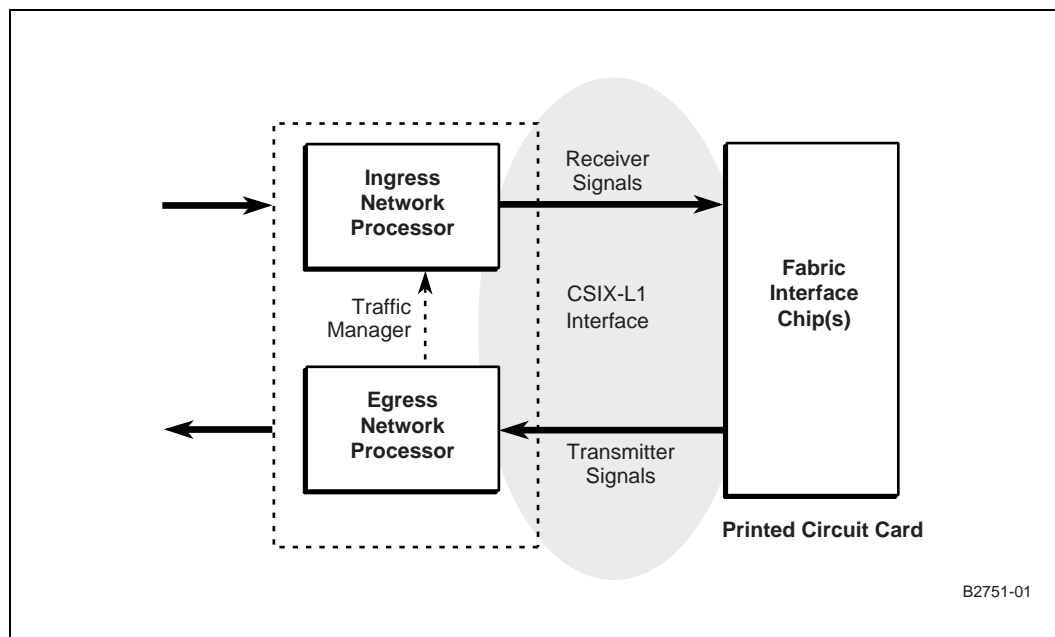
The transmitter scheduler, implemented on a Microengine, is responsible for reacting to the status information provided by the PHY device. The status information can be read via registers. The status information is available in two formats: a single status per register and status for 16 ports in a single register. For more information, see [Section 8.3.4](#).

8.9.4 CSIX-L1 Protocol Support

8.9.4.1 CSIX-L1 Interface Reference Model: Traffic Manager and Fabric Interface Chip

The CSIX-L1 protocol operates between a Traffic Manager and a Fabric Interface Chip(s) across a full-duplex interface. It supports mechanisms to interface to a fabric that avoid congestion using virtual output queue (VOQ) flow control and enables a fabric that offers lossless, non-blocking transfer of data from ingress port to egress ports. Both data and control information pass over the receiver and transmitter interfaces.

Figure 113. CSIX-L1 Interface Reference Model with Receiver and Transmitter Labels Corresponding to Fabric Interface Chip Functions



The Traffic Manager on fabric ingress is responsible for segmentation of packet data and scheduling the transmission of data segments into the fabric. The fabric on ingress is responsible for influencing the scheduling of data transmission through link-level flow control and Virtual Output Queue (VOQ) flow control so that the fabric does not experience blocking or data loss due to congestion. The fabric on egress is responsible for scheduling the transfer of data to the Traffic Manager according to the flow control indications from the Traffic Manager.

The CSIX-L1 protocol supports addressing up to 4096 fabric ports and identifies up to 256 classes of unicast traffic. It optionally supports multicast and broadcast traffic, supporting identification of up to 256 queues of such traffic. Virtual output queue flow control is supported at the ingress to the fabric and the egress from the fabric.

The standard CSIX-L1 interface supports interface widths of 32, 64, 94, and 128 bits. A single clocked transfer of information across the interface is called a CWord. The CWord size is the width of the interface.

Information is passed across the interface in CFrames. CFrames are padded out to an integer multiple of CWords. CFrames consist of a 2-byte base header, an optional 4-byte extension header, a payload of 1 to 256 bytes, padding, and a 2-byte vertical parity. Transfers across the interface are protected by a horizontal parity. When there is no information to pass over the interface, an alternating sequence of Idle CFrames and Dead Cycles are passed across the interface.

There are 16 possible codes for CFrame types. Each CFrame type is either a data CFrame or a control CFrame. Data CFrame types include Unicast, Multicast Mask, Multicast ID, Multicast Binary Copy, and Broadcast. Control CFrames include Flow Control.

CSIX-L1 supports independent link-layer flow control for data CFrames and control CFrames by using “ready bits” (CRdy and DRdy) in the base header. The response time for link-level flow control is specified to be 32 interface clock ticks, but allows for additional time to complete transmission of any CFrame already in progress at the end of that interval.

8.9.4.2 Intel® IXP2800 Support of the CSIX-L1 Protocol

The adaptation of the CSIX-L1 protocol to the network processor physical interface has been accomplished in a straightforward manner.

8.9.4.2.1 Mapping to 16-Bit Wide DDR LVDS

The CSIX-L1 interface is built in units of 32 data bits. For each group of 32 data signals, there is a clock signal (RxClk, TxClk), a start-of-frame signal (RxSOF, TxSOF) and a horizontal-parity signal (RxPar, TxPar). If the CWord or interface width is greater than 32 bits, the assertion of the Start-of-Frame signal associated with each group of 32 data bits is used to synchronize the transfers across the independently clocked individual 32-bit interfaces.

The network processor supports 32-bit data transfers across two transfers or clock edges of the SPI-4.2 16-bit DDR LVDS data interface. The CSIX-L1 RxSOF and TxSOF signals are mapped to the SPI-4.2 TCTL and RCTL signals. For the transfer of CFrames, the start-of-frame signal is asserted on only the first edge of the 32-bit transfer. (Assertion of the start-of-frame signal for multiple contiguous clock edges denotes the start of a de-skew training sequence as described below.)

Receiver logic for the interface should align the start of 32-bit transfers to the assertion of the start-of-frame signal. The network processor always transmits the high order bits of a 32-bit transfer on the rising edge of the transmit clock, but a receiver may de-skew the signals and align the received data with the falling edge of the clock. The network processor receiver always aligns the received data according to the assertion of the start-of-frame signal.

The network processor supports CWord widths of 32, 64, 96, and 128 bits. It will pad out CFrames (including Idle CFrames) and Dead Cycles according to this CWord width. The physical interface remains just 16 data bits. The start-of-frame signal is only asserted for the high order 16 bits of the first 32-bit transfer. It is not asserted for each 32-bit transfer. Support for multiple CWord widths is intended to facilitate implementation of Intel® IXP2800-to-CSIX-L1 translator chips and to facilitate implementation of chips with native network processor interfaces, but with wider internal transfer widths.

The network processor supports a horizontal parity signal (RPAR, TPAR). The horizontal parity signal covers the 16 data bits that are transferred on each edge of the clock. It does not cover 32 bits as in CSIX-L1. Support for horizontal-parity requires an additional physical signal beyond that required for SPI-4.2. Checking of the horizontal parity can be optionally disabled on reception. If a fabric interface chip does not support TPAR, then the checking of RPAR should be disabled.



The network processor supports a variation of the standard CSIX-L1 vertical parity. Instead of a single vertical XOR for the calculation of the vertical parity, the network processor can be configured to calculate as DIP-16 code, as documented within the SPI-4.2 specification (see Figure 6.8 of that document). If horizontal parity is not enabled for the interface, the use of the DIP-16 code is recommended to provide for better error coverage than that provided by a vertical parity.

8.9.4.2.2 Support for Dual Chip, Full-Duplex Operation

A dual-chip configuration of network processors consisting of an ingress and egress network processor, can present a full-duplex interface to a fabric interface chip, consistent with the expectations of the CSIX-L1 protocol. A flow control interface is supported between the ingress and egress chips to forward necessary flow control information from the egress network processor to the ingress network processor. Additional information can be transferred between the ingress and egress network processors through the PCI bus.

The flow control interface consists of a data transfer signal group, a serial signal for conveying the state of the CSIX-L1 "ready bits" (TXCSRB, RXCSRB), and a backpressure signal (TXCFC, RXCFC) to avoid overrunning the receiver in the ingress network processor. (The orientation of the signal names is consistent with the egress network processor, receiving CFrames from the fabric, and forwarding flow control information out through the transmit flow control pins.) The data transfer signal group consists of:

- four data signals (TXCDAT[0..3], RXCDAT[0..3])
- a clock (TXCCLK, RXCCLK)
- a start-of-frame signal (TXCSOF, RXCSOF)
- a horizontal-parity signal (TXCPAR, RXCPAR)

The network processor receiver forwards Flow Control CFrames from the fabric in a cut-through fashion over the flow control interface. The flow control interface has one-fourth of the bandwidth of the network processor fabric data interface. The Crdy bit in the base header of the CSIX-L1 protocol (link-level flow control) prevents overflowing of the FIFO for transmitting out the flow control interface from the egress network processor. The fabric can implement a rate limit on the transmission of Flow Control CFrames to the egress network processor, consistent with the bandwidth available on the flow control interface. With a rate limit, the fabric can detect congestion of Flow Control CFrames earlier, instead of waiting for the assertion of cascaded backpressure signals.

The CRdy and DRdy bits of CFrames sent across the flow control interface are set to 0 on transmission and ignored upon reception at the ingress network processor. If no CFrames are available to send from the egress network processor to the ingress network processor, an alternating sequence of Idle CFrames and Dead Cycles is sent from the egress to the ingress network processor, consistent with the CSIX-L1 protocol.

The state of the CRdy and DRdy bits sent to the egress network processor by the fabric and the state of the CRdy and DRdy bits that should be sent to the fabric by the ingress network processor, reflecting the state of the egress network processor buffering, are sent through the TXCSRB signal and received through the RXCSRB signal. A new set of bits are conveyed every 10 clock edges or five clock cycles, of the interface. A de-assertion of a "ready bit" is forwarded immediately upon processing the "ready bit". An assertion of a "ready bit" is forwarded only after all of the horizontal parities and the vertical parity of the CFrame are checked. A configuration of ingress and egress network processors is expected to respond to the de-assertion of a CRdy or DRdy bit within 32 clock cycles (RCLK), consistent with the formulation described for CSIX-L1.

The backpressure signal (TXCFC, RXCFC) is an asynchronous signal and is asserted by the ingress network processor to prevent overflow of the ingress network processor ingress flow control FIFO. If the egress network processor is so optionally configured, it will react to assertion of the backpressure signal for 32 clock cycles (64 edges) as a request for a de-skew training sequence to be transmitted on the flow control interface.

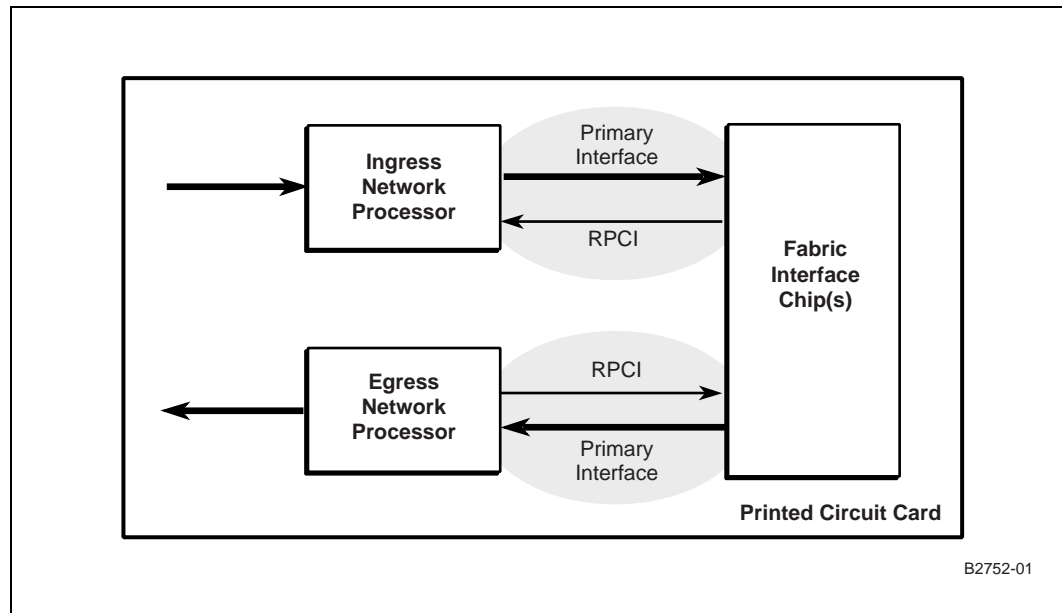
The flow control interface only supports a 32-bit CWord. Flow Control CFrames that are received by the egress network processor are stripped of any padding associated with large CWord widths and forwarded to the flow control interface.

The various options for parity calculation and checking supported on the data interface are supported on the flow control interface. Horizontal parity checking may be optionally disabled. The standard calculation of vertical parity may be replaced with a DIP-16 calculation.

8.9.4.2.3 Support for Simplex Operation

The network processor supports a mode of operation that supports the CSIX-L1 protocol, but offers an independent interface for the ingress and egress network processors. In this mode, the ingress and egress network processors each offer an independent full-duplex CSIX-L1 flavor of interface to the fabric, but the NPU-to-fabric interface on the egress network processor and the fabric-to-NPU interface of the ingress network processor are of reduced width, consisting of four (instead of 16) data signals. These narrow interfaces are referred to as Reverse Path Control Interfaces and use the same physical interface as the flow control interface in the dual-chip, full duplex configuration. They support the transfer of Flow Control CFrames and the CRdy and DRdy “ready” bits, but are not intended to support the transfer of data CFrames.

Figure 114. Reference Model for Intel® IXP2800 Support of the Simplex Configuration Using Independent Ingress and Egress Interfaces



The Reverse Path Control Interfaces (RPCI) support only the 32-bit CWord width of the dual chip, full duplex flow control interface. The variations of parity support provided by the data interface and the flow control interface are supported by the RPCI.

The transfer time of CFrames across the RPCI is four times that of the data interface. The latency of link-level flow control notifications depends on the frequency of sending new CFrame base headers. As such, the maximum size of CFrames supported on the RPCI should be limited to provide sufficient link-level flow control responsiveness.

The behavior of state machines for a full-duplex interface regarding interface initialization, link-level flow control, and requests to send a de-skew training sequence is supported by the data interface in combination with its reverse path control interface as if the two interfaces were equivalent to a full-duplex interface.

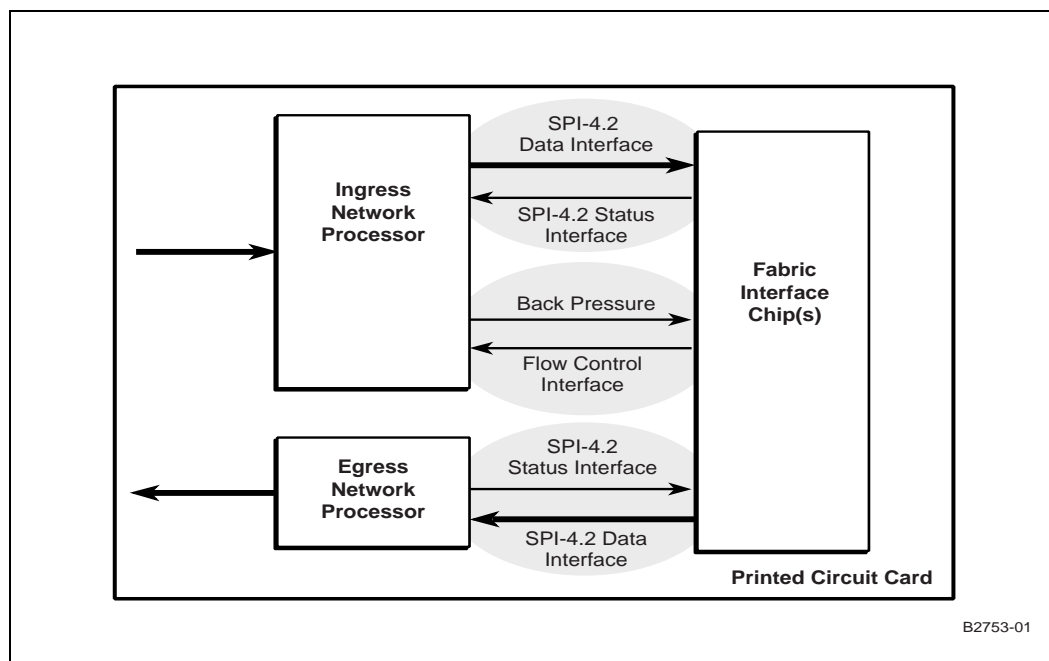
The simplex mode of interfacing to the ingress and egress network processor is an alternative to the dual chip full-duplex configuration. It provides earlier notification of Flow Control CFrame congestion within the ingress network processor and marginally less latency for delivery of Flow Control CFrames to the ingress network processor. It allows more of the bandwidth on the data interface to be used for the transfer of data CFrames as Flow Control CFrames are transferred on the RPCI.

The simplex configuration provides a straightforward mechanism for the egress network processor to send VOQ flow control to the fabric if the fabric supports such functionality. In the dual chip, full-duplex configuration, the egress network processor sends a request across the PCI to the ingress network processor, requesting that a Flow Control CFrame be sent to the fabric.

8.9.4.2.4 Support for Hybrid Simplex Operation

The SPI-4.2 interface may be used to transfer data to and from a fabric, although there is no standard protocol for such conveyance. The necessary addressing information for the fabric and egress network processor may be encoded within the address bits of the preceding control word or stored in the initial data words of the SPI-4.2 burst. The LVTTTL status interface may be used to provide link-level flow control for the data bursts. (The SPI-4.2 LVDS status interface cannot be used, because it shares the same pins with the fabric flow control interface.)

Figure 115. Reference Model for Hybrid Simplex Operation



The SPI-4.2 interface does not support a virtual output queue (VOQ) flow control mechanism. The Intel® IXP2800 Network Processor supports use of the CSIX-L1 protocol-based flow control interface (as used in the dual chip, full-duplex configuration) on the ingress network processor, while SPI-4.2 is operational on the data interface. This interface can provide VOQ flow control information from the fabric and allow the transmitter scheduler, implemented in a Microengine within the ingress network processor, to avoid sending data bursts to congested destinations.

The fabric should send alternating Idle CFrames and Dead Cycles when there are no Flow Control CFrames to transmit. The CRdy and DRdy “ready bits” should be set to 0 on transmission and are ignored on reception.

The fabric should respond to the RXCFC backpressure signal. In this mode of operation, the RXCSRB signal that would normally receive the state of the CRdy and DRdy “ready bits” is not used. If dynamic de-skew is configured on the interface, and the backpressure signal is asserted for 32 clock cycles, the fabric sends a (de-skew) training sequence on the flow control interface. It may be acceptable in this configuration to operate the flow control interface at a sufficiently low clock rate that dynamic de-skew is not required.

Operation in the hybrid simplex mode for the ingress network processor is slightly more taxing on the transmit scheduler computation than the homogenous CSIX-L1 protocol configurations. The status reported for the data interface must be polled by the transmit scheduler. In this configuration, the response to link-level flow control is performed in software and is slower than in the homogenous CSIX-L1 protocol configurations where it is accomplished in hardware.

Intel® reference software does not currently support this mode of fabric inter-operation.

8.9.4.2.5 Support for Dynamic De-Skew Training

The SPI-4.2 interface incorporates a training sequence for dynamic de-skew of its signals relative to the source synchronous clock. This training sequence has been extended and incorporated into the CSIX-L1 protocol support of the Intel® IXP2800 Network Processor.

The training pattern for the 16-bit data interface consists of 20 words, 10 repetitions of 0x0fff followed by 10 repetitions of 0xf000. The CTL and PAR signals are asserted for the first 10 words and de-asserted for the second 10 words. The PROT signal (see below) is de-asserted for the first 10 words and asserted for the second 10 words. A training sequence consists of “alpha” repetitions of the training pattern. The idle control word that precedes a training sequence in SPI-4.2 is not used in conjunction with the CSIX-L1 protocol. See [Section 8.6.1](#) for more information.

A receiver should detect a training sequence in the context of the CSIX-L1 protocol implementation by the assertion of the start-of-frame signal for three adjacent clock edges and the correct value on the data signals for those three adjacent clock edges.

A receiver may request a training sequence to be sent by transmitting continuous Dead Cycles on the interface. Reception of two adjacent Dead Cycles triggers the transmission of a training sequence in the opposite direction. If an interface is sending Dead Cycles and a training sequence becomes pending, the interface must send the training sequence at a higher priority than the Dead Cycles. Otherwise, a deadlocked situation may arise.

In the simplex configuration, the request for training, and the response to it, occur between a primary interface and its associated reverse path control interface. In the dual chip, full-duplex configuration, requests for training and Dead Cycles are encoded across the flow control interface as either continuous Dead Cycles or continuous Idle CFrames, both of which violate the standard CSIX-L1 protocol.



The training pattern for the flow control data signals consists of 10 nibbles of 0xc followed by 10 nibbles of 0x3. The parity and serial "ready bits" signal is de-asserted for the first 10 nibbles and asserted for the second 10 nibbles. The start-of-frame signal is asserted for the first 10 nibbles and de-asserted for the second 10 nibbles. See [Section 8.6.2](#) for more information.

When a training sequence is received, the receiver should update the state of the received CRdy and DRdy "ready bits" to a de-asserted state until they are updated by a subsequent CFrame.

8.9.4.3 CSIX-L1 Protocol Receiver Support

The Intel® IXP2800 Network Processor receiver support for the CSIX-L1 protocol is similar to that for SPI-4.2. CFrames are stored in the receiver data buffers. The buffers are configured to be of a size of 64, 128, or 256 bytes. The contents of the CFrame base header and extension header are stored in separate storage with the reception status of the CFrame. Unlike SPI-4.2 data bursts, the entire CFrame must fit into a single buffer. The receiver does not progress to the next buffer to store subsequent parts of a single CFrame. (The buffer is required only to be sufficiently large to accommodate the payload, not the header, the padding, or the vertical parity.) Designated CFrame types, typically Flow Control CFrames, are forwarded in cut-through mode directly to the flow control egress FIFO and not stored in the receiver buffers.

The receiver resources are separately allocated to the processing of data and control CFrames. Separate free lists of buffers and Microengine threads for each category of CFrame type are maintained. The size of the buffers in each resource pool is separately configurable. The mapping of CFrame type to data or control category is completely configurable via the CSIX_Type_Map register. This register also allows for any types to be designated for cut-through forwarding to the flow control egress FIFO. Typically, only the Flow Control CFrame type is configured in this way.

The receiver buffers are partitioned into two pools via MSF_Rx_Control[RBUF_Partition], providing 75% of the buffer memory (6 Kbytes) for data CFrames and 25% of the buffer memory (2 Kbytes) for control CFrames. The number of buffers available per pool depends on the configured buffer size. For 64-byte buffers, there are 96 and 32 buffers, respectively. For 128-byte buffers, there are 48 and 16 buffers, respectively. For 256-byte buffers, there are 24 and 8 buffers, respectively.

As with SPI-4.2, link-level flow control for a buffer pool can be asserted by configuration when buffer consumption reaches 25%, 50%, 75%, or 87.5% within that pool. The receiver has an additional 1024 bytes of packed FIFO storage for each traffic category to accept additional CFrames after link-level flow control (CRdy or DRdy) is asserted. Link-level flow control for control CFrames (CRdy) is also asserted if the flow-control egress FIFO contents exceeds a threshold as configured by HWM_Control[FCEFIFO_HWM]. The threshold may be set to 16, 32, 64, or 128 32-bit words. The total capacity of the FIFO is 512 32-bit words.

Within the base header, the receiver hardware processes the CRdy bit, the DRdy bit, the Type field, and the Payload Length. *Only the Flow Control Frame CFrame is expected to lack the 32-bit extension header.* The receiver hardware validates the vertical parity of the CFrame and only writes it to the receiver buffer if the write operation also includes payload data. The hardware supports configuration options for processing all 16 CFrame types. In all other respects, processing of the CFrame contents is done entirely by software. Variations in the CSIX-L1 protocol are supported that only affect the software processing. These variations might include address swapping (egress port address swapping with ingress port address) and use of reserve bits to encode start and end of packets.

When the network processor is configured to forward Flow Control Frame CFrames to the flow control egress FIFO, software does not process those CFrames. Processor interrupts occur if there are reception errors, but the actual CFrames are not made available for further processing.

8.9.4.4 CSIX-L1 Protocol Transmitter Support

The Intel® IXP2800 Network Processor transmitter support for the CSIX-L1 protocol is similar to that for SPI-4.2. The transmitter fetches CFrames from transmitter buffers. An entire CFrame must fit within a single buffer. In the case of SPI-4.2, the array of transmitter buffers operates as a single ring. In the case of CSIX-L1 protocol support, the array of buffers operates as two rings, one for data CFrames and another for control CFrames. The partitioning of the transmitter buffers is configured via `MSF_Tx_Control[TBUF_Partition]`. The portion of the aggregate transmitter buffer storage (8 Kbytes) allocated to data CFrames is 75% (6 Kbytes), with the remainder (2 Kbytes) allocated to control CFrames. The size of the buffers within each partition is independently configurable to a size of 64, 128, or 256 bytes. The payload size of CFrames sent from the buffers may vary from 1 to the size of the buffer.

The CSIX-L1 protocol link-level flow control operates directly upon the hardware that processes the two (control and data) transmitter rings. The transmitter services the two rings in round-robin order when allowed by link-level flow control. The transmitter transmits Idle CFrames and Dead Cycles according to the CSIX-L1 protocol if there are no CFrames to transmit.

Virtual output queue flow control is accommodated by a transmit scheduler implemented on a Microengine. In all three network processor ingress configurations, Flow Control CFrames are loaded by hardware into the flow control ingress FIFO. Two bits of state associated with this FIFO are distributed to all of the Microengines:

- The FIFO is non-empty.
- The FIFO contains more than a threshold amount of CFrame 32-bit words (`HWM_Control[FCIFIFO_Int_HWM]`).

Any Microengine can perform transmitter scheduling by sensing the state associated with the flow control ingress FIFO, using the branch-on-state instruction. If the FIFO is not empty, the transmit scheduler processes some of the FIFO by performing a read of the `FCIFIFO` registers. A single Microengine instruction can perform a block read of up to 16 32-bit words. The data for the read is likely to arrive after several subsequent scheduling decisions. The scheduler should incorporate the new information from the newly-read Flow Control CFrame(s) in its later scheduling decisions. If the FIFO state indicates that the threshold capacity has been exceeded, the scheduler should suspend further scheduling decisions until the FIFO is sufficiently processed, otherwise it risks making scheduling decisions with information that is too stale.

The responsiveness of the network processor to VOQ flow control depends on the length of the transmit pipeline, from transmit scheduler to CFrames on the interface signals. For rates at and above 10 Gb/s, the pipeline length is likely to be 32 to 64 CFrames, assuming four pipeline stages (schedule, de-queue, data movement, and transmit) and 8 to 16 CFrames concurrently processed per stage.

In the simplex configuration, the egress network processor can send CFrames over the Reverse Path Control Interface. The CFrames are loaded into the flow control egress FIFO by performing writes of 32-bit words to the `FCEFIFO` registers. The base header, the extension header, the payload, the padding, and a dummy vertical parity must be written to the FIFO. The transmitter hardware calculates the actual vertical parity as the CFrame is transmitted.



Note: The transmitter hardware for the transmitter buffers and the flow control egress FIFO expects that *only* the Flow Control CFrame type does not have an extension header of 32 bits. All other types have a 32-bit extension header. The hardware disregards the contents of the extension header or the payload.

The limited gather capability described for SPI-4.2 also is available for CFrames. A prefix header of up to 31 bytes and a disjoint payload is supported. The prefix header may start at an offset of 0 to 7 bytes. The payload may start at an offset of 0 to 7 bytes from the octal-byte boundary following the end of the prefix header. For more complicated merging or shifting of data within a CFrame, the data should be passed through a Microengine to perform any arbitrary merging and/or shifting.

8.9.4.5 Implementation of a Bridge Chip to CSIX-L1

The Intel® IXP2800 Network Processor support for the CSIX-L1 protocol in the dual chip, full-duplex configuration minimizes the difficulty in implementing a bridge chip to a standard CSIX-L1 interface. If dynamic de-skew training is not employed, the bridge chip can directly pass through the different CSIX-L1 protocol elements, CFrames, and Dead Cycles. The horizontal parity must be re-calculated on each side of the bridge chip. If the standard CSIX-L1 interface implements a CWord width that is greater than 32 bits, it must implement a synchronization mechanism for aligning the received 32-bit portions of the CWord before passing the CWord to the network processor.

For transmitting the standard CSIX-L1 interface, the bridge chip must assert the start-of-frame signal for each 32-bit portion of the CWord, as the network processor only asserts it for the first 32-bit portion. If the bridge chip requires clock frequencies on the network processor interface and the standard CSIX-L1 interface to be appropriate, exact multiples of each other (2x for 32-bit CWord, 4x for 64-bit CWord, 6x for 96-bit CWord, and 8x for 128-bit CWord), then the bridge chip requires only minimal buffering and does not need to implement any flow control mechanisms.

A slightly more complicated bridge allows incorporating dynamic de-skew training and/or independent clock frequencies for the network processor and standard CSIX-L1 interfaces. The bridge chip must implement a control and data FIFO for each direction and the link-level flow control mechanisms specified in the protocol using CRdy and DRdy. The FIFOs must be large enough to accommodate the response latency of the link-level flow control mechanisms. Idle Cframes and Dead Cycles are not directly passed through this more complicated bridge chip, but are discarded on reception and generated on transmission. The network processor interface of this bridge chip can support the dynamic de-skew training protocol extensions implemented on the network processor because it can send a training sequence to the network processor between CFrames without regard to CFrames arriving over the standard CSIX-L1 interface. (In the simpler bridge design, these CFrames must be forwarded immediately to the network processor.)

8.9.5 Dual Protocol (SPI and CSIX-L1) Support

In many system designs that are less bandwidth-intensive, a single network processor can forward and process data from the framer to the fabric and from the fabric to the framer. A bridge chip must pass data between the network processor and multiple physical devices. The network processor supports multiplexing SPI-4.2 and CSIX-L1 protocol elements over the same transmitter and receiver physical interfaces, differentiated by a protocol signal that is de-asserted for SPI-4.2 protocol elements and asserted for CSIX-L1 protocol elements.

In the dual protocol configuration, the CSIX-L1 configuration of the network processor corresponds to the dual chip, full duplex configuration. The flow control transmitter interface is looped back to the flow control receiver interface, either externally or internally. Only the LVTTL status interface is available for the SPI-4.2 interface.

8.9.5.1 Dual Protocol Receiver Support

When the network processor receiver is configured for dual protocol support, the aggregate receiver buffer is partitioned in three ways: 50% for data CFrames (4 Kbytes), 37.5% for SPI-4.2 bursts (3 Kbytes) and 12.5% for control CFrames (1 Kbyte). The buffer sizes within each partition are independently configurable. Link-level flow control can be independently configured for assertion at thresholds of 25%, 50%, 75%, or 87.5%. For the traffic associated with each partition, an additional 680 bytes of packed FIFO storage is available to accommodate received traffic after assertion of link-level flow control.

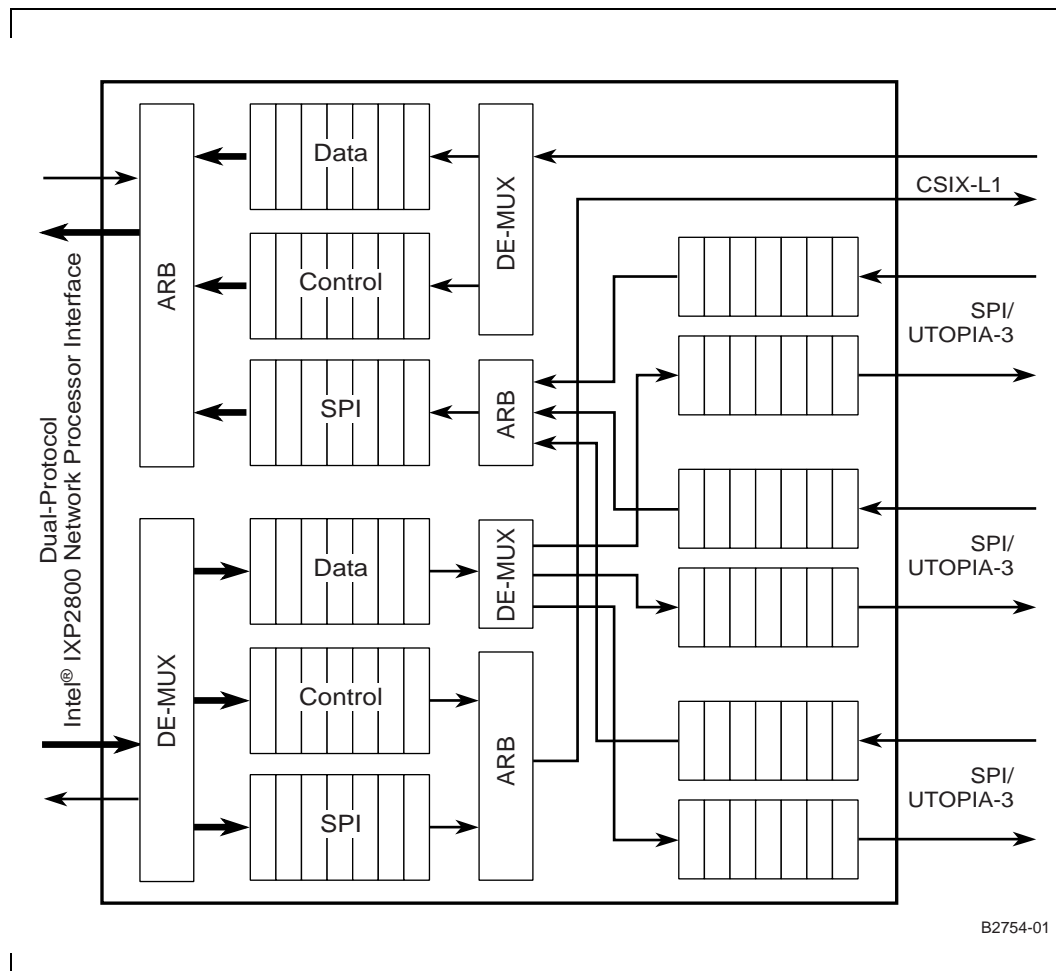
8.9.5.2 Dual Protocol Transmitter Support

When the network processor transmitter is configured for dual protocol support, the aggregate transmitter buffer is partitioned three ways, in the same proportions as the receiver. Each partition operates as a separate ring. The transmitter services each ring in round-robin order. If no CFrames are pending, an Idle CFrame is transmitted to update link-level flow control. If no SPI-4.2 bursts are pending, idle control words are not sent.

8.9.5.3 Implementation of a Bridge Chip to CSIX-L1 and SPI-4.2

A bridge chip can provide support for both standard CSIX-L1 and standard physical layer device interfaces such as SPI-3 or UTOPIA Level 3. The bridge chip must implement the functionality of the less trivial CSIX-L1 bridge chip described previously and additionally, implement bridge functionality between SPI-4.2 and the other physical device interfaces. The size of the FIFOs must be in accordance with the response times of the flow control mechanisms. [Figure 116](#) is a block diagram of a dual protocol (SPI-4.2 and CSIX-L1) bridge chip.

Figure 116. Block Diagram of Dual Protocol (SPI-4.2 and CSIX-L1) Bridge Chip



8.9.6 Transmit State Machine

Table 116 describes the transmitter state machine by providing guidance in interfacing to the network processor. The state machine is described as three separate state machines for SPI-4.2, training, and CSIX-L1. When each machine is inactive, it tracks the states of the other two state machines.



8.9.6.1 SPI-4.2 Transmitter State Machine

The SPI-4.2 Transmit State Machine makes state transitions on each bus transfer of 16 bits, as described in [Table 116](#).

Table 116. SPI-4.2 Transmitter State Machine Transitions on 16-Bit Bus Transfers

Current State	Next State	Conditions
Idle Control	Idle Control	No data pending and no training sequence pending, CSIX-L1 mode disabled.
	Payload Control	Data pending and no training sequence pending, CSIX-L1 mode disabled.
	Training	Training sequence pending, CSIX-L1 mode disabled.
	CSIX	CSIX-L1 mode enabled.
Payload Control	Data Burst	Always
Data Burst	Data Burst	Until end of burst as programmed by software.
	Payload Control	Data pending and no training sequence pending and CSIX-L1 mode not enabled.
	Idle Control	No data to send or training sequence pending or CSIX-L1 mode enabled.
Tracking Other State Machine States		
Training	Training	Training SM not entering CSIX-L1 or SPI state.
	CSIX	Training SM entering CSIX-L1 state.
	Payload Control	Training SM entering SPI state and data pending.
	Idle Control	Training SM entering SPI state and no data pending.
CSIX	CSIX	CSIX-L1 SM not entering Training or SPI state.
	Training	CSIX-L1 SM entering Training state.
	Payload Control	CSIX-L1 SM entering SPI state and data pending.
	Idle Control	CSIX-L1 SM entering SPI state and no data pending.

8.9.6.2 Training Transmitter State Machine

The Training State Machine makes state transitions on each bus transfer of 16 bits, as described in Table 117.

Table 117. Training Transmitter State Machine Transitions on 16-Bit Bus Transfers

Current State	Next State	Conditions
Training Control	Training Control	Until 10 control cycles.
	Training Data	After 10 control cycles.
Training Data	Training Data	Until 10 data cycles.
	Training Control	After 10 data cycles and repetitions of training sequence or new training sequence pending.
	CSIX	After 10 data cycles and no training sequence pending and CSIX-L1 mode enabled.
	SPI	After 10 data cycles and No training sequence pending and CSIX-L1 mode disabled.
Tracking Other State Machine States		
CSIX	CSIX	CSIX-L1 SM not entering SPI or Training state.
	SPI	CSIX-L1 SM entering SPI state.
	Training Control	CSIX-L1 SM entering Training state.
SPI	SPI	SPI SM not entering CSIX-L1 or Training state.
	CSIX	SPI SM entering CSIX-L1 state.
	Training Control	SPI SM entering Training state.

8.9.6.3 CSIX-L1 Transmitter State Machine

The CSIX-L1 Transmit State Machine makes state transitions on CWord boundaries. CWords can be configured to consist of 32, 64, 96, or 128 bits, corresponding to 2, 4, 6, or 8 bus transfers, as described in Table 118.

Table 118. CSIX-L1 Transmitter State Machine Transitions on CWord Boundaries

Current State	Next State	Conditions
SoF CWord	CFrame CWord	CFrame longer than a CWord.
	Dead Cycle	CFrame fits in a CWord.
CFrame CWord	CFrame CWord	CFrame remainder pending.
	SoF CWord	Un-flow-controlled CFrame pending, no training sequence pending, and SPI mode not enabled.
	Dead Cycle	No un-flow-controlled CFrame pending or training sequence pending or requesting training sequence or SPI mode enabled and data pending.
Dead Cycle	SoF CWord	Un-flow-controlled CFrame pending and no training sequence pending and no SPI data pending and not requesting training sequence.
	Idle CFrame	No un-flow-controlled CFrame pending and no training sequence pending and no SPI data pending and not requesting training sequence.

Table 118. CSIX-L1 Transmitter State Machine Transitions on CWord Boundaries (Continued)

Current State	Next State	Conditions
	Dead Cycle	Requesting reception of training sequence and no training sequence pending.
	Training	Training sequence pending.
	SPI	Training sequence not pending and SPI data pending and not requesting training sequence.
Idle CFrame	Dead Cycle	Always.
Tracking Other State Machine States		
SPI	SPI	SPI SM not entering CSIX-L1 or Training state.
	SoF CWord	SPI SM entering CSIX-L1 state and un-flow-controlled CFrame pending.
	Idle CFrame	SPI SM entering CSIX-L1 state and un-flow-controlled CFrame not pending.
	Training	SPI SM entering Training state.
Training	Training	Training SM not entering CSIX-L1 or Training state.
	SoF CWord	Training SM entering CSIX-L1 state and un-flow-controlled CFrame pending.
	Idle CFrame	Training SM entering CSIX-L1 state and un-flow-controlled CFrame not pending.
	SPI	Training SM entering SPI state.

8.9.7 Dynamic De-Skew

The Intel® IXP2800 Network Processor supports optional dynamic de-skew for the signals of the 16-bit data interface and the signals of the 4-bit flow control interface or the signals of the 2-bit SPI-4.2 LVDS status interface. (The flow control interface and the LVDS status interface are alternate configurations of the same signal balls and pads. They share the same de-skew circuits.)

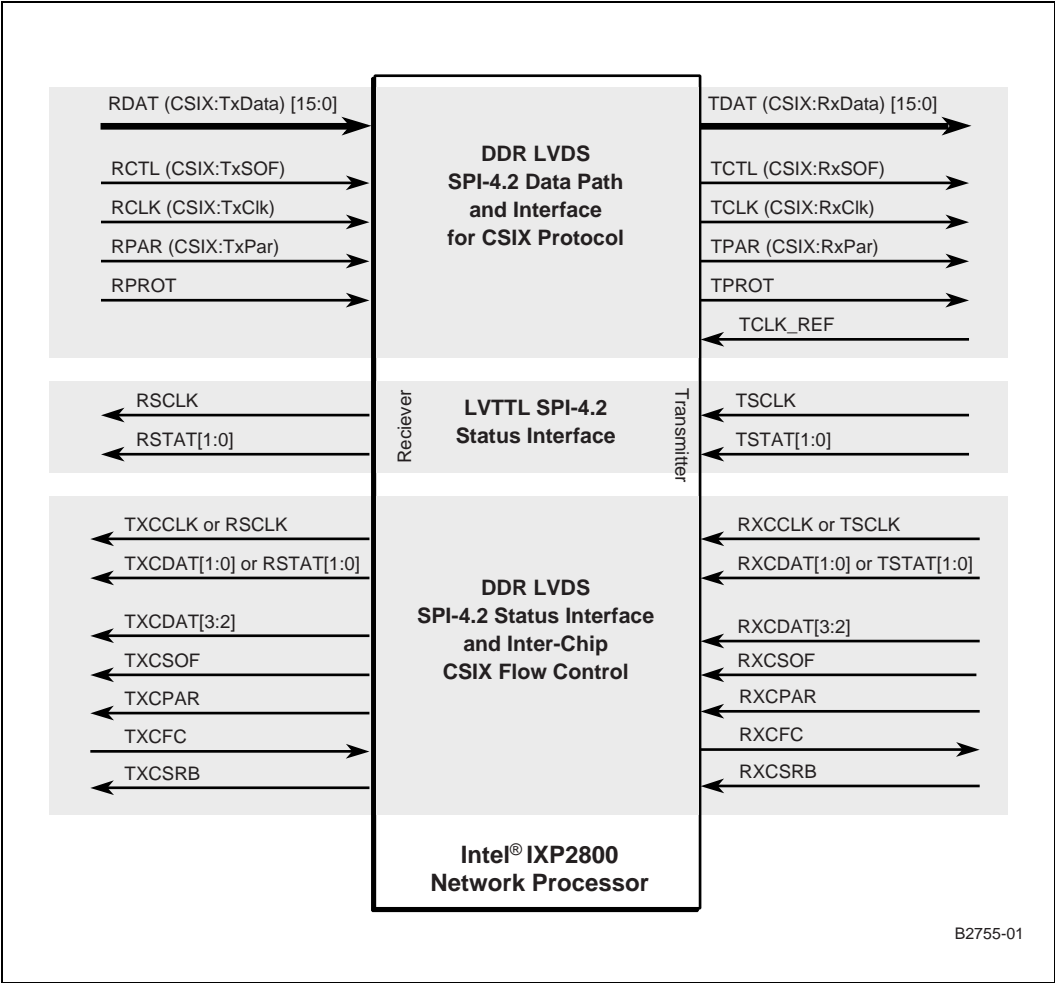
In both cases, eight evenly-spaced phases of the received clock are generated for each bit time. As the transition occurs during training a pattern, the best pair of clock phases is identified for sampling each received signal. An interpolated clock is generated from a pair of clock phases for each signal and that clock is used as a reference for sampling the data. This provides maximum quantization error in the sampling of the signals of 6.25%.



8.9.8 Summary of Receiver and Transmitter Signals

Figure 117 summarizes the Receiver and Transmitter Signals.

Figure 117. Summary of Receiver and Transmitter Signaling



This section contains information on the IXP2800 Network Processor PCI Unit.

9.1 Overview

The PCI Unit allows PCI target transactions to internal registers, SRAM, and DRAM. It also generates PCI initiator transactions from the DMA Engine, Intel XScale® core, and Microengines.

The PCI Unit main functional blocks are shown in [Figure 118](#) and include:

- PCI Core Logic
- PCI Bus Arbiter
- DRAM Interface Logic
- SRAM Interface Logic
- Mailbox and Message registers
- DMA Engine
- Intel XScale® core Direct Access to PCI

The main function of the PCI Unit is to transfer data between the PCI Bus and the internal devices, which are the Intel XScale® core, the internal registers and memories.

These are the data transfer paths supported as shown in [Figure 119](#):

- PCI Slave read and write between PCI and internal buses
 - CSRs (PCI_CSR_BAR)
 - SRAM (PCI_SRAM_BAR)
 - DRAM (PCI_DRAM_BAR)
- Push/Pull Master (Intel XScale® core, Microengine, or PCI) accesses to internal registers within PCI unit
- DMA
 - Descriptor read from SRAM
 - Data transfers between PCI and DRAM
- Push/Pull Master (Intel XScale® core and Microengines) direct read and write to PCI Bus

Note: Detailed information about CSRs is contained in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

Figure 118. PCI Functional Blocks

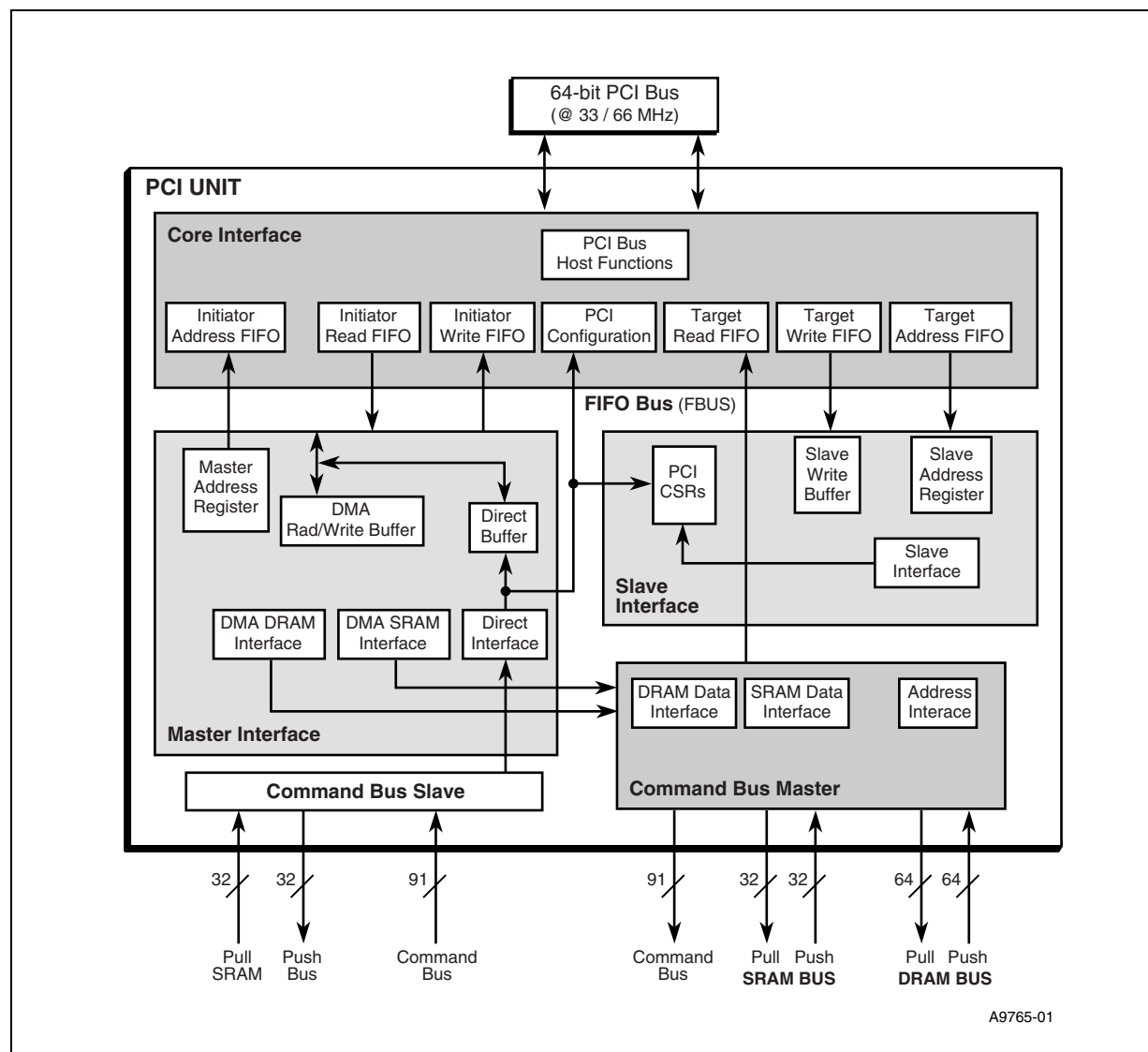
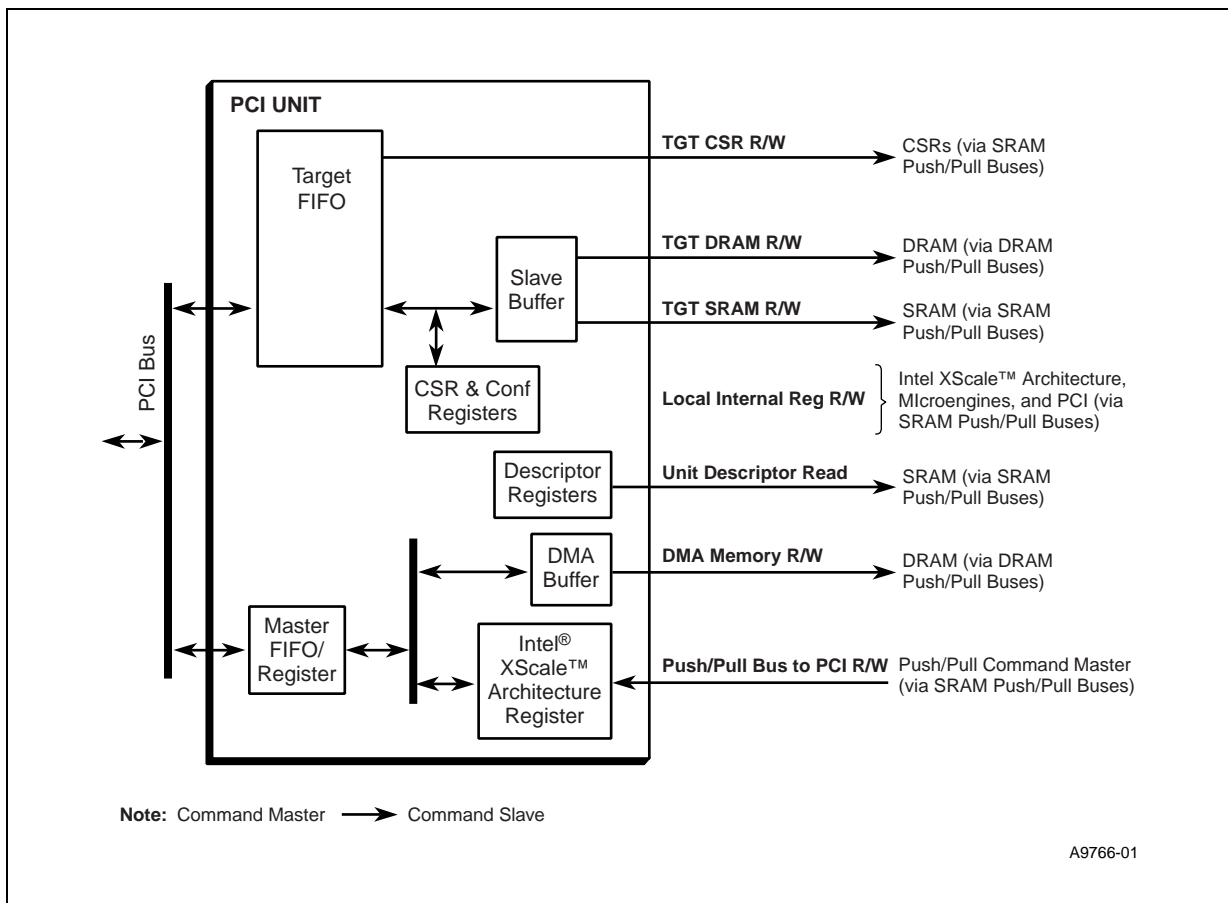




Figure 119. Data Access Paths



9.2 PCI Pin Protocol Interface Block

This block generates the PCI compliant protocol logic. It operates either as an initiator or a target device on the PCI Bus. As an initiator, all bus cycles are generated by the core. As a PCI target, the core responds to bus cycles that have been directed towards it.

On the PCI Bus, the interface supports interrupts, 64-bit data path, 32-bit addressing, and single configuration space. The local configuration registers are accessible from the PCI Bus or from the Intel XScale® core through an internal path.

The PCI block interfaces with the other sub-blocks with a FIFO bus called FBus. The FBus speed is the same as the internal Push/Pull bus speed. The FIFOs are implemented with clock synchronization logic between the PCI speed and the internal Push/Pull bus speed.

There are four data FIFOs and two address FIFOs in the core. The separate slave and master data FIFOs allows simultaneous operations and multiple outstanding PCI bus transfers. Table 119 lists the FIFO sizes. The target address FIFO latches up to four PCI read or write addresses.



If a read address is latched, the subsequent cycles will be retried and no address will be latched until the read completes. The initiator address FIFO can accumulate up to four addresses which can be PCI reads or writes.

These FIFOs are inside the PCI Core which stores data that are received from the PCI Bus or to be sent out to the PCI Bus. There are additional buffers implemented in other sub-blocks that buffers data to and from the internal push/pull buses.

Table 119. PCI Block FIFO Sizes

Location	Depth
Target Address	4
Target Write Data	8
Target Read Data	8
Initiator Address	4
Initiator Write Data	8
Initiator Read Data	8

Table 120 lists the maximum PCI Interface loading.

Table 120. Maximum Loading¹

Bus Interface	Max # of Loads	Trace Length (inches)
PCI	Four loads at 66 MHz bus frequency Eight loads at 33 MHz bus frequency	5 to 7

1. These specifications are currently under evaluation.

9.2.1 PCI Commands

Table 121 lists the supported PCI commands and identifies them as either a target or initiator.

Table 121. PCI Commands (Sheet 1 of 2)

C_BE_L	Command	Support	
		Target	Initiator
0x0	Interrupt Acknowledge	Not Supported	Supported
0x1	Special Cycle	Not Supported	Supported
0x2	IO Read cycle	Not Supported	Supported
0x3	IO Write cycle	Not Supported	Supported
0x4	Reserved	-	-
0x5	Reserved	-	-
0x6	Memory Read	Supported	Supported
0x7	Memory Write	Supported	Supported
0x8	Reserved	-	-
0x9	Reserved	-	-
0xA	Configuration Read	Supported	Supported



Table 121. PCI Commands (Sheet 2 of 2)

C_BE_L	Command	Support	
		Target	Initiator
0xB	Configuration Write	Supported	Supported
0xC	Memory Read Multiple	Aliased as Memory Read except SRAM accesses where the number of Dwords to read is given by the cache line size	Supported
0xD	Reserved		
0xE	Memory read line	Aliased as Memory Read except SRAM accesses where the number of Dwords to read is given by the cache line size	Supported
0xF	Memory Write and Invalidate	Aliased as Memory Write	Not Supported

PCI functions not supported by the PCI Unit include:

- IO Space response as a target
- Cacheable memory
- VGA palette snooping
- PCI Lock Cycle
- Multi-function devices
- Dual Address cycle

9.2.2 IXP2800 Network Processor Initialization

When the IXP2800 Network Processor is a target, the internal CSR, DRAM, or SRAM address is generated when the PCI address matches the appropriate base address register. The window sizes to the SRAM and DRAM Base Address Registers (BARs) can be optionally set by PCI_SWIN and PCI_DWIN strap pins or mask registers depending on the state of the PROM_BOOT signal.

There are two initialization modes supported. They are determined by the PROM_BOOT signal sampled on the de-assertion edge of Chip Reset. If PROM_BOOT is asserted, which indicates that there is a boot prom in the system. The Intel XScale® core will boot from the prom and be able to program the BAR space mask registers. If PROM_BOOT is not asserted, the Intel XScale® core is held in reset and the BAR sizes are determined by strap pins.

9.2.2.1 Initialization by the Intel XScale® Core

The PCI unit is initialized to an inactive, disabled state until the Intel XScale® core has set the Initialize Complete bit in the Control register. This bit is set after the Intel XScale® core has initialized the various PCI base address and mask registers (which should occur within 1 ms of the end of PCI_RESET). The mask registers are used to initialize the PCI base address registers to values other than the default power-up values which includes the base address visible to the PCI host and the prefetchable bit in the base registers (see [Table 122](#)).

Table 122. PCI BAR Programmable Sizes

Base Address Register	Address Space	Sizes
PCI_CSR_BAR	CSR	1Mbyte
PCI_SRAM_BAR	SRAM	0Byte, 128Kbyte, 256Kbyte, 512Kbyte, 1Mbyte, 2Mbyte, 4Mbyte, 8Mbyte, 16Mbyte, 32Mbyte, 64MByte, 128Mbyte, 256Mbyte
PCI_DRAM_BAR	DRAM	0Byte, 1Mbyte, 2Mbyte, 4Mbyte, 8Mbyte, 16Mbyte, 32Mbyte, 64Mbyte, 128Mbyte, 256Mbyte, 512Mbyte, 1Gbyte

When the PCI unit is in the inactive state, it returns retry responses as the target of PCI configuration cycles if the PCI Unit is not configured as the PCI host. In the case of PCI Unit being configured as the PCI host, the PCI bus will be held in reset until the Intel XScale® core completes the PCI Bus configurations and clears the PCI Reset (as described in [Section 9.2.11](#)).

9.2.2.2 Initialization by a PCI Host

In this mode, the PCI Unit is not hosting the PCI Bus regardless of the PCI_CFG[0] signal. The host processor is allowed to configure the internal CSRs while the Intel XScale® core is held in reset. The host processor configures the PCI address space, the memory controllers, and other interfaces. Also, the program code for the Intel XScale® core may be downloaded into local memory.

The host processor then clears the Intel XScale® core reset bit in the PCI Reset register. This de-asserts the internal reset signal to the Intel XScale® core and the core begins its initialization process. The PCI_SWIN and PCI_DWIN strap signals are used to select the window sizes to SRAM BAR and DRAM BAR (see [Table 123](#)).

Table 123. PCI BAR Sizes with PCI host Initialization

Base Address Register	Address Space	Sizes
PCI_CSR_BAR	CSR	1MByte
PCI_SRAM_BAR	SRAM	32M/64MByte/128MByte/256MByte
PCI_DRAM_BAR	DRAM	128MByte/256MByte/512MByte/1GByte

9.2.3 PCI Type 0 Configuration Cycles

A PCI access to a configuration register occurs when the following conditions are satisfied:

- PCI_IDSEL is asserted. (PCI_IDSEL only support PCI_AD[23:16] bits).
- The PCI command is a configuration write or read.
- The PCI_AD [1:0] are 00.

A configuration register is selected by PCI_AD[7:2]. If the PCI master attempts to do a burst longer than one 32-bit Dword, the PCI unit signals a target disconnect. PCI unit does not issue PCI_ACK64 for configuration cycle.



9.2.3.1 Configuration Write

A **write** occurs if the PCI command is a Configuration Write. The PCI byte enables determine which bytes are written. If a nonexistent configuration register is selected within the configuration register address range, the data is discarded and no error action is taken.

9.2.3.2 Configuration Read

A **read** occurs if the PCI command is a Configuration Read. The data from the configuration register selected by PCI_AD[7:2] is returned on PCI_AD[31:0]. If a nonexistent configuration register is selected within the configuration register address range, the data returned are zeros and no error action is taken.

9.2.4 PCI 64-Bit Bus Extension

The PCI Unit is in 64-bit mode when PCI_REQ64# is sampled active on the de-assertion edge of PCI Reset. These are the general rules in assertions of PCI_REQ64# and PCI_ACK64#:

As a target:

1. PCI Unit asserts PCI_ACK64# only in 64-bit mode.
2. PCI Unit asserts PCI_ACK64# only to target cycles that matches the PCI_SRAM_BAR and PCI_DRAM_BAR and a 64-bit transaction is negotiated.
3. PCI Unit does not assert PCI_ACK64# target cycles that matches the PCI_CSR_BAR even a 64-bit transaction is negotiated.

As an initiator:

1. PCI Unit asserts PCI_REQ64# only in 64-bit mode.
2. PCI Unit asserts PCI_REQ64# to negotiate a 64-bit transaction only if the address is double Dword aligned (PCI_AD[2] must be 0 during the address phase).
3. If the target responses to PCI_REQ#64 with PCI_ACK64# de-asserted, PCI Unit will complete the transaction acting as a 32-bit master by not asserting PCI_REQ64# on subsequent cycle.
4. If the target responses to PCI_REQ#64 with PCI_ACK64# de-asserted and PCI_STOP# asserted, PCI Unit will complete the transaction by not asserting PCI_REQ64# on subsequent cycles.



9.2.5 PCI Target Cycles

The following PCI transactions are not supported by the PCI Unit as a target:

- IO read or write
- Type 1 configuration read or write
- Special cycle
- IACK cycle
- PCI Lock cycle
- Multi-function devices
- Dual Address cycle

9.2.5.1 PCI Accesses to CSR

A PCI access to a CSR occurs if the PCI address matches the CSR base address register (PCI_CSR_BAR). The PCI Bus will be disconnected after the first data-phase if the data is more than one data phase. For 64-bit CSR accesses, the PCI Unit will not assert PCI_ACK64# on the PCI bus.

9.2.5.2 PCI Accesses to DRAM

A PCI access to DRAM occurs if the PCI address matches the DRAM base address register (PCI_DRAM_BAR).

9.2.5.3 PCI Accesses to SRAM

A PCI access to SRAM occurs if the PCI address matches the SRAM base address register (PCI_SRAM_BAR). The SRAM is organized as three distinct channel and the address is not contiguous. The PCI_SRAM_BAR programmed window size will be used as the total memory space. The upper two bits of the address will be used as channel number in addressing the particular channel and the remaining address bits will be used as the memory address.

9.2.5.4 Target Write Accesses From PCI Bus

A PCI write occurs if the PCI address matches one of the base address registers and the PCI command is either a Memory Write or Memory Write and Invalidate. The core will store up to four write addresses into the target address FIFO along with the BAR IDs of the transaction. The write data will be stored into the target write FIFO. When either the address FIFO or data FIFO is full, a retry is forced on the PCI Bus in response to write accesses.

The FIFO data is forwarded to an internal slave buffer before being written into SRAM or DRAM. If the FIFO fills during the write, the address is crossing the 64-byte address boundary, or in the case of the command being a burst to the CSR space, the PCI unit signals target disconnect to the PCI master.



9.2.5.5 Target Read Accesses From PCI Bus

A PCI read occurs if the PCI address matches one of the base address registers and the PCI command is either a Memory Read, Memory Read Line, or Memory Read Multiple.

The read is completed as a PCI delayed read. That is, on the first occurrence of the read, the PCI unit signals a retry to the PCI master. If there is no prior read pending, the PCI unit latches the address and command and places it into the target address FIFO. When the address reaches the head of the FIFO, the PCI unit reads the DRAM. Subsequent reads will also get retry responses until data is available.

When the read data is returned into the PCI Read FIFO, the PCI unit begins to decrement its discard timer. If the PCI bus master has not repeated the read by the time the timer reaches zero, the PCI unit discards the read data, invalidates the delayed read address and sets Discard Timer Expired (bit 16) in the Control register (PCI_CONTROL). If enabled, the PCI unit interrupts the Intel XScale® core. The discard timer counts 2^{15} (32768) PCI clocks.

When the master repeats the read command, the PCI unit compares the address and checks that the command is a Memory Read, a Memory Read Line, or a Memory Read Multiple. If there is a match, the response is as follows:

- If the read data has not yet been read, the response is retry.
- If the read data has been read, assert `trdy_1` and deliver the data. If the master attempts to continue the burst past the amount of data read, the PCI unit signals a target disconnect.
- CSR reads are always 32-bit reads.
- If the discard timer has expired for a read, the subsequent read will be treated as a new read.

9.2.6 PCI Initiator Transactions

PCI master transactions are caused by either the Intel XScale® core loads and stores that fall into the various PCI address spaces, Microengine read and write commands, or by DMA engine. The command register (PCI_COMMAND) bus master bit (BUS_MASTER) must be set for the PCI unit to perform any of the initiator transactions.

The PCI cycle is initiated when there is an entry in the PCI Core Interface initiator address FIFO. The core handshakes with the master interface with the FBus FIFO status signals. The PCI core supports both burst and non-burst master read transfers by the burst count inputs (`FB_BstCnt[7:0]`), driven by Master Interface to inform the core the burst size. For a Master write, `FB_WBstonN` indicates to the PCI core whether the transfers are burst or non-burst, on a 64-bit double Dword basis.

The PCI core supports read and write memory cycles as an initiator while taking care of all disconnect/retry situations on the PCI Bus.

9.2.6.1 PCI Request Operation

If an external arbiter is used (PCI_CFG_ARB[1] is not active), the req_l[0] and gnt_l[0] are connected to the PCI_REQ# and PCI_GNT# pins. Otherwise, they are connected to the internal arbiter.

The PCI unit asserts req_l[0] to act as a bus master on the PCI. If gnt_l[0] is asserted, the PCI unit can start a PCI transaction regardless of the state of req_l[0]. When the PCI unit requests the PCI bus, it performs a PCI transaction when gnt_l[0] is received. Once req_l[0] is asserted, the PCI unit never de-asserts it prior to receiving gnt_l[0] or de-asserts it after receiving gnt_l[0] without doing a transaction. PCI Unit de-asserts req_l[0] for two cycles when it receives a retry or disconnect response from the target. However,

9.2.6.2 PCI Commands

The following PCI transactions are not generated by PCI Unit as an initiator:

- PCI Lock Cycle
- Dual Address cycle
- Memory Write and Invalidate

9.2.6.3 Initiator Write Transactions

The following general rules apply to the write command transactions:

- If the PCI unit receives either a target retry response or a target disconnect response before all of the write data has been delivered, it resumes the transaction at the first opportunity, using the address of the first undeliverable data.
- If the PCI unit receives a master abort, it discards all of the write data from that transaction and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the Intel XScale® core.
- If the PCI unit receives a target abort, it discards all of the remaining write data from that transaction, if any, and sets the status registers (PCI_STATUS) received target abort bit, which, if enabled, interrupts the Intel XScale® core.
- The PCI unit can dessert frame_l prior to delivering all data due to the master latency timer, If this occurs, it resumes the write at the first opportunity, using the address of the first undeliverable data.

9.2.6.4 Initiator Read Transactions

The following general rules apply to the read command transactions:

- If the PCI unit receives a target retry, it repeats the transaction at the first opportunity until the whole transaction is completed.
- If the PCI unit receives a master abort, it substitutes 0xFFFF FFFF for the read data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the Intel XScale® core.
- If the PCI unit receives a target abort, it sets the status registers (PCI_STATUS) received target abort bit, which, if enabled, interrupts the Intel XScale® core and does not try to get any more read data. PCI unit will substitute 0xFFFF FFFF for the data which are not read and complete the cycle.



9.2.6.5 Initiator Latency Timer

When the PCI unit begins PCI transaction as an initiator, asserting frame_1, it begins to decrement its master latency timer. When the timer value reaches zero, the PCI unit checks the value of gnt_1[0]. If gnt_1[0] is de-asserted, the PCI unit de-asserts frame_1 (if it is still asserted) at the earliest opportunity. This is normally the next data phase for all transactions.

9.2.6.6 Special Cycle

As an initiator, special cycles are broadcast to all PCI agents, so DEVSEL# is not asserted and no error can be received.

9.2.7 PCI Fast Back to Back Cycles

The core supports fast back-to-back target cycles on the PCI Bus. The core does not generate initiator fast back-to-back cycles on the PCI Bus regardless of the value in the fast back to back enable bit of the Status and Command register in the PCI configuration space.

9.2.8 PCI Retry

As a slave, the PCI Unit generates retry on:

- A slave write when the Data write FIFO is full.
- When address FIFO is full
- Data read is handled as delay transactions. If the HOG_MODE bit is set in the PCI_CONTROL register, the bus will be held for 16 PCI clocks before asserting retry.

As an initiator, the core supports retry by maintaining an internal counter of the current address. On receiving a retry, the core de-asserts PciFrameN and then re-assert PciFrameN with the current address from the counter.

9.2.9 PCI Disconnect

As a slave, it disconnects for the following conditions:

- Bursted PCI configuration cycle.
- Bursted access to PCI_CSR_BAR.
- PCI reads past the amount of data in the read FIFO.
- PCI burst cycles that cross 1K PCI address boundary which includes PCI burst cycles that cross memory decodes from the core as a target to decodes that are outside the core (e.g., started inside a BAR and ends outside of that BAR).

As an initiator, the core supports retry and disconnect by maintaining an internal counter of the current address. On receiving a retry or disconnect, the core de-asserts PciFrameN and then re-assert PciFrameN with the current address + “current transfer byte size” from the counter.

9.2.10 PCI Built In System Test

The IXP2800 Network Processor supports BIST when there is an external PCI host. The PCI host will set the STRT bit in the PCI_CACHE_LAT_HDR_BIST configuration register. An interrupt is generated to the Intel XScale® core if it is enabled by the Intel XScale® core Interrupt Enable register. The Intel XScale® software can respond to the interrupt by running an application specific test. Upon successful completion of the test, the Intel XScale® core will reset the STRT bit. If this bit is not reset 2 seconds after the PCI host sets the STRT bit, the host will indicate that the IXP failed the test.

9.2.11 PCI Central Functions

The CFG_RSTDIR pin is active high for enabling the PCI Unit central function.

The CFG_PCI_ARB(GPIO[2]) pin is the strap pin for the internal arbiter. When this strap pin is high during reset then the XPI Unit owns the arbitration.

The CFG_PCI_BOOT_HOST(GPIO[1]) pin is the strap pin for the PCI host. When PCI_BOOT_HOST is asserted during reset then PCI Unit will support as a PCI host.

Table 124. Legal Combinations of the Strap Pin Options

	CFG_PCI_BOOT_HOST (GPIO[1])	CFG_PCI_Arbiter (GPIO[2])	CFG_PCI_RSTDIR (Central function)	CFG_PROM_BOOT (GPIO[0])
OK	0	0	0	0
OK	0	0	0	1
OK	0	0	1	1
Not supported	0	1	0	x
OK	0	1	1	1
Not supported	1	0	0	x
OK	1	0	1	1
Not supported	1	1	0	x
OK	1	1	1	1

Note

- * CFG_PCI_RSTDIR = 1 then central function.
- * PCI_Host must be central function.
- * PCI_Arbiter must be central function.

9.2.11.1 PCI Interrupt Inputs

The PCI Unit supports two interrupt lines from the PCI Bus as host. One of the interrupt lines will be open-drain output and input. The other interrupt line will be selected as PCI interrupt input. Both the interrupt lines can be enabled in the Intel XScale® core Interrupt Enable register.



9.2.11.2 PCI Reset Output

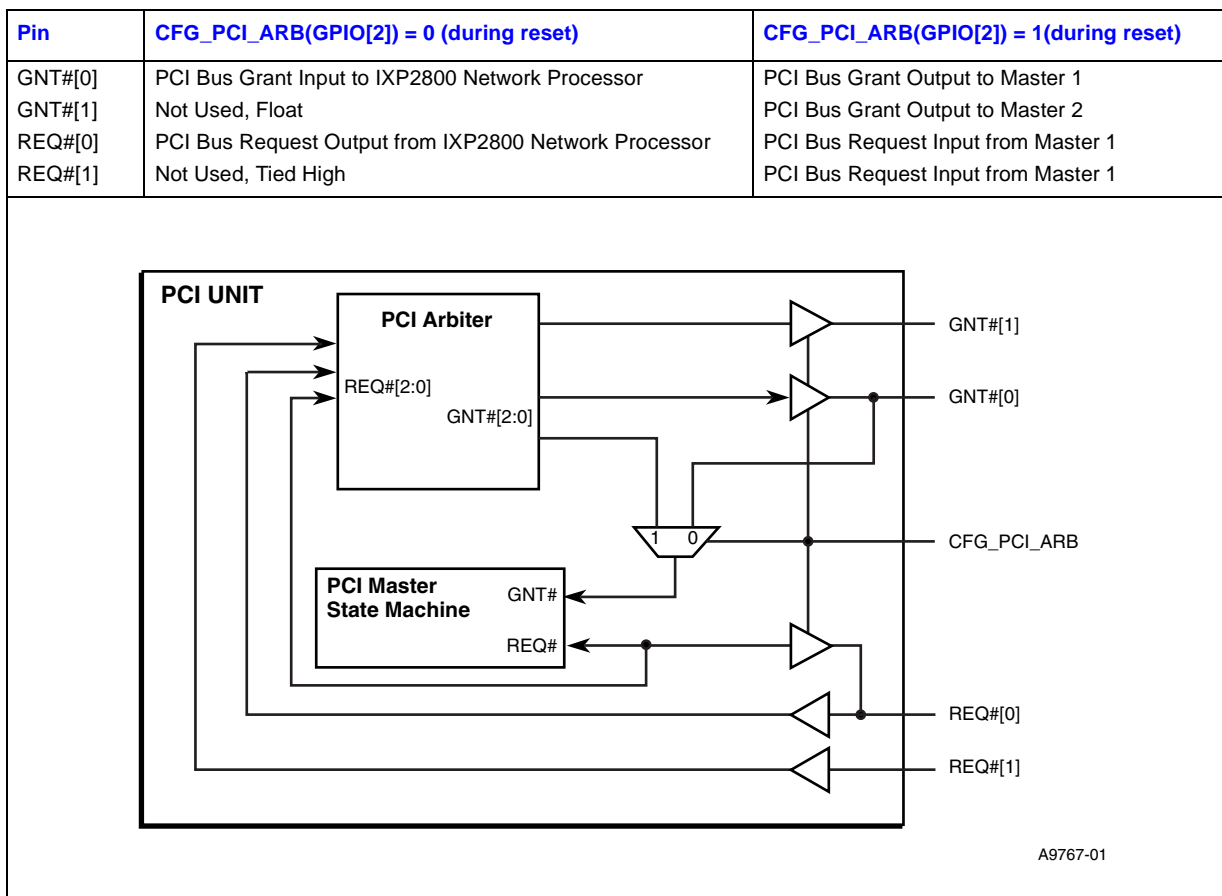
If the IXP2800 Network Processor is central function (CFG_RSTDIR =1), PCI Unit will be asserting the PCI_RST# after the system power-on. The Intel XScale® core has to write to the PCI External Reset bit in the IXP Reset register to de-assert the PCI_RST#. In this case, chip reset (SYS_RESET_L) is driven by a signal other than PCI_RST#.

When the PCI Unit is not configured as the central function (CFG_RSTDIR =0), PCI_RST# is used as a chip reset input.

9.2.11.3 PCI Internal Arbiter

The PCI unit contains a PCI bus arbiter that supports two external masters in addition to the PCI Unit's initiator interface. To enable the PCI arbiter, the CFG_PCI_ARB(GPIO[2]) strapping pin must be 1 during reset. As shown in Figure 120, the local bus request and grant pair become externally not visible. These signals will be made available to external debug pins for debug purpose.

Figure 120. PCI Arbiter Configuration Using CFG_PCI_ARB(GPIO[2])



The arbiter uses a simple round-robin priority algorithm. The arbiter asserts the grant signal corresponding to the next request in the round-robin during the current executing transaction on the PCI bus (this is also called hidden arbitration). If the arbiter detects that an initiator has failed to



assert frame_1 after 16 cycles of both grant assertion and PCI bus idle condition, the arbiter de-asserts the grant. That master does not receive any more grants until it de-asserts its request for at least one PCI clock cycle. Bus parking is implemented in that the last bus grant will stay asserted if no request is pending.

To prevent bus contention, if the PCI bus is idle, the arbiter never asserts one grant signal in the same PCI cycle in which it de-asserts another. It de-asserts one grant, and then asserts the next grant after one full PCI clock cycle has elapsed to provide for bus driver turnaround.

9.3 Slave Interface Block

The slave interface logic supports internal slave devices interfacing to the target port of the FBus.

- CSR—register access cycles to local CSRs.
- DRAM—memory access cycles to the DRAM push/pull Bus.
- SRAM—memory access cycles to the SRAM push/pull Bus.

The slave port of the FBus is connected to a 64-byte write buffer to support bursts of up to 64 bytes to the memory interfaces. The slave read data are directly downloaded into the FBus read FIFO. See [Table 125](#).

Table 125. Slave Interface Buffer Sizes

Location	Slave Address	Slave Write	Slave Read
Buffer Depth	1	64Byte	0
Usage	CSR, SRAM, DRAM	SRAM, DRAM	NONE

As a push/pull command bus master, the PCI Unit translates these accesses into different types of push/pull command. As the push/pull data bus target, the write data is sent through the pull data bus and the read data is received on the push data bus.

9.3.1 CSR Interface

The internal Control and Status registers data is directed to or from the Slave FIFO port of the PCI core FBus when the BAR id matches PCI_CSR_BAR (BAR0). The CSR accesses from the PCI Bus directed towards CSRs not in PCI Unit is translated into a push/pull CSR type command. PCI local CSRs are handled within the PCI Unit.

For writes, the data is sent when the pull bus is valid and the ID matches. The address is unloaded from the FBus target address FIFO as indication to the PCI core logic that the cycle is completed. The slave write buffer is not used for CSR access.

For reads, the data is loaded into the target receive FIFO as soon as the push bus is valid and the ID matches. The address is unloaded from the FBus address FIFO.

One example of a PCI host access to internal registers is the initialization of internal registers and memory to enable the Intel XScale® core to boot off the DRAM in the absence of a boot up PROM.

The accesses to the CSRs inside the PCI Unit are completed internally without sending the transaction out to the push pull bus, just like the other internal register accesses.



9.3.2 SRAM Interface

The SRAM interface connects the FBus to the internal push/pull command bus and the SRAM push/pull data buses. Request to memory is sent on the command bus. Data request is received as valid push/pull ID sent by the SRAM push/pull data bus.

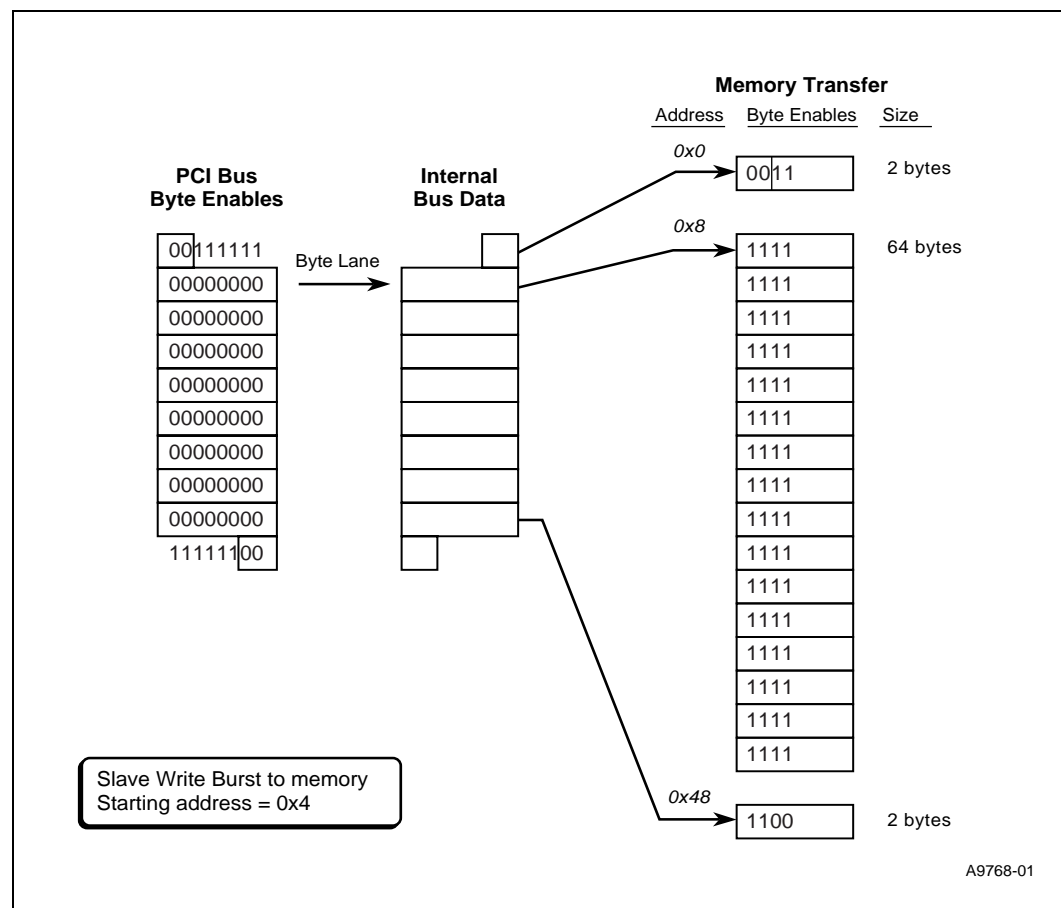
If the PCI_SRAM_BAR is used, the target state machine generates a request to the command bus for SRAM access. Once the grant is received, the address, then data is directed between the slave FIFOs of the PCI core and the SRAM push/pull bus.

9.3.2.1 SRAM Slave Writes

The slave write buffer is used to support memory burst accesses. The buffer is added to guarantee data transfer for each clock and burst size can be determined before memory request is issued. Data is assembled in the buffers before being sent to memory for SRAM write.

On the push/pull bus, AM access can start at any address and have length up to 16 Dwords as shown in Figure 121. For masked writes, only size 1 is supported to transfer up to four bytes.

Figure 121. Example of Target Write to SRAM of 68 Bytes



The slave interface also has to make sure there is enough data in the slave write buffer to complete the memory data transfer before making a memory request.



9.3.2.2 SRAM Slave Reads

For a slave read from SRAM, a 32-bit DWORD is fetched from the memory for memory read command, one cache line is fetched for memory read line command, and two cache lines are read for memory read multiple command. Cache line size is programmable in the CACHE_LINE field of the PCI_CACHE_LAT_HDR_BIST configuration register. If the computed read size is greater than 64 bytes, the PCI SRAM read will default to the maximum of 64 bytes. No pre-fetch is supported in that the PCI Unit will not read beyond the computed read size.

The PCI core resets the target read FIFO before issuing a memory read data request on FBus. The maximum size of SRAM data read is 64 bytes. The PCI core will disconnect at the 64-byte address boundary.

9.3.3 DRAM Interface

The memory is accessed using the push/pull mechanism. Request to memory is sent on the command bus. If the PCI_DRAM_BAR is used, the target state machine generates a request to the command bus for DRAM access with the address in the slave address FIFO. Once the push/pull request is received. The data is directed between the Slave FIFOs of the PCI core and DRAM push/pull bus.

9.3.3.1 DRAM Slave Writes

The slave write buffer is used to support memory burst accesses. The buffer is added to guarantee data transfer for each clock and burst size can be determined before memory request is issued. Data is assembled in the buffers before being sent to memory for memory write.

DRAM target write access is only required to be 8-byte address aligned and the address does not wrap around the 64-byte address boundary on a DRAM burst. Each 8-byte access which is a partial write to the memory is treated as single write. Remaining writes of the 64-byte segment is written as one single burst. Transfers which cross a 64 -byte segment are split in to separate transfers.

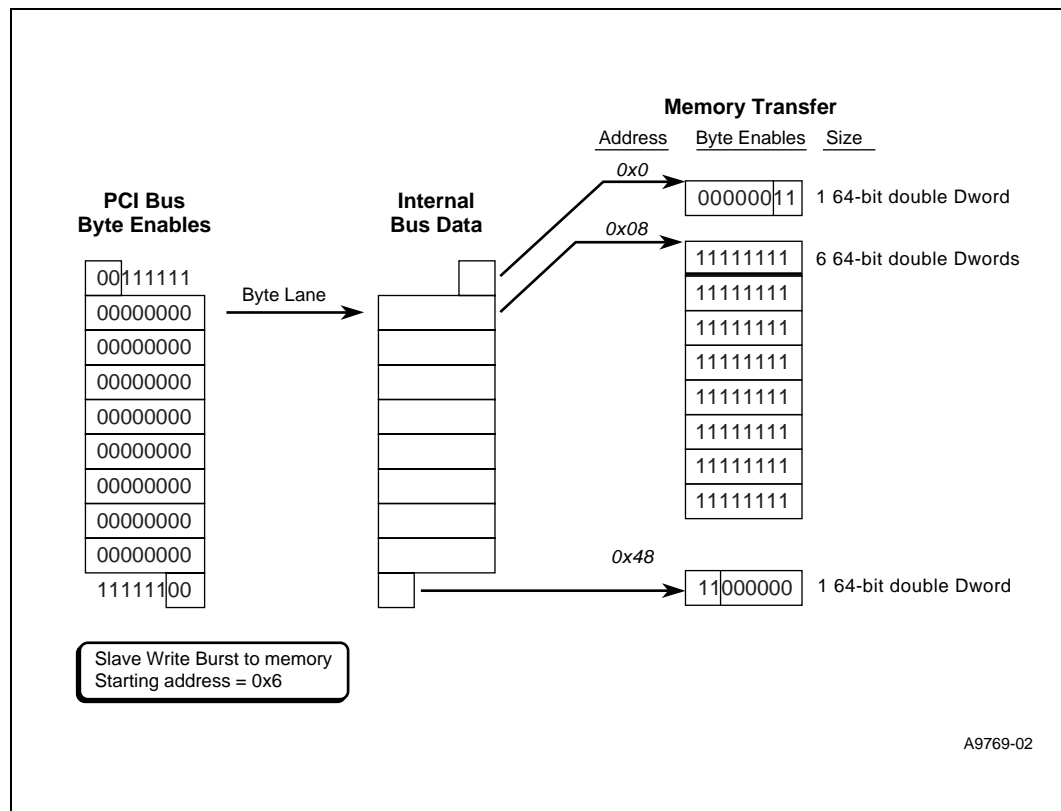
[Figure 123](#) splits the 68 bytes transfers in to two partial 8-byte transfer to address 06 and address 48 and one 56 byte burst transfer in the first 64-byte segment from address 08 to 38 and one 8-byte transfer to address 40.

For write to DRAM on the push/pull bus, the burst must be broken down into address aligned smaller transfer sizes (see [Figure 122](#)).

The Target interface also must make sure there is enough data in the target write buffer to complete the memory data transfer before making a memory request.



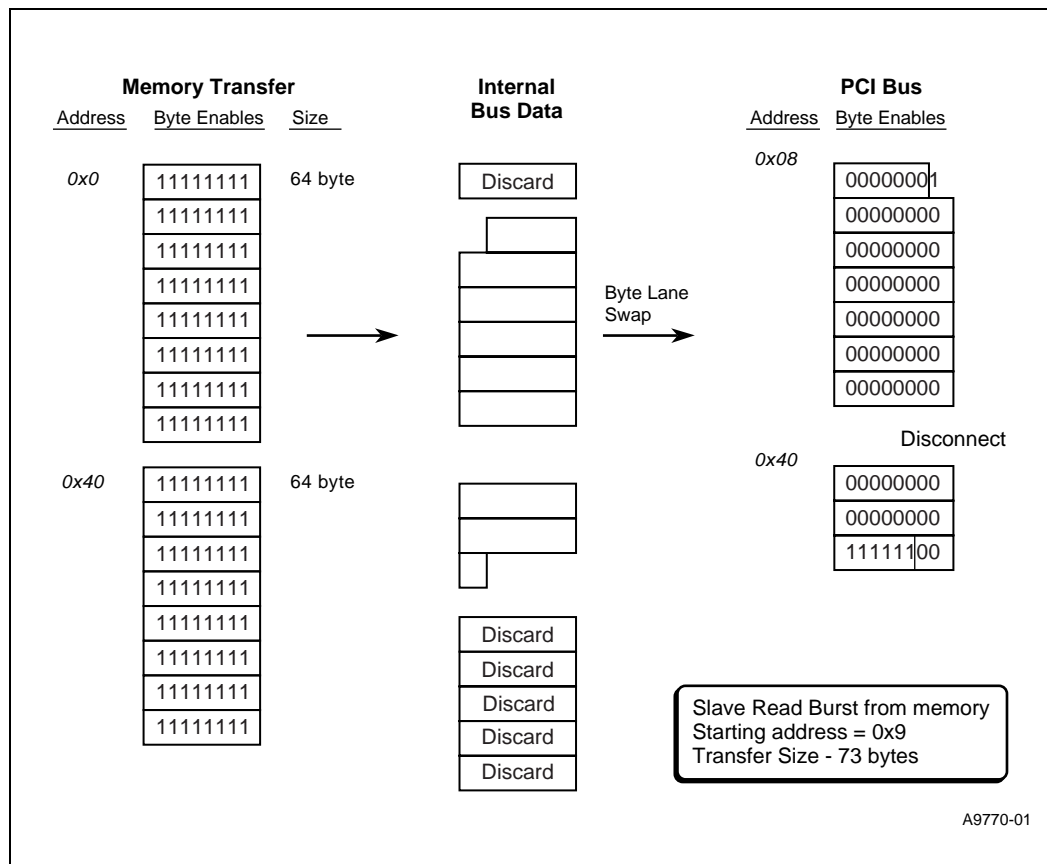
Figure 122. Example of Target Write to DRAM of 68 Bytes



9.3.3.2 DRAM Slave Reads

For target reads from IXP2400 Network Processor memory, the entire 64-byte block is fetched from DRAM. For target reads from IXP2800 Network Processor memory, the block size is 16 bytes. Depending on the address for the target request, extra data is discarded at the beginning until the target address is reached. Also, extra data is discarded at the end of the transfer also when the burst ends in the middle of a data block. No pre-fetch is supported for DRAM access. See Figure 123.

Figure 123. Example of Target Read from DRAM Using 64-Byte Burst



The PCI core resets the read FIFO before issuing a memory read data request on FBus. The PCI core will disconnect at the 64-byte address boundary.



9.3.4 Mailbox and Doorbell Registers

Mailbox and Doorbell registers provide hardware support for communication between the Intel XScale® core and a device on the PCI Bus.

Four mailbox registers are provided so that messages can be passed between the Intel XScale® core and a PCI device. All four registers are 32 bits and can be read and written with byte resolution from both the Intel XScale® core and PCI. How the registers are used is application dependent and the messages are not used internally by the PCI Unit in any way. The mailbox registers are often used with the Doorbell interrupts.

Doorbell interrupts provide an efficient method of generating an interrupt as well as encoding the purpose of the interrupt. The PCI Unit supports an Intel XScale® core Doorbell register that is used by a PCI device to generate an Intel XScale® core FIQ and a separate PCI Doorbell register that is used by the Intel XScale® core to generate a PCI interrupt. A source generating the Doorbell interrupt can write a software defined bitmap to the register to indicate a specific purpose. This bitmap is translated into a single interrupt signal to the destination (either a PCI interrupt or a IXP2800 Network Processor interrupt). When an interrupt is received, the Doorbell registers can be read and the bit mask can be interpreted. If a larger bit mask is required than that is provided by the Doorbell register, the Mailbox registers can be used to pass up to four 32-bit blocks of data.

The doorbell interrupts are controlled through the registers shown in [Table 126](#).

Table 126. Doorbell Interrupt Registers

Register Name	Description
Intel XScale® core Doorbell	Used to generate the Intel XScale® core Doorbell interrupts.
Intel XScale® core Doorbell Setup	Used to initialize the Intel XScale® core Doorbell register and for diagnostics.
PCI Doorbell	Used to generate the PCI Doorbell interrupts.
PCI Doorbell Setup	Used to initialize the PCI Doorbell register and for diagnostics.

The Intel XScale® core and PCI devices write to the corresponding DOORBELL register to generate up to 32 doorbell interrupts. Each bit in the DOORBELL register is implemented as an SR flip-flop. The Intel XScale® core writes a 1 to set the flip-flop and the PCI device writes a 1 to clear the flip-flop. Writing a 0 has no effect on the registers. The PCI interrupt signal is the output of an NOR functions of all the PCI DOORBELL register bits (outputs of the SR flip-flops). The Intel XScale® core interrupt signal is the output of an NAND function of all the Intel XScale® core DOORBELL register bits (outputs of the SR flip-flops).

To assert an interrupt (i.e., to “push a doorbell”):

- A write of 1 to the corresponding bit of the DOORBELL register generates an interrupt. This is the case for either PCI device or the Intel XScale® core, since writing 1 changes the doorbell bit to the proper asserted state (i.e., 0 for an Intel XScale® core interrupt and 1 for a PCI interrupt).

To dismiss an interrupt:

- A write of 1 to the corresponding bit of the DOORBELL register clears an interrupt. This is the case for either PCI device or the Intel XScale® core, since writing 1 changes the doorbell bit to the proper de-asserted state (i.e., 1 for an Intel XScale® core interrupt and 0 for a PCI interrupt).

Figure 124 and Figure 125 illustrates how a Doorbell interrupt is asserted and cleared by both the Intel XScale® core and a PCI device.

Figure 124. Generation of the Doorbell Interrupts to PCI

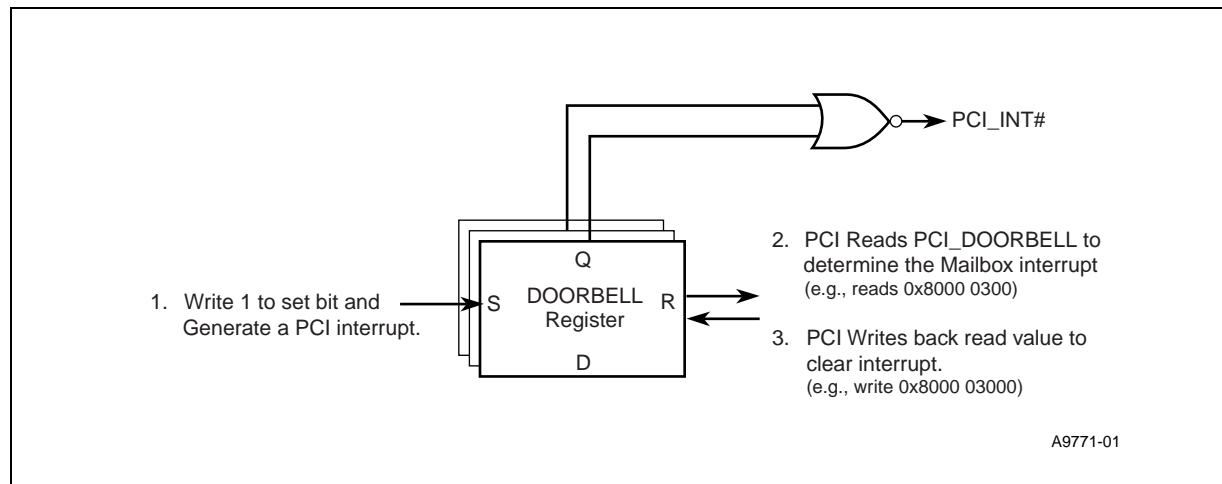
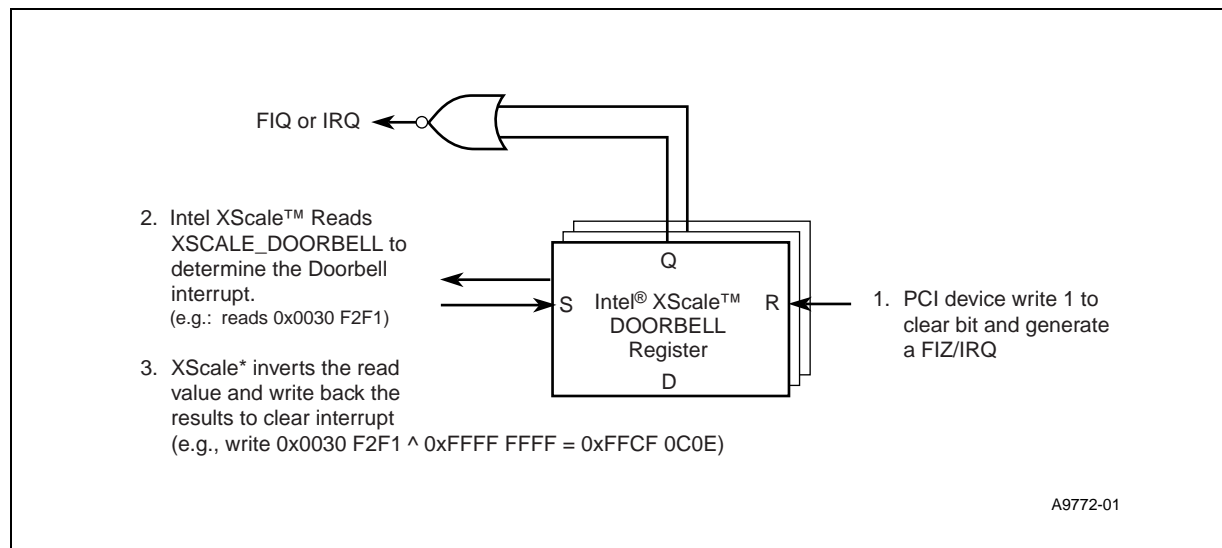


Figure 125. Generation of the Doorbell Interrupts to the Intel XScale® Core



The Doorbell Setup register allows the Intel XScale® core and a PCI device to perform two functions that are not possible using the Doorbell register. This register is used during setup and diagnostics and is not used during normal operations. First, it allows the Intel XScale® core and PCI device to clear an interrupt that it has generated to the other device. If the Intel XScale® core sets an interrupt to PCI device using the Doorbell register, the PCI device is the only one that can use the Doorbell register to clear the interrupt by writing one. With the Doorbell setup register, the Intel XScale® core can clear the interrupt by write 0 to it.

Second, it allows the Intel XScale® core and PCI device to generate a doorbell interrupt to itself. This can be used for diagnostic testing. Each bit in the Doorbell Setup register is mapped directly to the data input of the Doorbell register such that the data is directly written into the Doorbell register.



During system initialization, the doorbell registers must be initialized by clearing the interrupt bits in the Doorbell register using the Doorbell Setup register by writing zeros to the PCI Doorbell setup register and ones to the Intel XScale® core Doorbell setup register.

9.3.5 PCI Interrupt Pin

An external PCI interrupt can be generated in the following way:

- The Intel XScale® core initiates a Doorbell interrupt XSCALE_INT_ENABLE.
- One or more of the DMA channels have completed the DMA transfers.
- The PNI bit is cleared by the Intel XScale® core to generate a PCI interrupt
- An internal functional unit generates either an interrupt or an error directly to the PCI host.

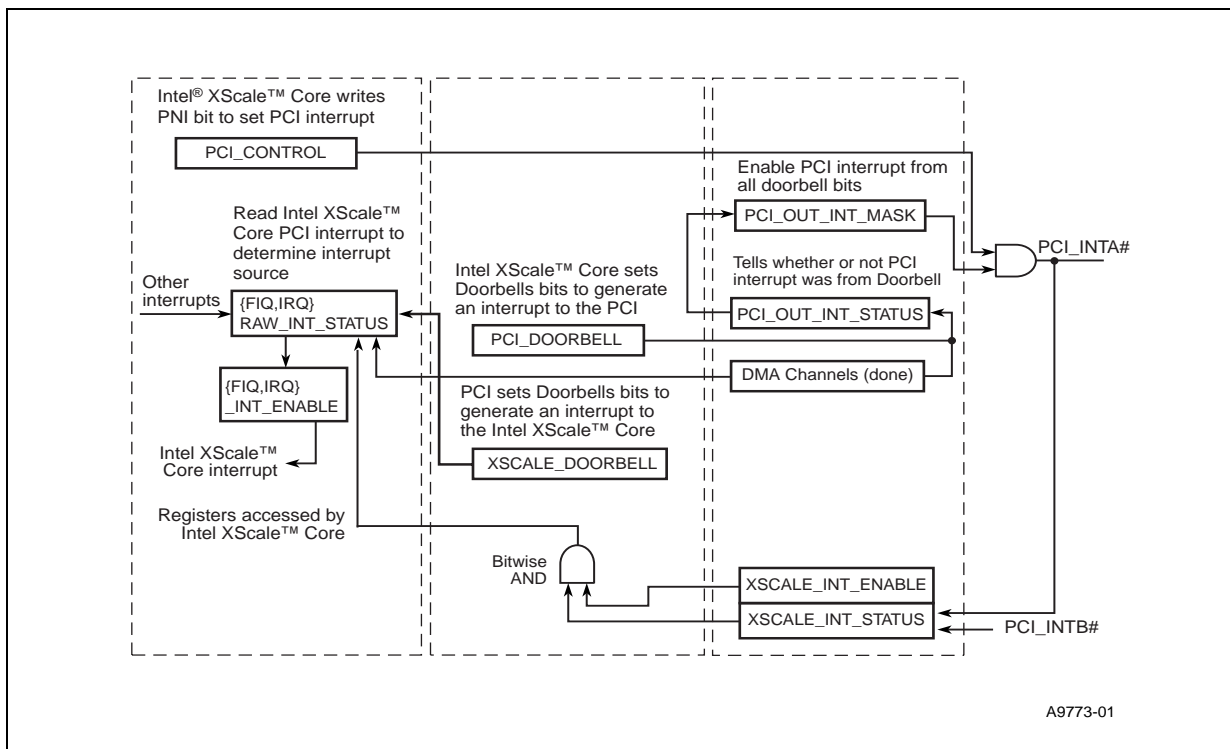
Table 127 describes how IRQ are generated for each silicon stepping.

Table 127. IRQ Interrupt Options by Stepping

Stepping	Description
A stepping	IRQ interrupts can be handled only by the Intel XScale® core.
B Stepping	IRQ interrupts can be handled by either the Intel XScale® core or a PCI host. Refer to the description of the PCI_OUT_INT_MASK and PCI_OUT_INT_STATUS registers in the in the <i>Intel® IXP2400/ IXP2800 Network Processor Programmer's Reference Manual</i> .

Figure 126 shows how PCI interrupts are managed via the PCI and the Intel XScale® core

Figure 126. PCI Interrupts





9.4 Master Interface Block

The Master Interface consists of the DMA engine and the Push/pull target interface. Both can generate initiator PCI transactions:

9.4.1 DMA Interface

There are two DMA channels, each of which can move blocks of data from DRAM to the PCI or from the PCI to DRAM. The DMA channels read parameters from a list of descriptors in SRAM, perform the data movement to or from DRAM, and stop when the list is exhausted. The descriptors are loaded from predefined SRAM entries or may be set directly by CSR writes to DMA registers. There is no restriction on byte alignment of the source address or the destination address. For PCI to DRAM transfers, the PCI command is Memory Read, Memory Read line, or Memory Read Multiple. For DRAM to PCI transfers, the PCI command is Memory Write. Memory Write Invalidate is not supported.

DMA reads are unmasked reads (all byte enables asserted) from DRAM. After each transfer, the byte count is decremented by the number of bytes read, and the source address is incremental by one 64-bit double Dword. The whole data block is fetched from the DRAM. For a system using RDRAM (like the IXP2800 Network Processor), the block size is 16 bytes.

DMA reads are masked reads from the PCI and writes are masked for both the PCI and DRAM. When moving a block of data, the internal hardware adjusts the byte enables so that the data is aligned properly on block boundaries and that only the correct bytes are transferred if the initial and final data requires masking.

For DMA data, the DMA FIFO consists of two separate FBus initiator read FIFOs and two initiator write FIFOs, which are inside the PCI Core and three DMA buffers (corresponding to the DMA channels), which is for buffering data to and from the DRAM. Since there is no simultaneous DMA read and write outstanding, one shared 64-byte buffer is used for both read and write DRAM data

Up to two DMA channels are running at a time with three descriptors outstanding. The two DMA channels and the direct access channel to PCI Bus from Command Bus Master are contending to use the address, read and write FIFOs inside the Core.

Effectively, the active channels interleave bursts to or from the PCI Bus. Each channel is required to arbitrate for the PCI FIFOs after each PCI burst request.

9.4.1.1 Allocation of the DMA Channels

Static allocation are employed such that the DMA resources are controlled exclusively by a single device for each channel. The Intel XScale® core, a Microengine and the external PCI host can access the two DMA channels. The first two channels can function in one of the following modes, as determined by the DMA_INF_MODE register:

- The Intel XScale® core owns both DMA channel 1 and channel 2.
- The Microengines owns both DMA channel 1 and channel 2.
- PCI host owns both DMA channel 1 and channel 2.
- The Intel XScale® core owns both DMA channel 1 and channel 2.

The third channel can be allocated to either the Intel XScale® core, PCI host, or Microengines.



The DMA mode can be changed only by the Intel XScale® core under software control. The software should signal to suspend DMA transactions and wait until all DMA channels are free before changing the mode. Software should determine when all DMA channels are free either by polling XSCALE_INT_STATUS register bits DMA1 and DMA3 until both DMA channels are done.

9.4.1.2 Special Registers for Microengine Channels

Interrupts are generated at the end of DMA operation for the Intel XScale® core and PCI initiated DMA. However, the Microengine does not provide the interrupt mechanism. The PCI Unit will instead use an “Auto-Push” mechanism to signal the particular Microengine on completion of DMA.

When the Microengine sets up the DMA channel, it would also write the CHAN_X_ME_PARAM with Microengine number, Context number, Register number, and Signal number. When the DMA channel completes, it writes some status information (Error or OK status) to the Microengine/Context/Register/Signal. PCI Unit will arbitrate for the SRAM Push bus. The Push ID is from the parameters in the register.

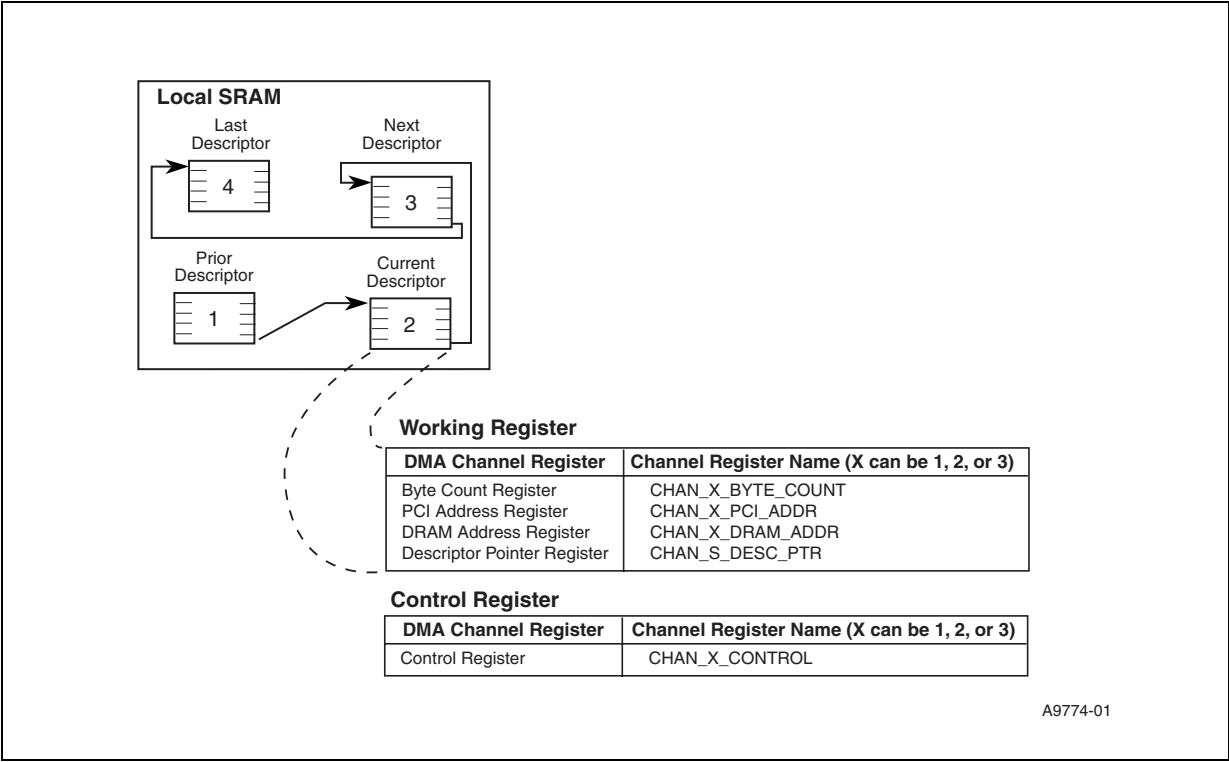
The ME_PUSH_STATUS reflects the DMA Done bit in each of the CHAN_X_CONTROL registers. The Auto-Push operation will proceed after the DMA is done for the particular DMA channel if the corresponding enable bit in the ME_PUSH_ENABLE is set.

9.4.1.3 DMA Descriptor

Each descriptor occupies four 32 bit Dwords and is aligned on a 16-byte boundary. The DMA channels read the descriptors from local SRAM into the four DMA working registers once the control register has been set to initiate the transaction. This control must be set explicitly. This starts the DMA transfer. The register names for the DMA channels are listed in [Figure 127](#).



Figure 127. DMA Descriptor Reads



After a descriptor is processed, the next descriptor is loaded in the working registers. This process repeats until the chain of descriptors is terminated (i.e., the End of Chain bit is set). See [Table 128](#).

Table 128. DMA Descriptor Format

Offset from Descriptor Pointer	Description
0x0	Byte Count
0x4	PCI Address
0x8	DRAM Address
0xC	Next Descriptor Address



9.4.1.4 DMA Channel Operation

Since a PCI device, Microengine, or the Intel XScale® core can access the internal CSRs and memory in a similar way, the DMA channel operation description that follows will apply to all channels. CHAN_1_, CHAN_2_, or CHAN_3_ can be placed before the name for the DMA registers.

The DMA channel owner can either set up the descriptors in SRAM or it can write the first descriptor directly to the DMA channel registers.

When descriptors and the descriptor list are in SRAM, the procedure is as follows:

1. The DMA channel owner writes the address of the first descriptor into the DMA Channel Descriptor Pointer register (DESC_PTR).
2. The DMA channel owner writes the DMA Channel Control register (CONTROL) with miscellaneous control information and also sets the channel enable bit (bit 0). The channel initial descriptor bit (bit 4) in the CONTROL register must also be cleared to indicate that the first descriptor is in SRAM.
3. Depending on the DMA channel number, the DMA channel reads the descriptor block into the corresponding DMA registers, BYTE_COUNT, PCI_ADDR, DRAM_ADDR, and DESC_PTR.
4. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done (bit 2) in the CONTROL register.
5. If the end of chain bit (bit 31) in the BYTE_COUNT register is clear, the channel checks the Chain Pointer value. If the Chain Pointer value is not equal to 0, it reads the next descriptor and transfers the data (step 3 and 4 above). If the Chain Pointer value is equal to 0, it waits for the Descriptor Added bit of the Channel Control register to be set before reading the next descriptor and transfers the data (step 3 and 4 above). If bit 31 is set, the channel sets the channel chain done bit (bit 7) in the CONTROL register and then stops.
6. Proceed to the Channel End Operation.

When single descriptors are written directly into the DMA channel registers, the procedure is as follows:

1. The DMA channel owner writes the descriptor values directly into the DMA channel registers. The end of chain bit (bit 31) in the BYTE_COUNT register must be set, and the value in the DESC_PTR register is not used.
2. The DMA channel owner writes the base address of the DMA transfer into the PCI_ADDR to specify the PCI starting address.
3. When the first descriptor is in the BYTE_COUNT register, the DRAM_ADDR register must be written with the address of the data to be moved.
4. The DMA channel owner writes the CONTROL register with miscellaneous control information, along with setting the channel enable bit (bit 0). The channel initial descriptor in register bit (bit 4) in the CONTROL register must also be set to indicate that the first descriptor is already in the channel descriptor registers.
5. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit (bit 2) in the CONTROL register.
6. Since the end of the chain bit (bit 31) in the BYTE_COUNT register is set, the channel sets the channel chain done bit (bit 7) in the CONTROL register and then stops.
7. Proceed to the Channel End Operation.

9.4.1.5 DMA Channel End Operation

1. Channel owned by PCI
If not masked via the PCI Outbound Interrupt Mask register, the DMA channel interrupts the PCI host after the setting of the DMA done bit in the `CHAN_X_CONTROL` register, which is readable in the PCI Outbound Interrupt Status register.
2. Channel owned by the Intel XScale® core
If enabled via the Intel XScale® core Interrupt Enable registers, the DMA channel interrupts the Intel XScale® core by setting the DMA channel done bit in the `CHAN_X_CONTROL` register, which is readable in the Intel XScale® core Interrupt Status register.
3. Channel owned by Microengine
If enabled via the Microengine Auto-Push Enable registers, the DMA channel signals the Microengine after setting the DMA channel done bit in the `CHAN_X_CONTROL` register, which is readable in the Microengine Auto-Push Status register.

9.4.1.6 Adding Descriptor to an Unterminated Chain

It is possible to add a descriptor to a chain while a channel is running. To do so the chain should be left un-terminated, that is the last descriptor should have End of Chain clear, and the Chain Pointer value equal to 0. A new descriptor (descriptors) can be added to the chain by overwriting the Chain Pointer value of the un-terminated descriptor (in SRAM) with the Local Memory address of the (first) added descriptor (Note that the added descriptor must actually be valid in Local Memory prior to that). After updating the Chain Pointer field, the software must write a 1 to the Descriptor Added bit of the Channel Control register. This is necessary for the case where the channel was paused in order to re-activate the channel. However, software need not check the state of the channel before writing that bit; there is no side-effect of writing that bit in the case where the channel had not yet read the unlinked descriptor.

If the channel was paused or had read an unlinked Pointer, it will re-read the last descriptor processed (i.e., the one that originally had the zero value for Chain Pointer) to get the address of the newly added descriptor.

A descriptor can not be added to a descriptor which has End of Chain set.

9.4.1.7 DRAM to PCI Transfer

For a DRAM-to-PCI transfer, the DMA channel reads data from DRAM and places it into the DMA buffer for transfer to the FBus FIFO when the following conditions are met:

- There is at least free space for a read block in the buffer.
- The DRAM controller issues data valid on DRAM push data bus to the DMA engine.
- DMA transfer is not done.

Before data is stored into the DMA buffer, the DRAM starting address is evaluated. Extra data will be discarded in case the DRAM starting address does not start at aligned addresses. The lower address bits determine the byte enables for the first data double Dword. At the end of the DMA transfer, extra data will be discarded and byte enables are calculated for the last 64-bit double Dword. After the data is loaded into the buffer, the PCI starting address is evaluated and the buffer is shifted byte wise to align the starting DRAM data with the starting PCI starting address.



A 64-bit double Dword with byte enables is pushed into the FBus FIFO from the DMA buffers as soon as there is data available in the buffer and there is space in the FBus FIFO. The Core logic will transfer the exact number of bytes to the PCI Bus. The maximum burst size on the PCI bus varies according to the stepping and is described in [Table 129](#)

Table 129. PCI Maximum Burst Size

Stepping	Description
A Stepping	The maximum burst size is 64 bytes.
B Stepping	<p>The maximum burst size can be greater than 64 bytes for certain operations.</p> <p>The register PCI_IXP_PARAM configures the burst length for target write operations.</p> <p>The register CHAN_#_CONTROL configures the burst length for DMA read and write operations.</p> <p>The register PCI_CONTROL configures the atomic feature for target write operations of 64 bytes or fewer.</p> <p>Note: Bursts longer than 64 bytes are not supported for PCI target read operations.</p>

9.4.1.8 PCI to DRAM Transfer

The DMA channel issues a sequence of PCI read request commands through the FBus address FIFO to read the precise byte count from PCI.

The DMA engine will continue to load the DMA write buffer with FBus FIFO data as soon as data is available.

The DMA engine determines the largest size of memory request possible with the current DRAM address and remaining byte count. It also has to make sure there is enough data in the write buffer before sending the memory request.

9.4.2 Push/Pull Command Bus Target Interface

Through the command bus target interface, the command bus masters (PCI, Intel XScale® core, and Microengines) can access the PCI Unit internal registers including the local PCI configuration registers and the local PCI Unit CSRs. Also, the Microengine and the Intel XScale® core can issue transactions on the PCI bus. The requests are generated from the command master to the command bus arbiter. The arbiter selects a master and sends it a grant. That master then sends a command, which is passed through by the arbiter.

PCI Unit will issue the push and pull data responses to the SRAM push/pull data buses. When the read command is received, the PCI Unit will issue the push data request on the SRAM push data bus. When the write command is received, PCI Unit will issue the pull command on the SRAM pull data bus.

9.4.2.1 Command Bus Master Access to Local Configuration Registers

The configuration register within the PCI unit can be accessed by push/pull command bus access to configuration space through the FBus interface of the PCI core. When the IXP2800 Network Processor is a PCI host, these registers have to be accessed through this internal path and no PCI bus cycle will be generated.

9.4.2.2 Command Bus Master Access to Local Control and Status Registers

These are CSRs within the PCI Unit that are accessible from push/pull bus masters. The masters include the Intel XScale® core, Microengines. There is no PCI bus cycles generated. The CSRs within the PCI Unit can be accessed internally by external PCI devices.

9.4.2.3 Command Bus Master Direct Access to PCI Bus

The Intel XScale® core and Microengines are the only command bus masters that have direct access to the PCI bus as a PCI Bus initiator. The PCI Bus can be accessed by push/pull command bus access to PCI bus address space. The PCI Unit will share the internal SRAM push/pull data bus with SRAM for the data transfers.

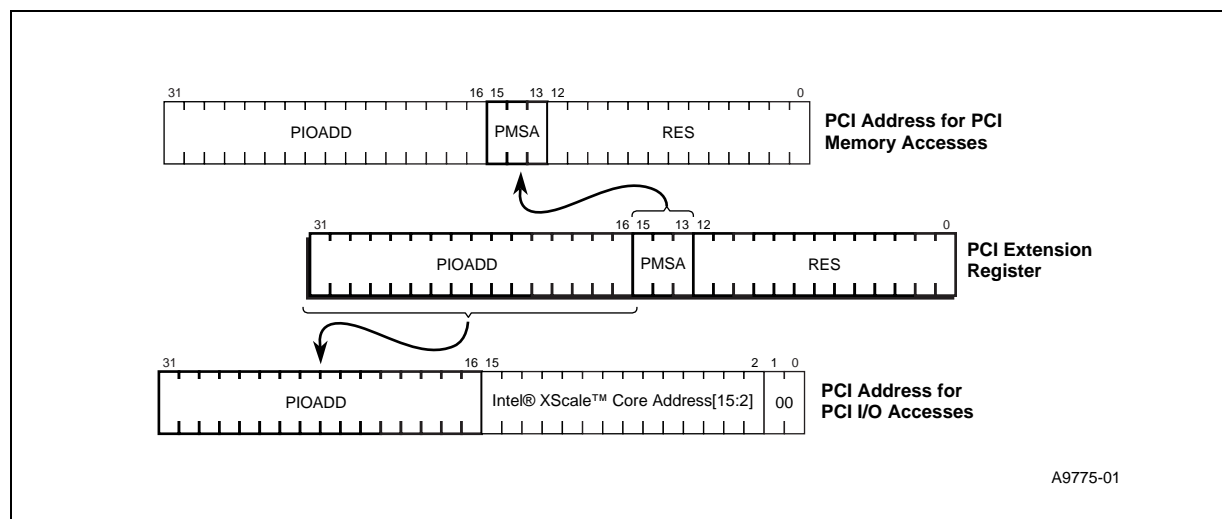
The data from the SRAM push/pull data bus are transferred through the master data port of the FBus interface of the PCI core. The PCI Core will handle all the PCI Bus protocol handshakes. The SRAM pull data received for a write command will be transferred to the Master write FIFO for PCI writes. For PCI reads, data is transferred from the read FIFO to the SRAM push data bus. A 32-byte Direct buffer is used to support up to 32 bytes of data responses to the direct access to PCI Bus.

The Command Bus Master access to PCI bus will require internal arbitration to gain access to the data FIFOs inside the core, which are shared between the DMA engine and direct access to PCI.

9.4.2.3.1 PCI Address Generation for IO and MEM cycles

When push/pull command bus master is accessing the PCI Bus, the PCI address is generated based on the PCI address extension register (PCI_ADDR_EXT). Figure 128 shows how the address is generated from a Command Bus Master address.

Figure 128. PCI Address Generation for Command Bus Master to PCI





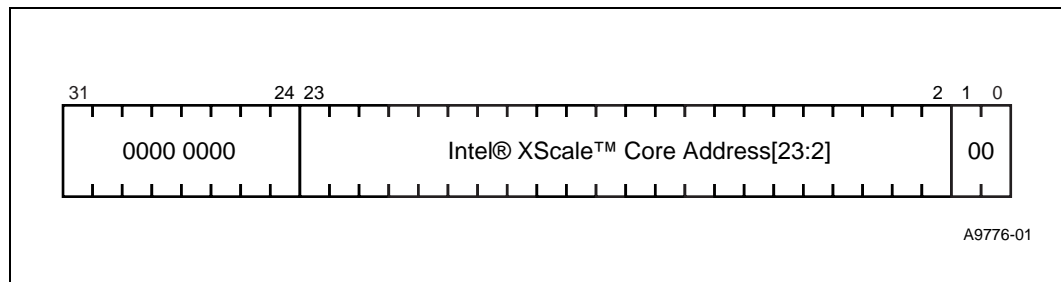
9.4.2.3.2 PCI Address Generation for Configuration Cycles

When a push/pull command bus master is accessing the PCI Bus to generate a configuration cycle, the PCI address is generated based on the a Command Bus Master address as shown in [Table 130](#) and [Figure 129](#):

Table 130. Command Bus Master Configuration Transactions

Cycle	Result
Type 1 Configuration Cycle	Command Bus address bits [31:24] are equal to 0xDA
Type 0 Configuration Cycle	Command Bus address bits [31:24] are equal to 0xDB.

Figure 129. PCI Address Generation for Command Bus Master to PCI Configuration Cycle



9.4.2.3.3 PCI Address Generation for Special and IACK Cycles

The PCI address is undefined for special and IACK PCI cycles

9.4.2.3.4 PCI Enables

The PCI byte enables are generated based on the Command Bus Master instruction and the PCI unit does not change the states of the enables.

9.4.2.3.5 PCI Command

The PCI command is derived from the Command Bus Master address space map. The different spaces supported are listed in [Table 131](#):

Table 131. Command Bus Master Address Space Map to PCI

PCI Command	Intel XScale® Core Address Space
PCI Memory	0xE000 0000 to 0xFFFF FFFF
Local CSR	0xDF00 0000 to 0xDFFF FFFF
Local Configuration Register	0xDE00 0000 to 0xDEFF FFFF
PCI Special Cycle/PCI IACK Read	0xDC00 0000 to 0xDDFF FFFF
PCI Type 1 Configuration Cycle	0xDB00 0000 to 0xDBFF FFFF
PCI Type 0 Configuration Cycle	0xDA00 0000 to 0DAFF FFFF
PCI I/O	0xD800 0000 to 0xD8FF FFFF

9.5 PCI Unit Error Behavior

9.5.1 PCI Target Error Behavior

9.5.1.1 Target Access Has an Address Parity Error

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. PCI core will not claim the cycle regardless of internal device select signal.
 - b. PCI core will let the cycle terminate with master abort.
 - c. PCI core will not assert PCI_SERR#.
 - d. Slave Interface sets PCI_CONTROL[TGT_ADR_ERR], which will interrupt the Intel XScale® core if enabled.

9.5.1.2 Initiator Asserts PCI_PERR# in Response to One of Our Data Phases

1. Core does nothing.
2. Responsibility lies with the initiator to discard data, report this to the system, etc.

9.5.1.3 Discard Timer Expires on a Target Read

1. PCI unit discards the read data.
2. PCI Unit invalidates the delayed read address
3. PCI Unit sets Discard Timer Expired bit (DTE) in the PCI_CONTROL.
4. If enabled (XSCALE_INT_ENABLE [DTE]), the PCI unit interrupts the Intel XScale® core.

9.5.1.4 Target Access to the PCI_CSR_BAR Space Has Illegal Byte Enables

Note: The acceptable byte enables are BE[3:0] = 0x0 or 0xF.

1. Slave Interface will set PCI_CONTROL[TGT_CSR_BE]
2. Slave Interface will issue target abort for target read and drop the transaction for target write.

9.5.1.5 Target Write Access Receives Bad Parity PCI_PAR with the Data

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. core asserts PCI_PERR# and sets PCI_CMD_STAT[PERR].
 - b. Slave Interface sets PCI_CONTROL[TGT_WR_PAR], which will interrupt the Intel XScale® core if enabled.
 - c. Data is discarded.



9.5.1.6 SRAM Responds With a Memory Error on One or More Data Phases on a Target Read

1. Slave Interface sets PCI_CONTROL[TGT_SRAM_ERR], which will interrupt the Intel XScale® core if enabled.
2. Assert PCI Target Abort at or before the data in question is driven on PCI.

9.5.1.7 DRAM Responds With a Memory Error on One or More Data Phases on a Target Read

1. Slave Interface sets PCI_CONTROL[TGT_DRAM_ERR], which will interrupt the Intel XScale® core if enabled.
2. Slave Interface asserts PCI Target Abort at or before the data in question is driven on PCI.

9.5.2 As a PCI Initiator During a DMA Transfer

9.5.2.1 DMA Read From DRAM (Memory-to-PCI Transaction) Gets a Memory Error

1. Set PCI_CONTROL[DMA_DRAM_ERR] which will interrupt the Intel XScale® core if enabled.
2. Master Interface terminates transaction before bad data is transferred (okay to terminate earlier).
3. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
4. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
5. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
6. Master Interface resets the state machines and DMA buffers.

9.5.2.2 DMA Read From SRAM (Descriptor Read) Gets a Memory Error

1. Set PCI_CONTROL[DMA_SRAM_ERR] which will interrupt the Intel XScale® core if enabled.
2. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
3. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
4. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
5. Master Interface resets the state machines and DMA buffers.



9.5.2.3 DMA From DRAM Transfer (Write to PCI) Receives PCI_PERR# on PCI Bus

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. Master Interface sets PCI_CONTROL[DPE] which will interrupt the Intel XScale® core if enabled.
 - b. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
 - c. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
 - d. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
 - e. Master Interface resets the state machines and DMA buffers.
 - f. Core sets PCI_CMD_STAT[PERR] if properly enabled.

9.5.2.4 DMA To DRAM (Read from PCI) Has Bad Data Parity

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. Core asserts PCI_PERR# on PCI if PCI_CMD_STAT[PERR_RESP] is set.
 - b. Master Interface sets PCI_CONTROL[DPED] which can interrupt the Intel XScale® core if enabled.
 - c. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
 - d. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
 - e. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
 - f. Master Interface resets the state machines and DMA buffers.

9.5.2.5 DMA Transfer Experiences a Master Abort (Time-Out) on PCI

Note: That is, nobody asserts DEVSEL during the DEVSEL window.

1. Master Interface sets PCI_CONTROL[RMA] which will interrupt the Intel XScale® core if enabled.
2. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
3. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
4. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
5. Master Interface resets the state machines and DMA buffers



9.5.2.6 DMA Transfer Receives a Target Abort Response During a Data Phase

1. Core terminates the transaction.
2. Master Interface sets PCI_CONTROL[RTA] which can interrupt the Intel XScale® core if enabled.
3. Master Interface clears the Channel Enable bit in CHAN_X_CONTROL.
4. Master Interface sets DMA channel error bit in CHAN_X_CONTROL.
5. Master Interface does not reset the DMA CSRs; This leaves the descriptor pointer pointing to the DMA descriptor of the failed transfer.
6. Master Interface resets the state machines and DMA buffers.

9.5.2.7 DMA Descriptor Has a 0x0 Word Count (Not an Error)

1. No data is transferred.
2. Descriptor is retired normally.

9.5.3 As a PCI Initiator During a Direct Access from the Intel XScale® Core or Microengine

9.5.3.1 Master Transfer Experiences a Master Abort (Time-Out) on PCI

1. Core aborts the transaction.
2. Master Interface sets PCI_CONTROL[RMA] which will interrupt the Intel XScale® core if enabled.

9.5.3.2 Master Transfer Receives a Target Abort Response During a Data Phase

1. Core aborts the transaction.
2. Master Interface sets PCI_CONTROL[RTA] which will interrupt the Intel XScale® core if enabled.

9.5.3.3 Master from the Intel XScale® Core or Microengine Transfer (Write to PCI) Receives PCI_PERR# on PCI Bus

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. Core sets PCI_CMD_STAT[PERR].
 - b. Master Interface sets PCI_CONTROL[DPE] which will interrupt the Intel XScale® core if enabled.



9.5.3.4 Master Read From PCI (Read from PCI) Has Bad Data Parity

1. If PCI_CMD_STAT[PERR_RESP] is not set, PCI Unit will ignore the parity error.
2. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. Core asserts PCI_PERR# on PCI.
 - b. Master Interface sets PCI_CONTROL[DPED] which will interrupt the Intel XScale® core if enabled.
 - c. Data that has been read from PCI is sent to the Intel XScale® core or Microengine.

9.5.3.5 Master Transfer Receives PCI_SERR# from the PCI Bus

Master Interface sets PCI_CONTROL[RSERR] which will interrupt the Intel XScale® core if enabled.

9.5.3.6 Intel XScale® Core Microengine Requests Direct Transfer when the PCI Bus is in Reset

Master Interface will complete the transfer and drop the write data and return all ones on the read data.

9.6 PCI Data Byte Lane Alignment

During any endian conversion, PCI does not need to do any long word swapping between two 32 bits long words(LW1, LW0). But PCI may need to do byte swapping within the 32-bits long word. Because of the different endian convention between PCI Bus and the memory, all data going between the PCI core FIFO and memory data bus passes through the byte lane reversal as shown in Table 132 through Table 139.

PCI allows byte-enable swapping only without the data swapping or allow data swapping only without byte enable swapping. When PCI handle the mis align data in above two cases, PCI will only care about valid data. So PCI will drive any data values for those misalign invalid data portions.

Table 132. Byte Lane Alignment for 64-Bit PCI Data In (64 Bits PCI Little Endian to Big Endian with Swap)

PCI Data	IN[63:56]	IN[55:48]	IN[47:40]	IN[39:32]	IN[31:24]	IN[23:16]	IN[15:8]	IN[7:0]
SRAM Data	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 (32 bits) LW0 drive first			
DRAM Data	OUT[39:32]	OUT[47:40]	OUT[55:48]	OUT[63:56]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]



Table 133. Byte Lane Alignment for 64-bit PCI Data In (64 Bits PCI Big Endian to Big Endian without Swap)

PCI Data	IN[39:32]	IN[47:40]	IN[55:48]	IN[63:56]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
SRAM Data	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 (32 bits) LW0 drive first			
DRAM Data	OUT[39:32]	OUT[47:40]	OUT[55:48]	OUT[63:56]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]

Table 134. Byte Lane Alignment for 32-bit PCI Data In (32 Bits PCI Little Endian to Big Endian with Swap)

	PCI Add[2]=1				PCI Add[2]=0			
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
PCI Data	IN[31:24]	IN[23:16]	IN[15:8]	IN[7:0]	IN[31:24]	IN[23:16]	IN[15:8]	IN[7:0]
SRAM Data	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
DRAM Data	OUT[39:32]	OUT[47:40]	OUT[55:48]	OUT[63:56]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]

Table 135. Byte Lane Alignment for 32-bit PCI Data In (32 Bits PCI Big Endian to Big Endian without Swap)

	PCI Add[2]=1				PCI Add[2]=0			
	Long Word1 (32 bits) LW1 drive first				Long Word0 ((32 bits) LW0 drive first			
PCI Data	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
SRAM Data	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
direct map pci to dram	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
DRAM Data	OUT[39:32]	OUT[47:40]	OUT[55:48]	OUT[63:56]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]

Table 136. Byte Lane Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Little Endian with Swap)

SRAM Data	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
DRAM Data	IN[39:32]	IN[47:40]	IN[55:48]	IN[63:56]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
PCI Side	OUT[63:56]	OUT[55:48]	OUT[47:40]	OUT[39:32]	OUT[31:24]	OUT[23:16]	OUT[15:8]	OUT[7:0]



Table 137. Byte Lane Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Big Endian without Swap)

SRAM Data	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
DRAM Data	IN[39:32]	IN[47:40]	IN[55:48]	IN[63:56]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
direct map pci to dram	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
PCI Side	OUT[39:32]	OUT[47:40]	OUT[55:48]	OUT[63:56]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]

Table 138. Byte Lane Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Little Endian with Swap)

SRAM Data	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
DRAM Data	IN[39:32]	IN[47:40]	IN[55:48]	IN[63:56]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
PCI Data	OUT[31:24]	OUT[23:16]	OUT[15:8]	OUT[7:0]	OUT[31:24]	OUT[23:16]	OUT[15:8]	OUT[7:0]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
	PCI Add[2]=1				PCI Add[2]=0			

Table 139. Byte Lane Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Big Endian without Swap)

SRAM Data	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
DRAM Data	IN[39:32]	IN[47:40]	IN[55:48]	IN[63:56]	IN[7:0]	IN[15:8]	IN[23:16]	IN[31:24]
PCI Data	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]	OUT[7:0]	OUT[15:8]	OUT[23:16]	OUT[31:24]
	Long Word1 (32 bits) LW1 drive after LW0				Long Word0 ((32 bits) LW0 drive first			
	PCI Add[2]=1				PCI Add[2]=0			

The BE_DEMI bit of the PCI_CONTROL register can be set to enable big endian on the incoming data from the PCI Bus to both the SRAM and DRAM. The BE_DEMO bit of the PCI_CONTROL register can be set to enable big endian on the outgoing data to the PCI Bus from both the SRAM and DRAM.



9.6.1 Endian for Byte Enable

During any endian conversion, PCI does not need to do any long word byte enable swapping between two 32-bit long words(LW1, LW0). But PCI may need to do byte enable swapping within the 32 bits long word byte enable. Because of the different endian convention between PCI Bus and the memory, all data going between the PCI core FIFO and memory data bus passes through the byte lane reversal as shown in Table 140 through Table 147:

Table 140. Byte Enable Alignment for 64-bit PCI Data In (64 Bits PCI Little Endian to Big Endian with Swap)

PCI Data	IN_BE[7]	IN_BE[6]	IN_BE[5]	IN_BE[4]	IN_BE[3]	IN_BE[2]	IN_BE[1]	IN_BE[0]
SRAM Data	OUT_BE[3]	OUT_BE[2]	OUT_BE[1]	OUT_BE[0]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]
	Long Word1 byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	OUT_BE[4]	OUT_BE[5]	OUT_BE[6]	OUT_BE[7]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]

Table 141. Byte Enable Alignment for 64-bit PCI Data In (64 Bits PCI Big Endian to Big Endian without Swap)

PCI Data	IN_BE[4]	IN_BE[5]	IN_BE[6]	IN_BE[7]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
SRAM Data	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]
	Long Word1 byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	OUT_BE[4]	OUT_BE[5]	OUT_BE[6]	OUT_BE[7]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]

Table 142. Byte Enable Alignment for 32-bit PCI Data In (32 bits PCI Little Endian to Big Endian with Swap)

	PCI Add[2]=1				PCI Add[2]=0			
	Long Word1 byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
PCI Data	IN_BE[3]	IN_BE[2]	IN_BE[1]	IN_BE[0]	IN_BE[3]	IN_BE[2]	IN_BE[1]	IN_BE[0]
SRAM Data	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]
	Long Word1 byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	OUT_BE[4]	OUT_BE[5]	OUT_BE[6]	OUT_BE[7]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]

Table 143. Byte Enable Alignment for 32-bit PCI Data In (32 Bits PCI Big Endian to Big Endian without Swap)

	PCI Add[2]=1				PCI Add[2]=0			
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
PCI Data	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
SRAM Data	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
direct map pci to dram	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
DRAM Data	OUT_BE[4]	OUT_BE[5]	OUT_BE[6]	OUT_BE[7]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]

Table 144. Byte Enable Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Little Endian with Swap)

SRAM Data	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	IN_BE[4]	IN_BE[5]	IN_BE[6]	IN_BE[7]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
PCI Side	OUT_BE[7]	OUT_BE[6]	OUT_BE[5]	OUT_BE[4]	OUT_BE[3]	OUT_BE[2]	OUT_BE[1]	OUT_BE[0]

Table 145. Byte Enable Alignment for 64-bit PCI Data Out (Big Endian to 64 Bits PCI Big Endian without Swap)

SRAM Data	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	IN_BE[4]	IN_BE[5]	IN_BE[6]	IN_BE[7]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
PCI Side	OUT_BE[4]	OUT_BE[5]	OUT_BE[6]	OUT_BE[7]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]

Table 146. Byte Enable Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Little Endian with Swap)

SRAM Data	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	IN_BE[4]	IN_BE[5]	IN_BE[6]	IN_BE[7]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
PCI Data	OUT_BE[3]	OUT_BE[2]	OUT_BE[1]	OUT_BE[0]	OUT_BE[3]	OUT_BE[2]	OUT_BE[1]	OUT_BE[0]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
	PCI Add[2]=1				PCI Add[2]=0			



Table 147. Byte Enable Alignment for 32-bit PCI Data Out (Big Endian to 32 Bits PCI Big Endian without Swap)

SRAM Data	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
DRAM Data	IN_BE[4]	IN_BE[5]	IN_BE[6]	IN_BE[7]	IN_BE[0]	IN_BE[1]	IN_BE[2]	IN_BE[3]
PCI Data	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]	OUT_BE[0]	OUT_BE[1]	OUT_BE[2]	OUT_BE[3]
	Long Word1byte enable LW1 byte enable drive after LW0 byte enable				Long Word0 byte enable LW0 byte enable drive first			
	PCI Add[2]=1				PCI Add[2]=0			

The BE_BEMI bit of the PCI_CONTROL register can be set to enable big endian on the incoming byte enable from the PCI Bus to both the SRAM and DRAM. The BE_BEMO bit of the PCI_CONTROL register can be set to enable big endian on the outgoing byte enable to the PCI Bus from both the SRAM and DRAM.

The B-stepping silicon provides a mechanism to enable byte swapping for PCI I/O operations as described in [Table 148](#).

Table 148. PCI I/O Cycles with Data Swap Enable

Stepping	Description			
A Stepping	A PCI IO cycle is treated like CSR where the data bytes are not swapped. It is sent in the same byte order whether the PCI bus is configured in Big Endian or Little Endian mode.			
B Stepping	When PCI_CONTROL[IEE] is 0, PCI data is sent in the same byte order whether the PCI bus is configured in Big Endian or Little Endian mode. When PCI_CONTROL[IEE] is 1, PCI IO data will follow the same memory space swapping rule. The address always follows the physical location, Example:			
	BEs not Swapped(1 byte access)		BEs Swapped(1 byte access)	
	ad[1:0]	BE3 BE2 BE1 BE0	ad[1:0]	BE3 BE2 BE1 BE0
	0 0	1 1 1 0	1 1	0 1 1 1
	0 1	1 1 0 1	1 0	1 0 1 1
	1 0	1 0 1 1	0 1	1 1 0 1
	1 1	0 1 1 1	0 0	1 1 1 0
	BEs not Swapped(2 byte access)		BEs Swapped(2 byte access)	
	ad[1:0]	BE3 BE2 BE1 BE0	ad[1:0]	BE3 BE2 BE1 BE0
	0 0	1 1 0 0	1 0	0 0 1 1
	0 1	1 0 0 1	0 1	1 0 0 1
	1 0	0 0 1 1	0 0	1 1 0 0
	BEs not Swapped(3 byte access)		BEs Swapped(3 byte access)	
	ad[1:0]	BE3 BE2 BE1 BE0	ad[1:0]	BE3 BE2 BE1 BE0
	0 0	1 0 0 0	0 1	0 0 0 1
	0 1	0 0 0 1	0 0	1 0 0 0
	BEs not Swapped(4 byte access)		BEs Swapped(4 byte access)	
	ad[1:0]	BE3 BE2 BE1 BE0	ad[1:0]	BE3 BE2 BE1 BE0
	0 0	0 0 0 0	0 0	0 0 0 0



Clocks, Reset, and Initialization

10

This section describes the IXP2800 Network Processor clocks, reset, and initialization sequence.

10.1 Clocks

The block diagram in [Figure 130](#) shows how the IXP2800 Network Processor implements an onboard clock generator to generate the internal clocks used by the various functional units in the device. It takes an external reference frequency and multiplies it to a higher frequency clock using a PLL. That clock is then divided down by a set of programmable dividers to provide clocks to SRAM and DRAM controllers. The Intel XScale® core and MEs get clocks using fixed divide ratios. The Media and Switch Fabric Interface clock is selected based on the strap pin (CFG_MSF_FREQ_SEL) so that when CFG_MSF_FREQ_SEL is high, internally generated clock using the programmable divider is used and when CFG_MSF_FREQ_SEL is low, externally received clock on MSF interface is used. PCI controller use external clocks. Each of the units also interfaces to internal busses, which run at ½ the Microengine frequency. [Figure 130](#) shows the overall clock generation and distribution. [Table 149](#) summarizes the clock usage.

Figure 130. Overall Clock Generation and Distribution

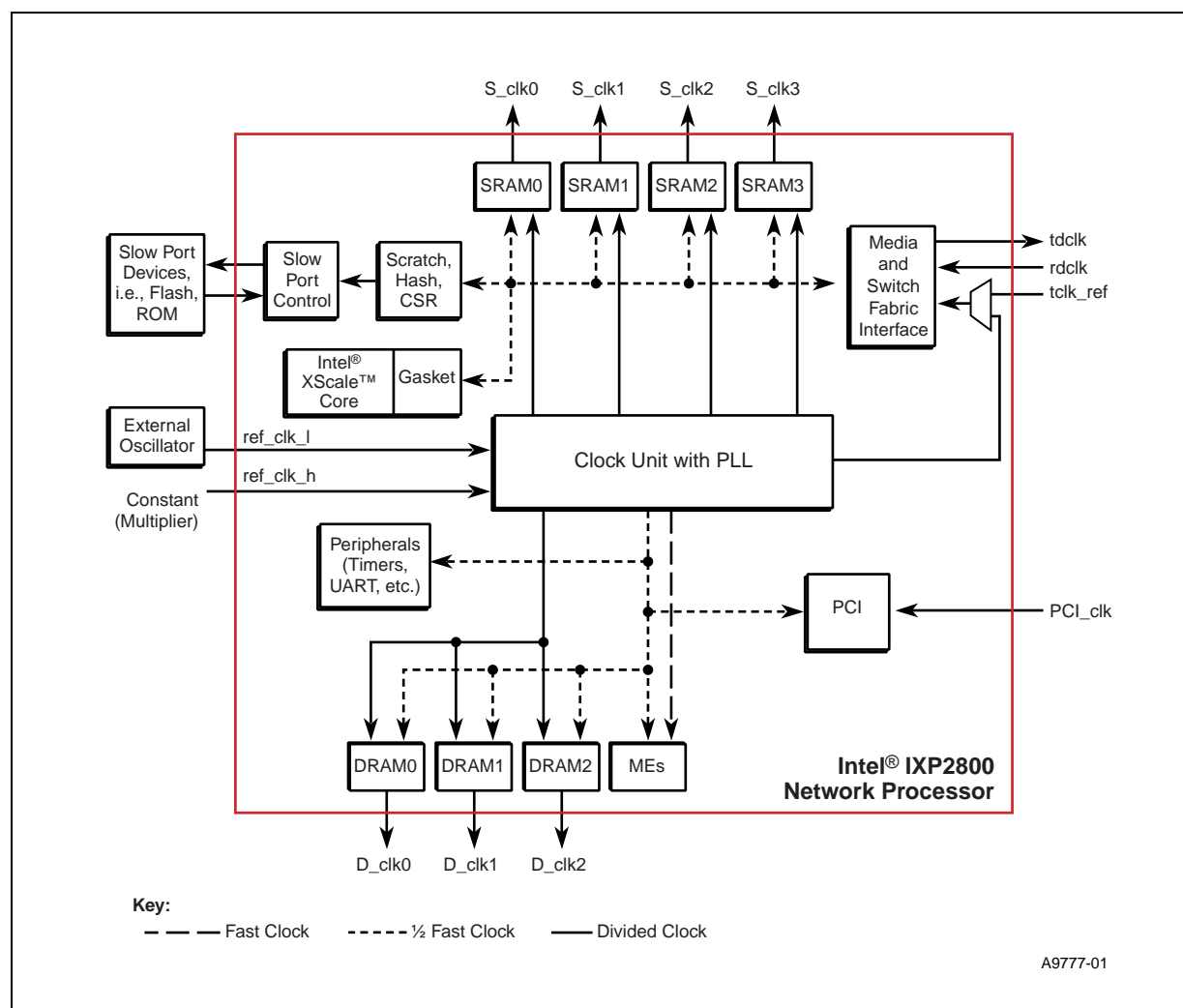


Table 149. Clock Usage Summary

Unit Name	Description	Comment
Microengine	MEs internal.	
Internal Busses	Command/Push/Pull interface of DRAM, SRAM, Intel XScale® core, Peripheral, MSF, and PCI Units.	1/2 Microengine frequency.
Intel XScale® core	Intel XScale® core microprocessor, caches, microprocessor side of Gasket.	1/2 of Microengine frequency.
DRAM	DRAM pins and control logic (all of DRAM unit except Internal Bus interface).	Divide of Microengine frequency. All DRAM channels use the same frequency. Clocks are driven by IXP2800 to external DRAMs.

Table 149. Clock Usage Summary (Continued)

Unit Name	Description	Comment
SRAM	SRAM pins and control logic (all of SRAM unit except Internal Bus interface).	Divide of Microengine frequency. Each SRAM channel has its own frequency selection. Clocks are driven by IXP2800 to external SRAMs and/or Coprocessors.
Scratch, Hash, CSR	Scratch RAM, Hash Unit, CSR access block	½ of Microengine frequency. Note that Slow Port has no clock. Timing for Slow Port accesses is defined in Slow Port registers.
MSF	Receive and Transmit pins and control logic.	The transmit clock for the Media and Switch interface can be derived in two different ways. <ul style="list-style-type: none"> From TCLK input signal (supplied by PHY device). Divided from internal clock. For details please refer to Chapter 8, “Media and Switch Fabric Interface” .
APB	APB logic	Divide of Microengine frequency
PCI	PCI pins and control logic.	External reference. Either from Host system or on-board oscillator.

The fast frequency on the IXP2800 Network Processor is generated by an on-chip PLL that multiplies a reference frequency provided by an on-board LVDS oscillator (frequency 100 MHz) by a selectable multiplier. The multiplier is selected by using external strap pins SP_AD[5:0] and can be viewed by software via the STRAP_OPTIONS[CFG_PLL_MULT] CAP CSR register bits. The multiplier range is even multiples between 16 and 48, so the PLL can generate a 1.6 GHz to 4.8 GHz clock (with a 100Mhz reference frequency).

The PLL output frequency is divided by 2 to get the ME clock and by 4 to get the Intel XScale® core and the internal Command/Push/Pull bus frequency. An additional division (after the divide by 2) is used to generate the clock frequencies for the other internal units. The divisors are programmable via the CLOCK_CONTROL CSR. APB divisor specified in the CLOCK_CONTROL CSR clock is scaled by 4 (that is a value of 2 in the CSR selects a divisor of 8).

[Table 150](#) shows the frequencies that are available based on a 100Mhz oscillator and various values of PLL multipliers, for the supported divisor values of 3 to 15.

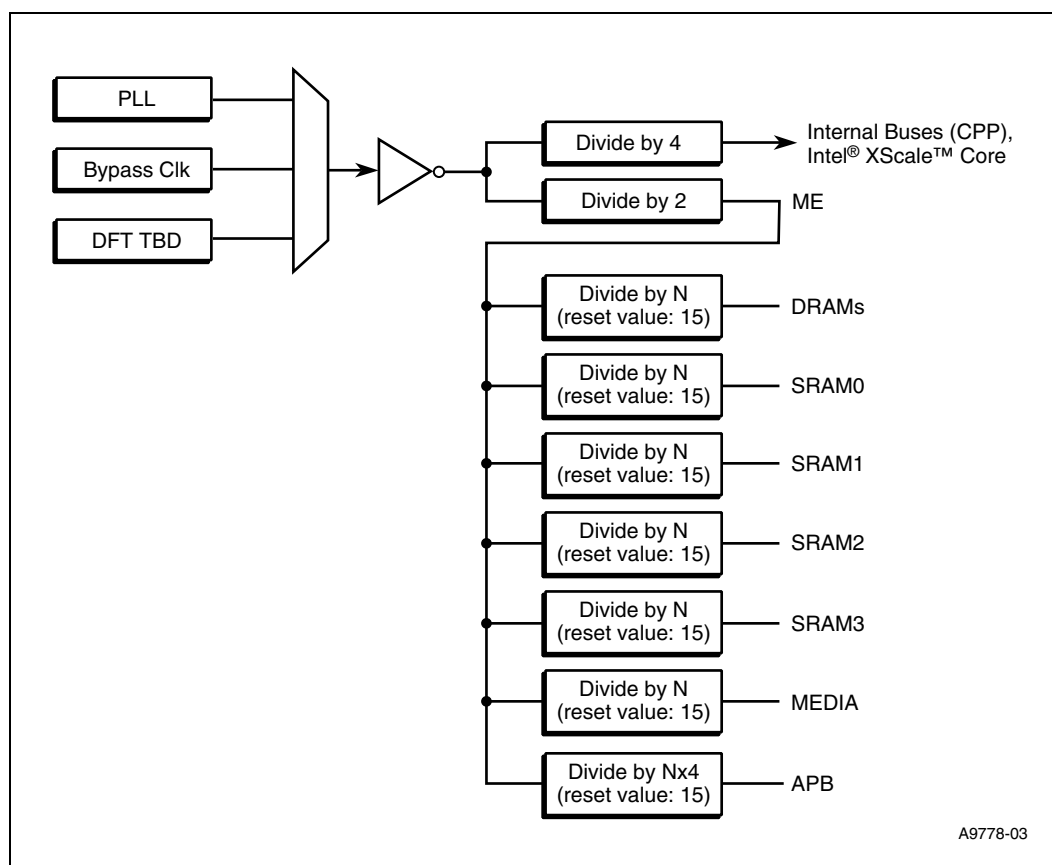
Table 150. Clock Rates Examples

Input Oscillator Frequency (MHz)			100						
PLL Output Frequency (MHz) [PLL Multiplier] ¹			2000 [20]	2200 [22]	2400 [24]	2600 [26]	2800 [28]	4000 [40]	4800 [48]
Microengine Frequency ²			1000	1100	1200	1300	1400	2000	2400
Intel XScale® core & Command/Push/ Pull Bus Frequency ³			500	550	600	650	700	1000	1200
Divide Ratio for other Units (except APB) ⁴	Divisor ⁵	2 ⁶	500	550	600	650	700	1000	1200
		3	333	367	400	433	467	666	800
		4	250	275	300	325	350	500	600
		5	200	220	240	260	280	400	480
		6	167	183	200	217	233	334	400
		7	143	157	171	186	200	286	342
		8	125	138	150	163	175	250	300
		9	111	122	133	144	156	222	266
		10	100	110	120	130	140	200	240
		11	91	100	109	118	127	182	218
		12	83	92	100	108	117	166	200
		13	77	85	92	100	107	154	184
		14	71	79	86	93	100	142	172
		15	67	73	80	87	93	134	160

1. This multiplier is selected via SP_AD[5:0] strap pins.
2. This frequency is the PLL output frequency divided by 2.
3. This frequency is the PLL output frequency divided by 4.
4. The ABP divisor specified in the CLOCK_CONTROL CAP CSR is scaled by an additional x4.
5. This divisor is selected via the CLOCK_CONTROL CAP CSR. The Base Frequency is the PLL output frequency divided by 2.
6. This divide ratio is only used by test logic. In the normal functional mode, this ratio is reserved for Push/Pull clocks only.

Figure 131 shows the clocks generation circuitry for the IXP2800. When the chip is powered up, bypass clock will be sent to all the units. After the PLL is locked, clock unit will switch all units from bypass clock to a fixed frequency clock which is generated by dividing PLL OUTPUT FREQUENCY by 16. Once Clock Control CSR is written, clock unit will replace fixed frequency clock with the defined clocks for different units.

Figure 131. IXP2800 Network Processor Clock Generation



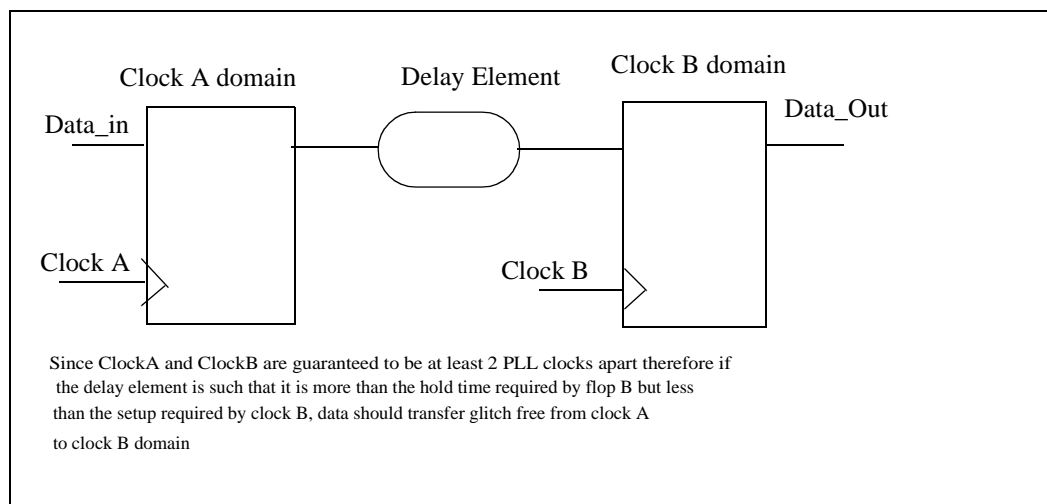
10.2 Synchronization Between Frequency Domains

Due to the internal design architecture of the IXP2800, it is guaranteed that one of the clock domains of an asynchronous transfer will be the Push/Pull domain (PLL/4). Additionally, all other clocks are derived by further dividing the ME clock ($PLL/2n$ where n is 3 or more); refer to Figure 132.

Note: The exception is the PCI unit where the PCI clock is fully asynchronous with the PP clock. Therefore in the PCI unit, data is synchronized using the usual 3 flop synchronization method.

Therefore the clock A and clock B relationship will always be apart by at least 2 PLL clocks. To solve hold problem between clock A and clock B, a delay is added anytime data is transferred from clock A to clock B. The characteristic of this delay element is such that it is high enough to resolve any hold issue in fast environment but in the slow environment its delay is still less than 2 PLL clocks.

Figure 132. Synchronization Between Frequency Domains



10.3 Reset

The IXP2800 Network Processor can be reset four ways.

- Hardware Reset Using nRESET or PCI_RST#
- PCI Initiated Reset
- Watchdog Timer Initiated Reset
- Software Initiated Reset

10.3.1 Hardware Reset Using nRESET or PCI_RST#

The IXP2800 Network Processor provides the nRESET pin so that it can be reset by an external device. Asserting this pin resets the internal functions and generates an external reset via the nRESET_OUT pin.

Upon power-up, nRESET (or PCI_RST#) must remain asserted for 1ms after VDD is stable to properly reset the IXP2800 Network Processor and ensure that the external clocks are stable. While nRESET is asserted, the processor is held in reset. When nRESET is released, the Intel XScale® core begins executing from address 0X0. If PCI_RST# is input to the chip, nRESET should be removed before or at the same time as PCI_RST#.

All the strap options are latched with nRESET except for PCI strap option BOARD_IS_64 which is latched with PCI_RST# only (by latching the status of REQ64# at the trailing edge of PCI_RST#).

If nRESET is asserted, while the Intel XScale® core is executing, the current instruction is terminated abnormally and the reset sequence is initiated.

The nRESET_OUT signal de-assertion depends upon settings of “reset_out_strap” and IXP_RESET0[22] also called EXTRST_EN bit. During power up, IXP_RESET0[22] is reset to “0” therefore value to be driven on nRESET_OUT is defined by “reset_out_strap”. When

“reset_out_strap” is sampled “0” on the trailing edge of reset, nRESET_OUT is de-asserted based on the value of IXP_RESET0[17] which is written by software. If “reset_out_strap” is sampled “1” on the trailing edge of reset, nRESET_OUT is de-asserted after PLL locks.

During normal function mode, if software wants to assert nRESET_OUT, it should set IXP_RESET0[22] and then set IXP_RESET0[17]. To de-assert nRESET_OUT again, software should write IXP_RESET0[17] bit back to “0”.

Figure 133. Reset Out Behavior

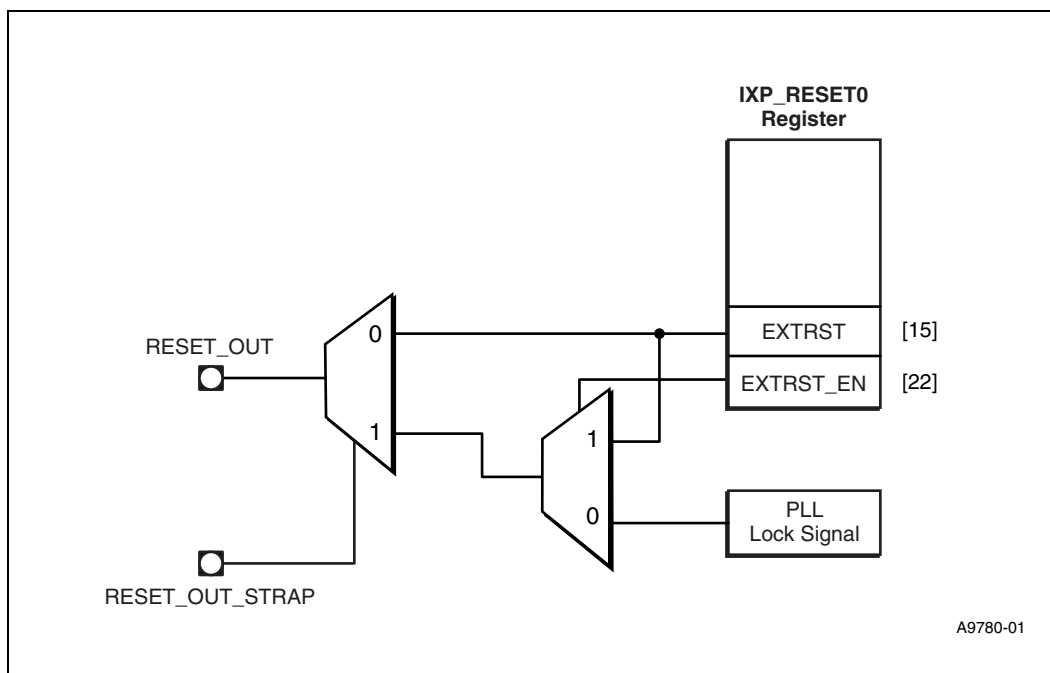
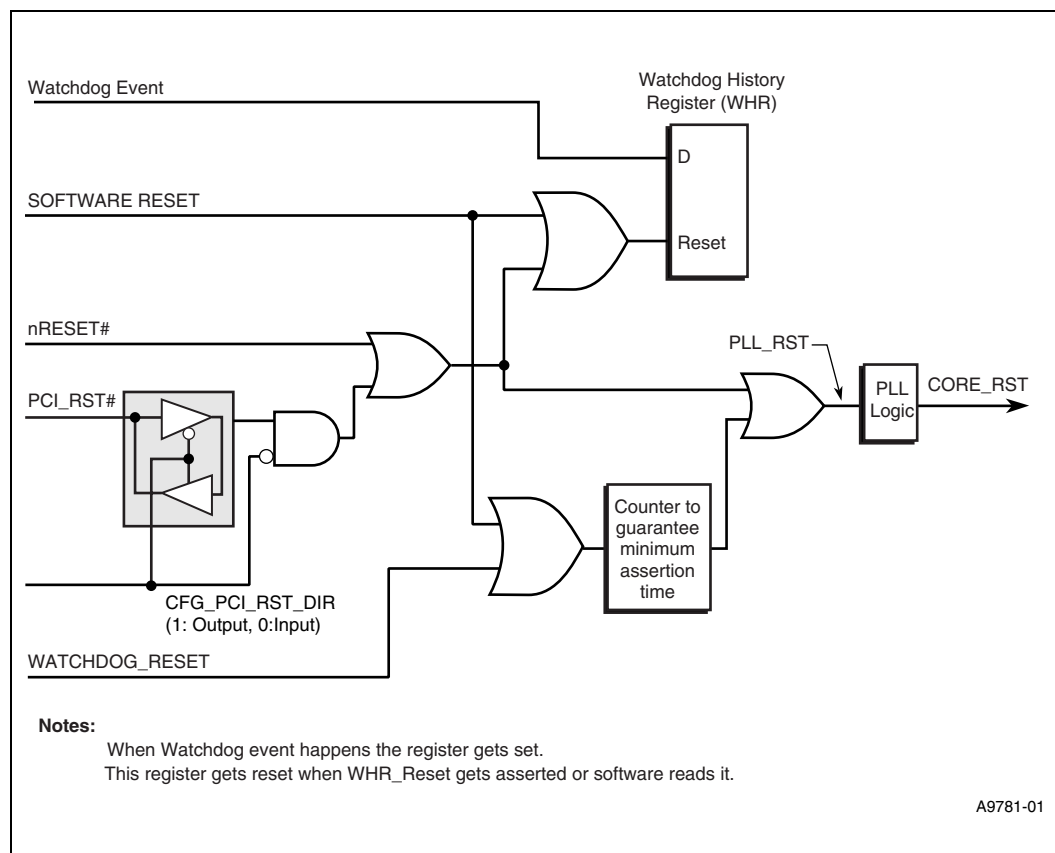


Figure 134. Reset Generation



10.3.2 PCI Initiated Reset

CFG_RST_DIR is not asserted and PCI_RST# is asserted.

When the CFG_RST_DIR strap pin is not asserted (sampled “0”), PCI_RST# is input to the IXP2800 Network Processor and is used to reset all the internal functions. Its behavior is the same as a hardware reset using nRESET pin.

10.3.3 Watchdog Timer Initiated Reset

The IXP2800 Network Processor provides a watchdog timer that can cause a reset if the Watchdog timer expires and the Watchdog enable bit WDE in Timer Watchdog Enable Register is also set. The Intel XScale® core should be programmed to reset the watch dog timer periodically to ensure that the timer does not expire. If a watchdog timer expires, it is assumed that the Intel XScale® core has ceased executing instructions properly. When the timer expires, the Watchdog History Register bit[0] is set which can be read by the software later on.

IXP2800 Network Processor behavior for the watchdog event is defined in the sections that follow.

10.3.3.1 Slave IXP (Non-Central Function)

- If the Watchdog timer reset enable bit set to 1, Watchdog reset will trigger the soft reset
- If the Watchdog timer reset enable bit set to 0, Watchdog reset will trigger the PCI interrupt to external PCI host (if interrupt is enabled by PCI Outbound Interrupt Mask Register[3]). External PCI host can check the IXP2800 error status and log the error then reset the Slave IXP2800 Network Processor only or reset all the PCI devices (assert the PCI_RST_L).
- If the Watchdog history bit is already set when a new watchdog event happens, Watchdog timer reset enable bit is disregarded and soft reset is generated.

10.3.3.2 Master IXP (PCI Host, Central Function)

- If Watchdog timer reset enable bit set to 1, Watchdog reset will trigger the soft reset and set watchdog history bit.
- If Watchdog timer reset enable bit set to 0, check watchdog history bit. If is already set, generate soft reset. If watchdog history bit is not set already, watchdog reset will just set the watchdog history bit and no further action is taken.

10.3.3.3 Master IXP (Central Function)

- If Watchdog timer reset enable bit set to 0, Watchdog reset will trigger the PCI interrupt to external PCI host (if interrupt is enabled by PCI Outbound Interrupt Mask Register[3]).
- If Watchdog history bit is already set when a new watchdog event happens, Watchdog timer reset enable bit is disregarded, and soft reset is generated.
- If Watchdog timer reset enable bit set to 1, Watchdog reset will trigger the soft reset.

10.3.4 Software Initiated Reset

The Intel XScale® core or external PCI bus master can reset specific functions in the IXP2800 Network Processor by writing to the IXP_RESET0 and IXP_RESET1 registers. All the individual micro-engines and specific units can be reset individually in this fashion.

Software reset initiated by “Reset All” bit in IXP_RESET0 register behaves almost the same as hardware resets in the sense that PLL and rest of the core gets reset. The only difference between soft reset and hard reset is that a 512 cycle counter is added at the output of “RESET_ALL” bit going to PLL unit for chip reset generation. PCI unit in the meantime detects the bus idle condition and generates local reset. This local reset is removed once chip reset is generated and chip reset reset then takes over the reset function of PCI unit.

Both hardware and software resets (software reset after 512 cycles delay) combined generate PLL_RST for the PLL logic. During the assertion of PLL_RST, PLL block remains in the bypass mode and passes the incoming clock directly to the core logic. At this time everyone inside the core gets the same basic clock. The Clock Control Register is reset to “0x0FFF_FFFF” using the same signal.

Once PLL_RST signal goes away, PLL starts generating divide_by_2 clock for MicroEngines, divide_by_4 clock for the Intel XScale® core and divide_by_16 clock for the rest of the chip (not using divide_by_4 clock) after inserting 16–32 idle clocks. Once clock control CSR is written by software, PLL block detects it by finding change in value of this register.



Once in operation, if watchdog timer expires with watchdog timer enable bit WDE from Timer Watchdog Enable Register set, reset pulse from the watchdog timer logic goes to PLL unit after passing through a counter to guarantee minimum assertion time which in turn resets the IXP_RESETEn registers that causes entire chip to be reset.

Figure 134 explains the reset generation for PLL logic and for the rest of the core. CORE_RST is used inside the IXP2800 to reset everything. PLL_RST can be disabled

10.3.5 Reset Removal operation based on CFG_PROM_BOOT

Reset removal operation based on the CFG_PROM_BOOT strap option (BOOT_PROM) can be divided into two parts:

1. When CFG_PROM_BOOT is “1” (BOOT_PROM is present).
2. When CFG_PROM_BOOT is “0” (BOOT_PROM is not present)

10.3.5.1 When CFG_PROM_BOOT is 1 (BOOT_PROM is Present)

After CORE_RST is de-asserted, reset from the Intel XScale® core, SHAC and CMDARB is removed. Once the Intel XScale® core reset is removed, the Intel XScale® core starts initializing the chip. The Intel XScale® core writes “clock control CSR” to define the operating frequencies of different units. The Intel XScale® core writes IXP_RESETE0[21] to allow PCI logic to start accepting transactions on the PCI bus as part of initialization process.

10.3.5.2 When CFG_PROM_BOOT is 0 (BOOT_PROM is Not Present)

After CORE_RST is de-asserted, IXP_RESETE0[21] is set allowing PCI unit to start accepting transactions on the PCI bus. In this mode, the Intel XScale® core is kept in reset. Reset from DRAM logic is removed by the PCI host by writing “0” to specific bits in the IXP_RESETE0 register. C.

10.3.6 Strap Pins

The IXP2800 Strap pins for reset and initialization operation are described in Table 151.

Table 151. IXP2800 Network Processor Strap Pins

Signal	Name	Description
CFG_RST_DIR	RST_DIR	PCI_RST direction pin: (Also called PCI_HOST) Need to be a dedicated pin. 1: IXP is the host supporting central function. PCI_RST# is output 0: IXP is not central function. PCI_RST# is input This pin is Stored at XSC[31] (XScale_Control Register) at the trailing edge of reset.
CFG_PROM_BOOT	GPIO[0]	PCI PROM BOOT Pin: 1: IXP will boot from PROM: Whether Intel XScale® core will configure the system or not will be defined by CFG_PCI_BOOT_HOST strap option. 0: IXP will not boot from PROM. So after host has downloaded image of boot code into DRAM, Intel XScale® core will boot from DRAM address "0". This pin is Stored at XSC[29] (XScale_Control Register) at the trailing edge of reset.
CFG_PCI_BOOT_HOST	GPIO[1]	PCI BOOT HOST pin 1: IXP2800 Network Processor will configure the PCI system 0: IXP2800 Network Processor will not configure the PCI system This pin is Stored at XSC[28] (XScale_Control Register) at the trailing edge of reset.
CFG_PCI_ARB	GPIO[2]	PCI Arbiter Pin 1: IXP2800 Network Processor is the arbiter on the PCI bus 0: IXP2800 Network Processor is not the arbiter on the PCI bus
PLL_MULT[5:0]	SP_AD[5:0]	PLL Multiplier Valid values are 010000-110000 for multiplier range of 16 to 48. Other values will result in undefined behavior by PLL.
RESET_OUT_STRAP	SP_AD[7]	When "1": nRESET_OUT is removed after PLL locks When "0": nRESET_OUT is removed by software using bit IXP_RESET0[17]
CFG_PCI_SWIN[1:0]	GPIO[6:5]	SRAM Bar Window 11: SRAM BAR size of 256 MByte 10: SRAM BAR size of 128 MByte 01: SRAM BAR size of 64 MByte 00: SRAM BAR size of 32 MByte
CFG_PCI_DWIN[1:0]	GPIO[4:3]	DRAM BAR Window 11: DRAM BAR size of 1024 MByte 10: DRAM BAR size of 512 MByte 01: DRAM BAR size of 256 MByte 00: DRAM BAR size of 128 MByte
CFG_MSF_FREQ_SEL	SP_AD[6]	Select source of MSF Tx Clock 0—TCLK_Ref input pin 1— Internally generated clock



Table 152 lists the supported Strap combinations of CFG_PROM_BOOT, CFG_RST_DIR, and CFG_PCI_BOOT_HIST.

Table 152. Supported Strap Combinations

{CFG_PROM_BOOT, CFG_RST_DIR, CFG_PCI_BOOT_HOST}	Result
000	ALLOWED
001	ALLOWED
010	NOT ALLOWED
011	NOT ALLOWED
100	ALLOWED
101	ALLOWED
110	ALLOWED
111	ALLOWED

One more restriction in PCI unit is that if IXP2800 Network Processor is PCI_HOST or PCI_ARBITER, it should also be PCI_CENTRAL_FUNCTION.

10.3.7 Powerup Reset Sequence

When the system is powered up, bypass clock is sent to all the units as the chip begins to power up. It will merely be used to allow a gradual power up and to begin clocking state elements to remove possible circuit contention. When PLL gets locked after nRESET is de-asserted, it will start generating divide_by_16 clocks for all the units. Reset from IXP_RESET register is also removed at the same time. When software updates the clock count register, clocks are again stopped for 32 cycles and then start again.

The reset sequence described above is the same in the case when reset happens through the PCI_RST# signal and CFG_RST_DIR is asserted.

Once in operation, if watchdog timer expires with watchdog timer enable bit (bit [0] in Timer Watchdog Enable Register ON, a reset pulse from the watchdog timer logic resets the IXP_RESETn registers and in turn causes entire chip to be reset.

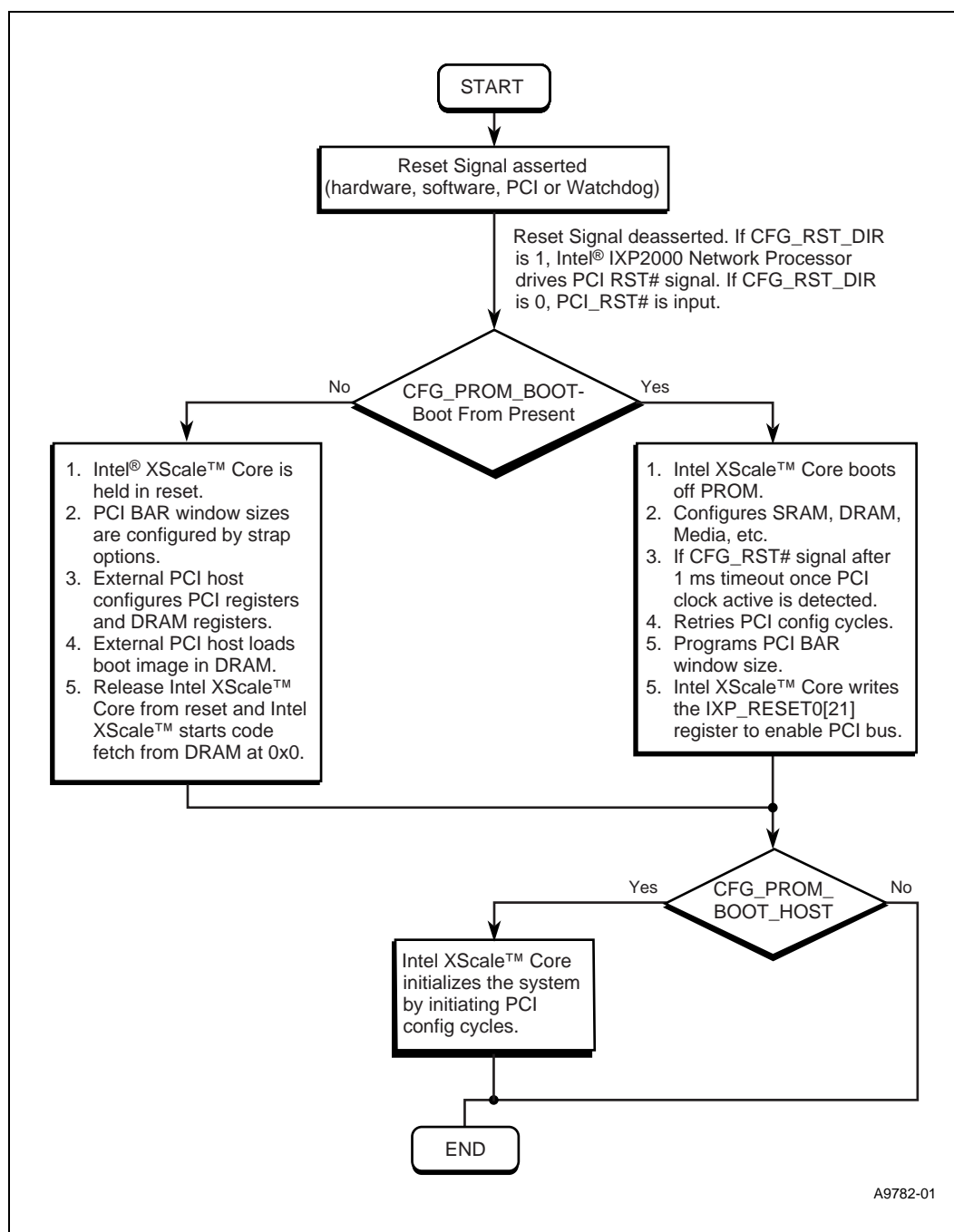
10.4 Boot Mode

The IXP2800 can boot in following two modes:

- Flash ROM
- PCI Host Download

Figure 135 shows the IXP2800 Network Processor Boot process.

Figure 135. Boot Process





10.4.1 Flash ROM

At power up, if FLASH_ROM is present, strap pin CFG_PROM_BOOT should be sampled “1” (should be pulled up). Therefore after reset being removed by the PLL logic from the IXP_RESET0 register, the Intel XScale® core reset is automatically removed. “Flash Alias Disable” (bit [8] of Misc Control Register) information is used by the Intel XScale® gasket to decide where to forward address “0” from the Intel XScale® core when the Intel XScale® core wakes up and starts accessing the code from address 0. In this mode, since “flash alias disable: bit is reset to “0”, the Intel XScale® gasket will convert access to address “0” to PROM access from address “0” using the CAP command. Based on the code residing inside PROM, the Intel XScale® core starts removing reset from SRAM, PCI, DRAM, MicroEngines etc. by writing “0” in their corresponding bit location of IXP_RESETh register and then initializing their configuration registers.

Boot code in PROM can change flash alias disable bit to "1" anytime to map DRAM at address zero and therefore block further accesses to PROM at address "0". This change should be done before putting any data in DRAM at address “0”.

The Intel XScale® core also sets different BARs inside PCI unit to define memory requirements for different windows.

The Intel XScale® core behavior as a host is controlled by CFG_PCI_BOOT_HOST strap option. If CFG_PCI_BOOT_HOST is sampled asserted in the de-asserting edge of reset, the Intel XScale® core will behave as boot host and configure the PCI system.

10.4.2 PCI Host Download

At power up, if FLASH_ROM is not present, strap pin CFG_PROM_BOOT should be sampled “0” (should be pulled down). In this mode CFG_RST_DIR pin should be “0” at power up signaling PCI_RST# pin is an input that behaves as global chip reset.

1. Even after reset is removed by the PLL logic from IXP_RESET0 register (after PCI_RST# reset is de-asserted), the Intel XScale® core reset is not removed.
2. PCI Reset through IXP_RESET0 [16] is removed automatically after being set and reset being removed.
3. IXP_RESET0[21] is set after PCI_RST# has been removed and PLL_LOCK is sampled asserted.
4. Once IXP_RESET0[21] is set, PCI unit starts responding to transactions.
5. PCI Host first configures CSR, SRAM and DRAM base address registers after reading size requirements for these BARs. The size for CSR, SRAM and DRAM is defined by the use of Strap pins. Pre-fetchability for the window is defined by bit [3] of the respective BAR registers therefore when host reads these registers, bit [3] is returned as “0” for CSR, SRAM and DRAM defining CSRs and also if SRAM and DRAM are to be non-prefetchable. “Type” Bits [2:0] are always Read-Only and return the value of “0x0” when read for CSR, SRAM and DRAM BAR registers.
6. PCI Host also programs “Clock Control CSR”, for PLL unit to generate proper clocks for SRAM, DRAM and other units.

Once these base address registers have been programmed, PCI host programs DRAM channels by initializing SDRAM_CSR, SDRAM_MEMCTL0, SDRAM_MEMCTL1 and SDRAM_MEMINIT registers. Once these registers have been programmed, PCI host writes the BOOT Code in DRAM

starting at DRAM address “0”. PCI Host can also program other registers if required. Once the boot code is written in DRAM, PCI host writes “1” at bit [8] of Misc_Control register called “Flash Alias Disable” (Reset value “0”). Alias Disable bit can be wired to the Intel XScale® gasket directly so that gasket knows how to transform address 0 from the Intel XScale® core. After writing “1” at “Flash Alias Disable” bit, host removes reset from the Intel XScale® core by writing “0” in bit [0] of IXP_RESET0 register. The Intel XScale® core starts booting from address 0, which is now directed by the gasket to DRAM.

10.5 Initialization

Boot Sequence task must be performed by the IXP2800 Network Processor after reset for proper processor function. The boot sequence tasks configure the IXP2800 Network Processor resources to a determined state by writing predetermined values to certain registers. Some register settings are determined by the components selected, such as SDRAM, SRAM, and BootROM. Other register settings are determined by the desired processor performance and system configuration.

The resources that must be configured after reset are the Phase-Locked Loop (PLL), PROM interface, the SRAM controller, the SDRAM controller, and the Memory Management Unit (MMU). There are other resources that if used during the boot sequence must be configured at this time. They are the UART and the PCI Interface. For a more detailed description of the registers and their settings, please refer to the appropriate sections in the *IXP2400/IXP2800 Network Processor Programmers Reference Manual*.

The configuration tasks must be performed in the following sequence.

1. Configure PLL. Since PLL output frequency is determined using the configuration pins (SP_AD[5:0]), these pins should be pulled up or pulled down to define the operating frequency of PLL. These strap options are stored in Strap_Options_Register as defined in the *IXP2400/IXP2800 Network Processor Programmers Reference Manual*.

2. Configure Clock Switching: Details will be added later.

3. Configure XPI Interface to access PROM if CFG_PROM_BOOT is set:

Following registers should be programmed.

- SP_CCR: To configure the clocks for the slow port. Initially these clocks start at some default value which may not be optimal.
- SP_WTC: This register should be programmed for PROM interface to define proper write timing
- SP_RTC: This register should be programmed for PROM interface to define proper read timing
- SP_FAC: To define the address size of flash memory device used.
- SP_FRM: To define the data width of the read back from the flash memory.

4. Configure clock logic:

To define the operating frequency of SRAM and DRAM interface, following registers that define the operation of stepping stone logic must be initialized:

- CCR: Clock Control CSR to define the frequency of SRAM and DRAM channels, MSF and APB



5. Release from Reset. After reset, units not coming out of reset automatically are brought out of reset by programming the following registers.

- IXP_RESET0
- IXP_RESET1

6. Configure SRAM.

Configure the SRAM controller. The registers that configure the SRAM controller are:

SRAM_Control: To define the configuration of SRAM Controller

SRAM_Parity_Status1: For parity control and recording of last faulty address

SRAM_Parity_Status2: Recording of source of request which generated parity error

7. Configure DRAM channels. Configure the in-use DRAM channels. This is done through a sequence of register writes.

- DU_CSRA_[2:0]
- DU_CSRB_[2:0]
- DU_INIT_[2:0]

8. Configure and Enable MMU (Optional). Configure the Memory Mapped Unit, Cache, and Buffer.

This is done by configuring the following register:

StrongARM Coprocessor 15—CONTROL_CP15

9. Configure PCI.

If CFG_PROM_BOOT is not set, loading of boot image by the PCI host is required into DRAM. For this to happen, de-asserting edge of reset should set these registers to their required value.

IXP_RESET0: IXP_RESET0[21] should be set to “1”.

- DRAM_BASE_ADDR_MASK
- PCI_DRAM_BAR: Strap pins define the window size
- PCI_SRAM_BAR: Strap pins define the window size
- PCI_CSR_BAR: Strap pins define the window size

After boot image is loaded into DRAM, “Flash_Alias_Disable” bit in Misc Control register from IXP_CHASSIS should be set to “1” so that DRAM appears at address 0.

If CFG_PROM_BOOT is set, configure the following four registers:

- PCI_MEM_BAR
- PCI_IO_BAR
- PCI_DRAM_BAR
- PCI_CMD_STAT
- IXP_RESET0[21]

In this mode, code jumps to normal flash location and then disables the "map flash to zero" feature. If CFG_PCI_BOOT_HOST is not true, then CFG_RST_DIR will program the IXP2800 Network Processor PCI interface based on its memory requirements. If CFG_PCI_BOOT_HOST is true, then the IXP2800 Network Processor will program its PCI interface.



10. Configure Serial Port.

If serial interface is required, the following registers must be configured.

- UART_DLRH
- UART_DLRL
- UART_IER
- UART_FCR
- UART_LCR

