



Intel[®] IXP2400 Network Processor

Hardware Reference Manual

November 2003



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.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The IXP2400 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, 2003

BunnyPeople, CablePort, Celeron, Chips, Dialogic, DM3, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel Centrino, Intel Centrino logo, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Xeon, Intel XScale, IPLink, Itanium, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, RemoteExpress, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside., The Journey Inside, TokenExpress, VoiceBrick, VTune, and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Contents

| | | |
|---------|---|----|
| 1 | Introduction..... | 25 |
| 1.1 | About this Document..... | 25 |
| 1.2 | Related Documentation..... | 25 |
| 1.3 | Conventions..... | 25 |
| 1.3.1 | Data Terminology..... | 26 |
| 1.3.2 | Definitions..... | 26 |
| 2 | Hardware Overview..... | 27 |
| 2.1 | Overview..... | 27 |
| 2.2 | Intel® XScale® Core..... | 29 |
| 2.2.1 | ARM Compatibility..... | 29 |
| 2.2.2 | Features..... | 30 |
| 2.2.2.1 | Multiply/Accumulate (MAC)..... | 30 |
| 2.2.2.2 | Memory Management..... | 30 |
| 2.2.2.3 | Instruction Cache..... | 30 |
| 2.2.2.4 | Branch Target Buffer..... | 30 |
| 2.2.2.5 | Data Cache..... | 30 |
| 2.2.2.6 | Interrupt Controller..... | 31 |
| 2.2.2.7 | Address Map..... | 31 |
| 2.3 | Microengine..... | 33 |
| 2.4 | SRAM..... | 35 |
| 2.5 | DRAM..... | 35 |
| 2.5.1 | Feature List..... | 36 |
| 2.6 | Media and Switch Fabric (MSF) Interface..... | 36 |
| 2.6.1 | Reference Documents..... | 37 |
| 2.7 | PCI Controller..... | 37 |
| 2.7.1 | Target Access..... | 38 |
| 2.7.2 | Master Access..... | 38 |
| 2.7.3 | DMA Channels..... | 38 |
| 2.7.3.1 | DMA Descriptor..... | 39 |
| 2.7.3.2 | DMA Channel Operation..... | 40 |
| 2.7.3.3 | DMA Channel End Operation..... | 41 |
| 2.7.3.4 | Adding Descriptor to Unterminated Chain..... | 41 |
| 2.7.4 | Mailbox and Message Registers..... | 41 |
| 2.7.5 | PCI Arbiter..... | 42 |
| 2.8 | Scratchpad Memory..... | 42 |
| 2.9 | Hash Unit..... | 43 |
| 2.10 | Control and Status Register Access Proxy..... | 44 |
| 2.11 | Intel® XScale® Core Peripherals..... | 45 |
| 2.11.1 | Interrupt Controller..... | 45 |
| 2.11.2 | Timers..... | 45 |
| 2.11.3 | GPIO..... | 45 |
| 2.11.4 | UART..... | 46 |
| 2.11.5 | SlowPort..... | 46 |
| 3 | Intel® XScale® Core..... | 49 |
| 3.1 | Introduction..... | 49 |

| | | |
|---------|---|----|
| 3.2 | Features | 50 |
| 3.2.1 | Multiply/Accumulate (MAC) | 50 |
| 3.2.2 | Memory Management | 50 |
| 3.2.3 | Instruction Cache | 51 |
| 3.2.4 | Branch Target Buffer | 51 |
| 3.2.5 | Data Cache | 51 |
| 3.2.6 | Performance Monitoring | 51 |
| 3.2.7 | Power Management | 51 |
| 3.2.8 | Debug | 51 |
| 3.2.9 | JTAG | 52 |
| 3.3 | Memory Management | 52 |
| 3.3.1 | Architecture Model | 52 |
| 3.3.1.1 | Version 4 vs. Version 5 | 52 |
| 3.3.1.2 | Memory Attributes | 53 |
| 3.3.2 | Exceptions | 55 |
| 3.3.3 | Interaction of the MMU, Instruction Cache, and Data Cache | 55 |
| 3.3.4 | Control | 55 |
| 3.3.4.1 | Invalidate (Flush) Operation | 55 |
| 3.3.4.2 | Enabling/Disabling | 55 |
| 3.3.4.3 | Locking Entries | 56 |
| 3.3.4.4 | Round-Robin Replacement Algorithm | 58 |
| 3.4 | Instruction Cache | 59 |
| 3.4.1 | Instruction Cache Operation | 60 |
| 3.4.1.1 | Operation When Instruction Cache is Enabled | 60 |
| 3.4.1.2 | Operation When The Instruction Cache Is Disabled | 60 |
| 3.4.1.3 | Fetch Policy | 60 |
| 3.4.1.4 | Round-Robin Replacement Algorithm | 61 |
| 3.4.1.5 | Parity Protection | 61 |
| 3.4.1.6 | Instruction Fetch Latency | 61 |
| 3.4.1.7 | Instruction Cache Coherency | 62 |
| 3.4.2 | Instruction Cache Control | 62 |
| 3.4.2.1 | Instruction Cache State at Reset | 62 |
| 3.4.2.2 | Enabling/Disabling | 62 |
| 3.4.2.3 | Invalidating the Instruction Cache | 63 |
| 3.4.2.4 | Locking Instructions in the Instruction Cache | 63 |
| 3.4.2.5 | Unlocking Instructions in the Instruction Cache | 65 |
| 3.5 | Branch Target Buffer | 65 |
| 3.5.1 | Branch Target Buffer (BTB) Operation | 65 |
| 3.5.1.1 | Reset | 66 |
| 3.5.2 | Update Policy | 66 |
| 3.5.3 | BTB Control | 66 |
| 3.5.3.1 | Disabling/Enabling | 66 |
| 3.5.3.2 | Invalidation | 67 |
| 3.6 | Data Cache | 67 |
| 3.6.1 | Overviews | 67 |
| 3.6.1.1 | Data Cache Overview | 67 |
| 3.6.1.2 | Mini-Data Cache Overview | 68 |
| 3.6.1.3 | Write Buffer and Fill Buffer Overview | 69 |
| 3.6.2 | Data Cache and Mini-Data Cache Operation | 70 |
| 3.6.2.1 | Operation When Caching is Enabled | 70 |
| 3.6.2.2 | Operation When Data Caching is Disabled | 70 |
| 3.6.2.3 | Cache Policies | 70 |

| | | |
|----------|---|-----|
| 3.6.2.4 | Round-Robin Replacement Algorithm | 72 |
| 3.6.2.5 | Parity Protection | 72 |
| 3.6.2.6 | Atomic Accesses | 72 |
| 3.6.3 | Data Cache and Mini-Data Cache Control | 73 |
| 3.6.3.1 | Data Memory State After Reset | 73 |
| 3.6.3.2 | Enabling/Disabling | 73 |
| 3.6.3.3 | Invalidate and Clean Operations | 73 |
| 3.6.4 | Re-configuring the Data Cache as Data RAM | 75 |
| 3.6.5 | Write Buffer/Fill Buffer Operation and Control | 76 |
| 3.7 | Configuration | 76 |
| 3.8 | Performance Monitoring | 77 |
| 3.8.1 | Performance Monitoring Events | 78 |
| 3.8.1.1 | Instruction Cache Efficiency Mode | 79 |
| 3.8.1.2 | Data Cache Efficiency Mode | 79 |
| 3.8.1.3 | Instruction Fetch Latency Mode | 79 |
| 3.8.1.4 | Data/Bus Request Buffer Full Mode | 80 |
| 3.8.1.5 | Stall/Writeback Statistics | 80 |
| 3.8.1.6 | Instruction TLB Efficiency Mode | 81 |
| 3.8.1.7 | Data TLB Efficiency Mode | 81 |
| 3.8.2 | Multiple Performance Monitoring Run Statistics | 81 |
| 3.9 | Performance Considerations | 81 |
| 3.9.1 | Interrupt Latency | 82 |
| 3.9.2 | Branch Prediction | 82 |
| 3.9.3 | Addressing Modes | 83 |
| 3.9.4 | Instruction Latencies | 83 |
| 3.9.4.1 | Performance Terms | 83 |
| 3.9.4.2 | Branch Instruction Timings | 84 |
| 3.9.4.3 | Data Processing Instruction Timings | 85 |
| 3.9.4.4 | Multiply Instruction Timings | 86 |
| 3.9.4.5 | Saturated Arithmetic Instructions | 87 |
| 3.9.4.6 | Status Register Access Instructions | 88 |
| 3.9.4.7 | Load/Store Instructions | 88 |
| 3.9.4.8 | Semaphore Instructions | 88 |
| 3.9.4.9 | Coprocessor Instructions | 89 |
| 3.9.4.10 | Miscellaneous Instruction Timing | 89 |
| 3.9.4.11 | Thumb Instructions | 89 |
| 3.10 | IXP2400 Network Processor Endianness | 89 |
| 3.10.1 | Read and Write Transactions Initiated by the Intel® XScale® Core | 91 |
| 3.10.1.1 | Reads Initiated by Intel® XScale® Core | 91 |
| 3.10.1.2 | The Intel® XScale® Core Writing to the IXP2400 | 93 |
| 3.11 | Intel® XScale® Gasket Unit | 95 |
| 3.11.1 | Overview | 95 |
| 3.11.2 | Intel® XScale® Gasket Functional Description | 97 |
| 3.11.2.1 | Core Memory Bus to Command Push/Pull Conversion | 97 |
| 3.11.3 | CAM Operation | 97 |
| 3.11.4 | Atomic Operations | 98 |
| 3.11.4.1 | Intel® XScale® Access to SRAM Q-Array | 99 |
| 3.11.5 | I/O Transaction | 99 |
| 3.11.6 | Hash Access | 100 |
| 3.11.7 | Gasket Local CSR | 100 |
| 3.11.8 | Interrupt | 101 |

| | | |
|----------|--|-----|
| 3.12 | Intel® XScale® Core Peripheral Interface..... | 103 |
| 3.12.1 | XPI Overview | 104 |
| 3.12.1.1 | Data Transfers..... | 105 |
| 3.12.1.2 | Data Alignment..... | 107 |
| 3.12.1.3 | Address Spaces for XPI Internal Devices | 107 |
| 3.12.2 | UART Overview | 108 |
| 3.12.2.1 | Baud Rate Generator | 110 |
| 3.12.2.2 | UART FIFO Operation..... | 111 |
| 3.12.3 | General Purpose I/O (GPIO)..... | 112 |
| 3.12.4 | Timers | 114 |
| 3.12.4.1 | Timer Operation | 115 |
| 3.12.5 | SlowPort Unit | 118 |
| 3.12.5.1 | PROM Device Support..... | 118 |
| 3.12.5.2 | µP interface support for the Framer | 119 |
| 3.12.5.3 | SlowPort Unit Interfaces..... | 120 |
| 3.12.5.4 | Address Space | 121 |
| 3.12.5.5 | SlowPort Interfacing Topology | 121 |
| 3.12.5.6 | SlowPort 8-bit Device Bus Protocols..... | 123 |
| 3.12.5.7 | SONET/SDH Microprocessor Access Support..... | 127 |
| 3.12.6 | PROM Device Timing Information for IXP2400 A0/A1 | 150 |
| 3.12.7 | PROM Device Timing Information (IXP2400 B0) | 153 |
| 4 | Microengines..... | 155 |
| 4.1 | Overview | 155 |
| 4.1.1 | Control Store | 156 |
| 4.1.2 | Contexts | 157 |
| 4.1.3 | Datapath Registers | 159 |
| 4.1.3.1 | General-Purpose Registers (GPRs)..... | 159 |
| 4.1.3.2 | Transfer Registers..... | 159 |
| 4.1.3.3 | Next Neighbor Registers | 160 |
| 4.1.3.4 | Local Memory (LM) | 161 |
| 4.1.4 | Addressing Modes | 163 |
| 4.1.4.1 | Context-Relative Addressing Mode..... | 163 |
| 4.1.4.2 | Absolute Addressing Mode | 164 |
| 4.1.4.3 | Indexed Addressing Mode..... | 164 |
| 4.2 | Local CSRs | 165 |
| 4.3 | Execution Datapath..... | 166 |
| 4.3.1 | Byte Align | 166 |
| 4.3.2 | CAM | 168 |
| 4.4 | CRC Unit..... | 171 |
| 4.5 | Event Signals | 172 |
| 4.5.1 | Microengine Endianness | 172 |
| 4.5.1.1 | Read from RBUF (64-bits)..... | 173 |
| 4.5.1.2 | Write to TBUF..... | 173 |
| 4.5.1.3 | Read/Write from/to SRAM..... | 174 |
| 4.5.1.4 | Read/Write from/to DRAM..... | 174 |
| 4.5.1.5 | Read/Write from/to SHAC and Other CSRs..... | 174 |
| 4.5.1.6 | Write to Hash Unit | 174 |
| 4.6 | Summary of the Differences Between MEv2 and MEv1 | 175 |
| 4.6.1 | General Purpose Registers and Transfer Registers | 175 |
| 4.6.2 | Next Neighbor Registers | 176 |
| 4.6.3 | Local Memory..... | 176 |

| | | |
|--------|--|-----|
| 4.6.4 | Contexts | 176 |
| 4.6.5 | Larger Microstore | 176 |
| 4.6.6 | CAM | 176 |
| 4.6.7 | Event Signals | 176 |
| 4.6.8 | Larger Immediate Field | 176 |
| 4.6.9 | Fast Write—Wider Data Field, Access to More Registers..... | 176 |
| 4.6.10 | Local CSR Instruction Uses Absolute/Relative Register Addressing ... | 177 |
| 4.6.11 | Timestamp..... | 177 |
| 4.6.12 | Future_Count Event | 177 |
| 4.6.13 | Multiply Hardware Support..... | 177 |
| 4.6.14 | No +IFsign ALU Opcode | 177 |
| 4.6.15 | New Find First Bit Instruction | 177 |
| 4.6.16 | ctx_arb[bpt]..... | 177 |
| 4.6.17 | Pseudo-Random Number CSR | 177 |
| 4.6.18 | Local CSR Access for External Agents | 178 |
| 4.6.19 | New Branch Types to Support Signed and Unsigned Numbers | 178 |
| 4.6.20 | Branch Guess Taken Has No Effect in Hardware | 178 |
| 4.6.21 | No Shift with Add or XOR..... | 178 |
| 4.6.22 | No A + 4 B ALU Opcode | 178 |
| 4.6.23 | Local_CSR_Rd Returns 32 Bits | 178 |
| 4.6.24 | Profile Count Local CSR | 178 |
| 4.6.25 | CSR_CTX_Pointer | 179 |
| 5 | DDR SDRAM Controller..... | 181 |
| 5.1 | Overview | 181 |
| 5.2 | Feature List | 182 |
| 5.3 | Configurations | 182 |
| 5.4 | Initialization..... | 183 |
| 5.5 | Supported Frequencies | 183 |
| 5.6 | Interleaving..... | 184 |
| 5.6.1 | Interleaving across Banks | 184 |
| 5.7 | Error Correction..... | 184 |
| 5.8 | Supported Requests..... | 185 |
| 5.8.1 | Reads and Writes..... | 185 |
| 5.8.2 | Register Access | 185 |
| 5.9 | Microengine Signals..... | 186 |
| 5.10 | Read/Write Ordering Requirements | 187 |
| 5.11 | Design Overview | 187 |
| 5.11.1 | Read Requests..... | 189 |
| 5.11.2 | Write Requests..... | 189 |
| 5.11.3 | Request Scheduling Algorithm | 190 |
| 5.11.4 | DDR Pin FSM..... | 191 |
| 5.12 | Register Descriptions | 191 |
| 5.12.1 | Register Map | 191 |
| 6 | SRAM Interface..... | 193 |
| 6.1 | Overview | 193 |
| 6.2 | SRAM Interface Configurations..... | 194 |
| 6.3 | SRAM Interface Configurations..... | 196 |
| 6.3.1 | Internal Interface | 196 |

| | | | |
|-----|---------|--|-----|
| | 6.3.2 | Number of Channels | 196 |
| | 6.3.3 | Coprocessor and/or SRAMs Attached to a Channel | 196 |
| 6.4 | | SRAM Controller Configurations | 197 |
| 6.5 | | Command Overview | 199 |
| | 6.5.1 | Basic Read/Write Commands | 199 |
| | 6.5.2 | Atomic Operations | 199 |
| | 6.5.3 | Queue Data Structure Commands | 200 |
| | 6.5.3.1 | Read_Q_Descriptor Commands | 205 |
| | 6.5.3.2 | Write_Q_Descriptor Commands | 207 |
| | 6.5.3.3 | ENQ and DEQ Commands | 207 |
| | 6.5.4 | Ring Data Structure Commands | 209 |
| | 6.5.5 | Journaling Commands | 209 |
| | 6.5.6 | CSR Accesses | 210 |
| 6.6 | | Parity | 210 |
| 6.7 | | Address Map | 211 |
| 6.8 | | Reference Ordering | 211 |
| | 6.8.1 | Reference Order Tables | 212 |
| | 6.8.2 | Microcode Restrictions to Maintain Ordering | 213 |
| 6.9 | | Coprocessor Mode | 213 |
| 7 | | SHaC Unit | 217 |
| | 7.1 | Prerequisite Reading | 217 |
| | 7.2 | Introduction | 217 |
| | 7.3 | Unit Overview | 217 |
| | 7.4 | High Level Block Diagrams | 218 |
| | 7.4.1 | Full Chip Diagram | 218 |
| | 7.4.2 | SHaC Unit Block Diagram | 219 |
| | 7.5 | Unit Design Details | 219 |
| | 7.5.1 | Scratchpad | 219 |
| | 7.5.1.1 | Scratchpad Description | 219 |
| | 7.5.1.2 | Scratchpad Interface | 220 |
| | 7.5.1.3 | Scratchpad Command Overview | 221 |
| | 7.5.2 | Hash Unit | 229 |
| | 7.5.2.1 | Hash Unit Description | 229 |
| | 7.5.2.2 | Hash Unit Block Diagram | 230 |
| | 7.5.2.3 | Hash Operation | 231 |
| | 7.5.2.4 | Hash Algorithm | 234 |
| 8 | | Media and Switch Fabric Interface | 237 |
| | 8.1 | Overview | 237 |
| | 8.2 | Reference Documents | 238 |
| | 8.3 | Media Bus Interface | 238 |
| | 8.3.1 | UTOPIA | 239 |
| | 8.3.1.1 | Single-PHY (SPHY) Mode | 240 |
| | 8.3.1.2 | Multi-PHY (MPHY) Mode | 243 |
| | 8.3.2 | POS-PHY | 244 |
| | 8.3.2.1 | Single PHY (SPHY) Mode | 245 |
| | 8.3.2.2 | Multi-PHY (MPHY) Mode | 247 |
| | 8.3.2.3 | SPI3 Slave Mode | 248 |
| | 8.3.2.4 | Transmit Slave Operation | 250 |
| | 8.3.2.5 | Receive Slave Operation | 250 |

| | | |
|---------|--|-----|
| 8.3.3 | CSIX | 251 |
| 8.4 | MSF Mode Signal Usage | 252 |
| 8.5 | Receive | 252 |
| 8.5.1 | Receive Pins and Protocol Logic..... | 253 |
| 8.5.1.1 | UTOPIA SPHY | 253 |
| 8.5.1.2 | UTOPIA MPHY..... | 253 |
| 8.5.1.3 | POS-PHY SPHY | 253 |
| 8.5.1.4 | POS-PHY MPHY | 254 |
| 8.5.1.5 | CSIX | 254 |
| 8.5.2 | RBUF..... | 254 |
| 8.5.3 | Receive Status Word..... | 256 |
| 8.5.3.1 | UTOPIA Mode | 256 |
| 8.5.3.2 | POS-PHY Mode | 258 |
| 8.5.3.3 | CSIX Mode | 261 |
| 8.5.4 | Full Element List..... | 263 |
| 8.5.5 | Rx_Thread_Freelists | 264 |
| 8.5.5.1 | UTOPIA and POS-PHY SPHY Modes | 265 |
| 8.5.5.2 | UTOPIA/POS-PHY MPHY-4 Mode | 265 |
| 8.5.5.3 | UTOPIA/POS-PHY MPHY-32 Mode | 265 |
| 8.5.5.4 | UTOPIA/POS-PHY 16 Bit MPHY + 8 Bit SPHY + 8 Bit SPHY..... | 266 |
| 8.5.5.5 | UTOPIA/POS-PHY 16 Bit MPHY + 16 Bit SPHY | 266 |
| 8.5.5.6 | CSIX Mode | 266 |
| 8.5.6 | Rx_Thread_Freelist Timeout..... | 266 |
| 8.5.7 | Receive Operation Summary | 267 |
| 8.5.8 | Receive Flow Control Status | 268 |
| 8.5.8.1 | UTOPIA and POS-PHY Mode..... | 269 |
| 8.5.8.2 | CSIX Mode | 269 |
| 8.5.9 | Parity | 269 |
| 8.5.9.1 | UTOPIA Mode | 269 |
| 8.5.9.2 | POS-PHY Mode | 269 |
| 8.5.9.3 | CSIX Mode | 269 |
| 8.5.10 | Error Cases | 270 |
| 8.6 | Transmit | 270 |
| 8.6.1 | Transmit Pins | 271 |
| 8.6.2 | TBUF and Transmit Control Word..... | 271 |
| 8.6.2.1 | UTOPIA/POS-PHY SPHY TBUF Partitioning..... | 274 |
| 8.6.2.2 | UTOPIA/POS-PHY MPHY-4 TBUF Partitioning..... | 275 |
| 8.6.2.3 | UTOPIA/POS-PHY MPHY-32 TBUF Partitioning..... | 275 |
| 8.6.2.4 | UTOPIA/POS-PHY 16 Bit MPHY + 8 Bit SPHY + 8 Bit SPHY..... | 276 |
| 8.6.2.5 | UTOPIA/POS-PHY 16 Bit MPHY + 16 Bit SPHY | 276 |
| 8.6.2.6 | CSIX TBUF Partitioning..... | 276 |
| 8.6.2.7 | UTOPIA Transmit Control Word Format | 277 |
| 8.6.2.8 | POS-PHY Transmit Control Word Format..... | 277 |
| 8.6.2.9 | CSIX Mode | 278 |
| 8.6.3 | Transmit Operation Summary | 279 |
| 8.6.3.1 | UTOPIA SPHY Mode | 280 |
| 8.6.3.2 | UTOPIA MPHY-4 Mode | 280 |
| 8.6.3.3 | UTOPIA MPHY-32 Mode | 281 |
| 8.6.3.4 | POS-PHY SPHY Mode | 281 |
| 8.6.3.5 | POS-PHY MPHY-4 Mode..... | 282 |

| | | | |
|-----|---------|---|-----|
| | 8.6.3.6 | POS-PHY MPHY-32 Mode..... | 282 |
| | 8.6.3.7 | CSIX Mode..... | 284 |
| | 8.6.3.8 | Transmit Summary..... | 285 |
| | 8.6.4 | Transmit Flow Control Status..... | 285 |
| | 8.6.5 | Parity..... | 286 |
| | 8.6.5.1 | UTOPIA Mode..... | 286 |
| | 8.6.5.2 | POS-PHY Mode..... | 286 |
| | 8.6.5.3 | CSIX Mode..... | 286 |
| | 8.6.6 | RBUF and TBUF Summary..... | 286 |
| 8.7 | | CBus Interface..... | 287 |
| | 8.7.1 | CBus Signals..... | 288 |
| | 8.7.1.1 | TXCSOF/TXCDATA/TXCPAR and RXCSOF/RXCDATA/RXCPAR..... | 288 |
| | 8.7.1.2 | TXCSRB and RXCSRB..... | 289 |
| | 8.7.1.3 | TXCFC and RXCFC..... | 289 |
| | 8.7.2 | Full Duplex Mode..... | 289 |
| | 8.7.2.1 | Link Level..... | 290 |
| | 8.7.2.2 | Buffering and Link Level Flow Control Latency..... | 292 |
| | 8.7.2.3 | Fabric Level..... | 293 |
| | 8.7.3 | Simplex Mode..... | 294 |
| | 8.7.3.1 | Link Level..... | 295 |
| | 8.7.3.2 | Fabric Level..... | 296 |
| 8.8 | | Interface to Command and Push and Pull Buses..... | 297 |
| | 8.8.1 | RBUF or CSR to ME SRAM Read Transfer Register..... | 297 |
| | 8.8.2 | ME SRAM Write Transfer Register to TBUF or CSR..... | 298 |
| | 8.8.3 | ME to MSF CSR Fast Write..... | 298 |
| | 8.8.4 | Transfer from RBUF to DRAM..... | 298 |
| | 8.8.5 | Transfer from DRAM to TBUF..... | 298 |
| 8.9 | | Registers..... | 299 |
| 9 | | PCI Unit..... | 301 |
| | 9.1 | Overview..... | 301 |
| | 9.2 | PCI Protocol Interface Block..... | 303 |
| | 9.2.1 | PCI Commands..... | 304 |
| | 9.2.2 | IXP2400 Network Processor Initialization..... | 305 |
| | 9.2.2.1 | Initialization by the Intel® XScale® Core..... | 306 |
| | 9.2.2.2 | Initialization by an External PCI Host..... | 306 |
| | 9.2.3 | PCI Type 0 Configuration Cycles..... | 306 |
| | 9.2.3.1 | Configuration Write..... | 307 |
| | 9.2.3.2 | Configuration Read..... | 307 |
| | 9.2.4 | PCI 64-Bit Bus Extension..... | 307 |
| | 9.2.5 | PCI Target Cycles..... | 307 |
| | 9.2.5.1 | PCI Accesses to CSR..... | 308 |
| | 9.2.5.2 | PCI Accesses to DRAM..... | 308 |
| | 9.2.5.3 | PCI Accesses to SRAM..... | 308 |
| | 9.2.5.4 | Target Write Accesses From PCI Bus..... | 308 |
| | 9.2.5.5 | Target Read Accesses From PCI Bus..... | 308 |
| | 9.2.6 | PCI Initiator Transactions..... | 309 |
| | 9.2.6.1 | PCI Request Operation..... | 309 |
| | 9.2.6.2 | PCI Commands..... | 309 |
| | 9.2.6.3 | Initiator Write Transactions..... | 310 |
| | 9.2.6.4 | Initiator Read Transactions..... | 310 |

| | | | |
|-----|----------|---|-----|
| | 9.2.6.5 | Initiator Latency Timer | 310 |
| | 9.2.6.6 | Special Cycle..... | 310 |
| | 9.2.7 | PCI Fast Back to Back Cycles..... | 310 |
| | 9.2.8 | PCI retry | 311 |
| | 9.2.9 | PCI Disconnect..... | 311 |
| | 9.2.10 | PCI Built In System Test | 311 |
| | 9.2.11 | PCI Central Functions | 311 |
| | 9.2.11.1 | PCI Interrupt Inputs | 312 |
| | 9.2.11.2 | PCI Reset Output | 312 |
| | 9.2.11.3 | PCI Internal Arbiter..... | 312 |
| 9.3 | | Slave Interface Block..... | 313 |
| | 9.3.1 | CSR Interface | 314 |
| | 9.3.2 | SRAM Interface | 314 |
| | 9.3.2.1 | SRAM Slave Writes..... | 314 |
| | 9.3.2.2 | SRAM Slave Reads..... | 315 |
| | 9.3.3 | DRAM Interface | 316 |
| | 9.3.3.1 | DRAM Slave Writes..... | 316 |
| | 9.3.3.2 | DRAM Slave Reads | 317 |
| | 9.3.4 | Mailbox and Doorbell Registers | 318 |
| | 9.3.5 | PCI Interrupt Pin..... | 321 |
| 9.4 | | Master Interface Block..... | 322 |
| | 9.4.1 | DMA Interface | 322 |
| | 9.4.1.1 | Allocation of the DMA Channels..... | 323 |
| | 9.4.1.2 | Special Registers for Microengine Channels | 323 |
| | 9.4.1.3 | DMA Descriptor | 323 |
| | 9.4.1.4 | DMA Channel Operation | 324 |
| | 9.4.1.5 | DMA Channel End Operation..... | 325 |
| | 9.4.1.6 | Adding Descriptor to an Unterminated Chain..... | 325 |
| | 9.4.1.7 | DRAM to PCI Transfer | 326 |
| | 9.4.1.8 | PCI to DRAM Transfer | 326 |
| | 9.4.2 | Push/Pull Command Bus Target Interface | 326 |
| | 9.4.2.1 | Command Bus Master Access to Local Configuration Registers..... | 327 |
| | 9.4.2.2 | Command Bus Master Access to Local Control and Status Registers | 327 |
| | 9.4.2.3 | Command Bus Master Direct Access to PCI Bus | 327 |
| 9.5 | | PCI Unit Error Behavior..... | 329 |
| | 9.5.1 | PCI Target Error Behavior..... | 329 |
| | 9.5.1.1 | Target Access Has an Address Parity Error..... | 329 |
| | 9.5.1.2 | Initiator Asserts PCI_PERR# in Response to One of Our Data Phases..... | 329 |
| | 9.5.1.3 | Discard Timer Expires on a Target Read..... | 329 |
| | 9.5.1.4 | Target Access to the PCI_CSR_BAR Space Has Illegal Byte Enables | 330 |
| | 9.5.1.5 | Target Write Access Receives Bad Parity PCI_PAR With the Data | 330 |
| | 9.5.1.6 | SRAM Responds With a Memory Error on One or More Data Phases on a Target Read..... | 330 |
| | 9.5.1.7 | DRAM Responds With a Memory Error on One or More Data Phase on a Target Read | 330 |
| | 9.5.2 | As a PCI Initiator During a DMA Transfer | 330 |
| | 9.5.2.1 | DMA Read From DRAM (Memory-to-PCI Transaction) | |

| | | |
|----------|--|-----|
| | Gets a Memory Error..... | 330 |
| 9.5.2.2 | DMA Read From SRAM (Descriptor Read) Gets a Memory Error..... | 331 |
| 9.5.2.3 | DMA From DRAM Transfer (Write to PCI) Receives PCI_PERR# on PCI Bus | 331 |
| 9.5.2.4 | DMA To DRAM (Read from PCI) Has Bad Data Parity..... | 331 |
| 9.5.2.5 | DMA Transfer Experiences a Master Abort (Time-Out) on PCI..... | 332 |
| 9.5.2.6 | DMA Transfer Receives a Target Abort Response During a Data Phase..... | 332 |
| 9.5.2.7 | DMA Descriptor Has a 0x0 Word Count (Not an Error)..... | 332 |
| 9.5.3 | As a PCI Initiator During a Direct Access from the Intel® XScale® Core or Microengine | 332 |
| 9.5.3.1 | Master Transfer Experiences a Master Abort (Time-Out) on PCI..... | 332 |
| 9.5.3.2 | Master Transfer Receives a Target Abort Response During a Data Phase..... | 332 |
| 9.5.3.3 | Master from the Intel® XScale® Core or Microengine Transfer (Write to PCI) Receives PCI_PERR# on PCI bus..... | 333 |
| 9.5.3.4 | Master Read From PCI (Read from PCI) Has Bad Data Parity..... | 333 |
| 9.5.3.5 | Master Transfer Receives PCI_SERR# from the PCI Bus... | 333 |
| 9.5.3.6 | Intel® XScale® Core Microengine Requests Direct Transfer When the PCI Bus is in Reset..... | 333 |
| 9.6 | PCI Data Byte Lane Alignment | 333 |
| 9.6.1 | Endian for Byte Enable | 336 |
| 9.7 | PCI Strap Pins Options | 339 |
| 10 | Clocks, Reset, and Initialization | 341 |
| 10.1 | Overview | 341 |
| 10.2 | Clocks | 341 |
| 10.2.1 | CSRs..... | 343 |
| 10.2.1.1 | Clock Control CSR (CCR)..... | 343 |
| 10.2.1.2 | MSF Clock Control CSR (MCCR) | 343 |
| 10.2.1.3 | Reset History CSR | 344 |
| 10.3 | Reset..... | 344 |
| 10.3.1 | Hardware Reset | 344 |
| 10.3.2 | PCI Initiated Reset | 345 |
| 10.3.3 | Watchdog Timer Initiated Reset..... | 345 |
| 10.3.4 | Software-Initiated Reset..... | 346 |
| 10.3.5 | Strap Pins..... | 347 |
| 10.3.6 | Power Up Reset Sequence..... | 348 |
| 10.3.7 | Power-Down Sequence | 350 |
| 10.4 | Reset Register | 350 |
| 10.5 | Boot Mode..... | 350 |
| 10.5.1 | Flash ROM | 352 |
| 10.5.2 | PCI Host Download..... | 352 |
| 10.6 | Reset Strategy for Different Sections in IXP2400 | 353 |
| 10.7 | Initialization | 356 |
| 11 | Performance Monitor Unit | 359 |
| 11.1 | Introduction | 359 |

| | | |
|----------|---|-----|
| 11.1.1 | Motivation for Performance Monitors | 359 |
| 11.1.2 | Motivation for Choosing CHAP Counters | 360 |
| 11.1.3 | Functional Overview of CHAP Counters | 360 |
| 11.1.4 | Basic Operation of Performance Monitor Unit..... | 362 |
| 11.1.5 | Definition of CHAP Terminology..... | 363 |
| 11.2 | Interface and CSR Description..... | 364 |
| 11.2.1 | APB Bus Peripheral..... | 365 |
| 11.2.2 | CAP Description | 365 |
| 11.2.2.1 | Selecting the Access Mode | 365 |
| 11.2.2.2 | PMU CSR..... | 365 |
| 11.2.2.3 | CAP Writes..... | 365 |
| 11.2.2.4 | CAP Reads..... | 365 |
| 11.2.3 | Configuration Registers..... | 365 |
| 11.3 | Performance Measurements | 366 |
| 11.3.1 | XScale®..... | 366 |
| 11.3.1.1 | DRAM Read Head of Queue Latency Histogram | 366 |
| 11.3.1.2 | SRAM Read Head of Queue Latency Histogram | 367 |
| 11.3.1.3 | Interrupts | 367 |
| 11.3.2 | Microengines | 367 |
| 11.3.2.1 | Command Fifo number of commands | 367 |
| 11.3.2.2 | Control Store Measures | 367 |
| 11.3.2.3 | Execution Unit Status | 367 |
| 11.3.2.4 | Command Fifo Head of Queue Wait Time Histogram (Latency)..... | 367 |
| 11.3.3 | SRAM | 368 |
| 11.3.3.1 | SRAM Commands..... | 368 |
| 11.3.3.2 | SRAM Bytes, Cycles Busy | 368 |
| 11.3.3.3 | Queue Depth Histogram..... | 368 |
| 11.3.4 | DDRAM | 368 |
| 11.3.4.1 | DRAM Commands | 368 |
| 11.3.4.2 | DRAM Bytes, Cycles Busy | 368 |
| 11.3.4.3 | Maskable by Read/Write, ME, PCI or XScale® | 368 |
| 11.3.5 | Chassis/Push-Pull | 369 |
| 11.3.5.1 | Command Bus utilization..... | 369 |
| 11.3.5.2 | Push and Pull Bus Utilization. | 369 |
| 11.3.6 | Hash | 369 |
| 11.3.6.1 | Number of Accesses by Command type | 369 |
| 11.3.6.2 | Latency of Histogram | 369 |
| 11.3.7 | Scratch | 369 |
| 11.3.7.1 | Number of Accesses by Command type | 369 |
| 11.3.7.2 | Number of bytes transfer..... | 369 |
| 11.3.7.3 | Latency of Histogram | 370 |
| 11.3.8 | PCI | 370 |
| 11.3.8.1 | Master Accesses | 370 |
| 11.3.8.2 | Slave Accesses | 370 |
| 11.3.8.3 | Master/Slave Read Byte Count | 370 |
| 11.3.8.4 | Master/Slave Write Byte Count | 370 |
| 11.3.8.5 | Burst Size Histogram..... | 370 |
| 11.3.9 | Media Interface..... | 371 |
| 11.3.9.1 | TBUF Occupancy Histogram..... | 371 |
| 11.3.9.2 | RBUF Occupancy Histogram | 371 |
| 11.3.9.3 | Packet/Cell/Frame Count on a Per Port Basis | 371 |

| | | | |
|------|-----------|--|-----|
| | 11.3.9.4 | Inter-Arrival Time for Packets on a Per Port Basis..... | 371 |
| | 11.3.9.5 | Burst Size Histogram..... | 371 |
| 11.4 | | Events Monitored in Hardware..... | 371 |
| | 11.4.1 | Queue Statistics Events..... | 372 |
| | 11.4.1.1 | Queue Latency..... | 372 |
| | 11.4.1.2 | Queue Utilization..... | 372 |
| | 11.4.2 | Count Events..... | 372 |
| | 11.4.2.1 | Hardware Block Execution Count..... | 372 |
| | 11.4.3 | Design Block Select Definitions..... | 372 |
| | 11.4.4 | Null Event..... | 374 |
| | 11.4.5 | Threshold Events..... | 375 |
| | 11.4.6 | External Input Events..... | 375 |
| | 11.4.6.1 | XPI Events Target ID(000001) / Design Block #(0100)..... | 375 |
| | 11.4.6.2 | SHaC Events Target ID(000010) / Design Block #(0101)..... | 380 |
| | 11.4.6.3 | XScale® Events Target ID(000100) / Design Block #(0111)..... | 384 |
| | 11.4.6.4 | PCI Events Target ID(000101) / Design Block #(1000)..... | 388 |
| | 11.4.6.5 | ME00 Events Target ID(100000) / Design Block #(1001)..... | 392 |
| | 11.4.6.6 | ME01 Events Target ID(100001) / Design Block #(1001)..... | 393 |
| | 11.4.6.7 | ME02 Events Target ID(100010) / Design Block #(1001)..... | 394 |
| | 11.4.6.8 | ME03 Events Target ID(100011) / Design Block #(1001) | 394 |
| | 11.4.6.9 | ME10 Events Target ID(110000) / Design Block #(1010) | 395 |
| | 11.4.6.10 | ME11 Events Target ID(110001) / Design Block #(1010) | 395 |
| | 11.4.6.11 | ME12 Events Target ID(110010) / Design Block #(1010) | 396 |
| | 11.4.6.12 | ME13 Events Target ID(110011) / Design Block #(1010) | 396 |
| | 11.4.6.13 | SRAM DP1 Events Target ID(001001) / Design Block #(0010)..... | 397 |
| | 11.4.6.14 | SRAM DP0 Events Target ID(001010) / Design Block #(0010)..... | 397 |
| | 11.4.6.15 | SRAM CH3 Events Target ID(001011) / Design Block #(0010)..... | 399 |
| | 11.4.6.16 | SRAM CH2 Events Target ID(001100) / Design Block #(0010)..... | 400 |
| | 11.4.6.17 | SRAM CH1 Events Target ID(001101) / Design Block #(0010)..... | 400 |
| | 11.4.6.18 | SRAM CH0 Events Target ID(001110) / Design Block #(0010)..... | 401 |
| | 11.4.6.19 | IXP2400 DRAM Events Target ID(010000) / Design Block #(0011)..... | 404 |
| 11.5 | | Software Support..... | 407 |
| | 11.5.1 | Introduction..... | 407 |
| | 11.5.2 | Mode of Operation..... | 407 |

Figures

| | | |
|---|---|----|
| 1 | IXP2400 Chassis Concept Block Diagram..... | 27 |
| 2 | Intel® XScale® Core 4GB (32-bit) Address Space..... | 32 |
| 3 | Microengine Block Diagram..... | 34 |

| | | |
|----|--|-----|
| 4 | An Expected Usage Model..... | 37 |
| 5 | DMA Descriptor Reads..... | 39 |
| 6 | Hash Unit Block Diagram | 43 |
| 7 | Generic SlowPort Connection | 47 |
| 8 | 8-bit SlowPort Interface Example (PMC-Sierra PM5351 S/UNI-TETRA)..... | 48 |
| 9 | Intel® XScale® Core Architecture Features..... | 50 |
| 10 | Example of Locked Entries in TLB | 58 |
| 11 | Instruction Cache Organization | 59 |
| 12 | Locked Line Effect on Round Robin Replacement..... | 64 |
| 13 | BTB Entry | 65 |
| 14 | Branch History..... | 66 |
| 15 | Data Cache Organization | 68 |
| 16 | Mini-Data Cache Organization | 69 |
| 17 | Locked Line Effect on Round Robin Replacement..... | 75 |
| 18 | Byte Steering for Read and Byte Enable Generation by the Intel® XScale® Core..... | 92 |
| 19 | Intel® XScale® Core Initiated Write to the IXP2400 Network Processor | 94 |
| 20 | Global Buses Connection to the XScale® Gasket..... | 96 |
| 21 | Flow Through the Intel® XScale® Core Interrupt Controller | 102 |
| 22 | Interrupt Masking Block Diagram | 103 |
| 23 | XPI Interfaces (IXP2400 A0/A1)..... | 104 |
| 24 | XPI Interfaces (IXP2400 B0) | 105 |
| 25 | PCI/XPI Data Flows Example (IXP2400 A0/A1)..... | 106 |
| 26 | PCI/XPI Data Flows Example (IXP2400 B0) | 106 |
| 27 | Second Example of Data Flows | 107 |
| 28 | UART Top Level Diagram | 109 |
| 29 | GPIO Functional Diagram | 113 |
| 30 | Timer Control Unit Interfacing Diagram..... | 115 |
| 31 | Timer Internal Logic Diagram | 116 |
| 32 | SlowPort Unit Interface Diagram | 120 |
| 33 | An Example of Address Space Hole Diagram..... | 121 |
| 34 | SlowPort Example Application Topology..... | 122 |
| 35 | Mode 0 Single Write Transfer for a Fixed-Timed Device (IXP2400 A0/A1) | 123 |
| 36 | Mode 0 Single Write Transfer for a Fixed-Timed Device (IXP2400 B0)..... | 124 |
| 37 | Mode 0 Single Write Transfer for a Self-Timing Device (IXP2400 A0/A1) | 124 |
| 38 | Mode 0 Single Write Transfer for a Self-Timing Device (IXP2400 B0)..... | 125 |
| 39 | Mode 0 Single Read Transfer for a Fixed-Timed Device (IXP2400 A0/A1) | 125 |
| 40 | Mode 0 Single Read Transfer for a Fixed-Timed Device (IXP2400 B0)..... | 126 |
| 41 | Mode 0 Single Read Transfer for a Self-Timing Device (IXP2400 A0/A1) | 127 |
| 42 | Mode 0 Single Read Transfer for a Self-Timing Device (IXP2400 B0)..... | 127 |
| 43 | An Interface Topology with Lucent TDAT042G5 SONET/SDH..... | 129 |
| 44 | Mode 1 Single Write Transfer for Lucent TDAT042G5 Device (IXP2400 A0/A1) | 130 |
| 45 | Mode 1 Single Write Transfer for Lucent TDAT042G5 | |

| | | |
|----|--|-----|
| | Device (IXP2400 B0)..... | 131 |
| 46 | Mode 1 Single Read Transfer for Lucent TDAT042G5 Device (IXP2400 A0/A1) | 132 |
| 47 | Mode 1 Single Read Transfer for Lucent TDAT042G5 Device (IXP2400 B0)..... | 133 |
| 48 | An Interface Topology with PMC-Sierra PM5351 S/UNI-TETRA | 134 |
| 49 | Mode 2 Single Write Transfer for PMC-Sierra PM5 | 351 |
| | Device (IXP2400 A0/A1) | 135 |
| 50 | Mode 2 Single Write Transfer for PMC-Sierra PM5351 Device (IXP2400 B0)..... | 136 |
| 51 | Mode 2 Single Read Transfer for PMC-Sierra PM5351 Device (IXP2400 A0/A1) | 137 |
| 52 | Mode 2 Single Read Transfer for PMC-Sierra PM5351 Device (IXP2400 B0)..... | 138 |
| 53 | An Interface Topology with Intel / AMCC SONET/SDH Device | 139 |
| 54 | Mode 3 Second Interface Topology with Intel / AMCC SONET/SDH Device ... | 140 |
| 55 | Mode 3 Single Write Transfer Followed by Read (IXP2400 A0/A1) | 141 |
| 56 | Mode 3 Single Write Transfer Followed by Read (IXP2400 B0) | 142 |
| 57 | Mode 3 Single Read Transfer Followed by Write (IXP2400 A0/A1) | 143 |
| 58 | Mode 3 Single Read Transfer Followed by Write (IXP2400 B0) | 144 |
| 59 | An Interface Topology with Intel / AMCC SONET/SDH Device in Motorola Mode | 145 |
| 60 | Second Interface Topology with Intel / AMCC SONET/SDH Device | 146 |
| 61 | Mode 4 Single Write Transfer (IXP2400 A0/A1) | 147 |
| 62 | Mode 4 Single Write Transfer (IXP2400 B0)..... | 148 |
| 63 | Mode 4 Single Read Transfer (IXP2400 A0/A1) | 149 |
| 64 | Mode 4 Single Read Transfer (IXP2400 B0)..... | 150 |
| 65 | Single Write Transfer for Fixed-Timed Device (IXP2400 A0/A1) | 151 |
| 66 | Framer Interrupt Enable Register Timing Diagram (IXP2400 A0/A1)..... | 152 |
| 67 | Single Write Transfer for Fixed-Timed Device (IXP2400 B0)..... | 153 |
| 68 | Framer Interrupt Enable Register Timing (IXP2400 B0) | 154 |
| 69 | Microengine Block Diagram | 156 |
| 70 | Context State Transition Diagram | 158 |
| 71 | Byte Align Block Diagram..... | 166 |
| 72 | CAM Block Diagram..... | 169 |
| 73 | Read from RBUF (64-bits) | 173 |
| 74 | Write to TBUF (64-bits) | 173 |
| 75 | 48-bit, 64-bit and 128-bits Hash Operand Transfer..... | 175 |
| 76 | Memory Controller's Communications | 181 |
| 77 | DRAM Controller | 188 |
| 78 | SRAM Controller/Chassis Block Diagram | 194 |
| 79 | Echo Clock Configuration..... | 196 |
| 80 | SRAM Clock Connection on a Channel | 198 |
| 81 | External Pipeline Registers Block Diagram..... | 198 |

| | | |
|-----|---|-----|
| 82 | Queue Descriptor with Four Links | 201 |
| 83 | Enqueueing One Buffer at a Time | 201 |
| 84 | Enqueue a String of Buffers to a Queue | 202 |
| 85 | Dequeue Buffer | 203 |
| 86 | Connection to a Coprocessor Through Standard QDR Interface | 214 |
| 87 | Coprocessor with Memory Mapped FIFO Ports | 215 |
| 88 | IXP2400 Chassis (APB and CSR Buses Not Shown) Block Diagram | 218 |
| 89 | SHaC Top Level Diagram | 219 |
| 90 | Ring Communication Logic Diagram | 223 |
| 91 | Hash Unit Block Diagram | 231 |
| 92 | An Expected Usage Model | 238 |
| 93 | IXP2400 Tx Master to Tx Slave Connection | 249 |
| 94 | IXP2400 Rx Master to Rx Slave Connection (WILL NOT WORK!) | 250 |
| 95 | Receive Functionality Simplified Block Diagram | 252 |
| 96 | RBUF Element State Transition Diagram | 268 |
| 97 | Transmit Function Simplified Block Diagram | 270 |
| 98 | TBUF State Transition Diagram | 285 |
| 99 | Full Duplex Mode Block Diagram | 290 |
| 100 | Simplex Mode Block Diagram | 295 |
| 101 | Block Diagram of the MSF Block to the Command and Push and Pull Buses .. | 297 |
| 102 | PCI Functional Blocks | 302 |
| 103 | Data Access Paths | 303 |
| 104 | PCI Arbiter Configuration Using CFG_PCI_ARB(GPIO[2]) | 313 |
| 105 | Example of Target Write to SRAM of 68 bytes | 315 |
| 106 | Example of Target Write to DRAM of 68 bytes | 317 |
| 107 | Example of Target Read from DRAM using 64-Byte Burst. | 318 |
| 108 | Generation of the Doorbell Interrupts to PCI | 320 |
| 109 | Generation of the Doorbell Interrupts to the Intel® XScale® Core | 320 |
| 110 | PCI Interrupts | 321 |
| 111 | PCI Address Generation for Command Bus Master to PCI | 328 |
| 112 | PCI Address Generation for Command Bus Master to PCI Configuration Cycle | 328 |
| 113 | Overall Clock Generation and Distribution | 341 |
| 114 | Reset Sequence | 349 |
| 115 | IXP2400 Initialization Sequence | 351 |
| 116 | Performance Monitor Interface Block Diagram | 360 |
| 117 | Block Diagram of a Single CHAP Counter | 362 |
| 118 | Basic Block Diagram of IXP2400 with PMU | 363 |
| 119 | CAP Interface to APB Bus | 364 |
| 120 | Conceptual Diagram of Counter Array | 366 |

Tables

| | | |
|---|---|----|
| 1 | Data Terminology | 26 |
| 2 | DMA Descriptor Format | 39 |
| 3 | Doorbell Interrupt Registers | 42 |
| 4 | Data Cache and Buffer Behavior when X = 0 | 53 |
| 5 | Data Cache and Buffer Behavior when X = 1 | 54 |
| 6 | Memory Operations that Impose a Fence | 55 |
| 7 | Valid MMU and Data/Mini-data Cache Combinations | 55 |

| | | |
|----|--|-----|
| 8 | Performance Monitoring Events | 78 |
| 9 | Some Common Uses of the PMU | 78 |
| 10 | Branch Latency Penalty | 82 |
| 11 | Latency Example..... | 84 |
| 12 | Branch Instruction Timings (Those predicted by the BTB)..... | 84 |
| 13 | Branch Instruction Timings (Those not predicted by the BTB)..... | 85 |
| 14 | Data Processing Instruction Timings..... | 85 |
| 15 | Multiply Instruction Timings..... | 86 |
| 16 | Multiply Implicit Accumulate Instruction Timings..... | 87 |
| 17 | Implicit Accumulator Access Instruction Timings | 87 |
| 18 | Saturated Data Processing Instruction Timings | 87 |
| 19 | Status Register Access Instruction Timings..... | 88 |
| 20 | Load and Store Instruction Timings | 88 |
| 21 | Load and Store Multiple Instruction Timings | 88 |
| 22 | Semaphore Instruction Timings | 88 |
| 23 | CP15 Register Access Instruction Timings | 89 |
| 24 | CP14 Register Access Instruction Timings | 89 |
| 25 | SWI Instruction Timings | 89 |
| 26 | Count Leading Zeros Instruction Timings | 89 |
| 27 | Little Endian Encoding | 90 |
| 28 | Big Endian Encoding..... | 90 |
| 29 | Byte Enable Generation by the Intel® XScale® Core for Byte Transfers in Little and Big Endian System | 91 |
| 30 | Byte Enable Generation by the Intel® XScale® Core for 16-bit Data Transfer in Little and Big Endian Systems | 93 |
| 31 | Byte Enable Generation by the Intel® XScale® Core for Byte Write In Little and Big Endian System..... | 93 |
| 32 | Byte Enable Generation by the Intel® XScale® Core for Word Writes in Little-Endian and Big-Endian Systems..... | 94 |
| 33 | CMB Write Command to CPP Command Conversion | 97 |
| 34 | IXP2400 Network Processor SRAM Q-Array Access Alias Addresses..... | 99 |
| 35 | GCSR Address Map (0xd700 0000) | 101 |
| 36 | Data Transaction Alignment..... | 107 |
| 37 | Address Spaces for XPI Internal Devices | 108 |
| 38 | UART Register Map | 110 |
| 39 | GPIO Register Map..... | 113 |
| 40 | Timer Register Map..... | 116 |
| 41 | 8-bit Flash Memory Device Density | 118 |
| 42 | SONET/SDH Devices | 119 |
| 43 | Byte Address..... | 120 |
| 44 | Single Write Transfer for Fixed-Timed Device Timing Parameters (IXP2400 A0/A1)..... | 151 |
| 45 | Framer Interrupt Enable Register Timing Parameters (IXP2400 A0/A1)..... | 152 |
| 46 | Single Write Transfer for Fixed-Timed Device Timing Parameters (IXP2400 B0) | 153 |
| 47 | Next Neighbor Write as a Function of CTX_Enable[NN_Mode]..... | 161 |
| 48 | Registers Used By Contexts in Context-Relative Addressing Mode | 164 |
| 49 | Align Value and Shift Amount | 166 |
| 50 | Register Contents for Example 21 | 167 |
| 51 | Register Contents for Example 22 | 167 |

| | | |
|-----|--|-----|
| 52 | Algorithm for Debug Software to Find out the Contents of the CAM..... | 171 |
| 53 | DDR Memory Auto Precharge Options | 182 |
| 54 | Supported Configurations..... | 182 |
| 55 | Clock Frequencies..... | 183 |
| 56 | DRAM Error Status..... | 185 |
| 57 | Supported Requests..... | 186 |
| 58 | Ordering Requirements | 187 |
| 59 | DDR Register Map | 191 |
| 60 | SRAM Controller Configurations | 197 |
| 61 | Total Memory Per Channel | 197 |
| 62 | Atomic Operations..... | 199 |
| 63 | Queue Format | 204 |
| 64 | Ring/Journal Format..... | 204 |
| 65 | Ring Size Encoding..... | 205 |
| 66 | Address Map | 211 |
| 67 | Address Reference Order | 212 |
| 68 | Q_array Entry Reference Order | 212 |
| 69 | Ring Full Signal Use — Number of Contexts and Length vs. Ring Size | 224 |
| 70 | Head/Tail, Base and Full by Ring Size..... | 225 |
| 71 | Ring CSR Summary and Addresses | 225 |
| 72 | Intel® XScale® Core and ME Instructions | 227 |
| 73 | Inter-Process Communication Register Summary | 228 |
| 74 | S Transfer Registers Hash Operands | 231 |
| 75 | Intel XScale® core Hash Operand Registers..... | 232 |
| 76 | Scratchpad Memory Register Summary | 235 |
| 77 | Hash Multiplier Register Summary..... | 235 |
| 78 | Global Chassis Registers..... | 235 |
| 79 | UTOPIA Levels 1-, 2-, and 3-Supported Specifications | 240 |
| 80 | Supported Bus Modes..... | 240 |
| 81 | Signal Usage in SPHY UTOPIA Mode | 241 |
| 82 | Supported Cell Sizes..... | 242 |
| 83 | Signal Usage in MPHY Mode..... | 243 |
| 84 | POS-PHY Levels 2 and 3-Supported Specifications..... | 245 |
| 85 | Signal Usage in POS-PHY Mode | 245 |
| 86 | Signal Usage in POS-PHY Mode | 248 |
| 87 | CFrames Assignment..... | 251 |
| 88 | Signal Usage in CSIX Mode..... | 251 |
| 89 | RBUF Partitioning Options | 255 |
| 90 | RBUF Byte Ordering | 255 |
| 91 | SRAM Read Transfer Register Byte Ordering | 256 |
| 92 | DRAM Byte Ordering..... | 256 |
| 93 | UTOPIA Receive Status Word Format..... | 257 |
| 94 | UTOPIA Receive Status Word Field Definitions..... | 257 |
| 95 | Cell Header Field Definitions..... | 258 |
| 96 | POS-PHY Receive Status Word Format | 259 |
| 97 | POS-PHY Receive Status Word Field Definitions..... | 259 |
| 98 | CSIX Receive Status Word Format..... | 262 |
| 99 | CSIX Receive Status Word Field Definitions..... | 262 |
| 100 | RX_Thread_Freelist Entries | 264 |
| 101 | Rx_Thread Freelist Association in UTOPIA and POS-PHY SPHY Modes | 265 |

| | | |
|-----|--|-----|
| 102 | Rx_Thread Freelist Association in UTOPIA/POS-PHY MPHY-4 Mode | 265 |
| 103 | Rx_Thread Freelist Association in UTOPIA/POS-PHY MPHY-32 Mode | 265 |
| 104 | Rx_Thread Freelist Association in CSIX Mode, MSF_Rx_Control[CSIX_Freelist]=0 | 266 |
| 105 | Rx_Thread Freelist Association in CSIX Mode, MSF_Rx_Control[CSIX_Freelist]=1 | 266 |
| 106 | UTOPIA, POS-PHY, and CSIX Mode Comparison | 268 |
| 107 | TBuf Byte Ordering | 271 |
| 108 | SRAM Write Transfer Register Byte Ordering | 272 |
| 109 | Example of Offset and Length Usage | 273 |
| 110 | Example of TBUF Element Transmission | 273 |
| 111 | UTOPIA/POS-PHY SPHY TBUF Partitioning | 274 |
| 112 | UTOPIA/POS-PHY MPHY-4 TBUF Partitioning..... | 275 |
| 113 | UTOPIA/POS-PHY MPHY-32 TBUF Partitioning..... | 275 |
| 114 | CSIX TBUF Partitioning | 276 |
| 115 | UTOPIA Transmit Control Word Format | 277 |
| 116 | UTOPIA Transmit Control Word Field Definitions | 277 |
| 117 | POS-PHY Transmit Control Word Format | 278 |
| 118 | POS-PHY Transmit Control Word Field Definitions | 278 |
| 119 | CSIX Transmit Control Word Format | 279 |
| 120 | CSIX Transmit Control Word Field Definitions | 279 |
| 121 | Transmit Control Word to CSIX Header Mapping | 284 |
| 122 | RBUF and TBUF Summary..... | 287 |
| 123 | Egress Processor Ready Bit Handling | 291 |
| 124 | Ingress Processor Ready Bit Handling | 291 |
| 125 | Atomic Target Write to Memory Option..... | 304 |
| 126 | PCI Core FIFO Sizes | 304 |
| 127 | Maximum Loading..... | 304 |
| 128 | PCI Commands..... | 304 |
| 129 | PCI BAR Programmable Sizes | 306 |
| 130 | PCI BAR Sizes with PCI host Initialization | 306 |
| 131 | Legal Combinations of the Strap Pin Options | 312 |
| 132 | Slave Interface Buffer Sizes..... | 314 |
| 133 | Doorbell Interrupt Registers | 319 |
| 134 | Internal Unit Interrupt Directly to PCI Option..... | 322 |
| 135 | DMA Descriptor Format | 324 |
| 136 | PCI Maximum Burst Size | 326 |
| 137 | Command Bus Master Configuration Transactions | 328 |
| 138 | Command Bus Master Address Space Map to PCI | 329 |
| 139 | Byte Lane Alignment for 64 bit PCI Data In (64 bits PCI little endian to big endian with Swap) | 334 |
| 140 | Byte Lane Alignment for 64 bit PCI Data In (64 bits PCI big endian to big endian without Swap)..... | 334 |
| 141 | Byte Lane Alignment for 32 bit PCI Data In (32 bits PCI little endian to big endian with Swap) | 334 |
| 142 | Byte Lane Alignment for 32 bit PCI Data In (32 bits PCI big endian to big endian without Swap)..... | 334 |
| 143 | Byte Lane Alignment for 64 bit PCI Data Out (big endian to 64 bits PCI little endian with Swap) | 335 |
| 144 | Byte Lane Alignment for 64 bit PCI Data Out (big endian to 64 bits PCI big endian without Swap)..... | 335 |

| | | |
|-----|---|-----|
| 145 | Byte Lane Alignment for 32 bit PCI Data Out (big endian to 32 bits PCI little endian with Swap) | 335 |
| 146 | Byte Lane Alignment for 32 bit PCI Data Out (big endian to 32 bits PCI big endian without Swap) | 335 |
| 147 | Byte Enable Alignment for 64 bit PCI Data In (64 bits PCI little endian to big endian with Swap) | 336 |
| 148 | Byte Enable Alignment for 64 bit PCI Data In (64 bits PCI big endian to big endian without Swap) | 336 |
| 149 | Byte Enable Alignment for 32 bit PCI Data In (32 bits PCI little endian to big endian with Swap) | 336 |
| 150 | Byte Enable Alignment for 32 bit PCI Data In (32 bits PCI big endian to big endian without Swap) | 337 |
| 151 | Byte Enable Alignment for 64 bit PCI Data Out (big endian to 64 bits PCI little endian with Swap) | 337 |
| 152 | Byte Enable Alignment for 64 bit PCI Data Out (big endian to 64 bits PCI big endian without Swap) | 337 |
| 153 | Byte Enable Alignment for 32 bit PCI Data Out (big endian to 32 bits PCI little endian with Swap) | 337 |
| 155 | PCI I/O Cycles Generate Data Swap Enable Option | 338 |
| 154 | Byte Enable Alignment for 32 bit PCI Data Out (big endian to 32 bits PCI big endian without Swap) | 338 |
| 156 | PCI Strap Pins | 339 |
| 157 | Clock Usage Summary | 342 |
| 158 | Available Clock Rates by Dividing Core PLL Output | 343 |
| 159 | Watchdog Timer Reset | 345 |
| 160 | Strap Pins required for IXP2400 | 347 |
| 161 | Legal Combinations of the Strap Pin Options | 348 |
| 162 | Description of the Content of Table 163 through Section 165 | 353 |
| 163 | Resetting IXP2400 Strategy—Logic Unrelated to IXP_RESETh | 353 |
| 164 | Resetting IXP2400 Scheme—Logic Related to IXP_RESET0[31:0] | 354 |
| 165 | Resetting IXP2400 Scheme—Logic Related to IXP_RESET1[31:0] | 356 |
| 166 | APB Bus Usage | 364 |
| 167 | PMU Design Unit Selection | 373 |
| 168 | Chap Counter Threshold Events (Design Block # 0001) | 375 |
| 169 | XPI PMU Event List | 375 |
| 170 | SHaC PMU Event List | 380 |
| 171 | XScale® Gasket PMU Event List | 384 |
| 172 | PCI PMU Event List | 388 |
| 173 | ME00 PMU Event List | 392 |
| 174 | ME01 PMU Event List | 393 |
| 175 | ME02 PMU Event List | 394 |
| 176 | ME03 PMU Event List | 394 |
| 177 | ME10 PMU Event List | 395 |
| 178 | ME11 PMU Event List | 395 |
| 179 | ME12 PMU Event List | 396 |
| 180 | ME13 PMU Event List | 396 |
| 181 | SRAM DP1 PMU Event List | 397 |
| 182 | SRAM DP0 PMU Event List | 397 |
| 183 | SRAM CH3 PMU Event List | 399 |
| 184 | SRAM CH3 PMU Event List | 400 |
| 185 | SRAM CH3 PMU Event List | 400 |



| | | |
|-----|-----------------------------------|-----|
| 186 | SRAM CH0 PMU Event List..... | 401 |
| 187 | IXP2400 DRAM PMU Event List | 404 |



Revision History

| Date | Revision | Description |
|---------------|----------|--|
| April 2002 | 001 | Initial release of preliminary manual for Intel® Field Personnel and customers under NDA. |
| June 2002 | 002 | Update for SDK 3.0 Pre-release 4. |
| July 2002 | 003 | Release of best known data for Intel® Field Personnel and customers under NDA. |
| October 2002 | 004 | Further updates for Pre-release 5; for Intel® Field Personnel and customers under NDA. |
| January 2003 | 005 | Incorporated IXP2400 B0 features, plus general updates for Intel® Field Personnel and customers under NDA. |
| April 2003 | 006 | General updates, especially to PCI Unit chapter, plus added new section, "Performance Monitor Unit"; for Intel® Field Personnel and customers under NDA. |
| November 2003 | 007 | Updated Media Switch Fabric maximum operating frequency in Section 2 and 8; updated Table 67 and 68; for Intel® Field Personnel and customers under NDA. |



This page intentionally left blank.

Introduction

1

1.1 About this Document

This document serves as the hardware reference manual for the Intel[®] IXP2400 Network Processor. This book is intended for use by developers and is organized as follows:

Section 2, “[Hardware Overview](#)” contains a hardware overview of the IXP2400 Network Processor.

Section 3, “[Intel[®] XScale[®] Core](#)” describes the operation of the embedded Intel XScale core.

Section 4, “[Microengines](#)” describes the operation of the Microengines.

Section 5, “[DDR SDRAM Controller](#)” describes the operation of the SDRAM Unit.

Section 6, “[SRAM Interface](#)” describes the operation of the SRAM Unit.

Section 7, “[SHaC Unit](#)” describes the Scratchpad, Hash Unit, and CSRs.

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 operation of the PCI Unit.

Section 10, “[Clocks, Reset, and Initialization](#)” describes the clocks, reset and initialization sequence.

1.2 Related Documentation

Further information on the IXP2400 is available in the following documents:

IXP2400 Network Processor Datasheet – Contains summary information on the *IXP2400* 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

This section describes the conventions used in this manual.

1.3.1 Data Terminology

Table 1. Data Terminology

| Term | Words | Bytes | Bits |
|----------|-------|-------|------|
| Byte | ½ | 1 | 8 |
| Word | 1 | 2 | 16 |
| Longword | 2 | 4 | 32 |
| Quadword | 4 | 8 | 64 |

1.3.2 Definitions

MPKT

The data read from a MAC device receive FIFO as the result of a single receive request to the receive state machine. The size of Mpkt is the same as the size of RBUF or TBUF entries. The size of RBUF and TBUF entries are user configurable and can be 64, 128, or 256 bytes in length.

Packet

The data framed between the assertion of an SOP signal and assertion of its associated EOP signal.

Hardware Overview

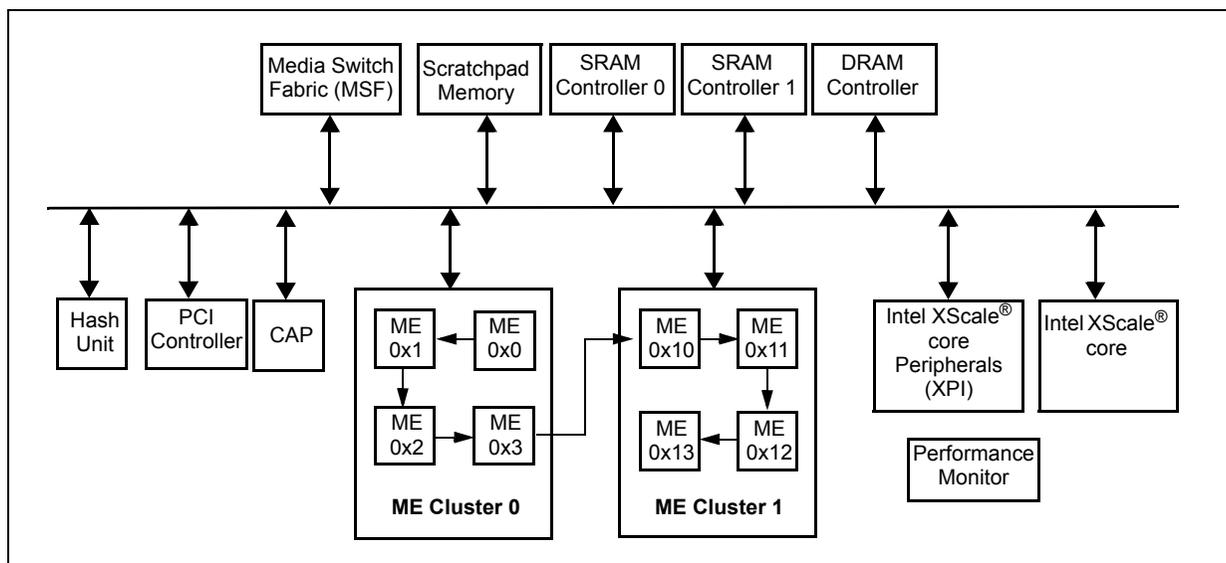
2

2.1 Overview

This chapter provides an introduction to the IXP2400 internal hardware. Specific details of each of the hardware functions are included in corresponding chapters in this manual.

Figure 1 shows a block diagram of the chip, including the major internal blocks.

Figure 1. IXP2400 Chassis Concept Block Diagram



The major blocks are:

- Intel XScale® core (Section 2.2)—A general-purpose, 32-bit RISC processor compatible to ARM Version 5 Architecture. The Intel XScale® core initializes and manages the chip, and can be used for higher layer network processing tasks.
 - A high-performance, low-power, 32-bit embedded RISC processor
 - 32-Kbyte Instruction Cache and 32-Kbyte Data Cache
 - 2-Kbyte mini-Data Cache that facilitates transient data processing
 - Four outstanding-pending read requests before stalling the processor
 - New instructions sets
 - Performance monitor features
 - JTAG/boundary scan debug support
- Microengines (MEs) (Section 2.3)—eight 32-bit programmable engines specialized for network processing; these MEs handle the main data plane processing per packet
 - Eight threads per ME with no overhead for context switching

- 4K x 40-bit instruction control store per ME
- Enhanced instructions sets (MEv2)
- 640 32-bit local memory per ME
- 256 GPRs
- Total of 512 transfer registers
- 128 Next Neighbor registers
- Multiplier per ME to support 8 x 24, 16 x 16, 32 x 32 multiplications

- SRAM Controller ([Section 2.4](#))—two independent controllers for QDR SRAM. Typically SRAM is used for control information storage.
 - Two independent channels
 - Peak bandwidth of 1.6 Gbyte/second per channel
 - Supports frequencies of 100, 150, or 200 MHz
 - Address up to 64 Mbytes per channel
 - Parity protected data
 - Enqueue/Dequeue support
 - Support atomic swap, bit set, bit clear, increment, decrement, add operations

- DRAM Controller ([Section 2.5](#))—1 DDR SDRAM controller. Typically DRAM is used for data buffer storage.
 - Peak bandwidth 2.4 GByte/sec. per channel at frequency of 150 and 100 MHz
 - 1 independent channels provided
 - Address up to 2 GB
 - ECC protected data

- Media and Switch Fabric Interface (MSF) ([Section 2.6](#))—Interface for network framers and/or Switch Fabric. Contains receive and transmit buffers.
 - Configurable to either of the following:
 - UTOPIA 1/2/3, POS-2, SPI-3, CSIX (only in 32-bit mode)
 - UTOPIA/POS/CSIX Interface:
 - Supports 1 UTOPIA 1/2/3 or POS-2 or SPI-3 interfaces at 104 MHz that can be overclocked to 133 MHz

- PCI Controller ([Section 2.7](#))—64-bit PCI Rev 2.2 compliant IO bus. PCI can be used to either connect to a Host processor, or to attach PCI compliant peripheral devices.
 - Compliant with PCI 2.2 spec.

- Support 64-bit interface at 66 MHz
 - Link-list-based DMA transfer to/from DRAM
 - Master/slave support
-
- The SHaC unit contains three main subblocks: the Scratchpad, the Hash units and the CAP (CSR Access Proxy)
 - Scratchpad Memory (Section 2.8)—16-Kbyte storage for general-purpose use with atomic operations and ring support
 - Hash Unit (Section 2.9)—Polynomial hash accelerator; the Intel XScale® core and Microengines can use it to offload hash calculations
 - CAP (Section 2.10)—Chip-wide control and status registers; these provide special inter-processor communication features to allow flexible and efficient inter-ME and ME-to-Intel XScale® core communications
 - Performance monitor—Counters that can be programmed to count selected internal chip hardware events; used to analyze and tune performance
 - Intel XScale® core peripherals (XPI) (Section 2.11)—Interrupt Controller, Four Timers, one serial UART port, eight general-purpose IO (GPIO) and interface to low-speed off-chip peripherals (such as maintenance port of network devices) and Flash ROM.

2.2 Intel® XScale® Core

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

2.2.1 ARM Compatibility

The Intel XScale® core is compatible to ARM Version 5 (V5) Architecture. It implements the integer instruction set 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.

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® core 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® 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
- enabling the write buffer to coalesce stores to external memory

2.2.2.3 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.

2.2.2.4 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.

2.2.2.5 Data Cache

The Intel XScale® core 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® 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.

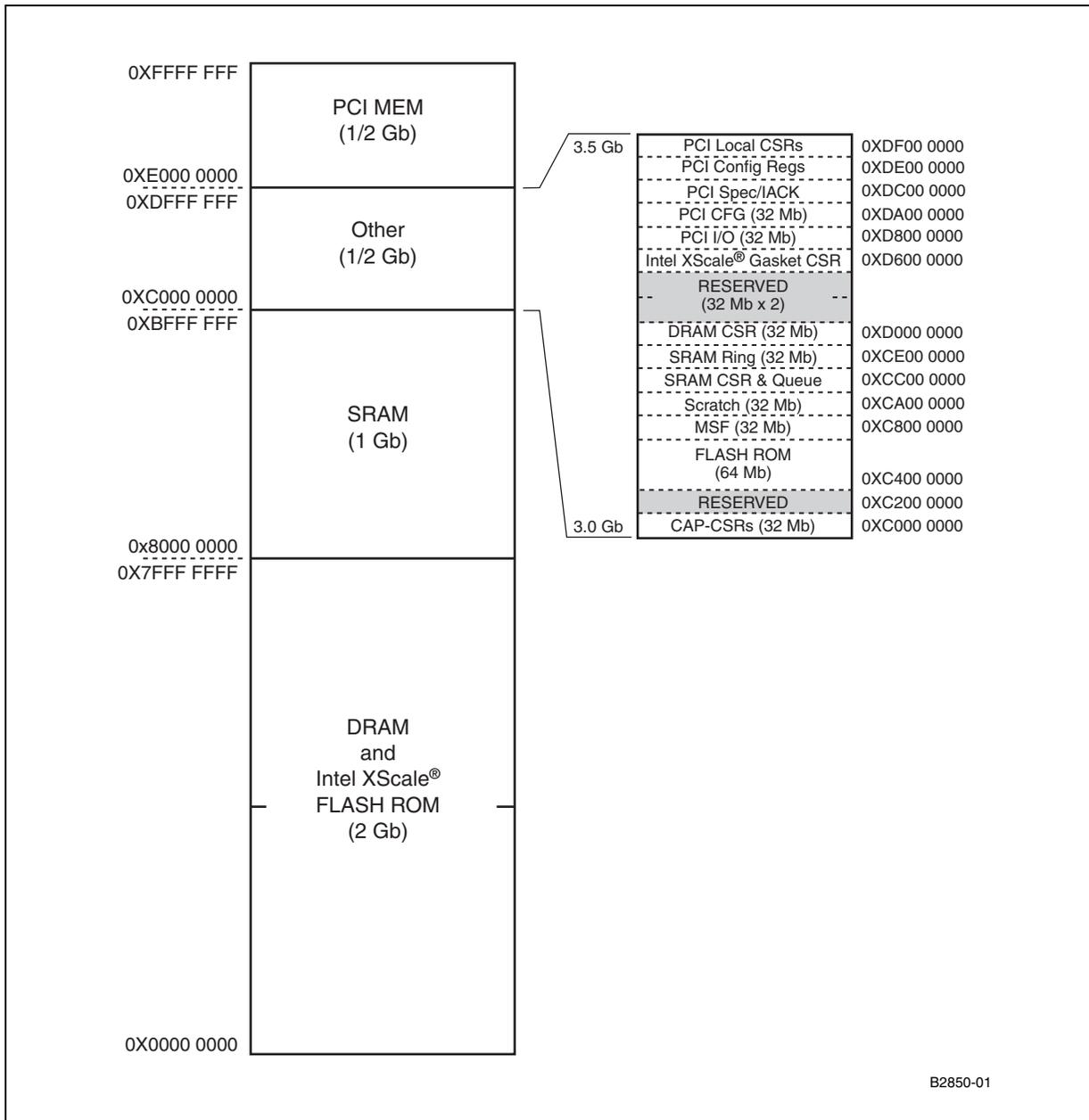
2.2.2.6 Interrupt Controller

The Intel XScale® core 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® core Peripherals (Section 2.11), which is used to mask and steer many chip-wide interrupt sources.

2.2.2.7 Address Map

Figure 2 shows the partitioning of the Intel XScale® core 4 GB address space.

Figure 2. Intel® XScale® Core 4GB (32-bit) Address Space



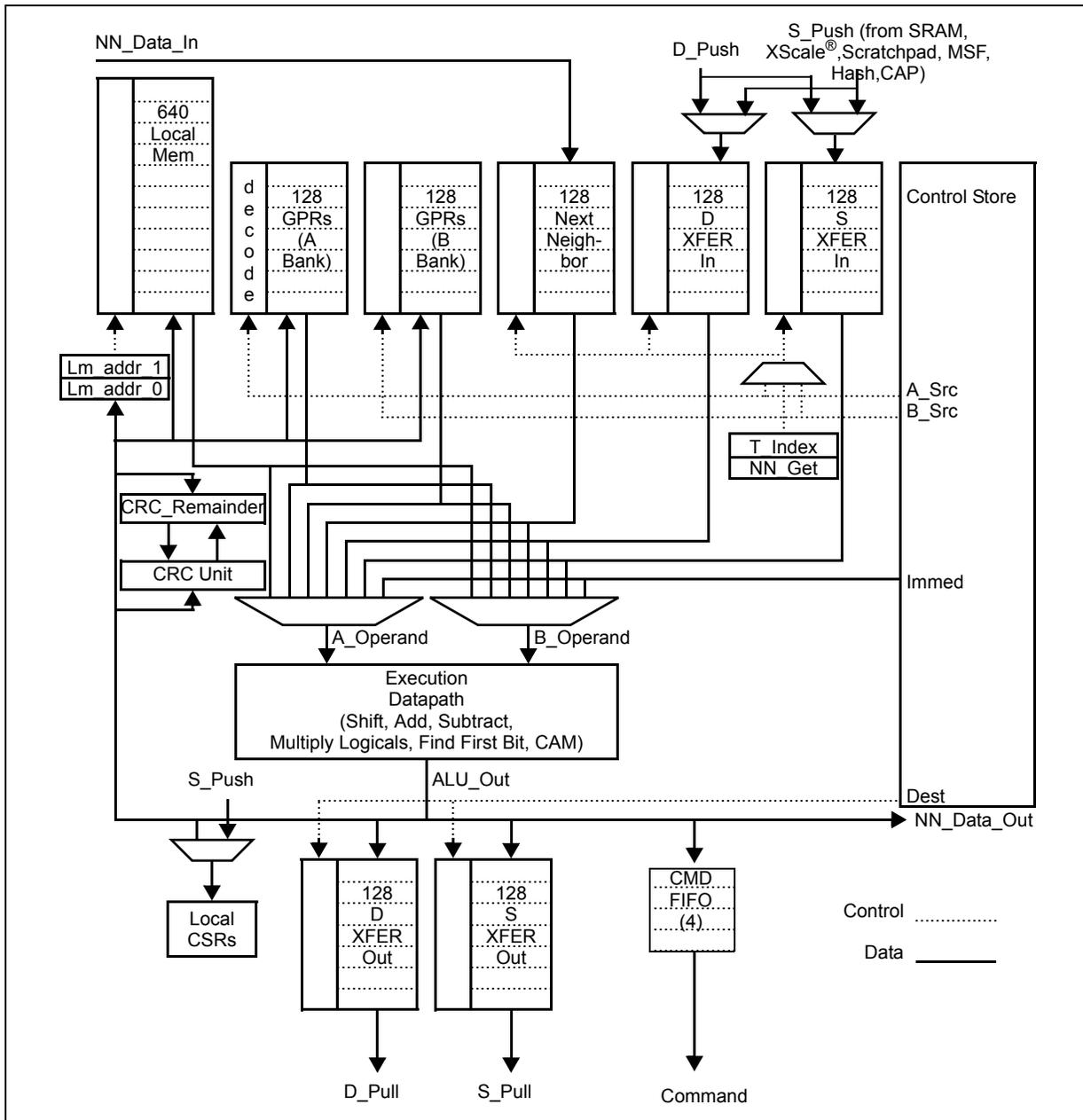
2.3 Microengine

The Microengines (MEs) do most of the programmable per packet processing in IXP2400. There are 8 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 3](#) is used in the overview of the Microengines. 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 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 3. Microengine Block Diagram



2.4 SRAM

The IXP2400 Network Processor has two 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. SRAMs 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 that select which bytes to write for writes of less than 32 bits.

Best efforts have been made to provide impedance controls within the IXP2400 for IXP2400-initiated signals driving to QDR devices. *Providing a clean signaling environment is critical to achieving 200-MHz QDR II data transfers.*

The configuration assumptions for IXP2400 I/O driver/receiver development includes 4 QDR loads and IXP2400. It should be noted that some future QDR II SRAMs require a burst of 4 to achieve higher frequency. The IXP2400 initial release will not support bursts of four QDR SRAM devices; the initial release supports bursts of two 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 DRAM

The Memory Controller is responsible for controlling the off-chip DRAM and provides a mechanism for other functional units in the IXP2400 to access the DRAM. The IXP2400 supports a single 64-bit channel (72 bit with ECC) of DRAM. DRAM sizes of 64, 128, 256, 512-Mb, and 1 Gb are supported. The DRAM channel can be populated with either a single- or double-sided DIMM.

An address space of 2 GB is allocated to DRAM. The memory space is guaranteed to be contiguous from a software perspective. If less than 2 GB of memory is present, the upper part of the address space is aliased into the lower part of the address space and should not be used by software.

Reads and writes to DRAM are generated by the Microengines, the Intel XScale® core and PCI bus masters. They are connected to the controllers via the Command Bus and Push and Pull Buses. The memory controller takes commands from these sources and enqueues them. The commands are dequeued, according to the priority defined later in this chapter, and the accesses to the DRAM are performed. The controller also does refresh cycles to the DRAMs.

ECC (Error Correcting Code) is supported, but can be disabled. Enabling ECC requires that x72 DIMMs be used. If ECC is disabled, x64 DIMMs can be used.

2.5.1 Feature List

- Supports one DDR SDRAM channel, 64b wide (72b with ECC)
- Supports DDR devices up to 300 MTs
- Supports 64-, 128-, 256-, 512-Mb, and 1-Gb technologies for x8 and x16 devices (DIMM and direct-soldered)
- Hardware-controlled interleaving to spread contiguous addresses across multiple banks
- All supported devices have four banks
- Configurable, optional error correction using ECC bits
- Supports 1 single- or double-sided DIMM
- Supports up to 2 GB memory capacity (using 1-Gb technology)

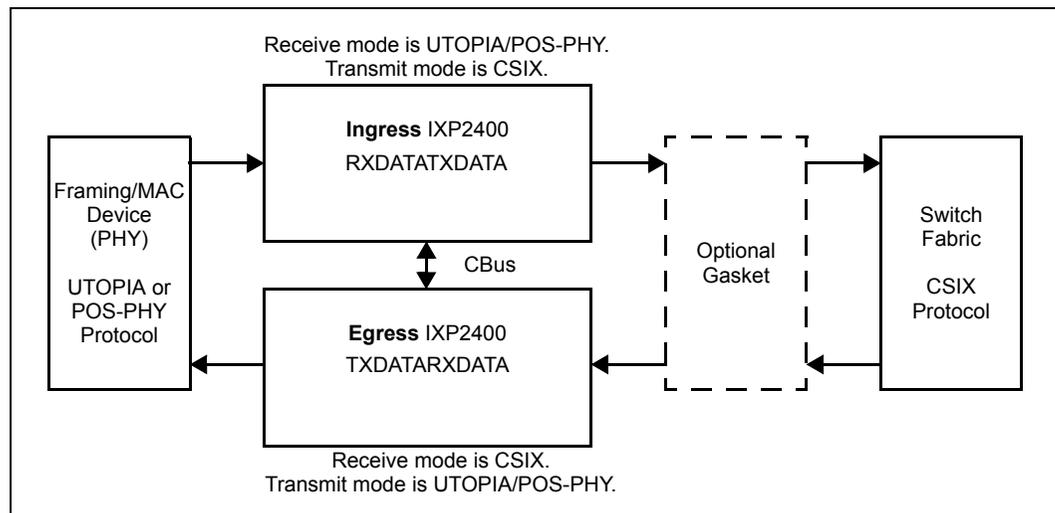
2.6 Media and Switch Fabric (MSF) Interface

The Media and Switch Fabric (MSF) Interface is used to connect IXP2400 to a physical layer device (PHY) and/or to a switch fabric. The MSF has the following major features:

- Separate and independently configurable 32-bit receive and transmit buses.
- A configurable bus interface; the bus may function as a single 32-bit bus, or it can be channelized into independent buses: two 16-bit buses, four 8-bit, or one 16-bit bus and two 8-bit buses. Each channel may be configured to operate in either UTOPIA or POS-PHY modes.
- The Media bus operates from 25 to 133 MHz.
- UTOPIA Level 1/2/3 and POS-PHY Level 2/3 single-PHY (SPHY) master operation; 8-, 16-, or 32-bit buses are supported.
- UTOPIA Level 3 multi-phy (MPHY) master operation with a 32-bit-wide bus; up to 16 slave ports are supported; polling may be single RxClav/TxClav, or Direct Status Indication (maximum of four slave ports).
- POS-PHY Level 3 multi-phy (MPHY) master operation with a 32-bit-wide bus with in-band addressing; up to 16 slave ports are supported; polling may be packet level or byte level.
- Support for CSIX-L1 protocol with a 32-bit-wide bus. The only deviation from the CSIX-L1 specification is that the IXP2400 is clocked by a globally synchronous clock and is electrically 3.3V LVTTTL.
- Support for interprocessor CBus for communicating link level and fabric level flow control information between egress and ingress processors in CSIX mode.
- Interface to internal buses: command, SRAM push/pull, and DRAM push/pull.

Figure 4 shows one expected usage model.

Figure 4. An Expected Usage Model



Note: In this document, *UTOPIA* always refers to cell transport; *POS-PHY* refers to variable length packet transport; *CSIX* refers to CFrame transport.

2.6.1 Reference Documents

The reader should be familiar with the following specifications:

1. *UTOPIA Specification, Level 1*, Version 2.01, March 21, 1994
2. *UTOPIA Level 2 Specification*, Version 1.0, June 1995
3. *UTOPIA 3 Physical Layer Interface*, November 1999
4. *POS-PHY Level 2 Specification*, Issue 5, December 1998
5. *POS-PHY Level 3 Specification*, Issue 4, June 2000
6. *SPI-3 Specification*, June 2000
7. *Frame Based ATM Interface (Level 3)*, March 2000
8. *CSIX-L1: Common Switch Interface Specification -L1*, Version 1.0, August 5, 2000

2.7 PCI Controller

The PCI Controller provides 64-bit, 66-MHz-capable PCI 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 (Intel XScale® core access to PCI Target devices)
- Three DMA channels

- Mailbox and Doorbell registers for Intel XScale® core-to-host communication
- PCI Arbiter

IXP2400 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.

2.7.1 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 IO device such as an Ethernet controller, SCSI controller, or encryption coprocessor.

Strapping Options (Without PROM Boot)

The SRAM BAR can be strapped to sizes of 32, 64, 128, or 256 MB.

The DRAM BAR can be strapped to sizes of 128, 256, 512 MB, or 1 GB.

The CSR BAR is 1 MB.

Programmable Options (With PROM Boot)

The SRAM BAR can be programmed to sizes of 0 bytes, 256 or 512 KB, 1, 2, 4, 8, 16, 32, 64, 128, or 256 MB.

The DRAM BAR can be programmed to sizes of 0 bytes, 1, 2, 4, 8, 16, 32, 64, 128, 256, or 512 MB, or 1 GB.

The CSR BAR is 1 MB.

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 SlowPort ([Section 2.11.5](#)).

2.7.2 Master Access

The Intel XScale® core processor and Microengines can directly access PCI bus. The Intel XScale® core can do loads and stores to specific address regions to generate all PCI command types (see [Figure 2](#)). Microengines use PCI instructions, and also use address regions to generate different PCI commands. Master access can also be generated by DMA.

2.7.3 DMA Channels

There are three 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.

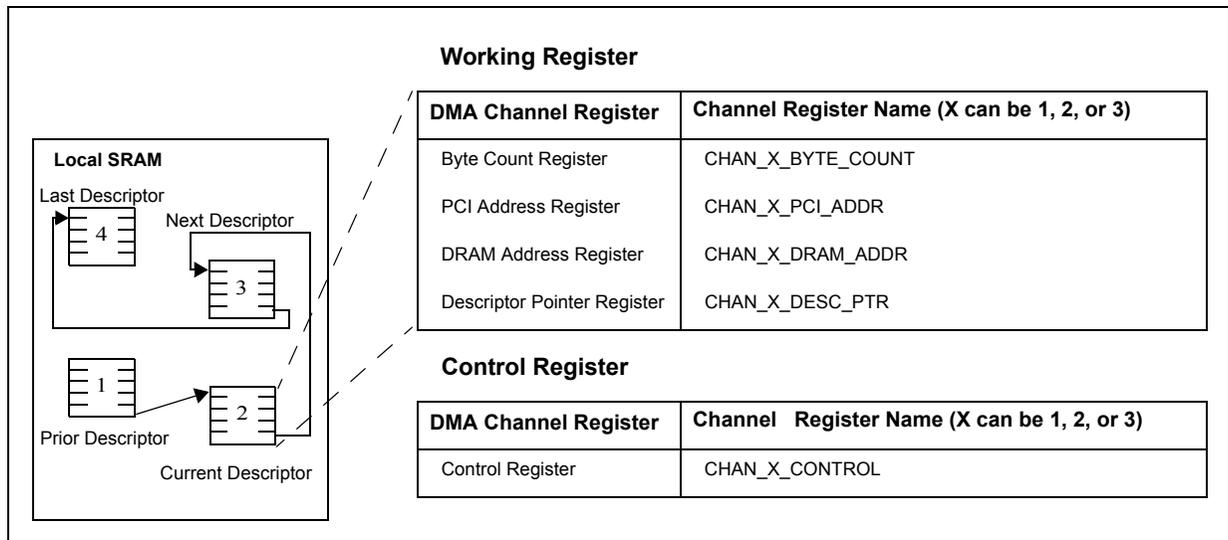
Up to three DMA channels are running at a time with three descriptors outstanding. Effectively, the active channels interleave bursts to or from the PCI Bus.

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

2.7.3.1 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 5. Table 2 lists the contents of the descriptor.

Figure 5. 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 (that is, until the *End of Chain* bit is set).

Table 2. DMA Descriptor Format

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

2.7.3.2 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. (If the end of chain bit is not set, the DESC_PTR will point to the next descriptor in SRAM).
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.

2.7.3.3 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.

2.7.3.4 Adding Descriptor to 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 (that is, 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.

2.7.4 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 Intel XScale® core DOORBELL register that is used by a PCI device to generate an 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 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 3](#).

Table 3. 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. |

2.7.5 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 IXP2400) arbiter used. In that case, IXP2400 will provide its PCI request signal to the external arbiter, and use that arbiters 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.

2.8 Scratchpad Memory

IXP2400 contains a 16KB Scratchpad Memory, organized as 4K 32-bit words, that is accessible by Microengines and Intel XScale® core.

The Scratchpad Memory provides the following operations:

- Normal reads and writes—from one to 16 32-bit words can be read/written with a single Microengine instruction
 - Note: Scratchpad is not byte-writeable (each write must write all 4 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
- 16 Hardware Assisted Rings¹ for interprocess communication

1. 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 Intel XScale® core. The Microengines 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.

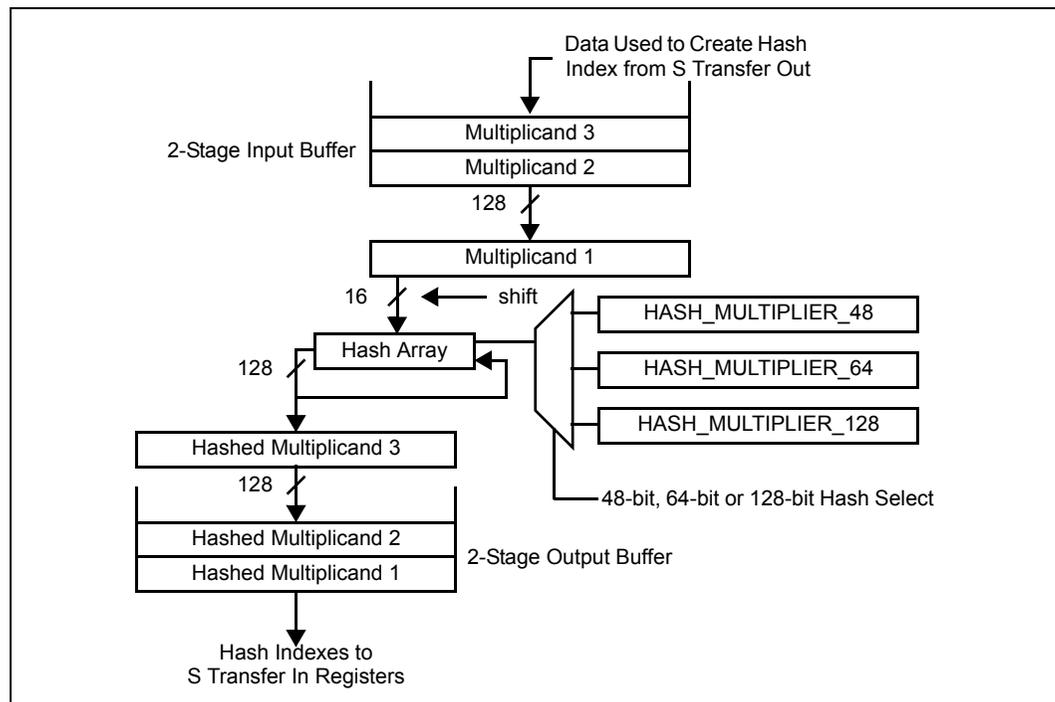
2.9 Hash Unit

IXP2400 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 Intel XScale® core, and is useful in doing table searches with large keys, for example L2 addresses. Figure 6 is a block diagram of the Hash Unit.

Up to three hash indices 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.

Figure 6. Hash Unit Block Diagram



The Hash Unit uses a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indices. 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 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 buffer.

After each hash index is completed, the Hash Unit returns the hash index to the Microengines S Transfer In Registers, or 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 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 is the sum of 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.

2.10 Control and Status Register Access Proxy

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-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 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

IXP2400 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 ([Section 2.11.3](#)). 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 GPIO

IXP2400 contains eight General Purpose IO (GPIO) pins. These can be programmed as either input or output and can be used for slow speed IO such as LEDs or input switches. They can also be used as interrupts to the Intel XScale® core or to clock the programmable timers ([Section 2.11.2](#)).

2.11.4 UART

IXP2400 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.

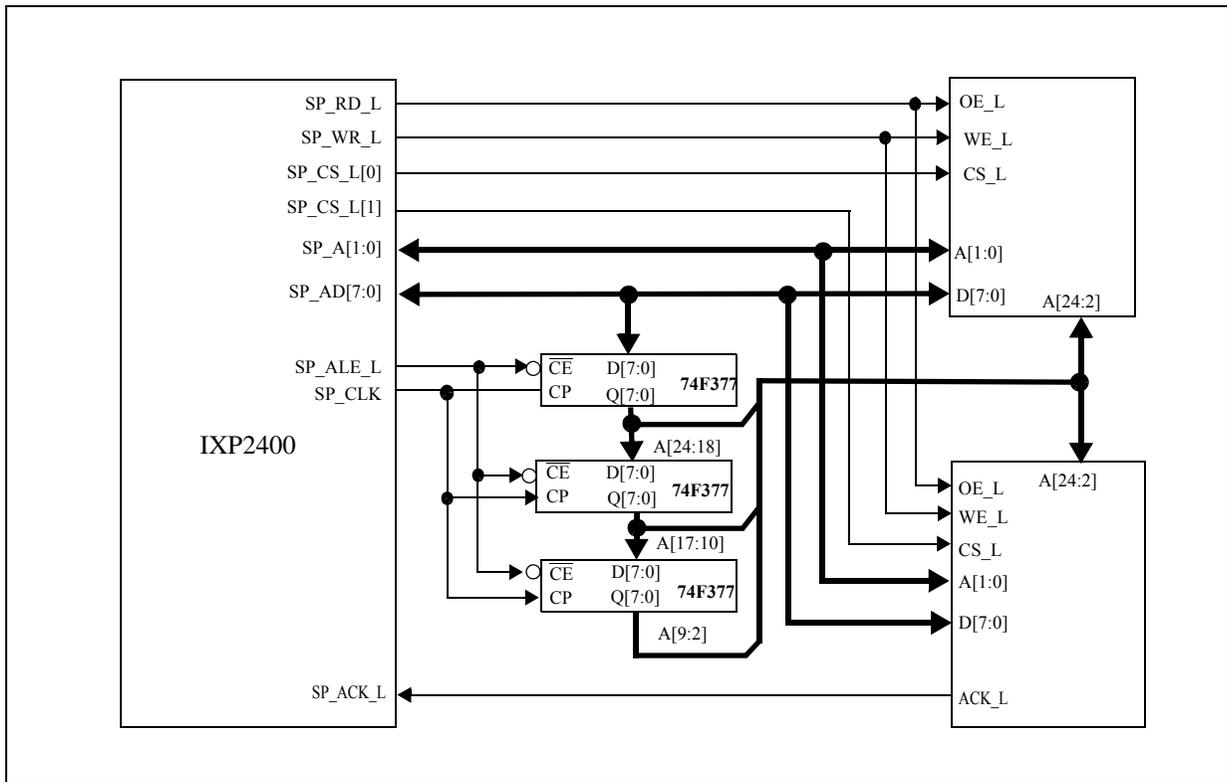
The UART includes a programmable baud rate generator which is capable of dividing the internal clock input by divisors of 1 to $2^{16} - 1$ and produces a 16X clock to drive the internal transmitter logic. It also drives the receive logic. The UART can be operated in polled or in interrupt driven mode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

2.11.5 SlowPort

The SlowPort is an external interface to IXP2400, 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 pincount. In addition, 24 bits of address are shifted out on three clock cycles. Therefore, an external set of buffers (such as 74F377) is needed to latch the address. Two chip selects are provided. See [Figure 7](#).

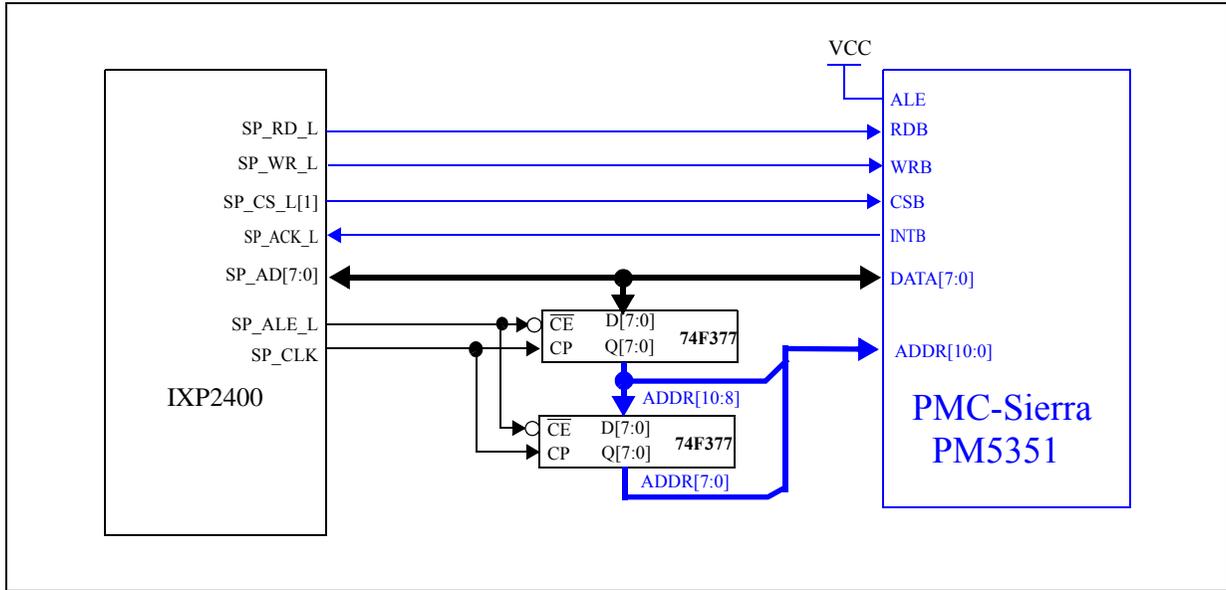
Figure 7. Generic SlowPort Connection



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.

Figure 8 shows an interface to an 8-bit device.

Figure 8. 8-bit SlowPort Interface Example (PMC-Sierra PM5351 S/UNI-TETRA)



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

For additional information about Intel XScale architecture, refer to the *Intel® XScale® Core Developer's Manual*, available on the Intel Developer website (<http://www.intel.com/design/intelxscale/273473.htm>).

3.1 Introduction

The 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 XScale core incorporates an extensive list of architecture features that allows it to achieve high performance. Many of the architectural features added to the 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 XScale core implements the integer instruction set architecture of ARM* V5, but does not provide hardware support of the floating point instructions.

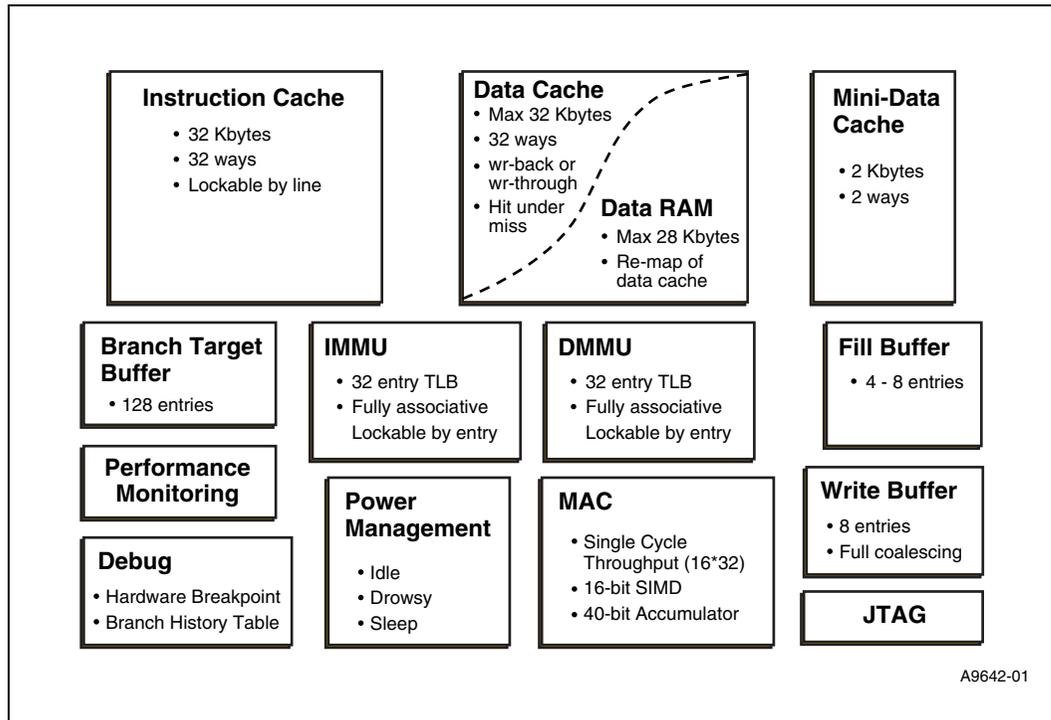
The XScale core provides the Thumb instruction set (ARM* V5T) and the ARM* V5E DSP extensions.

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

3.2 Features

Figure 9 shows the major functional blocks of the XScale core. The following sections give a brief, high-level overview of these blocks.

Figure 9. 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 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 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 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 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 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 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 XScale core incorporates a power and clock management unit that can assist ASSPs (Application Specific Standard Product) in controlling their clocking and managing their power.

3.2.8 Debug

The 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 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 XScale core such as built-in self-test, boundary-scan, and scan.

3.3 Memory Management

The XScale core implements the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Reference Manual*. To accelerate virtual to physical address translation, the 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 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 (Process ID) 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 4](#)), 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 5](#).

Table 4. 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 ^a |
| 0 1 | N | Y | - | - | |
| 1 0 | Y | Y | Write Through | Read Allocate | |
| 1 1 | Y | Y | Write Back | Read Allocate | |

a. 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 5. 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 ^a |
| 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 | |

a. 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 6 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 6. 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 7](#)). The invalid combination will cause undefined results.

Table 7. Valid MMU and 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 1](#).

Example 1. 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 2](#).

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 2](#).

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 2. 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 3](#).

Example 3. 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

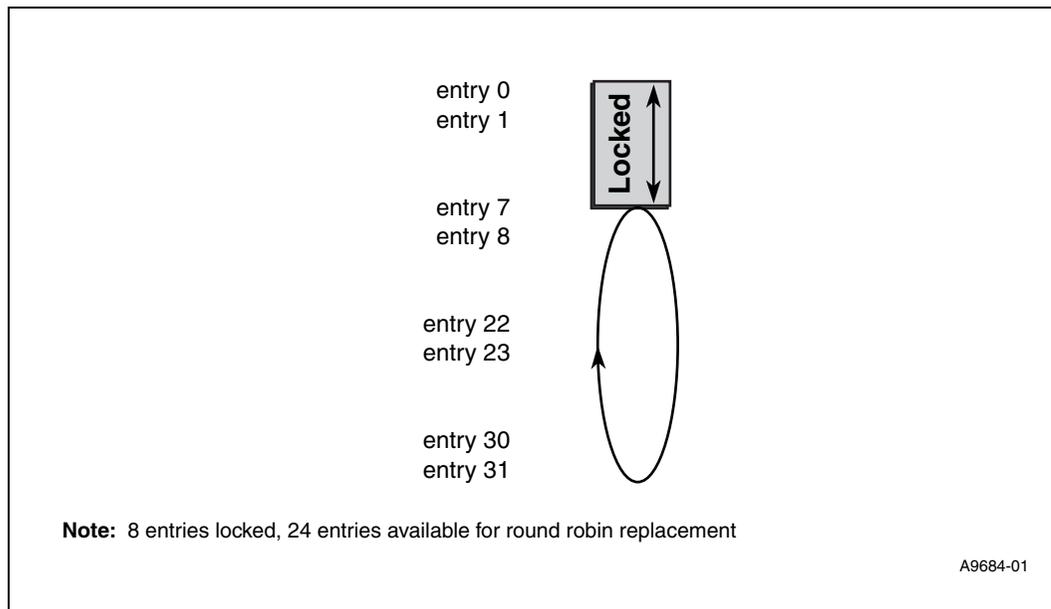
The line replacement algorithm for the TLBs is round-robin; there is a round-robin pointer that keeps track of the next entry to replace. The next entry to replace is the one sequentially after the last entry that was written. For example, if the last virtual to physical address translation was written 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 10. Example of Locked Entries in TLB



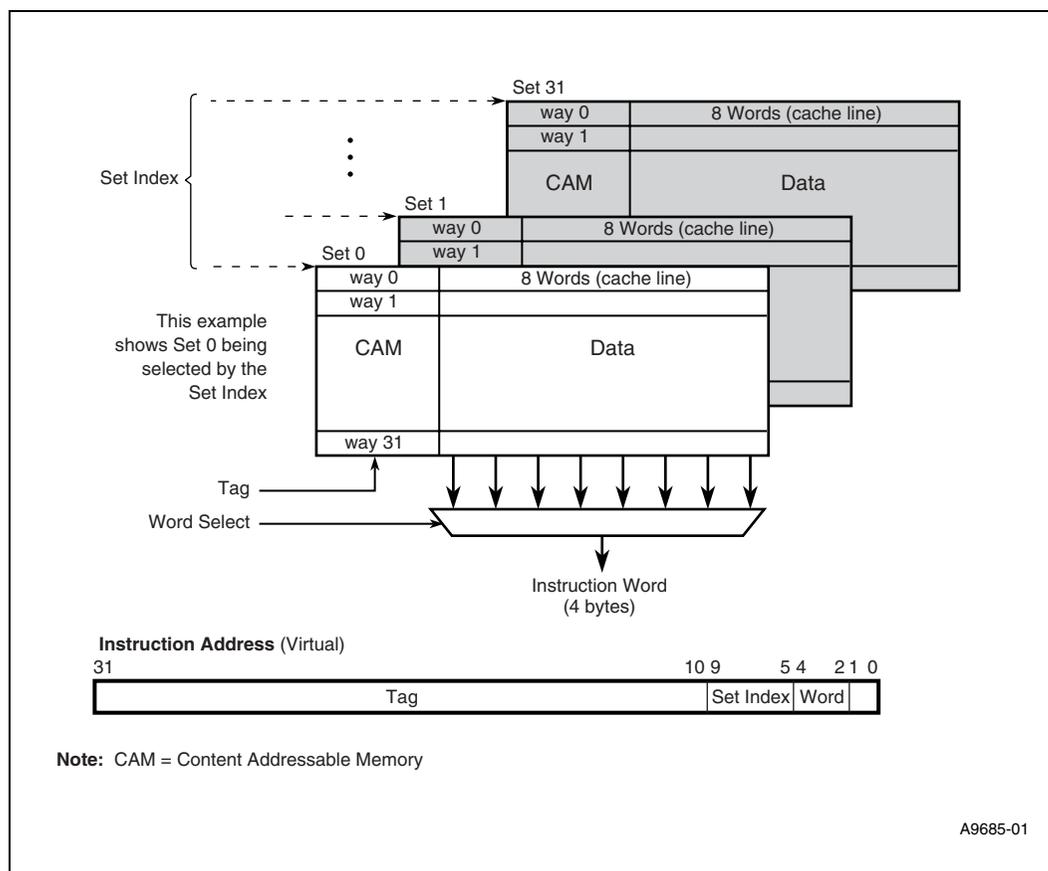
3.4 Instruction Cache

The 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 11 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 11. 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 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 4](#). A more complex handler might choose to invalidate the specific line that caused the exception and then invalidate the BTB.

Example 4. 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 Fetch Latency

The instruction fetch latency is dependent on the core to memory frequency ratio, system bus bandwidth, system memory, etc.

3.4.1.7 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 5](#).

Example 5. 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 6](#) can be used to guarantee proper synchronization.

Example 6. 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 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:

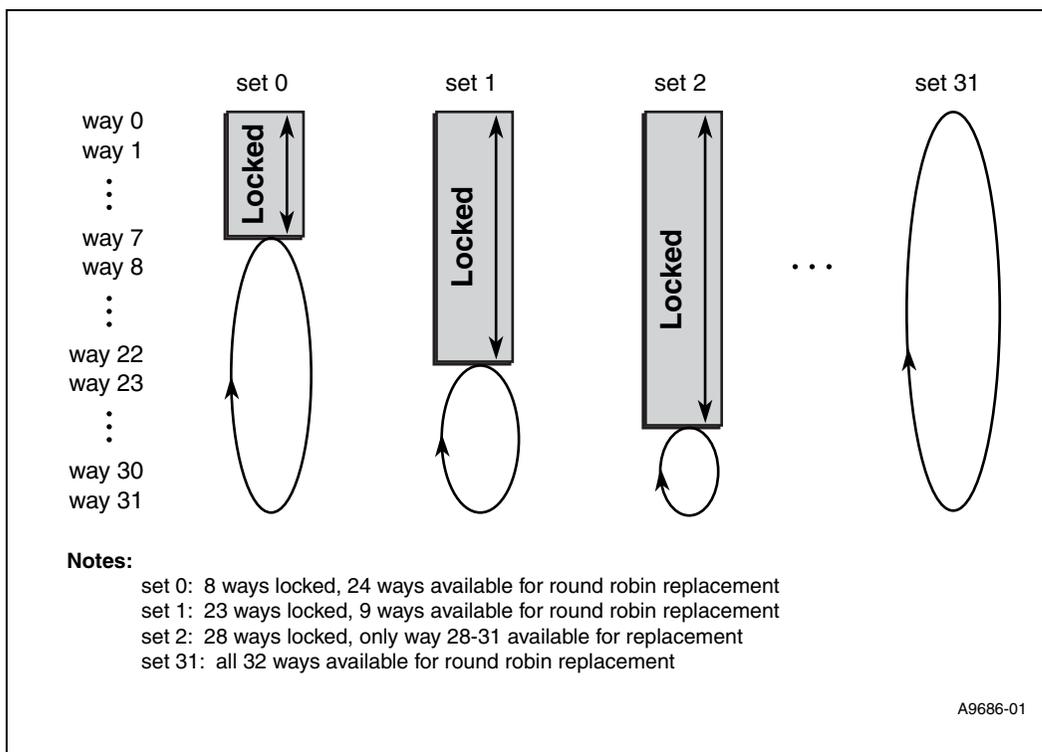
1. the routine used to lock lines down in the cache must be placed in non-cacheable memory, which means the MMU is enabled. As a corollary: no fetches of cacheable code should occur while locking instructions into the cache.
2. the code being locked into the cache must be cacheable
3. the instruction 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 12](#) is an example of where lines of code may be locked into the cache along with how the round-robin pointer is affected.

Figure 12. 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 12.

Example 7 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 7. 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 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.

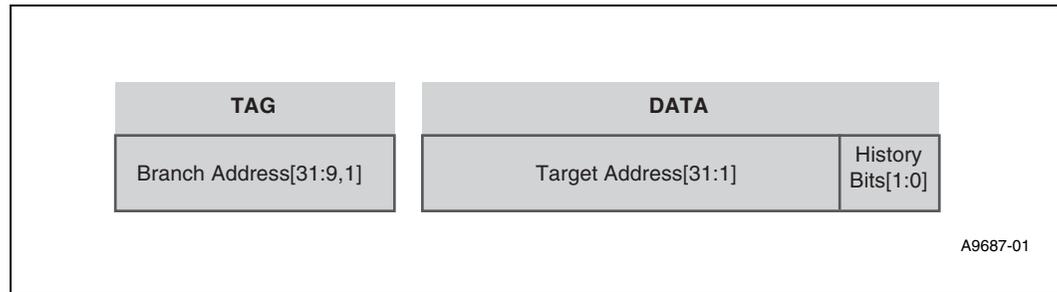
3.5 Branch Target Buffer

The XScale core uses dynamic branch prediction to reduce the penalties associated with changing the flow of program execution. The 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 13](#) 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 13. BTB Entry



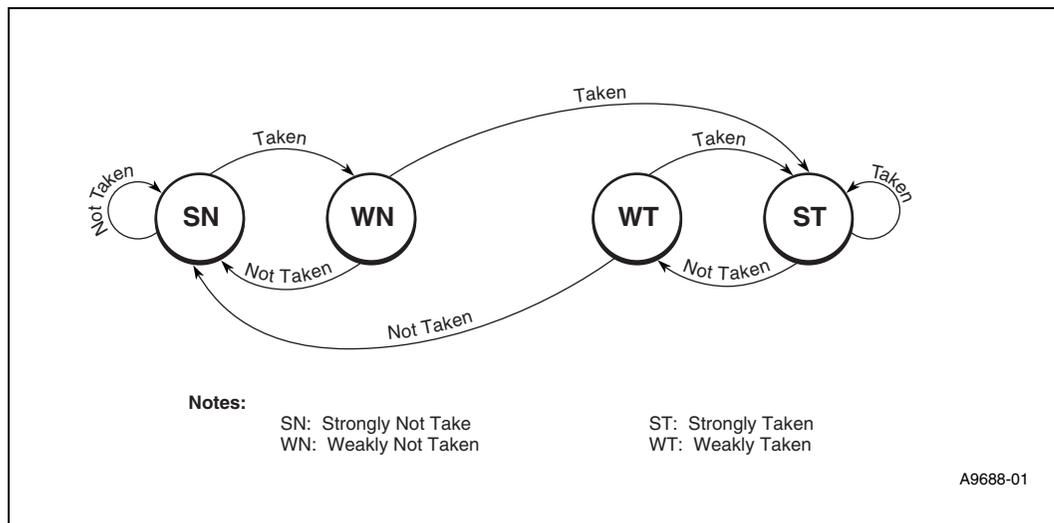
The BTB takes the current instruction address and checks to see if this address is a branch that was previously seen. It uses bits [8:2] of the current address to read out the tag and then compares this tag to bits [31:9,1] of the current instruction address. If the current instruction address matches 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 14](#) 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 14. 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 14.

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 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 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 XScale core instruction execution from external memory accesses, which increases overall system performance.

3.6.1 Overviews

3.6.1.1 Data Cache Overview

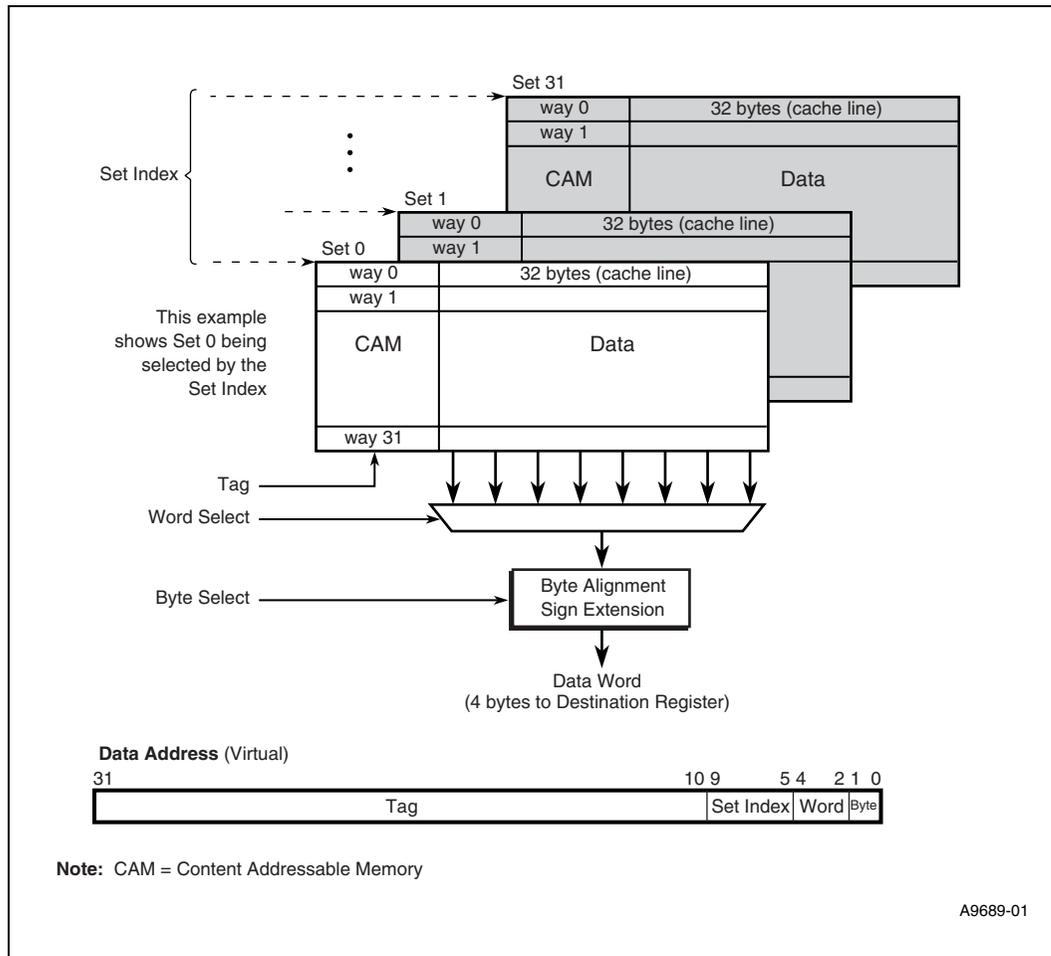
The data cache is a 32-Kbyte, 32-way set associative cache; this means there are 32 sets with each set containing 32 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There also exist two dirty bits for every line, one for the lower 16 bytes and the other one for the upper 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 15 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 15. Data Cache Organization



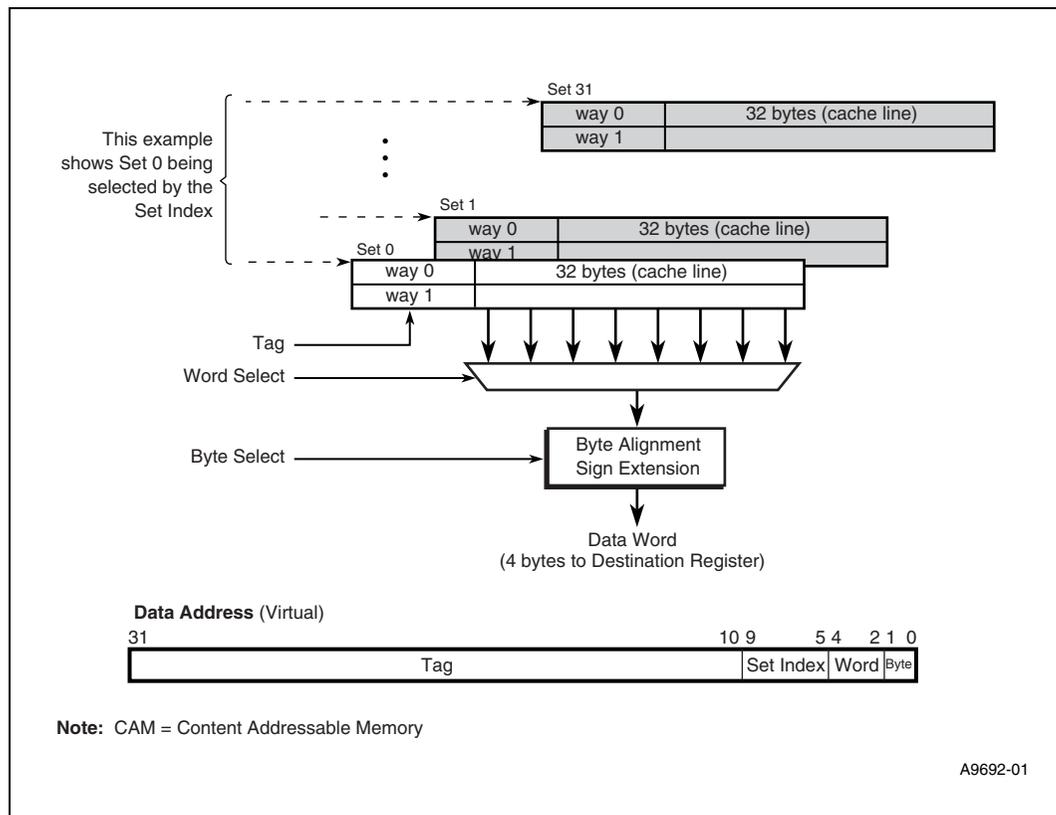
3.6.1.2 Mini-Data Cache Overview

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 16 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 16. Mini-Data Cache Organization



3.6.1.3 Write Buffer and Fill Buffer Overview

The 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 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 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. 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 XScale core will stall until an entry is available.
2. The 32-bytes of data can be returned back to the 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 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 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 8](#) shows code that enables the data and mini-data caches. Note that the MMU must be enabled to use the data cache.

Example 8. 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 9](#) shows how data cache can be cleaned.

Example 9. 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.

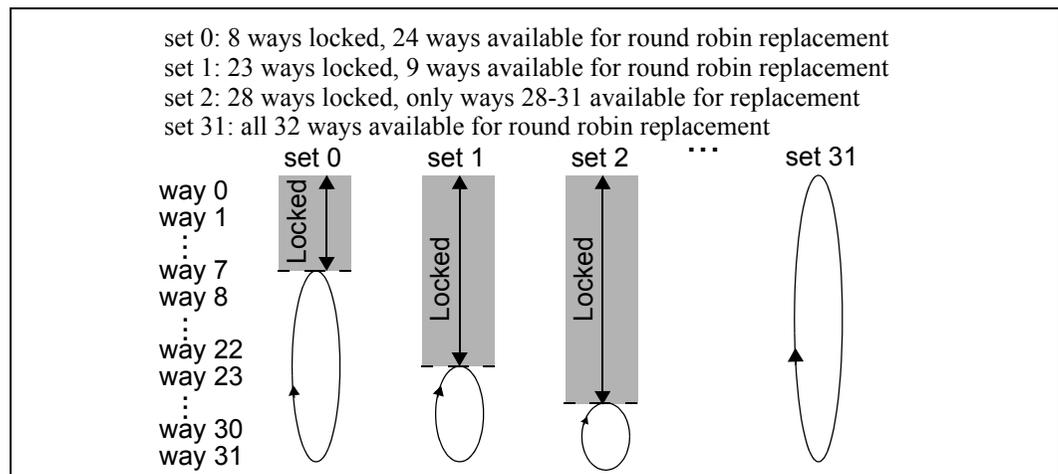
Refer to the *Intel® XScale® Core Developers Manual* for code examples.

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 way0 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 17** is an example of where lines of code may be locked into the cache along with how the round-robin pointer is affected.

Figure 17. Locked Line Effect on Round Robin Replacement



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 17](#).

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 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 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 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 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 Developer's Manual*.

3.8 Performance Monitoring

The XScale core hardware provides two 32-bit performance counters that allow two unique events to be monitored simultaneously. In addition, the 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 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. (Refer to the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*) Each counter has its own interrupt enable. The counters continue to monitor events even after an overflow occurs, until disabled by software.

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

To further augment performance monitoring, the 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 that 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 8 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 8. Performance Monitoring Events

| 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.) |
| 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 9. In this section, we call such an event combination a *mode*.

Table 9. 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.

Statistics derived from these two events are:

- 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 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 XScale core may be starved of the bus external to the 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 XScale core and what the memory access latency is for Data Cache miss requests to external memory. If the 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 XScale core is being starved of the bus external to the XScale core, which will effect the performance of the application running on the 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 XScale core may be starved of the bus external to the 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 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 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. 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 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 XScale core dynamically predicts the outcome of these instructions based on previous outcomes. [Table 10](#) 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 XScale core can execute the next instruction in the program flow in the cycle following the branch.

Table 10. 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. <ol style="list-style-type: none"> 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 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 10](#), 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 10](#) contains a code fragment and an example of computing latencies.

Example 10. Computing Latencies

```

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

[Table 11](#) 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 11](#), **UMLAL** starts to issue at cycle 0 and the **SUB** issues at cycle 5. thus the Result Latency is five.

Table 11. 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 12. 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 13. 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 14 | 4 + numbers in Table 14 |
| LDR PC, <> | 2 | 8 |
| LDM with PC in register list | 3 + numreg ^a | 10 + max (0, numreg-3) |

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

3.9.4.3 Data Processing Instruction Timings

Table 14. 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 ^a | Minimum Issue Latency | Minimum Result Latency ^a |
| 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 |

a. 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 15. Multiply Instruction Timings (Sheet 1 of 2)

| Mnemonic | Rs Value (Early Termination) | S-Bit Value | Minimum Issue Latency | Minimum Result Latency ^a | 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 |
| 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 |

Table 15. Multiply Instruction Timings (Sheet 2 of 2)

| Mnemonic | Rs Value (Early Termination) | S-Bit Value | Minimum Issue Latency | Minimum Result Latency ^a | Minimum Resource Latency (Throughput) |
|----------|------------------------------|-------------|-----------------------|-------------------------------------|---------------------------------------|
| 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 |

a. 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 16. 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 17. 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) ^a | 2 |

a. 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 18. 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 19. 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 20. 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 21. Load and Store Multiple Instruction Timings

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

a. 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 22. Semaphore Instruction Timings

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

3.9.4.9 Coprocessor Instructions

Table 23. CP15 Register Access Instruction Timings

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

Table 24. 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 25. SWI Instruction Timings

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

Table 26. 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 IXP2400 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 27. 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 28. 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}
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}
64-bit data 0x12345678 9ABCDE56 arranged as {78 56 34 12, 56 DE BC 9A}

Endianness for the IXP2400 processor can be divided into three major categories:

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

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

The XScale core may be used in either a little endian or big endian configuration. The configuration affects the entire system in which the XScale microarchitecture resides. 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 IXP2400 it is left to the software to interpret byte lanes in accordance with the endianness of the system. As shown in Figure 18, system byte lanes 0–3 are connected directly to the 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 XScale core, byte lane 1 (M[15:8]) of the system is connected to byte lane 1 (X[15:8]) of the XScale core and so on.

Interface operation of the XScale core and the rest of the IXP2400 can be divided into two parts:

- XScale core reading from the IXP2400
- XScale core writing to the IXP2400

3.10.1.1 Reads Initiated by Intel® XScale® Core

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 XScale core generates the byte_enable that corresponds to the proper byte lane as defined by the endianness setting. Table 29 summarizes byte enable generation for this mode.

Table 29. Byte Enable Generation by the Intel® XScale® Core for Byte Transfers in Little and Big Endian System

| 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 XScale core. Select signals for the mux are generated based on endian setting and ByteEnable generated by the XScale core as defined in Figure 18.

16-bit (Word) Read

When reading a word, the XScale core generates the byte_enable that corresponds to the proper byte lane as defined by the endianness setting. Figure 19 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 XScale core. The 2-to-1 mux steers Byte lane 1 or Byte lane 3 into Byte1 location of the read register inside the XScale core. The 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 XScale core as defined in Figure 18. Table 30 summarizes byte enable generation for this mode.

Figure 18. Byte Steering for Read and Byte Enable Generation by the Intel® XScale® Core

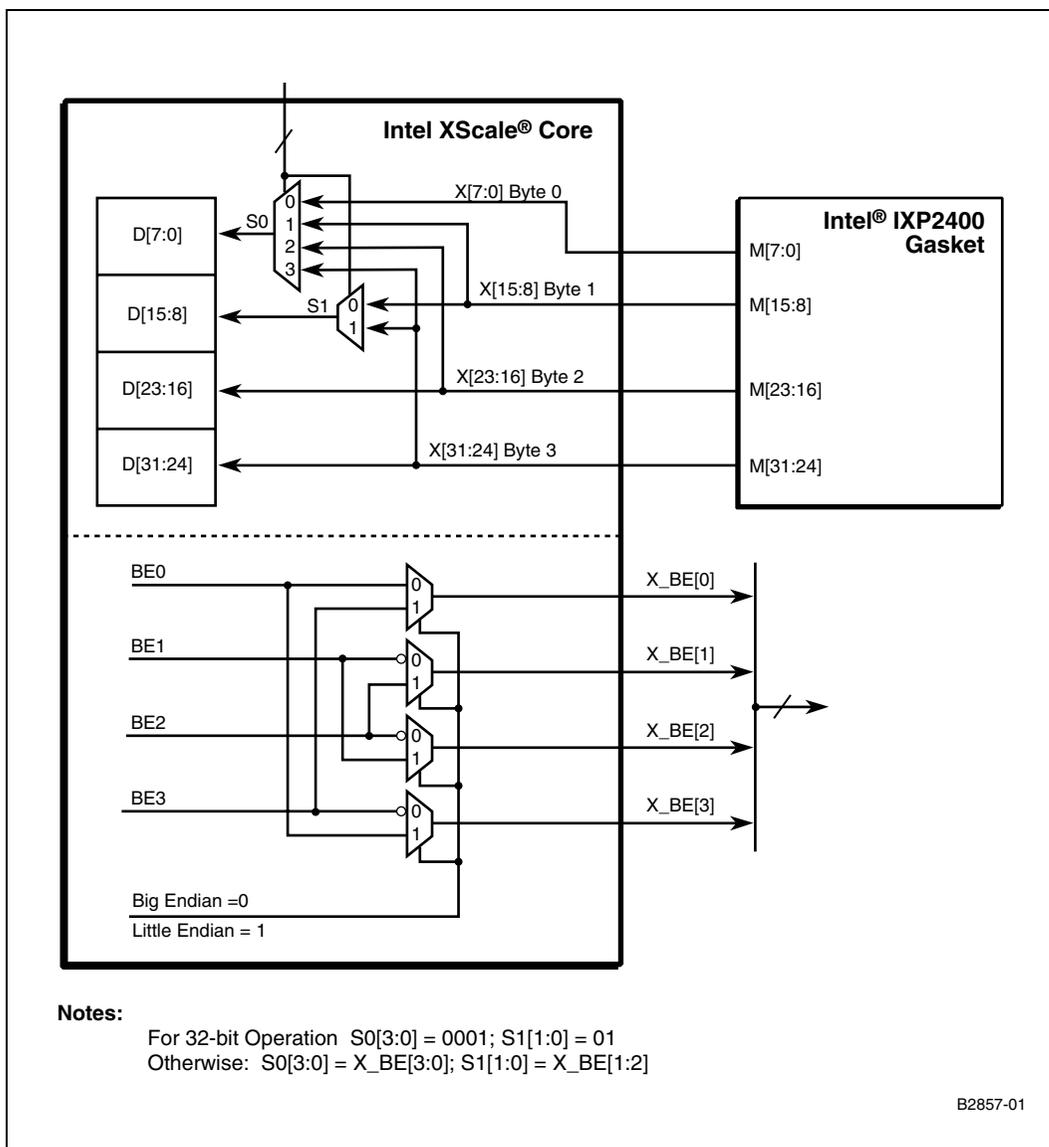


Table 30. Byte Enable Generation by the Intel® XScale® Core for 16-bit Data Transfer in Little and Big Endian Systems

| 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 XScale core's data bus gets into Byte 0 location of the read register inside XScale core, byte lane 1 from XScale core's data bus gets into Byte1 location of the read register inside XScale core and so on. It is up to the software to deal with byte location properly based on the endian setting.

3.10.1.2 The Intel® XScale® Core Writing to the IXP2400

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

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

Byte Write

When 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. XScale core register bits [7:0] always contain the byte to be written regardless of the B-bit setting. For example if the 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 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 31. Other byte lanes besides the one currently driven by the XScale core contain undefined data.

Table 31. Byte Enable Generation by the Intel® XScale® Core for Byte Write In Little and Big Endian System

| 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 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 XScale core does not allow a word write on an odd byte address. The XScale core register bits [15:0] always contain the word to be written regardless of the B-bit setting. For example if the 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 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 32. Other byte lanes besides the ones currently driven by the XScale core contain undefined data.

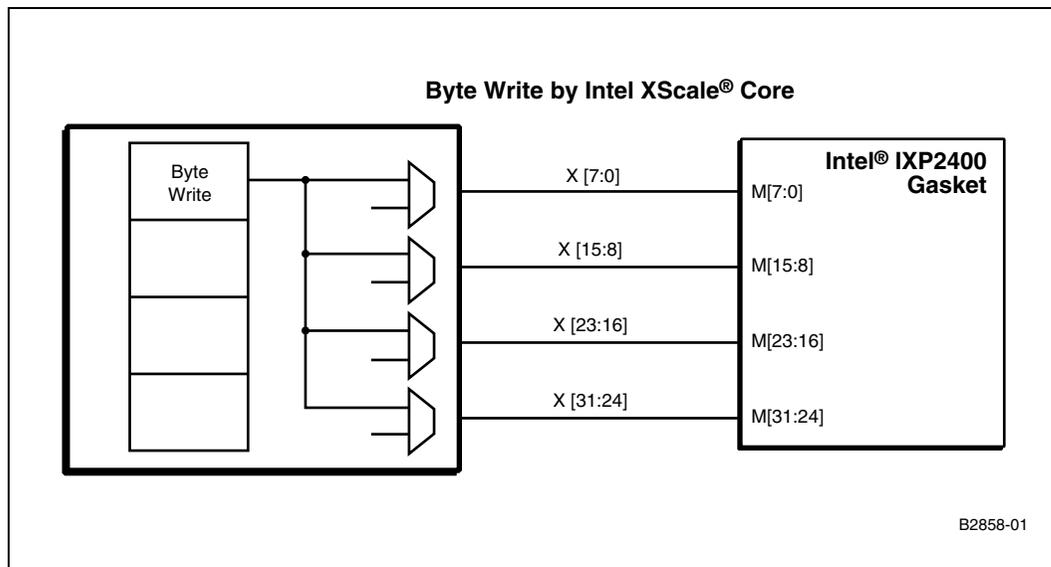
Table 32. Byte Enable Generation by the Intel® XScale® Core for Word Writes in Little-Endian and Big-Endian Systems

| 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

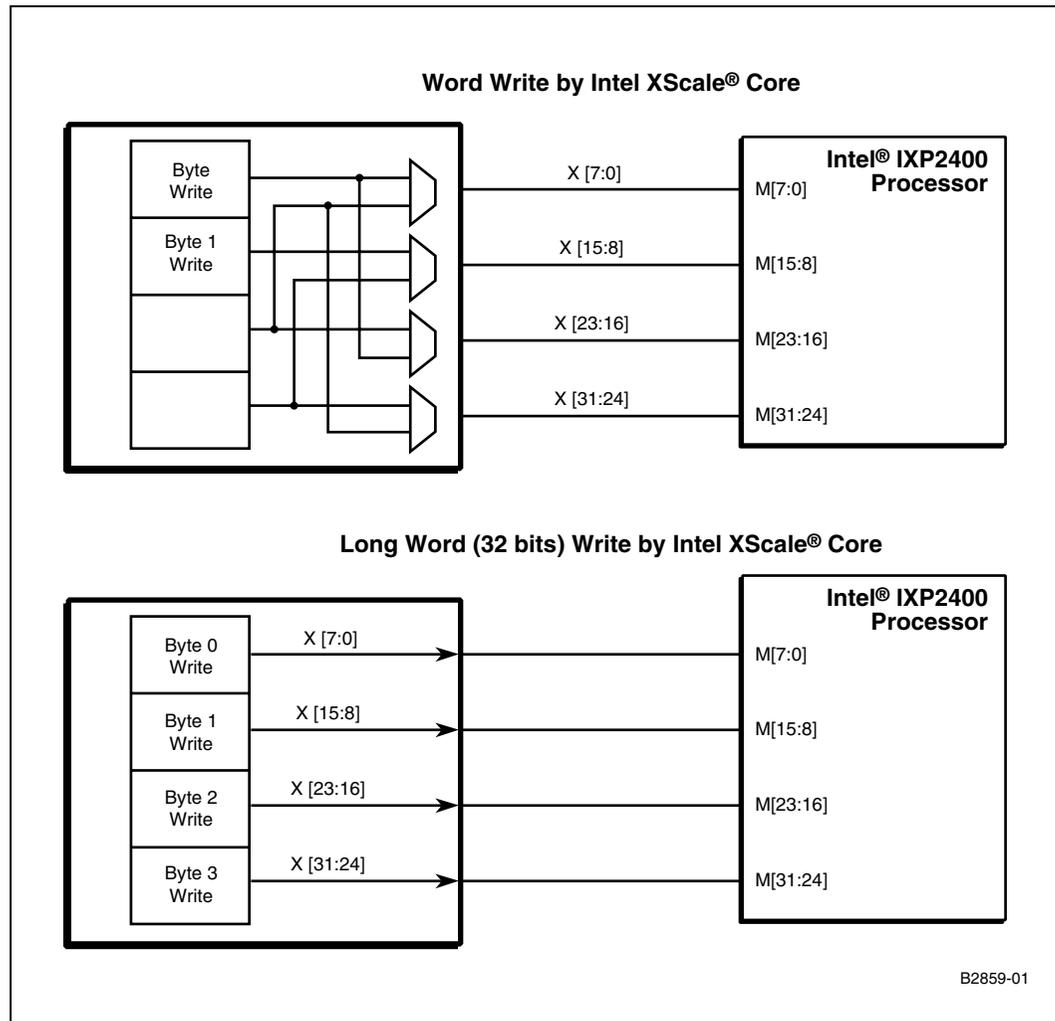
The long word to be written is put on the XScale core’s data bus with byte0 on X[7:0], byte1 on X[15:8], byte2 on X[23:16] and byte4 on X[31:24] (see Figure 19). All the byte enables are turned ON. A 32-bit long word write (0x12345678) by the XScale core to address 0x0000 irrespective of the endianness 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 19. Intel® XScale® Core Initiated Write to the IXP2400 Network Processor



B2858-01

Figure 19. Intel® XScale® Core Initiated Write to the IXP2400 Network Processor (Continued)



3.11 Intel® XScale® Gasket Unit

3.11.1 Overview

The XScale core uses the Core Memory Bus (CMB) to communicate with the functional blocks. The rest of the IXP2400 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 XScale core via the gasket internal bus. The CSR Access Proxy (CAP) is allowed to only do a set on these interrupt registers.

The XScale core includes Design for Test logic (DFT). The XScale core coprocessor bus is not used in the IXP2400 Network Processors, all accesses are only through the CMB.

3.11.2 Intel® XScale® Gasket Functional Description

3.11.2.1 Core Memory Bus to Command Push/Pull Conversion

The primary function of the XScale gasket unit is to translate commands initiated from the XScale core in the XScale command bus format, into the IXP2400 internal command format (Command Push/Pull format)

Table 33 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 33 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 33. 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 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 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 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 XScale 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 XScale core will issue a SWP command with an alias address if it needs data return. The 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® Access to SRAM Q-Array

The 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 34 to access the freelist. Each SRAM channel supports up to 64 lists, so there are 64 addresses per channel.

Table 34. IXP2400 Network Processor SRAM Q-Array Access Alias Addresses

| Channel ^a | Address Range |
|----------------------|---------------------------|
| 0 | 0xCC00 0100 – 0xCC00 01FC |
| 1 | 0xCC40 0100 – 0xCC40 01FC |
| 2 | 0xCC80 0100 – 0xCC80 01FC |
| 3 | 0xCCC0 0100 – 0xCCC0 01FC |

a. The IXP2400 has two SRAM Q-Array address ranges; channels 2 and 3 are reserved.

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 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

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 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 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 XScale core to setup a Hash access is as follows.

1. The 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 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 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 35](#).

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

Table 35. GCSR Address Map (0xd700 0000)

| Bits | Name | R/W | Description | Address Offset |
|--------|---------------|-----|---|--|
| [31:0] | XG_GCSR_HASH0 | R/W | Hash word 0 Write from XScale. 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 XScale. 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 XScale. Rd/Wr from CPP. | 0x28 : for 128bit Hash |
| [31:0] | XG_GCSR_HASH3 | R/W | Hash word 3 Write from XScale. Rd/Wr from CPP. | 0x2c : for 128bit Hash |
| [31:0] | XG_GCSR_CTR0 | R | [31:1] reserved. [0] hash valid flag. Read from XScale. 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 XScale debug breakpoint | 0x3c |

3.11.8 Interrupt

The XScale core CSR controller contains local CSR(s) and interrupts inputs from multiple sources. The diagram in [Figure 21](#) 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 XScale core's IRQ or FIQ inputs by multiple levels of enable registers.

Refer to [Figure 22](#).

- $\{\text{IRQ,FIQ}\}\text{Status} = (\text{RawStatus} \& \{\text{IRQ,FIQ}\}\text{Enable})$
- $\{\text{IRQ,FIQ}\}\text{ErrorStatus} = (\text{ErrorRawStatus} \& \{\text{IRQ,FIQ}\}\text{ErrorEnable})$
- $\{\text{IRQ,FIQ}\}\text{ThreadStatus}_{\$_\#} = (\{\text{IRQ,FIQ}\}\text{ThreadRawStatus}_{\$_\#} \& \{\text{IRQ,FIQ}\}\text{ThreadEnable}_{\$_\#})$

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 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 21. Flow Through the Intel® XScale® Core Interrupt Controller

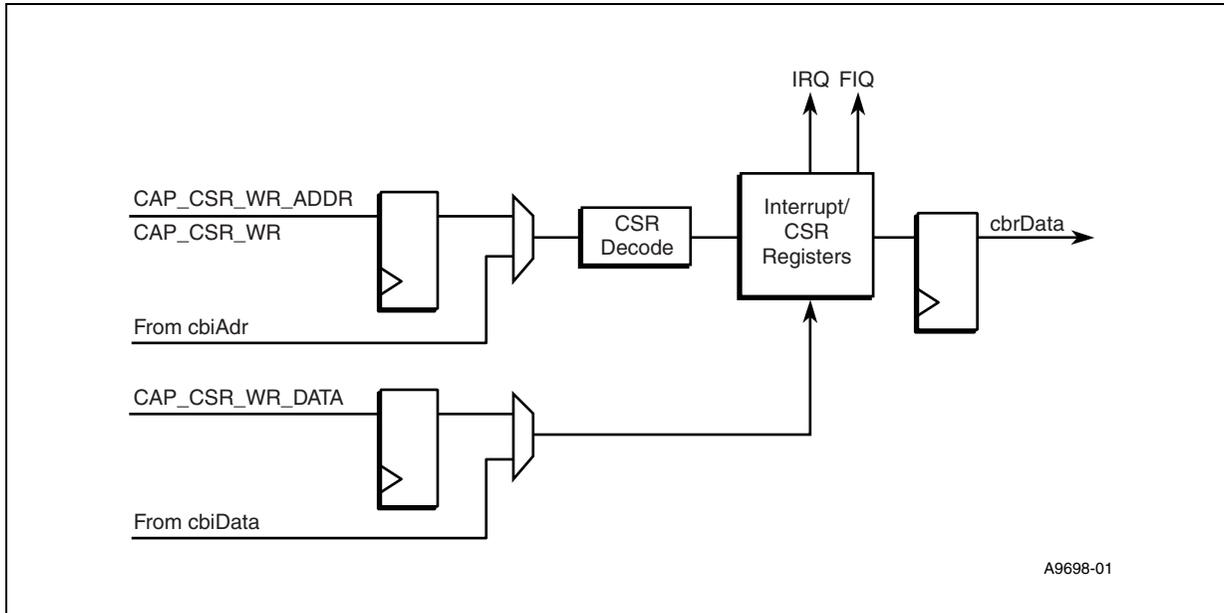
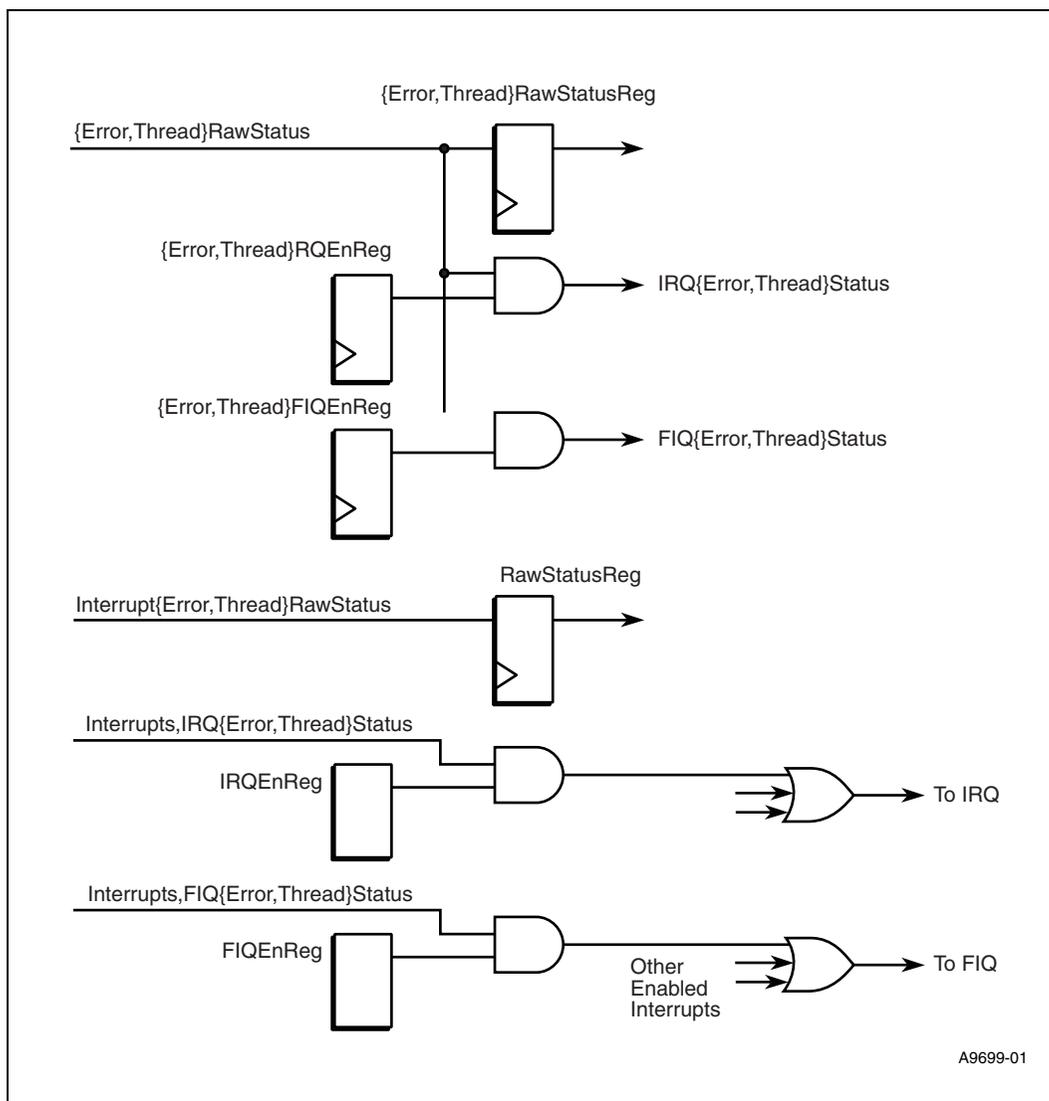


Figure 22. Interrupt Masking Block Diagram



3.12 Intel® XScale® Core Peripheral Interface

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

This section does not describe the 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 XScale point of view.

3.12.1 XPI Overview

Figure 23 shows the XPI location in the IXP2400 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 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, XPSH_APB_RDY_RAPBH, 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 23. XPI Interfaces (IXP2400 A0/A1)

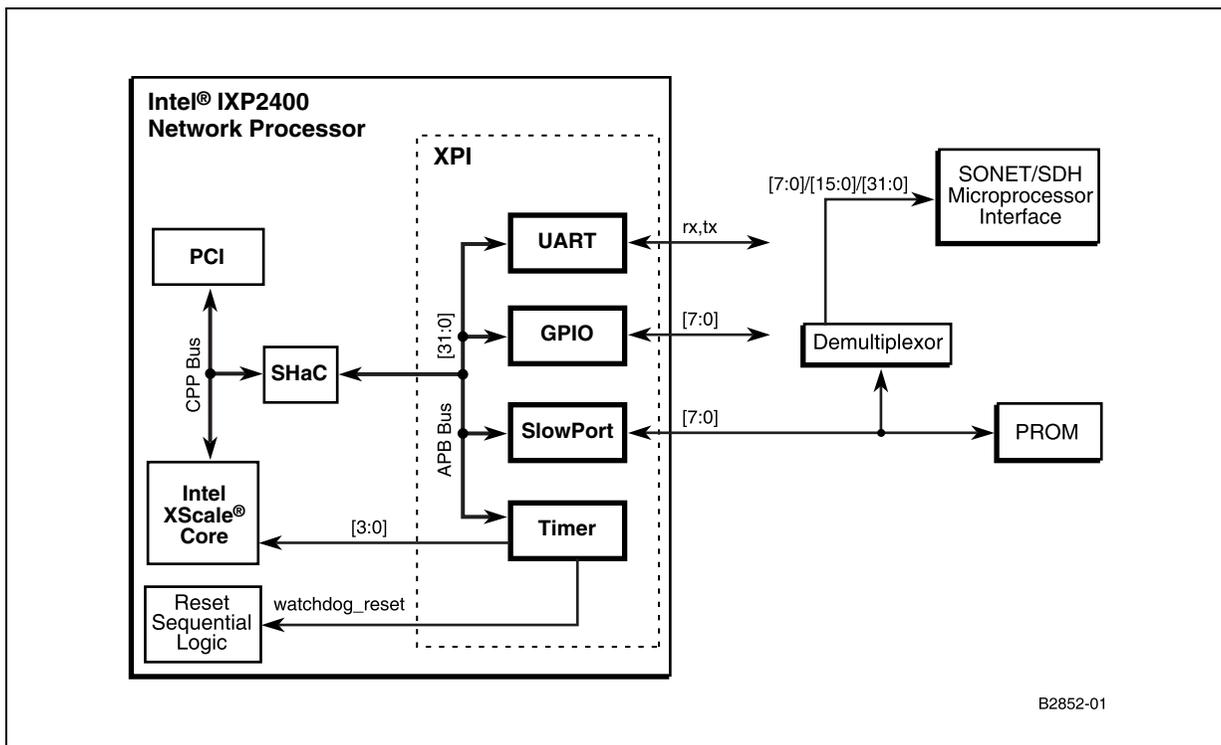
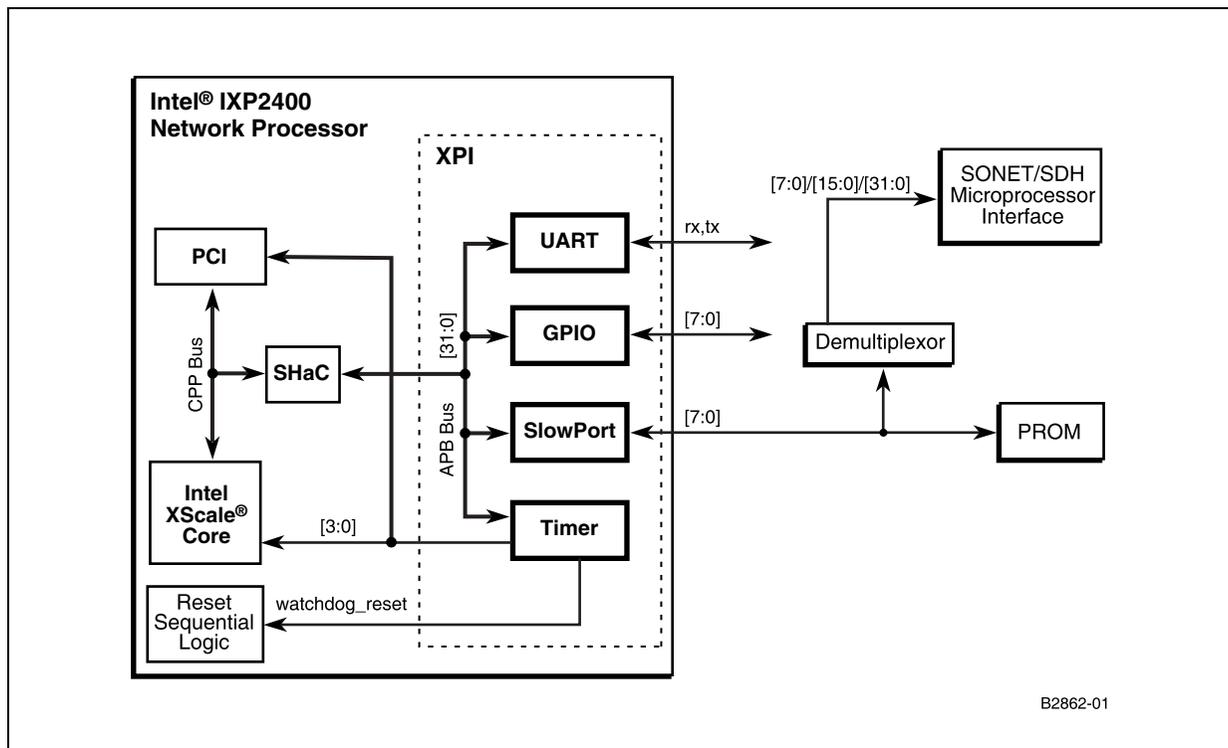


Figure 24. XPI Interfaces (IXP2400 B0)



B2862-01

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.

Figure 25 and Figure 26 displays one possible data flow issued by the external host on the PCI side. The external agent basically can request the access to the IXP2400 timer through the PCI bus.

Figure 27 displays the second possible data flow. This time the XScale issues a command to fetch the data from the boot PROM during the boot sequence. First XScale launches a fetch command to the SHaC. SHaC will launch a read transaction in the extended APB bus to XPI. XPI then access the external PROM device through the SlowPort bus. Data will be packed into 32-bit data and passed back the SHaC. SHaC will deliver these data back to the XScale core.

Figure 25. PCI/XPI Data Flows Example (IXP2400 A0/A1)

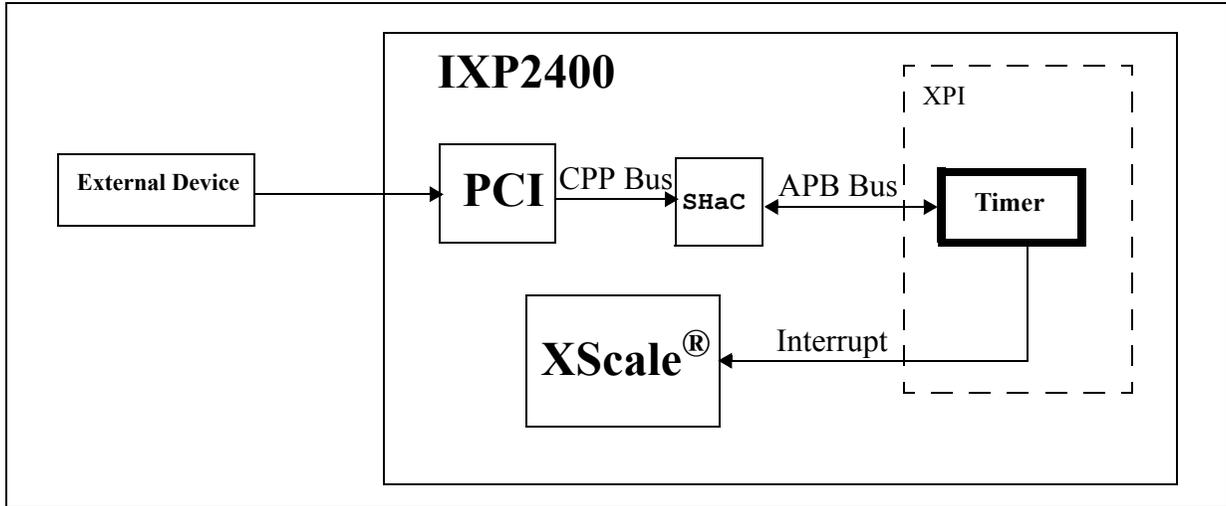


Figure 26. PCI/XPI Data Flows Example (IXP2400 B0)

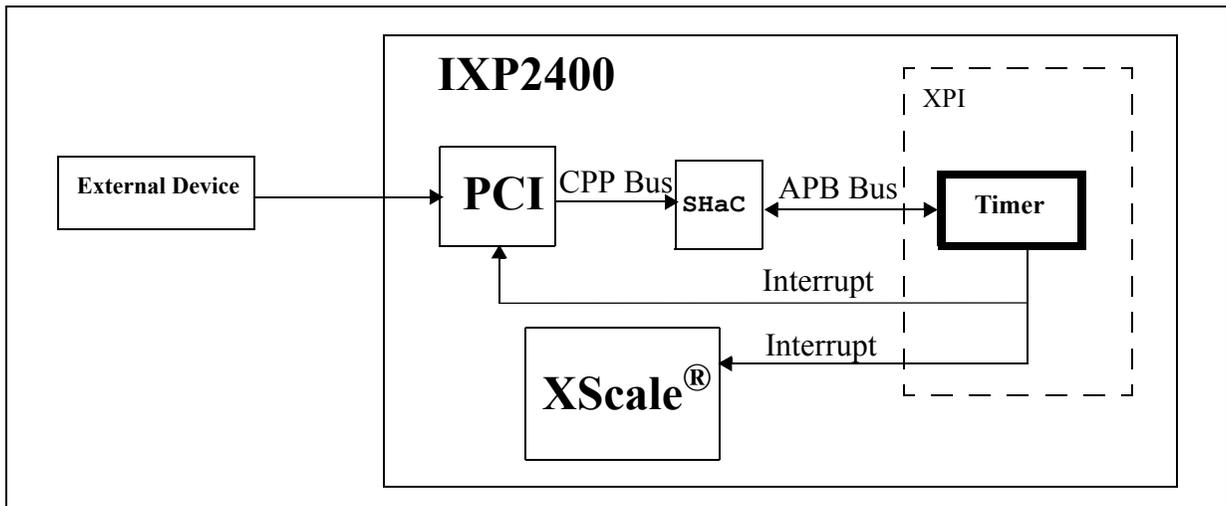
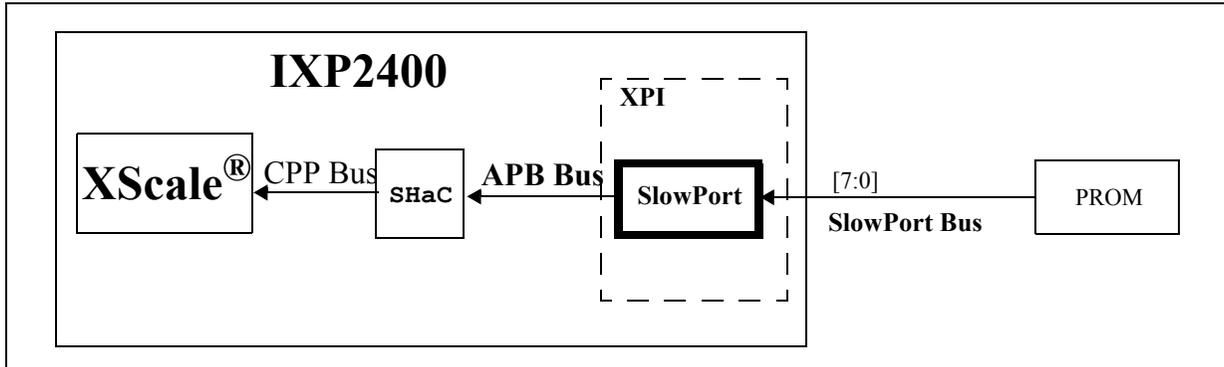


Figure 27. Second Example of Data Flows



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 36 for additional details on data alignment.

Table 36. 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 ^a | 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 |

a. 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 37 shows the address space assignment for XPI devices.

Table 37. Address Spaces for XPI Internal Devices

| Units | Starting Address Range | Ending Address |
|-----------------|------------------------|----------------|
| GPIO | 0xC0010000 | 0xC0010040 |
| Timer | 0xC0020000 | 0xC0020040 |
| UART | 0xC0030000 | 0xC003001C |
| PMU | 0xC0050000 | 0xC0050E00 |
| SlowPort CSR | 0xC0080000 | 0xC0080028 |
| SlowPort Device | 0xC4000000 | 0xC7FFFFFF |

3.12.2 UART Overview

The Universal Asynchronous Receiver/Transmitter (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).

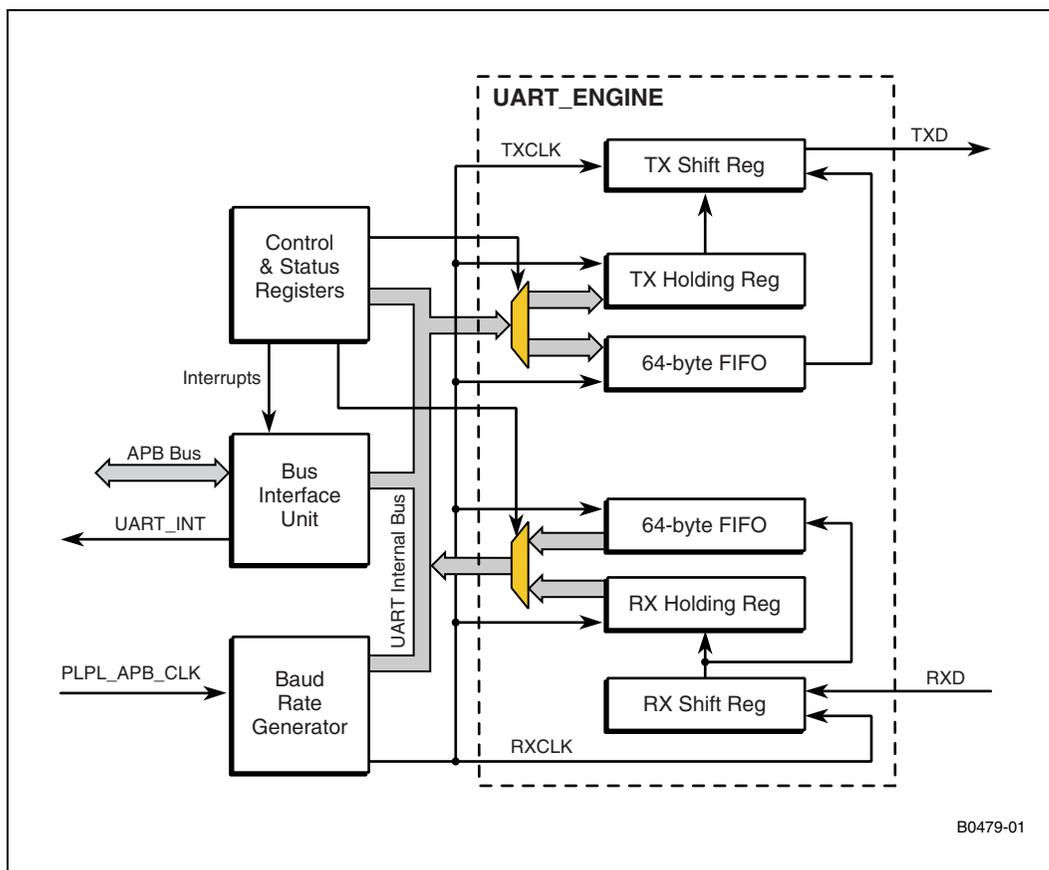
The serial ports can operate in either FIFO or non-FIFO mode. In FIFO mode, a 64-byte transmit FIFO holds data from the processor to be transmitted on the serial link and a 64-byte receive FIFO buffers data from the serial link until read by the processor.

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 two clocks: clock from baud rate generator for transmit operation and receive operation and clock from the XPI unit for register reads and writes.

Figure 28 shows the top level overview of UART. PLPL_APB_CLK is used in the Baud rate generator to produce the transmit CLK that is used in the transmit registers. The transmitters and receivers have shift registers, holding registers and the FIFO's as the main components.

Figure 28. UART Top Level Diagram^{a,b}



- a. For IXP2400 A0/A1, the UART FIFO control register can be programmed to cause an interrupt to XScale based on 1, 8, 16 or 32 entries.
- b. For IXP2400 B0, UART FIFO control register can be programmed to cause an interrupt to XScale or PCI based on 1, 8, 16 or 32 entries.

To prevent FIFO overflow, UART FIFO control register can be programmed to cause an interrupt and signal to the XScale core or PCI host.

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, and framing error simulation
- Fully prioritized interrupt system controls

3.12.2.1 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. The baud rate is calculated as follows:

$$\text{Baud Rate} = \text{System Clock} / (16 \times \text{Divisor})$$

The Divisor ranges from 2 to $(2^{16} - 1)$. For example, for a system clock of 50 MHz and baud rate of 115200 bps the divisor is 27. The divisor is not allowed to set to 0 and 1; otherwise, no internal clock is generated for operation of the UART unit.

Table 38. UART Register Map

| Abbreviation | Address [7:0] | Name | Description |
|--------------|---------------|--|---|
| UART_RBR | 0x00, DLAB=0 | UART Receive Buffer Register | It is used to buffer the received data. |
| UART_THR | 0x00, DLAB=0 | UART Transmit Holding Register | It is used to hold the transmitting data. |
| UART_DLRL | 0x00, DLAB=1 | UART Divisor Latch register Low | It is associated with UART_DLHR and used to control the baud rate together. |
| UART_DLRH | 0x04, DLAB=1 | UART Divisor Latch Register High | It is associated with UART_DLRL and used to control the baud rate together. |
| UART_IER | 0x04, DLAB=0 | UART Interrupt Enable Register | It is the interrupt enable register for all interrupt control. |
| UART_IIR | 0x08 | UART Interrupt Identification Register | This is a read only register and shares the same space as UART_FCR |
| UART_FCR | 0x08 | UART Fifo control register | This is a write only register. It is used to control the FIFO. |
| UART_LCR | 0x0C | UART Line Control Register | This is used to control the transmission line data format. |
| UART_LSR | 0x14 | UART Line Status Register | This stores the status of the previous transaction. |
| UART_SPR | 0x1C | UART scratch pad register | This allows the program to access for programming purpose. |

3.12.2.2 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.

3.12.2.2.1 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 timeout interrupt are enabled, a character timeout 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 timeout 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 timeout interrupt occurs, it is cleared and the timer is reset when the processor reads one character from the receiver FIFO. If a timeout interrupt has not occurred, the timeout timer is reset after a new character is received or after the processor reads the receiver FIFO.

Timeout interrupt is coupled with the FIFO interrupt trigger level/threshold level. If the data reach the threshold value, the timeout interrupt is prohibited. Therefore, no timeout interrupt occurs when the threshold value is set to 1 byte trigger. For 8-, 16-, and 32-byte trigger level, the timeout interrupt will occur if the data is left stranded in the 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.

3.12.2.2 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.

3.12.3 General Purpose I/O (GPIO)

The IXP2400 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 29](#) shows a GPIO functional diagram. The GPIO pin as seen can be programmed based on the configuration registers.

Figure 29. GPIO Functional Diagram

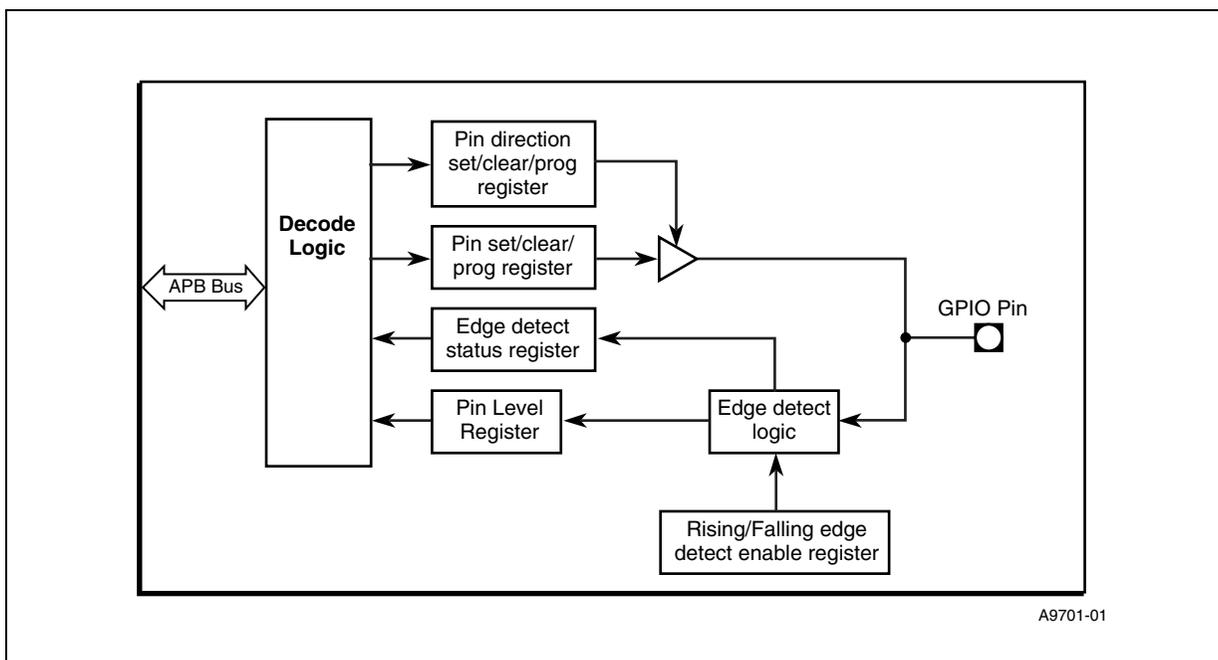


Table 39. GPIO Register Map

| Abbreviation | Address | Name | Description |
|--------------|---------|---|---|
| GPIO_PLR | 0x00 | GPIO Pin level register | This is used to determine the current value of a particular pin |
| GPIO_PDPR | 0x04 | GPIO Pin direction programmable register | This is to program a pin as an input or a output |
| GPIO_PDSR | 0x08 | GPIO Pin direction set register | This is to set a pin as an output |
| GPIO_PDCR | 0x0C | GPIO Pin direction clear register | This is to reset a pin as an input |
| GPIO_POPR | 0x10 | GPIO Output data programmable register | This is to program the output data register |
| GPIO_POSR | 0x14 | GPIO Output data set register | This is to set an output data register |
| GPIO_POCR | 0x18 | GPIO Output data clear register | This is to clear an output data register |
| GPIO_REDR | 0x1C | GPIO Rising edge detect enable register | This is to enable detects on rising edge |
| GPIO_FEDR | 0x20 | GPIO Falling edge detect enable register | This is to enable detects on falling edge |
| GPIO_EDSR | 0x24 | GPIO Edge detect status register | This is the logging of detected transitions |
| GPIO_LSHR | 0x28 | GPIO level sensitive high enable register | This is to enable detect on level sensitive high inputs. |

Table 39. GPIO Register Map (Continued)

| Abbreviation | Address | Name | Description |
|--------------|---------|--|--|
| GPIO_LSLR | 0x2C | GPIO level sensitive low enable register | This is to enable detect on level sensitive low inputs. |
| GPIO_LDSR | 0x30 | GPIO level detect status register | This is to log the logic level of inputs. |
| GPIO_INER | 0x34 | GPIO Interrupt Enable register | This is to enable the interrupt generation. |
| GPIO_INSR | 0x38 | GPIO Interrupt Set register | This is to set the interrupt enable register. |
| GPIO_INCR | 0x3C | GPIO Interrupt Reset register | This is to reset the interrupt enable register. |
| GPIO_INST | 0x40 | GPIO Interrupt Status Register | This is to capture the interrupts occurred to the corresponding pin by the external devices. |

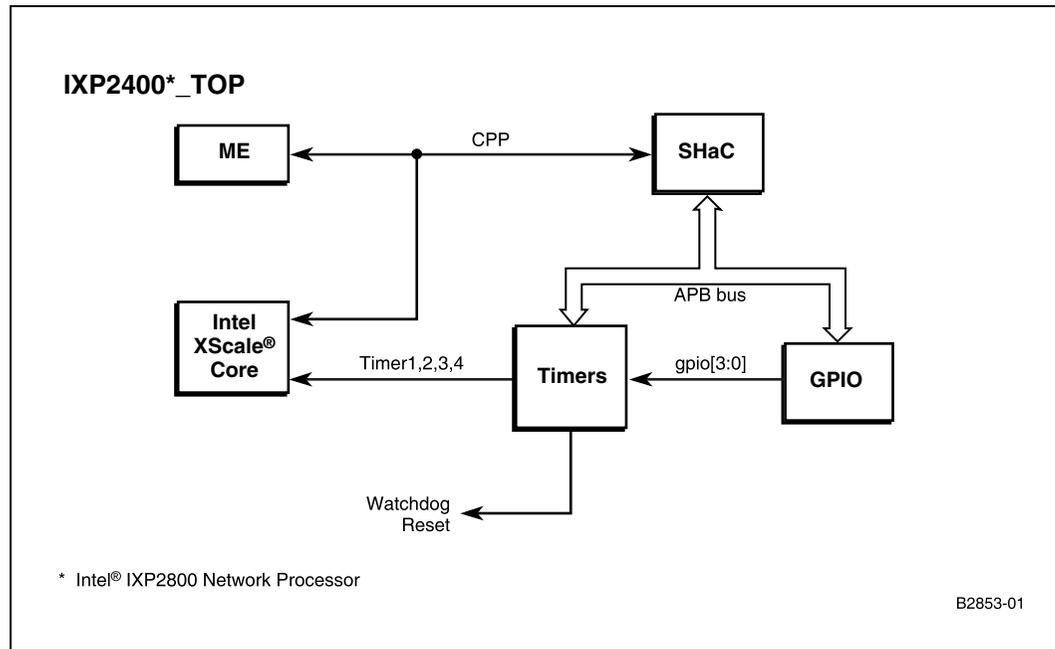
3.12.4 Timers

The IXP2400 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 30 shows the timer control unit interfacing with other functional blocks.

Figure 30. Timer Control Unit Interfacing Diagram



3.12.4.1 Timer Operation

Each timer consists of a 32-bit counter.

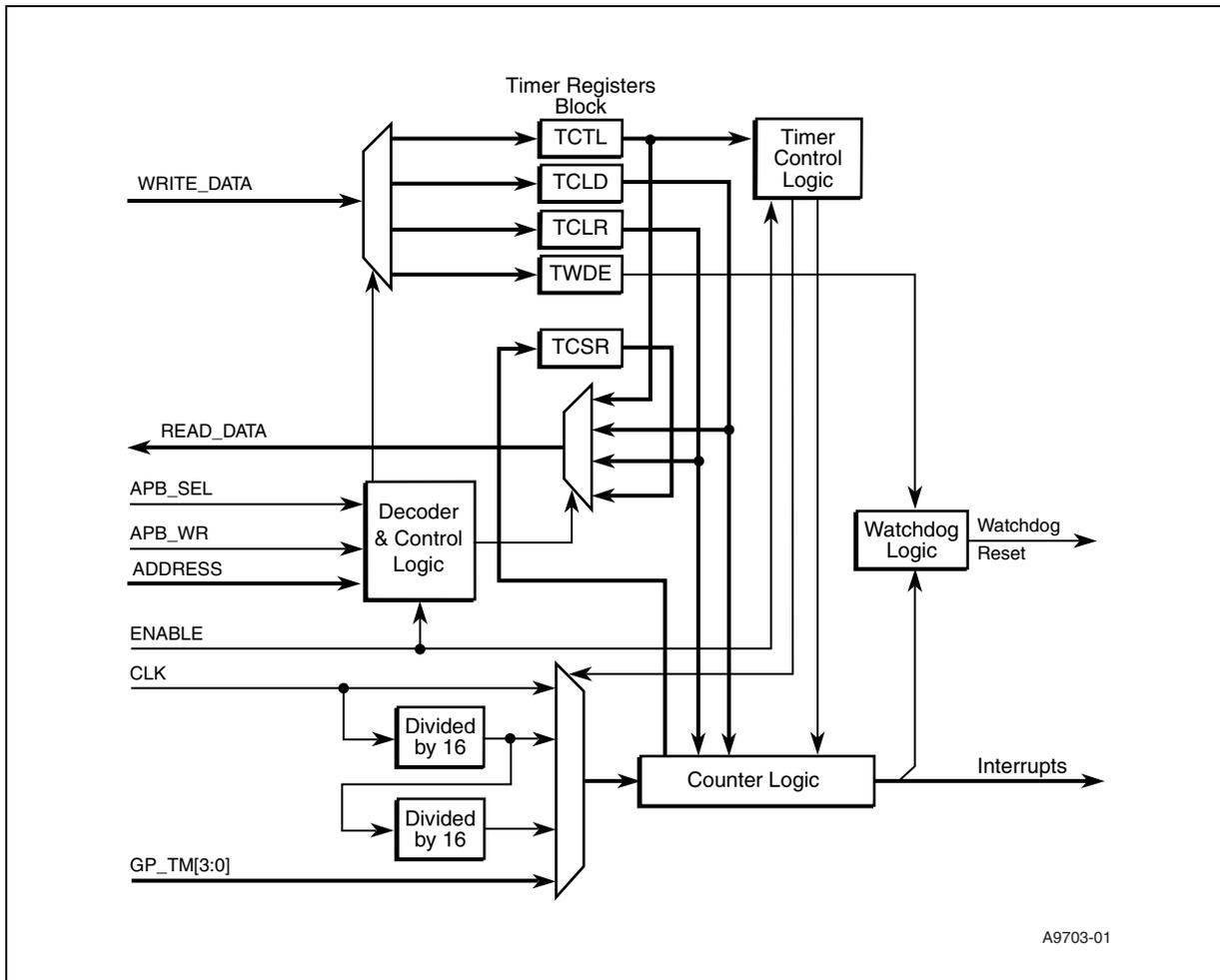
By default, the timer counter load register (TCLD) is set to 0xFFFFFFFF. The timer will count down from 0xFFFFFFFF to zero, then wrap back to 0xFFFFFFFF and continue to decrement if the TCLD is not programmed to any value. If a different value is programmed in the TCLD, then the counter will load this value every time it counts down to zero.

An interrupt is issued to the 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 signals are synchronized within the timer-clock domain before driving the counter.

Figure 31 shows the Timer Internal logic.

Figure 31. Timer Internal Logic Diagram



A9703-01

Table 40. Timer Register Map

| Name | Abbreviation | Address | Description |
|---------------------------------|--------------|---------|---|
| TIMER CONTROL registers | T1_CTL | 0x00 | This is used to determine the timer functions, mode, activation |
| | T2_CTL | 0x04 | |
| | T3_CTL | 0x08 | |
| | T4_CTL | 0x0C | |
| TIMER COUNTER LOADING registers | T1_CLD | 0x10 | These registers store the initial values for the timer counters. Writing to a register causes the timer to reload with its initial value. |
| | T2_CLD | 0x14 | |
| | T3_CLD | 0x18 | |
| | T4_CLD | 0x1C | |

Table 40. Timer Register Map (Continued)

| Name | Abbreviation | Address | Description |
|---|--------------------------------------|---------------------------------|---|
| TIMER COUNTER STATUS register | T1_CSR T2_CSR T3_CSR T4_CSR | 0x20 0x24 0x28 0x2C | This is to store the current counter values. |
| TIMER COUNTER CLEAR registers | T1_CLR T2_CLR T3_CLR T4_CLR | 0x30, 0x34, 0x38, 0x3C | Any write to these registers clear the associated timer interrupts. |
| TIMER WATCHDOG ENABLE register | TWDE | 0x40 | This is to enable the timer 4 to be a watchdog timer. |

3.12.5 SlowPort Unit

The IXP2400 Network Processor SlowPort Unit supports basic PROM access and 8-, 16-, and 32-bit microprocessor device access. It allows a master, (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[24:0] 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.

3.12.5.1 PROM Device Support

For all the Flash Memory access, only 8-bit devices are supported. APB write transactions are assumed to be eight 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 41](#):

Table 41. 8-bit Flash Memory Device Density

| Vendor | Part Number | Size |
|--------|-------------|------|
| Intel | 28F128J3A | 16MB |
| Intel | 28F640J3A | 8MB |
| Intel | 28F320J3A | 4MB |

3.12.5.2 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 42.

Table 42. SONET/SDH Devices

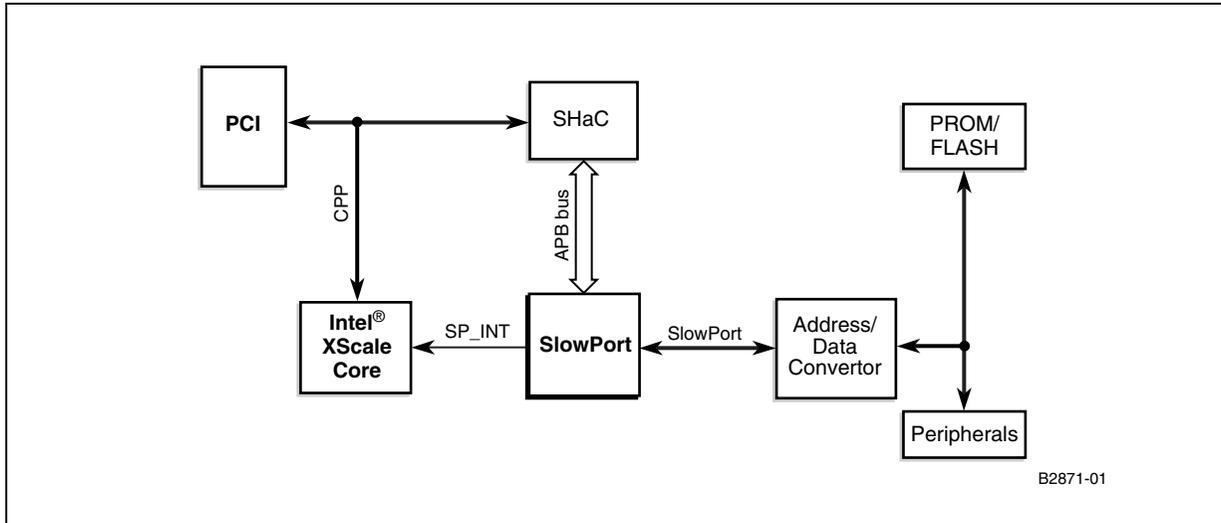
| Vendor | Part Number | μ P Interface | SP_PCR register Setting | DW Setting in SP_ADC register |
|------------|--------------------|-------------------|-------------------------|-------------------------------|
| PMC-Sierra | PM3386 | 16 bits | 0x3 | 0x1 |
| PMC-Sierra | PM5345 | 8 bits | 0x2 | 0x0 |
| 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 ^a | 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 |

a. 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.

3.12.5.3 SlowPort Unit Interfaces

Figure 32 shows the SlowPort Unit interface diagram.

Figure 32. SlowPort Unit Interface Diagram



3.12.5.3.1 Endianness

Little-endianness is supported in the SlowPort. Therefore, the lowest address byte should be placed in the byte lane 0, D[7:0].

Table 43. Byte Address

| Byte Lane | Address | Data bus |
|-----------|---------|----------|
| 0 | 00 | [7:0] |
| 1 | 01 | [15:8] |
| 2 | 10 | [23:16] |
| 3 | 11 | [31:24] |

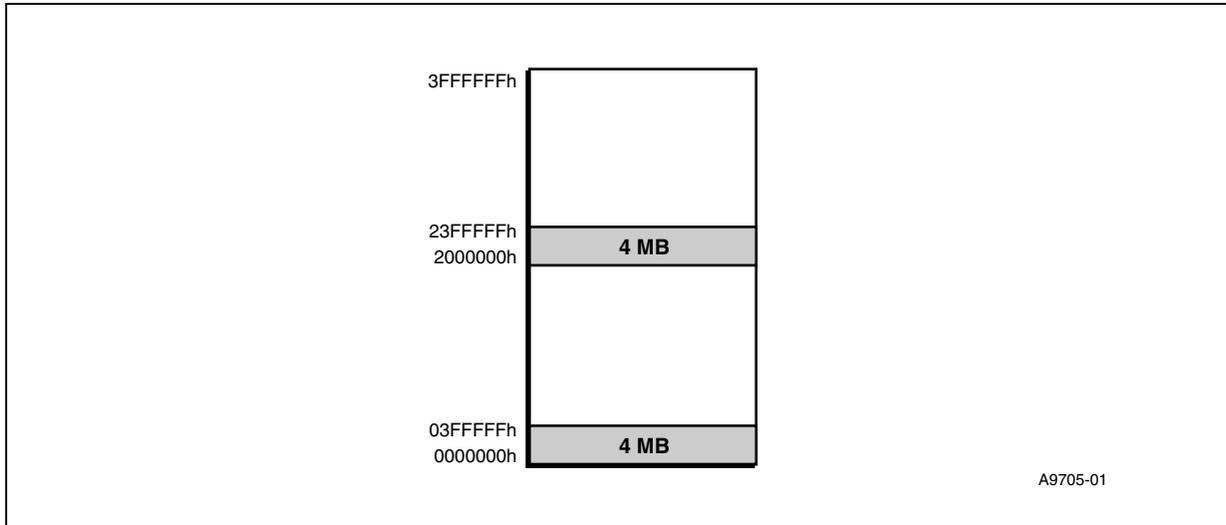
3.12.5.3.2 Byte Ordering

During byte write, the data should be placed according the Table 43 above. APB controller should place first byte in d[7:0], second byte in d[15:8] and so on. During half-word write, APB should place the first half word in d[15:0] and the second, in d[31:16]. However, during the byte read, the SlowPort should duplicate the byte into four byte lanes. Similar for the half word read, it duplicates twice and places it on both upper, d[31:16], and lower half word lanes, d[15:0]. For the word read, it is similar to the write case with the lowest byte placed in the lowest byte lane, second in the second byte lane, and so on.

3.12.5.4 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 be 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 33](#) 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 33. An Example of Address Space Hole Diagram



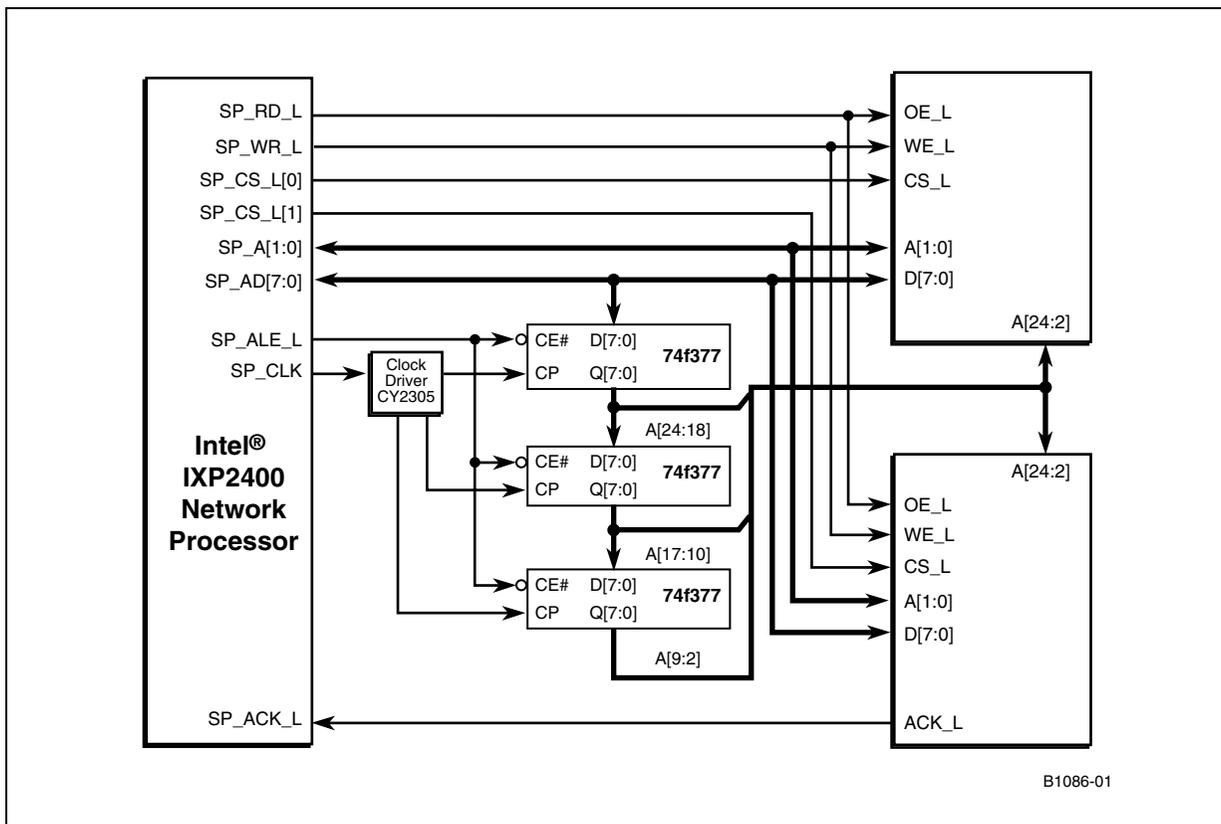
3.12.5.5 SlowPort Interfacing Topology

[Figure 34](#) 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 34](#). 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 34. SlowPort Example Application Topology



3.12.5.6 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.

3.12.5.6.1 Mode 0 Single Write Transfer for Fixed-Timed Device

Figure 35, shows the single write transfer for a fixed-timed device with the CSR programmed to a value of setup=4, pulse width=4, and hold=2, followed by another read transfer.

Figure 35. Mode 0 Single Write Transfer for a Fixed-Timed Device (IXP2400 A0/A1)

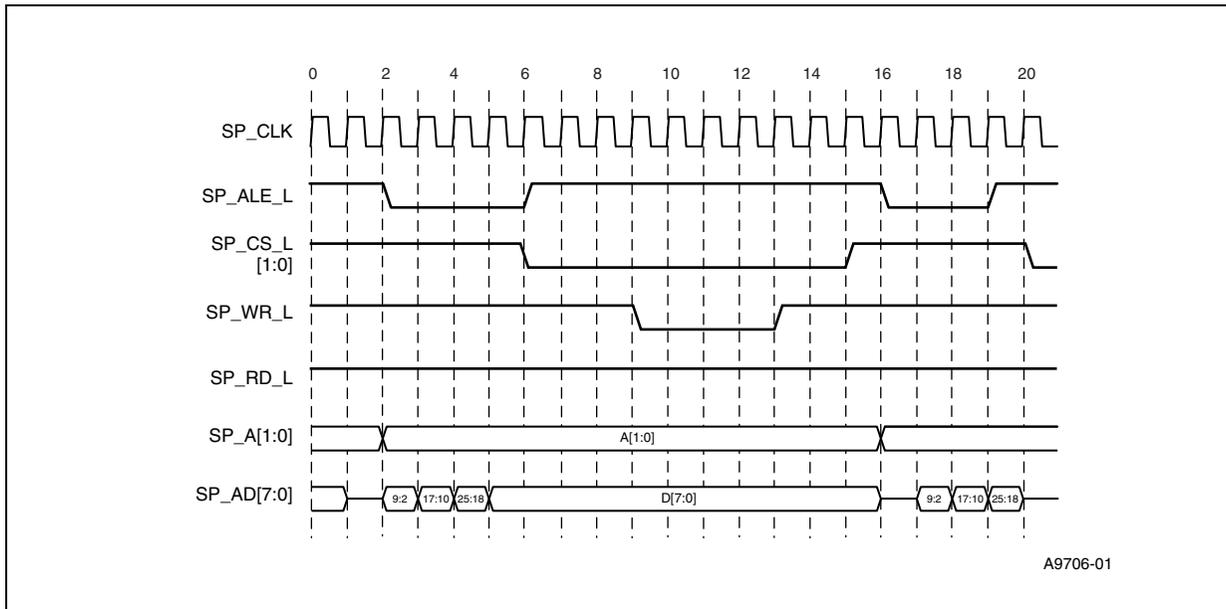
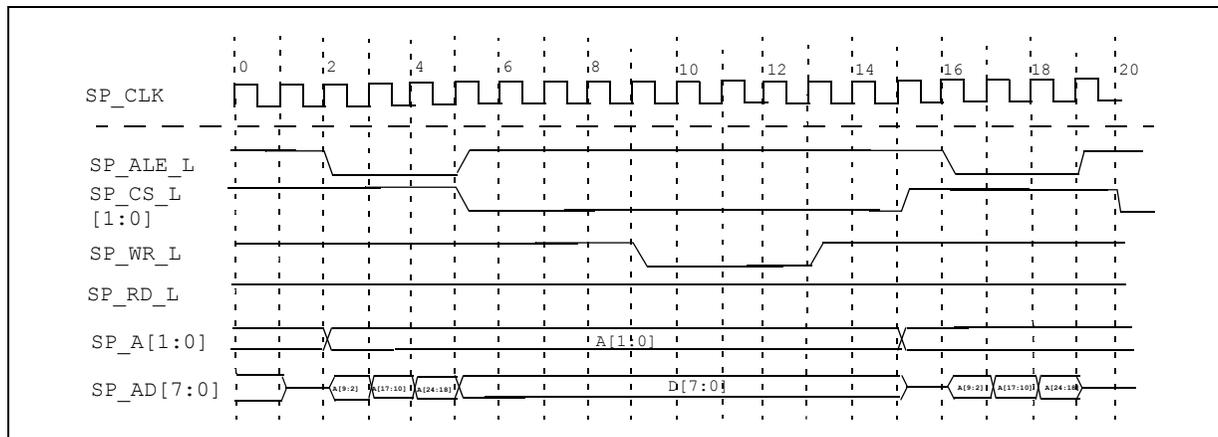


Figure 36. Mode 0 Single Write Transfer for a Fixed-Timed Device (IXP2400 B0)



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.

3.12.5.6.2 Mode 0 Single Write Transfer for a Self-timing Device

Figure 37 depicts the single write transfer for a self-timing device with the CSR programmed to setup=4, pulse width=0, and hold=3. Similarly, a read transaction is attached behind.

Figure 37. Mode 0 Single Write Transfer for a Self-Timing Device (IXP2400 A0/A1)

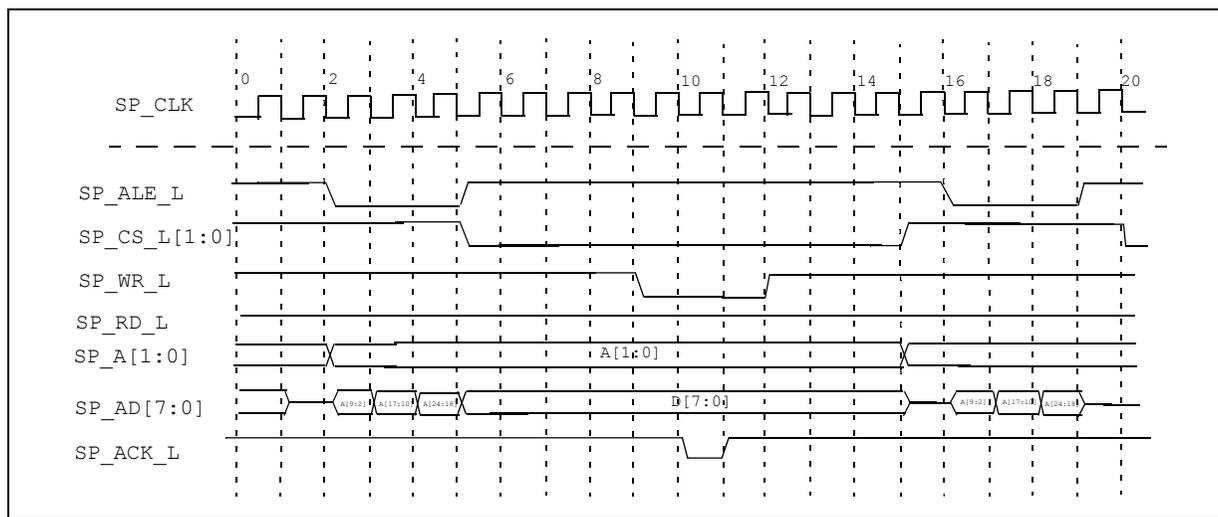
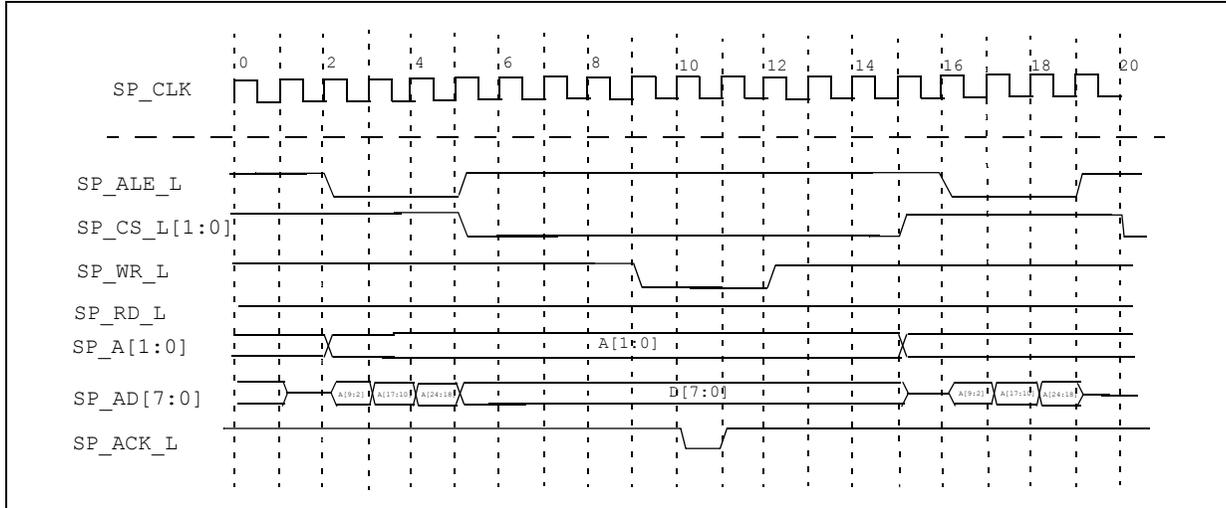


Figure 38. Mode 0 Single Write Transfer for a Self-Timing Device (IXP2400 B0)



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.

3.12.5.6.3 Mode 0 Single Read Transfer for Fixed-timed Device

Figure 39 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 39. Mode 0 Single Read Transfer for a Fixed-Timed Device (IXP2400 A0/A1)

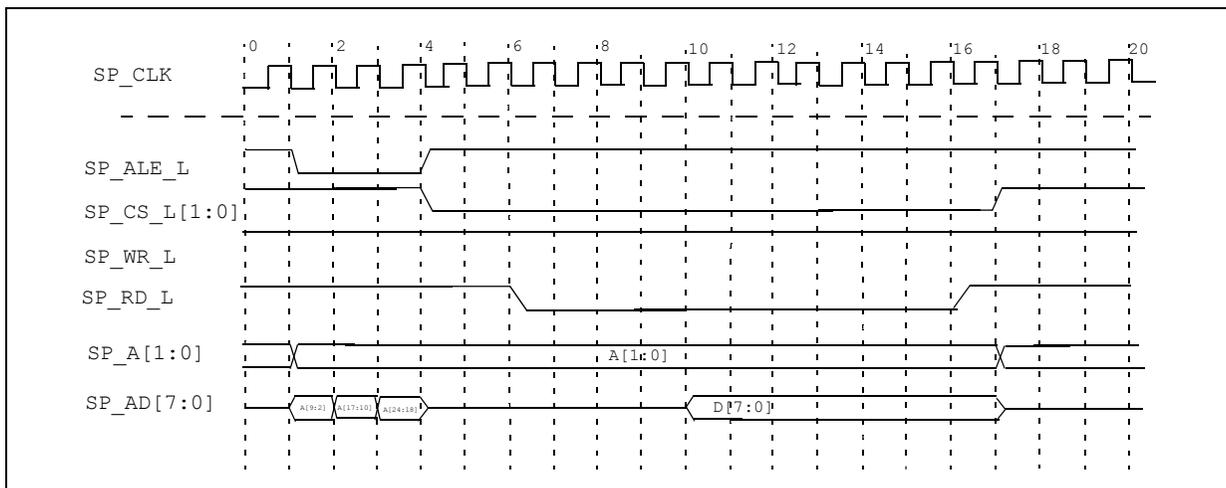
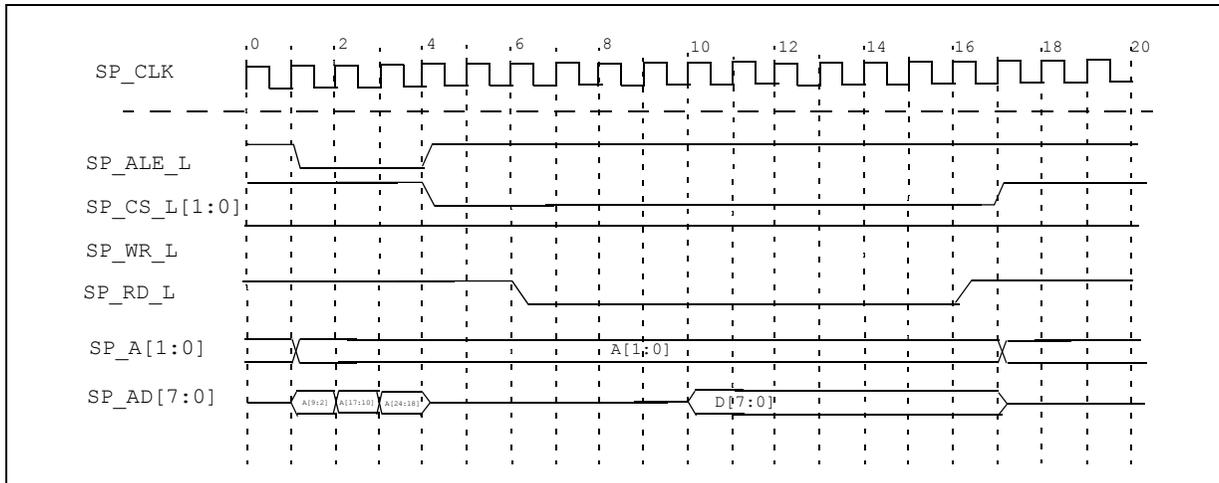


Figure 40. Mode 0 Single Read Transfer for a Fixed-Timed Device (IXP2400 B0)



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.

3.12.5.6.4 Single Read Transfer for a Self-timing Device

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=1.

Figure 41. Mode 0 Single Read Transfer for a Self-Timing Device (IXP2400 A0/A1)

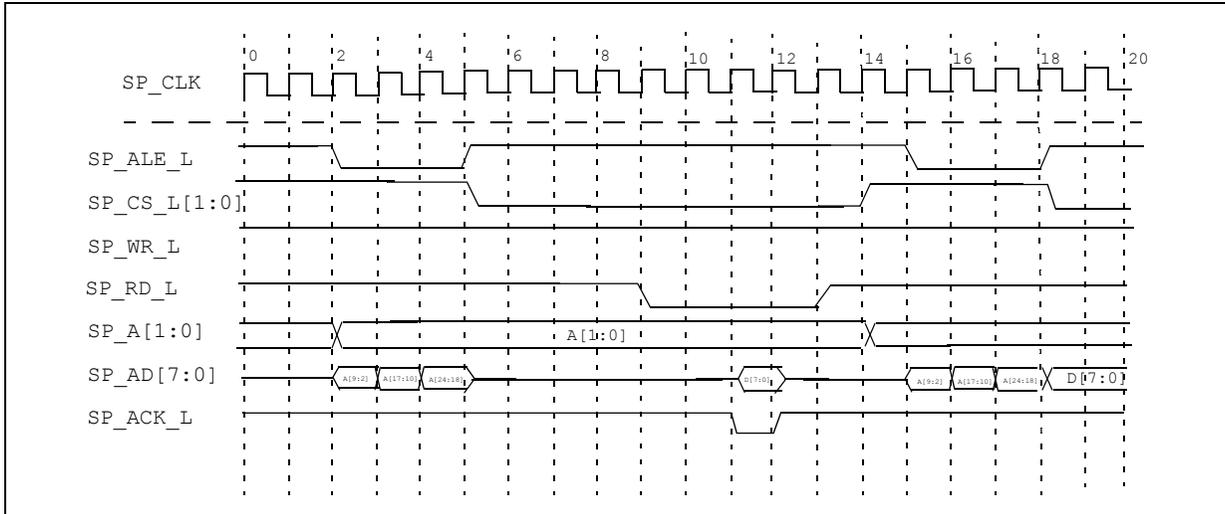
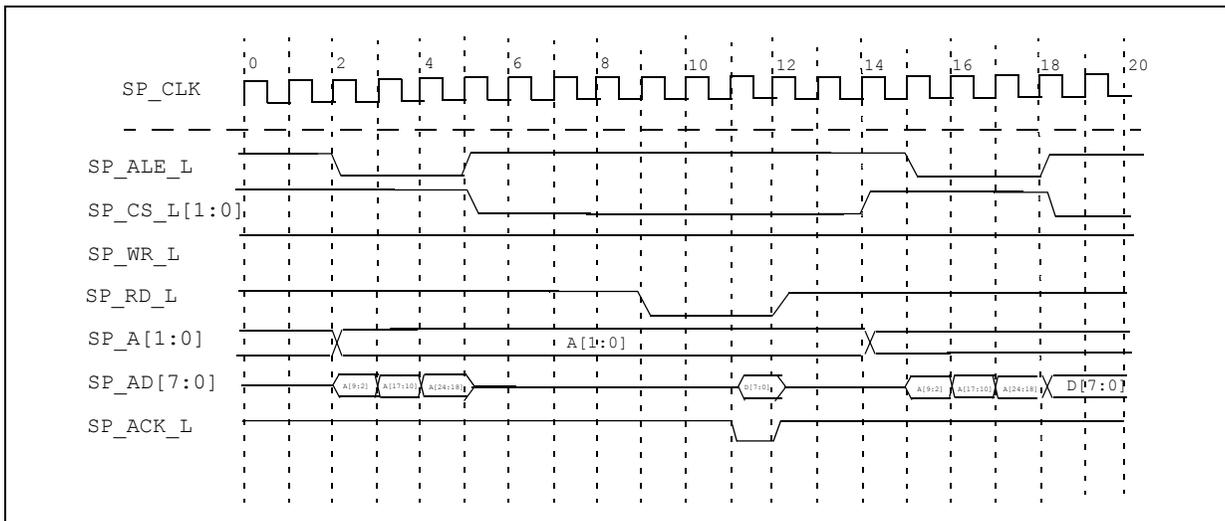


Figure 42. Mode 0 Single Read Transfer for a Self-Timing Device (IXP2400 B0)



The only difference for self-timed mode is in the SP_ACK_L signal. It has a dominant effect on the length of the transaction cycle or it overrides the value in the timing control register. A time-out counter is set to 255. The SP_ACK_L should arrive before the time-out counter counts down to zero. Similarly to the single write for self-timing device, an interrupt is launched for the time-out event and the time-out register is updated. In this case, the data will be sampled at clock cycle 12.

3.12.5.7 SONET/SDH Microprocessor Access Support

In order to support the SONET/SDH Microprocessor Interface, extra logic is added into this unit. Here we consider three SONET/SDH available components, including the Lucent TDAT042G5, PMC-Sierra PM5351, Intel, and AMCC SONET/SDH devices.

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.

3.12.5.7.1 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

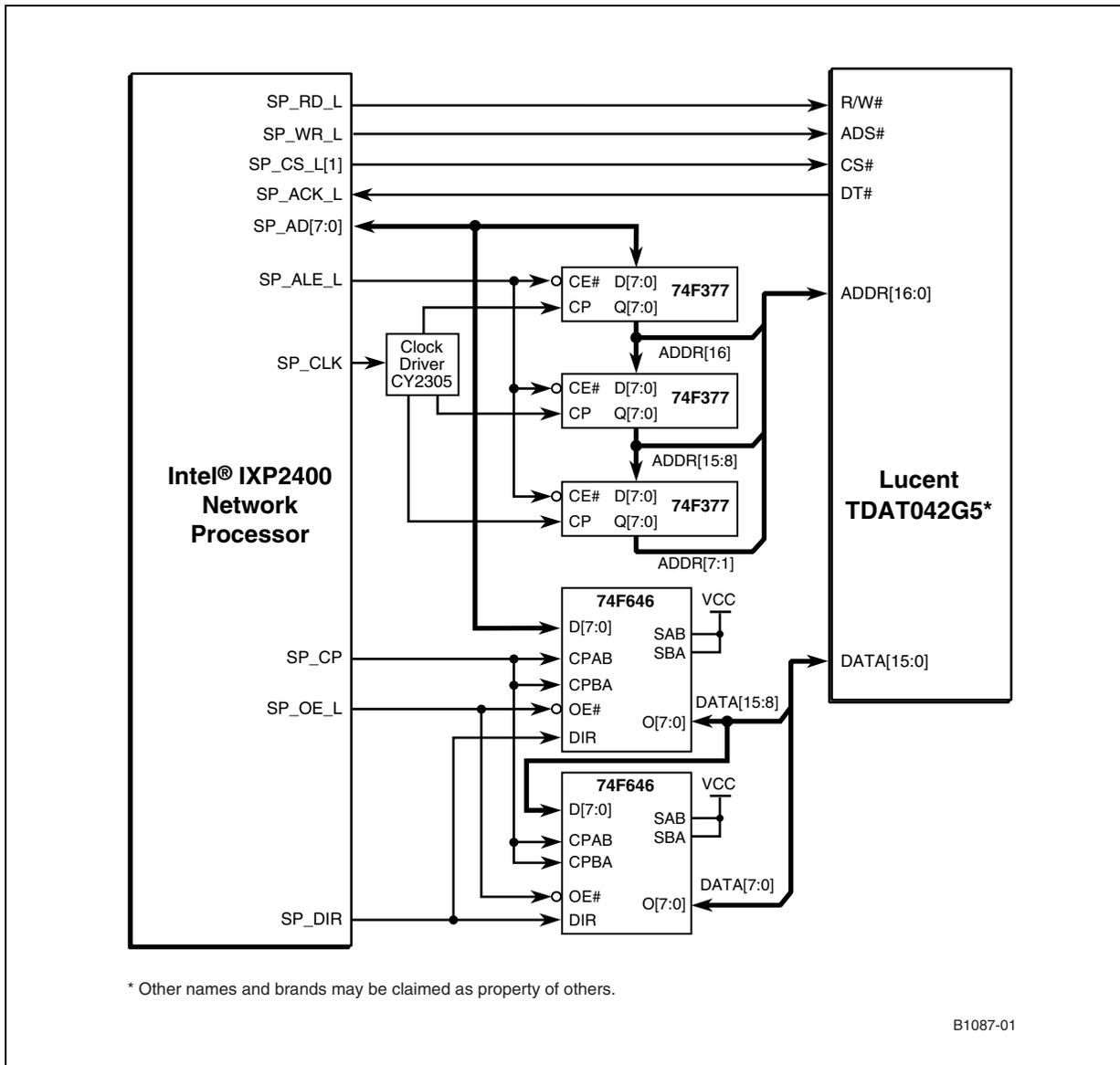
Figure 43 shows a solution for the 16-bit microprocessor interface. This solution bridges the Lucent TDAT042G5 SONET/SDH 16-bit interface. From Figure 43, we observe that the control pins SP_RD_L and SP_WR_L are converted to R/\overline{W} and \overline{ADS} . The \overline{CS} and \overline{DT} are still compatible 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 \overline{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 43. An Interface Topology with Lucent TDAT042G5 SONET/SDH



16-bit Microprocessor Write Interface Protocol

Figure 44 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} , 5 clock cycle for DT delay, and 1 clock cycle for \overline{ADS} . They are intervened with two idle cycles.

From Figure 44, 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 44. Mode 1 Single Write Transfer for Lucent TDAT042G5
 Device (IXP2400 A0/A1)**

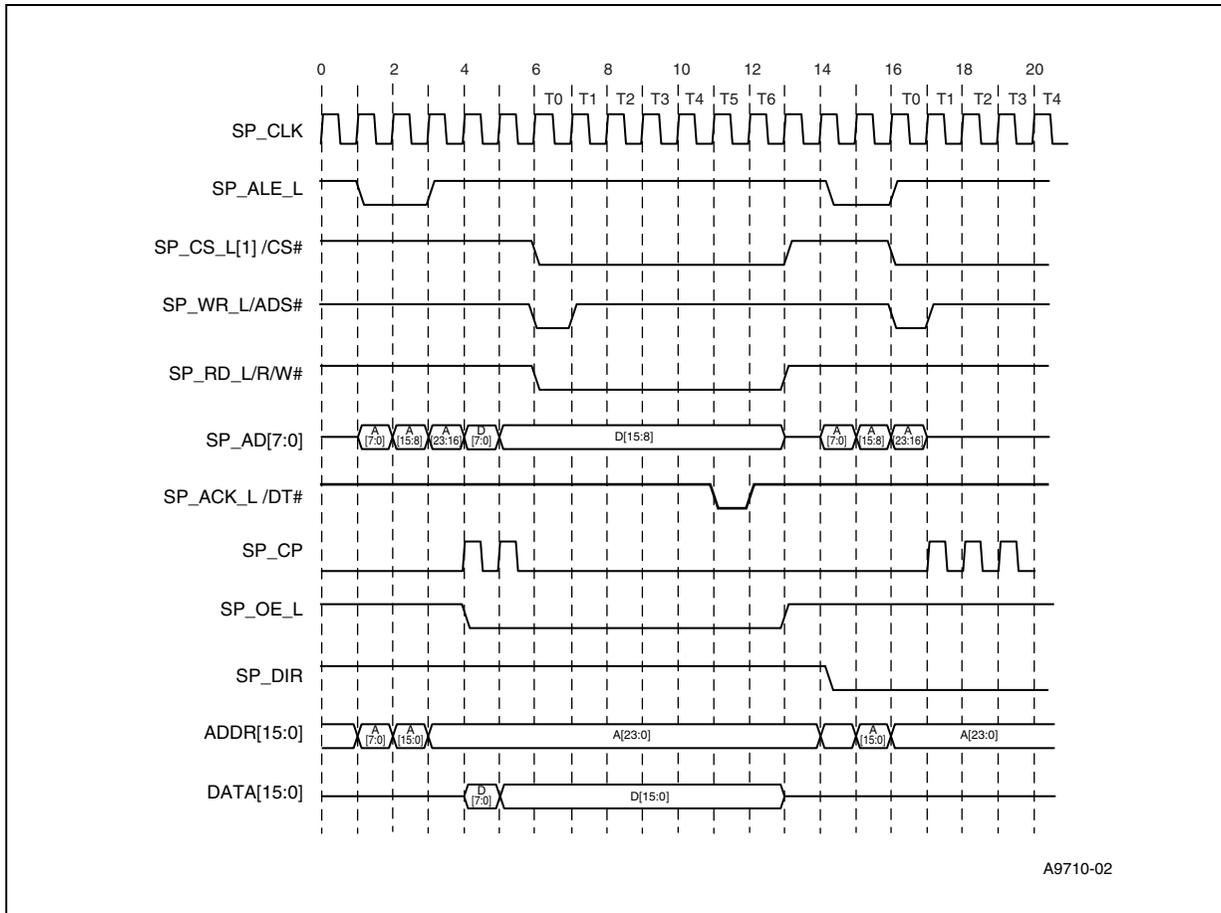
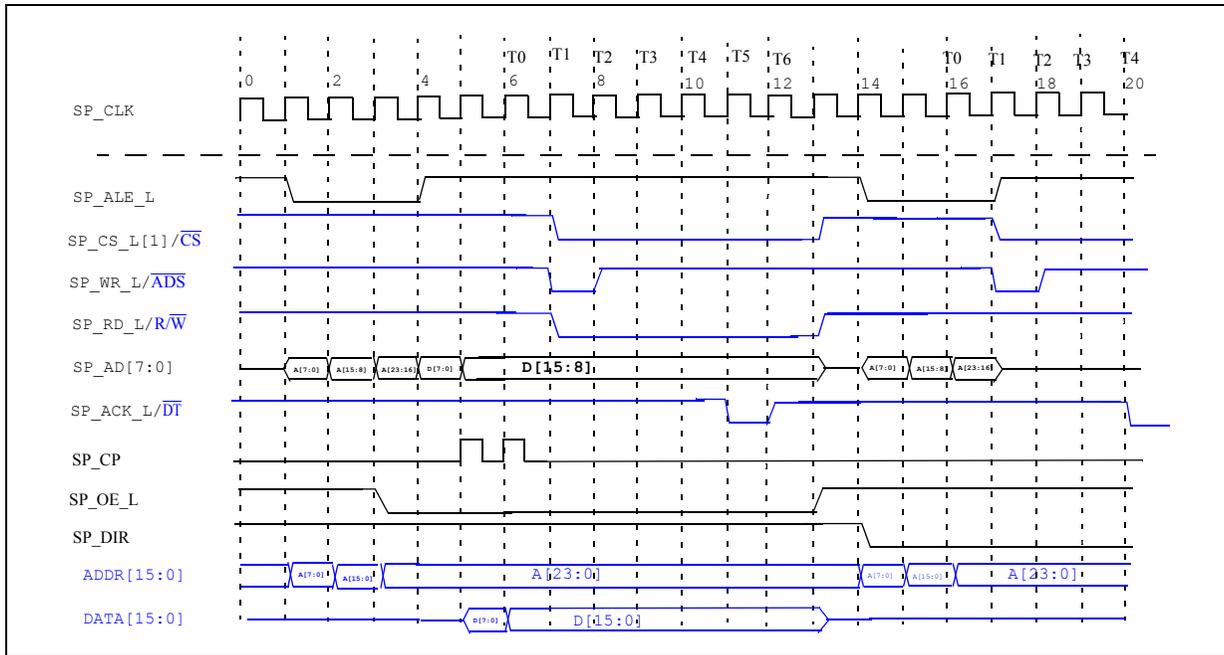


Figure 45. Mode 1 Single Write Transfer for Lucent TDAT042G5 Device (IXP2400 B0)



16-bit Microprocessor Read Interface Protocol

Figure 46, likewise depicts a single read transaction launched from the IXP2400 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=1.

In Figure 46, we can count twelve clock cycles used for the read transaction in total, (i.e., 240 ns) for a clock cycle of 20 ns, excluding a turnaround cycle after that.

**Figure 46. Mode 1 Single Read Transfer for Lucent TDAT042G5
 Device (IXP2400 A0/A1)**

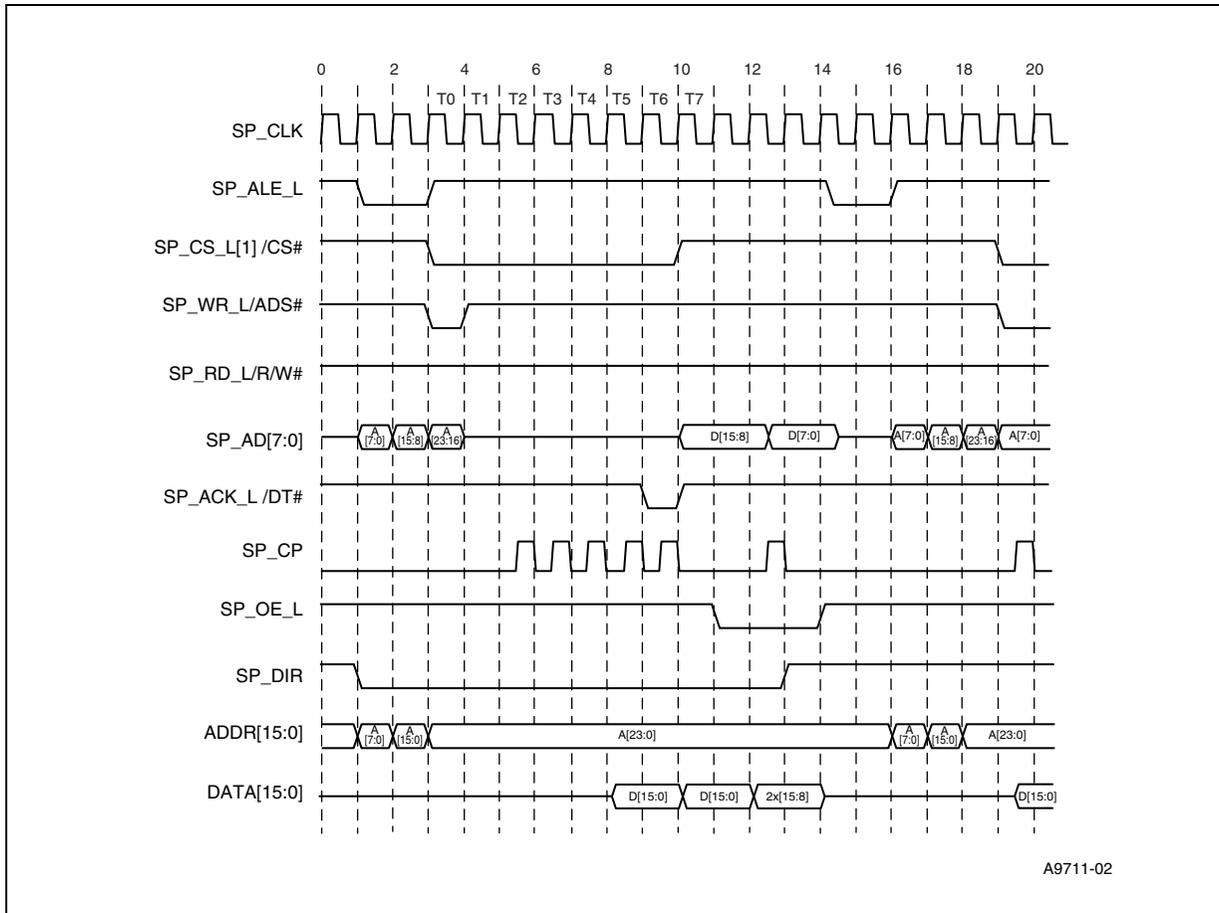
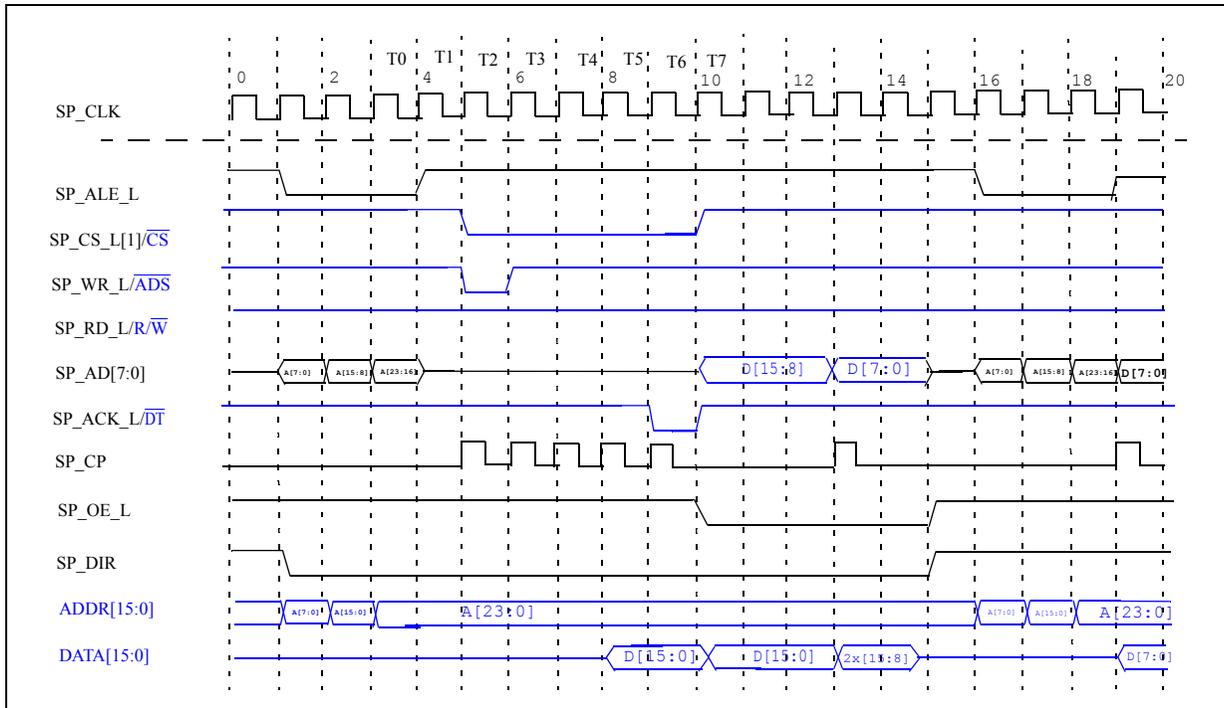


Figure 47. Mode 1 Single Read Transfer for Lucent TDAT042G5 Device (IXP2400 B0)



3.12.5.7.2 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 48 displays one of the topologies used to connect to the SlowPort with the PMC-Sierra PM5351 S/UNI-TETRA device.

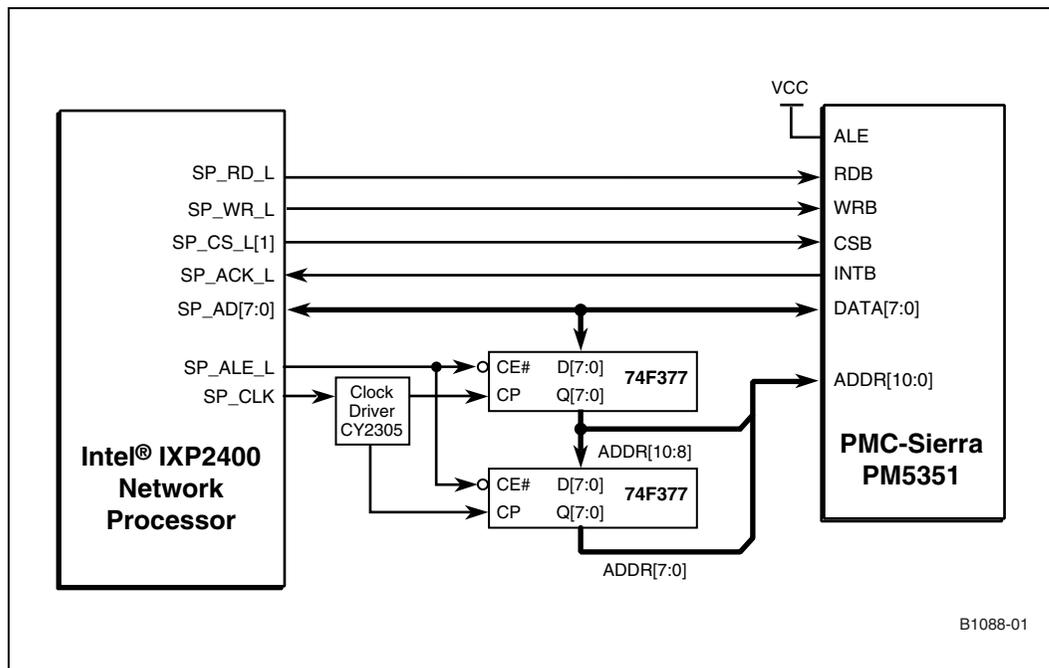
From Figure 48, because the protocols are very close to the generic SlowPort protocol, the pin counts and the functionality is quite compatible. We don't need to use any more pins in this case. The only difference is in the INTB signal, which will be connected to the SP_ACK_L. Therefore the SP_ACK_L 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 48. An Interface Topology with PMC-Sierra PM5351 S/UNI-TETRA



B1088-01

PMC-Sierra PM5351 S/UNI-TETRA Write Interface Protocol

Figure 49 depicts a single write transaction launched from the IXP2400 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 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 49. Mode 2 Single Write Transfer for PMC-Sierra PM5351 Device (IXP2400 A0/A1)

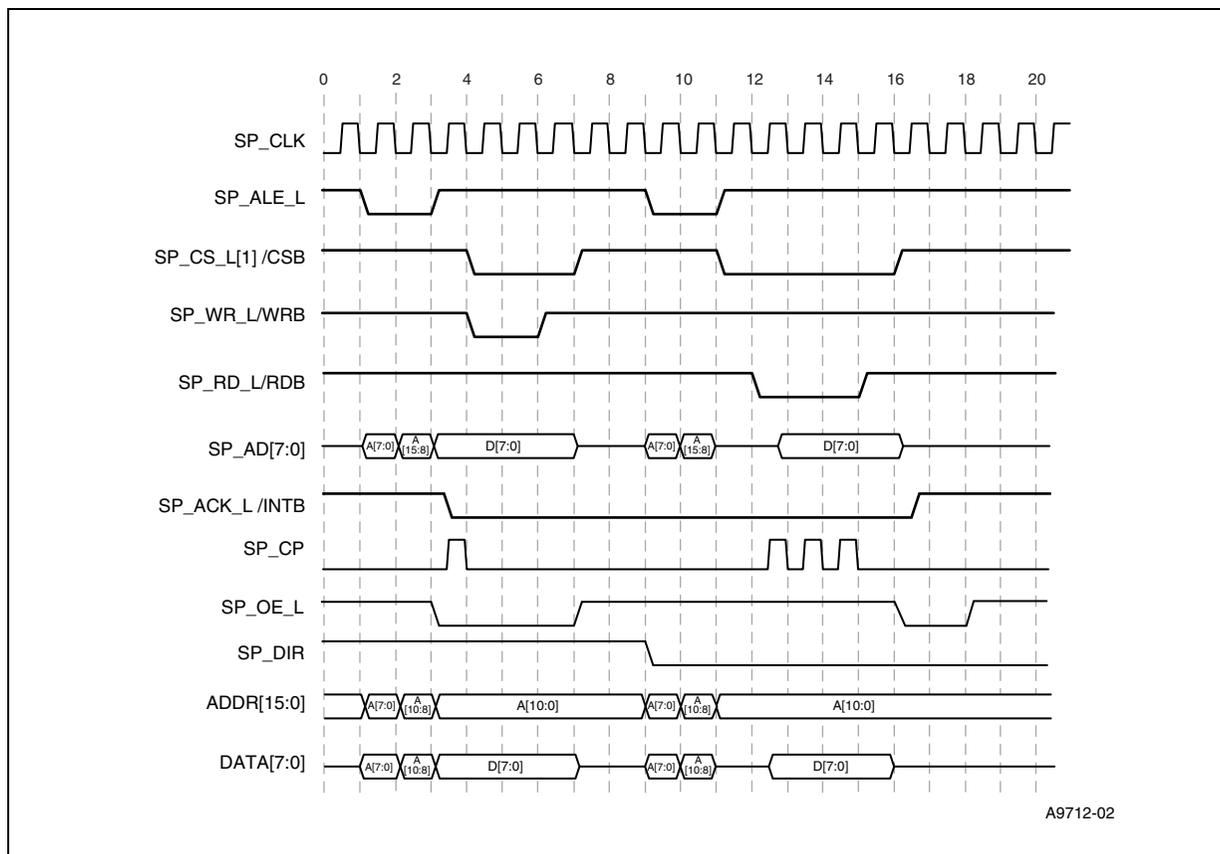
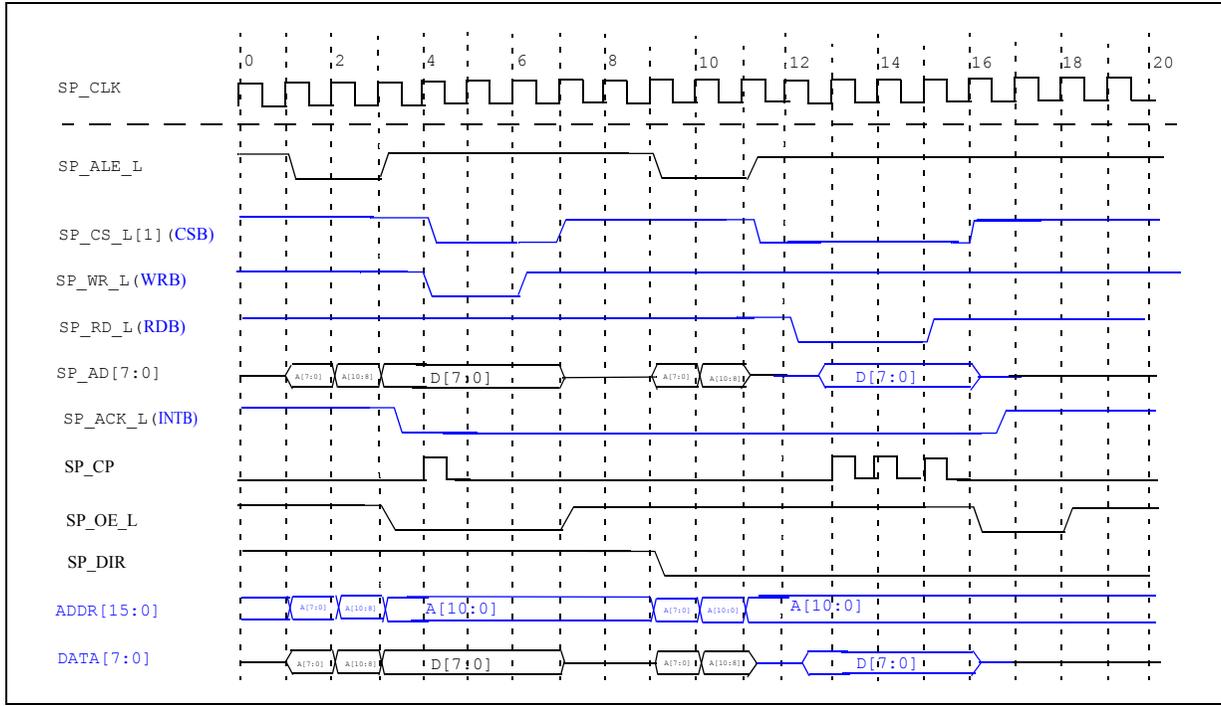


Figure 50. Mode 2 Single Write Transfer for PMC-Sierra PM5351 Device (IXP2400 B0)



PMC-Sierra PM5351 S/UNI-TETRA Read Interface Protocol

Figure 51, depicts a single read transaction launched from the IXP2400 Network Processor to the PMC-Sierra PM5351 device followed by a single write transaction.

In this case, there are eight clock cycles of access time, or 160 ns of a 50 MHz clock in total with a turnaround cycle attached at the back. The SP_PCR is programmed to mode two and the fields of SU, PW, and HD in the SP_RTC are programmed to one, four, one, which represent the SP_CS_L[1], SP_RD_L, and SP_CP pulse width respectively.

**Figure 51. Mode 2 Single Read Transfer for PMC-Sierra PM5351
Device (IXP2400 A0/A1)**

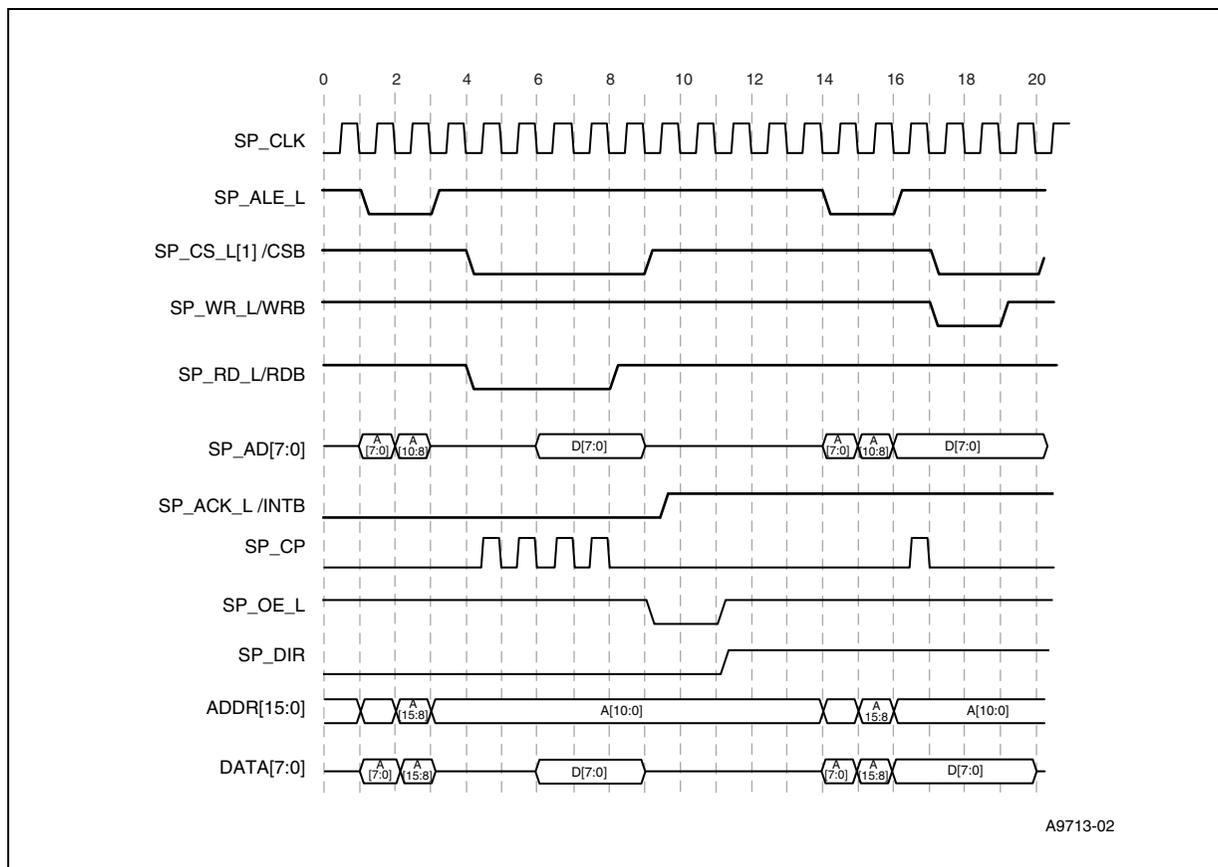
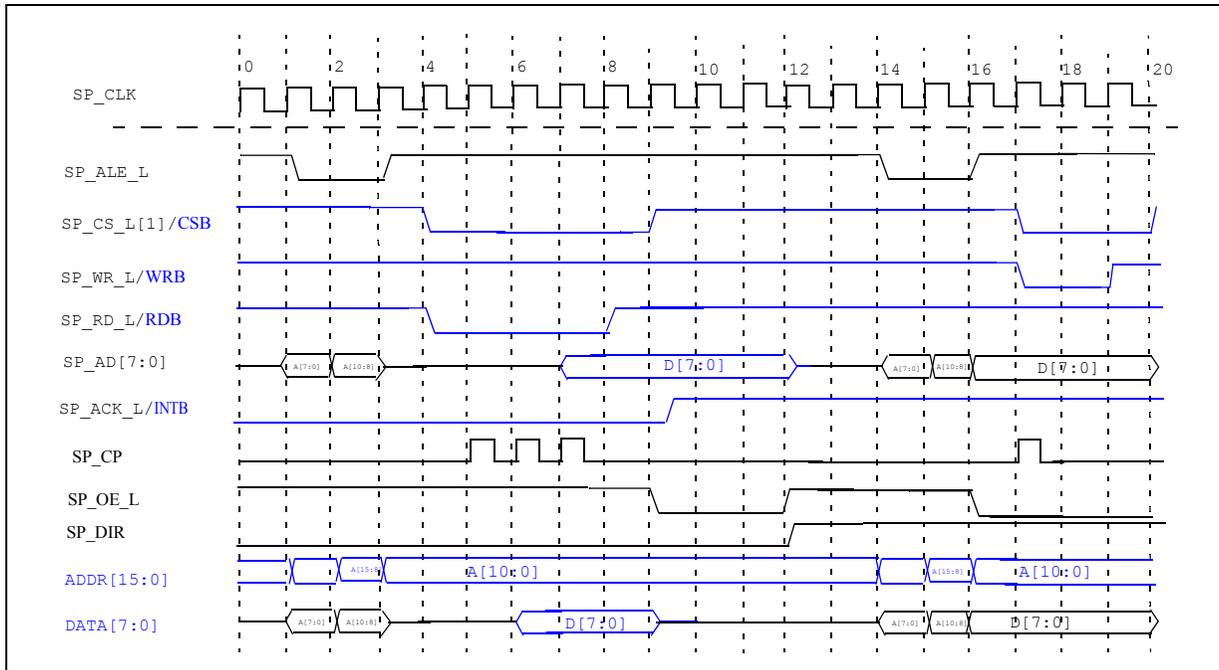


Figure 52. Mode 2 Single Read Transfer for PMC-Sierra PM5351 Device (IXP2400 B0)



3.12.5.7.3 Mode 3: Support the Intel and AMCC 2488 Mbps SONET/SDH Microprocessor Interface

The user can configure the address bus up to 24 bits.

Mode 3 Interfacing Topology

Figure 53 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 54. It employs the same method as Lucent's TDAT042G5's topology to pack and unpack the data between the IXP2400 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 53. An Interface Topology with Intel / AMCC SONET/SDH Device

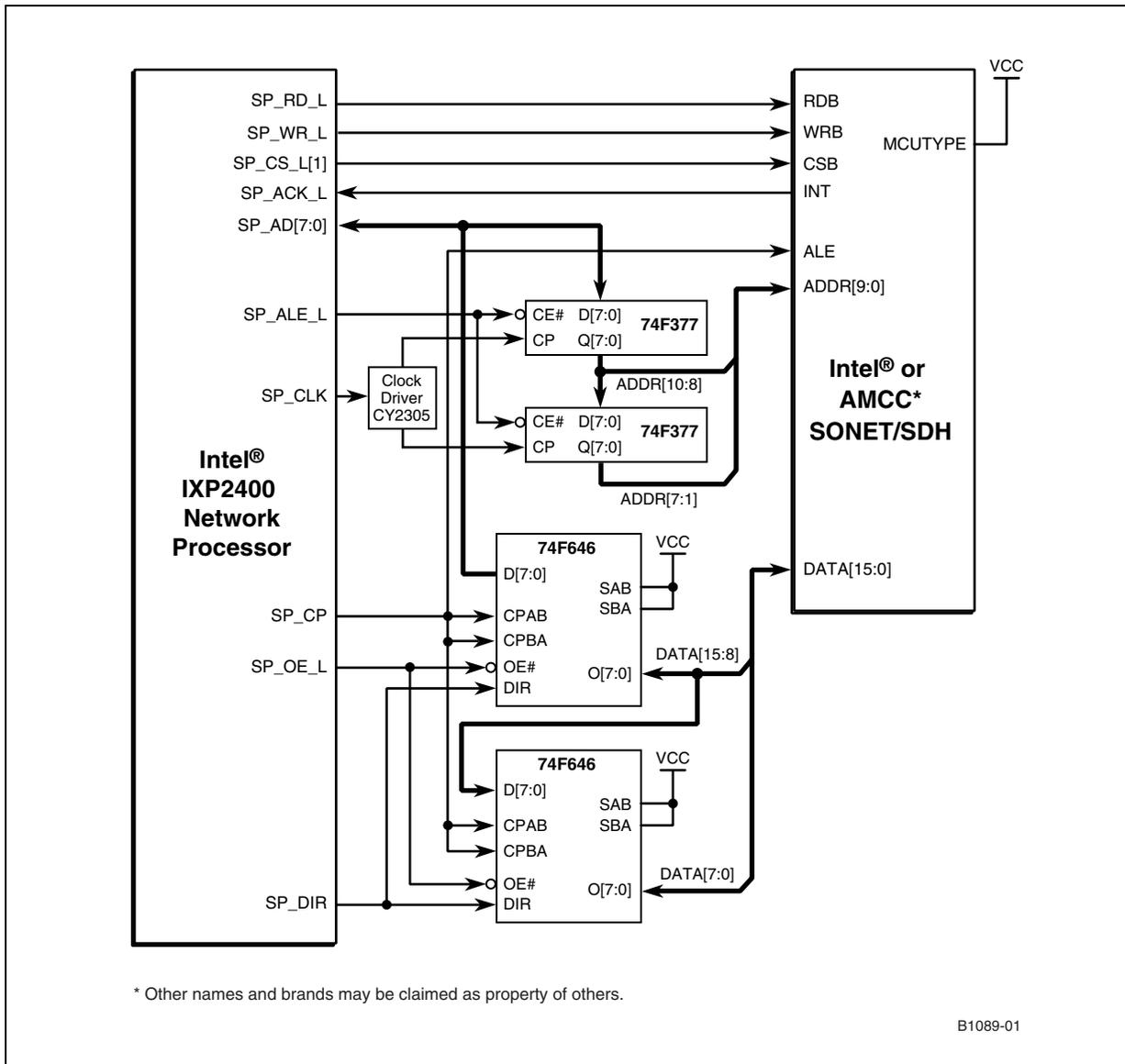
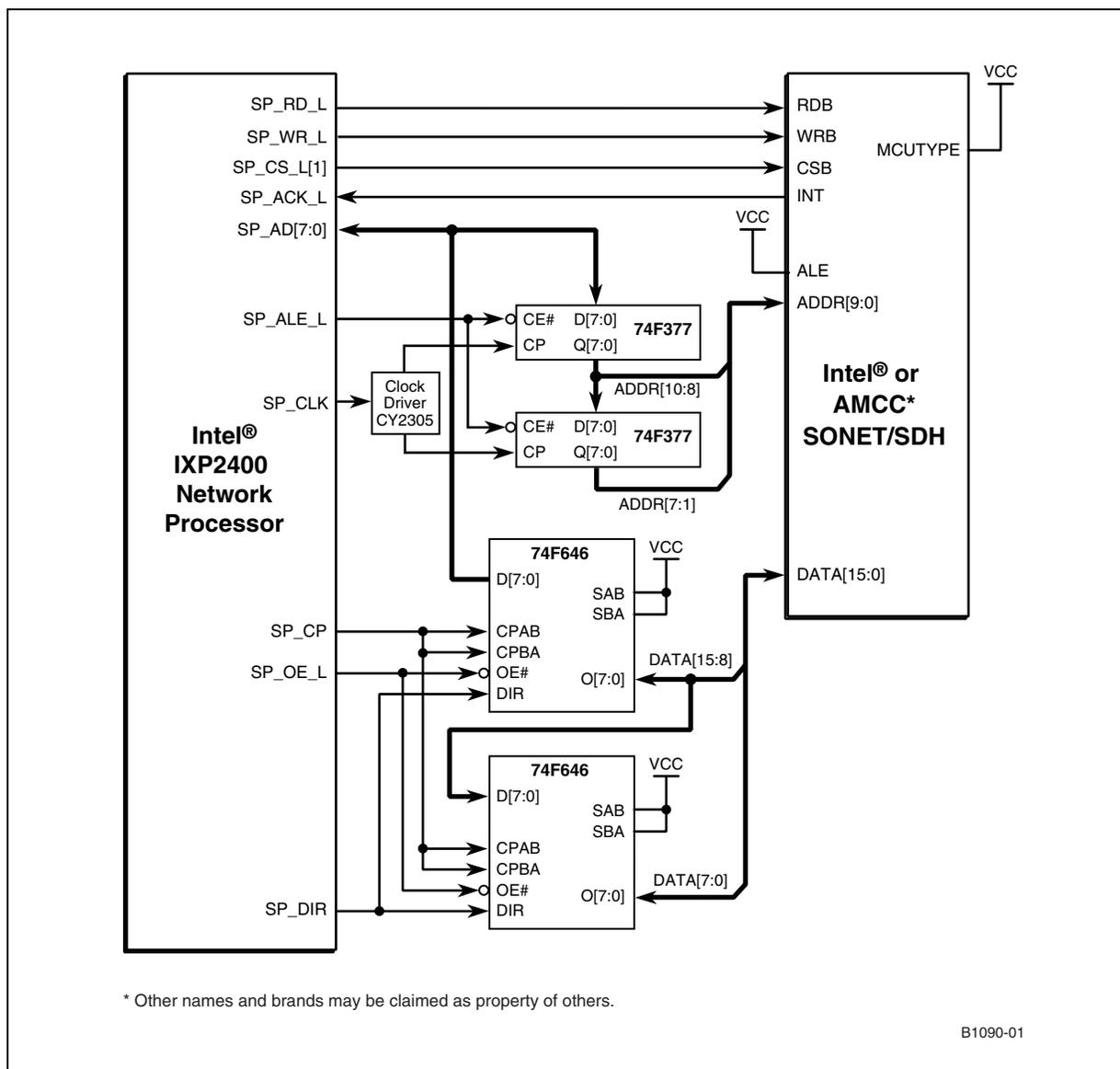


Figure 54. Mode 3 Second Interface Topology with Intel / AMCC SONET/SDH Device



Mode 3 Write Interface Protocol

Figure 55 depicts a single write transaction launched from the IXP2400 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 55. Mode 3 Single Write Transfer Followed by Read (IXP2400 A0/A1)

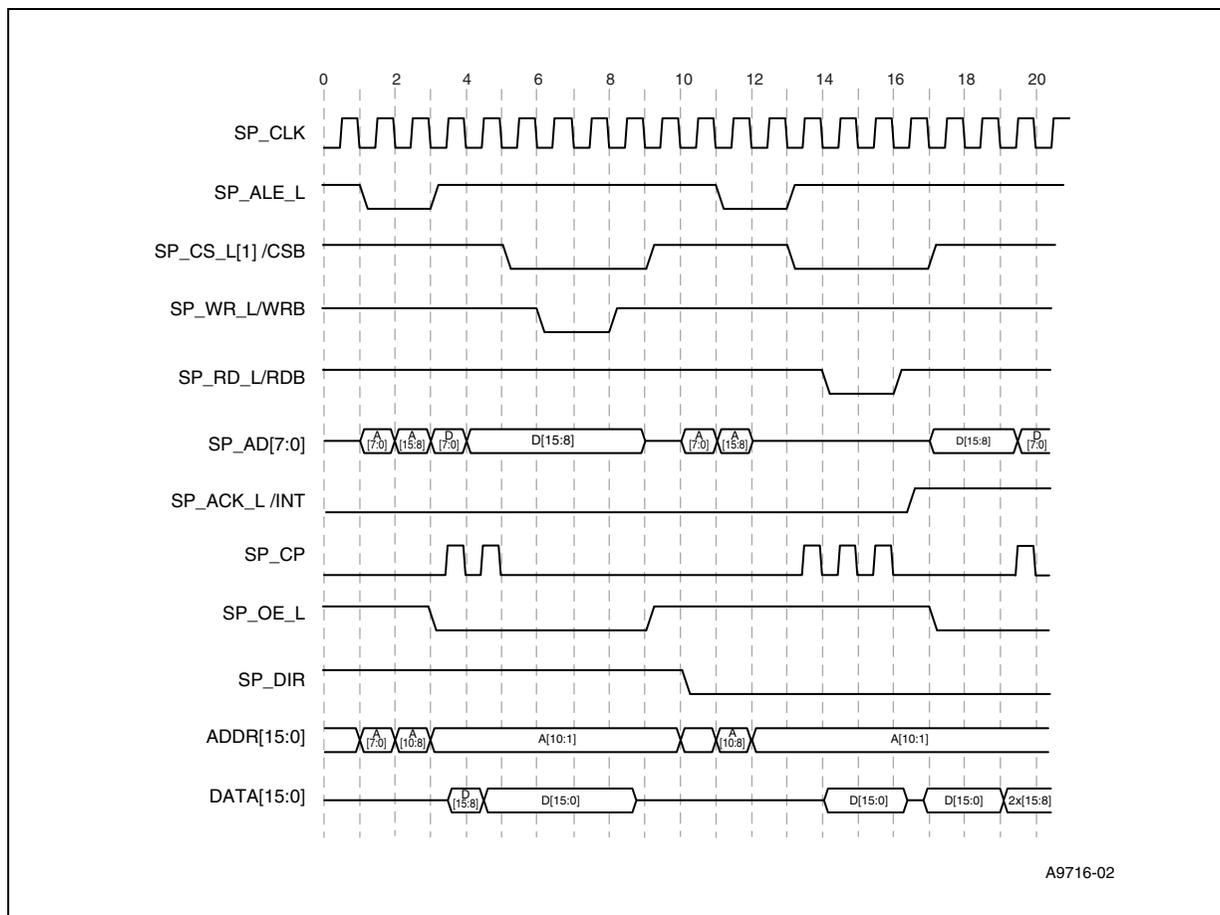
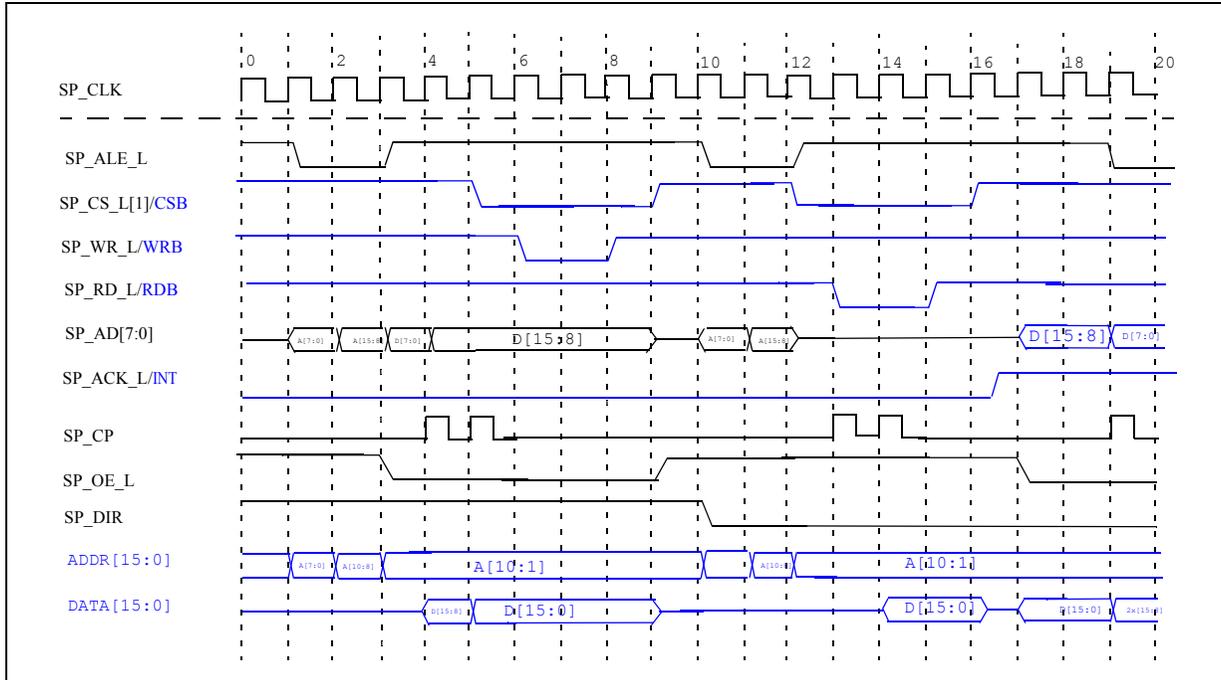


Figure 56. Mode 3 Single Write Transfer Followed by Read (IXP2400 B0)



Mode 3 Read Interface Protocol

Figure 57 depicts a single read transaction launched from the IXP2400 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.

Figure 57. Mode 3 Single Read Transfer Followed by Write (IXP2400 A0/A1)

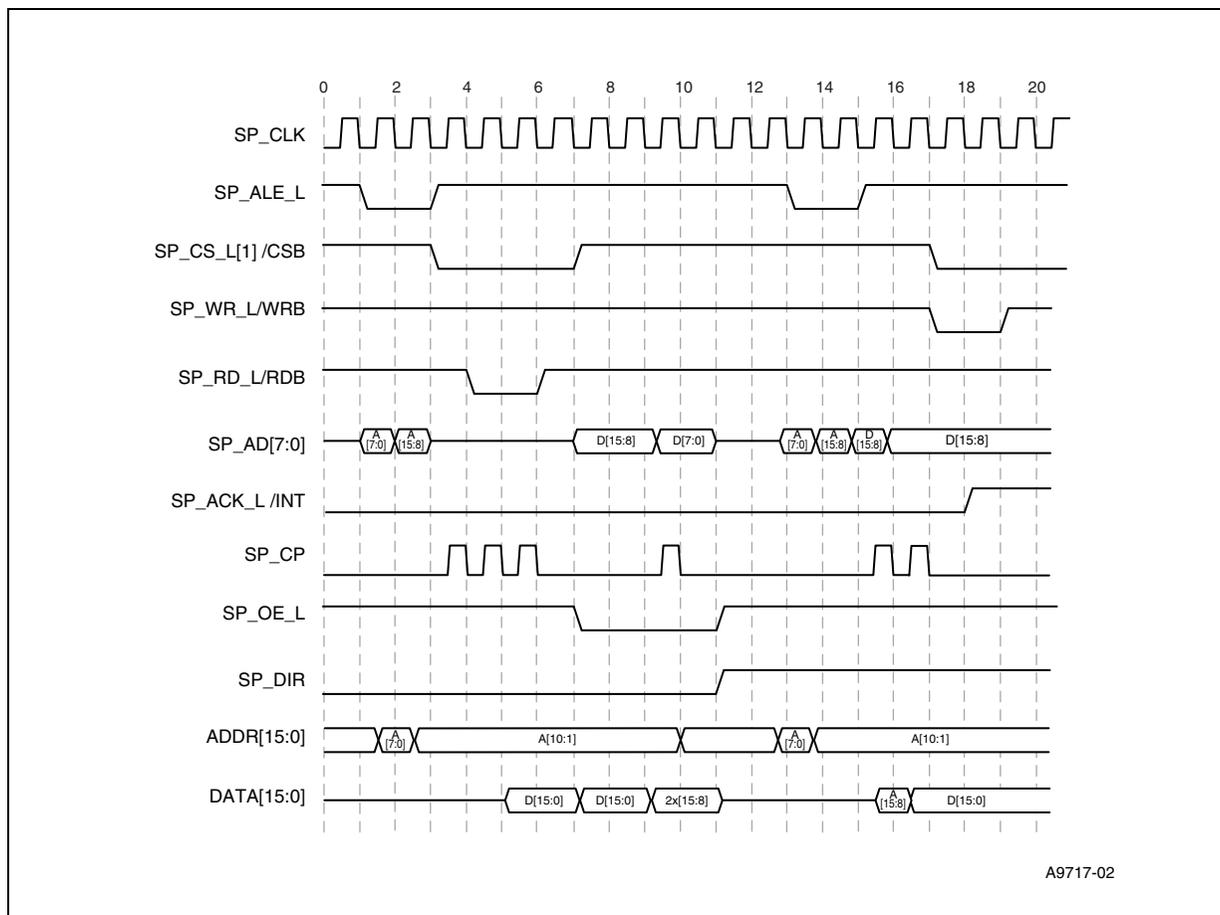
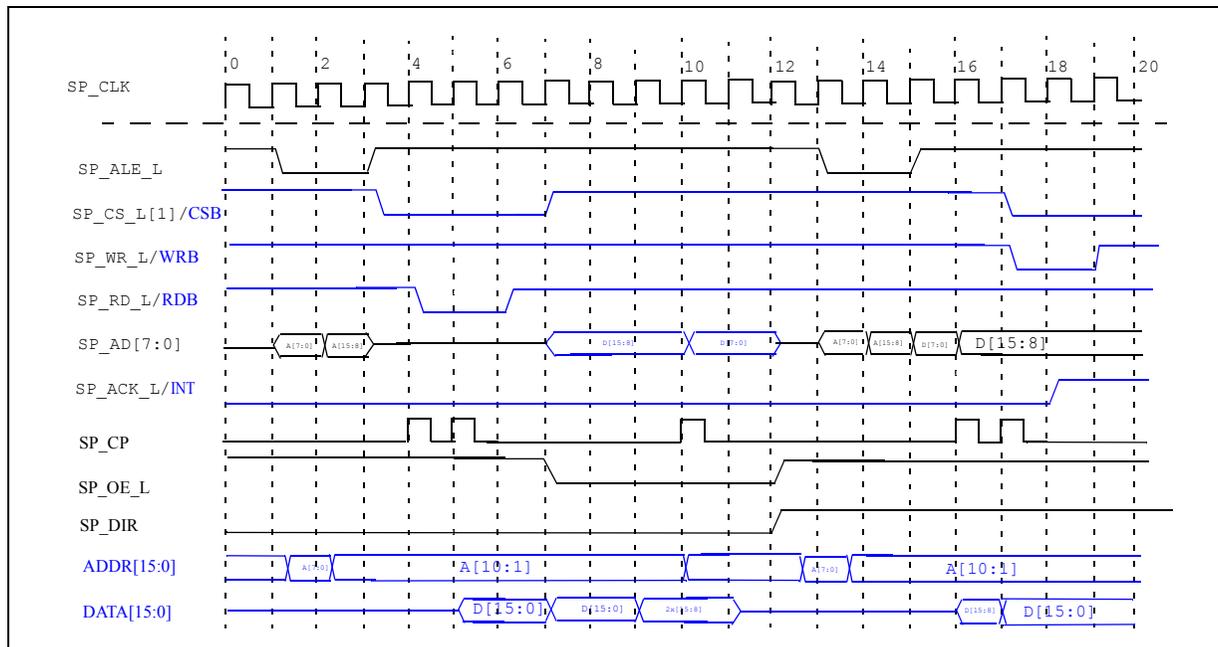


Figure 58. Mode 3 Single Read Transfer Followed by Write (IXP2400 B0)



Mode 4 Interfacing Topology

Figure 59 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 60.

It employs the same way to pack and unpack the data between the IXP2400 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 59. An Interface Topology with Intel / AMCC SONET/SDH Device in Motorola Mode

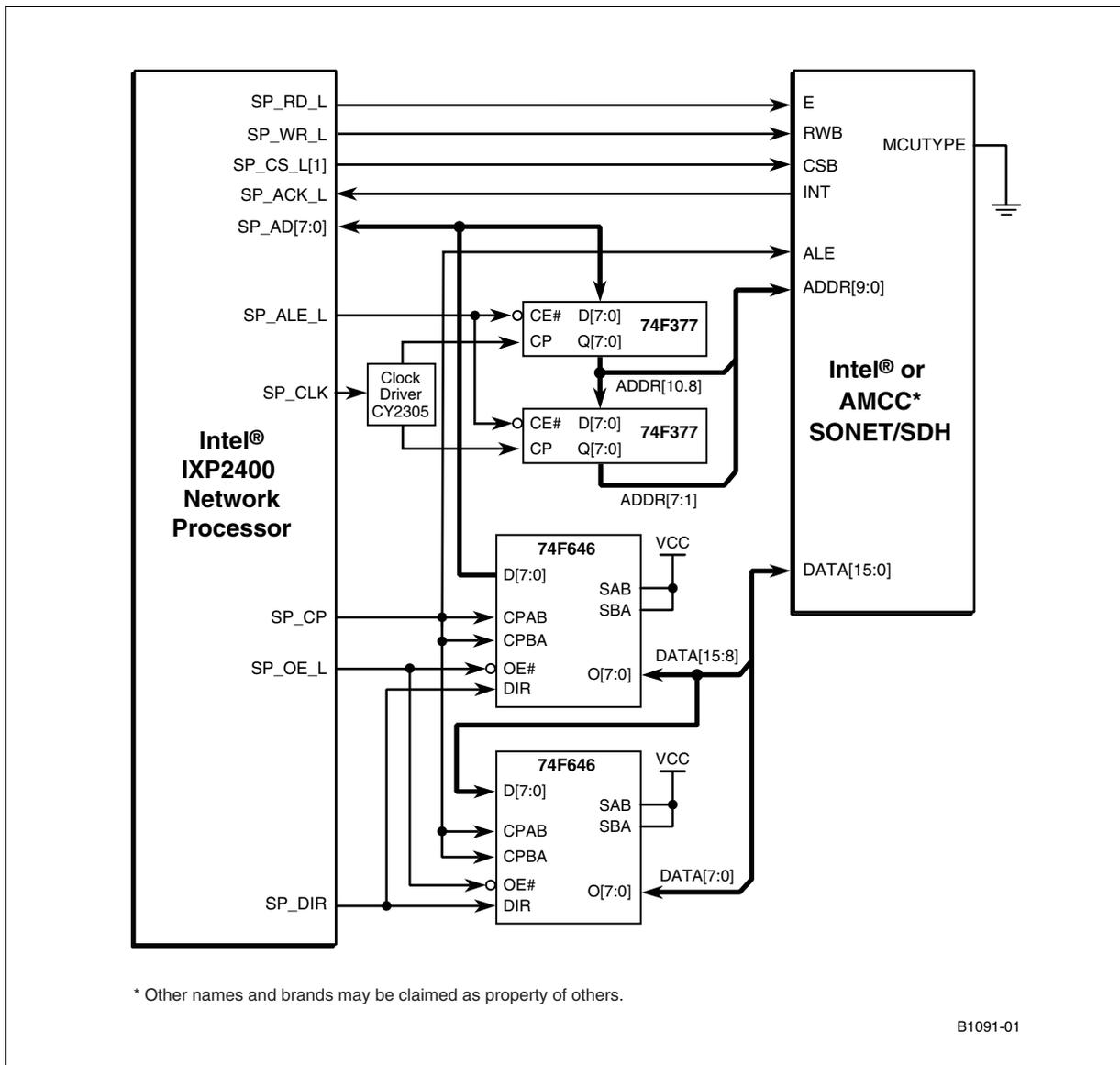
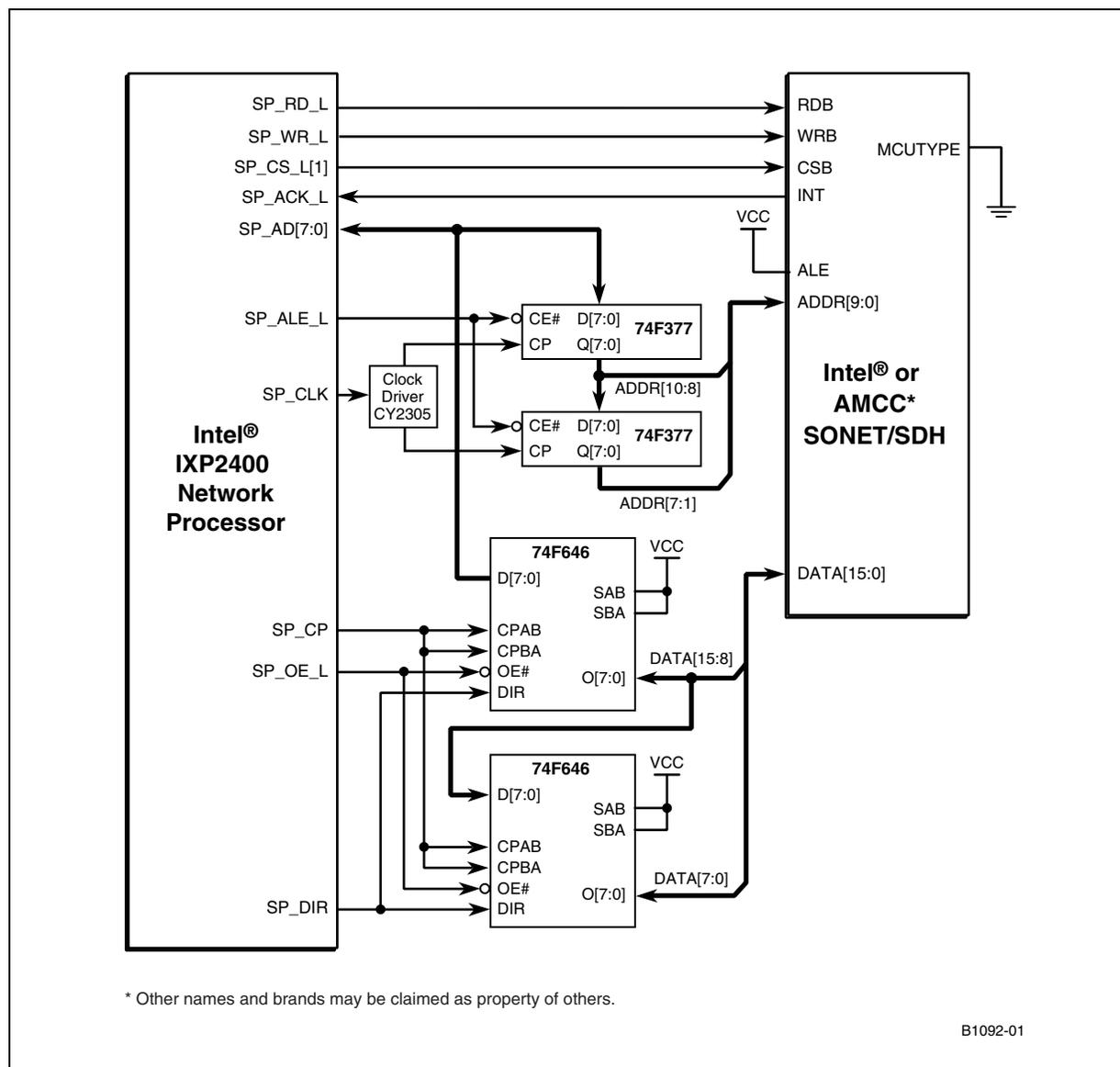


Figure 60. Second Interface Topology with Intel / AMCC SONET/SDH Device



Mode 4 Write Interface Protocol

Figure 61 depicts a single write transaction launched from the IXP2400 Network Processor to the Intel and AMCC SONET/SDH device, followed by two consecutive reads.

Comparing with the Lucent TDAT042G5 device, this device has a shorter access time, about 8 clock cycles, i.e., 160 ns. In this case, an intervened cycle may not be needed about the write transaction.

Figure 61. Mode 4 Single Write Transfer (IXP2400 A0/A1)

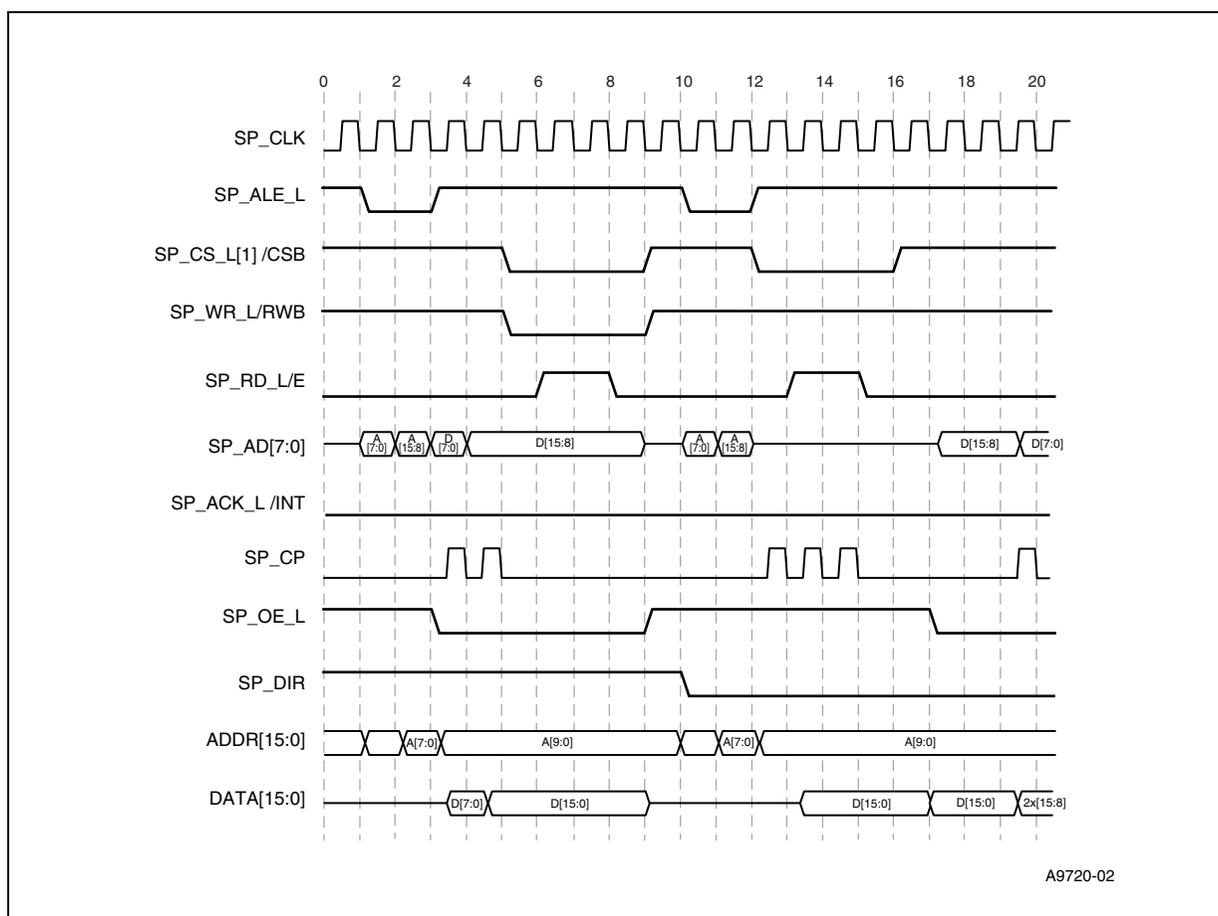
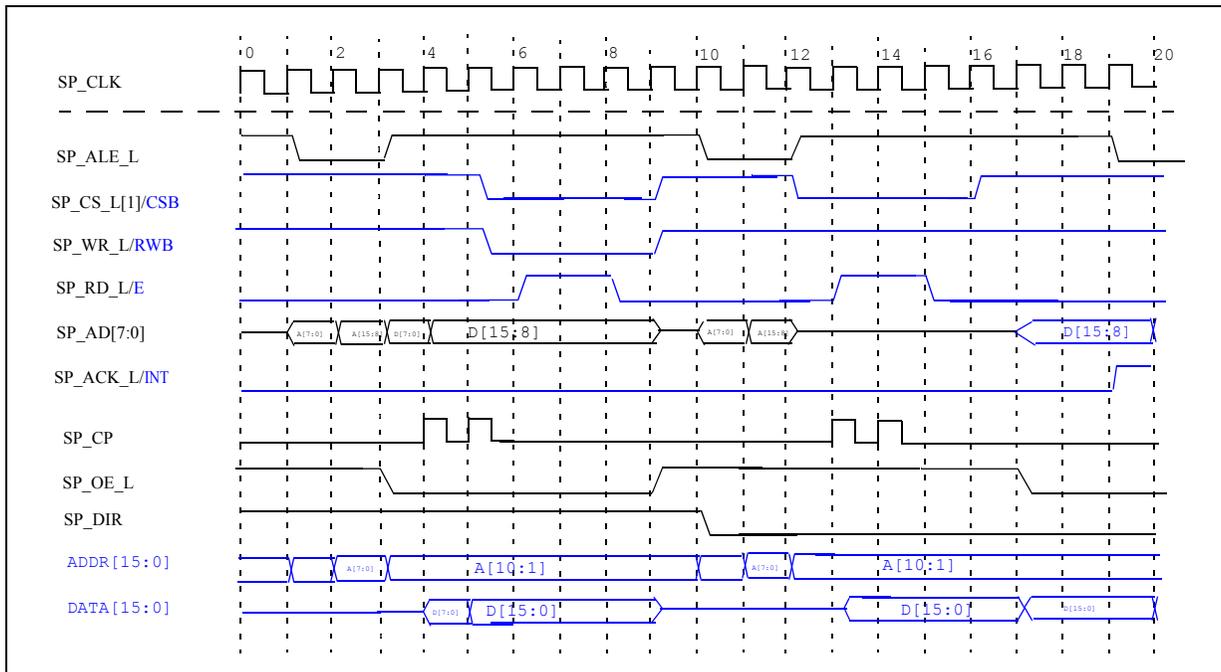


Figure 62. Mode 4 Single Write Transfer (IXP2400 B0)



Mode 4 Read Interface Protocol

Figure 63, depicts a single read transaction launched from the IXP2400 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.

Figure 63. Mode 4 Single Read Transfer (IXP2400 A0/A1)

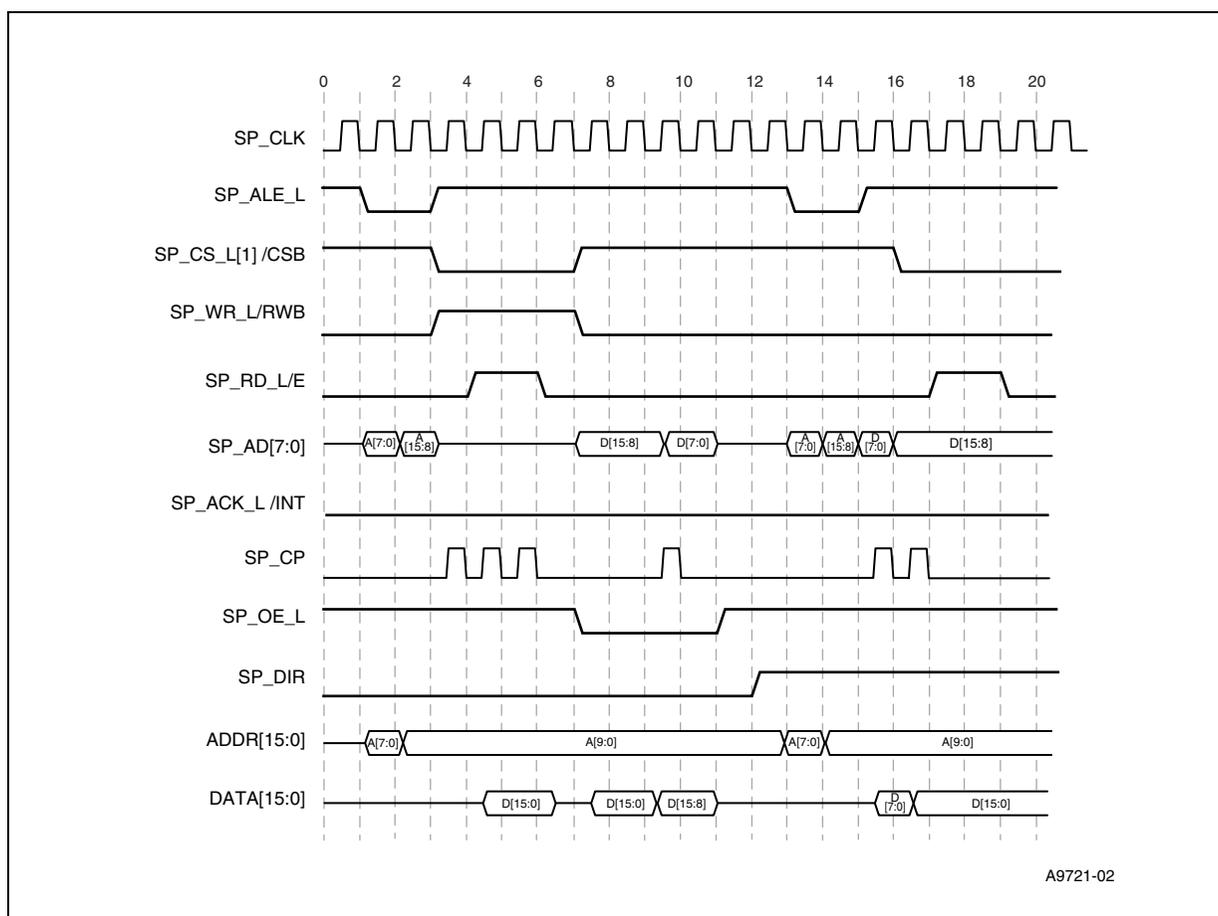
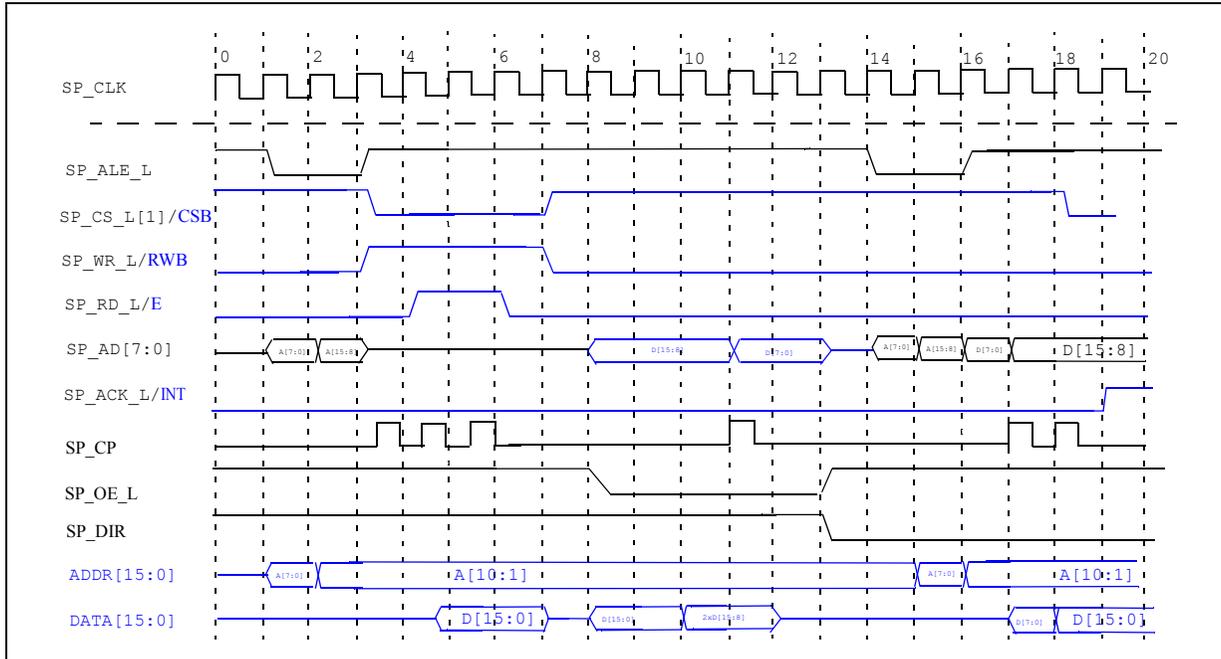


Figure 64. Mode 4 Single Read Transfer (IXP2400 B0)



3.12.6 PROM Device Timing Information for IXP2400 A0/A1

The following provides timing information of the SlowPort in mode 0.

Figure 65. Single Write Transfer for Fixed-Timed Device (IXP2400 A0/A1)

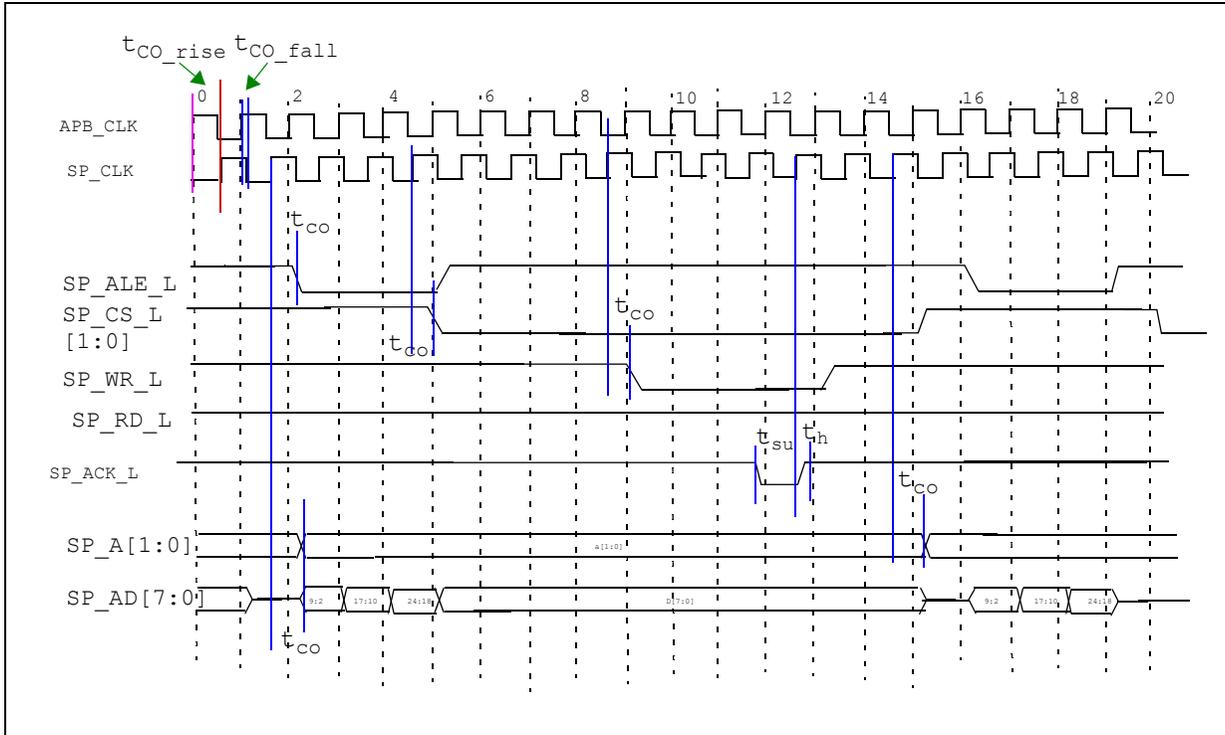


Table 44. Single Write Transfer for Fixed-Timed Device Timing Parameters (IXP2400 A0/A1)

| External Signals | tco fall max/min | tco rise max/min | th | tsu | tpw | unit |
|---|------------------|------------------|----|-------|-----|------|
| SP_CLK | 805/51 | 10125/10122 | | | | ps |
| SP_ALE | 10073/9966 | | | | | ps |
| SP_CS[0] | 10076/9967 | | | | | ps |
| SP_CS[1] | 10069/9964 | | | | | ps |
| SP_WR | 10073/9966 | 10034/9948 | | | a | ps |
| SP_RD | | | | | | ps |
| SP_ACK | | | 0 | 10216 | | ps |
| SP_AD[1:0] | 11493/9969 | 10093/9991 | | | | ps |
| SP_AD[7:0] output to external device | 10123/9930 | 10123/9930 | | | | ps |
| SP_AD[7:0] output to external device | 10069/9964 | 10030/9946 | | | | ps |

- a. The pulse width depends on the pulse-width parameter set in the SP_WTC1 and SP_WTC2 registers and the clock divisor as well. The minimum is 20 ns for one clock cycle at 50 MHz.

Figure 66. Framer Interrupt Enable Register Timing Diagram (IXP2400 A0/A1)

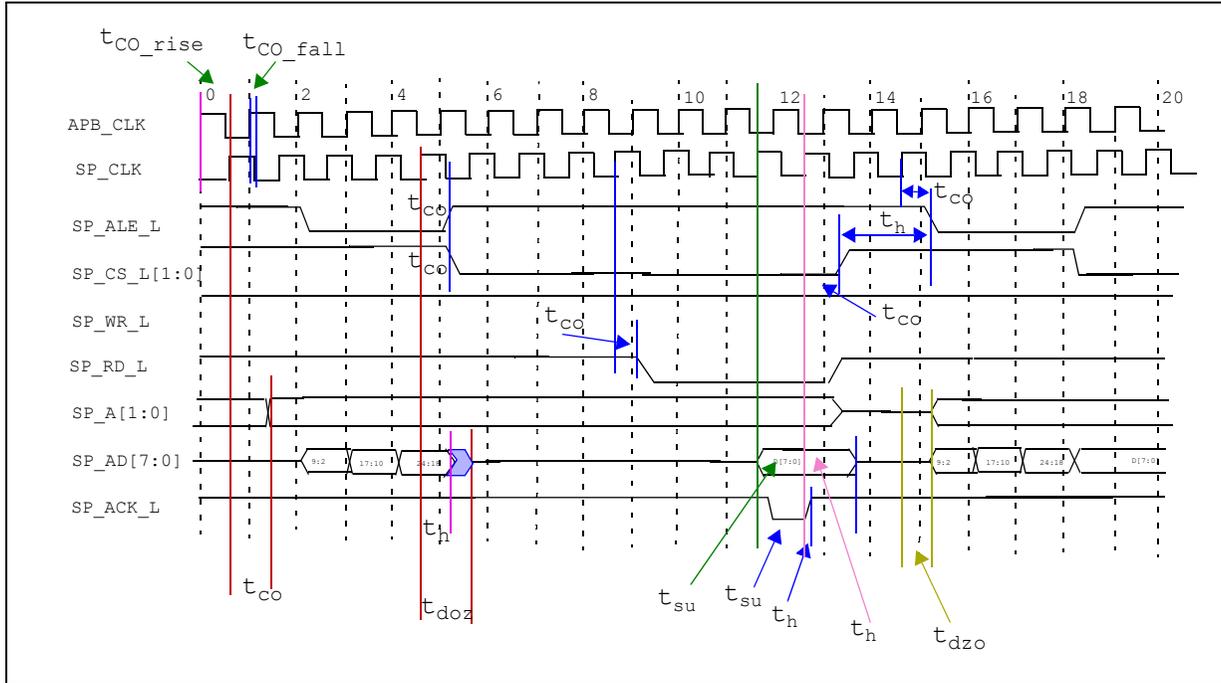


Table 45. Framer Interrupt Enable Register Timing Parameters (IXP2400 A0/A1)

| External Signals | tco fall max/min | tco rise max/min | th | tsu | tpw | toz/zo ^a max/min | unit |
|--|------------------|------------------|-------|-------|-----|-----------------------------|------|
| SP_ALE | 10073/9966 | 10034/9948 | | | | | ps |
| SP_CS[0] | 10076/9967 | | b | | | | ps |
| SP_CS[1] | 10069/9964 | | a | | | | ps |
| SP_WR | 10073/9966 | 10034/9948 | | | c | | ps |
| SP_RD | 10073/9966 | | | | | | ps |
| SP_ACK | | | 0 | 10216 | | | ps |
| SP_AD[1:0] | 11493/9969 | 10093/9991 | | | | | ps |
| SP_AD[7:0] output to external device | 11493/9969 | 10123/9930 | 9946 | | | 13113/11179 13717/11365 | ps |
| SP_AD[7:0] input from external device | | | 30000 | 10233 | | | ps |

- a. The values are only obtained without loading. We need to add the delay caused by loading according to the typical PN and slew bit table below.
- b. At least two clock cycles.
- c. The pulse width depends on the pulse-width parameter set in the SP_WTC1 and SP_WTC2 registers and the clock divisor as well. The minimum is 20 ns for one clock cycle at 50 MHz.

3.12.7 PROM Device Timing Information (IXP2400 B0)

The following provides timing information for the SlowPort in mode 0.

Figure 67. Single Write Transfer for Fixed-Timed Device (IXP2400 B0)

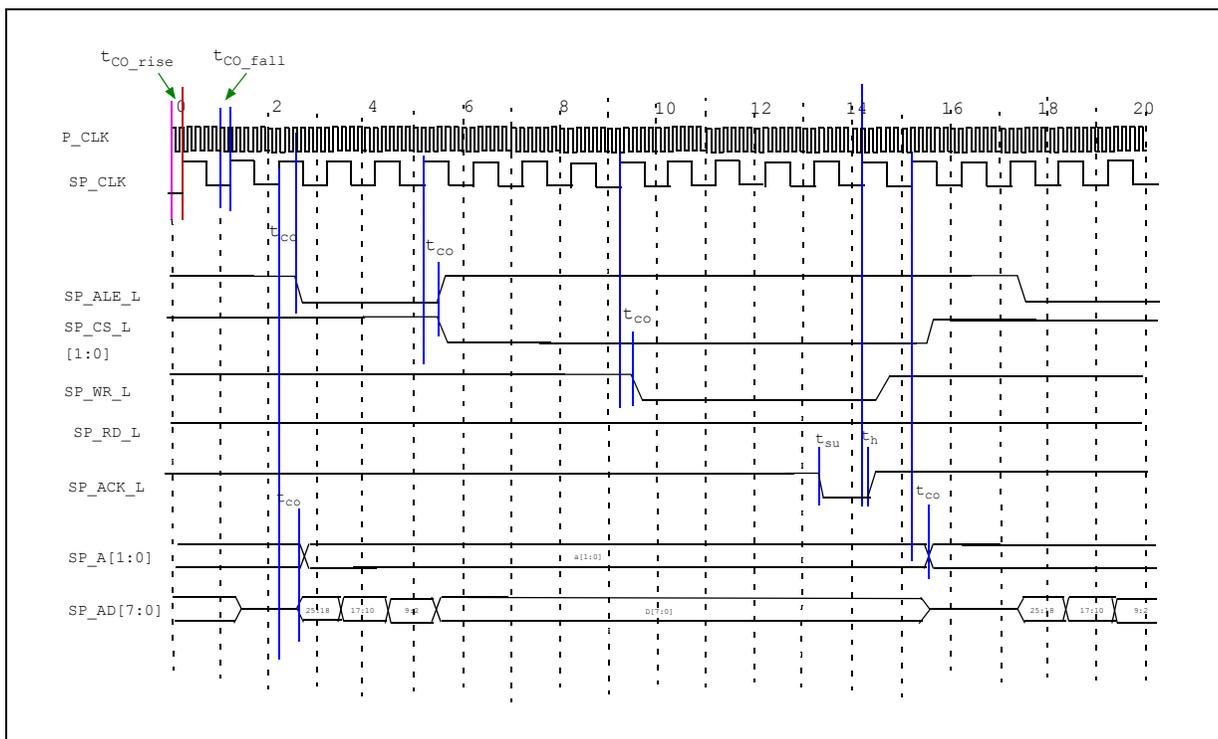


Table 46. Single Write Transfer for Fixed-Timed Device Timing Parameters (IXP2400 B0)

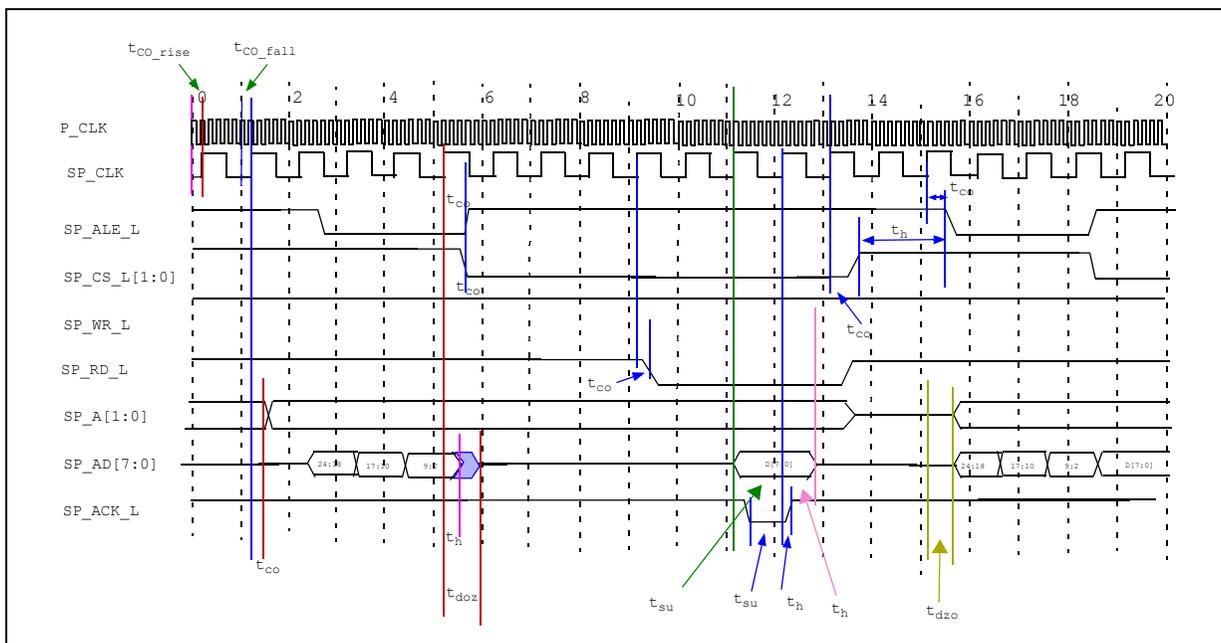
| External Signals | tco rise (default ^a) (ns) | | tco fall (default ^b) (ns) | | th (ns) | | tsu (ns) | | tpw (ns) | |
|------------------|---------------------------------------|-----|---------------------------------------|-----|---------|-----|----------|-----|----------|-----|
| | Max | Min | Max | Min | Max | Min | Max | Min | Max | Min |
| SP_CLK | 3.0 | 1.4 | 3.7 | 3.3 | | | | | | |
| SP_ALE | 8.5 | 5.3 | 9.0 | 5.4 | | | | | | |
| SP_CS[0] | 8.4 | 5.3 | 9.0 | 5.4 | | | | | | |
| SP_CS[1] | 8.4 | 5.3 | 9.0 | 5.4 | | | | | | |
| SP_WR | 9.1 | 5.5 | 9.2 | 5.6 | | | | | c | |

Table 46. Single Write Transfer for Fixed-Timed Device Timing Parameters (IXP2400 B0)

| External Signals | tco rise (default ^a) (ns) | | tco fall (default ^b) (ns) | | th (ns) | | tsu (ns) | | tpw (ns) | |
|---|---------------------------------------|-----|---------------------------------------|-----|---------|-----|----------|-----|----------|-----|
| | Max | Min | Max | Min | Max | Min | Max | Min | Max | Min |
| SP_RD | | | | | | | | | | |
| SP_ACK | | | | | 0 | 0 | 6.8 | 4.5 | | |
| SP_A[1:0] | 8.4 | 5.3 | 9.0 | 5.4 | | | | | | |
| SP_AD[7:0] output to external device | 9.0 | 5.5 | 9.2 | 5.6 | 9.2 | 5.5 | | | | |

- Default out timing delay is controlled by the TXE register. By default this register is set to 1, i.e. two P clock cycles delay or 6666.66 ps. minimum delay can be set to 0.
- Default out timing delay is controlled by the TXE register. By default, this register is set to 1, i.e. two P clock cycles delay or 6666.66 ps. Minimum delay can be set to 0.
- The pulse width depends on the pulse-width parameter set in the SP_WTC1 and SP_WTC2 registers and the clock divisor as well. The minimum is 20 ns for one clock cycle at 50 MHz.

Figure 68. Framer Interrupt Enable Register Timing (IXP2400 B0)



Microengines

4

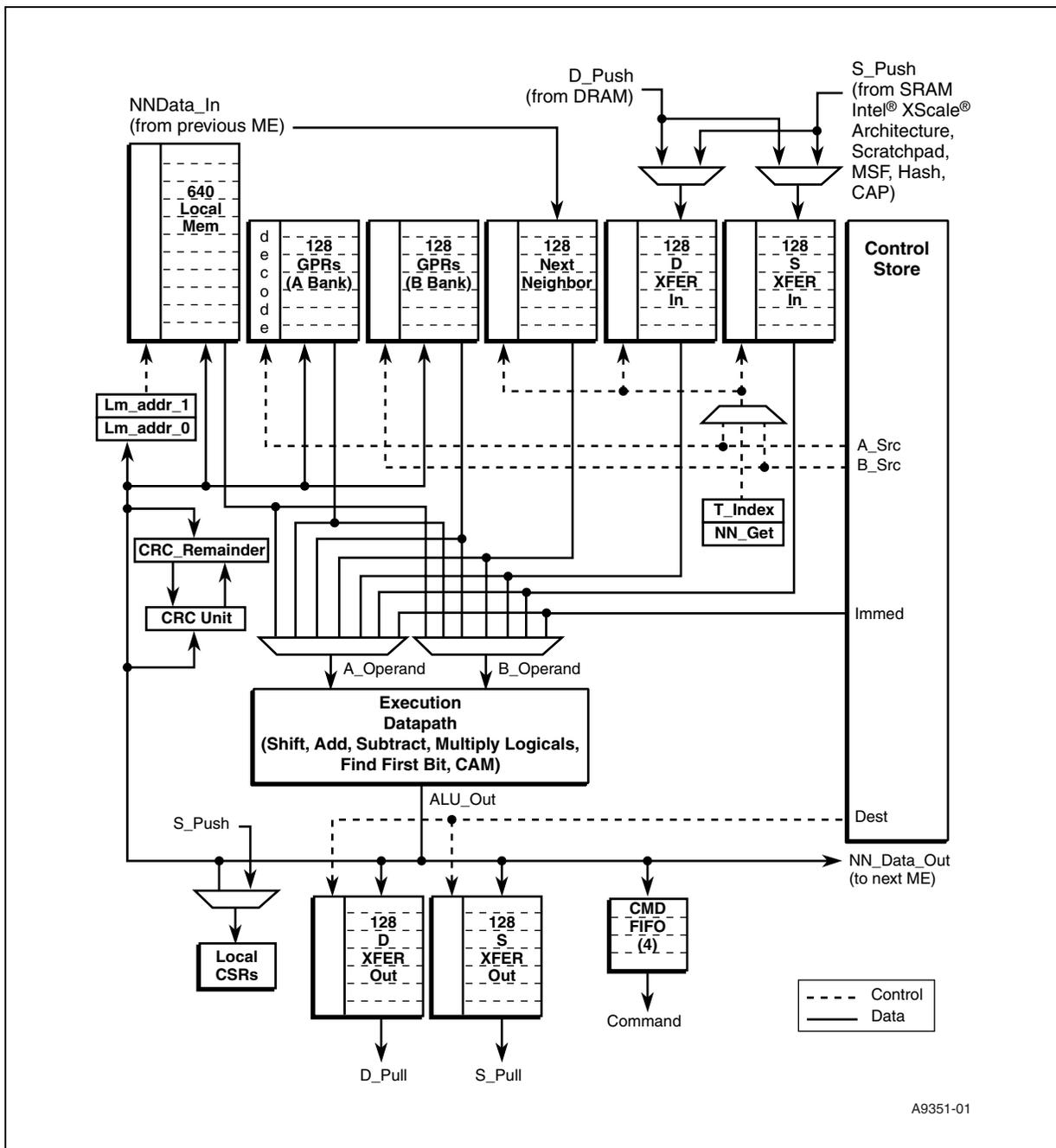
This section defines the Network Processor Microengine (ME). This is the second version of the Microengine, and is often referred to as the MEv2 (Microengine Version 2).

4.1 Overview

The following sections describe the programmer's view of the Microengine. The block diagram in [Figure 69](#) 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 69. Microengine Block Diagram



A9351-01

4.1.1 Control Store

The Control Store is a static RAM, which holds the program that the Microengine executes. It holds 4096 instructions, each of which is 40-bits wide. It is initialized by an external device (XScale), 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 (e.g., to XScale).

4.1.2 Contexts

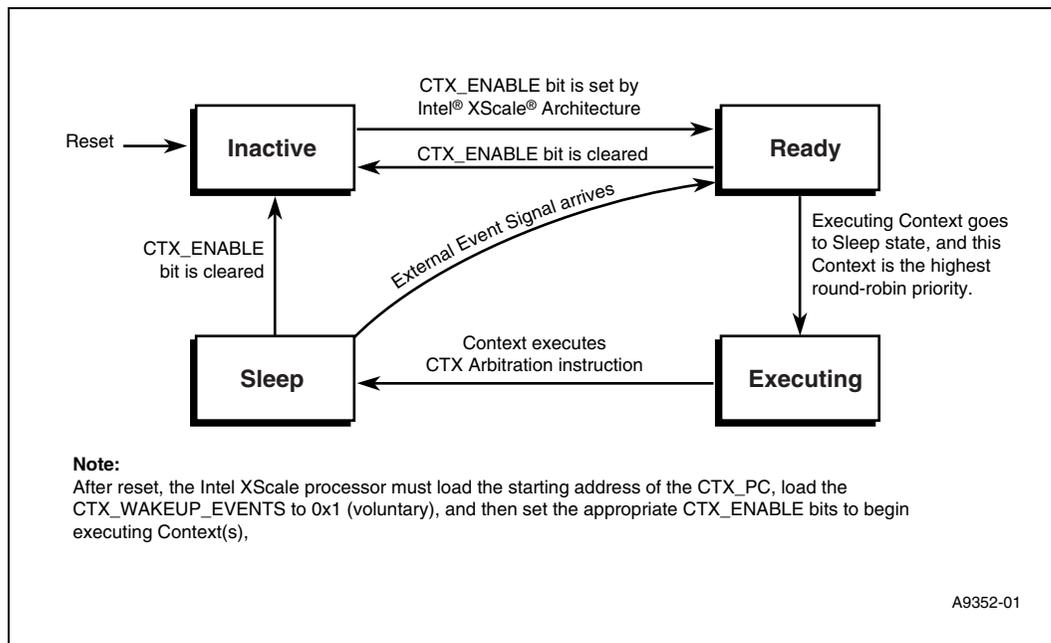
There are eight hardware Contexts available in the Microengine. To allow for efficient context swapping, each Context has its own register set, Program Counter, and Context specific Local Registers. Having a separate copy per Context eliminates the need to move Context specific information to/from shared 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 that 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 70](#) 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 70. 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 69](#), 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 Memory¹

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 48](#).

Note: The Microengine registers are defined in the *IXP2400 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.

1. S_Transfer_In
2. S_Transfer_Out
3. D_Transfer_In
4. 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 based on the Push_ID input to the Microengine.

As shown in [Figure 69](#), the mux between the S and D push buses allow data arriving on the buses to be written to either the S or D transfer registers. No mux exists for the pull buses so it is not possible to write data from the S transfer registers onto the D push bus or to write data from the D Transfer registers onto the S push bus.

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.

1. Some implementations may choose to include a different amount of Local Memory. The minimum allowed is 512 32-bit words. The maximum is 1024 32-bit words (limited by the size of LM_Addr).

The S_Transfer_In and S_Transfer_Out Registers connect to the S_Push and S_Pull buses, respectively.

The D_Transfer_In and D_Transfer_Out Transfer Registers connect to the D_Push and D_Pull buses, respectively.

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 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.

The number and types of external units connected to the Push and Pull buses is chip implementation specific.

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 6-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'.

Note: In this mode the datapath bypass is not used and the code must ensure it uses the newly written data correctly. The following example shows the earliest use of the new data. Any earlier use of the destination register would get the old contents of the register.

```
alu [n$_reg, ...]
inst_2
inst_3
inst_4
inst_5
inst_6
alu [..., n$_reg]
```

Table 47. 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 (LM)

Local Memory is addressable storage located in the Microengine. LM is read and written exclusively under program control. LM supplies operands to the execution datapath as a source, and receives results as a destination. The specific LM 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 11](#).

Example 11. 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, *1$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 `*1$index#++` and `*1$index#--`), as shown in [Example 12](#), where three consecutive LM locations are used in three consecutive instructions.

Example 12. Using LM_ADDR in Consecutive Instructions

```

alu[dest_reg1, src_reg1, op, *1$index0++]
alu[dest_reg2, src_reg2, op, *1$index0++]
alu[dest_reg3, src_reg3, op, *1$index0++]

```

LM is written by selecting it as a destination. [Example 13](#) shows copying a section of LM to another section. Each instruction accesses the next sequential LM location from the previous instruction.

Example 13. Copying One Section of LM to Another Section

```
alu[*1$index1++, --, B, *1$index0++]
alu[*1$index1++, --, B, *1$index0++]
alu[*1$index1++, --, B, *1$index0++]
```

Example 14 shows loading and using both LM addresses.

Example 14. Loading and Using both LM 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, *1$index0, op, src_reg1]
alu[dest_reg2, *1$index1, op, src_reg2]
```

As shown in Example 11 there is a latency in loading LM_ADDR. Until the new value is loaded the old value is still usable. Example 15 shows the maximum pipelined usage of LM_ADDR.

Example 15. 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, *1$index0, op, src_reg1] ; uses address from gpr_m
alu[dest_reg2, *1$index0, op, src_reg2] ; uses address from gpr_n
alu[dest_reg3, *1$index0, op, src_reg3] ; uses address from gpr_o
alu[dest_reg4, *1$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 16. The source and destination can use different offsets.

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

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

Note: LM has 640 32-bit words. LM_ADDR can hold values from 0 to 1023. Using LM_ADDR to address LM when its value is not in the range of 0..639 will cause unpredictable results.

To the programmer, all instructions using LM 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 17 shows using post-modify when the same LM_ADDR is both source and destination.

Example 17. LM_ADDR Use as Source and Destination

```
alu[*1$index0++, --, ~B, *1$index0]
```

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

1. Read `l_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

The rule that the assembler uses is that when the same index register is used as both a source and destination, any increment/decrement operator must be applied to the destination usage.

[Example 19](#) is not legal; LM offset cannot be used at the same time as post-modify.

Example 18. LM_ADDR Use as Source and Destination

```
alu[*l$index0++, --, ~B, *$index0] ; ***Illegal
```

In [Example 19](#), the second instruction will access the LM location, one past the source/destination of the first.

Example 19. LM_ADDR Post-increment

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

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 48](#)). Context-Relative addressing is a powerful feature that enables eight different contexts to share the same microcode, yet maintain separate data.

[Table 48](#) shows how the Context number is used in selecting the register number in relative mode. The register number in [Table 48](#) 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 48. 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 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 20 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 20. Use of Indexed Addressing Mode

| 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 |
| 3 | 0x0c | 0x0d | 0x0e | 0x0f |
| 4 | 0x10 | 0x11 | 0x12 | 0x13 |

Example 20. Use of Indexed Addressing Mode

| Transfer Register # | Data | | | |
|---------------------|-------|-------|------|------|
| | 31:24 | 23:16 | 15:8 | 7:0 |
| 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, *t_index++]
byte_align_be[dest2, *t_index++]
; etc
```

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

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. There is also a CAM in the Execution Datapath.

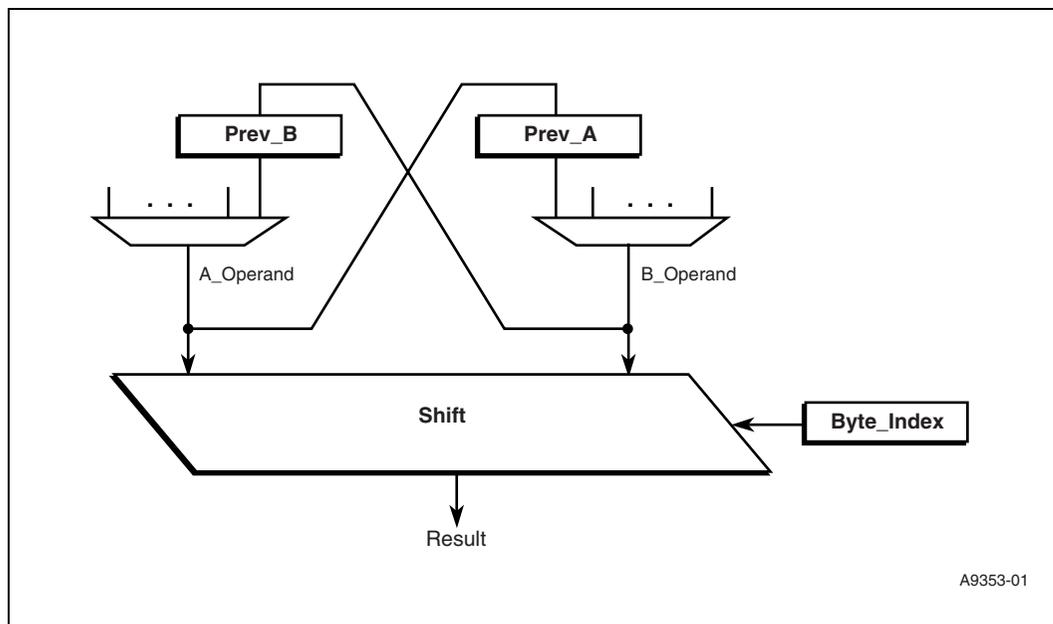
4.3.1 Byte Align

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 49. 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 71. The alignment is controlled by the 2 LSBs of the Byte_Index Local CSR.

Table 49. 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 71. Byte Align Block Diagram



Example 21 shows an align sequence of instructions and the value of the various operands. Table 50 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 50. Register Contents for Example 21

| 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 21. 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 22 shows another sequence of instructions and the value of the various operands. Table 51, 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 51. Register Contents for Example 22

| 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 22. 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 72](#) 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:

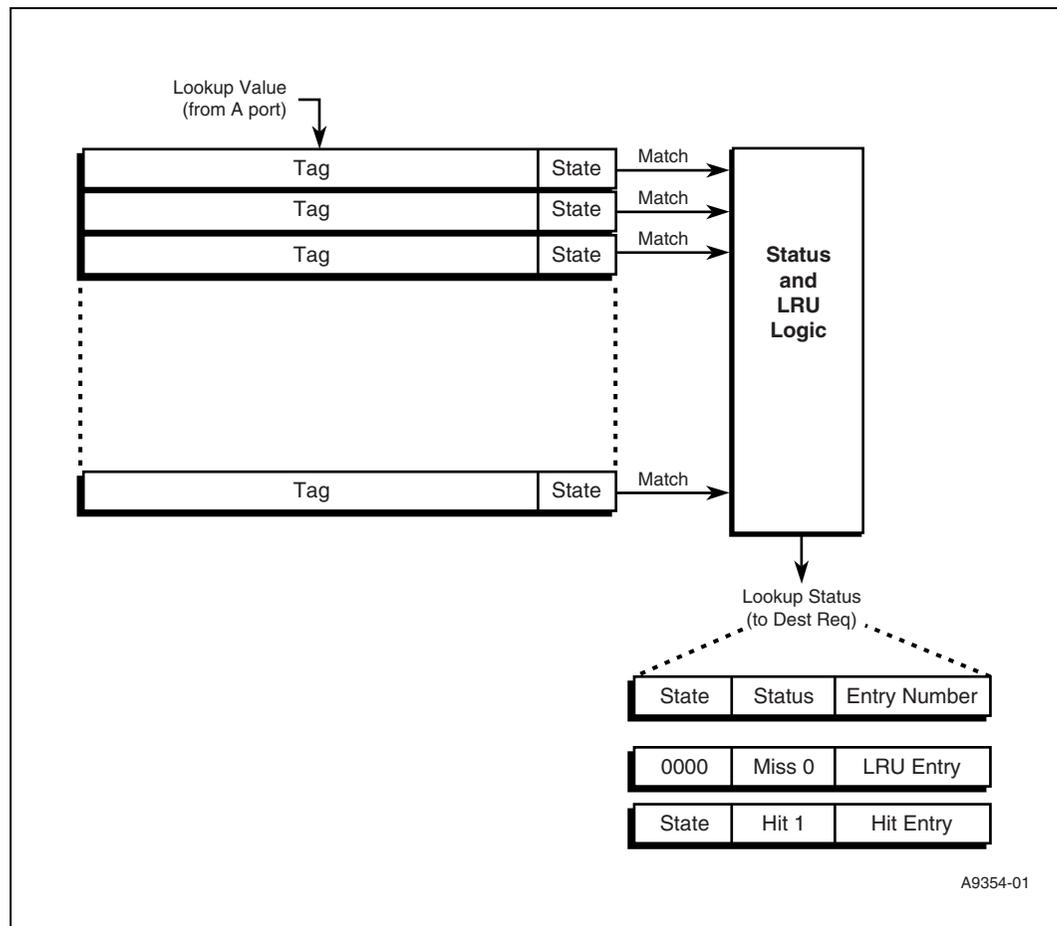
```
CAM_Lookup[dest_reg, source_reg]
```

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 72. CAM Block Diagram



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.

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 eight 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.

- 1) 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`.
- 2) 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 52](#).

Table 52. 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 and CRC-32 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 endianness point of view can be divided in following categories:

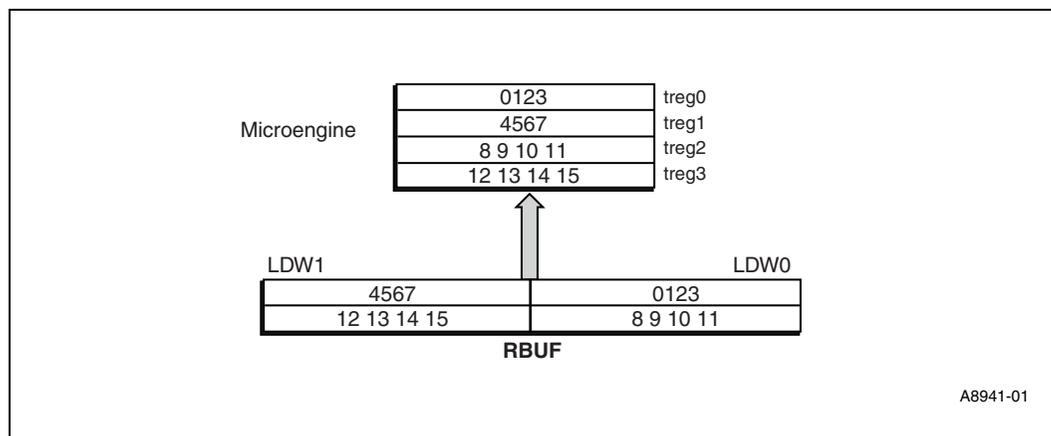
- Read from RBUF (64-bits)

- 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 (long word big endian) 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 73.

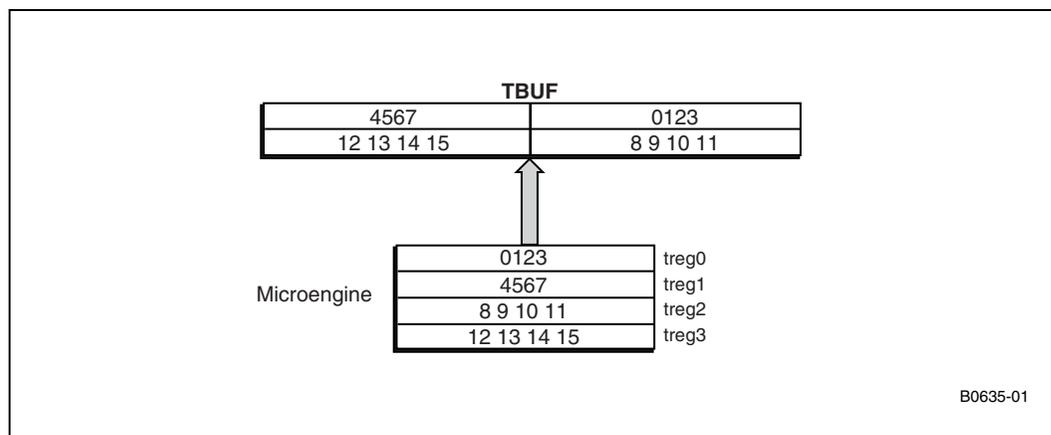
Figure 73. Read from RBUF (64-bits)



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 74.

Figure 74. 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.

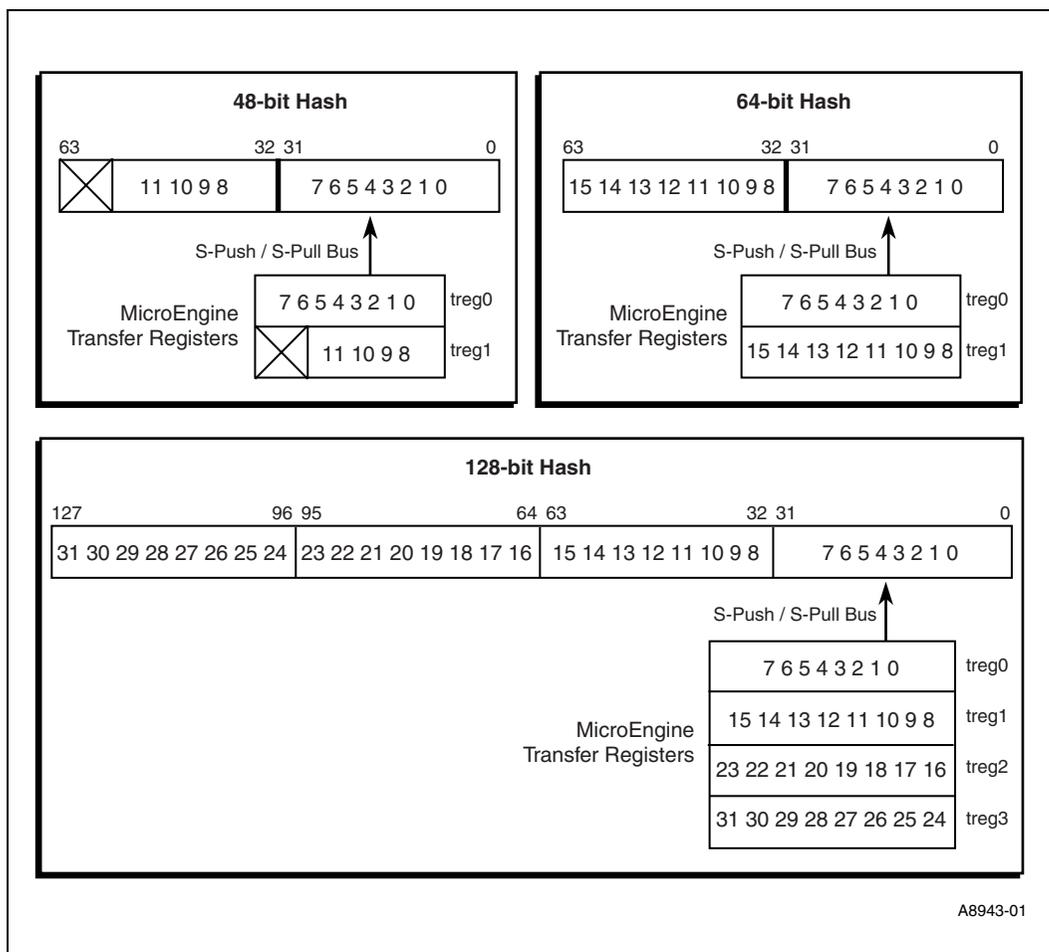
4.5.1.5 Read/Write from/to SHAC and Other CSRs

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 75](#) explains 48-bits, 64-bits and 128-bits hash operation. When Microengine transfers 48 bit hash operand to hash unit, operand resides in two transfer registers and is transferred as shown in [Figure 75](#). In the second long word transfer, only lower half is valid. Hash unit concatenates the two long words as shown in [Figure 75](#). Similarly 64-bits and 128-bits hash operand transfer from Microengine to hash unit happen as shown in the [Figure 75](#).

Figure 75. 48-bit, 64-bit and 128-bits Hash Operand Transfer



4.6 Summary of the Differences Between MEv2 and MEv1

This section documents the changes between MEv2 and MEv1. The purpose of this section is to provide those familiar with MEv1 with a quick review of removed, added, or modified features. Full descriptions of added features are in other sections.

4.6.1 General Purpose Registers and Transfer Registers

MEv2 has more GPRs and Transfer Registers than MEv1. There are

- 256 General Purpose Registers (vs. 128)
- 128 S Read Transfer Registers (vs. 32)
- 128 S Write Transfer Registers (vs. 32)
- 128 D Read Transfer Registers (vs. 32)

- 128 D Write Transfer Registers (vs. 32)

4.6.2 Next Neighbor Registers

MEv2 has 128 Next Neighbor Registers, vs. none for MEv1.

4.6.3 Local Memory

MEv2 has 640 32-bit words of Local Memory, vs. none for MEv1.

4.6.4 Contexts

MEv2 has eight available contexts versus four in MEv1. As in the MEv1, the registers can be accessed relative to the active context, or absolutely, so that a given context can access all registers. There is also a 4-context mode which partitions the registers four ways (doubling the number of Context-Relative registers from 8-context mode).

4.6.5 Larger Microstore

MEv2 provides 4K instructions, vs. 2K instructions for MEv1. The branch offset is expanded so that branch instructions can branch to any location in the Microstore.

4.6.6 CAM

MEv2 includes a 16 entry CAM with associated control logic. MEv1 does not contain a CAM.

4.6.7 Event Signals

MEv2 has 15 Event Signals per Context, that can be dynamically used in a flexible way. MEv1 has fixed binding between Event Signals and specific hardware or software events.

4.6.8 Larger Immediate Field

MEv2 provides for an 8-bit immediate field versus 5-bit for MEv1. Note that this refers to immediates used in ALU operations, not `IMMEDxx` instructions.

4.6.9 Fast Write—Wider Data Field, Access to More Registers

[Note that Command instruction is not required to be used to create Fast Write instruction. This description assumes that the implementation may contain CSRs and may use the Command instruction to access them.]

In MEv2 Fast write instruction can generate 14-bits of data in the instruction versus 10 for MEv1. In addition, there is a new variant of fast write (`fast_write_alu`) that can generate 32-bits of data, by using the ALU result of the previous instruction as the data.

4.6.10 Local CSR Instruction Uses Absolute/Relative Register Addressing

MEv2 Local CSR accesses use Absolute/Relative Register Addressing, vs. only Context-Relative Register Addressing in MEv1.

4.6.11 Timestamp

MEv2 has a Timestamp Local CSR. MEv1 does not have a Timestamp built, however there is a globally accessible Timestamp in the IXP1200 chip.

4.6.12 Future_Count Event

In MEv2 each Context has a Future_Count Local CSR which can be programmed to generate an event signal to the Context.

4.6.13 Multiply Hardware Support

MEv2 has multiply hardware support, which can retire 8-bits per cycle. MEv1 has +IFsign ALU instruction, which has been deleted from MEv2.

4.6.14 No +IFsign ALU Opcode

MEv2's 8-bit per cycle multiply hardware makes the +IFsign not useful, so it has been eliminated.

4.6.15 New Find First Bit Instruction

MEv2 adds new Find First Bit instructions in place of one already existing in MEv1. The reasons this is done relative to the existing instruction are:

1. Ability to test 32 bits at a time (vs. 16).
2. Better latency to use result (because existing datapath bypass infrastructure is used).

4.6.16 ctx_arb[bpt]

A new type of ctx_arb instruction has been added for use by debugger software. The operation is described in ctx_arb instruction section.

4.6.17 Pseudo-Random Number CSR

MEv2 has a Pseudo_Random_Number Local CSR. MEv1 has no special hardware support for random number generation.

4.6.18 Local CSR Access for External Agents

MEv1 allows Local CSR access from external agents or by code running on the Microengine. External accesses are typically done for debug, such as adding and removing breakpoints, and code profiling, for example histogramming of active context and PC by periodically reading those Local CSRs. However, there is only one port to the Local CSRs. If there is a simultaneous access (to *any* registers, not necessarily the exact same register) the external agent will take priority and the Microengine access will be lost. In MEv1 this is not normally a problem, since there is little reason for Microengine code to access registers.

In MEv2, there are several new Local CSRs that will be frequently accessed during some applications (for example the Pseudo-Random Number, and the Future Count). There is a new mechanism to allow for both external agents and MEv2 code to access Local CSRs reliably. It is described in the `Local_CSR_Status` CSR description.

4.6.19 New Branch Types to Support Signed and Unsigned Numbers

MEv2 adds more Branch on Condition Codes types relative to MEv1.

4.6.20 Branch Guess Taken Has No Effect in Hardware

The implementation cost vs. frequency of usage does not justify continued implementation of branch guess taken. Decision tree branches (such as header parsing) normally go forward, and thus do not benefit from guess taken. Loop ending branches can be replaced by unconditional branches at the top of the loop followed by a conditional branch taken prior to re-entering the loop.

4.6.21 No Shift with Add or XOR

In order to allow for faster cycle times on the MEv2, the operation of shifting and adding or subtracting in one cycle is disallowed. Shift with logicals is allowed, with the exception of XOR. A shift and add/subtract can be replaced by two instructions; first to shift and second to add/subtract.

4.6.22 No A + 4 B ALU Opcode

The A + 4 B opcode has been eliminated from MEv2. It had limited (or non-existent) utility, and required a special nibble select in the datapath. Note that A + 8 B and A + 16 B both still exist.

4.6.23 Local_CSR_Rd Returns 32 Bits

In MEv1 `local_csr_rd` returns only 16 bits (during the following `immed` instruction). MEv2 returns 32 bits.

4.6.24 Profile Count Local CSR

MEv2 has a counter that can be used for profiling code. MEv1 does not have one.

4.6.25 CSR_CTX_Pointer

MEv1 provided a separate Local CSR address for all of the per-Context Local CSRs. MEv2 does not provide separate addresses. Instead, each of the per-Context Local CSRs can be accessed in two ways; one address accesses the active Context's copy of the register; another address accesses the register for the Context number in **CSR_CTX_Pointer** Local CSR.

This page intentionally left blank.

DDR SDRAM Controller

5

5.1 Overview

The DDR SDRAM Controller is responsible for controlling the off-chip DRAM and provides a mechanism for other functional units in IXP2400 to access the DRAM. IXP2400 supports a single 64-bit channel (72-bit with ECC) of DRAM. DRAM sizes of 64, 128, 256, 512 Mb, and 1 Gb are supported. The DRAM channel can be populated with either a single or double sided DIMM.

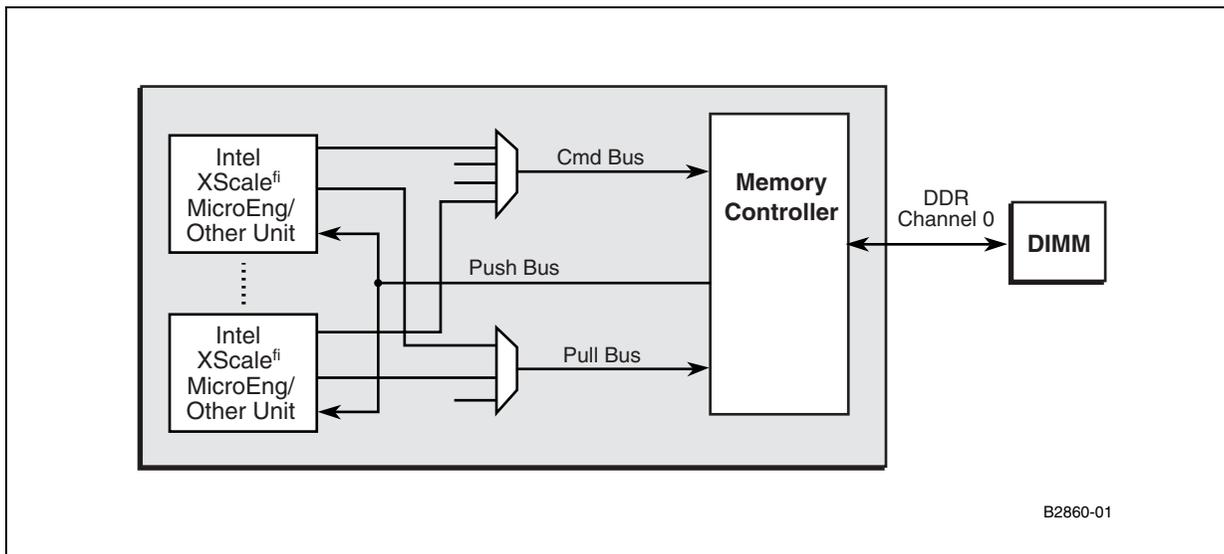
An address space of 2 GB is allocated to DRAM. The memory space is guaranteed to be contiguous from a software perspective. If less than 2 GB of memory is present, the upper part of the address space is aliased into the lower part of the address space and should not be used by software.

Reads and writes to DRAM are generated by Microengines, Intel XScale® core, and PCI bus masters. They are connected to the controllers via the Command Bus and Push and Pull Buses. The memory controller takes commands from these sources and enqueues them. The commands are dequeued—according to the priority defined later in this chapter—and the accesses to the DRAM are performed. The controller also does refresh cycles to the DRAMs.

ECC (Error Correcting Code) is supported, but can be disabled. Enabling ECC requires that x72 DIMMs be used. If ECC is disabled, x64 DIMMs can be used.

Figure 76 illustrates how the memory controllers communicate with other units and with the DDR DRAM.

Figure 76. Memory Controller's Communications



5.2 Feature List

- Supports One DDR SDRAM channels, 64 bits wide (72 bits with ECC)
- Supports DDR devices up to 300 MTs
- Supports 64-, 128-, 256-, 512-Mb, and 1-Gb, technologies for x8 and x16 devices (DIMM and direct soldered)
- Hardware controlled interleaving is done to spread contiguous addresses across multiple banks
- All supported devices have four banks
- Configurable optional Error Correction using ECC bits
- Supports one single- or double-sided DIMM
- Supports up to 2-Gb memory capacity (using 1-Gb technology)

Table 53. DDR Memory Auto Precharge Options

| Stepping | Description |
|---------------|--|
| IXP2400 A0/A1 | Supports only DDR memories with Concurrent Auto Precharge. |
| IXP2400 B0 | Supports DDR memories with or without Concurrent Auto Precharge. Set bit 30 (DIS_CAP) of the DU_CONTROL register to 1 to support DRAMs that do not feature optional Concurrent Auto Precharge. |

5.3 Configurations

Table 54 shows the memory that can be supported by the DDR channel in IXP2400. The first column shows the total memory capacity that can be supported. Each row represents different configuration options that are available to support a desired memory capacity.

Notes for Table 54:

- x16 parts can be used but the resulting datapath width is 80 bits
- 8 bits of the 80 bit x16 part widths must be discarded to accommodate the 72 bit wide data and ECC bus

Table 54. Supported Configurations

| Mem Capacity | DRAM Density | Part Width | Total Num of SDRAMs | Num of DIMMs | Num of Sides | Comments (sample DIMM vendors shown if information available) |
|--------------|--------------|------------|---------------------|--------------|--------------|---|
| 16 MB | 64 Mbit | x32 | 2 | 1 | 1 | No ECC and No Parity IXP2400 B0 |
| 32 MB | 128 Mbit | x32 | 2 | 1 | 1 | No ECC and No Parity IXP2400 B0 |
| 64 MB | 64 Mbit | x8 | 9 | 1 | 1 | |
| | 128 Mbit | x16 | 5 | 1 | 1 | |

Table 54. Supported Configurations (Continued)

| Mem Capacity | DRAM Density | Part Width | Total Num of SDRAMs | Num of DIMMs | Num of Sides | Comments (sample DIMM vendors shown if information available) |
|--------------|--------------|------------|---------------------|--------------|--------------|---|
| 128 MB | 64 Mbit | x8 | 18 | 1 | 2 | |
| | 128 Mbit | x8 | 9 | 1 | 1 | Samsung, Micron |
| | 128 Mbit | x16 | 10 | 1 | 2 | |
| | 256 Mbit | x16 | 5 | 1 | 1 | |
| 256 MB | 128 Mbit | x8 | 18 | 1 | 2 | Samsung, Micron |
| | 256 Mbit | x8 | 9 | 1 | 1 | Samsung |
| | 256 Mbit | x16 | 10 | 1 | 2 | |
| | 512 Mbit | x16 | 5 | 1 | 1 | |
| 512 MB | 256 Mbit | x8 | 18 | 1 | 2 | Samsung, Micron (3Q2001) |
| | 512 Mbit | x8 | 9 | 1 | 1 | |
| | 512 Mbit | x16 | 10 | 1 | 2 | |
| | 1 Gbit | x16 | 5 | 1 | 1 | |
| 1 GB | 512 Mbit | x8 | 18 | 1 | 2 | |
| | 1 Gbit | x8 | 9 | 1 | 1 | |
| | 1 Gbit | x16 | 10 | 1 | 2 | |
| 2 GB | 1 Gbit | x8 | 18 | 1 | 2 | |

5.4 Initialization

The DDR DRAM DIMMs contain a serial PROM which contains information about DIMM size, density, speed etc. This serial PROM is read by software running on the Intel XScale® core processor in order to configure the memory controller. GPIO pins will be used to read the PROM.

5.5 Supported Frequencies

The DRAM controller implements a few clock ratios in order to support different DRAM speeds and internal clock frequencies. The clock ratios supported are 1:1 and 3:2. [Table 55](#) lists the frequency targets for IXP2400 and the corresponding DRAM frequency (data transfer rate).

Table 55. Clock Frequencies

| | Internal Bus Frequency | Clock Ratio | DRAM Freq (Data transfer rate) |
|---------|------------------------|-------------|--------------------------------|
| IXP2400 | 300 | 1:1 | 300 MTS |
| | | 3:2 | 200 MTS |

5.6 Interleaving

In IXP2400, all accesses are directed to the single available channel. The maximum DIMM size supported is 2 GB, which is the maximum address space.

The DRAM memory banks are interleaved to improve concurrency and bandwidth utilization. Contiguous addresses are directed to different DRAM banks by remapping the physical address bits such that the new addresses are spread across the 4 DRAM banks. Note that the mapping of addresses to memory banks is completely transparent to software. Software deals with physical addresses in DRAM space; the mapping is done completely by hardware.

5.6.1 Interleaving across Banks

The addresses are interleaved across internal banks and DIMM sides. This improves memory utilization since certain operations to different banks can be performed concurrently.

Bits 8:7 of the command address are used as the bank select bits.

Bit 9 of the command address is used to select the side of the DIMM if a double sided DIMM is used.

5.7 Error Correction

Each 64-bit bank of DRAM is protected with an 8-bit SEC/DED ECC (Single Error Correct, Double Error Detect Error Correction Code.) Every store to the DRAM will cause the ECC to be calculated and stored with the data. Stores of less than 64 bits will incur a Read-Modify-Write penalty to generate the correct ECC. If ECC checking is enabled, then any data read with incorrect ECC will be corrected if the data had a single bit in error and a correctable error interrupt is generated. If two bits were in error, an uncorrectable error interrupt is generated. In both cases status is captured. The hardware does not guarantee error detection if more than 2 bits in a 64 bit data chunk are corrupted. If ECC checking is disabled the integrity of the data is unknown.

There is an address register, a status register and some control bits in the DRAM control register associated with memory integrity reporting. Control bits enable ECC checking, and allow diagnostic software to force error conditions to test the detection and reporting logic. An error address register and an error status register capture the address and syndrome of the first error to occur. That information is frozen in place until software has re-armed capture. The status register also has bits indicating UE (Uncorrectable Error) and CE (Correctable Error) to assist the program in determining the type of error seen as well as the ID of the initiator of the transaction and a bit indicating if this was a RMW cycle.

If the registers hold status and address for a CE, the first UE to occur will overwrite that information. The CE bit remains set to indicate that this occurred. If a UE happens first, but a CE is detected before the UE status is released then the CE bit sets but the UE status and address are unaffected, again to indicate that both had occurred. The DRAM Controller will assert the *Correctable Error* Interrupt if a correctable error occurs and the *Uncorrectable Error* Interrupt if an uncorrectable error occurs. See [Table 56](#).

Table 56. DRAM Error Status

| Sequence of Events | CE Bit | UE Bit | Address, Error Syndrome, Other Status |
|---|--------|--------|---------------------------------------|
| No error has occurred | 0 | 0 | x |
| Correctable Error has occurred | 1 | 0 | From first CE |
| Uncorrectable Error has occurred | 0 | 1 | From first UE |
| Correctable Error followed by a Uncorrectable Error | 1 | 1 | From first UE |
| Uncorrectable Error followed by a Correctable Error | 1 | 1 | From first UE |

DRAM scrub is not supported in hardware. The mechanism to correct errors is to have an ME do a write to the error address with all bytes masked. This will have the effect of causing an RMW and writing the corrected data back to DRAM.

5.8 Supported Requests

5.8.1 Reads and Writes

The DRAM controller supports read and write burst accesses of 16-128 bytes in multiples of 8 bytes starting on any 8-byte boundary, and single accesses of 8 bytes on an 8-byte boundary. Read accesses smaller than 8 bytes always return 8 bytes on the 8-byte-wide bus; the requesting unit is responsible for extracting the data it requested. Write accesses smaller than 8 bytes cause a read-modify-write cycle to merge new data and to generate correct ECC. If the read data had a correctable error, the data is corrected prior to the merge. These partial write accesses can come from Microengines using a write mask, from PCI, or from the Intel XScale® core issuing non-cached accesses. Write accesses smaller than 8 bytes will cause a read-modify-write cycle even if ECC is disabled. Cache writebacks from Intel XScale® core are aligned in units of 16 bytes, and cache fills are aligned reads of 32 bytes.

The Microengines access DRAM using the *dram* instruction with various options. Intel XScale® core uses the DRAM address space in the memory map to access DRAM. PCI accesses DRAM as a target and via DMA channels.

5.8.2 Register Access

There are a number of DRAM related Status and Control Registers which are located in the DRAM controller. Read and write operations to these Registers are supported. All the registers and the read/write operations are 4 bytes. However, due to the 64 bit data bus used by the DDR controller, all the register addresses are aligned on an 8 byte boundary.

To summarize, the DRAM controller can accept the request types listed in [Table 57](#).

Table 57. Supported Requests

| Request Type | Start Address Alignment | Min Length (bytes) | Max Length (bytes) | Increment Size (bytes) | Byte Selectable | Internal Buses used by transaction | | |
|----------------|-------------------------|--------------------|--------------------|------------------------|-----------------|------------------------------------|------|------|
| | | | | | | Cmd | Push | Pull |
| Read | 8B Boundary | 8B | 128B | 8B | No | Yes | Yes | No |
| Write | 8B Boundary | 16B | 128B | 8B | No | Yes | No | Yes |
| Write | 8B Boundary | 8B | 8B | N/A | Yes | Yes | No | Yes |
| Register Read | 4B Boundary | 4B | 4B | N/A | No | Yes | Yes | No |
| Register Write | 4B Boundary | 4B | 4B | N/A | No | Yes | No | Yes |

5.9 Microengine Signals

The Microengine tags each memory request with two signal numbers. The memory controller sends these signals back to the Microengine to indicate completion of the associated request. The Microengine waits for both the signals to be delivered in order to determine whether the request has been completed.

For memory reads, the signals are delivered when the data has been written into the transfer registers. This is implemented by including a *Signal Done* field in the Push Command.

For memory writes, the signals are delivered when the data is *pulled* out of the Microengine transfer registers. The Pull Command includes a *Signal Done* field in order to implement this functionality.

It is possible that a request could get split across two banks. In this case, the different components of the request could complete in any order. The signal protocol guarantees that a signal is delivered to the Microengine only when all the component parts of a request have completed. This is achieved by having each component of the request return one signal when it is done. The Microengine waits for 2 signals to arrive and thus will consider the request complete only after both components of the request have completed.

If the request is contained within one bank, the memory controller will return only 1 signal to the Microengine, and will assert a signal indicating No Split, when the request is complete. The arbiter uses the No Split signal to indicate to the ME that two signals should be set. From a microcode viewpoint there is no difference in behavior between requests that get split across banks and those that are contained within a single bank.

5.10 Read/Write Ordering Requirements

Reads/writes are not guaranteed to finish in order with respect to other reads/writes unless they are to the same address. If ordering is needed, it must be guaranteed by the command initiator. Refer to Table 58.

Table 58. Ordering Requirements

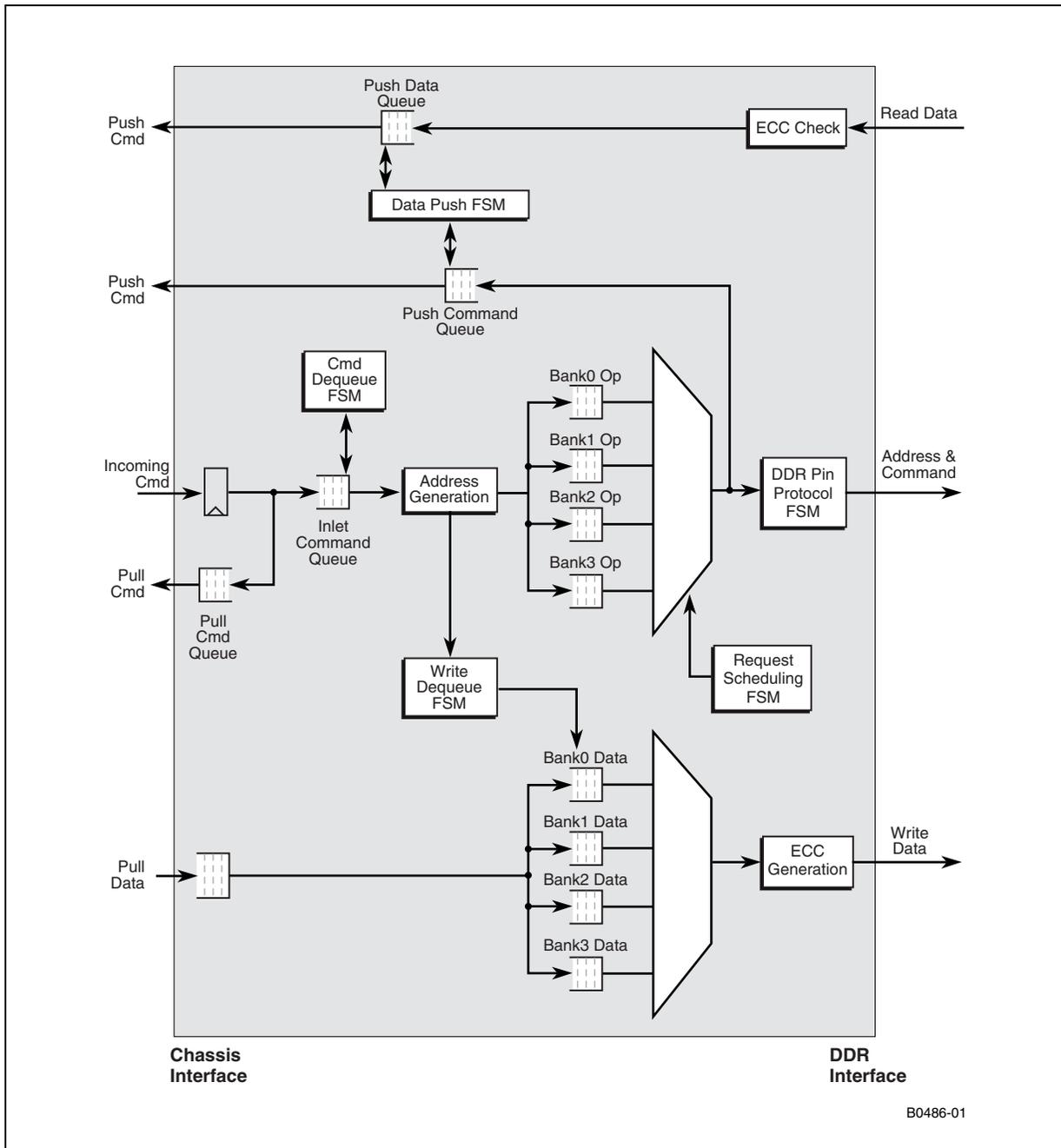
| First Access | Second Access | Ordering Requirement | Implementation Note |
|--------------|---------------|---|---|
| Read | Read | None. Reads to DRAM have no side effects, both readers will get the same data. | Reads which access the same location in memory are not re-ordered by the DRAM controller. |
| Read | Write | The read must return the pre-modified data. However, this does not have to be enforced by Hardware. For ME, software is responsible for issuing the Write only after the read is complete. | Reads and writes to the same address are issued in order. Therefore the write after read hazard is not present. |
| Write | Read | Read must return modified data. The read can be issued by the requestor when it receives the done signal for the write. The done signal for the write is sent when the write data is pulled out of the requestor (and not when the write data is actually written into DRAM). | Reads and writes to the same address are issued in order. Therefore the read after write hazard is not present. |
| Write | Write | No ordering is guaranteed. | The DRAM controller will issue writes to a particular address in the order that they were received. |

5.11 Design Overview

This section provides a brief overview of the implementation.

Figure 77 is a block diagram of the DRAM controller.

Figure 77. DRAM Controller



The following sections briefly describe the flow of a typical read and write request through the DRAM controller. The DRAM controller also supports CSR accesses and partial writes.

5.11.1 Read Requests

- **Incoming Command:** A requestor initiates a DRAM access by sending a command on the Cmd bus. IXP2400 has one command bus interfacing with the DRAM controller. The DRAM controller can receive only one command in any given clock cycle. Flow control is implemented as follows: the DRAM controller sends an *Almost Full* signal to the Cmd bus arbiter which, if asserted, prevents further requests from being generated.
- **Enqueue Command:** The incoming command is received by the DRAM controller. The controller inspects the Target_ID of the request to check if it targets the DRAM. The command is enqueued only if there is a match.
- **Op Generation:** When the request reaches the head of the Cmd FIFO, it is inspected to determine if it needs to be split into multiple DRAM ops. Splitting of requests is needed if the data requested by the read cmd spans multiple banks, rows or DRAM burst boundaries. If needed, the request is split and multiple DRAM ops are generated and latched. The address generation block uses the start address and length fields of the original request to generate new addresses for the DRAM ops.
- **Op Scheduling Queues:** The DRAM Ops are then enqueued onto a queue based on the DRAM bank that they target. There are four such queues, one for each internal DRAM bank. The purpose of these queues is to help minimize bank conflicts between successive ops so that better utilization can be achieved on the DRAM data bus. Separating Ops on the basis of bank (as opposed to read or write type) simplifies the design as a CAM is not needed to detect reads and writes with conflicting addresses. The *Request Scheduling FSM* picks ops from these queues and schedules them on to the Pin FSM. Also, these ops are enqueued onto the *Push Cmd Queue* to keep track of the returning data.
- **Pin FSM:** The pin FSM checks that all constraints specified by the DDR DRAM pin protocol (such as RAS to RAS delay etc.) are satisfied before it issues an Activate command for this read op. The pin FSM keeps track of this op and issues a read command when appropriate.
- **Data Return:** When the DRAM returns data, it is put into the *Push Data Queue*. The *Push Cmd Queue* then arbitrates for access to the Push Bus in order to return the data. On being granted the Push bus, the data is returned to the requesting agent. Each 64 bit data transfer is sent as a separate push with the push_id incremented for each 64 bit transfer. Also note that data is not necessarily returned in the same order that the read requests were made since the Op scheduling queues can reorder the reads.

5.11.2 Write Requests

Write requests go through essentially the same steps, with some differences:

- **Incoming Command:** A requestor initiates a DRAM access by sending a command on the Cmd bus. IXP2400 has one command bus interfacing with the DRAM controller. The DRAM controller can receive only one command in any given clock cycle. Flow control is implemented as follows: the DRAM controller sends an *almost full* signal to the Cmd bus arbiter which, if asserted, prevents further requests from being generated.
- **Enqueue Command:** The incoming command is received by the DRAM controller. The controller inspects the Target_ID of the request to check if it targets the DRAM. The command is enqueued only if there is a match.
- **Pull Request:** When the write command reaches the head of the Inlet queue and if there is room in the pull data fifo, a request is made to the Pull arbiter to transfer the write data from the Data source to the DRAM controller. If the pull data fifo does not have enough space, the request is stalled until space becomes available. The Pull arbiter will forward the request to the

data source and data will be written into the DRAM controller's pull data fifo when available. Completion signals are sent as part of the pull command.

- **Op Generation:** Concurrent with making the Pull request, the write command is inspected to determine if it needs to be split into multiple DRAM ops. Splitting of requests is needed if the data requested by the write cmd spans multiple banks, rows or DRAM burst boundaries. If needed, the request is split and multiple DRAM ops are generated and latched. The address generation block uses the start address and length fields of the original request to generate new addresses for the DRAM ops.
- **Op Scheduling Queues:** The DRAM Ops are then enqueued onto a queue based on the DRAM bank that they target. The *Request Scheduling FSM* picks ops from these queues and schedules them on to the Pin FSM. Note that the DRAM Ops are enqueued into the bank queues even if the corresponding write data is not yet available. The DRAM ops will not be dequeued out of the bank queues until the write data is available. Blocking the DRAM Ops in the bank queues as opposed to the Inlet command queue has the advantage of letting reads to other banks advance.
- **Pin FSM:** The pin FSM checks that all constraints specified by the DDR DRAM pin protocol (such as RAS to RAS delay etc.) are satisfied before it issues an Activate command for this write op. The pin FSM keeps track of this op and issues a write command when appropriate.

5.11.3 Request Scheduling Algorithm

This block selects a request from the four bank queues and schedules it to the DDR pin protocol FSM. The intent here is to schedule the requests such that the DRAM data bus utilization is improved. Specifically, the Request Scheduling FSM tries to perform the following optimizations:

- **Minimize bank conflicts.** The Request Scheduling FSM attempts to pick requests from the bank queues in a round robin manner. Since the requests in the bank queues have already been sorted on the basis of the bank they target, this results in successive requests accessing different banks.
- **Minimize read-write turnarounds.** There is a performance penalty associated with switching between read and write requests. The Request Scheduling FSM attempts to schedule requests of the same type in succession so that switching between reads and writes is minimized. For example if there were a number of read and write requests which were waiting to be scheduled, the FSM would schedule 4 read requests followed by 4 write requests and so on.

The round robin pointer is initialized to point to a particular bank queue (say bank 0). Also, the logic is initialized to favor a particular type of request (say read). The algorithm selects requests in round robin order as long as all requests are reads. When a write is encountered, the *Request_Skipped_Count* counter is incremented and the pointer skips to the next (in round robin order) bank queue. If that request is a read it is selected, if it is a write the *Request_Skipped_Count* is incremented once again and the pointer is incremented.

The algorithm keeps selecting and issuing reads till the *Request_Skipped_Count* reaches a programmable threshold value (see the description of bits 7:0 of the *DU_CONTROL2* CSR). When this happens, the favored type is changed to *Write* and the *Request_Skipped_Count* is reset to 0. The algorithm now selects writes until reads are skipped enough times to flip the favored type again.

5.11.4 DDR Pin FSM

The DDR pin FSM is responsible for issuing commands on the DRAM bus. Key features are listed here.

- The pin FSM attempts to keep the pins busy by overlapping Activate and Read/Write commands of different requests.
- Writes smaller than 8 bytes result in a Read-Modify-Write (RMW) sequence. Selectively writing fewer than 8 bytes by masking the write of specific bytes is not possible since the ECC needs to be computed over the entire 8 byte quantity. Commands targeting other banks can be issued in between the read and the write of the RMW since they do not conflict with the RMW.
- The FSM has a closed page policy—a page which is activated for a read or write request is precharged at the end of the operation. Reads and writes are issued with the Auto-Precharge bit set on the last burst of a request.

5.12 Register Descriptions

5.12.1 Register Map

The DDR registers are addressed at 8 byte offsets. Each register is 32 bits and data is transferred on the low 32 bits of the DRAM Push/Pull Data Buses.

Table 59. DDR Register Map

| Abbreviation | Offset | Name | Description |
|---------------------|------------------|------------------------------------|--|
| DU_CONTROL | 0x000 | DRAM Controller Control Register | Contains programmable delay/latency parameters to support various configurations |
| DU_ERROR_STATUS_1 | 0x008 | DRAM Error Status Register 1 | Logs the Address of transaction which had an ECC error |
| DU_ERROR_STATUS_2 | 0x010 | DRAM Error Status Register 2 | Logs details about type of ECC error |
| DU_ECC_TEST | 0x018 | DRAM ECC Test Register | Has control settings which can be used to inject false ECC errors for testing purposes |
| DU_INIT | 0x020 | DRAM Initialization Register | Contains controls for the DDR Mode register set, refresh, precharge commands |
| DU_CONTROL2 | 0x028 | DRAM Controller Control Register 2 | Contains additional DRAM Controller control fields |
| - | 0x030 - 0x0F8 | - | Reserved |
| DU_IO_CONFIG[1:224] | 0x100 - 0x7F8 | DRAM IO Configuration Registers | Contains Drive strength controls for various interface pins |



This page intentionally left blank.

SRAM Interface

6

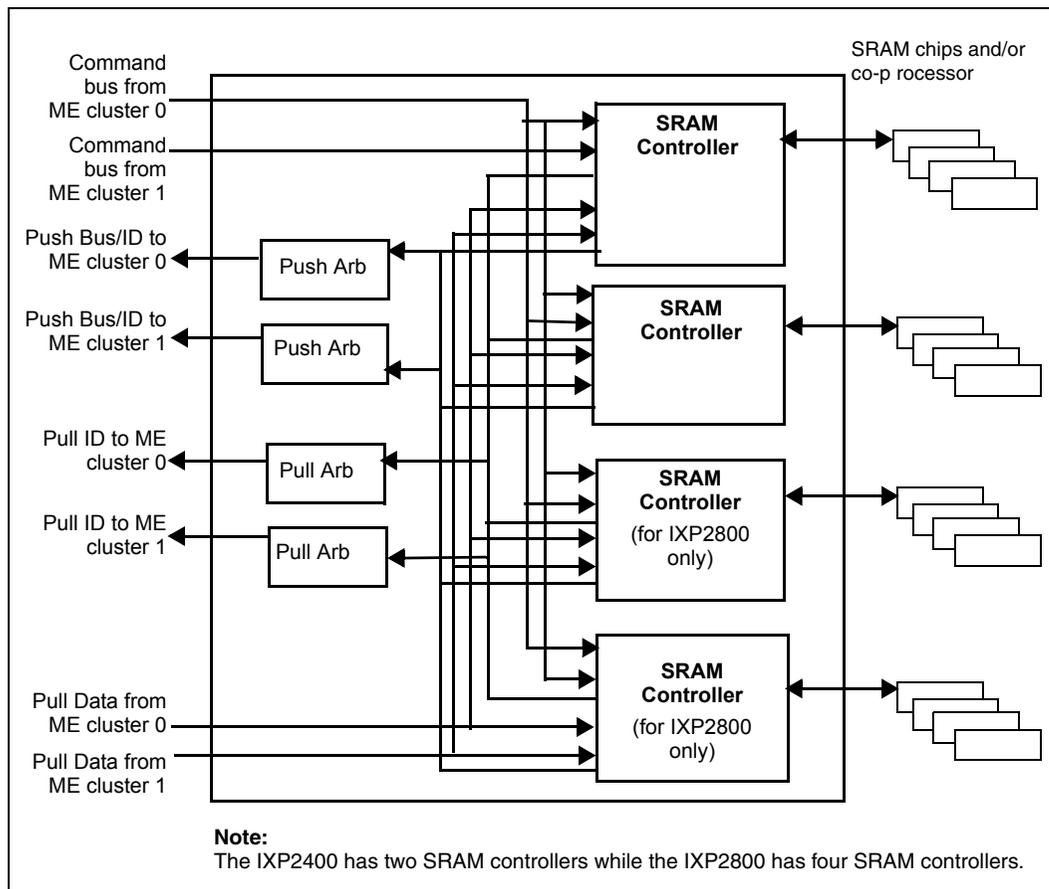
6.1 Overview

The Intel[®] IXP2400 and IXP2800 network processors contain two and four independent SRAM controllers, respectively. SRAM controllers support pipelined QDR and QDR II synchronous static RAM (SRAM) technologies and a coprocessor which 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), Intel XScale[®] core, and PCI Bus masters. They are connected to the controllers via Command Buses and Push and Pull Buses. Each SRAM controller enqueues commands from the command bus. The commands are dequeued and successive access to the SRAMs is 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 via the S-Push bus and S-Pull bus.

The overall structure of the SRAM controllers is shown in [Figure 78](#).

Figure 78. 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.

Examples of supported SRAMs are:

- Micron MT54V512H18A 9Mb QDR SRAM (512K x 18)
- IDT IDT71T6280H 9Mb Pipelined QDR SRAM Burst of 2 (512K x 18)
- Cypress CY7C1302V25 9-Mb Pipelined SRAM with QDR Architecture (512K x 18)

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.\

In general, QDR and QDR II burst of 2 SRAM will be supported at speeds up to 200 MHz. As other (larger) QDR SRAMs are introduced, they will also be supported.

Each of the 2 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.

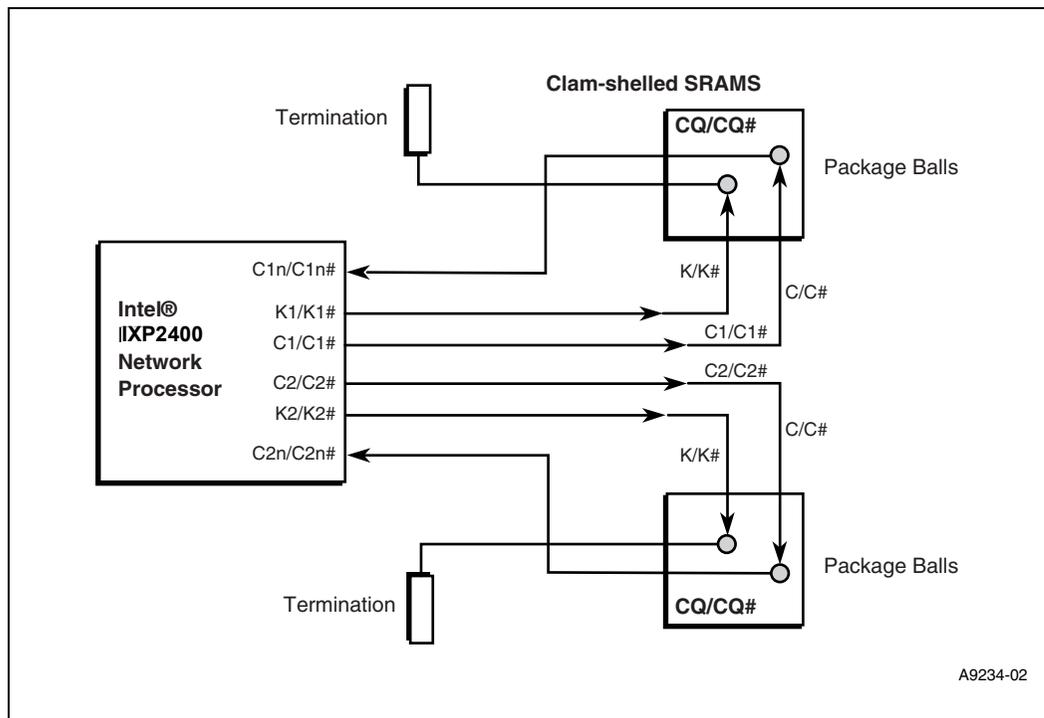
Extensive work has been performed providing impedance controls within IXP2400 for IXP2400-initiated signals driving to QDR parts. *Providing a clean signaling environment is critical to achieving 200 MHz QDRII data transfers.*

The configuration assumptions for IXP2400 IO driver/receiver development includes 4 QDR loads and IXP2400. It should be noted that some future QDRII SRAMs require a burst of 4 to achieve higher frequency. IXP2400 initial release will not support burst of 4 QDR SRAM parts. The IXP2400 Network Processor initial release supports bursts of 2 SRAMs.

The echo clocks are $C1n/Cn\#$ and $C2n/C2n\#$ (see [Figure 79](#)). IXP2400 uses one pair of the $Cn/Cn\#$ clocks for read data, the other pair is terminated on the die.

The SRAM controller can also be configured to interface to an external coprocessor that adheres to the QDR electricals and protocol.

Figure 79. Echo Clock Configuration



6.3 SRAM Interface Configurations

This section describes SRAM interface build configurations for communicating to 1 or 2 ME clusters and either 1, 2, 3, or 4 SRAM/co-processor channels to accommodate use of either IXP2400 or IXP2800 chips.

6.3.1 Internal Interface

Each SRAM channel receives commands via the command bus mechanism and transfers data to and from MEs, Intel XScale® core, and PCI via push and pull buses.

6.3.2 Number of Channels

The SRAM/coprocessor channels are supported via the instantiation of multiple SRAM controller FUBs. The IXP2800 supports 4 channels and the IXP2400 supports 2 SRAM/coprocessor 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 upon 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 5 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 60](#).

Note: All of the SRAMs on a given channel must be the same size.

Note: [Table 60](#) shows the capability of the logic—up to 4 loads will be supported, and the table reflects that information.

Table 60. 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 | 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.

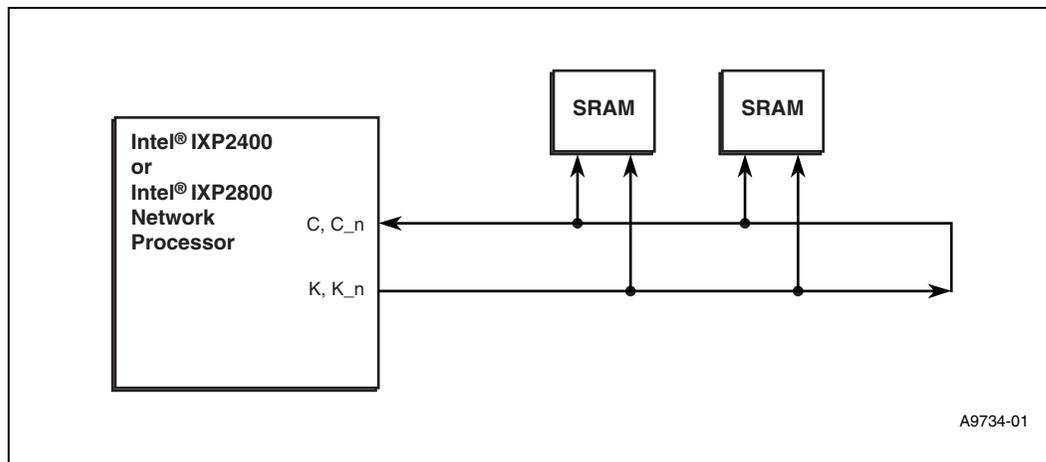
Maximum SRAM system sizes are shown in [Table 61](#). Shaded entries require external decoding, because they use more port enables than the SRAM controller can directly supply.

Table 61. 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 |

Figure 80 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 80. 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 81 is a simple block diagram showing 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 81. External Pipeline Registers Block Diagram

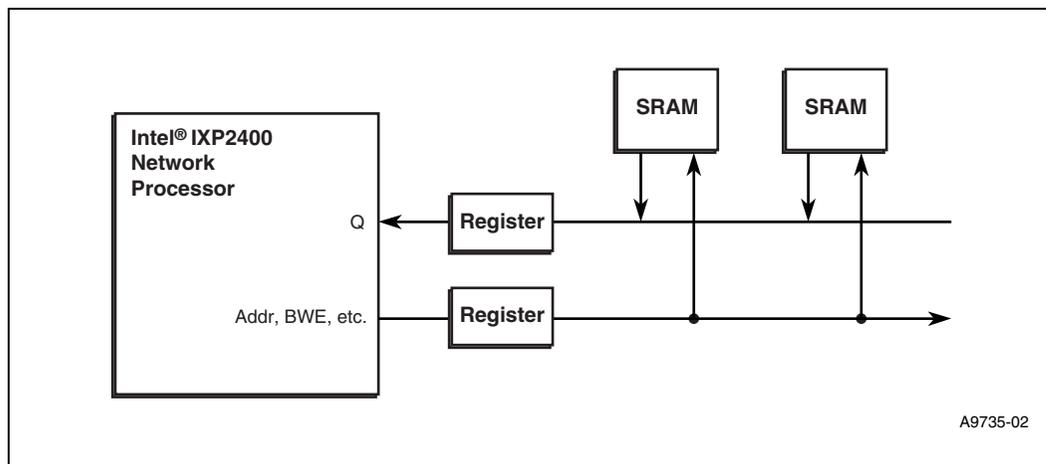


Figure 81 depicts use of external pipeline registers, which add two cycles of latency to reads.

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.

Note: A longword is a 32 bit (4 byte) data entity.

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 IXP2400 Network 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 ME) 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.

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 62](#) describes the atomic operations supported by the SRAM Controller.

Table 62. Atomic Operations

| Instruction | Pull Operand | Value Written to SRAM |
|------------------------|--------------|----------------------------------|
| Set_bits | Yes | SRAM_Read_Data OR Pull_Data |
| Clear_bits | Yes | SRAM_Read_Data AND NOT Pull_Data |
| Increment | No | SRAM_Read_Data + 0x00000001 |
| Decrement ^a | No | SRAM_Read_Data - 0x00000001 |
| Add ^{b,c} | Yes | SRAM_Read_Data + Pull_Data |
| Swap ^d | Yes | Pull_Data |

- Unsigned value that saturates at 0x00000000.
- Pull_Data is two's complement and saturates at 0x00000000 if Pull_Data is < 0.
- Return result (when enabled) can be tested to determine amount actually added or subtracted in order to detect the case where the operation saturated at zero. For example, assume the data in the addressed SRAM location is 0x10. Subtracting 0x3 (adding 0xFFFFFFF3) would return 0x10 and write 0xD into the SRAM. However, Subtracting 0x15 (adding 0xFFFFFFF3) would return 0x10 and write 0x0 into the SRAM, because the subtraction saturates at 0x0. The reader can test the return data to determine if the amount actually available is the minimum of the amount requested and the returned data.
- Swap will normally be used with return of read data enabled. Doing a Swap without return of read data will write the Pull data into memory without returning the original read data—that operation is better done by normal (non-atomic) write.

Up to two ME 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 (see [Table 62](#)), then the requested signal will be sent on the pull. If the pre-modified data is to be returned to the ME, then the ME will be sent (requested signal OR 1) when that data is pushed.

In [Example 23](#), there is both a pull and a push for an SRAM read-modify-write:

Example 23. SRAM Read-Modify-Write with Pull Data

```
IMMED [$xfer0, 0x1]
IMMED [test_address, 0x0]
SRAM [TEST_AND_SET, $xfer0, test_address, 0], SIG_DONE[SIGNAL_2]
CTX_ARB [SIGNAL_2]
; SIGNAL_2 is set when $xfer0 is pulled from this ME. SIGNAL_2+1 is
; set when $xfer0 is written with the test value. Sleep until both
; SIGNALS have arrived.
```

In [Example 24](#), there is no pull:

Example 24. SRAM Read-Modify-Write without Pull Data

```
SRAM [SET, $xfer0, test_address, 0], SIG_DONE[SIGNAL_2]
CTX_ARB [SIGNAL_2]
; SIGNAL_2 is set when $xfer0 is pulled from this ME. Sleep until that signal
arrives.
```

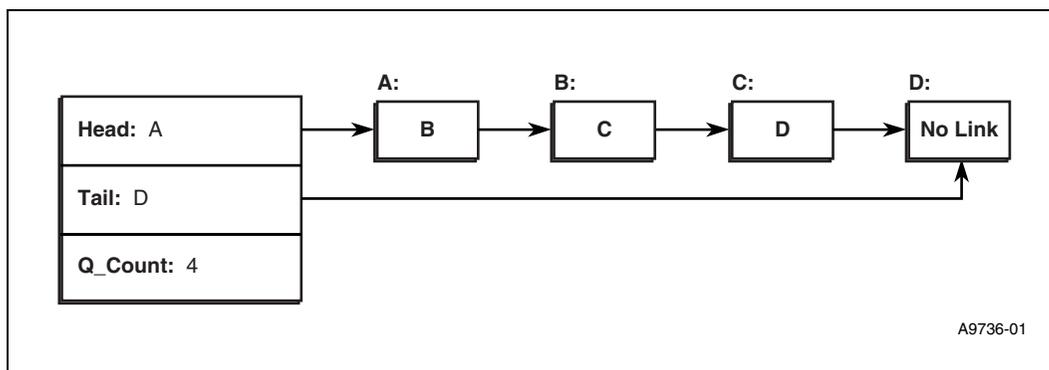
6.5.3 Queue Data Structure Commands

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 discontinuous 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 of head, tail, and count is referred to as the queue descriptor. In the IXP2400 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 82](#) shows a queue descriptor and queue links for a queue containing four entries.

Figure 82. Queue Descriptor with Four Links



There are two different versions of the enqueue command, `ENQUEUE` and `ENQUEUE_TAIL`. `ENQUEUE` is used to enqueue one buffer at a time. `ENQUEUE` followed by `ENQUEUE_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. These two versions are shown in Figure 83 and Figure 84 respectively.

Figure 83. Enqueueing One Buffer at a Time

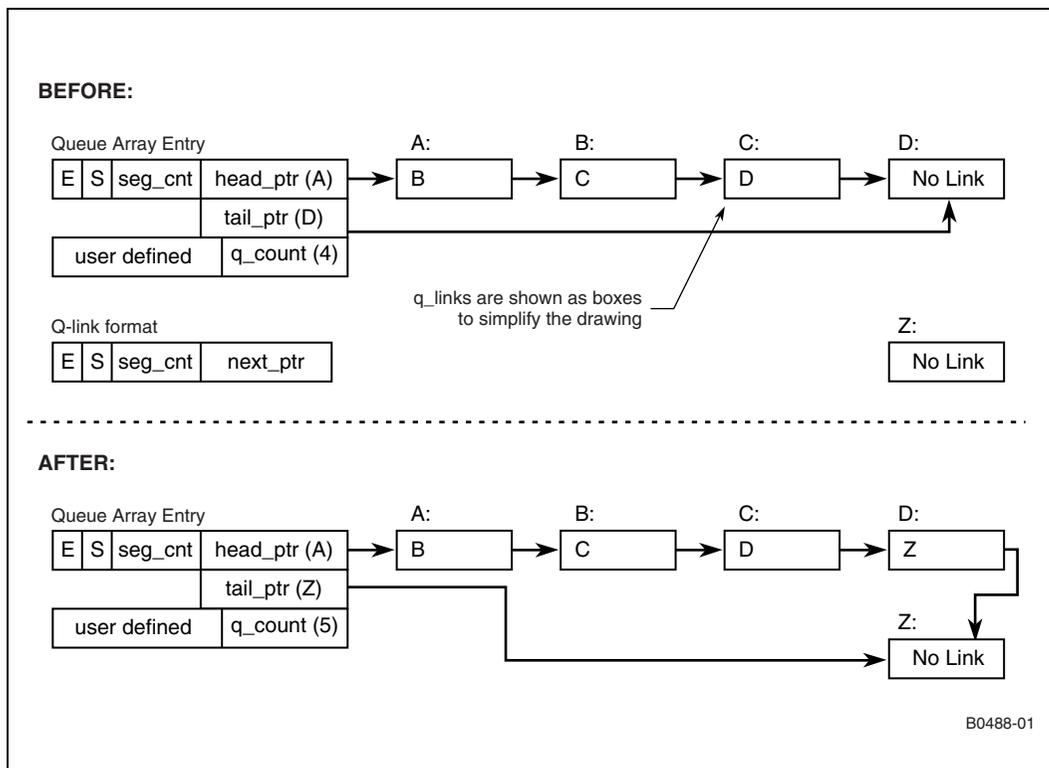
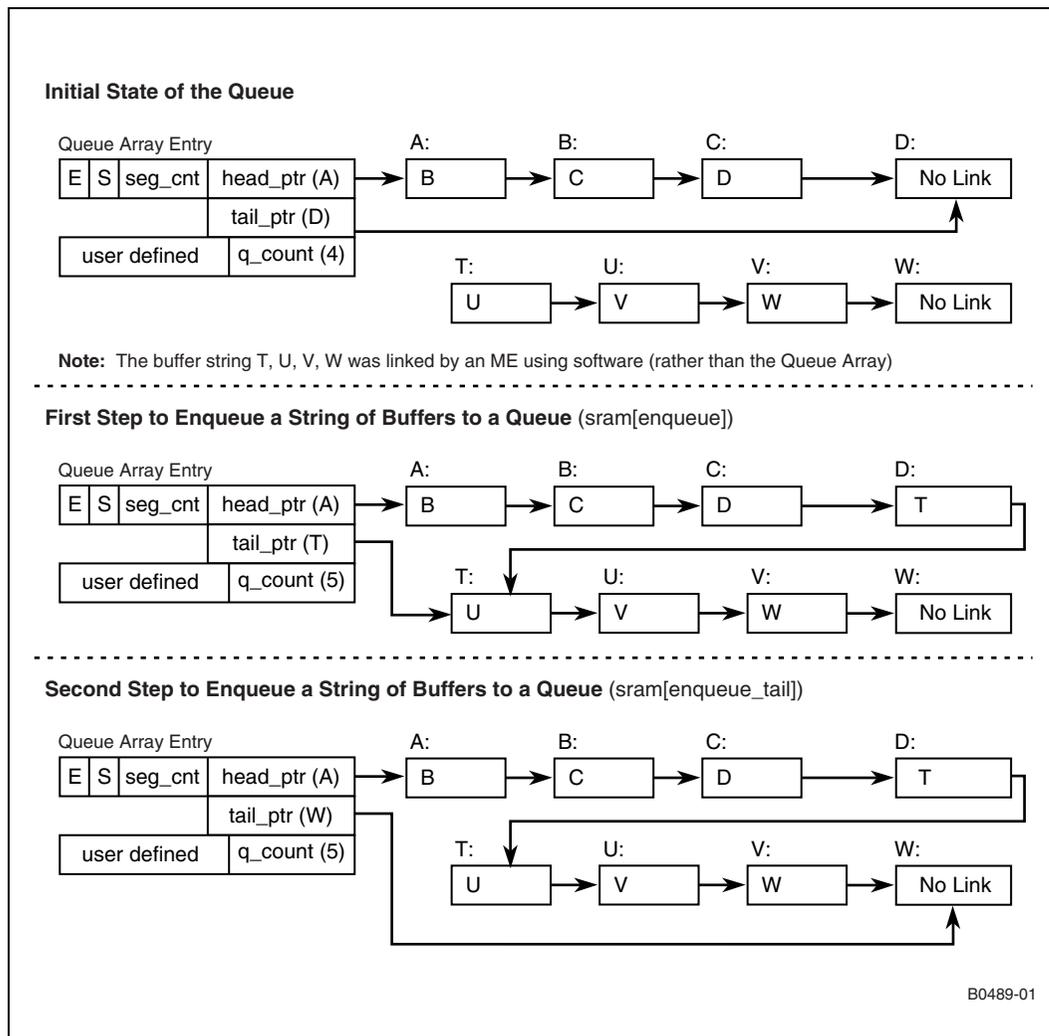


Figure 84. Enqueue a String of Buffers to a Queue



There are three different modes for dequeue command. The SRAM_CONTROL[QC_IGN_EOP:QC_IGN_SEG_CNT] register bits support the following three modes that determine the behavior of the dequeue command (Mode 2 is not supported).

Mode 0: Dequeue Segments and Count Packets

The seg_cnt is decremented for each SRAM[dequeue] command. Only when seg_cnt equals 0 is the q_link removed from the linked list. The EOP is used by hardware to determine if it should decrement the q_count, therefore it must be set on the last buffer of a packet for software designs that support multiple buffers per packet or on all buffers for software designs that support a single buffer per packet. The state of the EOP bit is returned with the data for each dequeue command. The state of the SOP bit is returned only with the data for the first dequeue command to a buffer. The SOP bit is clear for subsequent dequeue commands to the buffer.

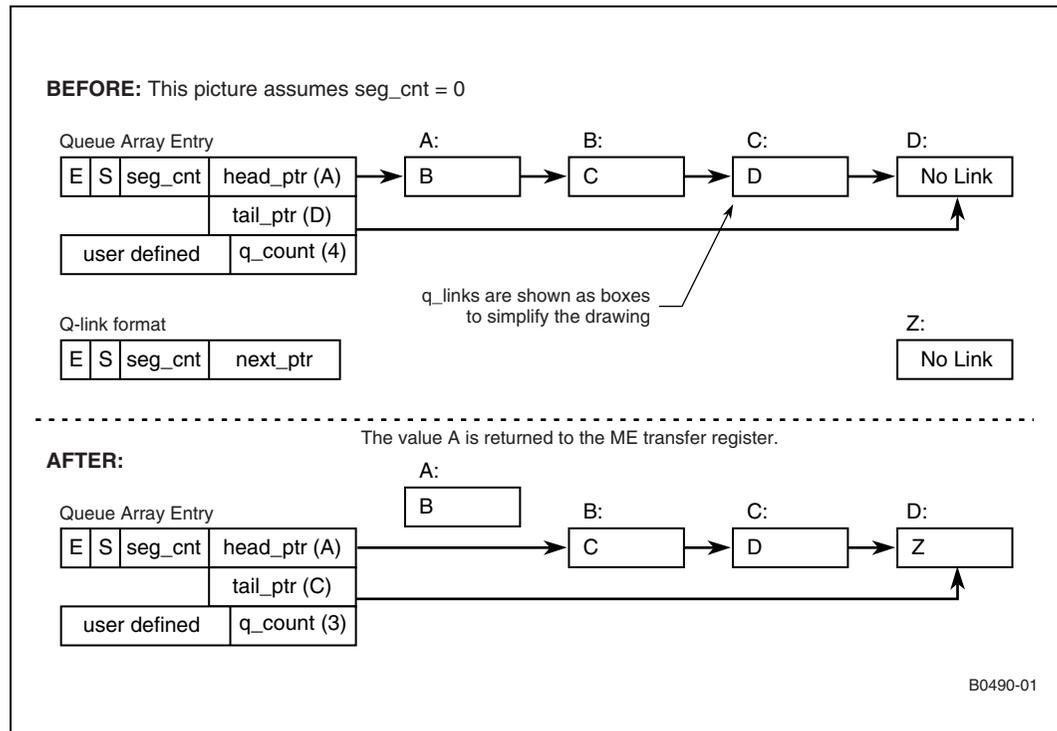
Mode 1: Dequeue Buffers and Count Packets

The `seg_cnt` is ignored in this mode so a `q_link` is removed from the linked list for each `sram[dequeue]` command. The EOP and SOP bits are treated the same as mode 1. The `seg_cnt` is returned unchanged on each dequeue.

Mode 3: Dequeue Buffers and Count Buffers

The `seg_cnt` is ignored in this mode so a `q_link` is removed from the linked list for each `sram[dequeue]` command. The EOP bit is also ignored by hardware so the `q_count` is always decremented for each dequeue command. Note: In this mode the `seg_cnt` is added to the `q_count` on every enqueue.

Figure 85. Dequeue Buffer



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 ME 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 an ME. 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 subsetted 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 63 and Table 64. All addresses are of type *longword*, and are 32 bits in length.

Table 63. Queue Format

| Name | Longword # | Bit # ^a | Definition |
|---------------|------------|--------------------|--|
| EOP | 0 | 31 | End of Packet—decrement Q_count on dequeue |
| SOP | 0 | 30 | Start of Packet |
| Segment Count | 0 | 29:24 | Number of segments in the buffer |
| Head | 0 | 23:0 | Head pointer |
| Tail | 1 | 23:0 | Tail pointer |
| Q_count | 2 | 23:0 | Number of packets on the queue or number of buffers on the queue |
| SW_Private | 2 | 31:24 | Ignored by hardware, returned to ME |
| Head Valid | N/A | | Cached head pointer valid—maintained by hardware |
| Tail Valid | N/A | | Cached tail pointer valid—maintained by hardware |

a. Bits 31:24 of longword number 2 are available for use by ucode.

Table 64. Ring/Journal Format

| Name | Longword # | Bit # | Definition |
|------------|------------|-------|----------------------------------|
| Ring Size | 0 | 31:29 | See Table 129 for size encoding. |
| Head | 0 | 23:0 | <i>Get</i> pointer |
| Tail | 1 | 23:0 | <i>Put</i> pointer |
| 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 65.

Table 65. 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. [Example 25](#) details the operations performed.

Example 25. Read_Q_Descriptor Commands

```

Rd_qdesc_head(xfer_addr, address,entry, length)
// Loads the head, EOP, SOP, segment_count, and queue_count
// for entry into the Q_array cache
head[entry]    <-- SRAM[address]<23:0>
segment_count[entry]<-- SRAM[address]<29:24>
SOP[entry]    <-- SRAM[address]<30>
EOP[entry]    <-- SRAM[address]<31>
head_valid[entry]<-- 1
ME[xfer_addr], Q_count[entry]<-- SRAM[address+2]
; optional parameter(s) sent to SRAM xfer
; registers if length >2
xfer_addr += 1
addr = address + 3
for (temp = 3, temp<=length, temp++)
    ME[xfer_addr]<-- SRAM[address]
    addr += 1
    xfer_addr += 1

Rd_qdesc_tail(xfer_addr, address,entry, length)
// Loads the tail and queue_count for entry into
// the Q_array cache

tail[entry]    <-- SRAM[address+1]
tail_valid[entry]<-- 1
ME[xfer_addr], Q_count[entry]<-- SRAM[address+2]
; optional parameter(s) sent to SRAM xfer
; registers if length > 2
xfer_addr += 1
addr = address + 3
for (temp = 3, temp<=length, temp++)
    ME[xfer_addr]<-- SRAM[addr]
    addr += 1
    xfer_addr += 1

Rd_qdesc_other(address, entry)
// Loads any missing information for line entry into
// the Q_array cache

if head_valid[entry] == 0
    begin
        head[entry]<-- SRAM[address]<23:0>
        segment_count[entry]<-- SRAM[address]<29:24>
SOP[entry]<-- SRAM[address]<30>
EOP[entry] <-- SRAM[address]<31>
        head_valid[entry]<-- 1
    end

if tail_valid[entry] == 0
    begin
        tail[entry]<-- SRAM[address+1]
        tail_valid[entry]<-- 1
    end

end

```

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 minimize QDR SRAM bandwidth utilization. [Example 26](#) describes the details of the operations performed.

Example 26. Write_Q_Descriptor Commands

```
Wr_qdesc(address, entry)
if (head_valid[entry] == 1)
  begin
    SRAM[address] <23:0>← head[entry]
    SRAM[address] <29:24>← segment_count[entry]
  SRAM[address] <30>← SOP[entry]
    SRAM[address] <31>← EOP[entry]
  end
if (tail_valid[entry] == 1)

SRAM[address+1] ← tail[entry]
SRAM[address+2]← Q_count[entry]
head_valid[entry]← 0
tail_valid[entry]← 0

Write_Q_Descriptor_Count(address, entry)
// This version is used to refresh just the Q_count in SRAM. The entry is not
evicted.
SRAM[address+2]← Q_count[entry]
```

6.5.3.3 ENQ and DEQ Commands

These commands add or remove elements from the queue structure while updating the Q_array registers. [Example 27](#) describes the details of the operations performed.

Example 27. ENQ and DEQ Commands

```

ENQ_tail_and_link( buff_desc_adr, cell_count, EOP, entry)
// Adds a buffer to the queue contained in Q_Array entry, and
// sets the tail to point to the buffer. If necessary, a link
// is established from the old tail buffer to the new buffer.
// This command is used to add an entire frame to the queue
// or to add the Start-of-Packet buffer of a multi-buffer frame
// to the queue.
If Q_count[entry]==0
begin
head[entry]          ← buff_desc_adr
cell_count[entry]   ← cell_count
EOP[entry]          ← EOP
SOP[entry]          ← SOP
head_valid[entry]   ← 1

end

If Q_count[entry] > 0
SRAM[tail[entry]]<29:24> ← buff_desc_adr
SRAM[tail[entry]]<30>   ← SOP
SRAM[tail[entry]]<31>   ← EOP
SRAM[tail[entry]]<23:0> ← cell_count

tail[entry]          ← buff_desc_adr
Q_count[entry]++

ENQ_tail( buff_desc_adr, entry)
// Updates the tail pointer only. This command must be
// preceded by a ENQ_tail_and_link to the same entry.
// This adds the End-of-Packet buffer of a multi-buffer frame
// to the queue.
tail[entry]          ← buff_desc_adr

DEQ(entry, xfer_addr)
// Removes a cell or a frame from the queue cached in line entry.
If Q_count[entry] > 0
begin
ME[xfer_addr]       ← {EOP[entry], SOP[entry], cell_count[entry],
                      head[entry]}

if cell_count[entry] == 0
begin
// update queue count only when an entire frame has been removed
if EOP[entry]
Q_count[entry]--
// load the buffer descriptor for the next buffer
cell_count[entry] ← SRAM[head[entry]]<29:24>
SOP[entry]        ← SRAM[head[entry]]<30>
EOP[entry]        ← SRAM[head[entry]]<31>
head[entry]       ← SRAM[head[entry]]<23:0>
end
else
cell_count[entry]--
end
else
// count was 0, so nil value indicates nothing was available to DEQ
ME[xfer_addr]     ← nil

```

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. [Example 28](#) describes the details of the operations performed.

Example 28. Ring Data Structure Commands

```

Get(entry, length, xfer_addr)
If count[entry] >= length //enough data in the ring?
  // Return length number of longwords
  For (temp=length, temp>0, temp--)
    ME[xfer_addr] ← SRAM[head[entry]]
    head[entry] = (head[entry] + 1) % ringSize
    count[entry] -= 1
    xfer_addr +=1
else
  ME[xfer_addr] ← nil // 1 data phase of 0 signals read off empty
list

Put(entry, length, xfer_addr)
If (ring_size - count[entry] < 16) // 16 is max value for length
  initial_xfer_addr = xfer_addr
  For (temp=length, temp>0, temp--)
    // Write length number of longwords
    SRAM[tail[entry]] ← ME[xfer_addr]
    tail[entry]= (tail[entry] + 1) % ringSize
    Count[entry] += 1
    xfer_addr += 1
  ME[initial_xfer_addr] ← { 1,count[entry] } // 1 for success

Else
  ME[initial_xfer_addr] ← { 0,count[entry] } // 0 for failure
  
```

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 IXP2400 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. [Example 29](#) describes the details of the operations performed.

Example 29. Journaling Commands

```
Journal(entry, length, xfer_addr)
    For (temp=length, temp>0, temp--)
        // Write length number of longwords
        SRAM[tail[entry]]← ME [xfer_addr]
        tail[entry] = (tail[entry] + 1) % ringSize
        Count[entry] += 1
        xfer_addr += 1

fastJournal(entry)// either constant or ALU output
    SRAM[tail[entry]]← {ME#, ctx#, ME command bus<address field<23:0>}
    tail[entry]= (tail[entry] + 1) % ringSize
    Count[entry]++
```

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 1s. 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 ME Signal if the read was originated by an ME (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, and so on. 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 66](#).

Table 66. Address Map

| Start Address | End Address | Responder |
|---------------|-------------|-----------|
| 0x0000 0000 | 0x3fff ffff | Channel 0 |
| 0x4000 0000 | 0x7fff ffff | Channel 1 |
| 0x8000 0000 | 0xbfff ffff | reserved |
| 0xc000 0000 | 0xffff ffff | reserved |

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 1MB 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 an ME 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 via the CSR encoding in the command field.

Note: Reads and writes to undefined CSR addresses will yield unpredictable results.

The IXP2400 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 writeable.

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 67 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 67 and Table 68). Note that a blank entry means no order is enforced.

Table 67. Address Reference Order

| 1 st ref → 2 nd ref ↓ | 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 68. |

Table 68 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 67 apply to Table 68.

Table 68. 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 ^a | | | | | |
| Read_Q_Descr other | Order | | | | | | | |
| Write_Q_Descr ^b | | | | | | | | |
| Enqueue | Order | Order | | Order | Order ^c | | | |
| Dequeue | Order | Order | | Order ^c | Order | | | |
| Put | | | | | | Order | | |
| Get | | | | | | | Order | |
| Journal | | | | | | | | Order |

- a. 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.
- b. 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.
- c. 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 insure 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 ucode 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:

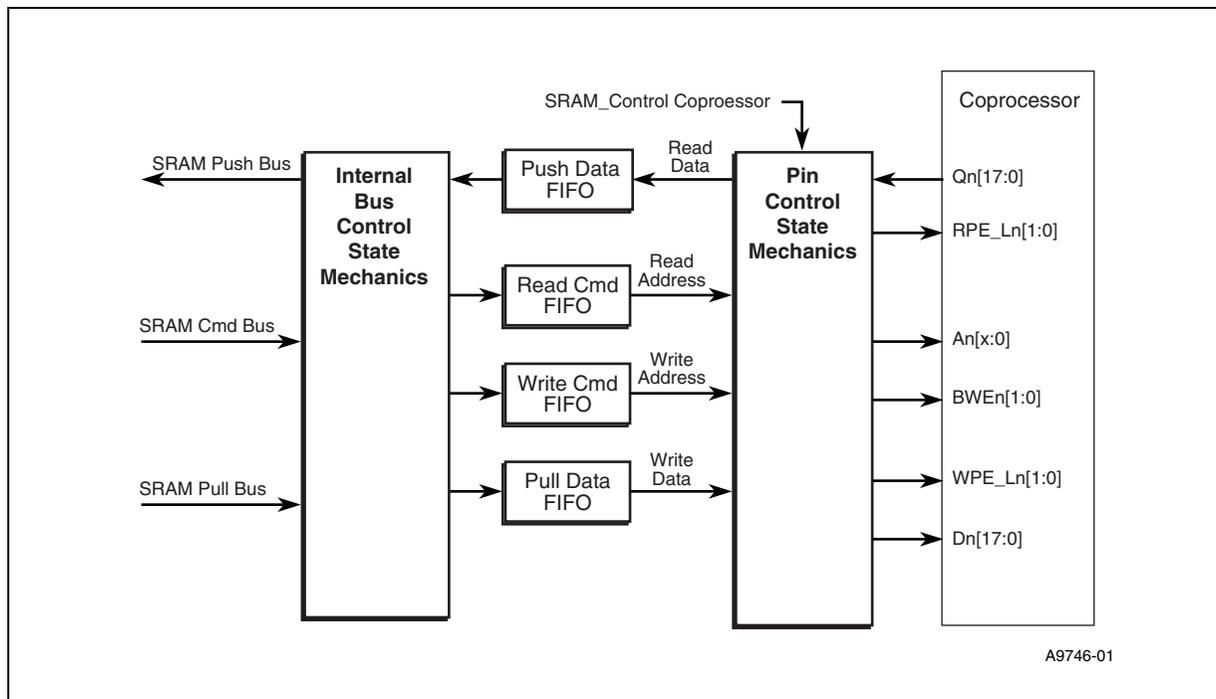
- All accesses to atomic variables should be through read-modify-write instructions.
- If the flow must know that a write is completed (actually in the SRAM itself), follow the write with a read to the same address. The write is guaranteed to be complete when the read data has been returned to the ME.
- 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 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 rd_qdesc_head followed by a rd_qdesc_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 78](#) is a simplified block diagram of the SRAM controller. [Figure 86](#) 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 86. Connection to a Coprocessor Though Standard QDR Interface



The external coprocessor interface is based on FIFO communication.

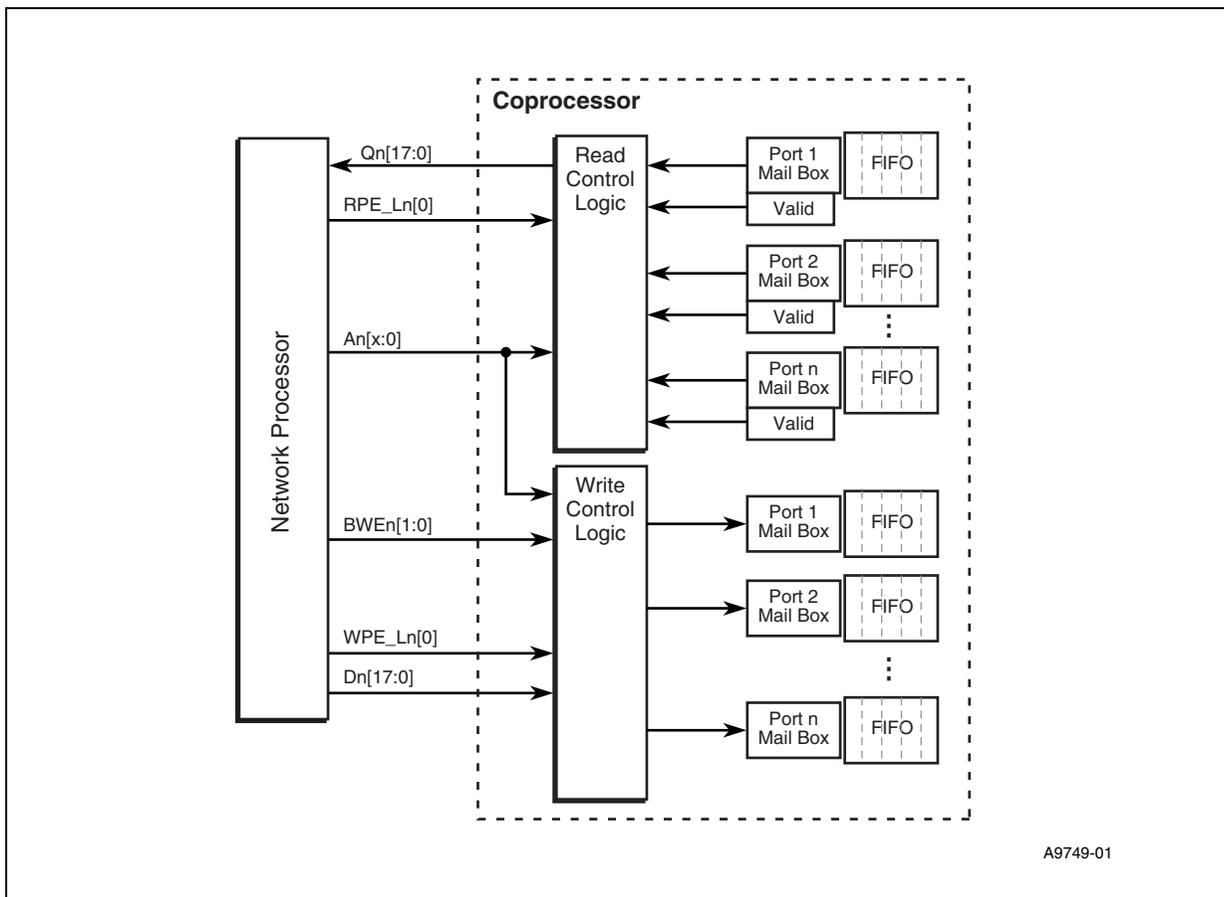
A thread can send parameters to the coprocessor by doing a normal SRAM write instruction:
`sram[write, $sram_xfer_reg, src1, src2, ref_count], optional_token`

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 87 shows the coprocessor with memory-mapped FIFO ports.

Figure 87. 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—ME 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 ME control. An ME 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 decremented by the thread.



This page intentionally left blank.

SHaC Unit

7

7.1 Prerequisite Reading

1. Section 2, “Hardware Overview”

7.2 Introduction

This chapter will cover the operation of the SHaC unit. The SHaC unit contains three main subblocks: the Scratchpad, the Hash units and the CAP (CSR Access Proxy). Each subblock will be described separately and in greater detail in the sections that follow.

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 “Scratchpad” on page 7-219.)

7.3 Unit 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.

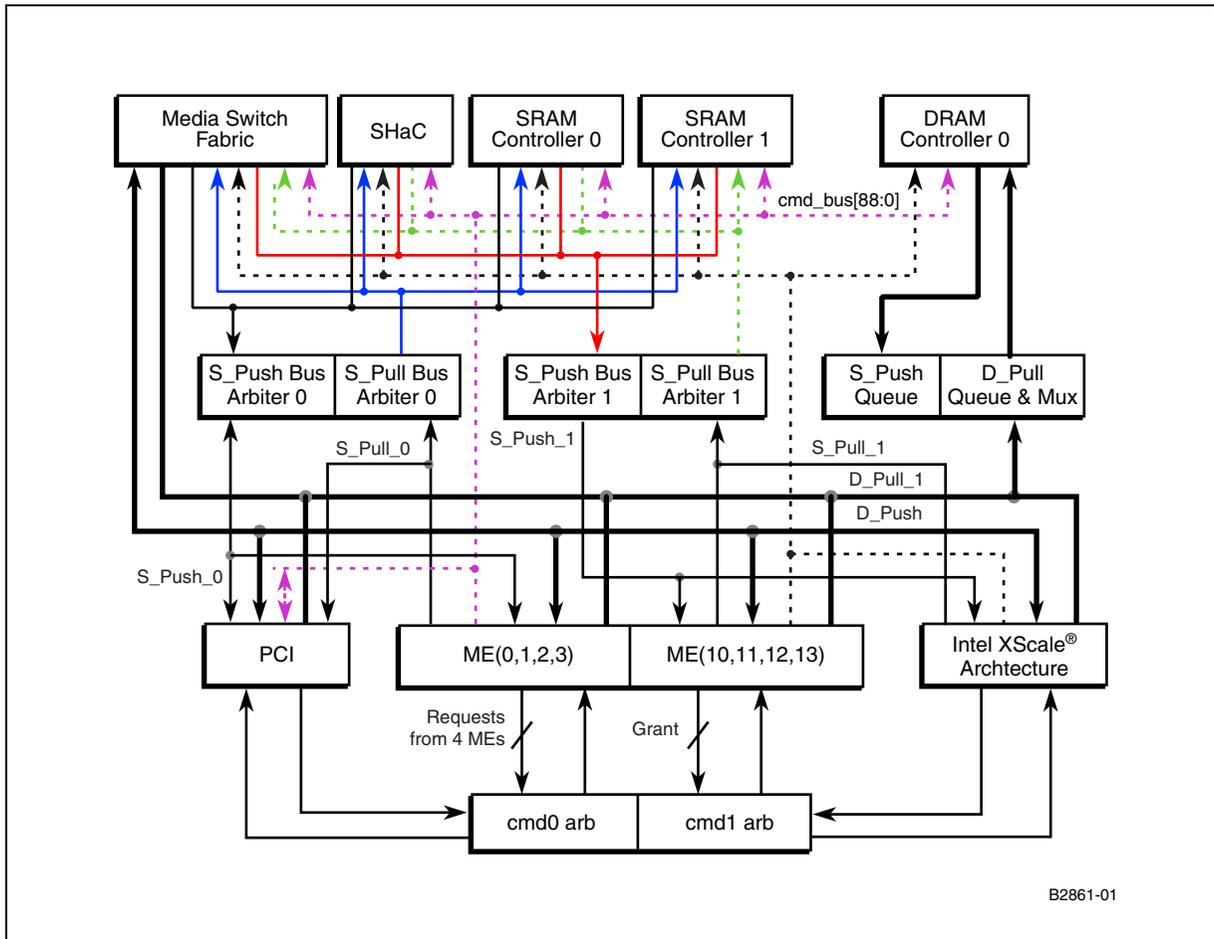
The SHaC unit provides the following features:

- 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
- A communication ring used by Microengines (MEs) for interprocess communication
- A Scratchpad memory storage option usable by Intel XScale® core and MEs
- A CSR bus interface to permit fast writes to CSRs, as well as standard read and writes
- A Push/Pull Reflector to transfer data from the Pull bus to the Push bus

7.4 High Level Block Diagrams

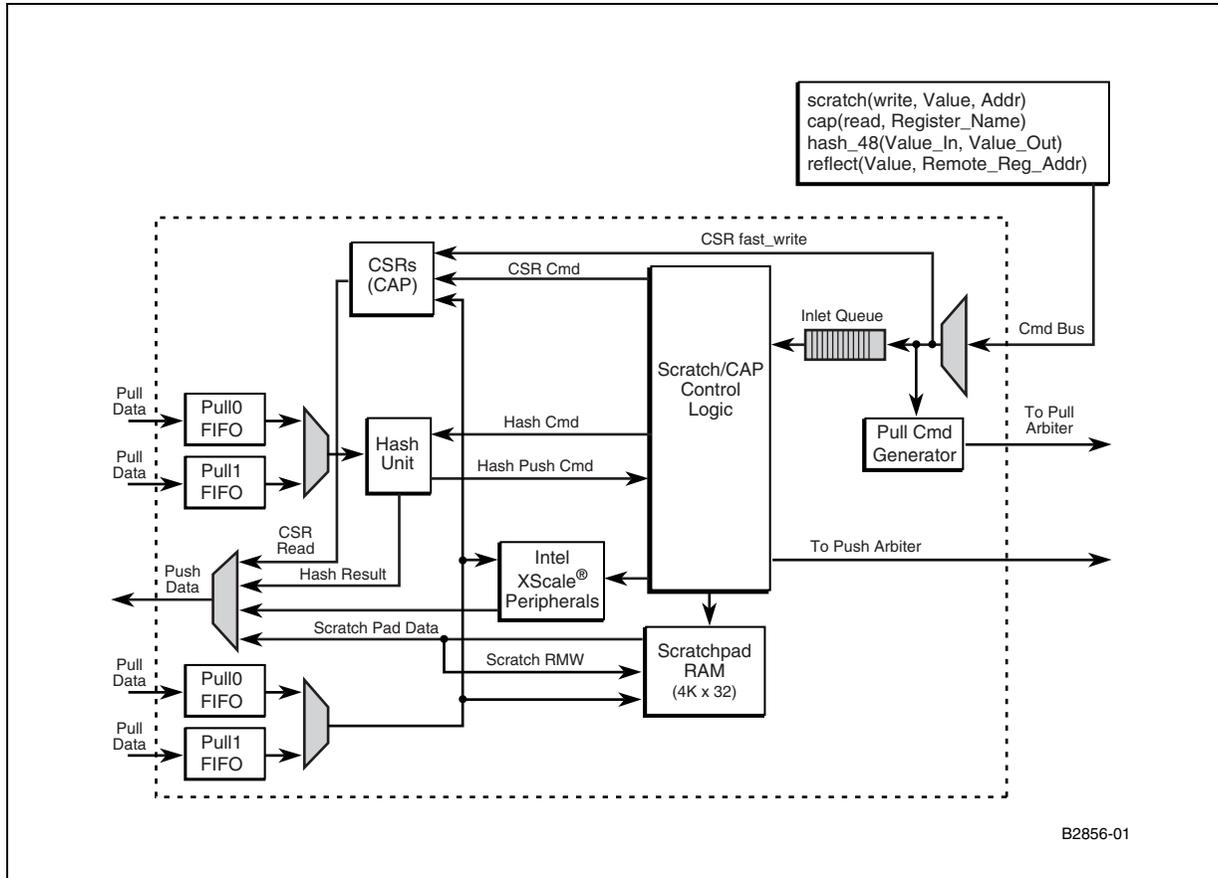
7.4.1 Full Chip Diagram

Figure 88. IXP2400 Chassis (APB and CSR Buses Not Shown) Block Diagram



7.4.2 SHaC Unit Block Diagram

Figure 89. SHaC Top Level Diagram



7.5 Unit Design Details

7.5.1 Scratchpad

7.5.1.1 Scratchpad Description

The SHaC Unit contains a 16KB Scratchpad memory, organized as 4 K 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.

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 an ME to read or write the transfer registers or CSRs in another ME.

Note: 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.

Collectively, operations to the CSRs and APB peripherals—as well as the Push/Pull Reflector Mode — form what is known as the CSR Access Proxy (CAP). The CAP is treated as a separate target to the MEs and the Intel XScale® core.

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.

7.5.1.2 Scratchpad Interface

The Scratchpad interfaces to the internal Command, S_Push, S_Pull, CSR, and APB buses.

The Scratchpad command and S_Push and S_Pull bus interfaces 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 the following sections.

7.5.1.2.1 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.

7.5.1.2.2 Push/Pull Interface

The Scratchpad has the capability to interface to either one or two 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 arbiter and Pull arbiter through which access to the Push bus and Pull bus, respectively, is regulated. Refer to [Section 6, “SRAM Interface”](#) for more information. When the Scratchpad is used in a chip that only utilizes one pair of PP buses, the other interface is unused.

1. A ring is a FIFO that uses a head and tail pointer to store/read information in Scratchpad memory.

7.5.1.2.3 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.

7.5.1.2.4 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 IXP2400 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.

7.5.1.3 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.

7.5.1.3.1 Scratchpad Commands

The basic read and write commands will transfer from 1 to 16 longwords of data to and 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 an ME, 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 an ME, 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 ME 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 ME, then the ME 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. (See the address map in the chapter.) 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.

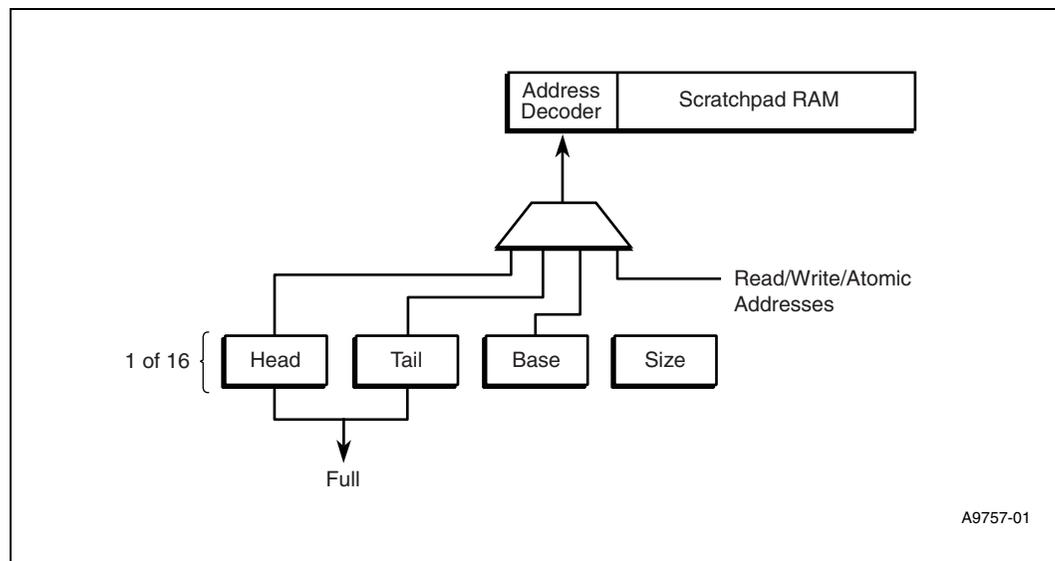
7.5.1.3.2 Ring Commands

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 90](#).

Figure 90. 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. 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 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. This no-overlap rule implies that many configurations are not legal. For example, programming 5 Rings to size of 1024 words would exceed the total size of Scratchpad memory, and therefore is not legal.

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

When the Ring is near full (see [Table 69](#) for the exact number of entries) it asserts an output signal which is used as a state input to the MEs. They must use that signal to test, by doing a Branch on Input Signal, 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 be guaranteed that a put

will no overflow 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 69. 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 69. Ring Full Signal Use — Number of Contexts and Length vs. Ring Size

| Number of Contexts | Ring Size ^a | | | |
|--------------------|------------------------|----------------------|-----|------|
| | 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 ^b | 2 | 5 | 12 |
| 48 | Illegal ^b | 2 | 4 | 10 |
| 64 | Illegal ^b | 1 | 3 | 7 |
| 128 | Illegal ^b | Illegal ^b | 1 | 3 |

- a. Number in each entry is the largest length that should be put. 16 is the largest length that a single put instruction can generate.
- b. Illegal with that number of Contexts, even a length of 1 could cause the Ring to overflow.

For IXP2400 B0, the Full Flag can be configured as an Empty Flag instead of Full Flag, by the RING_STATUS_FLAG bit in the SCRATCH_RING_BASE_# register. Note that each Ring has its own RING_STATUS_FLAG bit.

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 ME 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 ME thread issuing the command to check the SHTC_RING_FULL_RPH signal 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 70. Head/Tail, Base and Full by Ring Size

| Size (# of 32-bit words) | Base Address ^a | Head/Tail Offset | Full Threshold (number of empty 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 |

a. 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 through 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.

Table 71. Ring CSR Summary and Addresses

| Address | CSR name | Description |
|---------|--------------|--------------------------------|
| 0x0#00 | Ring[#] Base | Base address of the Ring |
| 0x0#00 | Ring[#] Head | Offset of head entry from Base |
| 0x0#00 | Ring[#] Tail | Offset of tail entry from Base |

(# = 0 - F)

7.5.1.3.3 CAP Commands

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 ME 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_H¹ signal is returned. 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 ME 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

1. The APB RDY signal is an extension to the APB bus specification specifically added for the IXP Chassis.

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 noncontiguous 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). 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.

Note: 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.

For reads using the Reflector mode, the write data is pulled from the source identified in ADDRESS (either an ME 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 an ME upon completion of a command. For APB and CAP CSR operations, CAP signals the ME 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 ME that is the target of the reflection, both, or neither.

7.5.1.3.4 XScale® Core and ME Instructions

Table 72 shows the Intel XScale® core and ME 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, an ME would execute a `csr[read]` instruction and Intel XScale® core would execute a `Load(1d)` instruction. Data is then moved between the CSR and the Intel XScale® core/ME by first reading the CSR via the APB bus and then writing the result to the Intel XScale® core/ME via the Push Bus.

Table 72. Intel® XScale® Core and ME Instructions

| Accessing | Read Operation | Write Operation |
|---|--|---|
| APB Peripheral | Access Method: ME: csr[read] Intel XScale® core: Id | Access Method: ME: 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: ME: csr[read] Intel XScale® core: Id | Access Method: ME: 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 |
| ME CSR or Xfer Register (Reflector Mode) | Access Method: ME: csr[read] Intel XScale® core: Id | Access Method: ME: 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) |

The following ME registers are normally used by MEs in network processing and are included in the CSR block. They are connected to the CAP CSR bus, implying that they can be written at the rate of one per cycle. They may also be accessed by other command bus masters (for example, XScale), typically for test and debug use.

Table 73. Inter-Process Communication Register Summary

| CSR Name | Address | Description |
|---|-------------|--|
| THD_MSG (Generic address) | 0x000 | Address for Microengine threads to write a message to their specific register. Refer to Note 1. |
| THD_MSG_CLR_#_\$_& # = ME cluster number 0 to 1 \$ = ME number in cluster. 0 to 7 for IXP2800. 0 to 3 for IXP2400 & = thread number 0 to 7 | 0x100–0x2FC | Address to read and clear each individual THD_MSG. Refer to Note 1. |
| THD_MSG_#_\$_& # = ME cluster number 0 to 1 \$ = ME number in cluster. 0 to 7 for IXP2800. 0 to 3 for IXP2400 & = thread number 0 to 7 | 0x500–0x6FC | Address to read each individual THD_MSG_#_\$_&. Refer to Note 1. For IXP2800, the offset for the 128 registers are $0x500 + (\text{cluster\#} * 64 + \text{ME\#} * 8 + \text{Thread\#}) * 4$ For IXP2400, the offset for the 64 registers are $0x500 + (\text{cluster\#} * 64 + \text{ME\#} * 8 + \text{Thread\#}) * 4$ |
| THD_MSG_SUMMARY_0_0 | 0x004 - | Bit vector registers that indicates which threads have new messages THD_MSG_SUMMARY_#_\$_ # = ME cluster number 0 to 1 \$ = register number 0 to 1 Refer to Note 1. |
| THD_MSG_SUMMARY_0_1 (IXP2800 only) | 0x008 | |
| THD_MSG_SUMMARY_1_0 | 0x00C | |
| THD_MSG_SUMMARY_1_1 (IXP2800 only) | 0x010 | |
| SELF_DESTRUCT_0 | 0x014 | Write bit number to set a bit in these registers and a read clears all the bits in the register |
| SELF_DESTRUCT_1 | 0x018 | |
| INTERTHREAD_SIG | 0x01C | Writing a thread and signal number to this register generates a signal event to the thread |

Table 73. Inter-Process Communication Register Summary (Continued)

| CSR Name | Address | Description |
|--|---------|--|
| XSCALE_INT_A | 0xb20 | Address for Microengine threads to set an interrupt to the XScale core |
| XSCALE_INT_B | 0xb24 | |
| <p>Note 1.</p> <p>Each Microengine thread can be programmed to write an 8-bit message to its own THD_MSG_#_\$_& register. The intent of these registers is to provide a mechanism to have the Microengine threads report their current processing status. The interpretation of the message is a software semantic between the sender and receiver.</p> <p>The numbering scheme of the Microengine threads involves the ME cluster, the ME number within the cluster and the thread number within each ME. There are two ME clusters for both IXP2800/2400. The IXP2800 offers eight MEs per cluster, with numbers 0 through 7. IXP2400 offers four MEs per cluster, with numbers 0 through 3.</p> <p>A Microengine thread writes this register using the fast_wr or csr[write] instruction with the generic THD_MSG register address. The data supplied with the instruction is written to the actual register associated with the Microengine thread number. CAP takes the generic address concatenates it with the ME and context number of the sender to create the specific address. The write will also set the bit corresponding to the sender in the THD_MSG_SUMMARY_#_\$_ Register.</p> <p>The csr[read] instruction or the Intel XScale® core processor read uses the actual THD_MSG register addresses to read these registers. There are two addresses to read THD_MSG. One will return the read data and clear the THD_MSG (and its corresponding THD_MSG_SUMMARY_#_\$_ bit), the other will return the read data and leave the contents of the register intact.</p> <p>The csr[read] instruction can use either the generic THD_MSG address or the actual thread specific THD_MSG_#_\$_& register addresses to read these registers. When the generic THD_MSG address is used for a read, CAP will determine the actual register in the same way as described above in the write description. There are two thread specific addresses for each THD_MSG; one will only read the data, the other will read the data and clear the THD_MSG register, and also clear the corresponding bit in the THD_MSG_SUMMARY_#_\$_ Register. Reading at the generic address does not do the clear function.</p> | | |

7.5.2 Hash Unit

7.5.2.1 Hash Unit Description

The SHaC unit 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 MEs and the Intel XScale® core.

7.5.2.1.1 Hashing Operation

Up to three hash indices can be created using a single ME instruction. The ME hash instructions are shown in [Example 32](#).

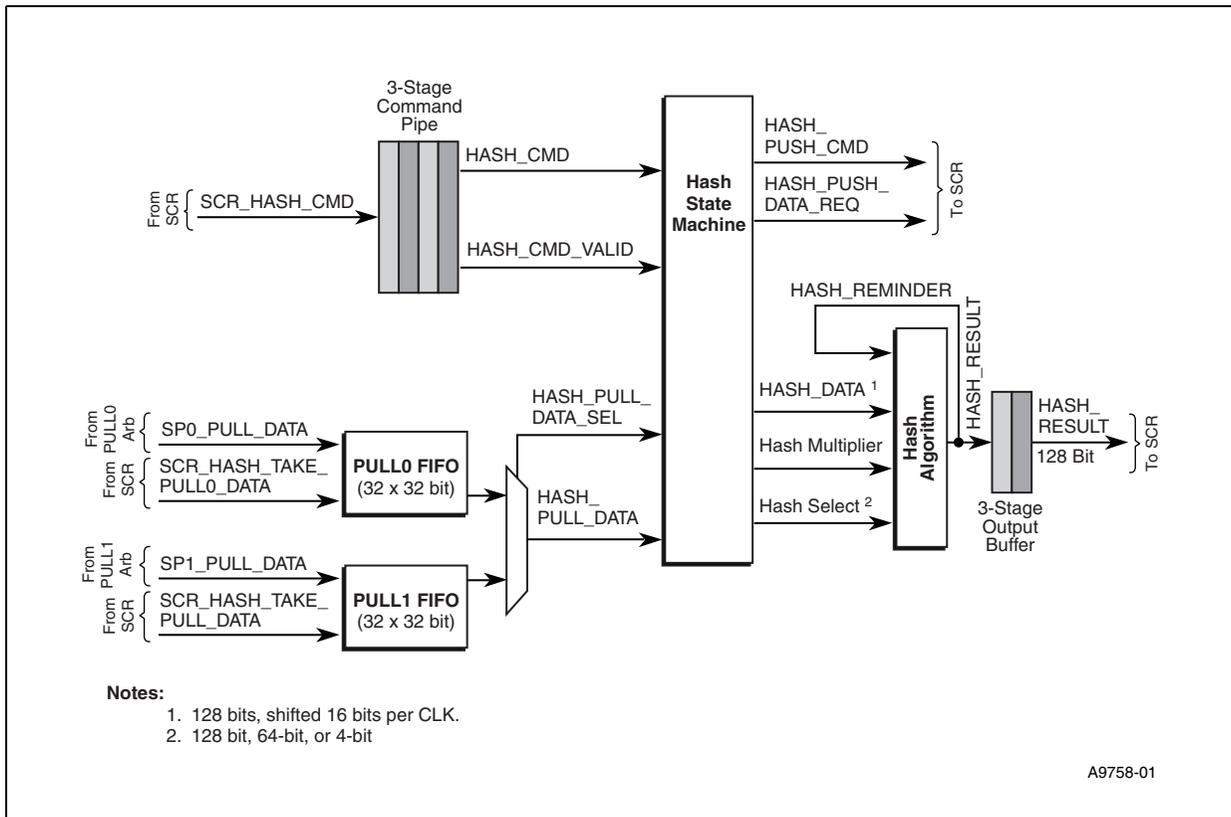
Example 32. ME Hash Instructions

| | |
|-----------------------------------|---|
| 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] |

7.5.2.2 Hash Unit Block Diagram

Figure 91 shows a block diagram of the Hash Unit. Refer to Figure 91 when reading the following subsections.

Figure 91. Hash Unit Block Diagram



7.5.2.3 Hash Operation

An ME 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 any of the SRAM Transfer Registers to be used. Two SRAM Transfer Registers are required to create hash indices for 48-bit and 64-bit and four SRAM Transfer Registers to create 128-bit hash indices, as shown in Table 74. In the case of the 48-bit hash, the Hash Unit ignores the upper two bytes of the second Transfer Register.

Table 74. 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 | | |

Table 74. S Transfer Registers Hash Operands (Continued)

| Register | Address |
|--------------------------------|-------------|
| 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 |
| 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, as shown in [Table 75](#). 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.

Table 75. Intel XScale® core Hash Operand Registers

| Register | Address |
|-------------------------------|--------------|
| 48-Bit Hash Operation | |
| Don't care | hash [47:32] |
| hash [31:0] | tbd |
| 64-Bit Hash Operation | |
| hash [63:32] | tbd |
| hash [31:0] | tbd |
| 128-Bit Hash Operation | |
| hash [127:96] | tbd |

Table 75. Intel XScale® core Hash Operand Registers (Continued)

| Register | Address |
|--------------|---------|
| hash [64:95] | tbd |
| hash [63:32] | tbd |
| hash [31:0] | tbd |

For both ME 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 indices. 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 is the sum of 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.

7.5.2.4 Hash Algorithm

The hashing algorithm used by IXP2400 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:

$$\text{Equation 1. } A_{48}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{46}x^{46} + a_{47}x^{47} \quad \text{(48-bit hash operation)}$$

$$\text{Equation 2. } A_{64}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{62}x^{62} + a_{63}x^{63} \quad \text{(64-bit hash operation)}$$

$$\text{Equation 3. } A_{128}(x) = a_0 + a_1x + a_2x^2 + \dots + a_{126}x^{126} + a_{127}x^{127} \quad \text{(128-bit hash operation)}$$

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.

$$\text{Equation 4. } M_{48}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{46}x^{46} + m_{47}x^{47} \quad \text{(48-bit hash operation)}$$

$$\text{Equation 5. } M_{64}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{62}x^{62} + m_{63}x^{63} \quad \text{(64-bit hash operation)}$$

$$\text{Equation 6. } M_{128}(x) = m_0 + m_1x + m_2x^2 + \dots + m_{126}x^{126} + m_{127}x^{127} \quad \text{(128-bit hash operation)}$$

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:

$$\text{Equation 7. } G_{48}(x) = 1 + x^{10} + x^{25} + x^{36} + x^{48} \quad \text{(48-bit hash operation)}$$

$$\text{Equation 8. } G_{64}(x) = 1 + x^{17} + x^{35} + x^{54} + x^{64} \quad \text{(64-bit hash operation)}$$

$$\text{Equation 9. } G_{128}(x) = 1 + x^{33} + x^{69} + x^{98} + x^{128} \quad \text{(128-bit hash operation)}$$

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:

$$\text{Equation 10. } A(x)M(x) = Q(x)G(x) + R(x)$$

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) (say 8 K or 16 K 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.

Table 76. Scratchpad Memory Register Summary

| CSR name | Address | Description |
|---------------------|---------|---------------------------------|
| SCRATCH_RING_BASE_# | 0x0#0 | Base address of the Ring. |
| SCRATCH_RING_HEAD_# | 0x0#4 | Offset of head entry from Base. |
| SCRATCH_RING_TAIL_# | 0x0#8 | Offset of tail entry from Base. |
| RESERVED | 0xFC | |

Table 77. Hash Multiplier Register Summary

| CSR name | Address | Description |
|-----------------------|---------|---|
| HASH_MULTIPLIER_48_0 | 0x00 | Least significant 32 bits of 48-bit Hash Multiplier. |
| HASH_MULTIPLIER_48_1 | 0x04 | Most significant 16 bits of 48-bit Hash Multiplier. |
| HASH_MULTIPLIER_64_0 | 0x08 | Least significant 32 bits of 64-bit Hash Multiplier. |
| HASH_MULTIPLIER_64_1 | 0x0C | Most significant 32 bits of 64-bit Hash Multiplier. |
| HASH_MULTIPLIER_128_0 | 0x10 | Least significant 32 bits of 128-bit Hash Multiplier. |
| HASH_MULTIPLIER_128_1 | 0x14 | Bits 32 to 63 of the 128-bit hash multiplier. |
| HASH_MULTIPLIER_128_2 | 0x18 | Bits 64 to 95 of the 128-bit hash multiplier. |
| HASH_MULTIPLIER_128_3 | 0x1C | Most significant 32 bits of 128-bit Hash Multiplier. |

Table 78. Global Chassis Registers

| Register Name | Address | Description |
|---------------------|---------|-------------|
| PRODUCT_ID | 0x00 | |
| MISC_CONTROL | 0x04 | |
| STRAP_OPTIONS | 0x18 | |
| RESET_0 | 0x0C | |
| RESET_1 | 0x10 | |
| CLOCK_CONTROL | 0x14 | |
| MCCR (IXP2400 Only) | 0x08 | |



This page intentionally left blank.

Media and Switch Fabric Interface 8

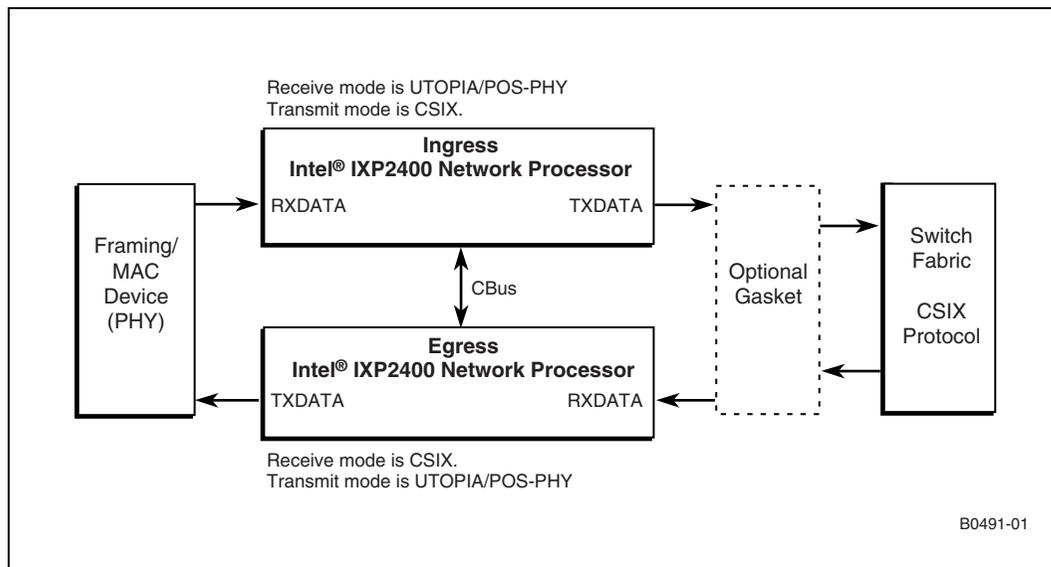
8.1 Overview

The Media and Switch Fabric (MSF) Interface is used to connect IXP2400 to a physical layer device (PHY) and/or to a switch fabric. The MSF has the following major features:

- Separate and independent 32-bit receive and transmit buses. Each bus may be configured independently.
- A configurable bus interface; the bus may function as a single 32-bit bus, or it can be channelized into independent buses: two 16-bit or four 8-bit buses, or one 16-bit bus and two 8-bit buses. Each channel may be configured to operate in either UTOPIA or POS-PHY modes.
- The Media bus operates from 25 to 133 MHz.
- UTOPIA Level 1/2/3 and POS-PHY Level 2/3 single-PHY (SPHY) master operation; 8-, 16-, or 32-bit buses are supported.
- UTOPIA Level 3 multi-PHY (MPHY) master operation with a 32-bit-wide bus; up to 32 slave ports are supported (16 ports in IXP2400 A0/A1); polling may be single RxClav/TxClav, or Direct Status Indication (maximum of four slave ports).
- POS-PHY Level 3 multi-PHY (MPHY) master operation with a 32-bit-wide bus with in-band addressing; up to 32 slave ports (16 ports in A0/A1 silicon) are supported, with packet-level polling.
- POS-PHY Level 2 and UTOPIA Level 2 master mode operation on one 16-bit-wide bus; up to 31 slave ports are supported; polled status mode is supported, and direct status indication is not supported. (This feature is only available in IXP2400 B0)
- POS-PHY Level 3 (SPI3) slave mode operation in SPHY mode on 32-bit bus, 2 x 16-bit bus, 4 x 8-bit bus, 1 x 16-bit + 2 x 8-bit bus. Note that this slave mode is not fully SPI3-specification-compliant and is intended primarily for daisy-chaining IXP2400s in certain applications. (This feature is only available in IXP2400 B0)
- Support for CSIX-L1 protocol with a 32-bit-wide bus. The only deviation from the CSIX-L1 specification is that the IXP2400 is clocked by a globally synchronous clock and is electrically 3.3V LVTTTL.
- Support for interprocessor CBus for communicating link level and fabric level flow control information between egress and ingress processors in CSIX mode.
- Interface to internal buses: command, SRAM push/pull, and DRAM push/pull.

Figure 92 shows one expected usage model.

Figure 92. An Expected Usage Model



Note: In this document, *UTOPIA* always refers to cell transport; *POS-PHY* refers to variable length packet transport; *CSIX* refers to CFrame transport.

8.2 Reference Documents

The reader should be familiar with the following specifications:

- UTOPIA Specification, Level 1, Version 2.01, March 21, 1994
- UTOPIA Level 2 Specification, Version 1.0, June 1995
- UTOPIA 3 Physical Layer Interface, November 1999
- POS-PHY Level 2 Specification, Issue 5, December 1998
- POS-PHY Level 3 Specification, Issue 4, June 2000
- SPI-3 Specification, June 2000
- Frame Based ATM Interface (Level 3), March 2000
- CSIX-L1: Common Switch Interface Specification -L1, Version 1.0, August 5, 2000

8.3 Media Bus Interface

The MSF consists of separate receive and transmit interfaces. Each of the receive and transmit interfaces can be separately configured for either UTOPIA (Level 1, 2, and 3), POS-PHY (Level 2 and 3) or CSIX protocols.

Note that any device that connects to the MSF interface must not exhibit any protocol violation with respect to the specifications of the protocols. Otherwise, the MSF hardware behavior is undefined. As an example, a device that operates in the POS-PHY Level 2 configuration must deassert RXVAL after the RXEOF of the frame has been observed, in order to comply with the POS-PHY Level 2 specification.

The receive and transmit ports are unidirectional and completely independent of each other. Each port has 32 data signals, two clocks, a set of control signals, and a set of parity signals, all of which use 3.3V LVTTTL signalling.

In UTOPIA and POS-PHY modes, each port can function as a single 32 bit interface, or can be subdivided into a combination of 16 bit or 8 bit channels. When running in channelized mode each channel operates independently. Each channel is a point-to-point connection to a single PHY. This is also known as single-PHY (SPHY) mode.

In addition to single-PHY mode, the IXP2400 also supports multi-PHY (MPHY) mode. In MPHY mode, the 32-bit bus is shared by up to 32 ports; per the UTOPIA Level 3 and POS-PHY Level 3 specifications, all ports must reside within one physical device. Also, one 16-bit bus is shared by up to 31 ports per the UTOPIA Level 2 and POS-PHY Level 2 protocol. On the 16-bit bus, the ports can be resident in up to four physical devices.

Note: Only master mode is supported in UTOPIA, POS_PHY Level 2 and POS-PHY Level 3 MPHY mode; POS-PHY Level 3 (SPI3) SPHY slave mode is supported; however, this is not fully compliant with the SPI3 specification.

Each interface has two clocks; RXCLK01/TXCLK01 is used by the ports associated with bits [15:0]; RXCLK23/TXCLK23 is used by the ports associated with bits [31:16]. This applies only to the 4x8, 2x16, and 1x16+2x8 SPHY modes, and allows each half of the bus to be clocked independently. In 1x32 SPHY, MPHY, or CSIX modes, only RXCLK01/TXCLK01 is used and is internally routed to all the logic; RXCLK23/TXCLK23 are tied to ground.

All signals are sampled only on the rising edge of the clock.

In addition to the UTOPIA, POS-PHY, and CSIX interfaces, there is also an interface called *CBus* which is used in CSIX mode to forward link level and fabric level flow control information from the egress (receive) processor to the ingress (transmit) processor.

The use of the pins is based on whether or not the port is in UTOPIA, POS-PHY, or CSIX mode. Tables in [Section 8.4](#) show how the external pin names map to the signal names referenced in the UTOPIA, POS-PHY, and CSIX specifications. The tables show all the possible signals that could be used for a particular standard. However, a particular mode within a standard, such as MPHY or SPHY, will not necessarily use all the signals shown in a column.

Note: The Media bus is 3.3V LVTTTL using globally synchronous (common) clocking. Thus the bus does not have electrical or clocking compatibility with the CSIX-L1 specification, which is 2.5V LVCMOS with source synchronous clocking.

8.3.1 UTOPIA

UTOPIA is a protocol for cell transfer between a physical layer (PHY) device and a link layer device (IXP2400). UTOPIA is optimized for the transfer of fixed sized ATM cells.

UTOPIA Levels 1, 2, and 3 are supported so that IXP2400 can talk to a wide variety of devices running at different speeds, as shown in [Table 79](#).

Table 79. UTOPIA Levels 1-, 2-, and 3-Supported Specifications

| Specification | Speed | Bus Width | Frequency |
|----------------|-------|-----------|-----------|
| UTOPIA Level 1 | OC-3 | 8 bits | 25 MHz |
| UTOPIA Level 2 | OC-12 | 16 bits | 50 MHz |
| UTOPIA Level 3 | OC-48 | 32 bits | 104 MHz |

IXP2400's implementation is more flexible in that all bus widths can be run from the frequency range of 25 to 133 MHz.

IXP2400 supports both single-PHY (SPHY) mode, described in [Section 8.3.1.1](#) and multi-PHY (MPHY) mode, described in [Section 8.3.1.2](#).

8.3.1.1 Single-PHY (SPHY) Mode

8.3.1.1.1 Bus Partitioning and Signal Grouping

In SPHY mode, the 32 bit interface may be subdivided into a combination of 8 or 16 bit channels (channelization); each channel has its own set of control signals (since there are only two clocks on each interface, adjacent 8 bit channels must share a common clock) and can operate independently of the other channels. Each channel is a point-to-point connection to a single PHY.

Note: The terms *port* and *channel* are used interchangeably throughout this document.

[Table 80](#) shows the supported bus modes.

Table 80. Supported Bus Modes

| Bus Partitioning | Port Number |
|------------------|-------------|
| 1x32 | 0 |
| | N/A |
| | N/A |
| | N/A |
| 2x16 | 0 |
| | N/A |
| | 2 |
| | N/A |
| 4x8 | 0 |
| | 1 |
| | 2 |
| | 3 |
| 1x16_2x8 | 0 |
| | N/A |
| | 2 |
| | 3 |

The bus partitioning is controlled by the MSF_Rx_Control[Rx_Width] and MSF_Tx_Control[Tx_Width] bits.

Each channel may be configured to operate in one of three modes: CSIX, UTOPIA, or POS-PHY. This is controlled using the MSF_Rx_Control[Rx_Mode], Rx_UP_Control_{0...3}[CP_Mode], MSF_Tx_Control[Tx_Mode], and Tx_UP_Control_{0...3}[CP_Mode] bits.

For example, if IXP2400 is configured for 4x8 mode, channels 0 and 3 can be configured for POS-PHY mode, and channels 1 and 2 can be configured for UTOPIA mode. (CSIX only runs in 1x32 mode.)

Table 81 shows which control and data signals are associated with a given port in SPHY UTOPIA mode.

Table 81. Signal Usage in SPHY UTOPIA Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-------------|--|
| 1x32 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXFA[0], RXDATA[31:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXFA[0], TXDATA[31:0] |
| | 1 | N/A |
| | 2 | N/A |
| | 3 | N/A |
| 2x16 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXFA[0], RXDATA[15:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXFA[0], TXDATA[15:0] |
| | 1 | N/A |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXPRTY[2], RXFA[2], RXDATA[31:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXPRTY[2], TXFA[2], TXDATA[31:16] |
| | 3 | N/A |
| 4x8 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXFA[0], RXDATA[7:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXFA[0], TXDATA[7:0] |
| | 1 | RX: RXCLK01, RXENB[1], RXSOF[1], RXPRTY[1], RXFA[1], RXDATA[15:8] TX: TXCLK01, TXENB[1], TXSOF[1], TXPRTY[1], TXFA[1], TXDATA[15:8] |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXPRTY[2], RXFA[2], RXDATA[23:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXPRTY[2], TXFA[2], TXDATA[23:16] |
| | 3 | RX: RXCLK23, RXENB[3], RXSOF[3], RXPRTY[3], RXFA[3], RXDATA[31:24] TX: TXCLK23, TXENB[3], TXSOF[3], TXPRTY[3], TXFA[3], TXDATA[31:24] |
| 1x16_2x8 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXFA[0], RXDATA[15:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXFA[0], TXDATA[15:0] |
| | 1 | N/A |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXPRTY[2], RXFA[2], RXDATA[23:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXPRTY[2], TXFA[2], TXDATA[23:16] |
| | 3 | RX: RXCLK23, RXENB[3], RXSOF[3], RXPRTY[3], RXFA[3], RXDATA[31:24] TX: TXCLK23, TXENB[3], TXSOF[3], TXPRTY[3], TXFA[3], TXDATA[31:24] |

8.3.1.1.2 Mode Selection

In order for a channel to operate in UTOPIA mode, the Rx_UP_Control_{0...3}[CP_Mode] or Tx_UP_Control_{0...3}[CP_Mode] bit must be 0.

In SPHY mode, each channel may be configured independently for either UTOPIA or POS-PHY operation.

8.3.1.1.3 Cell Size

The IXP2400 supports the following cell sizes, based on the port's bus width, as shown in Table 82. This is controlled by the Rx_UP_Control_{0...3}[Cell_Size]/Tx_UP_Control_{0...3}[Cell_Size] bits. The difference is that when Cell_Size = 0 it is expected that the PHY strips out the HEC/UDF byte, but when Cell_Size = 1, the HEC/UDF byte is left in the cell and replicated so that the cell size becomes an integral number of transfers on the bus.

Table 82. Supported Cell Sizes

| Bus Width | Cell_Size = 0 | Cell_Size = 1 |
|-----------|---------------|---------------|
| 8 | 52 bytes | 53 bytes |
| 16 | 52 bytes | 54 bytes |
| 32 | 52 bytes | 56 bytes |

Cell size is configurable on a per-port basis in SPHY mode.

Note: Cell_Size = 1 may cause the cell payload to fall on a non-longword (4 byte) boundary, making processing more difficult.

8.3.1.1.4 Decode Response Time

The decode response time is the number of clocks which are allowed to elapse between RXENB and receive control and data (RXDATA, RXSOF, and RXPRTY).

The UTOPIA Level 1 and Level 2 specifications specify one clock cycle; the UTOPIA Level 3 specification specifies two clock cycles. The Rx_UP_Control_{0...3}[DR_Time]/Tx_UP_Control_{0...3}[DR_Time] bit is used to tell the logic what the decode response time is.

Decode response time is configurable on a per-port basis in SPHY mode.

8.3.1.1.5 UTOPIA Level 3 Compatibility Mode

In IXP2400 A0/A1 chips, in UTOPIA SPHY mode, MSF keeps RXENB asserted as long as it has room in its receive FIFO. MSF essentially ignores RXFA. It monitors RXSOF for incoming cells. RXENB is deasserted only if there is no more room in the receive FIFO to hold any more cells. This is called “aggressive RXENB” and it is specifically allowed in the UTOPIA Level 2 specification, and is useful to avoid bubbles between cells.

The UTOPIA Level 3 is less clear on whether this mode of operation is allowed. In order to accommodate slaves which cannot handle “aggressive RXENB” a mode bit has been added to provide a “conservative RXENB”.

This mode only works for devices which provide the UTOPIA Level 3 “early” RxClav (RXFA) timing; it will not work for UTOPIA Level 1/2 devices in which the RxClav signal for the next cell isn't valid until after the end of the current cell.

In “conservative RXENB” mode, MSF will monitor RXFA and will not assert RXENB until RXFA is asserted. When the end of the cell is reached, the FSM will check RXFA again. If it is deasserted, it means no more cells are available and the FSM will deassert RXENB. If it is asserted, then the FSM will keep RXENB asserted.

“Conservative RXENB” works for x8, x16, and x32 SPHY modes.

8.3.1.1.6 Parity

UTOPIA uses single bit odd parity, independent of the bus width (x8, x16, or x32). Single bit odd parity mode is chosen using the Rx_UP_Control_{0..3}[Parity_Mode]/Tx_UP_Control_{0..3}[Parity_Mode] bit.

Parity mode is configurable on a per-port basis in SPHY mode.

8.3.1.1.7 Handshaking

Only cell level handshaking is supported; octet level handshaking is not supported.

8.3.1.2 Multi-PHY (MPHY) Mode

In MPHY mode, the UTOPIA bus is shared between multiple ports: UTOPIA Level 3 MPHY on 32-bit bus with up to 32 ports, and UTOPIA Level 2 operation on one 16-bit bus with up to 31 ports are supported. Both the above modes are referred to as MPHY-32 (the limitation of 31 ports on the 16-bit bus is due to the protocol). The 32-bit-wide bus must be a point-to-point connection between IXP2400 and the PHY. This implies that all the ports must be implemented inside one physical device. The 16-bit-wide bus can support up to four loads, i.e., the total ports can be split across up to four physical devices. An 8-bit-wide bus is not supported in MPHY mode.

In MPHY mode, all ports must have the same characteristics; for example, it is not possible to have some ports operate in cell mode and others in packet mode, nor is it possible to have some channels use odd parity and others use even parity. The port characteristics are chosen using the Rx_UP_Control_0 and Tx_UP_Control_0 registers in MPHY-32 mode; in MPHY-4 mode, Rx_UP_Control_{0..3} and Tx_UP_Control_{0..3} are used to select the operating mode, and must all be programmed to identical values.

8.3.1.2.1 Bus Partitioning and Signal Grouping

Table 83 shows signal usage when running in MPHY mode.

Table 83. Signal Usage in MPHY Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-----------------------------------|--|
| 1x32 | 0–3 with direct status indication | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXFA[3:0], RXADDR[4:0], RXDATA[31:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXFA[3:0], TXADDR[4:0], TXDATA[31:0] |
| | 0–31 with single RxClav/TxClav | RX: RXCLK01, RXENB[0], RXSOF[0], RXPRTY[0], RXPFA, RXADDR[4:0], RXDATA[31:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXPRTY[0], TXPFA, TXADDR[4:0], TXDATA[31:0] |

8.3.1.2.2 Mode Selection

UTOPIA MPHY mode is selected using MSF_Rx_Control[Rx_MPHY_Mode] and MSF_Tx_Control[Tx_MPHY_Mode]. MSF_Rx_Control[Rx_MPHY_Level2] and MSF_Tx_Control[Tx_MPHY_Level2] are used to select between UTOPIA Level 2 and Level 3 operation.

8.3.1.2.3 Cell Size

Cell size support is the same as in SPHY mode. It is selected using Rx_UP_Control_0[Cell_Size]/Tx_UP_Control_0[Cell_Size].

8.3.1.2.4 Decode Response Time

Decode response time is required to be two clock cycles, per the UTOPIA Level 3 specification. This is the time between the following event pairs:

- RXADDR[4:0] -> RXPFA
- RXENB -> RXSOF, RXDATA, RXPRTY

Rx_UP_Control_0[DR_Time] and Tx_UP_Control_0[DR_Time] must be programmed for two clock cycle decode response time.

The POS-PHY Level 2 specification specifies one clock cycle. However, the IXP2400 supports both 1- and 2-clock-cycle decode response times in POS-PHY Level 2 mode. The 2-clock-cycle option allows the bus to be overclocked beyond 50 MHz. Thus, the Rx_UP_Control_0[DR_Time] or Tx_UP_Control_0[DR_Time] bit may be programmed for 1- or 2-clock-cycle decode response time.

8.3.1.2.5 Parity

Parity is the same as SPHY mode. Parity mode is selected using Rx_UP_Control_0[Parity_Mode] or Tx_UP_Control_0[Parity_Mode].

8.3.1.2.6 Handshaking

There are two types of handshaking supported in MPHY mode. This is selected using MSF_Rx_Control[Rx_MPHY_Poll_Mode] and MSF_Tx_Control[Tx_MPHY_Poll_Mode].

The first type of handshaking is direct status. Each port has its own status signal, and no polling is required. The PHY cannot have more than four ports, and each port has its own status signal: RXFA[x] and TXFA[x].

The second is polled status. The PHY may have up to 32 ports, and provides only one shared status signal for all 32 ports: RXPFA and TXPFA. Status is obtained by polling using RxAddr[4:0] and TxAddr[4:0].

8.3.2 POS-PHY

IXP2400 supports POS-PHY (Packet Over SONET) mode. POS-PHY supports variable length packets rather than fixed size cells.

POS-PHY Levels 2 and 3 are supported so that IXP2400 can talk to a wide variety of devices running at different speeds, as shown in [Table 84](#).

Table 84. POS-PHY Levels 2 and 3-Supported Specifications

| Specification | Speed | Bus Width | Frequency |
|-----------------|-------|-----------|-----------|
| POS-PHY Level 2 | OC-12 | 16 bits | 50 MHz |
| POS-PHY Level 3 | OC-12 | 8 bits | 104 MHz |
| POS-PHY Level 3 | OC-48 | 32 bits | 104 MHz |

IXP2400’s implementation is more flexible in that all bus widths can be run from the frequency range of 25 to 133 MHz.

Note: POS-PHY Level 3 has also been standardized through the Optical Internetworking Forum and is called SPI-3, and through the ATM Forum, where it is called “Frame Based ATM Interface (Level 3)”.

Bus mode is programmable on a per-port basis. Bus mode is selected using the MSF_Rx_Control[Receive_Mode] and UP_Receive_Control{0..3}[CP_Mode] or MSF_Tx_Control[Transmit_Mode] and UP_Transmit_Control{0..3}[CP_Mode] bits.

The IXP2400 supports both single-PHY (SPHY) and multi-PHY (MPHY) modes. They are described separately in [Section 8.3.2.1](#) and [Section 8.3.2.2](#).

8.3.2.1 Single PHY (SPHY) Mode

8.3.2.1.1 Bus Partitioning and Signal Grouping

The bus partitioning is the same as for UTOPIA mode, but with additional signals and protocol added to support variable length packet transfer.

IXP2400 will support the POS-PHY protocol for 8-, 16-, 32-bit modes, with operating frequencies ranging from 25 to 133 MHz.

[Table 85](#) shows which control and data signals are associated with a given port in POS-PHY mode.

Table 85. Signal Usage in POS-PHY Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-------------|--|
| 1x32 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXFA[0], RXPADL[1:0], RXDATA[31:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXFA[0], TXPADL[1:0], TXDATA[31:0] |
| | 1 | N/A |
| | 2 | N/A |
| | 3 | N/A |

Table 85. Signal Usage in POS-PHY Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-------------|--|
| 2x16 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXFA[0], RXPADL[0], RXDATA[15:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXFA[0], TXPADL[0], TXDATA[15:0] |
| | 1 | N/A |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXEOF[2], RXVAL[2], RXERR[2], RXPRTY[2], RXFA[2], RXPADL[1], RXDATA[31:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXEOF[2], TXERR[2], TXPRTY[2], TXFA[2], TXPADL[1], TXDATA[31:16] |
| | 3 | N/A |
| 4x8 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXFA[0], RXDATA[7:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXFA[0], TXDATA[7:0] |
| | 1 | RX: RXCLK01, RXENB[1], RXSOF[1], RXEOF[1], RXVAL[1], RXERR[1], RXPRTY[1], RXFA[1], RXDATA[15:8] TX: TXCLK01, TXENB[1], TXSOF[1], TXEOF[1], TXERR[1], TXPRTY[1], TXFA[1], TXDATA[15:8] |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXEOF[2], RXVAL[2], RXERR[2], RXPRTY[2], RXFA[2], RXDATA[23:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXEOF[2], TXERR[2], TXPRTY[2], TXFA[2], TXDATA[23:16] |
| | 3 | RX: RXCLK23, RXENB[3], RXSOF[3], RXEOF[3], RXVAL[3], RXERR[3], RXPRTY[3], RXFA[3], RXDATA[31:24] TX: TXCLK23, TXENB[3], TXSOF[3], TXEOF[3], TXERR[3], TXPRTY[3], TXFA[3], TXDATA[31:24] |
| 1x16_2x8 | 0 | RX: RXCLK01, RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXFA[0], RXPADL[0], RXDATA[15:0] TX: TXCLK01, TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXFA[0], TXDATA[15:0] |
| | 1 | N/A |
| | 2 | RX: RXCLK23, RXENB[2], RXSOF[2], RXEOF[2], RXVAL[2], RXERR[2], RXPRTY[2], RXFA[2], RXDATA[23:16] TX: TXCLK23, TXENB[2], TXSOF[2], TXEOF[2], TXERR[2], TXPRTY[2], TXFA[2], TXDATA[23:16] |
| | 3 | RX: RXCLK23, RXENB[3], RXSOF[3], RXEOF[3], RXVAL[3], RXERR[3], RXPRTY[3], RXFA[3], RXDATA[31:24] TX: TXCLK23, TXENB[3], TXSOF[3], TXEOF[3], TXERR[3], TXPRTY[3], TXFA[3], TXDATA[31:24] |

8.3.2.1.2 Mode Selection

In order for a channel to operate in POS-PHY mode, the Rx_UP_Control_{0...3}[CP_Mode] or Tx_UP_Control_{0...3}[CP_Mode] bit must be 1.

8.3.2.1.3 Decode Response Time

The decode response time is the number of clocks which are allowed to elapse between RXENB and receive control and data (RXDATA, RXSOF, RXEOF, RXVAL, and RXPRTY).

The POS-PHY Level 2 specification specifies one clock cycle. The POS-PHY Level 3 specification specifies two clock cycles. The Rx_UP_Control_{0...3}[DR_Time] or Tx_UP_Control_{0...3}[DR_Time] bit is used to tell the logic what the decode response time is.

Decode response time is configurable on a per-port basis in SPHY mode.

8.3.2.1.4 Parity

POS-PHY allows both single bit even or odd parity, independent of the bus width (x8, x16, or x32). The parity mode is chosen using the Rx_UP_Control_{0...3}[Parity_Mode] or Tx_UP_Control_{0...3}[Parity_Mode] bit.

Parity mode is configurable on a per-port basis.

8.3.2.2 Multi-PHY (MPHY) Mode

Multi-PHY, or MPHY mode, is supported for POS-PHY mode. Both POS-PHY Level 2 and Level 3 MPHY modes are supported.

8.3.2.2.1 POS-PHY Level 3 Mode

One of the major functional changes in POS-PHY Level 3 is the addition of in-band addressing. In-band addressing allows up to 256 ports to be addressed. Addresses are transferred not using sideband address signals but using by multiplexing addresses on the data path (hence the name “in-band”). The IXP2400 supports a maximum of 32 ports.

On receive, the PHY or framer device will be programmed to deliver data in 64, 128, or 256 byte bursts. The PHY will provide an eight bit address on the data bus and assert RSX, followed by a burst of data. In this model, the PHY is responsible for picking which port goes next; IXP2400 cannot select the port to transfer receive data from.

On transmit, IXP2400 will also be limited to support 32 ports on the 32-bit bus.

In MPHY-4 mode, both direct status and polled modes will be supported. In direct status mode, status for each of the four ports carried on the TXFA[3:0] signals. In polled mode, TXADDR[4:0] is used to poll the TXPFA signal to determine which FIFOs in the PHY have room to hold transmit data.

In MPHY-32 mode, only polled mode will be supported. IXP2400 will use TXADDR[4:0] to poll the TXPFA signal to determine which FIFOs in the PHY have room to hold transmit data.

In-band addressing is supported only for 1x32 operation. The POS-PHY Level 3 spec also defines in-band addressing for 8 bit, but x8 (and x16) MPHY modes are not supported in IXP2400.

8.3.2.2.2 POS-PHY Level 2 Mode

POS-PHY Level 2 mode is supported only for a 16-bit bus. Only MPHY-32 mode is supported; in this mode a maximum of 31 ports is allowed. Per the specification, only out-of-band addressing is supported.

On receive, arbitration is performed by MSF rather than the PHY; this is the major functional difference between the Level 2 and Level 3 protocols.

8.3.2.2.3 Bus Partitioning and Signal Grouping

Table 86 shows which control and data signals are associated with a given port in POS-PHY mode.

Table 86. Signal Usage in POS-PHY Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-------------------|--|
| 1x32 | 0–3 byte level | RX: RXCLK01, RSX, RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXFA[3:0], RXPADL[1:0], RXDATA[31:0] TX: TXCLK01, TSX, TXADDR[4:0], TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXFA[3:0], TXPADL[1:0], TXDATA[31:0] |
| | 0–31 packet level | RX: RXCLK01, RSX, RXADDR[4:0], RXENB[0], RXSOF[0], RXEOF[0], RXVAL[0], RXERR[0], RXPRTY[0], RXPADL[1:0], RXDATA[31:0] TX: TXCLK01, TSX, TXADDR[4:0], TXPFA, TXSFA, TXENB[0], TXSOF[0], TXEOF[0], TXERR[0], TXPRTY[0], TXPADL[1:0], TXDATA[31:0] |

8.3.2.2.4 Mode Selection

POS-PHY MPHY mode is selected using MSF_Rx_Control[Rx_MPHY_En] or MSF_Tx_Control[Tx_MPHY_En] along with the Rx_UP_Control_0[CP_Mode] or Tx_UP_Control_0[CP_Mode].

8.3.2.2.5 Decode Response Time

The decode response time is the number of clocks which are allowed to elapse between the following event pairs:

- RXENB and receive control and data (RXDATA, RXSOF, RXEOF, RXVAL, and RXPRTY)
- TXADDR[4:0] and TXPFA

The POS-PHY Level 3 specification specifies two clock cycles. The Rx_UP_Control_0[DR_Time] or Tx_UP_Control_0[DR_Time] bit must be programmed for two clock cycle decode response time.

The POS-PHY Level 2 specification specifies one clock cycle. However, IXP2400 supports both 1- and 2-clock cycle decode response times in POS-PHY Level 2 mode. The 2-clock cycle option allows the bus to be overclocked beyond 50 MHz. Thus, the Rx_UP_Control_0[DR_Time] or Tx_UP_Control_0[DR_Time] bit may be programmed for 1- or 2-clock cycle decode response time.

8.3.2.2.6 Parity

Same as POS-PHY SPHY mode. It is selected using Rx_UP_Control_0[Parity_Mode] or Tx_UP_Control_0[Parity_Mode]

8.3.2.3 SPI3 Slave Mode

In SPI3 (POS-PHY Level 3) SPHY mode, any given port may be configured to act as a slave as well. The tables below show how pins are used in slave mode for each channel in all the SPHY modes:

1. 1x32
2. 2x16

- 3. 4x8
- 4. 1x16_2x8

It is possible to mix and match master and slave modes; that is, when configured in 2x16, 4x8, or 1x16+2x8 modes, some channels may act as master and other channels may act as slaves.

Note: The slave mode implementation is not fully SPI3-compliant. In particular, on the receive slave interface, when the master deasserts RXENB, MSF will send up to four more clock cycles of data before stopping. The SPI3 specification only allows a maximum of two more clock cycles of data. This means the receive slave interface will not work with any receive master which cannot deal with this situation. However, the MSF's receive master interface can handle this (as long as the receive FIFO high watermarks are correctly configured), so that loopback or daisy chaining of multiple IXP2400s are possible. In addition, the MSF will not hold the previous value on the outputs as required by the SPI3 specification when the master deasserts RXENB.

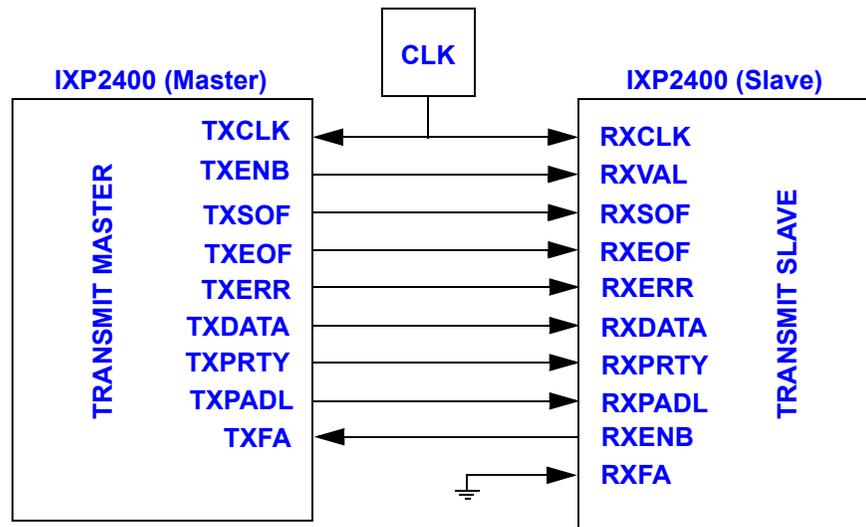
Note: SPI3 is not defined for a 16-bit bus; however, the non-compliant slave mode can be used even for a 16-bit bus.

Note: In MPHY mode, only master mode is supported.

The pin names assume master mode operation. If a port is configured for slave mode operation, then the functions of the pin changes. A receive master port becomes a transmit slave port; likewise, a transmit master port becomes a receive slave port.

Figure 93 and Figure 94 show generic examples of master/slave connections.

Figure 93. IXP2400 Tx Master to Tx Slave Connection

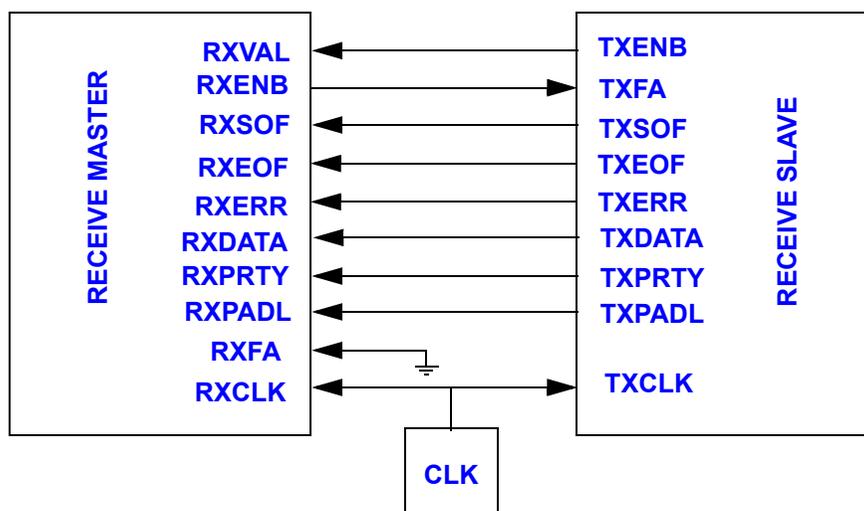


Note: One major implication of the spec non-compliance described in the previous note is that when daisy chaining IXP2400s, the sending IXP2400 must be configured as a Tx master and the receiving IXP2400 must be configured as a Tx slave. Configuring it the other way, i.e., Rx master

to Rx slave, will not work. That is because the Rx master expects no more than two more additional clock cycles of data after it deasserts RXENB. However, a IXP2400 Rx slave can send up to four more clock cycles of data.

Note: While it is possible to configure IXP2400 as an Rx slave, the Rx master must be able to handle the non-standard RXENB deassertion to RXDATA latency.

Figure 94. IXP2400 Rx Master to Rx Slave Connection (WILL NOT WORK!)



8.3.2.4 Transmit Slave Operation

A master receive port becomes a slave transmit port. All signals connect one-to-one (the master's TXSOF pin connects to the slave's RXSOF pin, etc.) with the following three exceptions:

1. The master's TXENB pin is connected to the slave's RXVAL pin. The slave transmit port will invert the TXENB signal internally so that internally it behaves the same as the RXVAL signal.
2. The slave's RXENB pin is connected to the master's TXFA pin. The slave transmit port will invert the RXENB signal internally so that internally it behaves the same as the TXFA signal. RXENB assertion/deassertion is controlled by the high watermarks in Rx_FIFO_Control CSR.
3. The RXFA pin on the transmit slave should be tied low.

8.3.2.5 Receive Slave Operation

A master transmit port becomes a slave receive port. All signals connect one-to-one (the master's RXSOF pin connects to the slave's TXSOF pin, etc.) with the following exceptions:

1. The master's RXVAL pin is connected to the slave's TXENB pin. The slave receive port will invert the RXVAL signal internally so that internally it behaves the same as the RXVAL signal.

8.3.3 CSIX

CSIX (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 CSIX specification is controlled by CSIX, an international consortium organized to create and promote a Common Switch Interface—www.csix.org.

CSIX mode is selected using MSF_Rx_Control[Receive_Mode] and MSF_Tx_Control[Transmit_Mode].

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. CFrames Assignment

| 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 IXP2400, CFrames are constructed for transmit under ME software control, and written into the transmit buffer. Vertical and horizontal parity generation is done by hardware. Also, hardware will automatically handle transmission of idle CFrames with link level RDY bits constantly updated when there are no data or control CFrames to transmit.

On receive to IXP2400, Idle CFrames are recognized by hardware and discarded; Flow Control CFrames, as well as link level flow control information (DRDY and CRDY bits) are handled by hardware (using CBus); all other types are buffered and passed to a ME to be parsed by software. However, Link Level Flow Control information in the Base Header of all CFrames (including Idle), is handled by hardware.

CSIX mode will only work for 1x32 bus mode. CSIX mode is illegal for 4x8 or 2x16 or 1x16_2x8 bus modes. [Table 88](#) shows which control and data signals are associated with a given port in CSIX mode.

Table 88. Signal Usage in CSIX Mode

| Bus Partitioning | Port Number | Signal Groupings |
|------------------|-------------|--|
| 1x32 | 0 | RX: RXCLK01, RXSOF[0], RXPRTY[0], RXDATA[31:0] TX: TXCLK01, TXSOF[0], TXPRTY[0], TXDATA[31:0] |
| | 1 | N/A |
| | 2 | N/A |
| | 3 | N/A |

8.4 MSF Mode Signal Usage

For tables specify the signal usage for each mode supported by the MSF and the mapping of these signals to the MSF pinout, see the *Intel IXP2400 Network Processor Datasheet*.

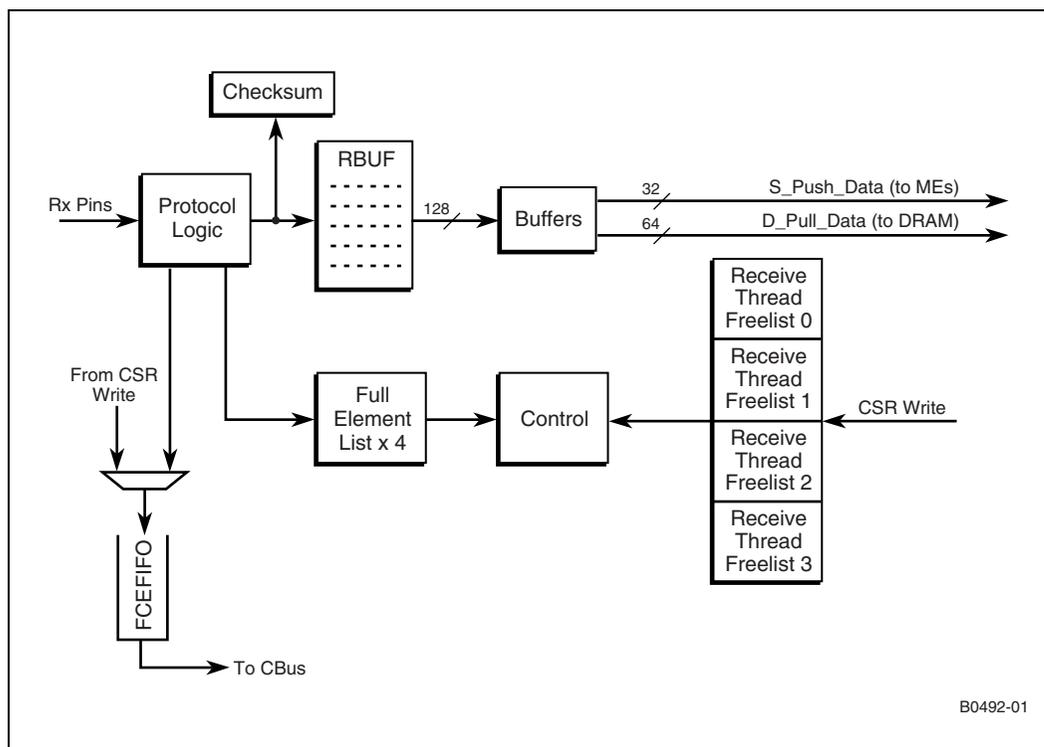
8.5 Receive

The [Section 8.5](#) discusses the following topics:

- Receive Pins in [Section 8.5.1](#).
- Receive Buffer (RBUF) in [Section 8.5.2](#).
- Receive Status Word in [Section 8.5.3](#).
- Full Element List in [Section 8.5.4](#).
- Rx_Thread_Freelists in [Section 8.5.5](#) and [Section 8.5.6](#).
- Receive Operation Summary in [Section 8.5.7](#)
- Flow Control Status in [Section 8.5.8](#)

[Figure 95](#) is a simplified Block Diagram of the Receive functionality.

Figure 95. Receive Functionality Simplified Block Diagram



8.5.1 Receive Pins and Protocol Logic

The receive pins are shared between the three protocols supported by the IXP2400 media block. The pins have been covered in detail in [Section 8.3, “Media Bus Interface”](#) on page 8-238, so this section will concentrate on the protocol logic.

There are three distinct sets of protocol logic:

- UTOPIA
- POS-PHY
- CSIX

8.5.1.1 UTOPIA SPHY

When running in UTOPIA/POS SPHY mode, the 32 receive data pins can be divided into one, two, three, or four independent channels. The total width of all the channels must be no more than 32 bits. Each receive channel has its own set of control signals; however, when running in any x8 mode, adjacent channels must share a clock. It is also possible to program different characteristics (cell vs. packet mode, decode response time, cell size, parity mode) for each individual channel.

There is a separate set of protocol logic for each channel. In UTOPIA SPHY mode, the protocol logic is responsible for monitoring the RXFA[3:0] signals to see which PHY has cells available in its receive FIFO, and for asserting RXENB to drain cells from the PHY into IXP2400's receive FIFOs. The protocol FSM monitors the RXSOF signal to determine the start of a cell. If IXP2400's receive FIFO backs up, the protocol FSM will implement flow control by deasserting RXENB.

Parity is checked for all receive data.

8.5.1.2 UTOPIA MPHY

There are two distinct modes of operation for UTOPIA MPHY.

The first is direct status. Each port in the PHY has its own status signal (RXFA[3:0]). Only a maximum of four ports may be supported in this mode. The protocol logic monitors the RXFA[3:0] signals to determine which port has valid cells and uses RXADDR[4:0] to select a port to drain the cell from. Flow control is achieved by deasserting RXENB.

The second is polled status. The protocol logic uses RXADDR[4:0] to poll for FIFO status, which is returned on the RXPFA signal. RXADDR[4:0] is also used to perform device selection during data transfer. A maximum of 32 ports is supported on a 32b bus and a maximum of 31 ports is supported on one 16b bus.

Parity is checked for all receive data.

8.5.1.3 POS-PHY SPHY

In POS-PHY SPHY mode, the protocol logic is responsible for monitoring the RXFA[3:0] signals to see which PHY has data available in its receive FIFO, and for asserting the RXENB signal associated with that port to drain the data from the PHY into IXP2400's receive FIFOs.

Data is drained in bursts, based on the size of the RBUF entry. The protocol FSM monitors the RXSOF signal to determine the start of a cell, the RXEOF to determine end of cell, and RXERR to determine if the packet should be marked as bad. RXVAL is used to qualify receive data. If IXP2400's receive FIFO backs up, the protocol FSM will implement flow control by deasserting RXENB to stop the flow of data from the PHY.

8.5.1.4 POS-PHY MPHY

In POS-PHY Level 3 MPHY mode, no polling is required. When IXP2400 is able and ready to accept receive data it asserts RXENB; the PHY will select a port to transfer data from, assert RSX to supply the address, then supply a burst of data. The protocol logic monitors RSX, address, RXSOF, RXEOF, RXVAL, and RXERR to delineate packet boundaries. Note that packets will generally not be delivered contiguously to IXP2400; packets from multiple ports will be interleaved. The PHY must support a minimum burst size of 64 bytes; 128 and 256 byte burst support is optional.

In POS-PHY Level 2 mode, polling is required. IXP2400 will output polling addresses on RXADDR[4:0] and examine the PHY's FIFO status on RXPFA. The FIFO watermarks on the PHY should be configured so RXPFA is not asserted unless there is at least one mpacket's (64, 128, or 256 bytes) worth of data (or an end-of-packet) in the FIFO. IXP2400 will also use RXADDR[4:0] for port selection. When IXP2400 selects a port, it will keep RXENB asserted for long enough to retrieve one mpacket of data. If an end-of-packet condition occurs and there is less than a full mpacket's worth of data, the slave is expected to assert RXEOF and during the last cycle of data and keep RXVAL deasserted afterwards. IXP2400 will then deassert RXENB and perform port selection again.

8.5.1.5 CSIX

In CSIX mode, the protocol logic is responsible for performing the following functions:

- monitoring the RXSOF signal for incoming CFrames
- checking the length field in the base header and writing the appropriate number of bytes into the RBUF element or FCEFIFO
- decoding the type field in the base header to determine the CFrame type (data vs. control vs. flow control) and routing to the appropriate destination (RBUF data, RBUF control, FCEFIFO, or dropping)
- checking horizontal and vertical parity
- sending incoming link level flow control information from the base header (SF_CRDY and SF_DRDY bits) to the ingress processor via CBus
- monitoring for error conditions (such as unexpected RXSOF) and taking appropriate action

8.5.2 RBUF

RBUF is a RAM that holds received data. It stores received data in sub-blocks (referred to as elements), and is accessed by MEs reading the received information. RBUF contains a total of 8KB of data. RBUF can be divided into one or two segments, depending on MSF_Rx_Control[RBUF_Element_Size]. When data is received, the associated status is put into the Full_Element_List FIFO and subsequently sent to MEs to process. The Full_Element_List insures that received elements are sent to the MEs in the order that they were received.

RBUF contains a total of 8 kb. of data. The data in each partition is divided into 64, 128, or 256 bytes elements, based on `MSF_Rx_Control[RBUF_Element_Size]`. In the case of two partitions, both partitions must have the same element size.

Table 89 shows the options for partitioning, partition usage, and element size.

Table 89. RBUF Partitioning Options

| Number of Partitions | Usage | Element Size | Partition 0 | Partition 1 |
|----------------------|-------------------|--------------|-------------------------|-------------------------|
| 0 | UTOPIA or POS-PHY | 64 bytes | 128 elements | N/A |
| | | 128 bytes | 64 elements | |
| | | 256 bytes | 32 elements | |
| 1 | CSIX | 64 bytes | 96 elements (0x00-0x5f) | 32 elements (0x60-0x7f) |
| | | 128 bytes | 48 elements (0x00-0x2f) | 16 elements (0x30-0x3f) |
| | | 256 bytes | 24 elements (0x00-0x17) | 8 elements (0x18-0x1f) |

In any of the UTOPIA/POS-PHY modes, RBUF functions as one large pool of elements which are shared by all channels.

In CSIX mode, RBUF is partitioned into two segments, partition 0 is meant to be used for data CFrames and partition 1 is meant to be used for control CFrames. Partitioning guarantees that there will always be room reserved for incoming control CFrames.

Table 90 below shows the order in which received data is stored in RBUF. Each number represents a byte, in order of arrival from the receive interface.

Table 90. RBUF Byte Ordering

| Byte Address (Hex) | | | | | | | | Address Offset |
|--------------------|----|----|----|----|----|----|----|----------------|
| 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 0x0 |
| C | D | E | F | 8 | 9 | A | B | 0x8 |
| 14 | 15 | 16 | 17 | 10 | 11 | 12 | 13 | 0x10 |

MEs can read data from the RBUF to ME transfer registers using the `msf[read]` instruction, where they specify the element number, offset into the element (which must be four byte aligned), and number of longwords to read. The length in the instruction can either be in units of longwords or quadwords, using the single or double instruction modifiers, respectively. Data is pushed to ME via SRAM Push Bus by RBUF control logic.

`msf[read, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token`

The source operands are added together to form the RBUF quadword address. `ref_cnt` is the number of longwords or quadwords, which are pushed into two sequential `S_Transfer_In` registers per quadword, starting with `$s_xfer_reg`.

Using the data in RBUF in Table 90, reading eight bytes from offset 0 into transfer registers 0 and 1 would yield the following results.

Table 91. SRAM Read Transfer Register Byte Ordering

| Transfer Register Number | [31:24] | [23:16] | [15:8] | [7:0] |
|--------------------------|---------|---------|--------|-------|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 | 7 |

MEs can move data from RBUF to DRAM using the instruction:

```
dram[rbuf_rd, $$s_xfer_reg, src_op1, src_op2, ref_cnt], optional_token
```

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 RBUF address. The `ref_cnt` operand is the number of quadwords which are read from RBUF.

Using data in RBUF in [Table 90](#), reading 16 bytes from offset 0 in RBUF into DRAM would yield the following.

Table 92. DRAM Byte Ordering

| DRAM Address | [63:56] | [55:48] | [47:40] | [39:32] | [31:24] | [23:16] | [15:8] | [7:0] |
|--------------|---------|---------|---------|---------|---------|---------|--------|-------|
| 0x0 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |
| 0x8 | C | D | E | F | 8 | 9 | A | B |

Note: DRAM addresses must be aligned to 8 byte boundaries.

For both types of RBUF read, reading an element does not move any RBUF pointers or destroy any data, so an element (or parts of an element) can be read as many times as desired.

RBUF elements are not time-stamped in the MSF block; they are time-stamped by the receive thread.

The status is specific to UTOPIA/POS or CSIX mode based on `MSF_Rx_Control[Receive_Mode]`.

A description of how RBUF elements are allocated and filled is based on the mode in `MSF_Rx_Control[Receive_Mode]` and is described in [Section 8.5.3.1–Section 8.5.3.3](#).

8.5.3 Receive Status Word

For each RBUF element, a 64 bit Receive Status Word is generated to describe the contents and status of the contents of the RBUF element. The format of the RSW depends upon the protocol. The RSWs are placed into the Full Elements FIFO to be sent to a receive thread for processing. The next three sections describe the loading of RBUF and the format of the Receive Status Word for the three different supported protocols.

8.5.3.1 UTOPIA Mode

The RBUF load procedure includes:

1. At chip reset, all elements are marked invalid (available).

- When RXSOF has been asserted, that is, when a new cell is received, an available RBUF element is allocated by receive control logic. The entire cell is written into the RBUF entry. Because the maximum size of the cell is 56 bytes (in 1x32 mode, with HEC/UDF bytes intact), it makes little sense to partition RBUF into 128 or 256 byte elements; 64 byte elements are the ideal size for RBUF entries in UTOPIA mode.

The status word contains the information in [Table 93](#):

Table 93. UTOPIA Receive Status Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---------|---------|-----|---|---|---|---|---|-----|------------|---|---|---|---|---|---|---|-----|-----|-----|----------|---------|----------|------|---------------|------------|-----|---|---|---------|---|---|--|
| 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | |
| Res | Element | | | | | | | | Byte Count | | | | | | | | SOP | EOP | Err | reserved | Par Err | reserved | Null | SOP Error Bit | MPHY-32 id | Res | | | Channel | | | |
| 6 | 6 | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 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 | |
| GFC_VPI | | VPI | | | | | | VCI | | | | | | | | | | | | PTI | | | | CLP | | | | | | | | |

The definitions of the fields are shown in [Table 94](#):

Table 94. UTOPIA Receive Status Word Field Definitions

| Field | Definition |
|----------------------------|--|
| Channel | The channel number from which the cell originated. Valid values are 0x0 to 0xf. |
| MPHY-32 id ^a | MPHY-32 Channel Identifier. This bit, when set, is used to indicate that the mpacket originated from the MPHY-32 port (port 0). This bit is provided to guarantee the uniqueness of channel numbers. Data received on the MPHY-32 port will have channel numbers 0x00 to 0x1f. However, in x16 MPHY-32 mode, the traffic received on the SPHY channels will have channel numbers of 0x01, 0x02, 0x03. |
| SOP Error Bit ^a | This bit is set under the following two conditions <ul style="list-style-type: none"> Multiple RXSOFs seen within one cell time (applies to both SPHY and MPHY modes). If the UTOPIA receive logic sees RXSOF asserted again while it is receiving a cell, it will set this bit in the Receive Status Word. The two cells (the first interrupted cell and the subsequent interrupting cell(s)) will essentially be merged together to form a single cell in the RBUF entry. This cell should be discarded. After the error occurs UTOPIA receive logic will be processing a cell when it sees the next RXSOF. Late RXSOF (applies only to MPHY mode). When the Rx MPHY arbiter grants a port, the port number is placed on the RXADDR[4:0] pins and RXENB is asserted to select the port number and drain the cell from the PHY. The PHY is expected to respond with RXSOF one or two cycles later, depending up the PHY's decode response time. If the PHY responds later than this, this indicates either a protocol violation or incorrect programming, and SOP Err will be set. ("Early" RXSOF and "unsolicited" RXSOF are ignored by MSF) |
| 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. |
| Par Err | Parity Error. If this bit is set, it means that a parity error was detected |

Table 94. UTOPIA Receive Status Word Field Definitions (Continued)

| Field | Definition |
|------------|---|
| Err | Error. If this bit is set, it means that a parity error or a protocol violation has been detected. In general, software should detect protocol violations and discard corrupted cells. The MSF hardware does not detect all kinds of protocol violations. In UTOPIA mode, the MSF hardware can detect whether RXSOF is asserted before the current cell receive completes. If so, the MSF hardware can flag this situation as protocol violation. Note that the A-step hardware does not detect or flag any UTOPIA Rx protocol violation. When such situation occurs, the MSF hardware ignores it. In B-0, the MSF hardware detects and flags this situation as protocol violation. |
| SOP | Start of packet bit. In UTOPIA mode this bit is always set. |
| EOP | End of packet bit. In UTOPIA mode this bit is always set. |
| Byte_Count | Indicates the number of total number of data bytes present in the cell; this includes both cell header and cell payload. Byte_Count should range from 52 to 56 bytes, depending upon bus width and cell size. Any value outside of this range indicates that an error has occurred. |
| 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 6/7/8 places based on the element size configured. |

a. Bits 7 and 8 of this field are only available in IXP2400 B0.

The entire four bytes of the ATM cell header is copied into the upper half of the Receive Status Word in order to help accelerate table lookups by having the VCI/VPI information available and avoiding the extra step of having the thread retrieve this information from the RBUF entry.

The definitions of the cell header fields are shown in [Table 95](#):

Table 95. Cell Header Field Definitions

| Field | Definition |
|---------|--|
| CLP | Cell Loss Priority. |
| PTI | Payload Type Identifier. |
| VCI | Virtual Circuit Identifier. |
| VPI | Virtual Path Identifier, bits [7:0] |
| GFC_VPI | UNI cells: Generic Flow Control NNI cells: Virtual Path Identifier, bits [11:8] |

The entire cell, including the header, is written into RBUF. If the HEC/UDF byte(s) are not stripped by the PHY (for example, 53 byte cell size), then the payload will not be nicely aligned on a four byte boundary.

8.5.3.2 POS-PHY Mode

The way in which RBUF is loaded is:

1. At chip reset, all elements are marked invalid (available).
2. The POS receive logic is burst oriented and will pull in 64, 128, or 256 byte bursts from the PHY depending upon RBUF entry size. Each burst is placed into an RBUF entry and a Receive Status Word is constructed. If this is the first burst (RXSOF was asserted), then the SOP bit is set. If this is the last burst (RXEOF was asserted) then the EOP bit is set. If this is neither the first nor last, then neither SOP or EOP is set.

- In POS-PHY SPHY or MPHY-4 operation, it is implied that all packets received through a given channel are contiguous; that is, there is no interleaving of mpackets from multiple packets; this is not true for MPHY-16 operation.

The status word contains the information in [Table 96](#)

The definitions of the fields are shown in [Table 97](#):

Table 96. POS-PHY Receive Status Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|---------|---|---|---|---|---|------------|---|---|---|---|---|-----|----------|-----|----------------------|---------|--------|------|-----------|-----------|-----|---|---|---------|---|---|---|---|---|
| 3 | 3 | 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 | | |
| Res | Element | | | | | | Byte Count | | | | | | SOP | EOP | Err | In-Band Addr Par Err | Par Err | RX Err | Null | SOP Error | MPHY32 id | res | | | Channel | | | | | |
| | 6 | 6 | 6 | 6 | 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 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| Reserved | | | | | | | | | | | | | | Checksum | | | | | | | | | | | | | | | | |

The definitions of the fields are shown in [Table 97](#):

Table 97. POS-PHY Receive Status Word Field Definitions

| Field | Definition |
|-------------------------|---|
| Channel | The channel number from which the cell originated. Valid values are 0x0 to 0xf. |
| MPHY-32 id ^a | MPHY-32 Channel Identifier. This bit, when set, is used to indicate that the mpacket originated from the MPHY-32 port (port 0). This bit is provided to guarantee the uniqueness of channel numbers. Data received on the MPHY-32 port will have channel numbers 0x00 to 0x1f. However, in x16 MPHY-32 mode, the traffic received on the SPHY channels will have channel numbers of 0x01, 0x02, 0x03. |
| SOP Error ^a | If the POS-PHY receive logic sees RXSOF asserted more than once within the same mpacket, without RXEOF being asserted in between, it will set this bit, indicating that a protocol violation has occurred, and that microcode needs to discard this mpacket and continue discarding mpackets until it sees an mpacket with RSW[EOP] set. |
| 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. |
| RX Err | Receive Error. If this bit is set, it means that RXERR was asserted during at the end of this packet. RX Err is valid if and only if EOP is also set. |
| Par Err | Parity Error. If this bit is set, it means that a parity error was detected |

Table 97. POS-PHY Receive Status Word Field Definitions (Continued)

| Field | Definition |
|----------------------|--|
| In-Band Addr Par Err | This is used only in SPI-3 MPHY-4/MPHY-32 mode to indicate that a parity error was seen during the in-band address cycle (RSX asserted). If the slave device sends one in-band address which results in a parity error, and sends multiple back-to-back bursts (mpackets) to that address, then all the mpackets will be marked with Addr Err. If the slave devices sends one in-band address for every burst (mpacket), then only mpackets that were preceded by an address parity error will be marked with Addr Err. The MSF_Interrupt_Status[HP_Error] bit will be set if an address parity error is seen. AddrErr has no meaning in POS-PHY L2 MPHY mode since in-band addressing is not used in that protocol, and should always be 0. |
| Err | Error. If this bit is set, it means that a Receive error, a Parity error, or a protocol violation is detected. If the POS-PHY receive protocol logic sees RXSOF asserted twice within the same mpacket, without RXEOF being asserted in between, it raises a protocol violation. In general, software should detect protocol violations and discard corrupted packets. The MSF hardware does not detect all kinds of protocol violations. |
| SOP | Start of Packet and End of Packet bits. These bits are used to delineate start and end of packet. {SOP,EOP} |
| EOP | 00: This is neither the first nor last mpacket, but one in the middle. This also implies that the mpacket contains 64 bytes of valid data. 01: This is the last mpacket. The number of valid bytes is specified in the Byte_Count field. 10: This is the first mpacket of the packet. Since EOP wasn't asserted, it is assumed that the mpacket contains 64 bytes of valid data. 11: The entire packet is contained within this mpacket. The length of the packet is in Byte_Count. |
| Byte_Count | Indicates the number of total number of data bytes present; valid values range from 1 to 256 bytes; 256 bytes is encoded as 0x00. |
| 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 6/7/8 places based on the element size configured. |
| Checksum | Ones complement 16-bit checksum for the mpacket. The checksum is calculated over the entire mpacket, but excludes the Receive Status Word. It is up to software to sum up all the checksums for the mpackets comprising the packet, and subtract out any headers, trailers, etc. |

a. Bit 7 and 8 of this field are only available in IXP2400 B0.

The Checksum is calculated over the entire packet, 16 bits at a time. The operator '+' indicates 1's complement addition. If the mpacket is n bytes long, and n is an even number, the formula used to calculate the checksum is:

$$\{\text{byte } 0, \text{ byte } 1\} + \{\text{byte } 2, \text{ byte } 3\} + \dots \{\text{byte } n-2, \text{ byte } n-1\}$$

If the mpacket is n bytes long, and n is an odd number, then the formula used to calculate the checksum is:

$$\{\text{byte } 0, \text{ byte } 1\} + \{\text{byte } 2, \text{ byte } 3\} + \dots \{\text{byte } n-1, 0x00\}$$

In general, it is up to microcode (software) to detect protocol violations. Protocol violations include such situations as two SOPs without an intervening EOP, or two EOPs without an intervening SOP. It is also up to microcode to drop the corrupted data. A properly designed slave device should never generate these protocol violations. The hardware does not filter out data, but attempts to put enough information in the Receive Status Word so that microcode can make the correct decision. Here are special situations that the receive microcode needs to be able to handle:

1. If the receive microcode has seen an SOP, and is waiting for the EOP, but instead receives another SOP, then it should discard all of the previous packet (which is probably corrupted) and the next entire packet as well (which may also be corrupted).
2. If the receive microcode has seen an EOP, and is expecting the SOP for the next packet, but instead receives either another EOP or a mpacket with neither SOP nor EOP, then it should start discarding mpackets until it sees an EOP. The EOP mpacket should be discarded as well. After that, it can resume looking for the next SOP. (**NOTE:** this situation can only occur in MPHY-4 or MPHY modes, and should never occur in SPHY mode.)
3. If the receive microcode sees an mpacket with the Err bit set, but Par Err and Rx Err are not set, then what has happened is that the receive protocol logic has seen two SOPs within the same mpacket. Microcode should discard this mpacket, and continue discarding mpackets, until it sees an EOP. The EOP mpacket should be discarded as well. After that, it can resume looking for the next SOP.

8.5.3.3 CSIX Mode

CSIX CFrames are placed into either RBUF or FCEFIFO as follows:

1. At chip reset all RBUF elements are marked invalid (available) and FCEFIFO is empty.
2. When RxSof is asserted and a base header is received, it is stored in a temporary holding register. The CRDY and DRDY fields are extracted and held to be placed into FC_Egress_Status[SF_CRDY] and [SF_DRDY] fields at the start of a CFrame. If a parity error is detected, the flags are cleared as soon as the error is detected. The Type field is used to index into the CSIX_Type_Map CSR to determine what to do with the CFrame:
 - a. Discard (except for the CRDY/DRDY fields as described above)
 - b. Place into RBUF control partition.
 - c. Place into RBUF data partition.
 - d. Place into FCEFIFO.

Note: The CSIX_Type_Map register provides processing flexibility. Normally, Idle CFrames (type 0x0) are discarded. Command/Status CFrames (type 0x7) should be placed in the control partition. Flow control CFrames (type 0x6) should be placed into FCEFIFO. All others (unicast, multicast, broadcast) should be placed into the data partition.

3. If the action is discard, the entire CFrame is discarded, but the CRDY/DRDY bits are placed into FC_Egress_Status as described above.
4. If the destination is FCEFIFO, the entire CFrame is placed into FCEFIFO to be sent to the ingress IXP2400 over the TXCDATA pins. If there is not enough room in FCEFIFO for the entire CFrame (based on the Payload Length in the base header), the entire CFrame is discarded and MSF_Interrupt_Status[FCEFIFO_Overflow] is set.
5. If the destination is RBUF, either control or data, an available RBUF element of the corresponding type is allocated by the receive control logic. If there is no available element the CFrame is discarded and MSF_Interrupt_Status[RBUF_Overflow] is set. If an element is allocated, the Type, Payload Length, CR (CSIX reserved), P (private), and extension header are placed into a temporary holding registers. As the payload is received, it is placed into the RBUF element starting at offset 0x00.

Note: For unicast, multicast, and broadcast CFrames, the first four bytes after the base header is the extension header. This will be placed in the Extension Header field of the Receive Status Word. However, flow control CFrames have no extension header, so in this case the Extension Header is

undefined and should be ignored. Reserved CFrame types are handled as if they have an extension header.

- When all of the payload data is received (per the Payload Length field), the element is marked valid. If another RxSof is received prior to receiving the entire payload, the element is 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 status bit is set and the payload bytes above the element size will be discarded. The temporary register value is put into Full_Element_List.

Note: In CSIX mode, the element partitioning 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.

Payload information is put into the Payload area of the element, and Base and Extension Header information is put into the Header Status area of the element.

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 longword, etc).

The Header Status status word is described in [Table 98](#):

Table 98. CSIX Receive Status Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------------------|---------|---|---|---|---|---|---|---|----------------|---|---|---|---|---|---|---|----|---|----|--------|-------|-------|------|----------|---|---|---|------|---|---|---|
| 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 | |
| Res | Element | | | | | | | | Payload Length | | | | | | | | CR | P | Er | Len Er | HP Er | VP Er | Null | Reserved | | | | Type | | | |
| | 6 | 6 | 6 | 6 | 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 | 3 |
| Extension Header | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

The definitions of the fields are in [Table 99](#):

Table 99. CSIX Receive Status Word Field Definitions

| Field | Definition |
|--------|--|
| Type | Type Field from the CSIX Base Header |
| 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. |
| VP Err | Vertical Parity Error was detected on the CFrame. See the description in Section 8.5.9 . |
| HP Err | Horizontal Parity Error was detected on the CFrame. See the description in Section 8.5.9 . |

Table 99. CSIX Receive Status Word Field Definitions

| Field | Definition |
|------------------|---|
| Length Err | Length Error; the MSF hardware detects that another RxSof has been asserted before the full Payload was received. The MSF hardware does not detect the late RsSof case, i.e., the actual amount of Payload received is more than the length that the base header specifies. In this case, a Vertical Parity Error will likely result because the MSF hardware will be expecting to receive vertical parity but receiving payload data instead. Length Error normally also results in a VP Err, although it is possible that only Length Err is asserted. This indicates that the CFrame that follows the interrupted CFrame contains the vertical parity for the interrupted CFrame. The user should not assume that if Length Err is asserted, VP Err is also asserted. |
| Err | Error. If this bit is set, it means that a Receive error, a Parity error, or a protocol violation is detected. If the POS-PHY receive protocol logic sees RXSOF asserted twice within the same mpacket, without RXEOF being asserted in between, it raises a protocol violation. In general, software should detect protocol violations and discard corrupted packets. The MSF hardware does not detect all kinds of protocol violations. |
| 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 6/7/8 places based on the element size configured. |
| P | P (Private) bit from the CSIX Base Header. |
| CR | CR (CSIX Reserved) bit from the CSIX Base Header. |
| Payload Length | Payload Length field from the CSIX Base Header. A value of 0x0 indicates 256 bytes. |
| Extension Header | The Extension Header from the CFrame. For flow control CFrames this field is undefined because flow control CFrames do not have an extension header. |

8.5.4 Full Element List

Receive control hardware maintains the Full Element List to hold the element numbers of valid RBUF elements, in the order in which they were received. When an element is marked valid (as described previously), its element number is added to the tail of the Full Element List. When a ME is notified of element arrival (by having the status written to its SRAM Transfer register; see [Section 8.5.5, “Rx_Thread_Freelists”](#)), it is removed from the head of the Full Element List.

Essentially, the Full Element list stores Receive Status Words. The RSW is autopushed to the receive thread in the following order:

1. [31:0] to xfer reg n
2. [63:32] to xfer reg n+1

There are four Full Element Lists, one for each SPHY channel or one of the four MPHY channels in MPHY4 mode. The capacity of these four Full Element Lists is as follows:

- Full Element List 0: 128 entries

- Full Element List 1: 32 entries
- Full Element List 2: 64 entries
- Full Element List 3: 32 entries

Full Element List 0 has the capacity to hold the maximum number of outstanding RBUF entries, which is 128 (8 Kbyte divided by 64-byte minimum RBUF element size). The size of the Full Element Lists does not change with RBUF element size. Since some of the Full Element Lists hold less than the maximum 128 entries, blocking may happen, even when there are available RBUF elements. For instance, when Full Element List 1 is full, a new mpacket that is targeted for channel 1 can block subsequent mpackets, even though free RBUF elements are available. Note that in this case, the RBUF_Overflow_Counter register does not get incremented.

8.5.5 Rx_Thread_Freelists

The Rx_Thread_Freelists are four FIFOs which indicate ME 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 the Freelist also has an associated SRAM transfer register and signal number. Each RX_Thread_Freelist is associated with a receive bus channel.

The number of entries in each RX_Thread_Freelist is shown in [Table 100](#).

Table 100. RX_Thread_Freelist Entries

| FreeList # | # of entries |
|----------------------|--------------|
| RX_Thread_Freelist_0 | 64 |
| RX_Thread_Freelist_1 | 16 |
| RX_Thread_Freelist_2 | 32 |
| RX_Thread_Freelist_3 | 16 |

To be added as ready to receive an element, an ME does a `msf[write]` or `msf[fast_write]` to the Rx_Thread_Freelist address; the data of the CSR write is the ME/Context/S_Transfer Register number/Signal 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 a valid element at the head of the Full Element List its status will be pushed to an ME. The receive control logic pushes the element status information (which includes the element number) to the ME in the head entry of Rx_Thread_Freelist. It then removes that thread from the Rx_Thread_Freelist, sends an Event Signal to the ME, and removes the element from Full Element List. See [Section 8.5.6, “Rx_Thread_Freelist Timeout”](#) on page 8-266 for more detail.

Note: A Context waiting on status must *not* read the S_Transfer register until it has been signaled.

In the event that Rx_Thread_Freelist is empty, valid element numbers will be held in Full Element List until an entry is put into Rx_Thread_Freelist.

The number of available Rx_Thread_Freelists and how they are assigned to channels depends upon the operating mode.

8.5.5.1 UTOPIA and POS-PHY SPHY Modes

Table 101 shows which Rx_Thread_Freelist is associated with a given channel when running in either UTOPIA or POS_PHY SPHY modes.

Table 101. Rx_Thread Freelist Association in UTOPIA and POS-PHY SPHY Modes

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| 1x32 | 0 | Rx_Thread_Freelist_0 |
| 2x16 | 2 | Rx_Thread_Freelist_2 |
| | 0 | Rx_Thread_Freelist_0 |
| 4x8 | 3 | Rx_Thread_Freelist_3 |
| | 2 | Rx_Thread_Freelist_2 |
| | 1 | Rx_Thread_Freelist_1 |
| | 0 | Rx_Thread_Freelist_0 |
| 1x16_2x8 | 3 | Rx_Thread_Freelist_3 |
| | 2 | Rx_Thread_Freelist_2 |
| | 0 | Rx_Thread_Freelist_0 |

8.5.5.2 UTOPIA/POS-PHY MPHY-4 Mode

In UTOPIA/POS-PHY MPHY-4 mode, one to four ports share the 32 bit bus; each port has its own freelist, as shown in Table 102.

Table 102. Rx_Thread Freelist Association in UTOPIA/POS-PHY MPHY-4 Mode

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| 1x32 | 0 | Rx_Thread_Freelist_0 |
| | 1 | Rx_Thread_Freelist_1 |
| | 2 | Rx_Thread_Freelist_2 |
| | 3 | Rx_Thread_Freelist_3 |

8.5.5.3 UTOPIA/POS-PHY MPHY-32 Mode

In UTOPIA or POS-PHY MPHY-32 mode, traffic from all 32 ports funnel into a single RBUF; each mpacket is tagged with the port number from which it originated, as shown in Table 103.

Table 103. Rx_Thread Freelist Association in UTOPIA/POS-PHY MPHY-32 Mode

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| 1x32 | 0–31 | Rx_Thread_Freelist_0 |

8.5.5.4 UTOPIA/POS-PHY 16 Bit MPHY + 8 Bit SPHY + 8 Bit SPHY

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| x16 MPHY | 0 | Rx_Thread_Freelist_0 |
| x8 SPHY | 2 | Rx_Thread_Freelist_2 |
| x8 SPHY | 3 | Rx_Thread_Freelist_3 |

8.5.5.5 UTOPIA/POS-PHY 16 Bit MPHY + 16 Bit SPHY

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| x16 MPHY | 0 | Rx_Thread_Freelist_0 |
| x16 SPHY | 2 | Rx_Thread_Freelist_2 |

8.5.5.6 CSIX Mode

In CSIX mode, data and control CFrames can either have individual Rx_Thread_Freelists or share a Rx_Thread_Freelist, depending upon the MSF_Rx_Control[CSIX_Freelist] bit.

Table 104 shows freelist parameters when freelists are individual.

Table 104. Rx_Thread Freelist Association in CSIX Mode, MSF_Rx_Control[CSIX_Freelist]=0

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|-------------|---------------------------|
| 1x32 | Data | Rx_Thread_Freelist_0 |
| | Control | Rx_Thread_Freelist_1 |

Table 105 shows freelist parameters when freelists are shared.

Table 105. Rx_Thread Freelist Association in CSIX Mode, MSF_Rx_Control[CSIX_Freelist]=1

| Receive Width | Port Number | RX Thread Freelist Number |
|---------------|------------------|---------------------------|
| 1x32 | Data and Control | Rx_Thread_Freelist_0 |

8.5.6 Rx_Thread_Freelist Timeout

Each Rx_Thread_Freelist has an associated countdown timer. If the timer expires and no new receive data is available, 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 MEs) moving even when the line has gone idle. It is especially useful if the pipeline is structured to handle mpackets in groups, for example, 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.

8.5.7 Receive Operation Summary

Received cells, CFrames, or packets (which in this context are both called mpackets) are placed into the RBUF, and then, when marked valid, are immediately handed off to ME to process. Normally some number of Contexts will be assigned to receive processing. Those Contexts will have their number added to the 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 previously described, receive control logic will autopush eight bytes of information (the entire Receive Status Word) for the element to the ME/Context/S_Transfer Registers at the head of Rx_Thread_Freelist

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 a Rx_Thread_Freelist entry is entered, the status of the head element of Full Element List will be pushed to it.

The MEs may read part of (or the entire) RBUF element to their SRAM transfer registers (via `msf[read]` instruction) for header processing, etc, and may also move the element data to SDRAM (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 data of the CSR write is the element number. This marks the element as free and available to be re-used.

The states that an RBUF element goes through are shown in [Figure 96](#).

Figure 96. RBUF Element State Transition Diagram

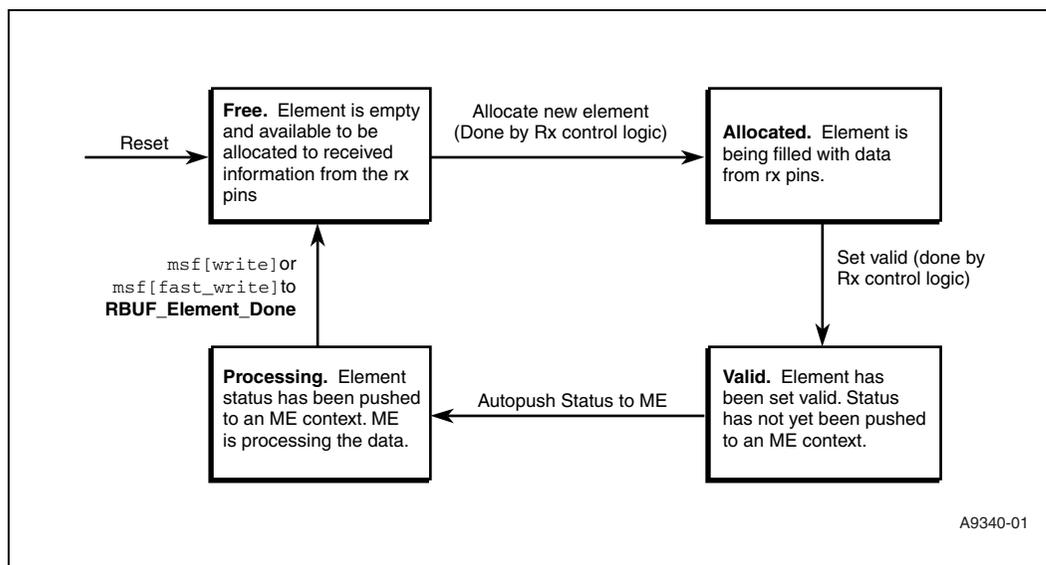


Table 106 summarizes the differences in UTOPIA mode, POS-PHY mode, and CSIX mode.

Table 106. UTOPIA, POS-PHY, and CSIX Mode Comparison

| Comparison Point | UTOPIA Mode | POS-PHY Mode | CSIX Mode |
|---|---|---|--|
| Point at which RBUF Element is allocated | When new cell is received (RXSOF asserted) | When the start of a new packet is received (RXSOF asserted) or a new burst of data arrives. | Start of Frame and Base Header Type is anything except Idle or Flow Control. |
| Quantity of Data put into Element | The entire cell. | A burst length worth of data, or until RXEOF is asserted (signaling end of packet); whichever comes first. The burst length is equal to the size of the RBUF entry (64, 128, or 256 bytes). | Number of bytes specified in Payload Length field of Base Header. |
| Mechanism for setting RBUF Element Valid | When the entire cell has been received and the Receive Status Word has been created for the cell. | When the entire burst has been received and the Receive Status Word has been created for the burst. | Number of bytes specified in Payload Length field of Base Header (or if new SOF, which is an error). |
| Mechanism to hand RBUF Element to ME | Element status is pushed to ME at the head of the appropriate Rx_Thread_Freelist. | | |
| Mechanism to return RBUF Element to free list | CSR write to RBUF_Element_Done. | | |

8.5.8 Receive Flow Control Status

Flow control is handled in hardware, and is based on MSF_Rx_Control[Receive_Mode], which selects UTOPIA/POS or CSIX mode.

8.5.8.1 UTOPIA and POS-PHY Mode

Link level flow control information is passed directly from the PHY to the protocol logic in IXP2400 using sideband signals. There is no need to communicate this information via CBus.

8.5.8.2 CSIX Mode

When running in CSIX half-duplex mode, both link level and fabric level flow control information is forwarded from the egress processor to the ingress processor using the CBus. This is described in detail in [Section 8.7, “CBus Interface”](#) on page 8-287.

8.5.9 Parity

8.5.9.1 UTOPIA Mode

UTOPIA requires odd single bit parity.

8.5.9.2 POS-PHY Mode

POS-PHY Level 2 allows both single bit even and odd parity; POS-PHY Level 3 only allows single bit odd parity. IXP2400 supports both single bit even and odd parity.

8.5.9.3 CSIX Mode

CSIX specifies both horizontal and vertical parity. Both are odd.

8.5.9.3.1 Horizontal Parity

The receive logic computes Horizontal Parity on each received CWord. There is an internal HP Error Flag. At the end of each CFrame the flag is reset.

As 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 MSF_Interrupt_Status[HP_Error] flag is set. The value of the flag becomes the element status HP Err bit.

If the HP Error Flag is set, the SF_CRDY and SF_DRDY bits are cleared, and the MSF_Interrupt_Enable[HP_Error] bit is set, this will send an interrupt to Intel XScale® core.

8.5.9.3.2 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. That is, 16 bits of vertical parity are formed on 32 bits of received data by treating the data as words; 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 vertical parity value should be 0xFFFF. If it is not, then the VP Error Flag is set. The value of the flag becomes the element status VP Err bit.

Any padding that was inserted to force the CFrame to be an integral multiple of CWords is also stripped out by the protocol logic.

If the VP Error Flag is set, the SF_CRDY and SF_DRDY bits are cleared, and the MSF_Interrupt_Status[VP_Error] bit is set, which can interrupt Intel XScale® core.

8.5.10 Error Cases

Receive errors are specific to the selected mode, UTOPIA, POS-PHY, or CSIX. The element status, described above, has appropriate error bits defined. Also, there are some IXP2400 specific error cases, like when an mpacket arrives with no free elements, which are logged in the MSF_Interrupt_Status register, which can interrupt Intel XScale® core if enabled.

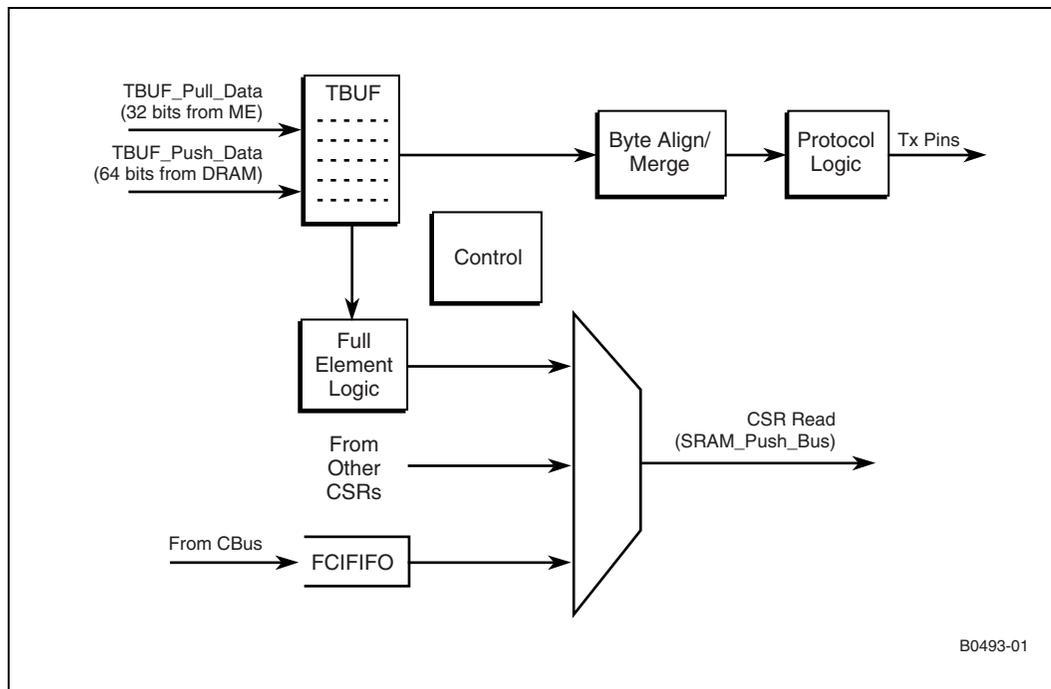
8.6 Transmit

This section covers:

- Transmit Pins (Section 8.6.1)
- TBUF and Transmit Control Word (Section 8.6.2)
- Byte Aligner

Figure 97 is a simplified Block Diagram of the Transmit function.

Figure 97. Transmit Function Simplified Block Diagram



8.6.1 Transmit Pins

The transmit pins are shared between the three protocols supported by the IXP2400 media block. There are three distinct sets of protocol logic:

- UTOPIA
- POS-PHY
- CSIX

The transmit pins include 32 transmit data pins, two transmit clocks, and four sets of control signals. There are two major modes: CSIX mode and UTOPIA/POS mode.

When running in CSIX mode, all 32 transmit data pins are used, but only one transmit reference clock and one set of control signals are needed.

When running in UTOPIA/POS-PHY SPHY mode, the 32 transmit data pins can be divided into one, two, three, or four independent channels. The total width of all the channels must be no more than 32 bits. Each transmit channel has its own set of control signals and may be programmed for either UTOPIA or POS-PHY operating modes; however, when running in x8 mode, adjacent channels must share a clock. It is also possible to program different characteristics (cell size, parity mode) for each individual channel.

When running in UTOPIA/POS-PHY MPHY mode, the 32 transmit data pins are shared by one to sixteen channels, all of which must reside in a single physical device.

8.6.2 TBUF and Transmit Control Word

The TBUF is a RAM that holds data and status to be transmitted. The data is written into subblocks referred to as elements, by MEs or Intel XScale® core. TBUF contains a total of 8KB of data and associated control.

The data is partitioned into elements, based on `MSF_Tx_Control[TBUF_Element_Size]`.

Table 107 shows the order in which data is written into TBUF. Each number represents a byte, in order of transmission on the TX interface. Note that this is reversed on a 32 bit basis relative to RBUF. The swap of the low longword and the high longword is done by hardware to facilitate the transmission of bytes as defined below.

Table 107. TBuf Byte Ordering

| Byte Address (Hex) | | | | | | | | Address Offset |
|--------------------|----|----|----|----|----|----|----|----------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0x0 |
| 8 | 9 | A | B | C | D | E | F | 0x8 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 0x10 |

MEs can write data from ME SRAM write transfer registers to the TBUF using the `msf[write]` instruction, where they specify the starting byte number (which must be aligned to four bytes) and the number of longwords to write. The length in the instruction can be either the number of longwords or the number of quadwords, using the single and double instruction modifiers, respectively. Data is pulled from ME to TBUF via SRAM 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 TBUF is 0x2000). `ref_cnt` is the number of longwords or quadwords, which are pulled from two `S_Transfer_Out` registers per quadword, starting with `$s_xfer_reg`. For example, when writing a quadword from `S_Transfer_Out` register 0/1 to address 0x0 in the TBUF entry (as shown in Table 107), the byte ordering is shown in Table 108.

Table 108. SRAM Write Transfer Register Byte Ordering

| Transfer Register Number | [31:24] | [23:16] | [15:8] | [7:0] |
|--------------------------|---------|---------|--------|-------|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 | 7 |

MEs can also write data from DRAM to TBUF using the `dram` instruction. Data is pushed to TBUF via DRAM Push Bus by DRAM controller.

```
dram[tbuf_wr, --, src_op_1, src_op_2, ref_cnt], indirect_ref
```

The `src_op_1` and `src_op_2` operands are added together to form address in DRAM, so the `dram` instruction must use indirect mode to specify the TBUF address. `ref_cnt` is the number of quadwords which are written into TBUF.

Data is stored in big-endian order. The most significant byte of each longword is transmitted first.

All data is transmitted in the order in which it is put into the TBUF. Control information associated with the element (and defined below), defines which bytes are valid. The data from the TBUF will be shifted and aligned to the TXDATA pins as required. Four parameters are defined.

1. Prepend Offset: number of the first byte to send. This is information that is prepended to the payload (for example, a header). The offset can range from 0x0 to 0x7 within the first quadword of the TBUF element.
2. Prepend Length: number of bytes in the prepend. This can range from 0 to 31 bytes.
3. Payload Offset: number of bytes to skip, from the last quadword of the prepend to the start of payload.
4. Payload Length: number of bytes in the payload.

Here are some rules and observations:

1. If prepend length is zero, then the prepend offset must also be zero.
2. The absolute byte number of the first byte of the payload is given by the following formula:

$$((\text{Prepend Offset} + \text{Prepend Length} + 0x7) \&\& 0xF8) + \text{Payload Offset}$$
3. The sum $((\text{Prepend Offset} + \text{Prepend Length} + 0x7) \&\& 0xF8) + \text{Payload Offset} + \text{Payload Length}$ must not exceed the size of the TBUF element.
4. In UTOPIA mode, the sum of Prepend Length and Payload Length must be equal to the cell size.
5. In POS-PHY mode, the sum of the Prepend Length and Payload Length must be an integral multiple of the bus width, except if EOP is set.

The following examples illustrates the use of the offset and length parameters. The element is shown as eight bytes wide, because that is the smaller unit which can be written to TBUF. In this example:

1. Prepend Offset = 0x6 (bytes 0x0 to 0x5 are skipped)
2. Prepend Length = 0x10 (16 bytes)
3. Payload Offset = 0x7 (bytes 0x16 to 0x1E are not transmitted). This starts in the next eight byte row (that is, the next *empty* row above where the prepend data stops), even if there is room in the last row containing prepend data. This is done because the TBUF does not have byte write capability, and the `msf[write]` and `dram[tbuf_write]` cannot be merged. 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.
4. Payload Length = 0x20 (32 bytes)

Table 109 shows a 64 byte TBUF entry. Bytes 0x06 to 0x15 are the prepend data and bytes 0x1f to 0x3E are the payload data.

Table 109. Example of Offset and Length Usage

| | | | | | | | |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 0x00 | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 |
| 0x08 | 0x09 | 0x0A | 0x0B | 0x0C | 0x0D | 0x0E | 0x0F |
| 0x10 | 0x11 | 0x12 | 0x13 | 0x14 | 0x15 | 0x16 | 0x17 |
| 0x18 | 0x19 | 0x1A | 0x1B | 0x1C | 0x1D | 0x1E | 0x1F |
| 0x20 | 0x21 | 0x22 | 0x23 | 0x24 | 0x25 | 0x26 | 0x27 |
| 0x28 | 0x29 | 0x2A | 0x2B | 0x2C | 0x2D | 0x2E | 0x2F |
| 0x30 | 0x31 | 0x32 | 0x33 | 0x34 | 0x35 | 0x36 | 0x37 |
| 0x38 | 0x39 | 0x3A | 0x3B | 0x3C | 0x3D | 0x3E | 0x3F |

Table 110 shows, in bold type and for a 32 bit bus, what data is transmitted. The transmit hardware will discard unwanted leading and trailing bytes, merge the prepend and payload data, and realign to fit the 32 bit bus.

Table 110. Example of TBUF Element Transmission

| clock cycle | [31:24] | [23:16] | [15:8] | [7:0] |
|-------------|-------------|-------------|-------------|-------------|
| 0 | 0x06 | 0x07 | 0x08 | 0x09 |
| 1 | 0x0A | 0x0B | 0x0C | 0x0D |
| 2 | 0x0E | 0x0F | 0x10 | 0x11 |
| 3 | 0x12 | 0x13 | 0x14 | 0x15 |
| 4 | 0x1f | 0x20 | 0x21 | 0x22 |
| 5 | 0x23 | 0x24 | 0x25 | 0x26 |
| 6 | 0x27 | 0x28 | 0x29 | 0x2A |
| 7 | 0x2B | 0x2C | 0x2D | 0x2E |
| 8 | 0x2F | 0x30 | 0x31 | 0x32 |
| 9 | 0x33 | 0x34 | 0x35 | 0x36 |
| 10 | 0x37 | 0x38 | 0x39 | 0x3A |
| 11 | 0x3B | 0x3C | 0x3D | 0x3E |

8.6.2.1 UTOPIA/POS-PHY SPHY TBUF Partitioning

The transmit control logic manages the TBUF as either one, two, three, or four separate segments based on the selected operating mode. The mapping of elements to channels, as well as to the associated freelist, depends on the operating mode.

Table 111. UTOPIA/POS-PHY SPHY TBUF Partitioning

| TX Channels | Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------|-------------------------|----------------|--------------------------|
| 1x32 | 32 x 256 bytes | 0 | 0–31 |
| | 64 x 128 byte | 0 | 0–63 |
| | 128 x 64 bytes | 0 | 0–127 |
| 2x16 | 32 x 256 bytes | 0 | 0–15 |
| | | 2 | 16–31 |
| | 64 x 128 bytes | 0 | 0–31 |
| | | 2 | 32–63 |
| | 128 x 64 bytes | 0 | 0–63 |
| | | 2 | 64–127 |
| 4x8 | 32 x 256 bytes | 0 | 0–7 |
| | | 1 | 8–15 |
| | | 2 | 16–23 |
| | | 3 | 24–31 |
| | 64 x 128 bytes | 0 | 0–15 |
| | | 1 | 16–31 |
| | | 2 | 32–47 |
| | | 3 | 48–63 |
| | 128 x 64 bytes | 0 | 0–31 |
| | | 1 | 32–63 |
| | | 2 | 64–95 |
| | | 3 | 96–127 |
| 1x16_2x8 | 32 x 256 bytes | 0 | 0–15 |
| | | 2 | 16–23 |
| | | 3 | 24–31 |
| | 64 x 128 bytes | 0 | 0–31 |
| | | 2 | 32–47 |
| | | 3 | 48–63 |
| | 128 x 64 bytes | 0 | 0–63 |
| | | 2 | 64–95 |
| | | 3 | 96–127 |

8.6.2.2 UTOPIA/POS-PHY MPHY-4 TBUF Partitioning

Table 112. UTOPIA/POS-PHY MPHY-4 TBUF Partitioning

| TX Channels | Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------|-------------------------|----------------|--------------------------|
| 1x32 | 32 x 256 bytes | 0 | 0–7 |
| | | 1 | 8–15 |
| | | 2 | 16–23 |
| | | 3 | 24–31 |
| | 64 x 128 bytes | 0 | 0–15 |
| | | 1 | 16–31 |
| | | 2 | 32–47 |
| | | 3 | 48–63 |
| | 128 x 64 bytes | 0 | 0–31 |
| | | 1 | 32–63 |
| | | 2 | 64–95 |
| | | 3 | 96–127 |

8.6.2.3 UTOPIA/POS-PHY MPHY-32 TBUF Partitioning

Table 113. UTOPIA/POS-PHY MPHY-32 TBUF Partitioning

| TX Channels | Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------|-------------------------|----------------|--------------------------|
| 1x32 | 32 x 256 bytes | 0–31 | 0–31 |
| | 64 x 128 bytes | 0–31 | 0–63 |
| | 128 x 64 bytes | 0–31 | 0–127 |

8.6.2.4 UTOPIA/POS-PHY16 Bit MPHY + 8 Bit SPHY + 8 Bit SPHY

| Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------------------|----------------|--------------------------|
| 32 x 256 bytes | 0 (x16 MPHY) | 0–15 |
| | 2 (x8 SPHY) | 16–23 |
| | 3 (x8 SPHY) | 24–31 |
| 64 x 128 bytes | 0 (x16 MPHY) | 0–31 |
| | 2 (x8 SPHY) | 32–47 |
| | 3 (x8 SPHY) | 48–63 |
| 128 x 64 bytes | 0 (x16 MPHY) | 0–63 |
| | 2 (x8 SPHY) | 64–95 |
| | 3 (x8 SPHY) | 96–127 |

8.6.2.5 UTOPIA/POS-PHY 16 Bit MPHY + 16 Bit SPHY

| Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------------------|----------------|--------------------------|
| 32 x 256 bytes | 0 (x16 MPHY) | 0–15 |
| | 2 (x16 SPHY) | 16–31 |
| 64 x 128 bytes | 0 (x16 MPHY) | 0–31 |
| | 2 (x16 SPHY) | 32–63 |
| 128 x 64 bytes | 0 (x16 MPHY) | 0–63 |
| | 2 (x16 SPHY) | 64–127 |

8.6.2.6 CSIX TBUF Partitioning

Three quarters of TBUF is allocated for data CFrames; the remaining quarter is allocated for control CFrames.

Table 114. CSIX TBUF Partitioning

| TX Channels | Number of TBUF Elements | Channel Number | Elements Used by Channel |
|-------------|-------------------------|----------------|--------------------------|
| 1x32 | 32 x 256 bytes | data | 0–23 |
| | | control | 24–31 |
| | 64 x 128 bytes | data | 0–47 |
| | | control | 48–63 |
| | 128 x 64 bytes | data | 0–95 |
| | | control | 96–127 |

8.6.2.7 UTOPIA Transmit Control Word Format

In UTOPIA mode, the complete cell is put into the data portion of the element, and information for the Control Word is written. The Control Word format is in [Table 115](#):

Table 115. UTOPIA Transmit Control Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------------|---|---|---|---|----------------|---|----------------|---|---|---|----------------|---|---|-----|------|-----|-----|-----|------------|-----|---|---------|---|---|---|---|---|---|---|
| 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 | ERR | SOP | EOP | MPHY-32 id | res | | Channel | | | | | | | |
| 6 | 6 | 6 | 6 | 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 | 3 |
| 3 | 2 | 1 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 6 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | |
| Res | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

The definitions of the fields are in [Table 116](#):

Table 116. UTOPIA Transmit Control Word Field Definitions

| Field | Definition |
|----------------|--|
| Payload Length | Indicates the number of bytes in the payload, from 1 to 256 bytes, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent, and should be equal to the cell size, as specified by MSF_Tx_Control[Transmit_Width] and Tx_UP_Control_{0..3}[Cell_Size]. The only valid cell sizes in UTOPIA mode are 52, 53, 54, and 56 bytes. |
| Prepend Offset | Indicates the first valid byte of the prepend, from 0 to 7 bytes. |
| Prepend Length | Indicates the number of bytes of the prepend from 0 to 31 bytes. |
| Payload Offset | Indicates the first valid byte of the payload, with respect to the last valid quadword of the prepend. |
| Skip | Allows software to allocated a TBUF element and then not transmit any data from it. 0: transmit data according to other fields of the Control Word 1: free the element without transmitting any data |
| ERR | Error bit. If this bit is set, the transmit logic will force bad parity on the entire cell. This is useful for testing only; this bit should never be set during normal operation. |
| SOP | Indicates if the element is the start of a packet. This field is ignored by hardware in UTOPIA mode, as each element must contain a complete cell. |
| EOP | Indicates if the element is the end of a packet. This field is ignored by hardware in UTOPIA mode, as each element must contain a complete cell. |
| MPHY-32 id | MPHY-32 Channel Identifier. This bit, when set, is used to indicate that the mpacket is intended for the MPHY-32 port (port 0). This bit is used by the hardware to differentiate between channels 0x00 to 0x1f of the MPHY-32 channel, and SPHY channels 0x1, 0x2, and 0x3. It is intended for use in x16 MPHY-32 mode; in x32 MPHY-32 mode, it is a don't care. In any MPHY-4 mode, it is a don't care |
| Channel | In MPHY mode other than MPHY4, the port number to which the data is directed. In SPHY or MPHY4 mode, this field has no effect. |

8.6.2.8 POS-PHY Transmit Control Word Format

In POS-PHY mode, a packet is divided into a number of mpackets, where the mpacket size is equivalent to the TBUF element size. The Control Word format is in [Table 117](#):

Table 117. POS-PHY Transmit Control Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------------|---|---|---|---|---|---|---|----------------|---|---|----------------|---|---|---|----------------|---|---|-----|------|-----|-----|-----|------------|-----|---|---|---------|---|---|---|--|
| 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 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 | ERR | SOP | EOP | MPHY-32 id | res | | | Channel | | | | |
| 6 | 6 | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 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 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | |
| Res | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

The definitions of the fields are in [Table 118](#):

Table 118. POS-PHY Transmit Control Word Field Definitions

| Field | Definition |
|----------------|--|
| Payload Length | Indicates the number of bytes in the payload, from 1 to 256 bytes, in the element. The value of 0x00 means 256 bytes. The sum of Prepend Length and Payload Length will be sent, and must be an integral multiple of the bus width (in bytes), except if EOP = 1. |
| Prepend Offset | Indicates the first valid byte of the prepend, from 0 to 7 bytes. |
| Prepend Length | Indicates the number of bytes of the prepend from 0 to 31 bytes. |
| Payload Offset | Indicates the first valid byte of the payload, with respect to the last valid quadword of the prepend. |
| Skip | Allows software to allocated a TBUF element and then not transmit any data from it. 0: transmit data according to other fields of the Control Word 1: free the element without transmitting any data |
| ERR | Error bit. If this bit is set, the transmit logic will force the TXERR signal to be asserted during the last word of the packet, when TXEOF is asserted. This bit is only valid if EOP is set, otherwise it is ignored. This is useful for testing only; this bit should never be set during normal operation. |
| SOP | Indicates if the element is the start of a packet. |
| EOP | Indicates if the element is the end of a packet. |
| MPHY-32 id | MPHY-32 Channel Identifier. This bit, when set, is used to indicate that the mpacket is intended for the MPHY-32 port (port 0). This bit is used by the hardware to differentiate between channels 0x00 to 0x1f of the MPHY-32 channel, and SPHY channels 0x1, 0x2, and 0x3. It is intended for use in x16 MPHY-32 mode; in x32 MPHY-32 mode, it is a don't care. In any MPHY-4 mode, it is a don't care |
| Channel | In POS-PHY MPHY mode other than MPHY4, the port number to which the data is directed. The port number will be sent in-band by the transmit logic. In SPHY or MPHY4 mode, this field has no effect. |

8.6.2.9 CSIX Mode

In CSIX Mode, the transmit control logic manages the TBUF as two separate segments, one for Data traffic and one for Control traffic. The lowest 8 elements are for Control traffic, and the highest 24 elements are for Data traffic.

Payload information is put into the Payload area of the element, and Base and Extension Header information is put into the Header Control area of the element.

Data is stored in big-endian mode. The most significant byte of each longword is transmitted first.

The status word contains the information in [Table 119](#).

Table 119. CSIX Transmit Control Word Format

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------------------|---|---|---|---|---|---|---|---|---|----------------|---|----------------|---|---|---|----------------|---|---|---|-----|------|-----|----|---|----------|---|---|---|------|---|---|--|
| 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 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 | reserved | | | | Type | | | |
| 6 | 6 | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 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 | 3 | 2 | |
| Extension Header | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

The definitions of the fields are in [Table 120](#).

Table 120. CSIX Transmit Control Word Field Definitions

| Field | Definition |
|------------------|--|
| Payload Length | Indicates the number of bytes in the payload, from 1 to 256 bytes, 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 Length field. |
| Prepend Offset | Indicates the first valid byte of the prepend, from 0 to 7 bytes. |
| Prepend Length | Indicates the number of bytes of the prepend from 0 to 31 bytes. |
| Payload Offset | Indicates the first valid byte of the payload, with respect to the last valid quadword of the prepend. |
| Skip | Allows software to allocate a TBUF element and then not transmit any data from it. <ul style="list-style-type: none"> 0—transmit data according to other fields of the Control Word 1—free the element without transmitting any data |
| CR | CR (CSIX Reserved) bit to put into the CSIX Base Header. |
| P | P (Private) bit to put into the CSIX Base Header. |
| Type | Type Field to put into the CSIX Base Header. Idle type is <i>not</i> legal here. |
| Extension Header | The Extension Header to be sent with the CFrame. For flow control CFrames this field is not used by the hardware because flow control CFrames do not have an extension header. |

8.6.3 Transmit Operation Summary

During transmit processing, data to be transmitted is placed into the TBUF under ME control. The ME 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.

MEs 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 bit per element, which marks the element as ready to be transmitted. MEs move all data into the element, by either or both the `msf[write]` and `dram[tbuf_wr]` instructions to the TBUF. MEs also write the element Transmit Control Word (TCW) using `msf[write]` with information about the element. The order of the TCW is:

1. `s_xfer reg n` contains TCW[31:0]
2. `s_xfer reg n+1` contains TCW[63:32]

When all the data movement is complete the ME sets the element valid bit. The complete sequence is:

1. Move data into TBUF by either or both of `msf[write]` and `dram[tbuf_wr]` instruction to the TBUF.
2. Wait for (1.) to complete.
3. Write the Transmit Control Word by `msf[write]` at the `TBUF_Element_Control_V_#` 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 ME does not get an Event Signal, to prevent use of the corrupt data. The error is recorded in the DRAM controller, and will interrupt 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. Note that the transmit pipeline will be stalled since all TBUF elements must be transmitted in order.

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.

The remainder of [Section 8.6.3](#) describes the detailed transmit flow for each mode of operation.

8.6.3.1 UTOPIA SPHY Mode

The TX thread decides which port a cell is to be sent out on. Because the TBUF entries are transmitted sequentially by the hardware, the thread also uses the TBUF entries sequentially. The thread knows if the TBUF entry is free because it keeps track of how many entries it has used and, by reading the `Tx_Sequence_{0...3}` CSR, it knows how many have been transmitted. It then writes the cell into the TBUF entry, and writes the Transmit Control Word into the control field. When the necessary valid bits have been set, the transmit protocol hardware will send this cell out through the transmit pins. When transmission of the cell has been completed, the transmit protocol hardware will update the appropriate `Tx_Sequence_{0...3}` CSR and clear the three associated bits (Data Valid, Control Valid, and Transmit Valid).

If the PHY's transmit FIFO is full, then transmission stalls until space is freed up.

8.6.3.2 UTOPIA MPHY-4 Mode

UTOPIA MPHY-4 mode is very similar to UTOPIA SPHY mode, except that TBUF is always partitioned into four independent segments; each port has its own segment. The same process used for UTOPIA SPHY is used here. If hardware is configured for Direct Status Indication, then it uses `TXFA[3:0]` for flow control; if hardware is configured for polling, then it uses `TXADDR[4:0]` to poll the PHY's transmit FIFOs and looks for the result on the `TXPFA` input signal. In either case, polling is taken care of automatically by hardware.

8.6.3.3 UTOPIA MPHY-32 Mode

UTOPIA MPHY-32 (supporting up to 32 ports in 1x32b mode and up to 31 ports one 16b bus) mode requires that software do some amount of software-driven status polling as well as transmit scheduling in order to minimize hardware requirements and to provide a solution which is scalable to an even larger number of ports in the future (i.e., 48 ports). Hardware will provide some hints to help software; this is described in detail below.

In UTOPIA MPHY-32 mode, TBUF functions as a single large segment; all traffic for the MPHY ports funnels into a single, large TBUF. (Partitioning TBUF into 31 or 32 equal sized segments would potentially short change ports which carry higher bandwidth. Also, that solution would not scale well to 48 ports.)

After the PHY has been initialized and enabled, the transmit polling FSM will update the Tx_MPHY_Status[Tx_Status] flags. Initially, the Tx_Status flags are all zero. The TX thread must poll the flags and wait for them to become set before initiating transmission to that channel. If the Tx_Status flag is set, that means that the transmit FIFO in the PHY is able to accept at least one cell. The TX thread pushes out one cell to each port which is able to accept. Note that only one cell may be pushed out to a port at any given time; the TXFA[3:0] status signal only says that the PHY can accept at least one more cell; it does not provide any lookahead beyond one cell.

When a transmit control word is written, thereby initiating the transmission of a cell to a certain port, the appropriate bit in Tx_MPHY_Status[Tx_Pending] corresponding to the port is set. This indicates two things:

- A cell transmission is in progress for that given port.
- The Tx_Status flag for that port is now stale and should be ignored.

When the cell appears on the TX pins, the transmit hardware will make a request to clear the Tx_Pending flag. However, the Tx_Pending does not actually get cleared until the polling FSM has updated the status for that port. That means that when Tx_Pending makes the transition back to 0, Tx_Status is guaranteed to be up to date again.

If Tx_Pending is not equal to 0, meaning that cell transfer is still pending, then the Tx_Status bit for that port must be ignored, as it is impossible to tell if it is stale or not.

Ideally, software should batch cell transfers for all the ports, rather than doing multiple CSR reads and writes for each cell transfer.

By avoiding pushing a cell out to port unless it is known that the port can accept the cell, this avoids head-of-line blocking. If a cell is written to a port which is already full, the cell will not be lost, as hardware performs its own polling. It may, however, result in sub-optimal performance due to head-of-line blocking. The Tx_MPHY_Status CSR is provided to give software some visibility and to allow software the option of doing transmit scheduling.

Alternatively, since TBUF should drain at a known rate, the TX thread can use a timer, based on the number of cells transmitted, to determine when to perform the Tx_MPHY_Status read.

8.6.3.4 POS-PHY SPHY Mode

The TX thread decides which port a packet is to be sent out on. It is up to software to break up the packet into a number of mpackets, where the mpacket size is equal to the TBUF entry size. Because the TBUF entries are transmitted sequentially by the hardware, the thread also uses the TBUF entries sequentially. The thread knows if the TBUF entry is free because it keeps track of how many entries it has used and, by reading the Tx_Sequence_{0...3} CSR, it knows how many

have been transmitted. It then writes the first mpacket into the TBUF data element, constructs the Transmit Control Word and writes it into the TBUF control element, making sure to set the SOP bit. It writes the Transmit Control Word by `msf[write]` at the `TBUF_Element_Control_V_#` address. Using this address sets the Transmit Valid bit. It continues this process until the entire packet has been pushed into TBUF. For the last mpacket, the TX thread must set the EOP bit, and for “middle” mpackets neither SOP nor EOP bits are set. In POS-PHY SPHY mode all mpackets that make up a packet must be transmitted contiguously for a given channel.

The transmit protocol hardware processes one mpacket at a time, send out the data, generating the correct parity, and generating the necessary control signals (TXENB, TXSOF, TXEOF, TXERR) and managing flow control for each direction. If the mpacket arrival rate is slower than the POS-PHY interface rate it deasserts TXENB; if the PHY’s transmit FIFO runs out of space it will deassert TXFA[3:0]; hardware will detect this and will stop transmitting until there is more space available (indicated by TXFA[3:0] being reasserted).

8.6.3.5 POS-PHY MPHY-4 Mode

POS-PHY MPHY-4 mode is similar to POS-PHY SPHY mode, except that TBUF is always partitioned into four independent segments; each port has its own segment. The same process used for POS-PHY SPHY is used here. If hardware is configured for Direct Status Indication, then it uses TXFA_x for flow control; if hardware is configured for polling, then it uses TXADDR[4:0] to poll the PHY’s transmit FIFOs and looks for the result on the TXPFA input signal. In either case, polling is taken care of automatically by hardware. Address is sent in-band.

8.6.3.6 POS-PHY MPHY-32 Mode

POS-PHY MPHY-32 (supporting up to 32 ports in 1x32b mode or supporting up to 31 ports on one 16b bus) mode requires that software do some amount of software-driven status polling as well as transmit scheduling in order to minimize hardware requirements and to provide a solution which is scalable to an even larger number of ports in the future (i.e., 48 ports). Hardware will provide some hints to help software; this is described in detail below.

In POS-PHY MPHY-32 mode, TBUF functions as a single large segment; all traffic for the MPHY ports funnels into a single, large TBUF. (Partitioning TBUF into 31 or 32 equal sized segments would potentially short change ports which carry higher bandwidth. Also, that solution would not scale well to 48 ports.)

After the PHY has been initialized and enabled, the transmit polling FSM will update the `Tx_MPHY_Status[Tx_Status]` flags. Initially, the `Tx_Status` flags are all zero. The TX thread must poll the flags and wait for them to become set before initiating transmission to that channel; the flags indicate which transmit FIFOs in the PHY is able to accept data. In POS-PHY mode, the transmit FIFO in the PHY has a programmable threshold. If the PHY’s transmit FIFO contains less data than the threshold, then TXPFA will be asserted in response to a status poll. If the transmit FIFO contains more data than the threshold, then TXPFA will be deasserted in response to a status poll. Since software knows both the size of the FIFO and the threshold, software also knows that if TXPFA is deasserted, then the FIFO should be able to accept at least “n” mpackets of data, where “n” is the difference between the FIFO size and the threshold.

The TX thread pushes out up to n mpackets of data to each port which is able to accept data. Hardware maintains an mpacket-based counter for each port. When a transmit control word is written, thereby initiating the transmission of a mpacket to a certain port, the counter for that port is incremented; whenever an mpacket has been completely sent to the PHY, the counter is

decremented. The summary bit for each port is visible in Tx_MPHY_Status[Tx_Pending]; if the counter is zero, indicating no pending transmit activity for that port, then Tx_Pending is 0; if the counter is non-zero, then Tx_Pending is 1.

Before Tx_Pending makes the transition back to 0, the polling FSM will update Tx_Status. When Tx_Pending = 0, this means that Tx_Status is guaranteed to be updated with the latest FIFO status.

After all transmit data is pushed out, the TX thread waits, then reads Tx_MPHY_Status.

1. If Tx_Pending[x] = 0, then Tx_Status[x] contains up to date status. If Tx_Status[x] = 1, it is OK to send another batch of “n” mpackets to that port. If Tx_Status[x] = 0, that port’s FIFO does not have sufficient space to accept a batch of “n” mpackets and the TX scheduler has to wait and poll again later. The value “n” is the difference between the FIFO size and the threshold.
2. If Tx_Pending[x] = 1, then Tx_Status[x] is indeterminate. It is impossible to tell if the port FIFO can accept any more data because it is unknown how many mpackets are in flight and when the last time Tx_Status has been updated. The TX schedule has to wait and poll again later.

By following the above rules, the TX thread may then push out more transmit data to ports which are able to accept more data. By avoiding pushing data out to port unless it is known that the port can accept the data helps avoid unnecessary head-of-line blocking.

If data is written to a port which is full or near full, the data will not be lost as long as the high watermarks in the PHY are correctly set to avoid overflow. It may, however, result in sub-optimal performance due to head-of-line blocking. The Tx_MPHY_Status CSR is provided to give software some visibility and to allow software the option of doing transmit scheduling.

POS-PHY Level 3 MPHY-4 and MPHY-32 modes support “packet level” transfers only, in that TXPFA or TXFA asserted high means that the PHY can accept some fixed amount of transmit data. In polled mode, TXPFA (in MPHY-4 or MPHY-32 polled mode) or TXFA[3:0] (in MPHY-4 direct status mode) is used to decide if the PHY’s transmit FIFO can accept more data. TXSFA is not used and should be tied low. Arbitration is mpacket based; once the arbiter selects a port, IXP2400 will send out one entire mpacket to the PHY (may be less than one full mpacket if TXEOF is asserted). This means that the high watermarks in the PHY that control TXPFA or TXFA[3:0] must be set low enough so that if TXPFA/TXFA is asserted, the slave is guaranteed to be able to accept at least one entire mpacket. In order to account for various latencies, the high watermark on the slave’s transmit FIFO should be configured so that TXPFA or TXFA[3:0] are deasserted if the amount of remaining space in the slave’s transmit FIFO is less than (TBUF element size + 8 clock cycles) of data, or (TBUF element size + 32 bytes).

In POS-PHY Level 2 MPHY-32 mode, two modes of transfer are supported: non-TXSFA and TXSFA.

In non-TXSFA mode, TXSFA is ignored and the behavior is the same as described in the previous paragraph for POS-PHY Level 3 MPHY mode. TXSFA should be tied high, and the same rule regarding high watermark settings applies. In order to avoid FIFO overflow, the high watermark on the slave’s transmit FIFO should be configured so that TXPFA or TXFA[3:0] are deasserted if the amount of remaining space in the slave’s transmit FIFO is less than (TBUF element size + 8 clock cycles) of data, or (TBUF element size + 16 bytes).

In TXSFA mode, TXSFA is used as described in the POS-PHY Level 2 spec. Here are the rules describing IXP2400’s behavior:

1. It is assumed that both TXPFA and TXSFA are controlled by the same high watermark level.

2. The decision to grant a port is based purely on TXPFA.
3. If there is data ready to be transmitted to a given port, and if the TXPFA for that port is high, the port will be selected. Once the port is selected, IXP2400 will ignore TXSFA for the first four clock cycles of data transfer.
4. Once IXP2400 sees TXSFA deasserted, it will send out up to four more clock cycles of data. This means that the high watermark in the PHY must be set lower than eight clock cycles, or 16 bytes, from the end of the FIFO.
5. Once data transferred is stopped, IXP2400 will not restart data transfer until it polls TXPFA high again for that port. IXP2400 will not switch to another port until the data transfer from the current port is completed.

The advantage of TXSFA mode over non-TXSFA mode is that the high watermark can be set lower.

In order to achieve maximum performance it is important to push out the maximum amount of data in the loop and overlap status updates with transmit.

8.6.3.7 CSIX Mode

Transmit control logic sends valid elements on the transmit pins in element order. Each element sends a single CFrame—the Base Header is sent first using the information in the Transmit Control Word for the element. The Ready Field placed into the Base Header is taken from TM_CRDY and TM_DRDY bits generated by the egress processor (described in greater detail in [Section 8.7, “CBus Interface” on page 8-287](#)). Next the Extension Header is sent, using the information in the Control for the element. Finally the Payload is sent. The Payload Length Field determines how many CWords of Payload are sent. Both Horizontal Parity and Vertical Parity are transmitted, as described below.

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 ignored by hardware for flow control CFrames.

Table 121. Transmit Control Word to CSIX Header Mapping

| CSIX Header Field | Derivation |
|-------------------|---|
| DRDY | FC_Ingress_Status[TM_DRDY]; received via RXCSR pin from egress processor |
| CRDY | FC_Ingress_Status[TM_CRDY]; received via RXCSR pin from egress processor |
| Type | Type field from Transmit Control Word |
| CR | CR bit from Transmit Control Word |
| P | P bit from Transmit Control Word |
| Payload Length | Prepend Length + Payload_Length |
| Extension Header | Extension Header field from Transmit Control Word. For flow control CFrames there is no extension header so the Extension Header field is not used. |

TBUF is divided into two segments, one for data and one for control. (Note: there is no hardware restriction on what type of CFrames can be placed into either segment. For example, it is possible to put control CFrames in the data segment and vice versa.)

Control elements and Data elements share use of the transmit pins. Each will alternately transmit a valid element in a round-robin fashion.

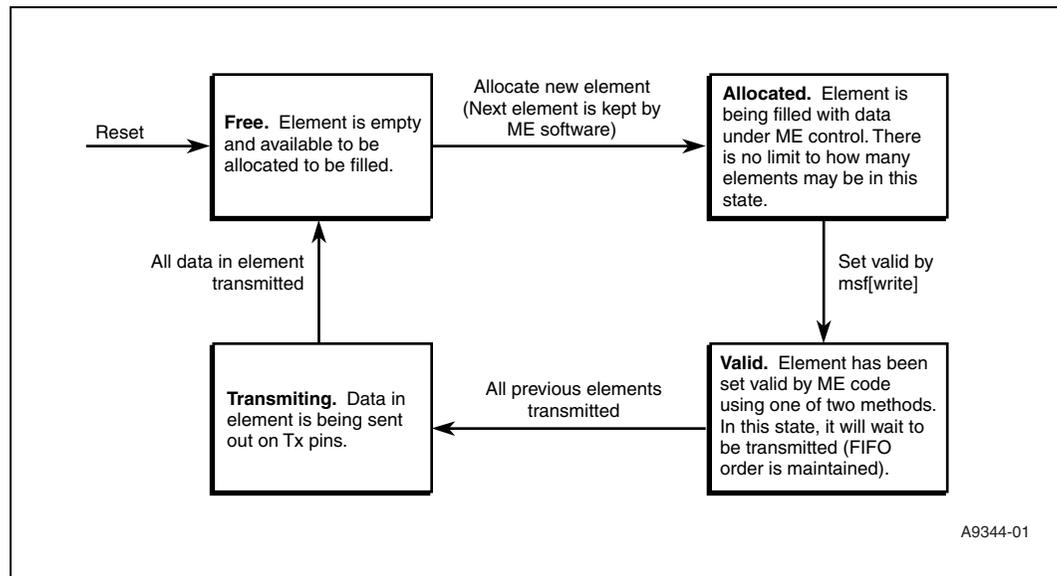
If the next sequential element is not valid when its turn comes up, then transmit logic will alternate sending Idle CFrames with Dead Cycles; it will continue to do so until a valid element is ready. Idle CFrames get the value for the Ready Field from the TM_CRDY and TM_DRDY bits, and the Payload Length is set to 0.

After an element has been sent on the transmit pins, all valid bits for that element are cleared, which marks the element as available to be re-used.

8.6.3.8 Transmit Summary

The states that a TBUF element goes through are shown in [Figure 98](#).

Figure 98. TBUF State Transition Diagram



8.6.4 Transmit Flow Control Status

Transmission of TBUF elements is controlled by MSF_Tx_Control[Tx_En] bit for a given TBUF segment. Software can allocate and fill TBUF elements, and then (temporarily) disable them from being transmitted by setting bits in MSF_Tx_Control[Tx_En] bit. Note that MSF_Tx_Control[Tx_En] bit does not invalidate any elements, nor prevent allocation of elements.

When MSF_Tx_Control[Tx_En] bit changes to disable transmission, any element whose transmission is in-progress will be completed.

Communication of flow control information between IXP2400 and other external devices is handled in hardware through the CBus interface, and is based on MSF_Tx_Control[Transmit_Mode], and is only applicable to CSIX mode. Link level and fabric level flow control information is sent by the egress processor to the ingress processor using CBus. This is described in further detail in [Section 8.7](#).

8.6.5 Parity

8.6.5.1 UTOPIA Mode

Single bit odd parity, as selected by the Tx_UP_Control_{0...3}[Parity_Mode] bits, is always generated for transmit data. If parity is not needed, then the TxPrty pin should be left unconnected.

8.6.5.2 POS-PHY Mode

Single bit odd or even parity, as selected by the Tx_UP_Control_{0...3}[Parity_Mode] bits, is always generated for transmit data. If parity is not needed, then the TxPrty pin should be left unconnected.

8.6.5.3 CSIX Mode

8.6.5.3.1 Horizontal Parity

The transmit logic computes Horizontal Parity for each transmitted Cword, and transmits it on TxPar.

8.6.5.3.2 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. The 16 bits of vertical parity are formed on 32 bits of transmitted data by treating the data as words; that is, 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 two bytes of Payload.

8.6.6 RBUF and TBUF Summary

Table 122 compares RBUF and TBUF operations.

Table 122. RBUF and TBUF Summary

| Operation | RBUF | TBUF |
|---|--|--|
| Allocate an element | <p>UTOPIA Hardware allocates an element upon receipt of a cell. Any available element may be allocated, however, elements are guaranteed to be handed to threads in the order they arrive.</p> <p>POS-PHY Hardware allocates an element upon the receipt of a burst. Any available may be allocated, however, elements are guaranteed to be handed to threads in the order they arrive.</p> <p>CSIX Hardware allocates an element upon receipt of RxSof asserted. Any available element may be allocated, however, elements are guaranteed to be handed to threads in the order they arrive. Any element can be allocated to Control or Data CFrame.</p> | <p>ME allocates an element. Because the elements are transmitted in FIFO order (within each TBUF partition), the ME can keep the number of the next element in software.</p> |
| Fill an element | <p>UTOPIA Hardware fills the element with the entire cell.</p> <p>POS-PHY Hardware fills the element with data.</p> <p>CSIX Hardware fills the element with Payload.</p> | <p>Microcode fills the element from DRAM using <code>dram[tbuf_wr...]</code> instruction and from ME registers using <code>msf[write]</code> instruction.</p> |
| Set an element valid | <p>UTOPIA Set valid by hardware then the entire cell has been received.</p> <p>POS-PHY Set valid by hardware when either the element has been filled or RXEOF is asserted, whichever comes first.</p> <p>CSIX 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 <code>Transmit_Control_Word_V_#CSR</code>, where # is the element number.</p> |
| Remove data from an element | <p>Microcode moves data from the element to DRAM using <code>dram[rbuf_rd...]</code> instruction and to ME registers using <code>msf[read]</code> instruction.</p> | <p>Hardware transmits information from the element to the Tx pins. Transmission of elements is in FIFO order; that is an element will be transmitted only when all preceding elements have been transmitted.</p> |
| Return an element to its Free List | <p>Microcode writes to <code>Rx_Element_Done</code> with the number of the element to free.</p> | <p>Hardware returns the element when its information has been transmitted.</p> |

8.7 CBus Interface

Note: Per the CSIX specification, the terms “egress” and “ingress” are with respect to the switch fabric. So the egress processor handles traffic received from the switch fabric and the ingress processor handles traffic sent to the switch fabric.

CBus has two major modes of operation: full duplex and simplex.

In **full duplex mode**, the CBus interface is used to communicate link level and fabric level flow control information from the egress (receive) processor to the ingress (transmit) processor. Full duplex mode is described in detail in [Section 8.7.2](#).

In **simplex mode**, the CBus interface is used to communicate link level and fabric level flow control information directly with the switch fabric. Simplex mode is described in detail in [Section 8.7.3](#).

Note: Per the CSIX specification, the terms *egress* and *ingress* are with respect to the switch fabric. So the egress processor handles traffic received from the switch fabric and the ingress processor handles traffic sent to the switch fabric.

8.7.1 CBus Signals

8.7.1.1 TXCSOF/TXCDATA/TXCPAR and RXCSOF/RXCDATA/RXCPAR

The CBus data width can be programmed to either 4-bits TXCDATA[3:0] or 8 bits TXCDATA[7:0] as specified in the MSF_Rx_Control and MSF_Tx_Control registers (see the *Intel® IXP2400/IXP2800 Programmer's Reference Manual* for details). The mapping of the exact pins used for these signals is shown in [Section 8.4](#).

CFrames are transmitted from the egress processor using TXCSOF, TXCDATA[3:0] (or TXCDATA[7:0] depending on the mode), and TXCPAR and received on the ingress processor using RXCSOF, RXCDATA[3:0] (or RXCDATA[7:0] depending on the mode), and RXCPAR. These signals can be thought of as a *thin* version of CSIX-L1. The protocol is identical except that data is transferred four or eight bits at a time rather than 32 bits at a time. Since CBus runs at the same clock rate as the CSIX interface, this implies that amount of traffic that is forwarded or generated as a result of CSIX traffic cannot be more than an eighth or a quarter of the total CSIX traffic depending on the mode selected.

TXCSOF is used to mark the start of frame. In the 4-bit mode, data is sent one nibble at a time, and is transferred in big-endian order:

1. [31:28]
2. [27:24]
3. [23:20]
4. [19:16]
5. [15:12]
6. [11:8]
7. [7:4]
8. [3:0]

TXCPAR is generated for every 32 bits and is valid only when the final nibble of the CWord (bits [3:0]) are valid on the bus.

In the 8-bit mode, data is sent one byte at a time, and is transferred in the big-endian order:

1. [31:24]
2. [23:16]
3. [15:8]
4. [7:0]

TXCPAR is generated for every 32 bits and is valid only when the final byte of the CWord (bits [7:0]) are valid on the bus.

8.7.1.2 TXCSRB and RXCSRB

TXCSRB and RXCSRB are used only in full duplex mode.

The egress processor uses TXCSRB to transmit the Serialized Ready Bits. This consists of five framing bits and the SF_xRDY and TM_xRDY bits. The bits are transmitted in the following order:

1. 00001 (framing)
2. SF_CRDY
3. SF_DRDY
4. TM_CRDY
5. TM_DRDY

The Serialized Ready Bits are received in the ingress processor using the RXCSRB input pin.

8.7.1.3 TXCFC and RXCFC

TXCFC and RXCFC are used only in full duplex mode.

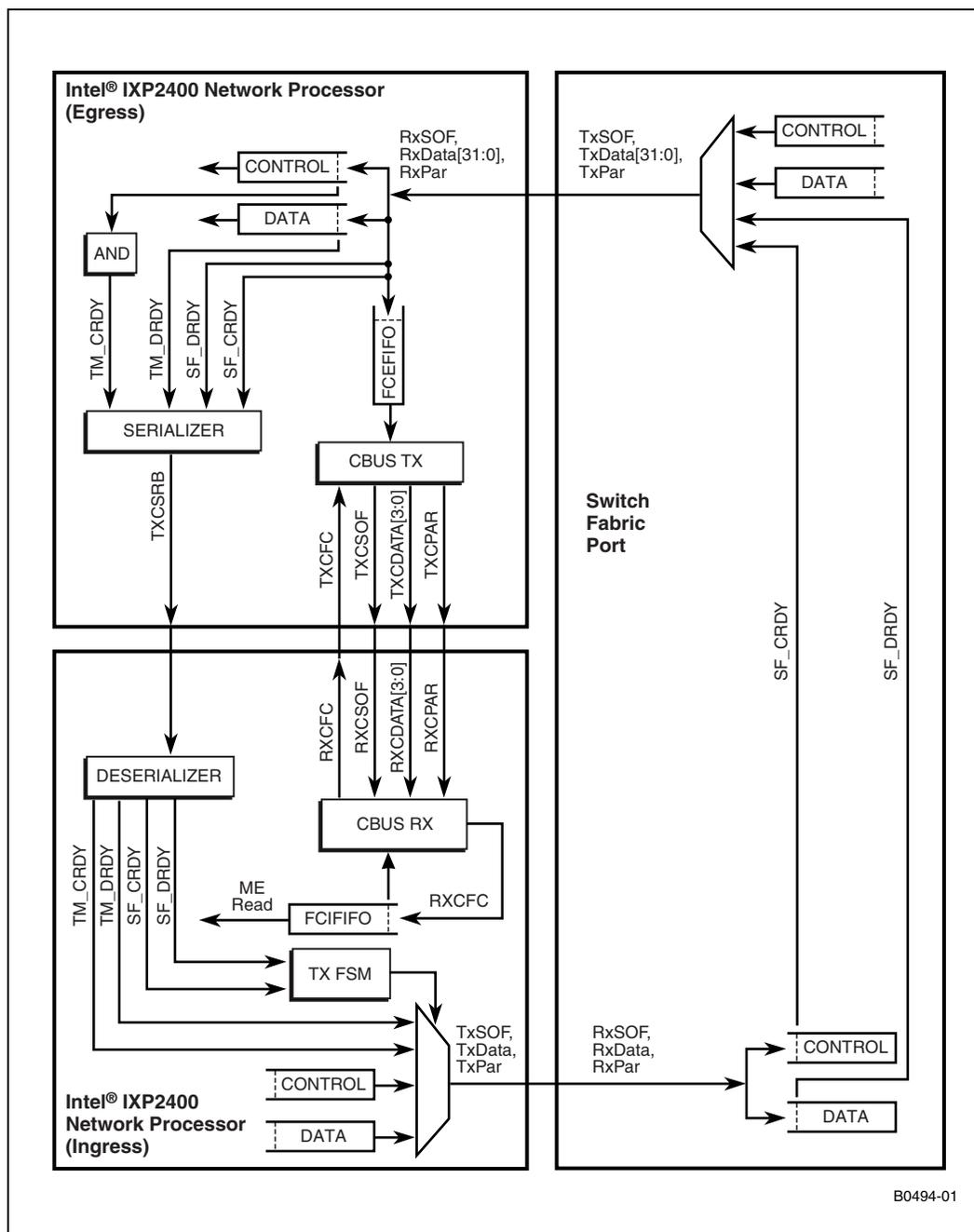
RXCFC is a flow control output signal that is used by the ingress processor to indicate that its FCIFIFO has exceeded a high watermark. It is connected to the TXCFC input pin of the egress processor.

8.7.2 Full Duplex Mode

In full duplex mode, flow control information is communicated from the egress processor to the ingress processor using CBus.

Figure 99 contains a block diagram which shows the CBus interconnections and usage.

Figure 99. Full Duplex Mode Block Diagram



8.7.2.1 Link Level

Link level flow control information is transmitted serially on the TXCSRIB output pin on the egress processor and received serially on the RXCSRIB input pin on the ingress processor. Four bits of information are sent: SF_CRDY, SF_DRDY, TM_CRDY, and TM_DRDY.

The switch fabric supplies link level flow control information in the base header of each CFrame that it sends to the egress processor. 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). These are referred to in this document as the SF_CRDY and SF_DRDY bits, because they originate from the switch fabric. The SF_*RDY bits reflect the state of the switch fabric's ingress FIFOs. If one of the bits is deasserted by the switch fabric, that means it is running out of space in its control or data ingress FIFO.

In addition to the SF_CRDY and SF_DRDY bits, the egress processor also provides TM_CRDY and TM_DRDY bits to the ingress processor. (TM stands for Traffic Manager; IXP2400 is the traffic manager.) These bits reflect the state of the control and data receive FIFOs in the egress processor and are sent to the ingress processor to be forwarded to the switch fabric in the base header of all outgoing CFrames. TM_CRDY and TM_DRDY are deasserted whenever the egress processor is running out of space in its receive FIFOs. These high watermarks are set using HWM_Control[RBUF_C_HWM] and HWM_Control[RBUF_D_HWM]. When the egress processor detects a horizontal or vertical parity error in any incoming CFrame, it deasserts SF_CRDY and SF_DRDY bits.

On the egress processor, the ready bits are readable via the FC_Egress_Status register. [Table 123](#) shows how the ready bits are derived in the egress processor.

Table 123. Egress Processor Ready Bit Handling

| Ready Bit | Derivation |
|-----------|---|
| SF_CRDY | READY[0] of all incoming base headers. This bit is visible to software as FC_Egress_Status[SF_CRDY]. |
| SF_DRDY | READY[1] of all incoming base headers. This bit is visible to software as FC_Egress_Status[SF_DRDY]. |
| TM_CRDY | RBUF control partition above/below high watermark set by HWM_Control[RBUF_C_HWM] or FCEFIFO above/below high watermark set by HWM_Control[FCEFIFO_HWM]. This bit is visible to software as FC_Egress_Status[TM_CRDY]. |
| TM_DRDY | RBUF data partition above/below high watermark set by HWM_Control[RBUF_D_HWM]. This bit is visible to software as FC_Egress_Status[TM_DRDY]. |

[Table 124](#) summarizes how the serialized ready bits should be handled by the ingress processor. On the ingress processor, these bits are readable via the FC_Ingress_Status register.

Table 124. Ingress Processor Ready Bit Handling

| Bit Value Ready Bit | = 0 | = 1 |
|---------------------|---|--|
| SF_CRDY | Stop sending control CFrames to the switch fabric. | OK to send control CFrames to the switch fabric. |
| | This bit is visible to software as FC_Ingress_Status[SF_CRDY]. | |
| SF_DRDY | Stop sending data CFrames to the switch fabric. | OK to send data CFrames to the switch fabric. |
| | This bit is visible to software as FC_Ingress_Status[SF_DRDY]. | |
| TM_CRDY | Place this bit in the READY[0] field of all outgoing base headers. This bit is visible to software as FC_Ingress_Status[TM_CRDY]. | |
| TM_DRDY | Place this bit in the READY[1] field of all outgoing base headers. This bit is visible to software as FC_Ingress_Status[TM_DRDY]. | |

On the egress processor, software has some control over what gets transmitted on the TXCSR_B output pin by using CBus_Control[TXCSR_B_En] and CBus_Control[TXCSR_B_Force]. If TXCSR_B_En = 0, then the SF_*RDY and TM_*RDY bits will be sent on the TXCSR_B pin; if TXCSR_B_En = 1, then the values specified in TXCSR_B_Force are transmitted. This can be used for debug, test, and to allow software to override hardware.

On the ingress processor, software also has some control over what gets used by hardware by using CBus_Control[RXCSR_B_En] and CBus_Control[RXCSR_B_Force]. If RXCSR_B_En = 0, then the values received on the RXCSR_B input pin are used; if RXCSR_B_En = 1, then the values specified in RXCSR_B_Force are used instead. This can be used for debug, test, and to allow software to override hardware.

8.7.2.2 Buffering and Link Level Flow Control Latency

In the full duplex connection, the latency for the switch fabric to respond to a change in the TM_CRDY/TM_DRDY is large. The worst case latency for the total time elapsed from the time the egress processor de-asserts the TM_xRDY bits to when the switch fabric stops sending traffic to the egress processor is estimated as shown in the following steps:

1. latency for sending TM_xRDY across the c-bus (pipeline delay, synchronization delays etc) from the egress processor to the ingress processor is 18 cycles.
2. The ingress processor will append the incoming TM_xRDY bits to the next cframe that will be sent out. Assuming that the ingress processor just missed a maximum size cframe that is being sent out - 2B base header + 4B extension header + 256B payload + 2B vertical parity = 264B = 66 cycles, the TM_xRDY bit will be delayed by 66 cycles before it is sent across the CSIX interface.
3. If the ingress processor appends the TM_xRDY bit on a maximum size cframe, then assuming that a parity error occurs on the READY[0/1] bit of the base header, it will be detected only when VP parity is computed for the entire cframe, adding another 66 cycles.
4. According to the CSIX spec. the fabric has 32 cycles to react to change in the TM_xRDY bits.
5. Assume that just before the fabric decided to stop, a maximum size cframe slipped out, adding another 66 cycles.

Thus the total latency for the link level flow control to respond can be 248 cycles. Assume that during this interval, the switch fabric transmits a series of minimum size cframes (2B base header + 4B extension header + 1B payload + 3B padding + 2B vertical parity = 12B or 3 cycles) followed by the last maximum size cframe. Thus the total number of cframes received will be 62 (61 cframes in the first 182 cycles + 1 cframe in the next 66 cycles). Assuming that the RBUF element size in the egress processor is configured to be 256B, the total number of elements is 32. In other words, the egress processor can buffer only up to 32 cframes. Thus, even if the high watermark control, HWM_Control[RBUF_x_HWM] is set to 0, buffer overflow may occur.

In order to overcome this condition, an additional level of buffering is provided on the receive interface of each processor. This buffer, comprising of Rx Data Fifo and Rx Status FIFO, is not visible to software. The Rx Data Fifo is sized to be 256 c-words (256 32-bit entries) to buffer the 248 cycles worth of data. The Rx Status FIFO is 64 entries, enough to buffer the status of the 62 received cframes.

8.7.2.3 Fabric Level

The CBus interface on the egress processor contains a 256 x 32 FIFO called FCEFIFO. The CSIX_Type_Map CSR allows any CFrame type to be routed into FCEFIFO, although typically only flow control CFrames are put into FCEFIFO. The entire CFrame is placed into FCEFIFO. All CFrames in FCEFIFO are sent out of the egress processor on the TXCSOF, TXCDATA[1:0], and TXCPAR output pins.

In order to minimize latency, CFrames are forwarded in a cut-through manner. CFrames which get routed to FCEFIFO never become visible to software running on the egress processor. In order to minimize latency, CFrames are forwarded using a cut-through approach. If there is no data in FCEFIFO then the egress processor sends Idle CFrames.

FCEFIFO has its own high watermark, set using HWM_Control[FCEFIFO_HWM]. If the high watermark is exceeded, this will cause TM_CRDY to be deasserted.

Flow control CFrames are received by the ingress processor on the RXCDATA, RXCPAR, and RXCSOF pins. They are deposited into the FCIFIFO. FCIFIFO is 256 x 32. The ingress processor is responsible for checking horizontal and vertical parity and length error (premature RXCSOF) for each incoming CFrame. If an error is detected, the CFrame is dropped and MSF_Interrupt_Status[FCIFIFO_Error] is set.

The FCIFIFO has two signals which are used to signal a thread: not-empty and nearly-full. These two signals are connected to the STATE inputs of every Microengine and can be tested using the BR_STATE instruction:

- **FCI_Not_Empty:** if asserted, this indicates that there is at least one CWord in FCIFIFO. This signal stays asserted until all CWords have been read. This signal is not asserted until there is a complete, valid, error-free CFrame in FCIFIFO. As the CFrame is dequeued by the ME handler thread, this signal stays asserted until all CWords have been removed, including subsequent CFrames.
- **FCI_Near_Full:** if asserted, this dictates that FCIFIFO is above the high watermark set in HWM_Control[FCIFIFO_Int_HWM].

The not-empty signal is asserted if and only if an entire CFrame has been received without error into the FCIFIFO. The thread that has been assigned to handle FCIFIFO is woken up, then must read the CFrame, 32 bits at a time, from the FCIFIFO by issuing `msf[read]` to the FCIFIFO CSR. (Burst read of up to sixteen words is allowed.) The FCIFIFO handler thread must examine the base header to determine how long the CFrame and perform the necessary number of CSR reads from the FCIFIFO register to dequeue the entire CFrame, including padding and vertical parity. If a read is issued to an empty FCIFIFO, or if the FCIFIFO does not yet contain a complete CFrame, then a four byte idle CFrame (0x0000FFFF) will be read back.

The nearly-full signal is based on the high watermark programmed into HWM_Control[FCIFIFO_Int_HWM]. The nearly-full is asserted, this means that the FCIFIFO handler thread should contain multiple CFrames (unless the high watermark has been set low) and that higher priority needs to be given to draining the FCIFIFO to prevent flow control from being asserted to the switch fabric.

In addition, FCIFIFO has a flow control signal called RXCFC which is connected to the TXCFC input pin of the egress processor. If asserted, this tells the egress processor to stop on the CBus. RXCFC is triggered by a high watermark set in HWM_Control[FCIFIFO_Ext_HWM].

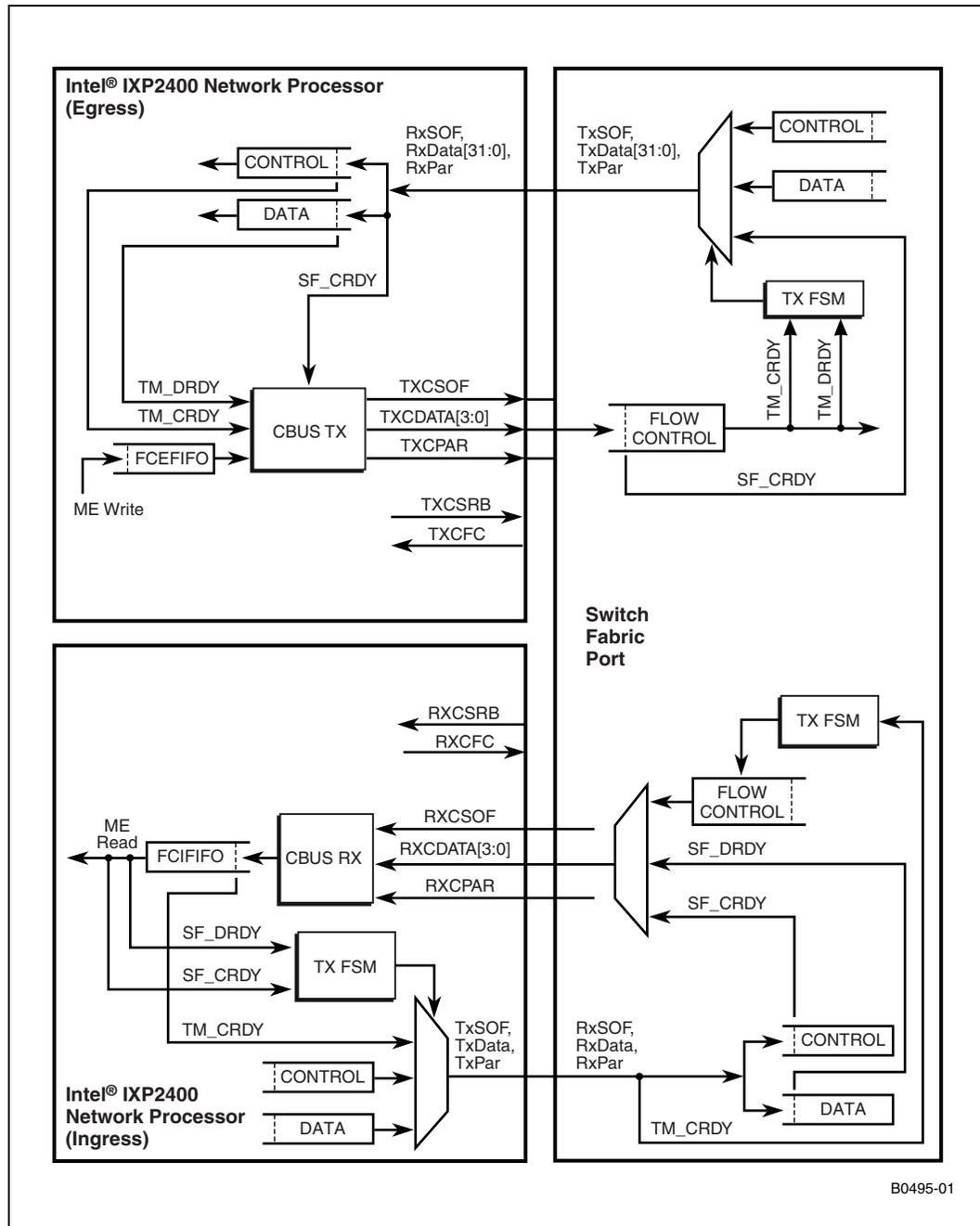
Note: FCIFIFO in the ingress processor never enqueues Idle CFrames in either full duplex or simplex modes. The transmitted Idle CFrames are injected by the control state machine, not taken from the FCEFIFO.

8.7.3 Simplex Mode

In simplex mode, the CBus is connected directly to the switch fabric; flow control information is exchanged directly between the egress processor and the switch fabric, and directly between the ingress processor and the switch fabric. IXP2400 MSF does not support transmit of flow control CFrames on CBus in simplex mode whose length is not $n*4$ bytes in length (there is no such restriction on non-flow control CFrames, they can have arbitrary length).

[Figure 100](#) contains a block diagram which shows how CBus is connected and used in simplex mode.

Figure 100. Simplex Mode Block Diagram



8.7.3.1 Link Level

The TXCSRFB/RXCSRFB and TXCFC/RXCFC pins are not used at all in simplex mode.

The egress processor uses the TXCSOF, TXCDATA, and TXCPAR pins to send flow control CFrames directly to the switch fabric. The TM_CRDY and TM_DRDY bits are placed directly into the base header of outgoing CFrames by hardware. The TM_xRDY bits are also visible to software through `FC_Egress_Status[TM_CRDY]` and `FC_Egress_Status[TM_DRDY]`.

The switch fabric uses the SF_CRDY bit to indicate that its flow control FIFO cannot accept any more data. This bit is set by the egress processor to stop transmission of CFrames from the FCEFIFO onto the CBus transmit bus. If SF_CRDY is deasserted, the CFrame currently being transmitted will complete. SF_DRDY is ignored by the egress processor.

The ingress processor uses the RXCSOF, RXCDATA, and RXCPAR pins to receive flow control CFrames directly from the switch fabric. The SF_CRDY and SF_DRDY bits are extracted from the base header of incoming CFrames. Transmit hardware in the ingress processor uses the SF_xRDY bits to flow control the data and control transmit. The SF_xRDY bits are also visible to software via `FC_Ingress_Status[SF_CRDY]` and `FC_Ingress_Status[SF_DRDY]`.

The ingress processor uses the TM_CRDY bit to indicate that FCIFIFO is nearly full and cannot accept any more CFrames. If the ingress processor deasserts TM_CRDY, the switch fabric should finish transmitting the current CFrame from its flow control FIFO, then stop until TM_CRDY is asserted again. TM_DRDY is not used by the ingress processor and is always asserted.

8.7.3.2 Fabric Level

The ingress processor receives flow control CFrames through the CBus interface into FCIFIFO. FCIFIFO is accessed exactly the same way as in half-duplex mode. The steps are the same.

If FCIFIFO exceeds its high watermark, as determined by `HWM_Control[FCIFIFO_Ext_HWM]`, it will cause the TM_CRDY bit to be deasserted. This bit is carried in all outgoing CFrames on the main CSIX interface.

The egress processor transmits flow control CFrames through the CBus interface using FCEFIFO. This is done by performing CSR writes, using `msf[write]`, to the FCEFIFO register. The ME must test `FC_Egress_Status[FCEFIFO_Full]` to check if the FCEFIFO has sufficient space before writing to it. The ME creating the CFrame must first write a valid header followed by the payload; hardware will generate horizontal and vertical parity and insert any necessary padding.

After the CFrame has been written to the FCEFIFO, the ME writes to the `FCEFIFO_Validate` CSR to indicate that the CFrame is ready to be sent out on TXCDATA. This is required to prevent underflow by insuring that the entire CFrame is in FCEFIFO before transmission is started. A validated CFrame at the head of FCEFIFO will be transmitted only if the SF_CRDY bit from the switch fabric is asserted; transmission is held off if it is deasserted. Once CFrame transmission begins, the entire CFrame is sent regardless of changes in SF_CRDY. SF_DRDY is ignored.

If there is no valid CFrame in FCEFIFO or if SF_CRDY is deasserted, then idle CFrames are sent on TXCDATA. The idle CFrames will carry TM_CRDY and TM_DRDY information. In all cases, the switch fabric must honor the ready bits to prevent overflowing RBUF.

Note: Although the intent is to transmit flow control CFrames via CBus, there are no hardware restrictions on transmitting any type of CFrame through this interface.

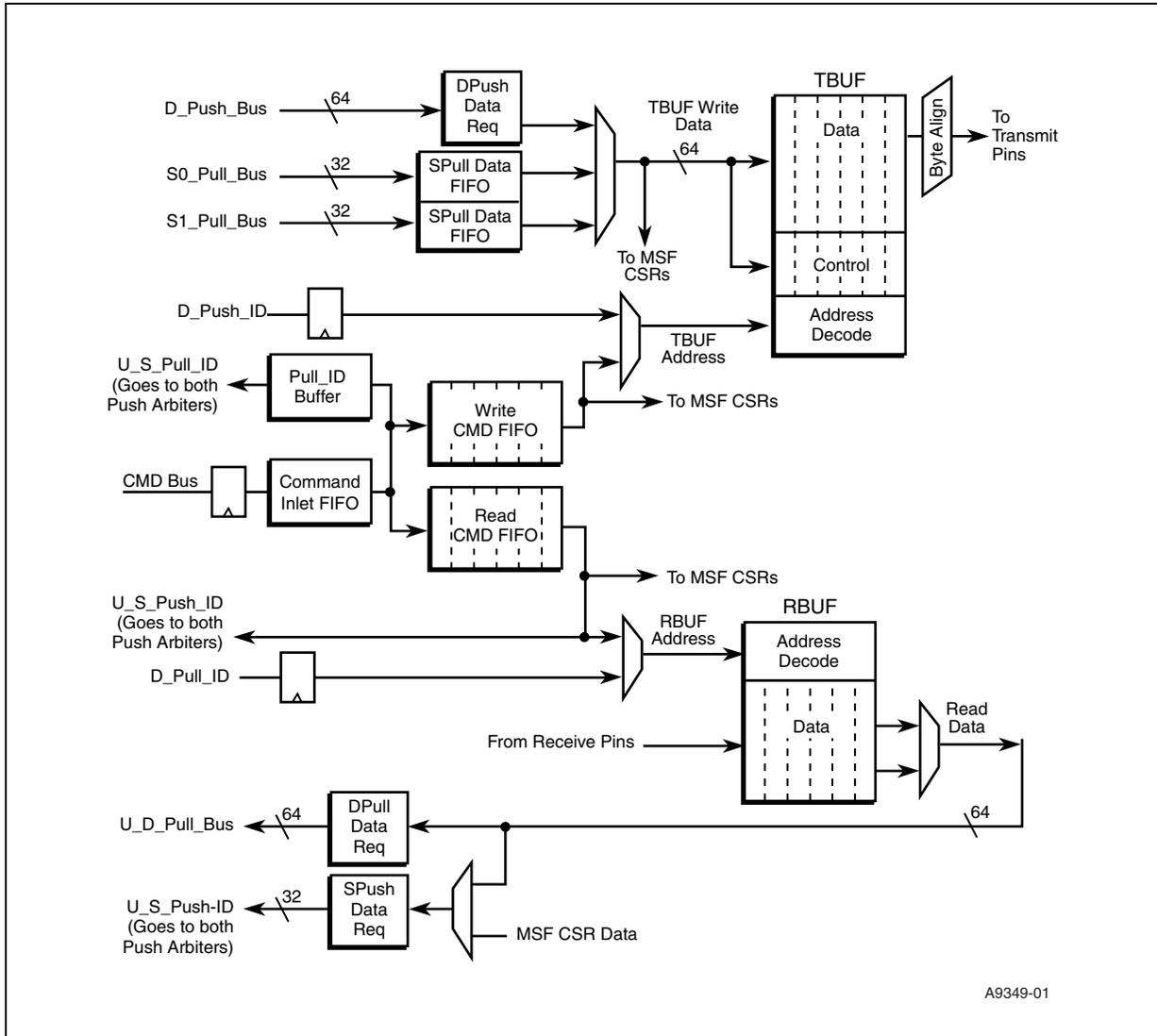
Note: If there is no switch fabric present, the CBus ports can be used for interchip communication. The hardware configuration would be the same as for Full Duplex Mode, but the CBus interfaces would be configured for Simplex mode operation. This makes it possible for software running on one processor to write messages, in CFrame format, into FCEFIFO and have the messages passed to

the other processor via CBus. TXCFC/RXCFC is used to indicate that the FCIFIFO has exceeded the high watermark.

8.8 Interface to Command and Push and Pull Buses

Figure 101 shows the interface of the MSF block to the command and push and pull buses.

Figure 101. Block Diagram of the MSF Block to the Command and Push and Pull Buses



Data transfers to and from the TBUF/RBUF are done in the following cases.

8.8.1 RBUF or CSR to ME SRAM Read Transfer Register

```
msf[read, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token
```

For transfers to ME, the MSF act as a target. Commands from MEs are received on the command bus. The commands are checked to see if they are targeted to MSF. If so they are enqueued into the Command Inlet FIFO, and then moved to the Read Command FIFO. When the Command Inlet FIFO is near full, it asserts a signal to the command arbiters. The command arbiters will prevent further commands to MSF until after the full signal is deasserted. 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 ME SRAM Write Transfer Register to TBUF or CSR

```
msf[write, $s_xfer_reg, src_op_1, src_op_2, ref_cnt], optional_token
```

For transfers from ME, MSF acts as a target. Commands from MEs are received on the command bus. The commands are checked to see if they are targeted to MSF. If so they are enqueued into the Command Inlet FIFO, and then moved to the Write Command FIFO. When the Command Inlet FIFO is near full, it asserts a signal to the command arbiters. The command arbiters will prevent further commands to MSF until after the full signal is deasserted. The control logic then arbitrates for S_PULL_BUS, and when granted it receives and registers the data from the ME 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 ME to MSF CSR Fast Write

```
msf[fast_write, src_op1, src_op2]
```

For fast write transfers from ME, MSF acts as a target. Commands from MEs are received on the command bus. The commands are checked to see if they are targeted to MSF. If so they are enqueued into the Command Inlet FIFO, and then moved to the Write Command FIFO. When the Command Inlet FIFO is near full, it asserts a signal to the command arbiters. The command arbiters will prevent further commands to MSF until after the full signal is deasserted. The control logic uses the address and data, both found in the address field of the command (see the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for register details). It then writes the data into the CSR specified.

8.8.4 Transfer from RBUF to DRAM

```
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 D_PULL_DATA Register. It is then multiplexed and driven to the DRAM channel on D_PULL_BUS.

8.8.5 Transfer from DRAM to TBUF

```
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 Registers

Please see the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for IXP2400 MSF register information.



This page intentionally left blank.

This section contains information on the IXP2400 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 102](#) and include:

- PCI Core Logic
- PCI Bus Arbiter
- DRAM Interface Logic
- SRAM Interface Logic
- Mailbox and Message Registers
- DMA Engine
- 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.

These are the data transfer paths supported as shown in [Figure 103](#):

- 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 (XScale core, Microengine) accesses to internal registers within PCI unit
- DMA
 - Descriptor read from SRAM.
 - Data transfers between PCI and DRAM.
- Push/Pull Master (XScale core and Microengines) direct read and write to PCI Bus

Figure 102. PCI Functional Blocks

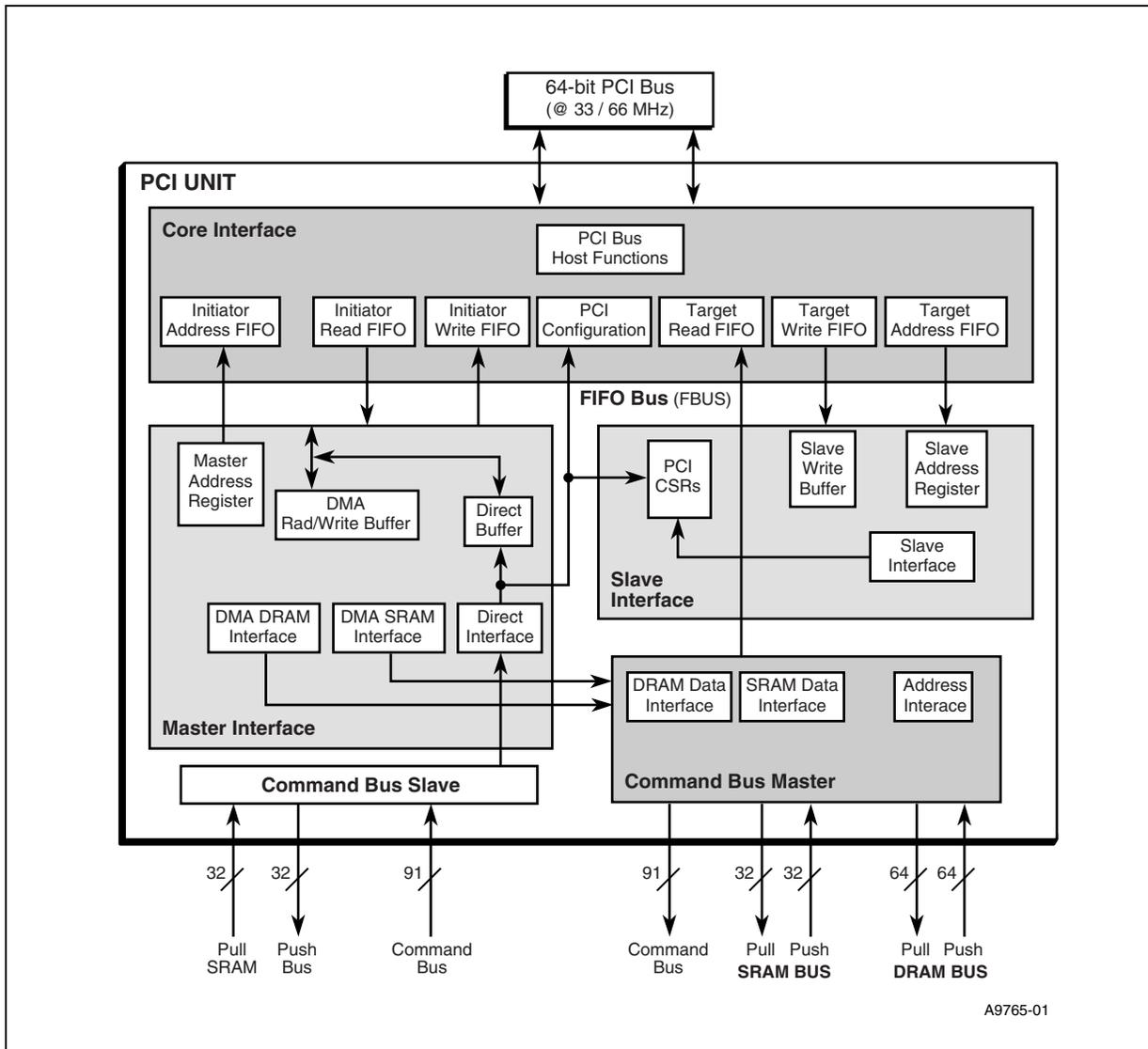
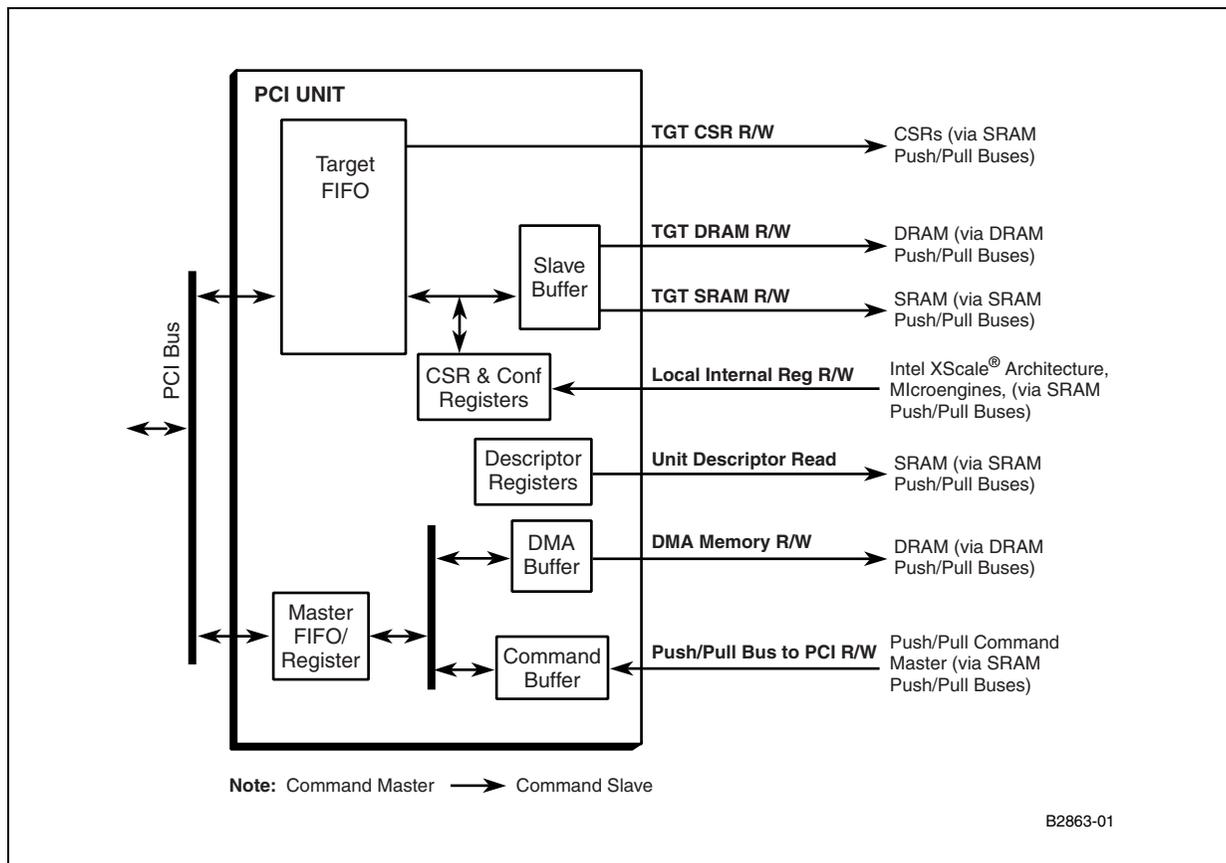


Figure 103. Data Access Paths



9.2 PCI 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 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 eight data FIFOs and four address FIFOs in the core. The separate slave and master data FIFOs allows simultaneous operations and multiple outstanding PCI bus transfers. 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.

Table 125. Atomic Target Write to Memory Option

| Stepping | Description |
|---------------|---|
| IXP2400 A0/A1 | The address FIFO is always set as a 4-entry FIFO. |
| IXP2400 B0 | If the PCI Control CSR bit[18] of the ATWE Atomic Write Enable register is set to 0, the address FIFO will always be set as a 4-entry FIFO. If the PCI Control CSR bit[18] of the ATWE Atomic Write Enable register is set to 1, the address FIFO will be set to a single-entry FIFO. The write to memory will always be a single burst if the address is aligned. And, the burst size must be 64 bytes (or less) for atomic write purposes. |

Table 126. PCI Core 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 |

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 127 lists the maximum PCI Interface loading.

Table 127. Maximum Loading

| Bus Interface | Max # of Loads | Trace Length (inches) |
|---------------|--|-----------------------|
| PCI | 4 loads at 66 MHz bus frequency 8 loads at 33 MHz bus frequency | 5 to 7 |

9.2.1 PCI Commands

Table 128 lists the supported PCI commands and identifies them as either a target or initiator.

Table 128. 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 |

Table 128. PCI Commands (Sheet 2 of 2)

| C_BE_L | Command | Support | |
|--------|-----------------------------|--|---------------|
| | | Target | Initiator |
| 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 |
| 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 IXP2400 Network Processor Initialization

When the IXP2400 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 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 XScale core will boot from the prom and be able to program the BAR space mask registers. If PROM_BOOT is not asserted, the 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 XScale core has set the Initialize Complete bit in the Control register. This bit is set after the 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 129).

Table 129. PCI BAR Programmable Sizes

| Base Address Register | Address Space | Sizes |
|-----------------------|---------------|--|
| PCI_CSR_BAR | CSR | 1Mbyte |
| PCI_SRAM_BAR | SRAM | 0Byte, 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 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 an External PCI Host

In this case, the internal PCI Unit is not hosting the PCI Bus. The host processor is allowed to configure the internal CSRs while the 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 XScale core may be downloaded into local memory.

The host processor then clears the XScale core reset bit in the PCI Reset Register. This de-asserts the internal reset signal to the 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 130).

Table 130. 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 | 128M/256M/512M/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. The 64-bit mode can be overridden by writing to IXP_PARA CSR bit 1.

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.

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 two distinct channels 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 4 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.

A long-burst enable mode ensures that long bursts will not be disconnected unless the write buffer data cannot be appropriately drained.

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 or writes 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 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 PCI_TRDY# 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 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_CMD_STAT) 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, 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 (CFG_PCI_ARB is not active), the req_l[0] and gnt[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.

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_CMD_STAT) received master abort bit, which, if enabled, interrupts the 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_CMD_STAT) received target abort bit, which, if enabled, interrupts the XScale core.
- The PCI unit can dessert frame_1 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_CMD_STAT) received master abort bit, which, if enabled, interrupts the XScale core.
- If the PCI unit receives a target abort, it sets the status registers (PCI_CMD_STAT) received target abort bit, which, if enabled, interrupts the 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 PCI_FRAME#, it begins to decrement its master latency timer. When the timer value reaches zero, the PCI unit checks the value of gnt_l[0]. If gnt_l[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 PCI_DEVSEL# will not be received, and therefore no errors 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 Command and Status 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.

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.
- Crossing the 64-byte boundary on the SRAM and DRAM BAR (except on burst writes, where the long-burst enable bit is set).
- 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 IXP2400 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 XScale core if it is enabled by the XScale core Interrupt Enable Register. The XScale software can respond to the interrupt by running an application specific test. Upon successful completion of the test, the 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_RST_DIR 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 PCI 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 131. Legal Combinations of the Strap Pin Options

| | CFG_PCI_HOST (GPIO[1]) | CFG_PCI_ARB (GPIO[2]) | CFG_RST_DIR (Central function) | CFG_PROM_BOOT (GPIO[0]) |
|---------------|---------------------------|--------------------------|-----------------------------------|----------------------------|
| ok | 0 | 0 | 0 | 0 |
| ok | 0 | 0 | 0 | 1 |
| ok | 0 | 0 | 1 | 1 |
| Not support | 0 | 1 | 0 | x |
| ok | 0 | 1 | 1 | 1 |
| Not support | 1 | 0 | 0 | x |
| ok | 1 | 0 | 1 | 1 |
| Not supported | 1 | 1 | 0 | x |
| ok | 1 | 1 | 1 | 1 |

Note

- * CFG_RST_DIR = 1 then central function.
- * CFG_PCI_HOST must be central function.
- * CFG_PCI_ARB 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 XScale core Interrupt Enable Register.

9.2.11.2 PCI Reset Output

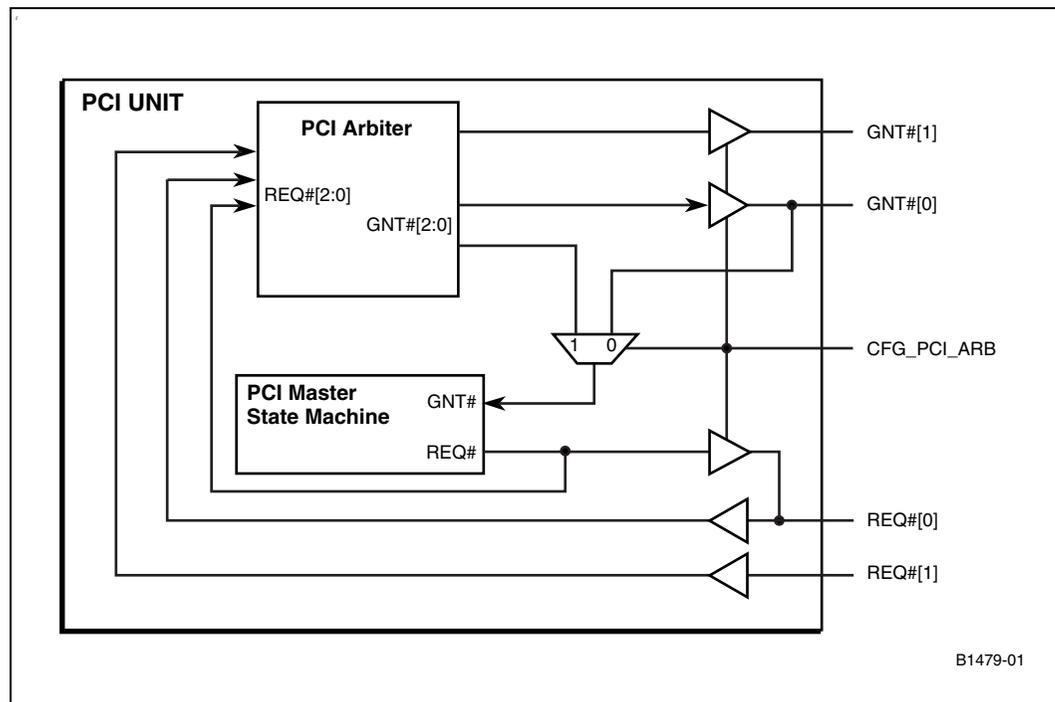
If the IXP2400 Network Processor is central function (CFG_RST_DIR =1), PCI Unit will be asserting the PCI_RST# after the system power-on. The XScale core has to write to the PCI External Reset bit in the IXP Reset register to de-assert the PCI_RST#. In this case, chip reset (SYS_RESET_L) is driven by a signal other than PCI_RST#.

When the PCI Unit is not configured as the central function (CFG_RST_DIR =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 104](#), the local bus request and grant pair become externally not visible.

Figure 104. 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 PCI_FRAME# 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 132](#).

Table 132. 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 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.

9.3.2 SRAM Interface

The SRAM interface connects the FBus to the internal push/pull command bus and the 32-bit 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.

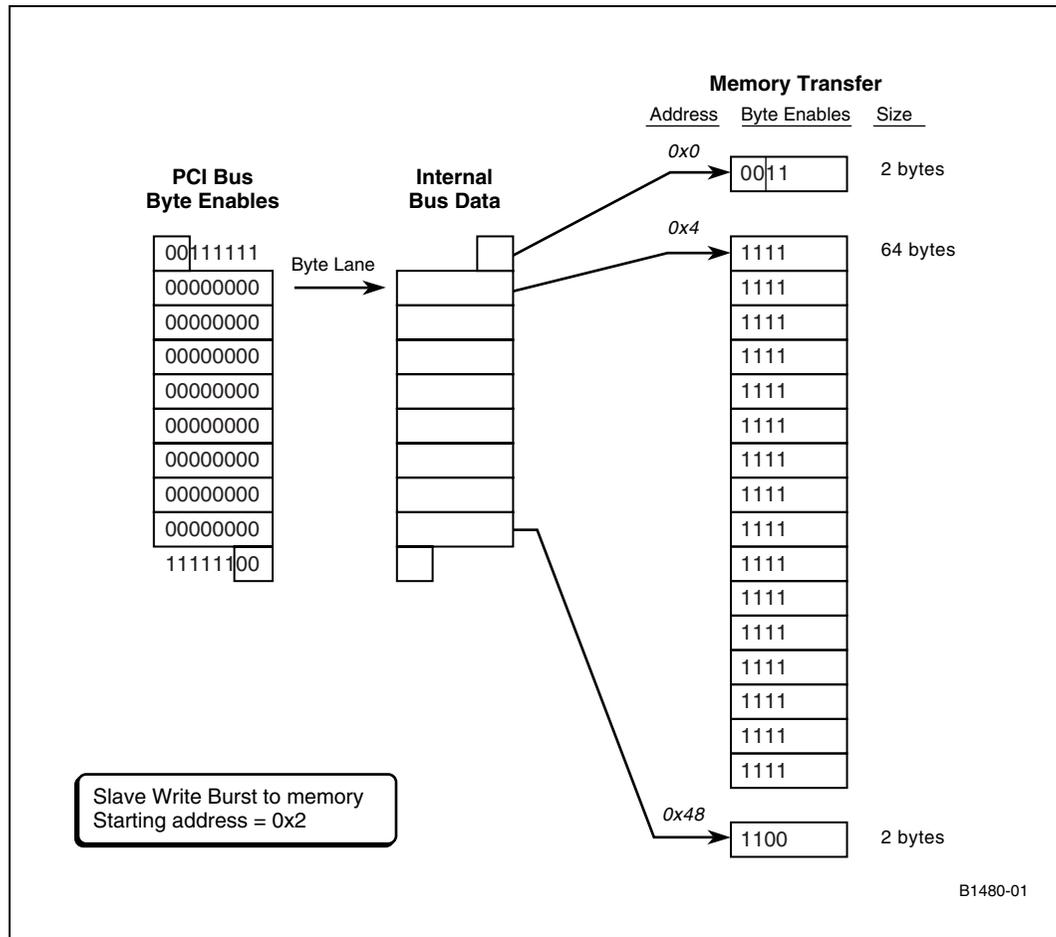
If the PCI_SRAM_BAR is used, the target state machine generates a request to the command bus for SRAM access. Once the grant is received, the address, then data is directed between the slave FIFOs of the PCI core and the SRAM push/pull bus.

9.3.2.1 SRAM Slave Writes

The slave write buffer is used to support memory burst accesses. The buffer is added so that the data transfer for each clock and burst size can be determined before a memory request is issued. Data is assembled in the buffers before being sent to memory for SRAM write.

On the push/pull bus, SRAM access can start at any address and have length up to 16 Dwords as shown in [Figure 105](#). For masked writes, only size 1 is supported to transfer up to four bytes.

Figure 105. 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 minimum of 8 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

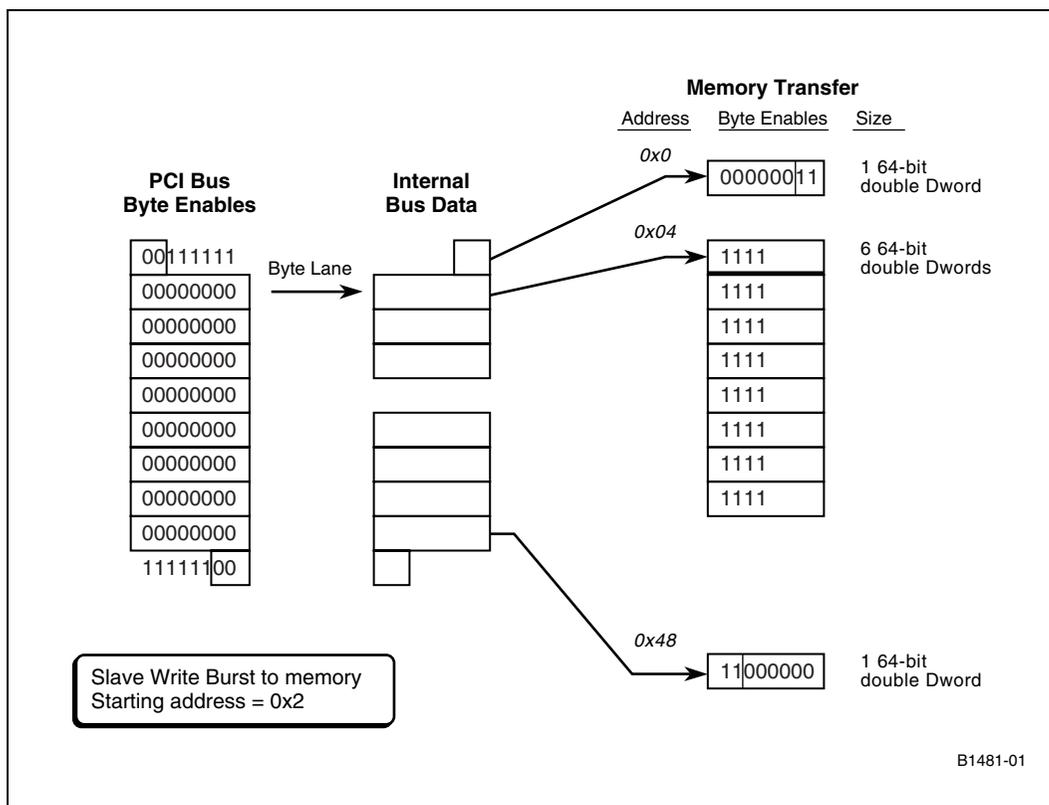
The slave write buffer is used to support memory burst accesses. The buffer is added to guarantee data transfer for each clock and burst size can be determined before memory request is issued. Data is assembled in the buffers before being sent to memory for memory write.

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 107](#) 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 106](#)).

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.

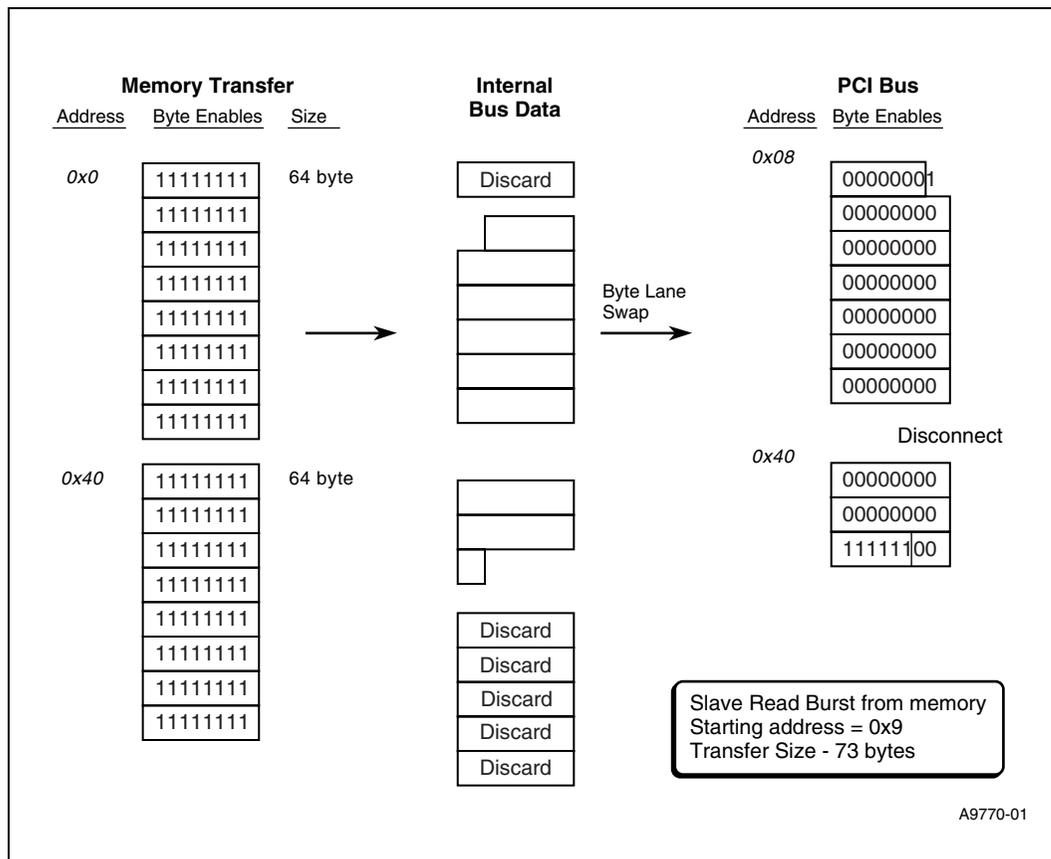
Figure 106. Example of Target Write to DRAM of 68 bytes



9.3.3.2 DRAM Slave Reads

For a target read from memory, the whole 64 byte is fetched from the DRAM in the case of system using DDR memory technology. In the case of a DRAM system using RDRAM, 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 107.

Figure 107. 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 maximum size of DRAM data read is 64 bytes. 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 XScale core and a device on the PCI Bus.

Four mailbox registers are provided so that messages can be passed between the XScale core and a PCI device. All four registers are 32 bits and can be read and written from both the 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 XScale core Doorbell register that is used by a PCI device to generate an XScale core FIQ and a separate PCI Doorbell register that is used by the 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 IXP2400 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 bits of data.

The doorbell interrupts are controlled through the registers shown in [Table 133](#).

Table 133. Doorbell Interrupt Registers

| Register Name | Description |
|----------------------------|---|
| XScale core Doorbell | Used to generate the XScale core Doorbell interrupts |
| XScale core Doorbell Setup | Used to initialize the 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 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 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 XScale core interrupt signal is the output of an NAND function of all the 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 XScale core, since writing 1 changes the doorbell bit to the proper asserted state (i.e., 0 for an 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 XScale core, since writing 1 changes the doorbell bit to the proper de-asserted state (i.e., 1 for an XScale core interrupt and 0 for a PCI interrupt).

[Figure 108](#) and [Figure 109](#) illustrates how a Doorbell interrupt is asserted and cleared by both the XScale core and a PCI device.

Figure 108. Generation of the Doorbell Interrupts to PCI

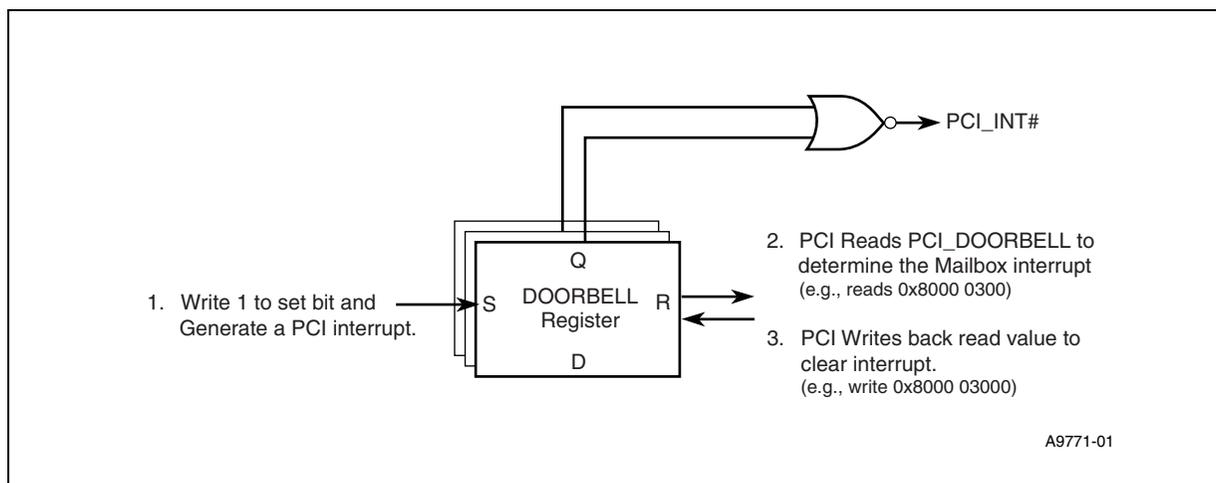
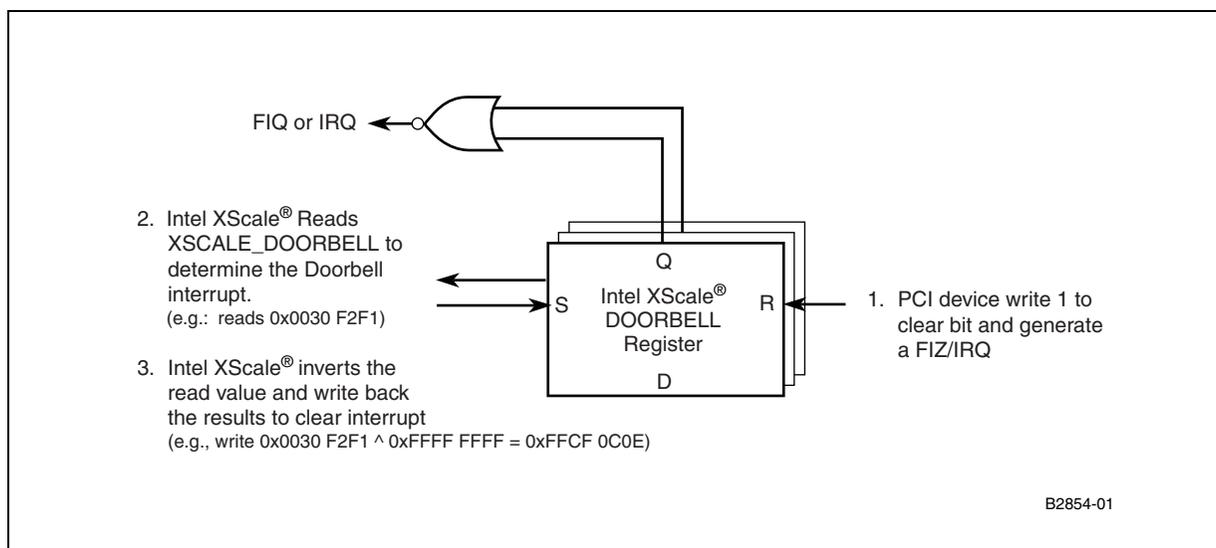


Figure 109. Generation of the Doorbell Interrupts to the Intel® XScale® Core



The Doorbell Setup register allows the 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 XScale core and PCI device to clear an interrupt that it has generated to the other device. If the 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 XScale core can clear the interrupt by write 0 to it.

Second, it allows the 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 XScale core Doorbell setup register.

9.3.5 PCI Interrupt Pin

An external PCI interrupt can be generated in the following ways:

- The XScale core initiates a Doorbell interrupt XSCALE_INT_ENABLE.
- One or more of the DMA channels have completed the DMA transfers.
- The XS_INT bit is set by XScale to generate a PCI Interrupt.
- Internal-unit-generated error or interrupt
- Watchdog interrupt

Figure 110 shows how PCI interrupts are managed via the PCI and the XScale core.

Figure 110. PCI Interrupts

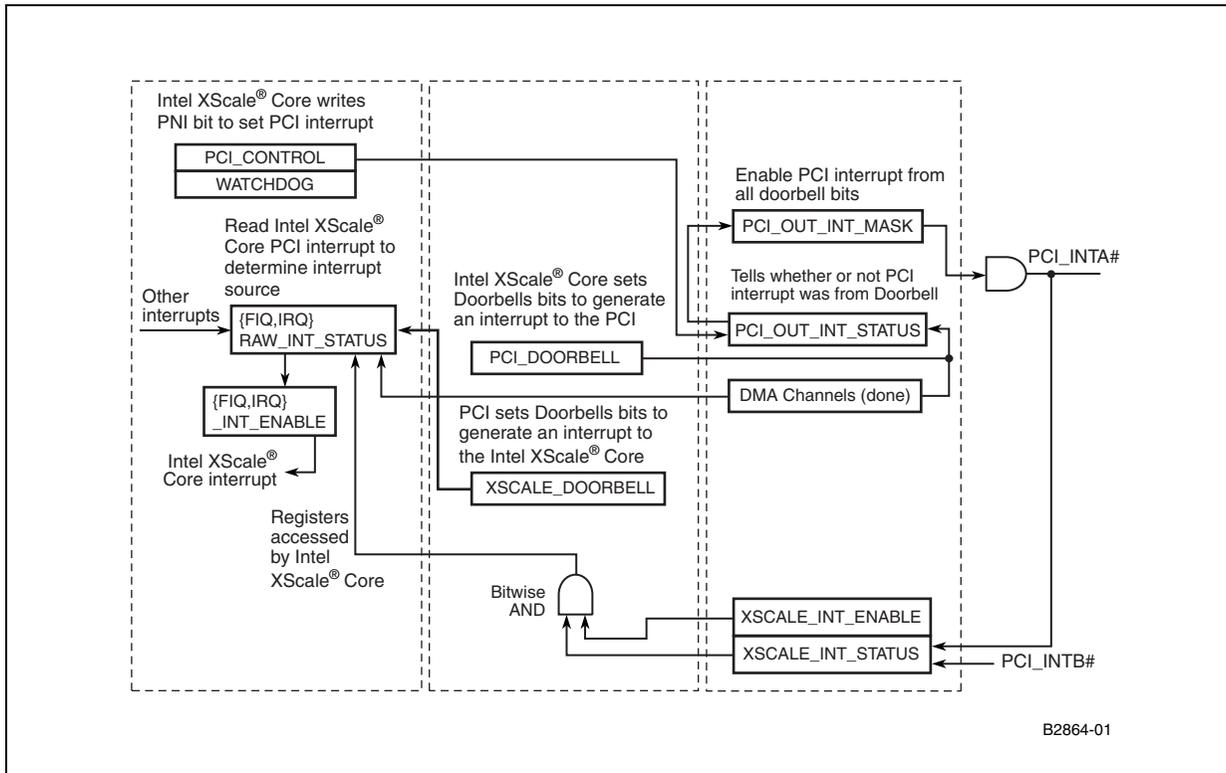


Table 134. Internal Unit Interrupt Directly to PCI Option

| Stepping | Description |
|----------|---|
| A0, A1 | The IRQ registers are in the XScale gasket; the PCI does not have IRQ status information. If the XScale gasket is in reset mode, the interrupt resources cannot be handled by the PCI host. |
| B0 | <p>The IRQ status registers are in both XScale gasket and PCI.</p> <p>If PCI_OUT_INT_MASK[0] set to 1 then the PCI will not pass the interrupt information to an external PCI.</p> <p>If PCI_OUT_INT_MASK[0] set to 0, the XScale gasket is in reset mode. The interrupt resources can be handled by the PCI host, PCI IRQ CSR.</p> <p>PCI_OUT_INT_STATUS[0] will collect all the different unit interrupts from XScale_Interrupt_Status or Xscale_Error_Status CSR to external PCI.</p> <p>Add DRAM, SRAM, Slowport, ME interrupt to the following CSR:</p> <ul style="list-style-type: none"> XScale Error Status (XSCALE_ERR_STATUS) XScale Error Enable (XSCALE_ERR_ENABLE) XScale Interrupt Status (XSCALE_INT_STATUS) XScale Interrupt Enable (XSCALE_INT_ENABLE) |

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 three 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. The DRAM read is always 64-byte.

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 separate FBus initiator read FIFO and initiator write FIFO, which are inside the PCI Core and three DMA buffers (corresponding to the three DMA channels), which is for buffering data to and from the DRAM. Since there is no simultaneous DMA read and write outstanding, one shared 128-byte buffer is used for both read and write DRAM data

Up to three DMA channels are running at a time with three descriptors outstanding. The three 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 XScale core, a Microengine and the external PCI host can access the three DMA channels. The first two channels can function in one of the following modes, as determined by the DMA_INF_MODE register:

- The XScale core owns both DMA channel 1 and channel 2
- The Microengines own both DMA channels 1 and channel 2
- PCI host owns both DMA channel 1 and channel 2
- The XScale core owns DMA channel 1 and the Microengines own DMA channel 2 (default).

The third channel can be allocated to either the XScale core, PCI host, or Microengines.

The DMA mode can be changed only by the 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, DMA2, and DMA3 until all three DMA channels are done.

9.4.1.2 Special Registers for Microengine Channels

Interrupts are generated at the end of DMA operation for the 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 the contents of DMA control 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. 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 135](#).

Table 135. 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 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 (CHAN_X_DESC_PTR).
2. The DMA channel owner writes the DMA Channel Control register (CHAN_X_CONTROL) with miscellaneous control information and also sets the channel enable bit (bit 0). The channel initial descriptor bit (bit 4) in the CHAN_X_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, CHAN_X_BYTE_COUNT, CHAN_X_PCI_ADDR, CHAN_DRAM_ADDR, and CHAN_X_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 CHAN_X_CONTROL register.
5. If the end of chain bit (bit 31) in the CHAN_X_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 CHAN_X_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 CHAN_X_BYTE_COUNT register must be set, and the value in the CHAN_X_DESC_PTR register is not used. (If the end of chain bit is not set, the CHAN_X_DESC_PTR will point to the next description in a chain.)
2. The DMA channel owner writes the base address of the DMA transfer into the CHAN_X_PCI_ADDR to specify the PCI starting address.

3. When the first descriptor is in the CHAN_X_BYTE_COUNT register, the CHAN_X_DRAM_ADDR register must be written with the address of the data to be moved.
4. The DMA channel owner writes the CHAN_X_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 CHAN_X_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 CHAN_X_CONTROL register.
6. Since the end of the chain bit (bit 31) in the CHAN_X_BYTE_COUNT register is set, the channel sets the channel chain done bit (bit 7) in the CHAN_X_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 XScale core
If enabled via the XScale core Interrupt Enable registers, the DMA channel interrupts the XScale core by setting the DMA channel done bit in the CHAN_X_CONTROL register, which is readable in the 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.

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.

Table 136. PCI Maximum Burst Size

| Stepping | Description |
|---------------|--|
| IXP2400 A0/A1 | The maximum burst size on the PCI Bus is always 64 bytes. |
| IXP2400 B0 | The maximum burst size on the PCI Bus is a 64-bytes if the PCI_LONG_EN bit is not set. If DMA Control (CHAN_1:3_CONTROL) register for DMA long burst enable is set, the PCI will continue burst DMA as long as data is available from the FIFO. If the PCI_CONTROL register for Target Write long burst enable is set, the PCI Target will continue to burst as long as data is available from the FIFO. PCI Target read will not support long burst. |

9.4.2 Push/Pull Command Bus Target Interface

Through the command bus target interface, the command bus masters (PCI, 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 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 IXP2400 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 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 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.

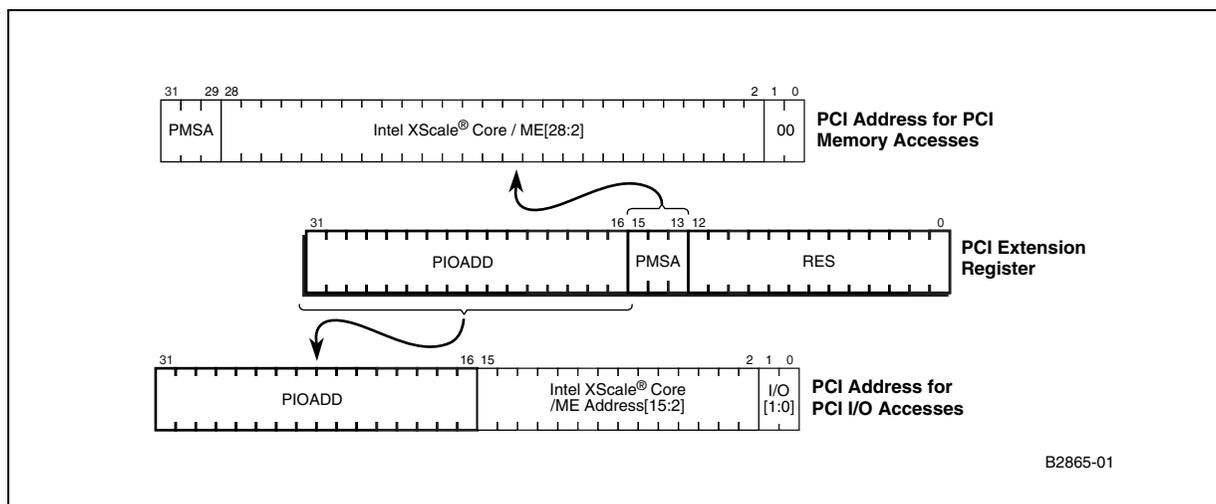
The data from the SRAM push/pull data bus are transferred through the master data port of the FBus interface of the PCI core. The PCI Core will handle all the PCI Bus protocol handshakes. The SRAM pull data received for a write command will be transferred to the Master write FIFO for PCI writes. For 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 byte 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 111](#) shows how the address is generated from a Command Bus Master address.

Figure 111. PCI Address Generation for Command Bus Master to PCI



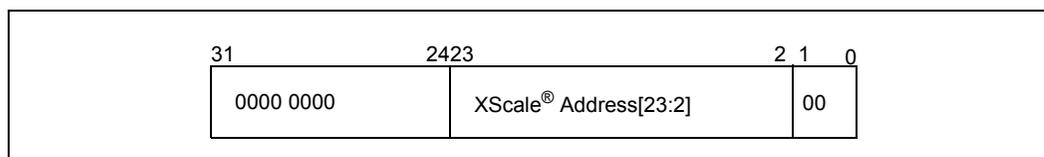
9.4.2.3.2 PCI Address Generation for Configuration cycles

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 137](#) and [Figure 112](#):

Table 137. 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 112. 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. The XScale core doesn't generate byte enables for reads; XScale-to-PCI, memory or IO reads require that all byte enables be active.

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 138](#):

Table 138. 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 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 (DTX) in the PCI_CONTROL.
4. If enabled (XSCALE_INT_ENABLE [DTE]), the PCI unit interrupts the 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 sets 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 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 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 Phase on a Target Read

1. Slave Interface sets PCI_CONTROL[TGT_DRAM_ERR], which will interrupt the 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 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 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.
1. If PCI_CMD_STAT[PERR_RESP] is set:
 - a. Master Interface sets PCI_CONTROL[DPE] which will interrupt the 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 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 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 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 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 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 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 XScale core if enabled.
 - c. Data that has been read from PCI is sent to the 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 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, the PCI doesn't need to do any long word swapping between two 32 bits long words (LW1, LW0). But the 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 139](#) through [Table 146](#):

PCI is allow to do byte enable swapping only without the data swapping or allow data swapping only without byte enable swapping. When the PCI handle the mis-aligned data in the above two cases, PCI will only care about valid data. So the PCI will drive any data values for those mis-aligned and invalid data portions.

**Table 139. 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 140. 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 141. 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 142. 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 after LW0 | | | | 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 143. 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 144. 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 145. 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 146. 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_DEI 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_DEO 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 do not need to do any long word byte enable swapping between two 32 bits 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 147 through Table 154:

Table 147. 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 Word1byte 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 148. 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 Word1byte 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 149. 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 Word1byte 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 Word1byte 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 150. 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 151. 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 152. 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 153. 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 154. 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 Word1 byte 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 Word1 byte 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 155. PCI I/O Cycles Generate Data Swap Enable Option

| Stepping | Description |
|--|---|
| A0, A1 | PCI I/O cycle is treated like a CSR, where the data bytes are not swapped; it will be sent in the same byte order even if the PCI bus is configured in Big or Little endian mode. |
| B0 | When PCI Control (PCI_CONTROL) CSR bit 17 IEE is 0, then it will be sent in the same byte order even if the PCI bus is configured in Big or Little endian mode. When PCI Control (PCI_CONTROL) CSR bit 17 IEE is 1, then PCI IO data will follow the same memory space-swapping rule. The address always follows the physical location, for example: |
| | BEs swapped (1 byte access) |
| | ad[1:0] |
| | 0 0 |
| | 0 1 |
| | 1 0 |
| | 1 1 |
| | BEs not swapped (1 byte access) |
| | ad[1:0] |
| | 0 0 |
| | 0 1 |
| | 1 0 |
| | 1 1 |
| | BEs swapped (2 byte access) |
| | ad[1:0] |
| | 0 0 |
| | 0 1 |
| | 1 0 |
| | BEs not swapped (2 byte access) |
| | ad[1:0] |
| 0 0 | |
| 0 1 | |
| 1 0 | |
| BEs swapped (3 byte access) | |
| ad[1:0] | |
| 0 0 | |
| 0 1 | |
| BEs not swapped (3 byte access) | |
| ad[1:0] | |
| 0 0 | |
| BEs swapped (4 byte access) | |
| ad[1:0] | |
| 0 0 | |

The BE_BEI 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_BEO 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.

9.7 PCI Strap Pins Options

Table 156. PCI Strap Pins

| Signal | Name | Description |
|-------------------|------|---|
| CFG_RST_DIR | PCF0 | PCI central function pin: 1: IXP is supporting central function 0: IXP is non central function |
| CFG_PCI_ARB | PCF1 | PCI Internal Arbiter pin:1—IXP2400 internal arbiter is used <ul style="list-style-type: none"> • PCI_Host must be central function. • PCI_Arbiter must be central function. |
| CFG_PROM_BOOT | PCF2 | PCI Prom Boot pin: 1—IXP will boot from PROM 0—IXP will boot from DRAM initialized by PCI Host. |
| CFG_PCI_BOOT_HOST | PCF3 | PCI Prom Boot pin: 1—IXP will configure the PCI system 0—The external host will configure the PCI system. <ul style="list-style-type: none"> • PCI_Host must be central function. • PCI_Arbiter must be central function. |
| CFG_PCI_SWIN[1:0] | SWIN | 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] | DWIN | DRAM BAR Window 11: SRAM BAR size of 1024 Mbyte 10: SRAM BAR size of 512 Mbyte 01: SRAM BAR size of 256 Mbyte 00: SRAM BAR size of 128 Mbyte |



This page intentionally left blank.

Clocks, Reset, and Initialization 10

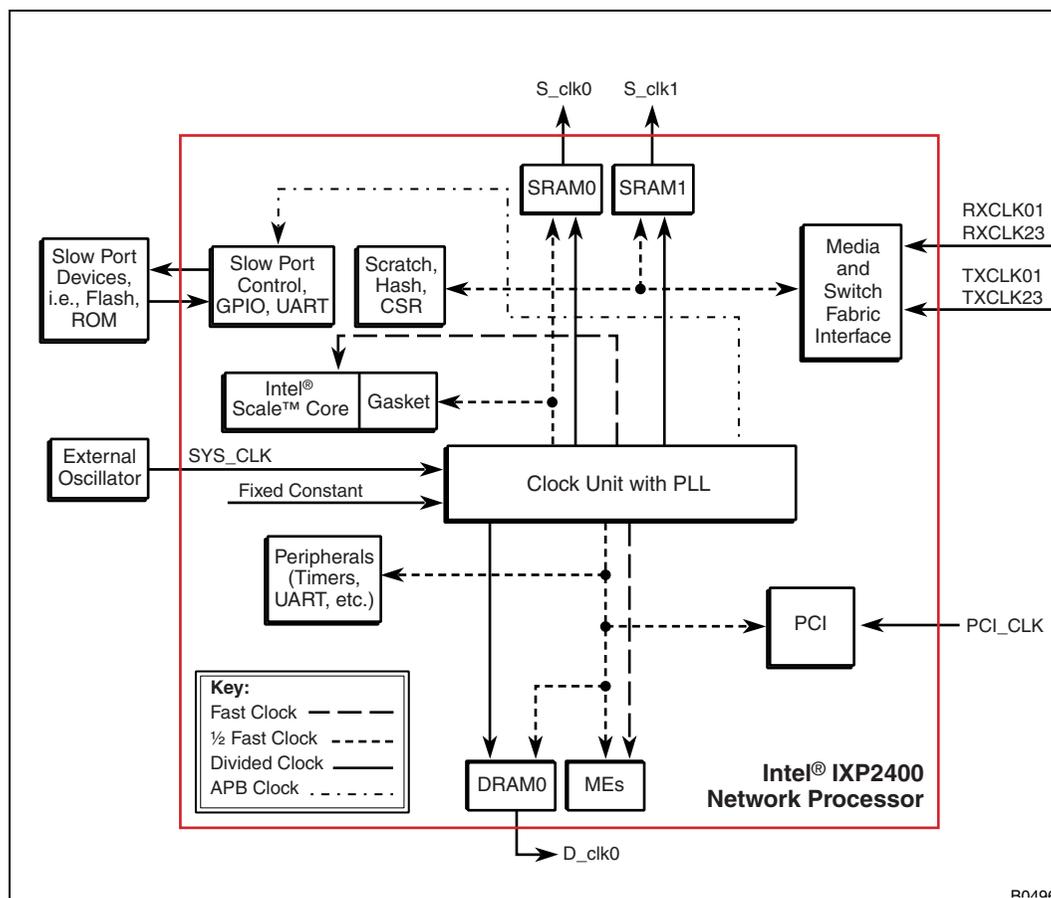
10.1 Overview

This chapter describes the IXP2400 clocks, reset and initialization sequence.

10.2 Clocks

Figure 113 shows the overall clock generation and distribution.

Figure 113. Overall Clock Generation and Distribution



As shown in Figure 113, there is a centralized clock generator. It takes an external reference clock 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 the Intel XScale® core, Microengines (MEs), SRAM and DRAM controllers, peripheral units. The Media, Switch Fabric Interface and PCI controller use external clocks.

Table 157. Clock Usage Summary

| Unit Name | Description | Comment |
|--------------------|--|---|
| ME | MEs internal. | Nominal value = 600 MHz |
| Internal Buses | Command/Push/Pull interface of DRAM controller, SRAM controller, XScale, Peripheral, MSF, and PCI Units. | ½ ME frequency. Nominal Value = 300 MHz. |
| XPI | Peripheral block consisting of UART, GPIO, timers and Performance Monitor Unit. This clock is APB clock. | Operating at 1/6 of CPP clock Frequency. i.e 50MHz. The 50-MHz clock is delayed with respect to the 300 MHz. Any use of APB clock logic to interface with the peripheral units clocks and the CPP clock units should involve synchronization and handshake logic. APB clock runs at 50 MHz and does not bear any phase relationship with other clocks in the design. |
| XScale | XScale core, caches, core side of Gasket. | Same as ME, nominal value = 600 MHz |
| DRAM | DRAM pins and control logic (all of DRAM unit except Internal Bus interface). | Divide of PLL frequency. The DRAM channel uses two clocks, one at the data rate (2x clock) and one at the output frequency (1x clock). Clocks are driven by IXP2400 to external DRAMs. |
| SRAM | SRAM pins and control logic (all of SRAM unit except Internal Bus interface). | Divide of PLL frequency. Each SRAM channel has its own frequency selection. Each SRAM channel uses two clocks, one at the data rate (2x clock) and one at the output frequency (1x clock). Clocks are driven by IXP2400 to external SRAMs and/or Coprocessors. |
| Scratch, Hash, CSR | Scratch RAM, Hash Unit, CSR access block | ½ of ME. Note that SlowPort has no clock. Timing for SlowPort accesses is defined in SlowPort registers. |
| MSF | Receive and Transmit pins and control logic. | The receive clock for the Media and Switch interface can be derived: <ul style="list-style-type: none"> • From two external RX reference clocks (supplied by media or switch fabric PHY device) to two internal RX PLL then driven by IXP2400. • From two external Tx reference clocks (supplied by media or switch fabric PHY device) to two internal TX PLL then driven by IXP2400. |
| PCI | PCI pins and control logic. | External reference. The receive clock for the PCI interface can be derived: <ul style="list-style-type: none"> • From external PCI reference clock (supplied by either from Host system or on-board oscillator) to internal PCI then driven by IXP2400. |

The fast frequency on IXP2400 is generated by an on-chip PLL that multiplies a reference frequency provided by an on the board oscillator (frequency 100MHz) by a fixed multiplier. The multiplier value is 12 so the PLL will generate 1.2 GHz clock. Dividing the PLL frequency clock by various programmable integers (in the Clock Control CSR) generates internal clocks. All of these frequencies will be at 50% duty cycle with the accuracy determined by the symmetry of the PLL output.

Table 158 shows the frequencies that are available for DRAM and SRAM units based on various values of fast clock (Core PLL Output Frequency), for the supported divisor values of 3 to 6. For each value of the divisor the divider will provide the 2x clock and divide again by two to drive the 1x clock. This is shown for divide by 3 as 400/200.

Table 158. Available Clock Rates by Dividing Core PLL Output

| Divisor(n) | PLL Output Frequency (MHz) / Sys_clock OSC (MHz) | |
|--|--|------------|
| | 800 / 66.7 | 1200 / 100 |
| Divide by 2 (Microengine, XScale core Frequency) | 400 | 600 |
| Divide by 4 (CPP Frequency) | 200 | 300 |
| Divide by 8 (DRAM Frequency) | 100 | 150 |
| Divide by 12 (DRAM Frequency) | 66.7 | 100 |
| Divide by 6 (SRAM Frequency) | 133.3 | 200 |
| Divide by 8 (SRAM Frequency) | 100 | 150 |
| Divide by 12 (SRAM Frequency) | 66.7 | 100 |
| Divide by 24 (APB Frequency) | 33.4 | 50 |

Table 158 shows the clock generation circuit in IXP2400. When the chip is powered up, a ring oscillator clock will be sent to all units as the chip begins to power up. When the internal RC detect circuit is active, the clock unit will switch from using the ring oscillator to using the PLL clock.

10.2.1 CSRs

10.2.1.1 Clock Control CSR (CCR)

Clock Control selects the clock ratio for the SRAM and DRAM controllers. This register must be programmed before accesses to the SRAM/DRAM are done. In all cases a value of 0x3 indicates divide by 3, 0x4 is divide by 4, etc. up to 0x6, which is divide by 6. This register is part of the Global Chassis registers. Please see the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for additional CSR register information.

10.2.1.2 MSF Clock Control CSR (MCCR)

MSF Clock Control Register (MCCR) selects the clock ratio for the four MSF PLLs: RX0, RX1, TX0, and TX1. This register must be programmed before an application accesses the MSF. Please see the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for additional MCCR register information.

These steps must be followed during power on initialization:

1. Set the PLL in PLL disable mode (MCCR[MSF_POWERDOWN] = 1 default)
2. Set the PLL in bypass mode (MCCR[MSF_BYPASS_SEL] = 1 default)
3. Set up each of MSF clock ratio (MCCR[MSF_CLKCFG])
4. Disable the PLL bypass mode (MCCR[MSF_BY_PASS_SEL] = 0)
5. Enable the PLL (MCCR[MSF_POWERDOWN] = 0)
6. Wait for MSF PLL lock (MCCR[MSF_PLL_LOCK] = 1)
7. Enable MSF block (via MSF_Rx_Control[Rx_En] or MSF_Tx_Control[Tx_En])

8. Initialize the MSF CSR registers.

10.2.1.3 Reset History CSR

| Bits | Field | Description | RW | Reset |
|--------|--------------------|---|----|-------|
| [0] | Watchdog history | Set when watchdog timer expires. Reset when hard reset is applied or the register is read. | RW | 0x0 |
| [1] | Soft Reset history | Reserved Set when application software sets the IXP_RESET0[16] RSTALL to 1. Reset when hard reset is applied or the register is read. | RW | 0x0 |
| [2] | Hard Reset history | Reserved Set when Hardware resets the IXP. Reset when the register is read. | RW | 0x1 |
| [31:3] | Reserved | reserved | RO | 0x0 |

10.3 Reset

There are four different ways IXP2400 can be reset.

- Hardware reset via SYS_RESET_L
- CFG_RSTDIR is not asserted and PCI_RST_L is asserted
 CFG_RSTDIR is a strap pin to select the IXP2400 to drive the PCI_RST_L signal. When CFG_RSTDIR is not asserted, PCI_RST_L signal is a chip input signal.
- Watchdog timer expires and Watchdog enable bit in Timer Watchdog Enable register is set
 Once the timer expires, reset sequence is initiated. In this reset sequence, IXP_RESET0 register is reset, which in turn generates PCI_RST_L (if CFG_RSTDIR is asserted).
- Software Initiated Reset
 PCI host or XScale writes 1 at bit [16] (RSTALL) in IXP_RESET0 register. In addition to this, individual units can be reset using their respective bits in IXP_RESET0 and IXP_RESET1 registers.

10.3.1 Hardware Reset

The IXP2400 provides the SYS_RESET_L pin so that an external device can reset the IXP2400. Asserting this pin resets the internal functions and generates an external reset via the RESET_OUT# pin.

Upon power-up, SYS_RESET_L must remain asserted for 1ms after VDD is stable to properly reset the IXP2400 and ensure that the external clock is stable. While SYS_RESET_L is asserted, the processor is held in reset. When SYS_RESET_L is released, the Intel XScale® core processor begins executing from address 0X0 after the initial reset sequence is completed.

If SYS_RESET_L is asserted, while the Intel XScale® core is executing, the current instruction is terminated abnormally and the reset sequence is initiated.

The RESET_OUT# signal remain asserted until deasserted by the Intel XScale® core. It deasserts the signal by writing bit [15] of the IXP_RESET0 register.

10.3.2 PCI Initiated Reset

The IXP2400 can be reset by an external PCI Bus master when the IXP2400 is not the PCI central function (CFG_RSTDIR = 0) and PCI_RST_L is an input. The entire IXP2400 is reset during a PCI initiated reset.

When the IXP2400 is reset and CFG_RSTDIR = 1 (the IXP2400 is assigned as the PCI central function), IXP2400 drives PCI_RST_L as an output to the other devices on the PCI bus.

The RESET_OUT# signal remains asserted until deasserted by the XScale core. It deasserts the signal by writing bit [15] of the IXP_RESET0 register.

10.3.3 Watchdog Timer Initiated Reset

The IXP2400 provides a watchdog timer that can reset the Intel XScale® core. There are four timers in the IXP2400 architecture. Timer 4 can be set to be a watch dog timer. Please refer the XPI EAS in the Timer section for more details. The Intel XScale® core should be programmed to reset the watchdog 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.

The reset generated by the Watchdog timer will reset each of the functions in the IXP2400 if Watchdog reset enable bit is set (IXP_RESET0[22]=1). In this reset sequence, IXP_RESET0 register is reset after 512 cycles later, which in turn generates PCI_RST_L (if CFG_RSTDIR = 1 is set, IXP is the PCI central function).

The reset generated by the Watchdog timer will not reset each of the functions in the IXP2400 if Watchdog reset enable bit is not set (IXP_RESET0[22]=0). Instead of reset each of the functions in the IXP2400, It will generate PCI interrupt (PCI_INTA_L) to external if *PCI Outbound Interrupt Mask Register[3] to 0*. It will also set *PCI Outbound Interrupt Status Register[3] to 1*.

The RESET_OUT# signal remain asserted until deasserted by the Intel XScale® core. It deasserts the signal by writing bit [15] of the IXP_RESET0 register.

Table 159. Watchdog Timer Reset

| XPI WD Timer Expired | WatchDog Reset Enable IXP RESET0[24] | PCI Iutbound Interrupt Mask[3] |
|----------------------|---|--|
| PCI Interrupt | Software can set the IXP_RESET0[24] to 0 if IXP is non central function or any cases [not grammatical but I don't know what it means.] | <ol style="list-style-type: none"> 1. WD Timer History bit will set to 1. 2. PCI Interrupt will be generated if PCI outbound Interrupt Mask[3] set to 0. 3. PCI outbound Interrupt status[3] set to 1 |
| Reset IXP | Software can set the IXP_RESET0[24] to 1 if IXP is PCI Host & Central Function or any cases [not grammatical but I don't know what it means.] | <ol style="list-style-type: none"> 1. WD Timer History bit will set to 1. 2. PCI Interrupt will not be generated. 3. PCI outbound Interrupt status[3] set to 0. |

10.3.4 Software-Initiated Reset

The Intel XScale® core or an external PCI bus master can reset specific functions in the IXP2400 by writing to the IXP_RESET0 and IXP_RESET1 register. All the individual microengines or specific units can be reset.

1. If software write IXP_RESET0[16] to 1(Reset All), Reset Unit need to hold 512 cycles then set the each of reset unit registers to 1 (include PCI).During this 512 cycles PCI should be in the PCI bus IDLE state, once PCI bus get IDLE PCI unit need to reset all the PCI blocks. IXP is bus parked during soft reset, IXP need to drive these I/O devices PCI_AD[31:0], PCI_BE[3:0], and PCI_PAR to known values even in soft reset mode. otherwise all these I/O devices to tristates.

2. If software write IXP_RESET0[1] to 1(Reset PCI only), PCI unit need to reset all the PCI blocks right away without PCI bus get idle. There is no 512 idle cycles in reset period. All these I/O devices to tristates regardless of bus parking.

The RESET_OUT# signal remain asserted until deasserted by the Intel XScale® core. It deasserts the signal by writing bit [15] of the IXP_RESET0 register.

10.3.5 Strap Pins

The following are the strap pins required for reset. These strap pins determine the initialization sequence.

Table 160. Strap Pins required for IXP2400

| Signal | Name | Description | External Pin |
|-------------------|------|--|--------------|
| CFG_RSTDIR | PCF0 | <p>PCI central function pin:</p> <ul style="list-style-type: none"> 1–IXP is supporting central functions <i>PCI_RST_L</i> is an output (<i>SYS_RST_L</i> is input). <i>PCI_REQ64</i> is an output.(drive low during PCI reset) <i>PCI AD[31:0]</i>, <i>PCI_BE[3:0]</i>, and <i>PCI_PAR</i> drive to Low during PCI reset. After PCI reset all these I/O lines to tristates unless IXP bus parked. <i>PCI AD[63:32]</i>, <i>BE[7:4]</i> and <i>PAR64</i> during PCI reset or after PCI reset all these I/O lines to tristates. <p>NOTE: If the PCI bus is 32 bits wide, the board must support external pull up of I/O lines.</p> 0–External PCI is supporting central functions <i>PCI_RST_L</i> is an input. Both <i>PCI_RST_L</i> and <i>SYS_RST_L</i> are inputs; tie both reset lines together. <i>PCI_REQ64</i> is an input. <i>PCI AD[31:0]</i>, <i>PCI_BE[3:0]</i>, and <i>PCI_PAR</i> during PCI reset or after PCI reset all these I/O to tristates. During PCI reset or after PCI reset <i>PCI AD[63:32]</i>, <i>BE[7:4]</i> and <i>PAR64</i> I/O are tristate. <p>NOTE: If PCI bus is 32 bits wide, the board need to support external pull up of I/O lines.</p> <p>This pin is stored at <i>XSC[31]</i> (<i>XSCALE_CONTROL</i> Register) at the trailing edge of reset.</p> | Explicit Pin |
| CFG_PROM_BOOT | PCF2 | <p>PCI Prom Boot pin (BOOT_PROM)</p> <ul style="list-style-type: none"> 1–IXP will boot from PROM 0–IXP will boot from DRAM initialized by PCI Host <p>This pin is stored at <i>XSC[29]</i> (<i>XSCALE_CONTROL</i> register) at the trailing edge of reset.</p> | GPIO[0] |
| CFG_PCI_BOOT_HOST | PCF3 | <p>PCI BOOT HOST</p> <ul style="list-style-type: none"> 1–Intel XScale® core will configure the PCI system 0–Intel XScale® core will not configure the PCI system <p>This pin is stored at <i>XSC[28]</i> (<i>XSCALE_CONTROL</i> register) at the trailing edge of reset. If <i>CFG_PCI_BOOT_HOST</i> set to 1 then <i>CFG_RSTDIR</i> must set to 1 (central function).</p> | GPIO[1] |

Table 160. Strap Pins required for IXP2400

| Signal | Name | Description | External Pin |
|-------------------|------|---|--------------|
| CFG_PCI_ARB | PCF1 | PCI Arbiter Pin <ul style="list-style-type: none"> • 1—IXP will do the PCI arbitration function • 0—IXP will not do the PCI arbitration function If <i>CFG_PCI_ARB</i> set to 1 then <i>CFG_RSTDIR</i> must set to 1 (central function). | GPIO[2] |
| CFG_PCI_DWIN[1:0] | DWIN | DRAM BAR Window <ul style="list-style-type: none"> • 11—DRAM BAR size: 1024 MByte • 10—DRAM BAR size: 512 MByte • 01—DRAM BAR size: 256 MByte • 00—DRAM BAR size: 128 MByte This is part of the <i>PCI_DRAM_BAR</i> register. | GPIO[4:3] |
| CFG_PCI_SWIN[1:0] | SWIN | SRAM BAR Window <ul style="list-style-type: none"> • 11—SRAM BAR size: 256 MByte • 10—SRAM BAR size: 128 MByte • 01—SRAM BAR size: 64 MByte • 00—SRAM BAR size: 32 MByte This is part of the <i>PCI_SRAM_BAR</i> register. | GPIO[6:5] |

Table 161. Legal Combinations of the Strap Pin Options^a

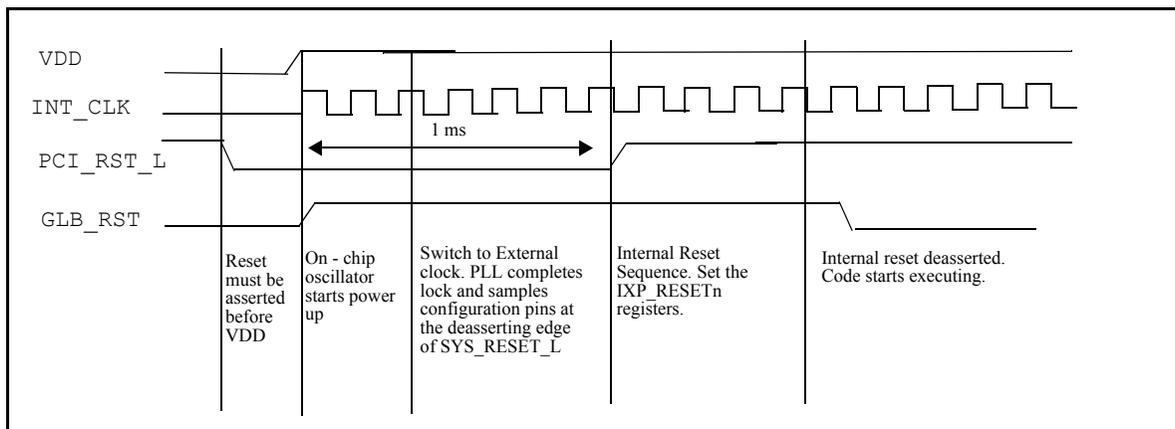
| Support for Combination | CFG_PCI_HOST | CFG_PCI_Arbiter | CFG_PCI_RSTDIR (Central function) | CFG_PROM_BOOT |
|-------------------------|--------------|-----------------|-----------------------------------|---------------|
| ok | 0 | 0 | 0 | 0 |
| ok | 0 | 0 | 0 | 1 |
| ok | 0 | 0 | 1 | 1 |
| <i>Not supported</i> | 0 | 1 | 0 | X |
| ok | 0 | 1 | 1 | 1 |
| <i>Not supported</i> | 1 | 0 | 0 | X |
| ok | 1 | 0 | 1 | 1 |
| <i>Not supported</i> | 1 | 1 | 0 | X |
| ok | 1 | 1 | 1 | 1 |

a. *PCI_Host* and *PCI_Arbiter* must be central function.

10.3.6 Power Up Reset Sequence

The basic sequence for reset is shown in [Figure 114](#).

Figure 114. Reset Sequence



When the system is powered up, the SYS_RESET_L must stay asserted (L) for at least 1ms after VDD and SYS_CLK have reached their proper DC and AC levels. All the I/O must de-asserted their output within 2 cycles.

When the system is powered up, a ring oscillator clock will be sent to all 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 the internal RC detect circuit is active, the clock unit will switch from using the ring oscillator to using the PLL clock. Throughout this time the SYS_RESET_L is asserted. After the SYS_RESET_L pin transitions to an inactive state, the internal reset signal (GLB_RST) will remain active for a number of clocks to allow the IXP2400 to achieve a clean reset state. The GLB_RST signal will be used to reset the IXP_RESETE₀ and IXP_RESETE₁ registers. During warm reset, the PLL is already locked and the internal reset sequence is initiated on detecting a SYS_RESET_L signal.

The reset sequence shown above is the same in the case when reset happens through the PCI_RST_L signal and CFG_RSTDIR is 0.

Once in operation, if watchdog timer expires with watchdog timer enable bit ON, reset pulse from the watchdog timer logic resets the IXP_RESETE_n registers and in turn causes entire chip to be reset.

The IXP2400 has the following power supplies:

1. VCC3.3 3.3V power supply for the Media Switch Fabric interface, PCI, GPIO, SlowPort and Misc.
2. VCC and VCCA 1.3V power supply for the Core and for the PLL
3. VCC2.5 2.5V power supply for the DDR DRAM
4. VCC1.5 1.5V power supply for the QDR SRAM
5. D_Vref 1.25V for the DDR DRAM
6. Sn_Vref 0.75V for QDR SRAM channel 0, and channel 1

The power supplies for the IXP2400 should be brought up in a controlled sequence. The delay between the power-up of the power supplies should be 5 ms or less.

1. The 3.3V must be brought up before the 1.3V
2. The 1.3V must be brought up before the 1.5V and 2.5V
3. The 1.5V must be brought up before or at the same time as the 0.75V
4. The 2.5V must be brought up before or at the same time as the 1.25V

10.3.7 Power-Down Sequence

All the power supplies should be brought down simultaneously. If the user cannot power down all the supplies simultaneously, the Power-down sequence is recommended to be the reverse order of the Power-up sequence shown in [Section 10.3.6](#).

Note: Please see the *IXMB2400 Base Board Design Guide* for the timing and algorithm of power sequencing.

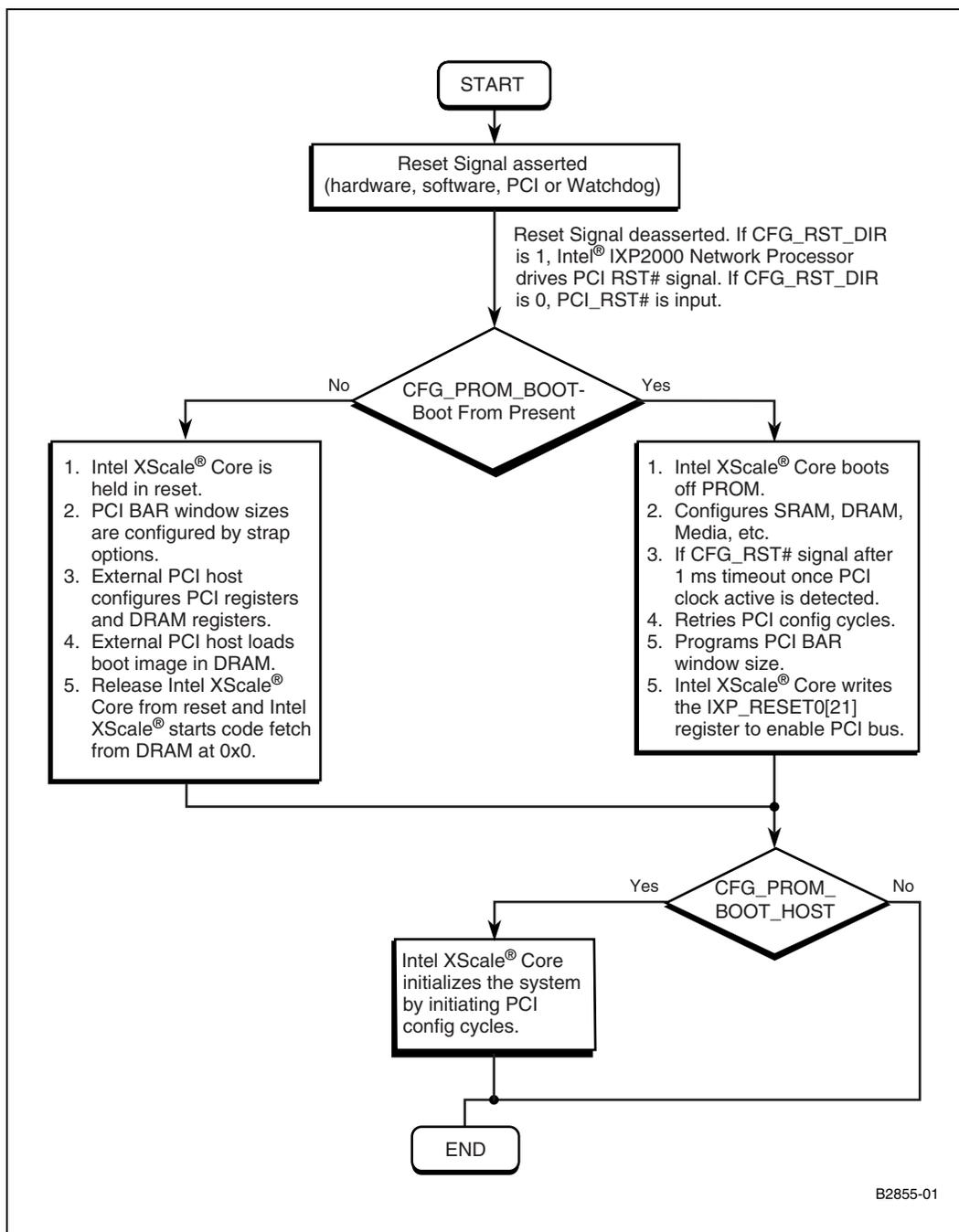
10.4 Reset Register

IXP System Reset Register consists of two 32-bit registers: IXP_RESET0 [31:0] and IXP_RESET1 [31:0]. IXP_RESET0 [31:0] is used to reset everything except microengines. IXP_RESET1 [31:0] is used to reset the microengines. Bits from these registers going to different modules should be synchronized when they are going to different frequency domain. These bits are read/write by both PCI host and the Intel XScale® core. Please see the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for additional IXP System Reset Register information.

10.5 Boot Mode

Upon deassertion of the external reset signals, the internal reset signals in the IXP_RESET0 registers are cleared based on the pin strapping options. [Figure 115](#) outlines the initialization sequence.

Figure 115. IXP2400 Initialization Sequence



As shown in Figure 115, IXP2400 can boot in following two boot modes:

- Flash ROM
- PCI Host Download

10.5.1 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 XScale gasket to decide where to forward address “0” from the XScale core when the 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 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 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" any time 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 XScale core also sets different BARs inside PCI unit to define memory requirements for different windows.

The 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 XScale core will behave as boot host and configure the PCI system.

10.5.2 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 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 DU_CONTROL, DU_CONTROL2 and DU_INIT 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 XScale gasket directly so that gasket knows how to transform address 0 from the XScale core. After writing “1” at “Flash Alias Disable” bit, host removes reset from the XScale core by writing “0” in bit [0] of IXP_RESET0 register. The XScale core starts booting from address 0, which is now directed by the gasket to DRAM.

10.6 Reset Strategy for Different Sections in IXP2400

Table 163 defines the strategy for resetting IXP2400 for logic not directly affected by IXP_RESETn registers. Table 164 and Table 165 defines the strategy for resetting IXP2400 for logic affected by IXP_RESETn registers. The contents of Table 163, Table 164, and Table 165 are described in Table 162.

Table 162. Description of the Content of Table 163 through Section 165

| Column | Contents |
|-------------|--|
| Unit | Name of the unit where reset is applied |
| Reset Bit # | Bit number from IXP_RESETn register that is used to reset this unit ^a |
| When Set | Conditions to set this bit |
| When Reset | Conditions to reset this bit |
| Comments | Special behavior |

a. Not applicable to Table 163.

Table 163. Resetting^a IXP2400 Strategy—Logic Unrelated to IXP_RESETn

| Unit | When Set Reset | When Clear Reset | Comment |
|---|----------------------|-----------------------------|--|
| PLL (Core_pll, MSF_pll, PCI_pll) | Hard Reset Active | Hardware Reset De-Active | PCI_RST_L (if CFG_RSTDIR =1) and SYS_RESET_L are combined to generate a HARD RESET for the PLL logic. Based on this input reset, IXP_RESET0 and IXP_RESET1 registers are reset causing the entire chip to be reset. |
| Stepping Stone Logic | Hard Reset Active | Hardware Reset De-Active | |

a. Reset definitions:
Hard Reset: combine *PCI_RST_L* (if *CFG_RSTDIR* =1) and *SYS_RESET_L*.
Soft Reset: combine *Watchdog timer*, *Reset_All_Reg* (*IXP_RESET0*[16] =1 and each reset bit set to 1).

Table 164. Resetting IXP2400 Scheme—Logic Related to IXP_RESET0[31:0]

| Unit | Reset Bit # | When Set Reset | When Clear Reset | Comment |
|-----------|-------------|--|--|---|
| XScale | [0] | Hard Reset Active or Soft Reset Active | <ul style="list-style-type: none"> CFG_PROM_BOOT=1 After Reset it will automatically take about 256 cycles to clear the reset bit. CFG_PROM_BOOT=0 This bit need to be cleared by software. | |
| PCI Unit | [1] | Hard Reset Active or Soft Reset Active | <ul style="list-style-type: none"> CFG_PROM_BOOT=1 After Reset happen it need to be cleared by XScale CFG_PROM_BOOT=0 After Reset happen it will automatically take about 256 cycles to clear the reset bit. | |
| PCIRST | [2] | See Comment | See Comment | <p>If the CFG_RSTDIR pin is asserted high, PCI_RST_L is an output and PCIRST is:</p> <ul style="list-style-type: none"> 1—IXP2400 asserts the PCI_RST_L pin 0—IXP2400 does not assert the PCI_RST_L pin <p>The Intel XScale® core clears this bit after a reset to release the PCI bus from reset.</p> <p>If the CFG_RSTDIR pin is asserted low, PCI_RST_L is an input and PCIRST is:</p> <ul style="list-style-type: none"> 1—reset was caused by a PCI device 0—reset was not caused by a PCI device <p>The Intel XScale® core can read this bit to determine whether a PCI device reset the IXP2400.</p> |
| SRAM[1:0] | [4:3] | Hard Reset Active or Soft Reset Active | After Reset it will automatically take about 256 cycles to clear the reset bit. | |
| SRAM[3:2] | [6:5] | Reserved | | Bits[6:5] are reserved. |

Table 164. Resetting IXP2400 Scheme—Logic Related to IXP_RESET0[31:0]

| Unit | Reset Bit # | When Set Reset | When Clear Reset | Comment |
|-------------------|-------------|--|---|---|
| Media[0] | [7] | Hard Reset Active or Soft Reset Active | This bit need to be cleared by software. Wait until all the MSF PLL are locked then clear the Media reset bit. | |
| Media[10:8] | [10:8] | Reserved | This bit need to be cleared by software. | Bits[10:8] are reserved right now. |
| DRAM[0] | [11] | Hard Reset Active or Soft Reset Active | After Reset happen it will automatically take about 256 cycles to clear the reset bit. | |
| DRAM[3:1] | [14:12] | Reserved | | Bits[14:12] are reserved right now. |
| EXRST | [15] | Hard Reset Active or Soft Reset Active | This bit need to be cleared by software. | |
| RSTALL | [16] | Soft Reset Active | When RSTALL is set, Intel XScale® core will idle ^a 512 cycles then it resets the whole unit except PLLs and some registers. After Reset it will automatically take about 256 cycles to clear the reset bit. | When this bit is set, both IXP_RESET registers are reset. |
| SHaC ^b | [17] | Hard Reset Active or Soft Reset Active | After Reset it will automatically take about 256 cycles to clear the reset bit. | |
| CMD_ARBITER | [18] | Hard Reset Active or Soft Reset Active | After Reset it will automatically take about 256 cycles to clear the reset bit. | |
| SBUS_ARBITER | [19] | Hard Reset Active or Soft Reset Active | After Reset it will automatically take about 256 cycles to clear the reset bit. | |

Table 164. Resetting IXP2400 Scheme—Logic Related to IXP_RESET0[31:0]

| Unit | Reset Bit # | When Set Reset | When Clear Reset | Comment |
|-------------------|-------------|--|---|--|
| DBUS_ARBITER | [20] | Hard Reset Active or Soft Reset Active | After Reset it will automatically take about 256 cycles to clear the reset bit. | |
| INIT_COMP | [21] | Hard Reset Active or Soft Reset Active | See comment | When CFG_PROM_BOOT (BOOT_PROM) = 0 INIT_COMP is set when GLB_RST is deasserted When CFG_PROM_BOOT (BOOT_PROM) = 1 INIT_COMP is set when Intel XScale® core sets |
| WatchDog_Reset_En | [24] | Set By Software | Reset By Software. | When WatchDog_Reset_En is: <ul style="list-style-type: none"> 0—Watchdog reset will trigger the PCI interrupt to external PCI host 1—Watchdog reset will trigger the soft reset (set IXP_RESET0[16] RSTALL to 1), then it will reset all units after 512 cycles |

- a. Waiting for PCI bus to IDLE and PCI unit to start PCI unit Reset.
b. SHaC is Scratch, Hash and CSR.

Table 165. Resetting IXP2400 Scheme—Logic Related to IXP_RESET1[31:0]

| Unit | Reset Bit # | When Set Reset | When Clear Reset | Comment |
|--------------|------------------|--|--|---------|
| Microengines | [3:0] [19:16] | Hard Reset Active or Soft Reset Active | This bit need to be cleared by software. | |

10.7 Initialization

The boot sequence task must be performed by the IXP2400 after reset for proper processor functioning. The boot sequence tasks configure IXP2400 resources to a predetermined state by writing values to certain registers. Some of these 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 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; specifically the UART and the PCI Interface.

The configuration tasks must be performed in the following sequence:

1. Configure XPI Interface to access PROM; if *CFG_PROM_BOOT (BOOT_PROM)* is present the following registers should be programmed:

- SP_CCR: configures the clocks for the SlowPort; initially these clocks start at some default value which may not be optimal
 - SP_WTC: program this register for PROM interface to define proper write timing
 - SP_RTC: program this register for PROM interface to define proper read timing
 - SP_FAC: defines the address size of flash memory device used
 - SP_FRM: defines the data width of the read back from the flash memory
2. Configure *Clock Ratio CCR* and *MCCR CSR*; to define the operating frequency of SRAM and DRAM interface the following registers define the operation of stepping stone logic and must be initialized:
 - CCR: Clock Control CSR to define the frequency of SRAM and DRAM channels
 - MCCR Media Clock Control CSR to define the frequency of RX and TX
 - Set up each of the MSF clock ratios (each of the MCCR[MSF_CLKCFG] bits)
 - Disable the PLL bypass mode (MCCR[MSF_BY_PASS_SEL] = 0)
 - Turn on PLL (MCCR[MSF_POWERDOWN] = 0)
 - Wait for MSF PLL lock (MCCR[MSF_PLL_LOCK] = 1)
 3. Release from Reset; after reset, units not coming out of reset automatically are brought out of reset by programming the IXP_RESET0 and IXP_RESET1 registers. For a description of what units come out of reset automatically, please refer to [Section 10.6](#) of this document.
 4. Configure the SRAM controller using the following registers:
 - 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
 The following registers are application specific and must be programmed if required.
 - Q_Array_Entry_nn_Low: To access EOP, Cell Count and Head Fields. “nn” is entry #
 - Q_Array_Entry_nn_Med: To access Tail Fields. “nn” is entry #
 - Q_Array_Entry_nn_High: To access Q Count. “nn” is entry #
 5. Configure the in-use DRAM channels through a sequence of register writes:
 - a. DU_CONTROL
 - b. DU_CONTROL2
 - c. DU_INIT
 6. Configure the Memory Mapped Unit, Cache, and Buffer by configuring the following register: XScale Coprocessor 15—CONTROL_CP15
 7. Configure PCI; if CFG_PROM_BOOT (BOOT_PROM) is not present, loading the boot image into DRAM by the PCI host is required. To do this,
 - a. The following registers should be set to their required value *on the de-asserting edge of reset*:
 - PCI_DRAM_BAR: strap pins define the window size
 - PCI_SRAM_BAR: strap pins define the window size

—PCI_CSR_BAR

- b. IXP_RESET0[21] should be set to “1”
- c. 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.
- d. If CFG_PROM_BOOT (BOOT_PROM) is present, configure the following four registers:
 - SRAM_BASE_ADDR_MASK:
 - DRAM_BASE_ADDR_MASK:
 - PCI_Command_Status:
 - PCI_Write_Address_Ext:

Note: In this mode, code jumps to normal flash location and then disables the “map flash to zero” feature. If PCI_CFN[3](BOOT_HOST) is not true, then the external PCI host will configure IXP2400 PCI interface based on its memory requirements. If PCI_CFN[3](BOOT_HOST) is true, then IXP2400 will program the PCI interface.

8. Configure Serial Port, if required by configuring the following registers:

- UART_DLRH
- UART_DLRL
- UART_IER
- UART_FCR
- UART_LCR

9. Configure Media

- Initialize the MSF
- Enable MSF block (via MSF_Rx_Control[Rx_En] or MSF_Tx_Control[Tx_En])

Performance Monitor Unit

11

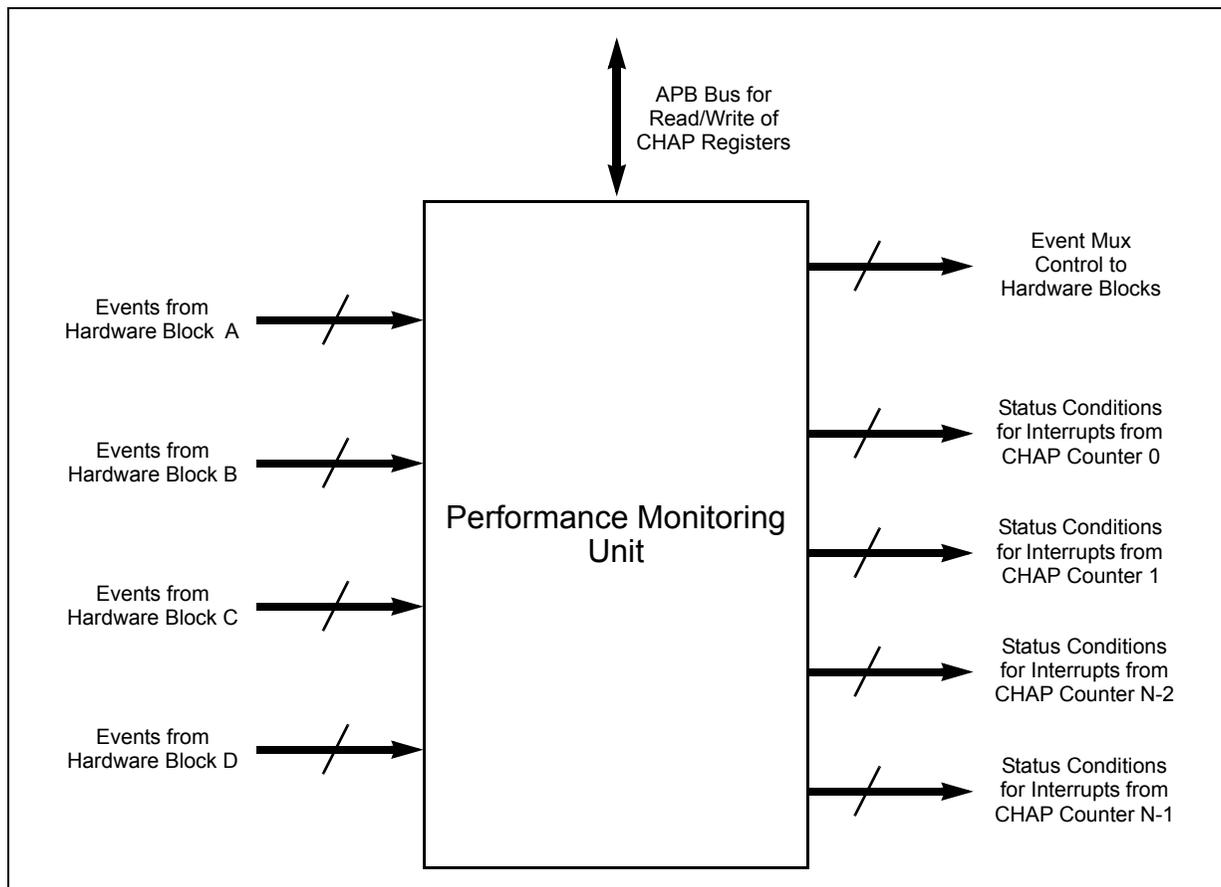
11.1 Introduction

Performance Monitor Unit is a hardware block consisting of counters and comparators which may be programmed and controlled using a set of configured registers to monitor and to fine tune performance of different hardware units in the IXP2400. The total number of such counters needed is determined based on the different events and functions that need to be monitored concurrently. Observation of such events on chip is used for statistical analysis, uncovering bottlenecks and to tune the software to fit the hardware resources.

11.1.1 Motivation for Performance Monitors

For a given set of functionality, a measure of performance is very important to make decisions on feature sets to be supported and to tune the embedded software on chip. An accurate estimate of latency and speed in hardware blocks enables firmware and software designers to understand the limitations of the chip and make prudent judgments of their software architecture. The current generation IXP1200 processors do not provide any performance monitor hooks. Since IXP2400 processors are targeted for high performance segments (OC48 and above), the need for tuning the software to get the most of out the hardware resources becomes extremely critical. The performance monitors provide valuable insight into the chip by providing real time data on latency and utilization of various resources. These monitors also enable hardware architects to make effective design trade-off in future generation of the product by uncovering implementation glass jaws, flaws and limitations. See [Figure 116](#).

Figure 116. Performance Monitor Interface Block Diagram



11.1.2 Motivation for Choosing CHAP Counters

The Chipset Hardware Architecture Performance (CHAP) counters enable gathering statistics of internal hardware events in real-time. This implementation provides users with direct event counting and timing for performance monitoring purposes, provides enough visibility into the internal architecture to perform utilization studies and workload characterization and can also be used for chipset validation, higher-performing future chipset, and applications tuned to the current chipset. The goal is that this will benefit both internal and external hardware and software development. Primary motivation for selecting the CHAP architecture for use in the IXP2400 product family is that it has been designed and validated in several Intel desktop chipset and the framework also provides a set of software suite which may be reused with a very limited modification.

11.1.3 Functional Overview of CHAP Counters

At the heart of the CHAP counters functionality are counters, each with associated registers. Each counter has a corresponding command, event, status, and data register. The smallest implementation will have 2 counters, but if justified for a particular product, this architecture can

support many more counters. The primary consideration is available silicon area. The memory-mapped space currently defined can accommodate registers for 256 counters. It could be configured for more, but that is beyond what is currently practical.

Signals representing events from throughout the chip are routed to the CHAP unit. Software can select events that will be recorded during a measurement session. The number of counters in an implementation defines the numbers of events that can be recorded simultaneously. Software and hardware events can control the starting, stopping, and sampling of the counters. This can be done in a time-based (polling) or event-based fashion. Each counter can be increment or decrement by different events. In addition to simple counting of events the unit can provide data for histograms, queue analysis, and conditional event counting (example: How many times did event A happen before the first event B took place).

When a counter is sampled, the current value of the counter is latched into the corresponding data register. The command, event, status, and data registers are accessible via standard APB memory mapped registers in order to facilitate high-speed sampling.

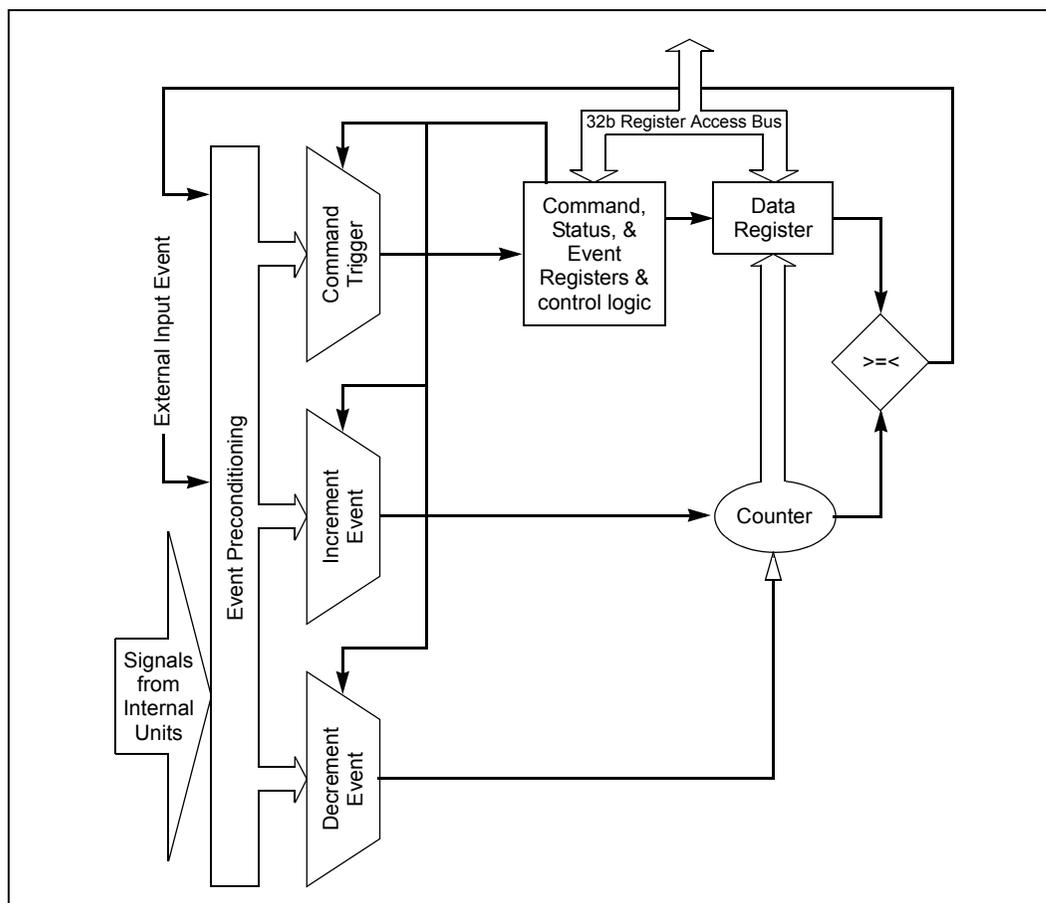
Two optional external pins allow for external visibility and control of the counters. The output pin signals that one of the following conditions generated an interrupt from any one of the counters:

- A programmable threshold condition was true,
- A command was triggered to begin
- A counter overflow or underflow occurred.

The input pin allows an external source to control when a CHAP command is executed.

Figure 117 represents a single counter block. The muxes, registers, and all other logic are repeated for each counter that is present. There is a threshold event from each counter block that feeds into each mux.

Figure 117. Block Diagram of a Single CHAP Counter



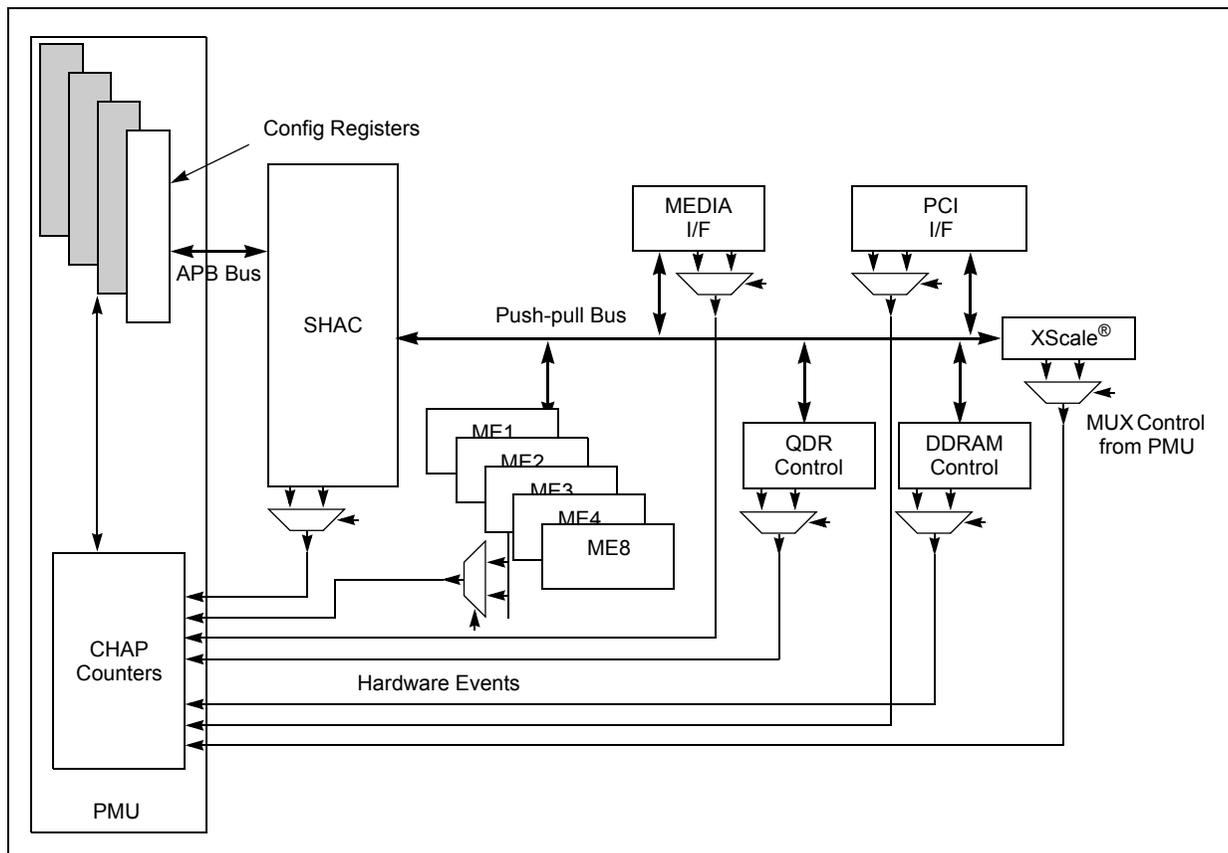
11.1.4 Basic Operation of Performance Monitor Unit

At power up, the XScale core invokes performance monitoring software code. PMU software has the application code to generate different types of data such as histograms and graphs. It also has device driver to configure and read data from PMU in IXP2400. This software programs the configuration registers in the PMU block to perform a certain set of monitoring and data collection. PMU CHAP counters execute the commands programmed by XScale and they collect various types of data such as latency and counts. Upon collection it triggers an interrupt to XScale to indicate the completion of monitoring.

XScale either periodically monitors the PMU registers or waits for an interrupt to collect the observed data. XScale uses the APB bus to communicate with the PMU configuration registers.

Figure 118 represents a block diagram of IXP2400 and Performance Monitor Unit's (PMU) in relation to other hardware blocks in the chip.

Figure 118. Basic Block Diagram of IXP2400 with PMU



11.1.5 Definition of CHAP Terminology

| | |
|------------------|--|
| Duration Count | The counter is increment for each clock for which the event signal is asserted logic high. |
| MMR | Memory Mapped Register |
| OA | Observation Architecture. The predecessor to CHAP counters that facilitates counting of hardware events. |
| Occurrence Count | The counter is increment each time a rising edge of the event signal is detected. |
| Preconditioning | Altering a design block signal that represents an event such that it can be counted by the CHAP unit. The most common preconditioning is likely to be a 'one-shot' in order to be able to count occurrences. |
| RO (register) | Read Only. If a register is read only, writes to this register location have no effect. |
| R/W (register) | Read/Write. A register with this attribute can be read and written |
| WO (register) | Write Once. Once written, a register with this attribute becomes Read Only. This Register can only be cleared by a Reset. |

WC (register) Write Clear. A register bit with this attribute can be read and written. However, a write of 1 clears (sets to 0) the corresponding bit and a write of 0 has no effect.

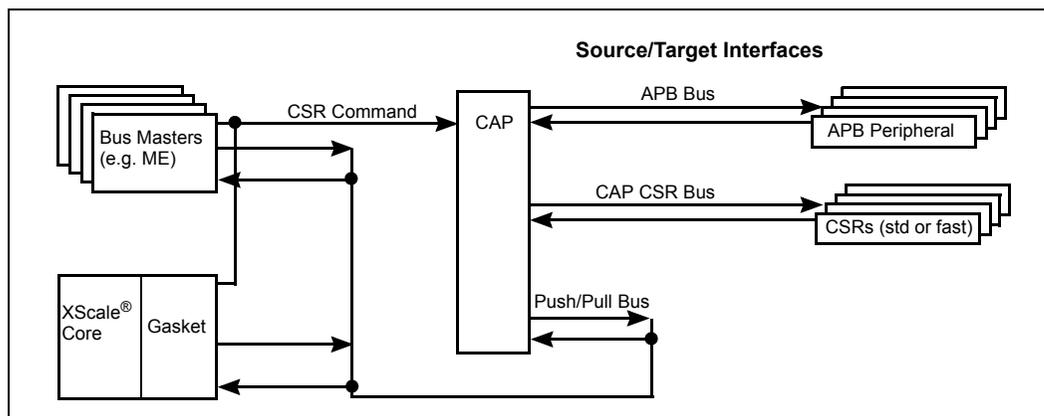
11.2 Interface and CSR Description

CAP is a standard logic block provided as part of the IXP Chassis that provides a method of interfacing to the following:

ARM Advanced Peripheral Bus (APB): This bus supports standard APB peripherals such as PMU, UART, Timers and GPIO as well as CSRs that are not required to be accessed by the MEs.

As shown in [Figure 119](#), CAP uses three bus interfaces to support these modes. CAP supports a target ID of 0101 which ME assemblers should identify as a CSR instruction.

Figure 119. CAP Interface to APB Bus



[Table 166](#) shows the XScale and ME 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, an ME would execute a `csr[read]` instruction and XScale would execute a `Load (ld)` instruction. Data is then moved between the CSR and the XScale/ME by first reading the CSR via the APB bus and then writing the result to the XScale/ME via the Push Bus.

Table 166. APB Bus Usage

| Accessing | Read Operation | Write Operation |
|----------------|--|--|
| APB Peripheral | Access Method: <ul style="list-style-type: none"> ME: <code>csr[read]</code> XScale®: <code>ld</code> | Access Method: <ul style="list-style-type: none"> ME: <code>csr[write]</code> XScale: <code>st</code> |
| | Bus Usages: <ul style="list-style-type: none"> Read source: APB bus Write dest: Push bus | Bus Usages: <ul style="list-style-type: none"> Read source: Pull Bus Write dest: APB bus |

11.2.1 APB Bus Peripheral

The Advanced Peripheral Bus (APB) is part of the Advanced Micro controller Bus Architecture (AMBA) hierarchy of buses that is optimized for minimal power consumption and reduced design complexity. PMU needs to operate as an APB peripheral interfacing with rest of the chip via APB bus. PMU needs to have APB bus interface unit, which can perform a APB bus reads and writes to enable data transfer to and from the PMU registers.

11.2.2 CAP Description

11.2.2.1 Selecting the Access Mode

The CAP selects the appropriate access mode based on the COMMAND and ADDRESS fields from Command Bus.

11.2.2.2 PMU CSR

Please refer to *Intel IXP2400/IXP2800 Programmer's Reference Manual*.

11.2.2.3 CAP Writes

For an APB write, CAP arbitrates for the S_Pull_Bus, pulls the write data from the source identified in PP_ID (either a ME transfer register or XScale core write buffer), and puts it into the CAP Pull Data FIFO. It then drives the address and writes data on to the appropriate bus. CAP CSRs locally decode the address to match their own. CAP generates a separate APB devices select signal for each CAP 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).

CAP supports write operations with burst counts greater than 1. 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.

11.2.2.4 CAP Reads

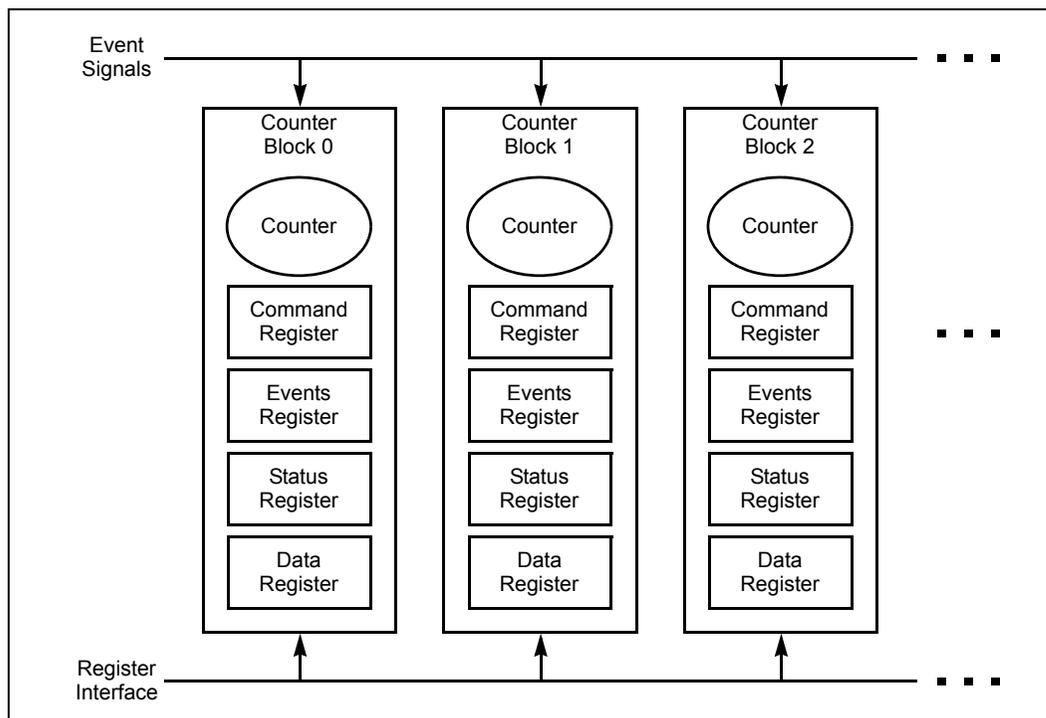
For an APB read, CAP drives the address, write, select, and enable signals, waits for the acknowledge signal (APB_RDY_H) from APB device, For a CAP CSR read, CAP drives the address, which controls a tree of multiplexors to select the appropriate CSR. CAP then waits for the acknowledge signal (CAP_CSR_RD_RDY). When the data is returned, CAP then puts the read data into the Push Data FIFO, arbitrates for the S_Push_Bus, and then the Push/Pull Arbiter pushes the data to the destination identified in PP_ID.

11.2.3 Configuration Registers

Because the CHAP unit resides on the APB bus, the offset associated with each of these registers is relative to the Memory Base Address that configuration software will set in the PMUADR register.

Each counter has one command, one event, one status, and one data register associated with it. Each counter is “packaged” with these four registers in a “counter block”. Each implementation selects the number of counters it will implement, and therefore how many counter blocks (or slices) it will have. These registers are numbered 0 through N - 1 where N represents the number of counters - 1. See Figure 120.

Figure 120. Conceptual Diagram of Counter Array



11.3 Performance Measurements

There are several measurements that could be made on each of the hardware blocks. These measurements together would enable improvements in hardware and software implementation and architectural issues. The following sections discuss different blocks and their associated performance measurement events.

11.3.1 XScale®

11.3.1.1 DRAM Read Head of Queue Latency Histogram

11.3.1.1.1 Description

The Intel XScale® core generates a read command or write command to the DRAM primarily to either push or pull data of the DDRAM. These commands are scheduled to the DRAM through the push pull arbiter through a command FIFO in the gasket. DRAM read head of queue enables the PMU to monitor when the read and write commands posted by XScale in the gasket gets fetched and delivered to DDRAM.

11.3.1.2 SRAM Read Head of Queue Latency Histogram

11.3.1.2.1 Description:

The XScale core generates a read command or write command to the SRAM primarily to either push or pull data of the SRAM. These commands are scheduled to the SRAM through the push pull arbiter through a command FIFO in the gasket. SRAM read head of queue enables the PMU to monitor when the read and write commands posted by XScale in the gasket gets fetched and delivered to SRAM.

11.3.1.3 Interrupts

11.3.1.3.1 Description:

Number of interrupts seen

Histogram of time between interrupts

11.3.2 Microengines

11.3.2.1 Command Fifo number of commands

11.3.2.1.1 Description:

This statistics will give the number of the commands issued by the ME in a particular period of time. It is also can count each different threads.

11.3.2.2 Control Store Measures

11.3.2.2.1 Description:

Count time between two micro store locations (locations can be set by instrumentation software).

Histogram time between two microstore locations (locations can be set by instrumentation software)

11.3.2.3 Execution Unit Status

11.3.2.3.1 Description:

Histogram of stall time. Histogram of aborted time. Histogram of swapped out time. Histogram of idle time.

11.3.2.4 Command Fifo Head of Queue Wait Time Histogram (Latency)

11.3.2.4.1 Description:

This is to measure the latency of a command, which is at the head to the queue and is waiting to be send out to the destination over the Chassis.

11.3.3 SRAM

11.3.3.1 SRAM Commands

11.3.3.1.1 Description:

A count of SRAM commands received. These are maskable by command type such as put and get.

11.3.3.2 SRAM Bytes, Cycles Busy

11.3.3.2.1 Description:

This measurement describes the number of bytes transferred and SRAM busy time.

11.3.3.3 Queue Depth Histogram

11.3.3.3.1 Description:

This measurement analyses the different queues such as Ordered, priority, push queue, pull queue, read lock fail, HW queues and provides the information on utilization.

11.3.4 DDRAM

11.3.4.1 DRAM Commands

11.3.4.1.1 Description:

This measurement lists the total commands issued to the DRAM and they can be counted based on command type and error type.

11.3.4.2 DRAM Bytes, Cycles Busy

11.3.4.2.1 Description:

This measurement indicates the DRAM busy time and bytes transferred.

11.3.4.3 Maskable by Read/Write, ME, PCI or XScale®

11.3.4.3.1 Description:

This measurement indicates different accesses that are initiated to the DRAM. These measurement s could be for all the accesses to the memory or can be masked using a specific source such as PCI, XScale, or ME. This can further measured based on read or write cycles.

11.3.5 Chassis/Push-Pull

11.3.5.1 Command Bus utilization

11.3.5.1.1 Description:

This statistics will give the number of the command request issued by the different Master in a particular period of time.

This measurement also indicates how long it take to issue the grant from request being issued by the different Master.

11.3.5.2 Push and Pull Bus Utilization.

11.3.5.2.1 Description:

This measurement keep track of the number of accesses issued and how long it take to send the data to destination.

11.3.6 Hash

11.3.6.1 Number of Accesses by Command type

11.3.6.1.1 Description:

This measurement indicates the number of hash accesses issued and this count is maskable based on command type.

11.3.6.2 Latency of Histogram

11.3.6.2.1 Description:

This monitors the latency through each of the HASH queues.

11.3.7 Scratch

11.3.7.1 Number of Accesses by Command type

11.3.7.1.1 Description:

This measurement indicates the number of Scratch accesses issued and this count is maskable based on command type.

11.3.7.2 Number of bytes transfer

11.3.7.2.1 Description:

This measurement indicates total number of bytes transferred to or from Scratch.

11.3.7.3 Latency of Histogram

11.3.7.3.1 Description:

This measurement indicates the latency of performing read or write from the Scratch. Latency is command executions may also be measured.

11.3.8 PCI

11.3.8.1 Master Accesses

11.3.8.1.1 Description:

This statistics will give the number of Master accesses that were generated by the PCI blocks. This measurement could be counted based on individual command type.

11.3.8.2 Slave Accesses

11.3.8.2.1 Description:

This statistics will give the number of Slave accesses that were generated by the PCI blocks. This measurement could be counted based on individual command type.

11.3.8.3 Master/Slave Read Byte Count

11.3.8.3.1 Description:

This statistics will give total the number of bytes of data that were generated by the PCI Master/Slave reads access. This measurement could be counted based on individual command type.

11.3.8.4 Master/Slave Write Byte Count

11.3.8.4.1 Description:

This statistics will give total the number of bytes of data that were generated by the PCI Master/Slave write accesses. This measurement could be counted based on individual command type.

11.3.8.5 Burst Size Histogram

11.3.8.5.1 Description:

This statistics will give a histogram of number of various burst size.

11.3.9 Media Interface

11.3.9.1 TBUF Occupancy Histogram

11.3.9.1.1 Description:

This measurement shows the occupancy rate at different depth of FIFO. This can help in better utilization of TBUF.

11.3.9.2 RBUF Occupancy Histogram

11.3.9.2.1 Description:

This measurement shows the occupancy rate at different depth of FIFO. This can help in better utilization of RBUF.

11.3.9.3 Packet/Cell/Frame Count on a Per Port Basis

11.3.9.3.1 Description:

This measurement give the count of number of packets or cells or frames transferred in Transmitting mode. This measurement give the count of number of packets or cells or frames transferred in the receiving mode.

This may be measured per port basis.

11.3.9.4 Inter-Arrival Time for Packets on a Per Port Basis

11.3.9.4.1 Description:

This measurement can provide information on gap between packets thereby indicating effective line rate.

11.3.9.5 Burst Size Histogram

11.3.9.5.1 Description

This measurement give the various burst sizes packets being transmitted and received.

11.4 Events Monitored in Hardware

Tables in this section describe the events that can be measured, including the name of the event, the Event Selection Code (ESC). Please refer to [Section 11.4](#) for tables showing event selection codes.

The acronyms in the event names typically represent unit names.

The guidelines for which events a particular component must implement are as follow:

11.4.1 Queue Statistics Events

11.4.1.1 Queue Latency

Latency of Queue is an indicator of performance of the control logic in terms of effective execution of the commands enqueued in to the Control/Command queue or performance of control logic to effectively transfer data from the Data Queue.

This kind of monitoring needs observation of specific events such as

- Enqueue in to the Queue
This event indicates when an entry was made to the queue.
- Dequeue in to the Queue
This event indicates when an entry was removed from the queue. Time period between when a particular entry was made in to the queue and when the entry was removed from the queue indicates the latency of the queue for that entry.
- Queue Full Event
This event indicates when the queue has no room for additional entries.
- Queue Empty Event
This event indicates when the queue has no entries

Queue Full and Queue Empty events can be used to determine Queue Utilization and bandwidth available in the queue to handle more traffic.

11.4.1.2 Queue Utilization

Utilization of Queue is determined by observing the percentage of time each queue is operating at a particular threshold level. Based on Queue size multiple threshold values can be predetermined and monitored. The result of these observations can be used to provide histograms for Queue utilization. This kind of observation helps us better utilize the available resources in the queue.

11.4.2 Count Events

11.4.2.1 Hardware Block Execution Count

On each of the hardware blocks events are importance such as number of commands executed, no of bytes transferred, total amount of clocks block was free, Total amount of time all the Contexts in the ME was idle can be counted for statistics and to better manage the available resources.

11.4.3 Design Block Select Definitions

Once an event is defined, its definition must remain consistent between products. If the definition changes it should have a new event selection code. This document contains the master list of all ESCs in all CHAP enabled products. Not all of the ESCs in this document are listed in numerical order. The recommendation is to group like events within the following ESC ranges. See [Table 167](#).

Table 167. PMU Design Unit Selection

| Target Device | Target ID | PMU Design Group Block # | Description |
|---------------|-----------|---|---|
| Null | xxx xxx | 0000 | Null (False) Event |
| PMU_Counter | xxx xxx | 0001 (PMU) | CHAP Counters Internal Threshold Events Event bit 0 CHAP Counter 0 Event bit 1 CHAP Counter 1 Event bit 2 CHAP Counter 2 Event bit 3 CHAP Counter 3 Event bit 4 CHAP Counter 4 Event bit 5 CHAP Counter 5 |
| SRAM Group | | | |
| SRAM_DP1 | 001 001 | 0010 (SRAM Group) one and only one will be selected from same group | IXP2800 only SRAM channel 0 SRAM channel 1 SRAM channel 2 SRAM channel 3 SRAM d-push SRAM d-pull IXP2400 only SRAM channel 0 SRAM channel 1 SRAM d-push SRAM d-pull |
| SRAM_DP0 | 001 010 | | |
| SRAM_CH3 | 001 011 | | |
| SRAM_CH2 | 001 100 | | |
| SRAM_CH1 | 001 101 | | |
| SRAM_CH0 | 001 110 | | |
| DRAM Group | | | |
| DRAM_CR1 | 010 000 | 0011 (DRAM) one and only one will be selected from same group | IXP2800 only DRAM channel 0 DRAM channel 1 DRAM channel 2 DRAM d-push DRAM d-pull IXP2400 only DRAM channel 0 DRAM d-push DRAM d-pull |
| DRAM_CR0 | 010 001 | | |
| DRAM_DPLA | 010 010 | | |
| DRAM_DPSA | 010 011 | | |
| DRAM_CH2 | 010 100 | | |
| DRAM_CH1 | 010 101 | | |
| DRAM_CH0 | 010 110 | | |
| | | | |
| XPI | 000 001 | 0100 (XPI) | XPI |
| SHaC | 000 010 | 0101 | SHaC HASH |

Table 167. PMU Design Unit Selection (Continued)

| Target Device | Target ID | PMU Design Group Block # | Description |
|--------------------|-----------|--|---|
| MSF | 000 011 | 0110 | Media |
| XScale® | 000 100 | 0111 | XScale® |
| PCI | 000 101 | 1000 | PCI |
| ME Cluster 0 Group | | | |
| ME07 | 100 111 | 1001 (MEC0) one and only one will be selected from same group | IXP2800 only ME Channel 0 ME00 ME01 ME02 ME03 ME04 ME05 ME06 ME07 |
| ME06 | 100 110 | | IXP2400 only ME Channel 0 ME00 ME01 ME02 ME03 |
| ME05 | 100 101 | | |
| ME04 | 100 100 | | |
| ME03 | 100 011 | | |
| ME02 | 100 010 | | |
| ME01 | 100 001 | | |
| ME00 | 100 000 | | |
| ME Cluster 1 Group | | | |
| ME17 | 110 111 | 1010 (MEC0) one and only one will be selected from same group | IXP2800 only ME Channel 1 ME10 ME11 ME12 ME13 ME14 ME15 ME16 ME17 |
| ME16 | 110 110 | | IXP2400 only ME Channel 1 ME10 ME11 ME12 ME13 |
| ME15 | 110 101 | | |
| ME14 | 110 100 | | |
| ME13 | 110 011 | | |
| ME12 | 110 010 | | |
| ME11 | 110 001 | | |
| ME10 | 110 000 | | |
| | | | 1011-1111 |

11.4.4 Null Event

Not an actual event. When used as an increment or decrement event, no action will take place. When used as Command Trigger it will cause command to be triggered immediately after command register is written to by software. Also called False Event. Not reserved.

11.4.5 Threshold Events

These are the outputs of the threshold comparators. When the value in a data register is compared to its corresponding counter value and the condition is true, a threshold event is generated. This results in:

Pulse on the signal lines that are routed to the events input port (one signal line from each comparator).

One piece of functionality this enables is to allow for CHAP commands to be completed only when a Threshold Event occurs. In other words, a Threshold Event can be used as a Command Trigger to control the execution of any CHAP command (start, stop, sample, etc.). See [Table 168](#).

Table 168. Chap Counter Threshold Events (Design Block # 0001)

| Mux # | Event Name | Clock Domain | Single pulse/ Long pulse | Burst | Description |
|-------|---------------------|--------------|--------------------------|----------|---|
| 000 | Counter 0 Threshold | pp | single | separate | Threshold Condition True on Event Counter 0 |
| 001 | Counter 1 Threshold | pp | single | separate | Threshold Condition True on Event Counter 1 |
| 010 | Counter 2 Threshold | pp | single | separate | Threshold Condition True on Event Counter 2 |
| 011 | Counter 3 Threshold | pp | single | separate | Threshold Condition True on Event Counter 3 |
| 100 | Counter 4 Threshold | pp | single | separate | Threshold Condition True on Event Counter 4 |
| 101 | Counter 5 Threshold | pp | single | separate | Threshold Condition True on Event Counter 5 |

11.4.6 External Input Events

11.4.6.1 XPI Events Target ID(000001) / Design Block #(0100)

Table 169. XPI PMU Event List (Sheet 1 of 6)

| Event Number | Event Name | Clock Domain | Single pulse/ Long pulse | Burst | Description |
|--------------|------------|--------------|--------------------------|----------|---|
| 0 | XPI_RD_P | APB | single | separate | It includes all the read accesses, PMU, timer, GPIO, UART, and SlowPort. |
| 1 | XPI_WR_P | APB | single | separate | It includes all the write accesses, PMU, timer, GPIO, UART, and SlowPort. |
| 2 | PMU_RD_P | APB | single | separate | It executes the read access to the PMU unit. |

Table 169. XPI PMU Event List (Sheet 2 of 6)

| | | | | | |
|----|------------|-----|--------|----------|---|
| 3 | PMU_WR_P | APB | single | separate | It executes the write access to the PMU unit. |
| 4 | UART_RD_P | APB | single | separate | It executes the read access to the UART unit. |
| 5 | UART_WR_P | APB | single | separate | It executes the write access to the UART unit. |
| 6 | GPIO_RD_P | APB | single | separate | It executes the read access to the GPIO unit. |
| 7 | GPIO_WR_P | APB | single | separate | It executes the write access to the GPIO unit. |
| 8 | TIMER_RD_P | APB | single | separate | It executes the read access to the Timer unit. |
| 9 | TIMER_WR_P | APB | single | separate | It executes the write access to the Timer unit. |
| 10 | SPDEV_RD_P | APB | single | separate | It executes the read access to the SlowPort Device. |
| 11 | SPDEV_WR_P | APB | single | separate | It executes the write access to the SlowPort Device. |
| 12 | SPCSR_RD_P | APB | single | separate | It executes the read access to the SlowPort CSR. |
| 13 | SPCSR_WR_P | APB | single | separate | It executes the write access to the SlowPort CSR. |
| 14 | TM0_UF_P | APB | single | separate | It shows the occurrence of timer 1 counter underflow. |
| 15 | TM1_UF_P | APB | single | separate | It shows the occurrence of timer 2 counter underflow. |
| 16 | TM2_UF_P | APB | single | separate | It shows the occurrence of timer 3 counter underflow. |
| 17 | TM3_UF_P | APB | single | separate | It shows the occurrence of timer 4 counter underflow. |
| 18 | IDLE0_0_P | APB | single | separate | It displays the idle state of the state machine 0 for the mode 0 of SlowPort. |
| 19 | START0_1_P | APB | single | separate | It enters the start state of the state machine 0 for the mode 0 of SlowPort. |
| 20 | ADDR10_3_P | APB | single | separate | It enters the first address state, AD[9:2], of the state machine 0 for the mode 0 of SlowPort. |
| 21 | ADDR20_2_P | APB | single | separate | It enters the second address state, AD[17:10], of the state machine 0 for the mode 0 of SlowPort. |
| 22 | ADDR30_6_P | APB | single | separate | It enters the third address state, AD[24:18], of the state machine 0 for the mode 0 of SlowPort. |
| 23 | SETUP0_4_P | APB | single | separate | It enters data setup state of the state machine 0 for the mode 0 of SlowPort. |
| 24 | PULW0_5_P | APB | single | separate | It enters data duration state of the state machine 0 for the mode 0 of SlowPort. |
| 25 | HOLD0_D_P | APB | single | separate | It enters data hold state of the state machine 0 for the mode 0 of SlowPort. |

Table 169. XPI PMU Event List (Sheet 3 of 6)

| | | | | | |
|----|-------------|-----|--------|----------|--|
| 26 | TURNA0_C_P | APB | single | separate | It enters the termination state of the state machine 0 for the mode 0 of SlowPort. |
| 27 | IDLE1_0_P | APB | single | separate | It displays the idle state of the state machine 1 for the mode 1 of SlowPort. |
| 28 | START1_1_P | APB | single | separate | It enters the start state of the state machine 1 for the mode 1 of SlowPort. |
| 29 | ADDR11_3_P | APB | single | separate | It enters the first address state, AD[7:0], of the state machine 1 for the mode 1 of SlowPort. |
| 30 | ADDR21_2_P | APB | single | separate | It enters the second address state, AD[15:8], of the state machine 1 for the mode 1 of SlowPort. |
| 31 | ADDR31_6_P | APB | single | separate | It enters the second address state, AD[23:16], of the state machine 1 for the mode 1 of SlowPort. |
| 32 | ADDR41_7_P | APB | single | separate | It enters the second address state, AD[24], of the state machine 1 for the mode 1 of SlowPort. |
| 33 | WRDATA1_5_P | APB | single | separate | It unpacks the data from the APB bus onto the SlowPort bus for the state machine 1 for the mode 1 of SlowPort. |
| 34 | PULW1_4_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 1 for the mode 1 of SlowPort. |
| 35 | CHPSEL1_C_P | APB | single | separate | It enters the chip select assertion pulse width when the state machine 1 is active for the mode 1 of SlowPort. |
| 36 | OUTEN1_E_P | APB | single | separate | It enters the cycle when the OE is asserted during running on the state machine 1 for the mode 1 of SlowPort. |
| 37 | PKDATA1_F_P | APB | single | separate | It enters the read data packing state when the state machine 1 is active for the mode 1 of SlowPort. |
| 38 | LADATA1_D_P | APB | single | separate | It enters the data capturing cycle when the state machine 1 is active for the mode 1 of SlowPort. |
| 39 | READY1_9_P | APB | single | separate | It enters the acknowledge state to terminate the read cycle when the state machine 1 is active for the mode 1 of SlowPort. |
| 40 | TURNA1_8_P | APB | single | separate | It enters the turnaround state of the transaction when the state machine 1 is active for the mode 1 of SlowPort. |
| 41 | IDLE2_0_P | APB | single | separate | It displays the idle state of the state machine 2 for the mode 2 of SlowPort. |
| 42 | START2_1_P | APB | single | separate | It enters the start state of the state machine 2 for the mode 2 of SlowPort. |
| 43 | ADDR12_3_P | APB | single | separate | It enters the first address state, AD[7:0], of the state machine 2 for the mode 2 of SlowPort. |

Table 169. XPI PMU Event List (Sheet 4 of 6)

| | | | | | |
|----|-------------|-----|--------|----------|--|
| 44 | ADDR22_2_P | APB | single | separate | It enters the second address state, AD[15:8], of the state machine 2 for the mode 2 of SlowPort. |
| 45 | ADDR32_6_P | APB | single | separate | It enters the second address state, AD[23:16], of the state machine 2 for the mode 2 of SlowPort. |
| 46 | ADDR42_7_P | APB | single | separate | It enters the second address state, AD[24], of the state machine 2 for the mode 2 of SlowPort. |
| 47 | WRDATA2_5_P | APB | single | separate | It unpacks the data from the APB bus onto the SlowPort bus for the state machine 2 for the mode 2 of SlowPort. |
| 48 | SETUP2_4_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 2 for the mode 2 of SlowPort. |
| 49 | PULW2_C_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 2 for the mode 2 of SlowPort. |
| 50 | HOLD2_E_P | APB | single | separate | It enters the data hold period for the state machine 2 for the mode 2 of SlowPort. |
| 51 | OUTEN2_F_P | APB | single | separate | It starts to assert the OE when the state machine 2 is active for the mode 2 of SlowPort. |
| 52 | PKDATA2_D_P | APB | single | separate | It enters the read data packing state during the active state machine 2 for the mode 2 of SlowPort. |
| 53 | LADATA2_9_P | APB | single | separate | It enters the data capturing cycle during the active state machine 2 for the mode 2 of SlowPort. |
| 54 | READY2_B_P | APB | single | separate | It enters the acknowledge state to terminate the read cycle when the state machine 2 is active for the mode 2 of SlowPort. |
| 55 | TURNA2_8_P | APB | single | separate | It enters the turnaround state of the transaction when the state machine 2 is active for the mode 2 of SlowPort. |
| 56 | IDLE3_0_P | APB | single | separate | It displays the idle state of the state machine 3 for the mode 3 of SlowPort. |
| 57 | START3_1_P | APB | single | separate | It enters the start state of the state machine 3 for the mode 3 of SlowPort. |
| 58 | ADDR13_3_P | APB | single | separate | It enters the first address state, AD[7:0], of the state machine 3 for the mode 3 of SlowPort. |
| 59 | ADDR23_2_P | APB | single | separate | It enters the second address state, AD[15:8], of the state machine 3 for the mode 3 of SlowPort. |
| 60 | ADDR33_6_P | APB | single | separate | It enters the second address state, AD[23:16], of the state machine 3 for the mode 3 of SlowPort. |
| 61 | ADDR43_7_P | APB | single | separate | It enters the second address state, AD[24], of the state machine 3 for the mode 3 of SlowPort. |

Table 169. XPI PMU Event List (Sheet 5 of 6)

| | | | | | |
|----|-------------|-----|--------|----------|--|
| 62 | WRDATA3_5_P | APB | single | separate | It unpacks the data from the APB bus onto the SlowPort bus for the state machine 3 for the mode 3 of SlowPort. |
| 63 | SETUP3_4_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 3 for the mode 3 of SlowPort. |
| 64 | PULW3_C_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 3 for the mode 3 of SlowPort. |
| 65 | HOLD3_E_P | APB | single | separate | It enters the data hold period for the state machine 3 for the mode 3 of SlowPort. |
| 66 | OUTEN3_F_P | APB | single | separate | It starts to assert the OE when the state machine 3 is active for the mode 3 of SlowPort. |
| 67 | PKDATA3_D_P | APB | single | separate | It enters the read data packing state during the active state machine 3 for the mode 3 of SlowPort. |
| 68 | LADATA3_B_P | APB | single | separate | It enters the data capturing cycle during the active state machine 3 for the mode 3 of SlowPort. |
| 69 | READY3_9_P | APB | single | separate | It enters the acknowledge state to terminate the read cycle when the state machine 3 is active for the mode 3 of SlowPort. |
| 70 | TURN3_8_P | APB | single | separate | It enters the turnaround state of the transaction when the state machine 3 is active for the mode 3 of SlowPort. |
| 71 | IDLE4_0_P | APB | single | separate | It displays the idle state of the state machine 4 for the mode 4 of SlowPort. |
| 72 | START4_1_P | APB | single | separate | It enters the start state of the state machine 4 for the mode 4 of SlowPort. |
| 73 | ADDR14_3_P | APB | single | separate | It enters the first address state, AD[7:0], of the state machine 4 for the mode 4 of SlowPort. |
| 74 | ADDR24_2_P | APB | single | separate | It enters the second address state, AD[15:8], of the state machine 4 for the mode 4 of SlowPort. |
| 75 | ADDR34_6_P | APB | single | separate | It enters the second address state, AD[23:16], of the state machine 4 for the mode 4 of SlowPort. |
| 76 | ADDR44_7_P | APB | single | separate | It enters the second address state, AD[24], of the state machine 4 for the mode 4 of SlowPort. |
| 77 | WRDATA4_5_P | APB | single | separate | It unpacks the data from the APB bus onto the SlowPort bus for the state machine 4 for the mode 4 of SlowPort. |
| 78 | SETUP4_4_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 4 for the mode 4 of SlowPort. |
| 79 | PULW4_C_P | APB | single | separate | It enters the pulse width of the data transaction cycle for the state machine 4 for the mode 4 of SlowPort. |

Table 169. XPI PMU Event List (Sheet 6 of 6)

| | | | | | |
|----|-------------|-----|--------|----------|--|
| 80 | HOLD4_E_P | APB | single | separate | It enters the data hold period for the state machine 4 for the mode 4 of SlowPort. |
| 81 | OUTEN4_F_P | APB | single | separate | It starts to assert the OE when the state machine 4 is active for the mode 4 of SlowPort. |
| 82 | PKDATA4_D_P | APB | single | separate | It enters the read data packing state during the active state machine 4 for the mode 4 of SlowPort. |
| 83 | LADATA4_B_P | APB | single | separate | It enters the data capturing cycle during the active state machine 4 for the mode 4 of SlowPort. |
| 84 | READY4_9_P | APB | single | separate | It enters the acknowledge state to terminate the read cycle when the state machine 4 is active for the mode 4 of SlowPort. |
| 85 | TURNA4_8_P | APB | single | separate | It enters the turnaround state of the transaction when the state machine 4 is active for the mode 4 of SlowPort. |

11.4.6.2 SHaC Events Target ID(00010) / Design Block #(0101)

Table 170. SHaC PMU Event List (Sheet 1 of 4)

| Event Number | Event Name | Clock Domain | Single pulse/ Long pulse | Burst | Description |
|--------------|----------------------------------|--------------|--------------------------|----------|---|
| 0 | Scratch Cmd_Inlet_Fifo Not_Empty | P_CLK | single | separate | !sh_scratch.sh_cmd_ctl.sh_qcmd_valid_rph |
| 1 | Scratch Cmd_Inlet_Fifo Full | P_CLK | single | separate | SHTA_CMD_Q_FULL_RPH |
| 2 | Scratch Cmd_Inlet_Fifo Enqueue | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.sh_scr_cmd_queue.shcmd_wr_queue_wph |
| 3 | Scratch Cmd_Inlet_Fifo Dequeue | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.sh_adv_cmd_wph |
| 4 | Scratch Cmd_Pipe Not_Empty | P_CLK | single | separate | sh_scratch.scr_cmd_valid_rph |
| 5 | Scratch Cmd_Pipe Full | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.sh_cmd_pipe_full_wph |
| 6 | Scratch Cmd_Pipe Enqueue | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.sh_adv_cmd_wph ; duplicate with event 3 |

Table 170. SHaC PMU Event List (Sheet 2 of 4)

| | | | | | |
|----|------------------------------------|-------|--------|----------|--|
| 7 | Scratch Cmd_Pipe Dequeue | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.scr_deq_cmd_pip_e_wph |
| 8 | Scratch Pull_Data_Fifo 0 Full | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.scr_pull0_fifo_full_wph |
| 9 | Scratch Pull_Data_Fifo 1 Full | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.scr_pull1_fifo_full_wph |
| 10 | Hash Pull_Data_Fifo 0 Full | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.hash_pull0_fifo_full_wph |
| 11 | Hash Pull_Data_Fifo 1 Full | P_CLK | single | separate | sh_scratch.sh_cmd_ctl.hash_pull1_fifo_full_wph |
| 12 | Scratch Pull_Data_Fifo 0 Not_Empty | P_CLK | single | separate | !sh_scratch.scr_pull0_data_valid_rph |
| 13 | Scratch Pull_Data_Fifo 0 Enqueue | P_CLK | single | separate | sh_scratch.sh_scr_take_pull0_data_wph |
| 14 | Scratch Pull_Data_Fifo 0 Dequeue | P_CLK | single | separate | sh_scratch.scr_read_pull0_data_wph |
| 15 | Scratch Pull_Data_Fifo 1 Not_Empty | P_CLK | single | separate | !sh_scratch.scr_pull1_data_valid_rph |
| 16 | Scratch Pull_Data_Fifo 1 Enqueue | P_CLK | single | separate | sh_scratch.sh_scr_take_pull1_data_wph |
| 17 | Scratch Pull_Data_Fifo 1 Dequeue | P_CLK | single | separate | sh_scratch.scr_read_pull1_data_wph |
| 18 | Scratch State Machine Idle | P_CLK | single | separate | sh_scratch.scr_sm_idle_wph |
| 19 | Scratch RAM Write | P_CLK | single | separate | RAM_SCRATCH_WR_WPH |
| 20 | Scratch RAM Read | P_CLK | single | separate | RAM_SCRATCH_RD_WPH |
| 21 | Scratch Ring_0 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[0] |
| 22 | Scratch Ring_1 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[1] |
| 23 | Scratch Ring_2 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[2] |
| 24 | Scratch Ring_3 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[3] |

Table 170. SHaC PMU Event List (Sheet 3 of 4)

| | | | | | |
|----|------------------------------|-------|--------|----------|---|
| 25 | Scratch Ring_4 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[4] |
| 26 | Scratch Ring_5 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[5] |
| 27 | Scratch Ring_6 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[6] |
| 28 | Scratch Ring_7 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[7] |
| 29 | Scratch Ring_8 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[8] |
| 30 | Scratch Ring_9 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[9] |
| 31 | Scratch Ring_10 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[10] |
| 32 | Scratch Ring_11 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[11] |
| 33 | Scratch Ring_12 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[12] |
| 34 | Scratch Ring_13 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[13] |
| 35 | Scratch Ring_14 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[14] |
| 36 | Scratch Ring_15 Full | P_CLK | single | separate | SHXX_RING_FULL_RPH[15] |
| 37 | CAP CSR Write | P_CLK | single | separate | sh_scratch.scr_csr_write_wph |
| 38 | CAP CSR Fast Write | P_CLK | single | separate | sh_scratch.SCR_FAST_WRITE_RPH |
| 39 | CAP CSR Read | P_CLK | single | separate | sh_scratch.scr_csr_read_wph |
| 40 | DEQUEUE APB data | P_CLK | single | separate | sh_scratch.sh_apb_slave.SCR_DEQ_APB_DATA_WPH |
| 41 | apb_push_cmd_wph | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_push_cmd_wph |
| 42 | APB_PUSH_DATA_REQ_RPH | P_CLK | single | separate | sh_scratch.sh_apb_slave.APB_PUSH_DATA_REQ_RPH |
| 43 | APB pull1 FIFO dequeue | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_deq_pull1_data_wph |
| 44 | apb_deq_pull1_data_wph | P_CLK | single | separate | sh_scratch.sh_apb_slave.SCR_APB_TAKE_PULL1_DATA_WPH |
| 45 | data valid in apb pull1 FIFO | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_pull1_data_valid_wph |
| 46 | APB pull0 FIFO dequeue | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_deq_pull0_data_wph |

Table 170. SHaC PMU Event List (Sheet 4 of 4)

| | | | | | |
|----|---------------------------------|-------|--------|----------|---|
| 47 | SCR_APB_TAKE_PULL0_DATA_WPH | P_CLK | single | separate | sh_scratch.sh_apb_slave.SCR_APB_TAKE_PULL0_DATA_WPH |
| 48 | data valid in apb pull0 FIFO | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_pull0_data_valid_wph |
| 49 | CAP APB read | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_rd_wph |
| 50 | CAP APB write | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_wr_wph |
| 51 | APB cmd dequeue | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_deq_cmd_wph |
| 52 | APB CMD FIFO enqueue | P_CLK | single | separate | sh_scratch.sh_apb_slave.SH_ENQ_APB_CMD_WPH |
| 53 | APB CMD FIFO FULL | P_CLK | single | separate | sh_scratch.sh_apb_slave.APB_CMD_Q_FULL_RPH |
| 54 | APB CMD valid | P_CLK | single | separate | sh_scratch.sh_apb_slave.apb_cmd_valid_wph |
| 55 | Hash Pull_Data_Fifo 0 Not_Empty | P_CLK | single | separate | sh_hash.hash_pull0_data_valid_rph |
| 56 | Hash Pull_Data_Fifo 0 Enqueue | P_CLK | single | separate | sh_hash.SCR_HASH_TAKE_DATA0_RPH |
| 57 | Hash Pull_Data_Fifo 0 Dequeue | P_CLK | single | separate | sh_hash.hash_read_pull0_data_wph |
| 58 | Hash Pull_Data_Fifo 1 Not_Empty | P_CLK | single | separate | sh_hash.hash_pull1_data_valid_rph |
| 59 | Hash Pull_Data_Fifo 1 Enqueue | P_CLK | single | separate | sh_hash.SCR_HASH_TAKE_DATA1_RPH |
| 60 | Hash Pull_Data_Fifo 1 Dequeue | P_CLK | single | separate | sh_hash.hash_read_pull1_data_wph |
| 61 | Hash Active | P_CLK | single | separate | sh_hash.hash_active_rph |
| 62 | Hash Cmd_Pipe Not_Empty | P_CLK | single | separate | sh_hash.hash_cmd_valid_p3_rph |
| 63 | Hash Cmd_Pipe Full | P_CLK | single | separate | !HASH_REQ_CMD_WPH |
| 64 | Hash Push_Data_Pipe Not_Empty | P_CLK | single | separate | !HASH_PUSH_DATA_REQ_RPH |
| 65 | Hash Push_Data_Pipe Full | P_CLK | single | separate | !sh_hash.hash_adv_push_data_wph |

11.4.6.3 XScale® Events Target ID(000100) / Design Block #(0111)

Table 171. XScale® Gasket PMU Event List (Sheet 1 of 4)

| Event Number | Event Name | Clock Domain | Single pulse/ Long pulse | Burst | Description |
|--------------|----------------------|--------------|--------------------------|----------|--|
| 0 | XG_CFIFO_WR_EVEN_XS | C_CLK | single | separate | XG command FIFO even enqueue |
| 1 | XG_CFIFO_WR_ODD_XS | C_CLK | single | separate | XG command FIFO odd enqueue |
| 2 | XG_DFIFO_WR_EVEN_XS | C_CLK | single | separate | XG DRAM data FIFO even enqueue |
| 3 | XG_DFIFO_WR_ODD_XS | C_CLK | single | separate | XG DRAM data FIFO odd enqueue |
| 4 | XG_SFIFO_WR_EVEN_XS | C_CLK | single | separate | XG SRAM data FIFO even enqueue |
| 5 | XG_SFIFO_WR_ODD_XS | C_CLK | single | separate | XG SRAM data FIFO odd enqueue |
| 6 | XG_LCFIFO_WR_EVEN_XS | C_CLK | single | separate | XG lcsr command FIFO even enqueue |
| 7 | XG_LCFIFO_WR_ODD_XS | C_CLK | single | separate | XG lcsr command FIFO odd enqueue |
| 8 | XG_LDFIFO_WR_EVEN_XS | C_CLK | single | separate | XG lcsr data FIFO even enqueue |
| 9 | XG_LDFIFO_WR_ODD_XS | C_CLK | single | separate | XG lcsr data FIFO odd enqueue |
| 10 | XG_LCSR_RD_EVEN_XS | C_CLK | single | separate | XG lcsr return data FIFO even dequeue |
| 11 | XG_LCSR_RD_ODD_XS | C_CLK | single | separate | XG lcsr return data FIFO odd dequeue |
| 12 | XG_LCSR_RD_OR_XS | C_CLK | single | separate | XG lcsr return data FIFO even_or_odd dequeue |
| 13 | XG_PUFF0_RD_EVEN_XS | C_CLK | single | separate | XG push fifo0 even dequeue |
| 14 | XG_PUFF0_RD_ODD_XS | C_CLK | single | separate | XG push fifo0 odd dequeue |
| 15 | XG_PUFF0_RD_OR_XS | C_CLK | single | separate | XG push fifo0 even_or_odd dequeue |
| 16 | XG_PUFF1_RD_EVEN_XS | C_CLK | single | separate | XG push fifo1 even dequeue |
| 17 | XG_PUFF1_RD_ODD_XS | C_CLK | single | separate | XG push fifo1 odd dequeue |
| 18 | XG_PUFF1_RD_OR_XS | C_CLK | single | separate | XG push fifo1 even_or_odd dequeue |
| 19 | XG_PUFF2_RD_EVEN_XS | C_CLK | single | separate | XG push fifo2 even dequeue |
| 20 | XG_PUFF2_RD_ODD_XS | C_CLK | single | separate | XG push fifo2 odd dequeue |
| 21 | XG_PUFF2_RD_OR_XS | C_CLK | single | separate | XG push fifo2 even_or_odd dequeue |
| 22 | XG_PUFF3_RD_EVEN_XS | C_CLK | single | separate | XG push fifo3 even dequeue |
| 23 | XG_PUFF3_RD_ODD_XS | C_CLK | single | separate | XG push fifo3 odd dequeue |
| 24 | XG_PUFF3_RD_OR_XS | C_CLK | single | separate | XG push fifo3 even_or_odd dequeue |
| 25 | XG_PUFF4_RD_EVEN_XS | C_CLK | single | separate | XG push fifo4 even dequeue |
| 26 | XG_PUFF4_RD_ODD_XS | C_CLK | single | separate | XG push fifo4 odd dequeue |
| 27 | XG_PUFF4_RD_OR_XS | C_CLK | single | separate | XG push fifo4 even_or_odd dequeue |
| 28 | XG_SYNC_ST_XS | C_CLK | single | separate | XG in sync. state |
| 29 | reserved | | | | |

Table 171. XScale® Gasket PMU Event List (Sheet 2 of 4)

| | | | | | |
|----|----------------------|-------|--------|----------|-----------------------------------|
| 30 | reserved | | | | |
| 31 | reserved | | | | |
| 32 | reserved | | | | |
| 33 | reserved | | | | |
| 34 | XG_CFIFO_EMPTYN_CPP | P_CLK | single | separate | XG command FIFO empty flag |
| 35 | XG_DFIFO_EMPTYN_CPP | P_CLK | single | separate | XG DRAM data FIFO empty flag |
| 36 | XG_SFIFO_EMPTYN_CPP | P_CLK | single | separate | XG SRAM data FIFO empty flag |
| 37 | XG_LCFIFO_EMPTYN_CPP | P_CLK | single | separate | XG lcsr command FIFO empty flag |
| 38 | XG_LDFIFO_EMPTYN_CPP | P_CLK | single | separate | XG lcsr data FIFO empty flag |
| 39 | reserved | | | | |
| 40 | XG_OFIFO_EMPTYN_CPP | P_CLK | single | separate | XG cpp command FIFO empty flag |
| 41 | XG_OFIFO_FULLN_CPP | P_CLK | single | separate | XG cpp command FIFO full flag |
| 42 | XG_DP_EMPTYN_CPP | P_CLK | single | separate | XG DRAM pull data FIFO empty flag |
| 43 | XG_SP_EMPTYN_CPP | P_CLK | single | separate | XG SRAM pull data FIFO empty flag |
| 44 | XG_HASH_48_CPP | P_CLK | single | separate | hash_48 command on cpp bus |
| 45 | XG_HASH_64_CPP | P_CLK | single | separate | hash_64 command on cpp bus |
| 46 | XG_HASH_128_CPP | P_CLK | single | separate | hash_128 command on cpp bus |
| 47 | XG_LCSR_FIQ_CPP | P_CLK | single | separate | XG FIQ generated by interrupt CSR |
| 48 | XG_LCSR_IRQ_CPP | P_CLK | single | separate | XG IRQ generated by interrupt CSR |
| 49 | XG_CFIFO_RD_CPP | P_CLK | single | separate | XG command FIFO dequeue |
| 50 | XG_DFIFO_RD_CPP | P_CLK | single | separate | XG DRAM data FIFO dequeue |
| 51 | XG_SFIFO_RD_CPP | P_CLK | single | separate | XG SRAM data FIFO dequeue |
| 52 | XG_LCFIFO_RD_CPP | P_CLK | single | separate | XG lcsr command FIFO dequeue |
| 53 | XG_LDFIFO_RD_CPP | P_CLK | single | separate | XG lcsr data FIFO dequeue |
| 54 | XG_LCSR_WR_CPP | P_CLK | single | separate | XG lcsr return data FIFO enqueue |
| 55 | XG_OFIFO_RD_CPP | P_CLK | single | separate | XG cpp command FIFO dequeue |
| 56 | XG_OFIFO_WR_CPP | P_CLK | single | separate | XG cpp command FIFO enqueue |
| 57 | XG_DPDATA_WR_CPP | P_CLK | single | separate | XG DRAM pull data FIFO enqueue |
| 58 | XG_DPDATA_RD_CPP | P_CLK | single | separate | XG DRAM pull data FIFO dequeue |
| 59 | XG_SPDATA_WR_CPP | P_CLK | single | separate | XG SRAM pull data FIFO enqueue |
| 60 | XG_SPDATA_RD_CPP | P_CLK | single | separate | XG SRAM pull data FIFO dequeue |
| 61 | XG_PUFF0_WR_CPP | P_CLK | single | separate | XG push fifo0 enqueue |
| 62 | XG_PUFF1_WR_CPP | P_CLK | single | separate | XG push fifo1 enqueue |
| 63 | XG_PUFF2_WR_CPP | P_CLK | single | separate | XG push fifo2 enqueue |
| 64 | XG_PUFF3_WR_CPP | P_CLK | single | separate | XG push fifo3 enqueue |
| 65 | XG_PUFF4_WR_CPP | P_CLK | single | separate | XG push fifo4 enqueue |
| 66 | XG_SRAM_RD_CPP | P_CLK | single | separate | XG SRAM read command on cpp bus |
| 67 | XG_SRAM_RD_1_CPP | P_CLK | single | separate | XG SRAM read length=1 on cpp bus |

Table 171. XScale® Gasket PMU Event List (Sheet 3 of 4)

| | | | | | |
|-----|--------------------|-------|--------|----------|--------------------------------------|
| 68 | XG_SRAM_RD_8_CPP | P_CLK | single | separate | XG SRAM read length=8 on cpp bus |
| 69 | XG_SRAM_WR_CPP | P_CLK | single | separate | XG SRAM write command on cpp bus |
| 70 | XG_SRAM_WR_1_CPP | P_CLK | single | separate | XG SRAM write length=1 on cpp bus |
| 71 | XG_SRAM_WR_2_CPP | P_CLK | single | separate | XG SRAM write length=2 on cpp bus |
| 72 | XG_SRAM_WR_3_CPP | P_CLK | single | separate | XG SRAM write length=3 on cpp bus |
| 73 | XG_SRAM_WR_4_CPP | P_CLK | single | separate | XG SRAM write length=4 on cpp bus |
| 74 | XG_SRAM_CSR_RD_CPP | P_CLK | single | separate | XG SRAM csr read command on cpp bus |
| 75 | XG_SRAM_CSR_WR_CPP | P_CLK | single | separate | XG SRAM csr write command on cpp bus |
| 76 | XG_SRAM_ATOM_CPP | P_CLK | single | separate | XG SRAM atomic command on cpp bus |
| 77 | XG_SRAM_GET_CPP | P_CLK | single | separate | XG SRAM get command on cpp bus |
| 78 | XG_SRAM_PUT_CPP | P_CLK | single | separate | XG SRAM put command on cpp bus |
| 79 | XG_SRAM_ENQ_CPP | P_CLK | single | separate | XG SRAM enq command on cpp bus |
| 80 | XG_SRAM_DEQ_CPP | P_CLK | single | separate | XG SRAM deq command on cpp bus |
| 81 | XG_S0_ACC_CPP | P_CLK | single | separate | XG SRAM channel0 access on cpp bus |
| 82 | XG_S1_ACC_CPP | P_CLK | single | separate | XG SRAM channel1 access on cpp bus |
| 83 | XG_S2_ACC_CPP | P_CLK | single | separate | XG SRAM channel2 access on cpp bus |
| 84 | XG_S3_ACC_CPP | P_CLK | single | separate | XG SRAM channel3 access on cpp bus |
| 85 | XG_SCR_RD_CPP | P_CLK | single | separate | XG scratch read command on cpp bus |
| 86 | XG_SCR_RD_1_CPP | P_CLK | single | separate | XG scratch read length=1 on cpp bus |
| 87 | XG_SCR_RD_8_CPP | P_CLK | single | separate | XG scratch read length=8 on cpp bus |
| 88 | XG_SCR_WR_CPP | P_CLK | single | separate | XG scratch write command on cpp bus |
| 89 | XG_SCR_WR_1_CPP | P_CLK | single | separate | XG scratch write length=1 on cpp bus |
| 90 | XG_SCR_WR_2_CPP | P_CLK | single | separate | XG scratch write length=2 on cpp bus |
| 91 | XG_SCR_WR_3_CPP | P_CLK | single | separate | XG scratch write length=3 on cpp bus |
| 92 | XG_SCR_WR_4_CPP | P_CLK | single | separate | XG scratch write length=4 on cpp bus |
| 93 | XG_SCR_ATOM_CPP | P_CLK | single | separate | XG scratch atomic command on cpp bus |
| 94 | XG_SCR_GET_CPP | P_CLK | single | separate | XG scratch get command on cpp bus |
| 95 | XG_SCR_PUT_CPP | P_CLK | single | separate | XG scratch put command on cpp bus |
| 96 | XG_DRAM_RD_CPP | P_CLK | single | separate | XG DRAM read command on cpp bus |
| 97 | XG_DRAM_RD_1_CPP | P_CLK | single | separate | XG DRAM read length=1 on cpp bus |
| 98 | XG_DRAM_RD_4_CPP | P_CLK | single | separate | XG DRAM read length=4 on cpp bus |
| 99 | XG_DRAM_WR_CPP | P_CLK | single | separate | XG DRAM write on cpp bus |
| 100 | XG_DRAM_WR_1_CPP | P_CLK | single | separate | XG DRAM write length=1 on cpp bus |

Table 171. XScale® Gasket PMU Event List (Sheet 4 of 4)

| | | | | | |
|-----|--------------------|-------|--------|----------|--------------------------------------|
| 101 | XG_DRAM_WR_2_CPP | P_CLK | single | separate | XG DRAM write length=2 on cpp bus |
| 102 | XG_DRAM_CSR_RD_CPP | P_CLK | single | separate | XG DRAM csr read command on cpp bus |
| 103 | XG_DRAM_CSR_WR_CPP | P_CLK | single | separate | XG DRAM csr write command on cpp bus |
| 104 | XG_MSF_RD_CPP | P_CLK | single | separate | XG msf read command on cpp bus |
| 105 | XG_MSF_RD_1_CPP | P_CLK | single | separate | XG msf read length=1 on cpp bus |
| 106 | reserved | | | | |
| 107 | XG_MSF_WR_CPP | P_CLK | single | separate | XG msf write command on cpp bus |
| 108 | XG_MSF_WR_1_CPP | P_CLK | single | separate | XG msf write length=1 on cpp bus |
| 109 | XG_MSF_WR_2_CPP | P_CLK | single | separate | XG msf write length=2 on cpp bus |
| 110 | XG_MSF_WR_3_CPP | P_CLK | single | separate | XG msf write length=3 on cpp bus |
| 111 | XG_MSF_WR_4_CPP | P_CLK | single | separate | XG msf write length=4 on cpp bus |
| 112 | XG_PCI_RD_CPP | P_CLK | single | separate | XG pci read command on cpp bus |
| 113 | XG_PCI_RD_1_CPP | P_CLK | single | separate | XG pci read length=1 on cpp bus |
| 114 | XG_PCI_RD_8_CPP | P_CLK | single | separate | XG pci read length=8 on cpp bus |
| 115 | XG_PCI_WR_CPP | P_CLK | single | separate | XG pci write command on cpp bus |
| 116 | XG_PCI_WR_1_CPP | P_CLK | single | separate | XG pci write length=1 on cpp bus |
| 117 | XG_PCI_WR_2_CPP | P_CLK | single | separate | XG pci write length=2 on cpp bus |
| 118 | XG_PCI_WR_3_CPP | P_CLK | single | separate | XG pci write length=3 on cpp bus |
| 119 | XG_PCI_WR_4_CPP | P_CLK | single | separate | XG pci write length=4 on cpp bus |
| 120 | XG_CAP_RD_CPP | P_CLK | single | separate | XG cap read command on cpp bus |
| 121 | XG_CAP_RD_1_CPP | P_CLK | single | separate | XG cap read length=1 on cpp bus |
| 122 | XG_CAP_RD_8_CPP | P_CLK | single | separate | XG cap read length=8 on cpp bus |
| 123 | XG_CAP_WR_CPP | P_CLK | single | separate | XG cap write command on cpp bus |
| 124 | XG_CAP_WR_1_CPP | P_CLK | single | separate | XG cap write length=1 on cpp bus |
| 125 | reserved | | | | |
| 126 | reserved | | | | |
| 127 | reserved | | | | |

11.4.6.4 PCI Events Target ID(000101) / Design Block #(1000)

Table 172. PCI PMU Event List (Sheet 1 of 5)

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|--------------|----------------------|--------------|-------------|----------|-----------------------------------|
| 0 | PCI_TGT_AFIFO_FULL | C_CLK | single | separate | PCI Target Address FIFO Full |
| 1 | PCI_TGT_AFIFO_NEMPTY | C_CLK | single | separate | PCI Target Address FIFO Not Empty |
| 2 | PCI_TGT_AFIFO_WR | C_CLK | single | separate | PCI Target Address FIFO Write |
| 3 | PCI_TGT_AFIFO_RD | C_CLK | single | separate | PCI Target Address FIFO Read |
| 4 | PCI_TGT_RFIFO_FULL | C_CLK | single | separate | PCI Target Read FIFO Full |
| 5 | PCI_TGT_RFIFO_NEMPTY | C_CLK | single | separate | PCI Target Read FIFO Not Empty |
| 6 | PCI_TGT_RFIFO_WR | C_CLK | single | separate | PCI Target Read FIFO Write |
| 7 | PCI_TGT_RFIFO_RD | C_CLK | single | separate | PCI Target Read FIFO Read |
| 8 | PCI_TGT_WFIFO_FULL | C_CLK | single | separate | PCI Target Write FIFO Full |
| 9 | PCI_TGT_WFIFO_NEMPTY | C_CLK | single | separate | PCI Target Write FIFO Not Empty |
| 10 | PCI_TGT_WFIFO_WR | C_CLK | single | separate | PCI Target Write FIFO Write |
| 11 | PCI_TGT_WFIFO_RD | C_CLK | single | separate | PCI Target Write FIFO Read |
| 12 | PCI_TGT_WBUF_FULL | C_CLK | single | separate | PCI Target Write Buffer Full |
| 13 | PCI_TGT_WBUF_NEMPTY | C_CLK | single | separate | PCI Target Write Buffer Not Empty |
| 14 | PCI_TGT_WBUF_WR | C_CLK | single | separate | PCI Target Write Buffer Write |
| 15 | PCI_TGT_WBUF_RD | C_CLK | single | separate | PCI Target Write Buffer Read |
| 16 | PCI_MST_AFIFO_FULL | C_CLK | single | separate | PCI Master Address FIFO Full |
| 17 | PCI_MST_AFIFO_NEMPTY | C_CLK | single | separate | PCI Master Address FIFO Not Empty |
| 18 | PCI_MST_AFIFO_WR | C_CLK | single | separate | PCI Master Address FIFO Write |
| 19 | PCI_MST_AFIFO_RD | C_CLK | single | separate | PCI Master Address FIFO Read |
| 20 | PCI_MST_RFIFO_FULL | C_CLK | single | separate | PCI Master Read FIFO Full |
| 21 | PCI_MST_RFIFO_NEMPTY | C_CLK | single | separate | PCI Master Read FIFO Not Empty |
| 22 | PCI_MST_RFIFO_WR | C_CLK | single | separate | PCI Master Read FIFO Write |

Table 172. PCI PMU Event List (Sheet 2 of 5)

| | | | | | |
|----|-----------------------|-------|--------|----------|---------------------------------|
| 23 | PCI_MST_RFIFO_RD | C_CLK | single | separate | PCI Master Read FIFO Read |
| 24 | PCI_MST_WFIFO_FULL | C_CLK | single | separate | PCI Master Write FIFO Full |
| 25 | PCI_MST_WFIFO_NEMPTY | C_CLK | single | separate | PCI Master Write FIFO Not Empty |
| 26 | PCI_MST_WFIFO_WR | C_CLK | single | separate | PCI Master Write FIFO Write |
| 27 | PCI_MST_WFIFO_RD | C_CLK | single | separate | PCI Master Write FIFO Read |
| 28 | PCI_DMA1_BUF_FULL | C_CLK | single | separate | PCI_DMA_Channel 1 |
| 29 | PCI_DMA1_BUF_NEMPTY | | | | |
| 30 | PCI_DMA1_BUF_WR | | | | |
| 31 | PCI_DMA1_BUF_RD | | | | |
| 32 | PCI_DMA2_BUF_FULL | | | | PCI_DMA_Channel 2 |
| 33 | PCI_DMA2_BUF_NEMPTY | | | | |
| 34 | PCI_DMA2_BUF_WR | P_CLK | single | separate | |
| 35 | PCI_DMA2_BUF_RD | P_CLK | single | separate | |
| 36 | PCI_DMA3_BUF_FULL | P_CLK | single | separate | PCI_DMA_Channel 3 |
| 37 | PCI_DMA3_BUF_NEMPTY | P_CLK | single | separate | |
| 38 | PCI_DMA3_BUF_WR | P_CLK | single | separate | |
| 39 | PCI_DMA3_BUF_RD | P_CLK | single | separate | |
| 40 | PCI_TCMD_FIFO_FULL | P_CLK | single | separate | PCI TARGET Command Fifo |
| 41 | PCI_TCMD_FIFO_NEMPTY | P_CLK | single | separate | |
| 42 | PCI_TCMD_FIFO_WR | P_CLK | single | separate | |
| 43 | PCI_TCMD_FIFO_RD | P_CLK | single | separate | |
| 44 | PCI_TDATA_FIFO_FULL | P_CLK | single | separate | PCI Push/Pull Data Fifo |
| 45 | PCI_TDATA_FIFO_NEMPTY | P_CLK | single | separate | |
| 46 | PCI_TDATA_FIFO_WR | P_CLK | single | separate | |
| 47 | PCI_TDATA_FIFO_RD | P_CLK | single | separate | |
| 48 | PCI_CSR_WRITE | P_CLK | single | separate | PCI Write to PCI_CSR_BAR |
| 49 | PCI_CSR_READ | P_CLK | single | separate | |
| 50 | PCI_DRAM_WRITE | P_CLK | single | separate | PCI Write to PCI_DRAM_BAR |
| 51 | PCI_DRAM_READ | P_CLK | single | separate | |

Table 172. PCI PMU Event List (Sheet 3 of 5)

| | | | | | |
|----|------------------------------|-------|--------|----------|--|
| 52 | PCI_DRAM_BURST_WRITE | P_CLK | single | separate | PCI Burst Write to PCI_CSR_BAR |
| 53 | PCI_DRAM_BURST_READ | P_CLK | single | separate | PCI Burst Read to PCI_CSR_BAR |
| 54 | PCI_SRAM_WRITE | P_CLK | single | separate | PCI Write to PCI_SRAM_BAR |
| 55 | PCI_SRAM_READ | P_CLK | single | separate | |
| 56 | PCI_SRAM_BURST_WRITE | P_CLK | single | separate | PCI Burst Write to PCI_SRAM_BAR |
| 57 | PCI_SRAM_BURST_READ | P_CLK | single | separate | |
| 58 | PCI_CSR_CMD | P_CLK | single | separate | PCI CSR Command Generated |
| 59 | PCI_CSR_PUSH | P_CLK | single | separate | PCI CSR Push Command |
| 60 | PCI_CSR_PULL | P_CLK | single | separate | PCI CSR Pull Command |
| 61 | PCI_SRAM_CMD | P_CLK | single | separate | PCI SRAM Command |
| 62 | PCI_SRAM_PUSH | P_CLK | single | separate | PCI SRAM Push Command |
| 63 | PCI_SRAM_PULL | P_CLK | single | separate | PCI SRAM Pull Command |
| 64 | PCI_DRAM_CMD | P_CLK | single | separate | PCI DRAM Command |
| 65 | PCI_DRAM_PUSH | P_CLK | single | separate | |
| 66 | PCI_DRAM_PULL | P_CLK | single | separate | |
| 67 | PCI_CSR_2PCI_WR | P_CLK | single | separate | PCI Target Write to PCI local CSR |
| 68 | PCI_CSR_2PCI_RD | P_CLK | single | separate | |
| 69 | PCI_CSR_2CFG_WR | P_CLK | single | separate | PCI Target Write to PCI local Config CSR |
| 70 | PCI_CSR_2CFG_RD | P_CLK | single | separate | |
| 71 | PCI_CSR_2SRAM_WR | P_CLK | single | separate | PCI Target Write to SRAM CSR |
| 72 | PCI_CSR_2SRAM_RD | P_CLK | single | separate | |
| 73 | PCI_CSR_2DRAM_WR | P_CLK | single | separate | PCI Target Write to DRAM CSR |
| 74 | PCI_CSR_2DRAM_RD | P_CLK | single | separate | |
| 75 | PCI_CSR_2CAP_WR | P_CLK | single | separate | PCI Target Write to CAPCSR |
| 76 | PCI_CSR_2CAP_RD | P_CLK | single | separate | |
| 77 | PCI_CSR_2MSF_WR | P_CLK | single | separate | PCI Target Write to MSFCSR |
| 78 | PCI_CSR_2MSF_RD | P_CLK | single | separate | |
| 79 | PCI_CSR_2SCRAPE_WR | P_CLK | single | separate | PCI Target Write to Scrape CSR |
| 80 | PCI_CSR_2SCRAPE_RD | P_CLK | single | separate | |
| 81 | PCI_CSR_2SCRATCH_RING_W R | P_CLK | single | separate | PCI Target Write to Scratch Ring CSR |
| 82 | PCI_CSR_2SCRATCH_RING_R D | P_CLK | single | separate | |
| 83 | PCI_CSR_2SRAM_RING_WR | P_CLK | single | separate | PCI Target Write to SRAM Ring CSR |

Table 172. PCI PMU Event List (Sheet 4 of 5)

| | | | | | |
|-----|-----------------------|-------|--------|----------|---|
| 84 | PCI_CSR_2SRAM_RING_RD | P_CLK | single | separate | |
| 85 | PCI_XS_LCFG_RD | P_CLK | single | separate | PCI XScale® Read Local Config CSR |
| 86 | PCI_XS_LCFG_WR | P_CLK | single | separate | |
| 87 | PCI_XS_CSR_RD | P_CLK | single | separate | PCI XScale Read Local CSR |
| 88 | PCI_XS_CSR_WR | P_CLK | single | separate | |
| 89 | PCI_XS_CFG_RD | P_CLK | single | separate | PCI XScale Read PCI Bus Config Space |
| 90 | PCI_XS_CFG_WR | P_CLK | single | separate | |
| 91 | PCI_XS_MEM_RD | P_CLK | single | separate | PCI XScale Read PCI Bus Memory Space |
| 92 | PCI_XS_MEM_WR | P_CLK | single | separate | |
| 93 | PCI_XS_BURST_RD | P_CLK | single | separate | PCI XScale Burst Read PCI Bus Memory Space |
| 94 | PCI_XS_BURST_WR | P_CLK | single | separate | |
| 95 | PCI_XS_IO_RD | P_CLK | single | separate | PCI XScale Read PCI Bus I/O Space |
| 96 | PCI_XS_IO_WR | P_CLK | single | separate | |
| 97 | PCI_XS_SPEC | P_CLK | single | separate | PCI XScale Read PCI Bus as Special |
| 98 | PCI_XS_IACK | P_CLK | single | separate | PCI XScale Read PCI Bus as IACK |
| 99 | PCI_ME_CSR_RD | P_CLK | single | separate | PCI ME Read Local CSR |
| 100 | PCI_ME_CSR_WR | P_CLK | single | separate | |
| 101 | PCI_ME_MEM_RD | P_CLK | single | separate | PCI ME Read PCI Bus Memory Space |
| 102 | PCI_ME_MEM_WR | P_CLK | single | separate | |
| 103 | PCI_ME_BURST_RD | P_CLK | single | separate | PCI ME Burst Read PCI Bus Memory Space |
| 104 | PCI_ME_BURST_WR | P_CLK | single | separate | |
| 105 | PCI_MST_CFG_RD | P_CLK | single | separate | PCI Initiator Read PCI Bus Config Space |
| 106 | PCI_MST_CFG_WR | P_CLK | single | separate | |
| 107 | PCI_MST_MEM_RD | P_CLK | single | separate | PCI Initiator Read PCI Bus Memory Space |
| 108 | PCI_MST_MEM_WR | P_CLK | single | separate | |
| 109 | PCI_MST_BURST_RD | P_CLK | single | separate | PCI Initiator Burst Read PCI Bus Memory Space |
| 110 | PCI_MST_BURST_WR | P_CLK | single | separate | |
| 111 | PCI_MST_IO_READ | P_CLK | single | separate | PCI Initiator Read PCI Bus I/O Space |
| 112 | PCI_MST_IO_WRITE | P_CLK | single | separate | |

Table 172. PCI PMU Event List (Sheet 5 of 5)

| | | | | | |
|-----|-------------------|-------|--------|----------|---|
| 113 | PCI_MST_SPEC | P_CLK | single | separate | PCI Initiator Read PCI Bus As a Special Cycle |
| 114 | PCI_MST_IACK | P_CLK | single | separate | PCI Initiator Read PCI Bus As IACK Cycle |
| 115 | PCI_MST_READ_LINE | P_CLK | single | separate | PCI Initiator Read Line Command to PCI |
| 116 | PCI_MST_READ_MULT | P_CLK | single | separate | PCI Initiator Read Line Multiple Command to PCI |
| 117 | PCI_ARB_REQ[2] | P_CLK | single | separate | Internal Arbiter PCI Bus Request 2 |
| 118 | PCI_ARB_GNT[2] | P_CLK | single | separate | Internal Arbiter PCI Bus Grant 2 |
| 119 | PCI_ARB_REQ[1] | P_CLK | single | separate | |
| 120 | PCI_ARB_GNT[1] | P_CLK | single | separate | |
| 121 | PCI_ARB_REQ[0] | P_CLK | single | separate | |
| 122 | PCI_ARB_GNT[0] | P_CLK | single | separate | |
| 123 | PCI_TGT_STATE[4] | P_CLK | single | separate | PCI Target State Machine State Bit 4 |
| 124 | PCI_TGT_STATE[3] | P_CLK | single | separate | |
| 125 | PCI_TGT_STATE[2] | P_CLK | single | separate | |
| 126 | PCI_TGT_STATE[1] | P_CLK | single | separate | |
| 127 | PCI_TGT_STATE[0] | P_CLK | single | separate | |

11.4.6.5 ME00 Events Target ID(100000) / Design Block #(1001)

Table 173. ME00 PMU Event List (Sheet 1 of 2)

| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |
|---|------------------|--------------|-------------|----------|--|
| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
| 0 | ME_FIFO_ENQ_EVEN | T_CLK | single | separate | Even version of Command FIFO Enqueue (pair with event #8) |
| 1 | ME_IDLE_EVEN | T_CLK | single | separate | Even version of No Thread running in ME (pair with event #9) |

Table 173. ME00 PMU Event List (Sheet 2 of 2)

| | | | | | |
|----|----------------------|-------|--------|----------|---|
| 2 | ME_EXECUTING_EVEN | T_CLK | single | separate | Even version of Valid Instruction (pair with event #10) |
| 3 | ME_STALL_EVEN | T_CLK | single | separate | Even version of ME stall caused by FIFO Full (pair with event #11) |
| 4 | ME_CTX_SWAPPING_EVEN | T_CLK | single | separate | Even version of Occurrence of context swap (pair with event #12) |
| 5 | ME_INST_ABORT_EVEN | T_CLK | single | separate | Even version of Instruction aborted due to branch taken (pair with event #13) |
| 6 | ME_FIFO_ENQ_ODD | T_CLK | single | separate | Odd version of Command FIFO Enqueue (pair with event #0) |
| 7 | ME_IDLE_ODD | T_CLK | single | separate | Odd version of No Thread running in ME (pair with event #3) |
| 8 | ME_EXECUTING_ODD | T_CLK | single | separate | Odd version of Valid Instruction (pair with event #4) |
| 9 | ME_STALL_ODD | T_CLK | single | separate | Odd version of ME stall caused by FIFO Full (pair with event #5) |
| 10 | ME_CTX_SWAPPING_ODD | T_CLK | single | separate | Odd version of Occurrence of context swap (pair with event #6) |
| 11 | ME_INST_ABORT_ODD | T_CLK | single | separate | Odd version of Instruction aborted due to branch (pair with event #7) |
| 12 | ME_FIFO_DEQ | P_CLK | single | separate | Command FIFO Dequeue |
| 13 | ME_FIFO_NOT_EMPTY | P_CLK | single | separate | Command FIFO NOT Empty |

11.4.6.6 ME01 Events Target ID(100001) / Design Block #(1001)

Table 174. ME01 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.7 ME02 Events Target ID(100010) / Design Block #(1001)

Table 175. ME02 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.8 ME03 Events Target ID(100011) / Design Block #(1001)

Table 176. ME03 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.9 ME10 Events Target ID(110000) / Design Block #(1010)

Table 177. ME10 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.10 ME11 Events Target ID(110001) / Design Block #(1010)

Table 178. ME11 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.11 ME12 Events Target ID(110010) / Design Block #(1010)

Table 179. ME12 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.12 ME13 Events Target ID(110011) / Design Block #(1010)

Table 180. ME13 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the ME has same event list. CC_Enable bit[2:0] is PMU_CTX_Monitor in ME CSR, This field holds the number of context to be monitored. The event count will only reflect the events that occurred when this context is executing. <p>CC_Enable[2:0] = 000, select context number 0, CC_Enable[2:0] = 001, select context number 1, CC_Enable[2:0] = 111, select context number 7.</p> <ol style="list-style-type: none"> T_CLK = 2x P_CLK | | | | | |

11.4.6.13 SRAM DP1 Events Target ID(001001) / Design Block #(0010)

Table 181. SRAM DP1 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <p>1. SRAM DP1/DP0 push/pull arbiter has same event lists.</p> <p>2. S_CLK = SRAM clock domain</p> <p>3. P_CLK = PP clock domain</p> <p>signals that begin with sps_ correspond to S-Push Arb</p> <p>signals that begin with spl_ correspond to S-Pull Arb</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _m_ (after the unit designation) correspond to the MSF target interface</p> <p>signals that contain _sh_ (after the unit designation) correspond to the SHAC target interface</p> <p>signals that contain _s0_ (after the unit designation) correspond to the SRAM0 target interface</p> <p>signals that contain _s1_ (after the unit designation) correspond to the SRAM1 target interface</p> <p>signals that contain _s2_ (after the unit designation) correspond to the SRAM2 target interface</p> <p>signals that contain _s3_ (after the unit designation) correspond to the SRAM3 target interface</p> | | | | | |

11.4.6.14 SRAM DP0 Events Target ID(001010) / Design Block #(0010)

Table 182. SRAM DP0 PMU Event List

| Event Number | Event Name | Clock Domain | Single pulse/Long pulse | Burst | Description |
|--------------|------------------------|--------------|-------------------------|----------|-------------|
| 0 | sps_pc_cmd_valid_rph | P_CLK | Long | separate | |
| 1 | sps_pc_enq_wph | P_CLK | single | separate | |
| 2 | sps_pc_deq_wph | P_CLK | single | separate | |
| 3 | sps_pc_push_q_full_wph | P_CLK | Long | separate | |
| 4 | sps_m_cmd_valid_rph | P_CLK | Long | separate | |
| 5 | sps_m_enq_wph | P_CLK | single | separate | |
| 6 | sps_m_deq_wph | P_CLK | single | separate | |
| 7 | sps_m_push_q_full_wph | P_CLK | Long | separate | |
| 8 | sps_sh_cmd_valid_rph | P_CLK | Long | separate | |
| 9 | sps_sh_enq_wph | P_CLK | single | separate | |
| 10 | sps_sh_deq_wph | P_CLK | single | separate | |
| 11 | sps_sh_push_q_full_wph | P_CLK | Long | separate | |

Table 182. SRAM DP0 PMU Event List

| | | | | | |
|----|-------------------------|-------|--------|----------|--|
| 12 | sps_s0_cmd_valid_rph | P_CLK | Long | separate | |
| 13 | sps_s0_enq_wph | P_CLK | single | separate | |
| 14 | sps_s0_deq_wph | P_CLK | single | separate | |
| 15 | sps_s0_push_q_full_wph | P_CLK | Long | separate | |
| 16 | sps_s1_cmd_valid_rph | P_CLK | Long | separate | |
| 17 | sps_s1_enq_wph | P_CLK | single | separate | |
| 18 | sps_s1_deq_wph | P_CLK | single | separate | |
| 19 | sps_s1_push_q_full_wph | P_CLK | Long | separate | |
| 20 | sps_s2_cmd_valid_rph | P_CLK | Long | separate | |
| 21 | sps_s2_enq_wph | P_CLK | single | separate | |
| 22 | sps_s2_deq_wph | P_CLK | single | separate | |
| 23 | sps_s2_push_q_full_wph | P_CLK | Long | separate | |
| 24 | sps_s3_cmd_valid_rph | P_CLK | Long | separate | |
| 25 | sps_s3_enq_wph | P_CLK | single | separate | |
| 26 | sps_s3_deq_wph | P_CLK | single | separate | |
| 27 | sps_s3_push_q_full_wph | P_CLK | Long | separate | |
| 28 | spl_pc_cmd_valid_rph | P_CLK | Long | separate | |
| 29 | spl_pc_enq_cmd_wph | P_CLK | single | separate | |
| 30 | spl_pc_deq_wph | P_CLK | single | separate | |
| 31 | spl_pc_cmd_que_full_wph | P_CLK | Long | separate | |
| 32 | spl_m_cmd_valid_rph | P_CLK | Long | separate | |
| 33 | spl_m_enq_cmd_wph | P_CLK | single | separate | |
| 34 | spl_m_deq_wph | P_CLK | single | separate | |
| 35 | spl_m_cmd_que_full_wph | P_CLK | Long | separate | |
| 36 | spl_sh_cmd_valid_rph | P_CLK | Long | separate | |
| 37 | spl_sh_enq_cmd_wph | P_CLK | single | separate | |
| 38 | spl_sh_deq_wph | P_CLK | single | separate | |
| 39 | spl_sh_cmd_que_full_wph | P_CLK | Long | separate | |
| 40 | spl_s0_cmd_valid_rph | P_CLK | Long | separate | |
| 41 | spl_s0_enq_cmd_wph | P_CLK | single | separate | |
| 42 | spl_s0_deq_wph | P_CLK | single | separate | |
| 43 | spl_s0_cmd_que_full_wph | P_CLK | Long | separate | |
| 44 | spl_s1_cmd_valid_rph | P_CLK | Long | separate | |
| 45 | spl_s1_enq_cmd_wph | P_CLK | single | separate | |

Table 182. SRAM DP0 PMU Event List

| | | | | | |
|----|-------------------------|-------|--------|----------|--|
| 46 | spl_s1_deq_wph | P_CLK | single | separate | |
| 47 | spl_s1_cmd_que_full_wph | P_CLK | Long | separate | |
| 48 | spl_s2_cmd_valid_rph | P_CLK | Long | separate | |
| 49 | spl_s2_enq_cmd_wph | P_CLK | single | separate | |
| 50 | spl_s2_deq_wph | P_CLK | single | separate | |
| 51 | spl_s2_cmd_que_full_wph | P_CLK | Long | separate | |
| 52 | spl_s3_cmd_valid_rph | P_CLK | Long | separate | |
| 53 | spl_s3_enq_cmd_wph | P_CLK | single | separate | |
| 54 | spl_s3_deq_wph | P_CLK | single | separate | |
| 55 | spl_s3_cmd_que_full_wph | P_CLK | Long | separate | |

11.4.6.15 SRAM CH3 Events Target ID(001011) / Design Block #(0010)

Table 183. SRAM CH3 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <ol style="list-style-type: none"> All the SRAM Channel has same event lists. S_CLK = SRAM clock domain P_CLK = PP clock domain <p>signals that begin with sps_ correspond to S-Push Arb signals that begin with spl_ correspond to S-Pull Arb</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface signals that contain _pc_ (after the unit designation) correspond to the PCI target interface signals that contain _m_ (after the unit designation) correspond to the MSF target interface signals that contain _sh_ (after the unit designation) correspond to the SHAC target interface signals that contain _s0_ (after the unit designation) correspond to the SRAM0 target interface signals that contain _s1_ (after the unit designation) correspond to the SRAM1 target interface signals that contain _s2_ (after the unit designation) correspond to the SRAM2 target interface signals that contain _s3_ (after the unit designation) correspond to the SRAM3 target interface</p> | | | | | |

11.4.6.16 SRAM CH2 Events Target ID(001100) / Design Block #(0010)

Table 184. SRAM CH3 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <p>1. All the SRAM Channel has same event lists.</p> <p>2. S_CLK = SRAM clock domain</p> <p>3. P_CLK = PP clock domain</p> <p>signals that begin with sps_ correspond to S-Push Arb</p> <p>signals that begin with spl_ correspond to S-Pull Arb</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _m_ (after the unit designation) correspond to the MSF target interface</p> <p>signals that contain _sh_ (after the unit designation) correspond to the SHAC target interface</p> <p>signals that contain _s0_ (after the unit designation) correspond to the SRAM0 target interface</p> <p>signals that contain _s1_ (after the unit designation) correspond to the SRAM1 target interface</p> <p>signals that contain _s2_ (after the unit designation) correspond to the SRAM2 target interface</p> <p>signals that contain _s3_ (after the unit designation) correspond to the SRAM3 target interface</p> | | | | | |

11.4.6.17 SRAM CH1 Events Target ID(001101) / Design Block #(0010)

Table 185. SRAM CH3 PMU Event List

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|---|------------|--------------|-------------|-------|-------------|
| <p>Note:</p> <p>1. All the SRAM Channel has same event lists.</p> <p>2. S_CLK = SRAM clock domain</p> <p>3. P_CLK = PP clock domain</p> <p>signals that begin with sps_ correspond to S-Push Arb</p> <p>signals that begin with spl_ correspond to S-Pull Arb</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _pc_ (after the unit designation) correspond to the PCI target interface</p> <p>signals that contain _m_ (after the unit designation) correspond to the MSF target interface</p> <p>signals that contain _sh_ (after the unit designation) correspond to the SHAC target interface</p> <p>signals that contain _s0_ (after the unit designation) correspond to the SRAM0 target interface</p> <p>signals that contain _s1_ (after the unit designation) correspond to the SRAM1 target interface</p> <p>signals that contain _s2_ (after the unit designation) correspond to the SRAM2 target interface</p> <p>signals that contain _s3_ (after the unit designation) correspond to the SRAM3 target interface</p> | | | | | |

11.4.6.18 SRAM CH0 Events Target ID(001110) / Design Block #(0010)

Table 186. SRAM CH0 PMU Event List (Sheet 1 of 3)

| Event Number | Event Name | Clock Domain | Single pulse/ Long pulse | Burst | Description |
|--------------|----------------------------|--------------|--------------------------|----------|----------------------------|
| 0 | QDR I/O Read | S_CLK | single | separate | QDR I/O Read |
| 1 | QDR I/O Write | S_CLK | single | separate | QDR I/O Write |
| 2 | Read Cmd Dispatched | P_CLK | single | separate | Read Cmd Dispatched |
| 3 | Write Cmd Dispatched | P_CLK | single | separate | Write Cmd Dispatched |
| 4 | Swap Cmd Dispatched | P_CLK | single | separate | Swap Cmd Dispatched |
| 5 | Set Dispatched | P_CLK | single | separate | Set Dispatched |
| 6 | Clear Cmd Dispatched | P_CLK | single | separate | Clear Cmd Dispatched |
| 7 | Add Cmd Dispatched | P_CLK | single | separate | Add Cmd Dispatched |
| 8 | Sub Cmd Dispatched | P_CLK | single | separate | Sub Cmd Dispatched |
| 9 | Incr Cmd Dispatched | P_CLK | single | separate | Incr Cmd Dispatched |
| 10 | Decr Cmd Dispatched | P_CLK | single | separate | Decr Cmd Dispatched |
| 11 | Ring Cmd Dispatched | P_CLK | single | separate | Ring Cmd Dispatched |
| 12 | Jour Cmd Dispatched | P_CLK | single | separate | Jour Cmd Dispatched |
| 13 | Deq Cmd Dispatched | P_CLK | single | separate | Deq Cmd Dispatched |
| 14 | Enq Cmd Dispatched | P_CLK | single | separate | Enq Cmd Dispatched |
| 15 | CSR Cmd Dispatched | P_CLK | single | separate | CSR Cmd Dispatched |
| 16 | WQDesc Cmd Dispatched | P_CLK | single | separate | WQDesc Cmd Dispatched |
| 17 | RQDesc Cmd Dispatched | P_CLK | single | separate | RQDesc Cmd Dispatched |
| 18 | FIFO Deque – CmdA0 Inlet Q | P_CLK | single | separate | FIFO Deque – CmdA0 Inlet Q |
| 19 | FIFO Enque – CmdA0 Inlet Q | P_CLK | single | separate | FIFO Enque – CmdA0 Inlet Q |
| 20 | FIFO Valid – CmdA0 Inlet Q | P_CLK | long | separate | FIFO Valid – CmdA0 Inlet Q |

Table 186. SRAM CH0 PMU Event List (Sheet 2 of 3)

| | | | | | |
|----|------------------------------|-------|--------|----------|----------------------------------|
| 21 | FIFO Full – CmdA1 Inlet Q | P_CLK | long | separate | FIFO Full – CmdA1 Inlet Q |
| 22 | FIFO Deque – CmdA1 Inlet Q | P_CLK | single | separate | FIFO Deque – CmdA1 Inlet Q |
| 23 | FIFO Enque – CmdA1 Inlet Q | P_CLK | single | separate | FIFO Enque – CmdA1 Inlet Q |
| 24 | FIFO Valid – CmdA1 Inlet Q | P_CLK | long | separate | FIFO Valid – CmdA1 Inlet Q |
| 25 | FIFO Full – CmdA1 Inlet Q | P_CLK | long | separate | FIFO Full – CmdA1 Inlet Q |
| 26 | FIFO Deque – Wr Cmd Q | S_CLK | single | separate | FIFO Deque – Wr Cmd Q |
| 27 | FIFO Enque – Wr Cmd Q | P_CLK | single | separate | FIFO Enque – Wr Cmd Q |
| 28 | FIFO Valid – Wr Cmd Q | S_CLK | long | separate | FIFO Valid – Wr Cmd Q |
| 29 | FIFO Full – Wr Cmd Q | P_CLK | long | separate | FIFO Full – Wr Cmd Q |
| 30 | FIFO Deque – Queue Cmd Q | S_CLK | single | separate | FIFO Deque – Queue Cmd Q |
| 31 | FIFO Enque – Queue Cmd Q | P_CLK | single | separate | FIFO Enque – Queue Cmd Q |
| 32 | FIFO Valid – Queue Cmd Q | S_CLK | long | separate | FIFO Valid – Queue Cmd Q |
| 33 | FIFO Full – Queue Cmd Q | P_CLK | long | separate | FIFO Full – Queue Cmd Q |
| 34 | FIFO Deque – Rd Cmd Q | S_CLK | single | separate | FIFO Deque – Rd Cmd Q |
| 35 | FIFO Enque – Rd Cmd Q | P_CLK | single | separate | FIFO Enque – Rd Cmd Q |
| 36 | FIFO Valid – Rd Cmd Q | S_CLK | long | separate | FIFO Valid – Rd Cmd Q |
| 37 | FIFO Full – Rd Cmd Q | P_CLK | long | separate | FIFO Full – Rd Cmd Q |
| 38 | FIFO Deque – Oref Cmd Q | S_CLK | single | separate | FIFO Deque – Oref Cmd Q |
| 39 | FIFO Enque – Oref Cmd Q | P_CLK | single | separate | FIFO Enque – Oref Cmd Q |
| 40 | FIFO Valid – Oref Cmd Q | S_CLK | long | separate | FIFO Valid – Oref Cmd Q |
| 41 | FIFO Full – Oref Cmd Q | P_CLK | long | separate | FIFO Full – Oref Cmd Q |
| 42 | FIFO Deque – SP0 Pull Data Q | S_CLK | single | separate | FIFO Deque – SP0 Pull Data Q |
| 43 | FIFO Enque – SP0 Pull Data Q | P_CLK | single | separate | 1 = FIFO Enque – SP0 Pull Data Q |

Table 186. SRAM CH0 PMU Event List (Sheet 3 of 3)

| | | | | | |
|----|------------------------------|-------|--------|----------|------------------------------|
| 44 | FIFO Valid – SP0 Pull Data Q | S_CLK | long | separate | FIFO Valid – SP0 Pull Data Q |
| 45 | FIFO Full – SP0 Pull Data Q | P_CLK | long | separate | FIFO Full – SP0 Pull Data Q |
| 46 | FIFO Deque – SP1 Pull Data Q | S_CLK | single | separate | FIFO Deque – SP1 Pull Data Q |
| 47 | FIFO Enque – SP1 Pull Data Q | P_CLK | single | separate | FIFO Enque – SP1 Pull Data Q |
| 48 | FIFO Valid – SP1 Pull Data Q | S_CLK | long | separate | FIFO Valid – SP1 Pull Data Q |
| 49 | FIFO Full – SP1 Pull Data Q | P_CLK | long | separate | FIFO Full – SP1 Pull Data Q |
| 50 | FIFO Deque – Push ID/Data Q | P_CLK | single | separate | FIFO Deque – Push ID/Data Q |
| 51 | FIFO Enque – Push ID/Data Q | S_CLK | single | separate | FIFO Enque – Push ID/Data Q |
| 52 | FIFO Valid – Push ID/Data Q | P_CLK | long | separate | FIFO Valid – Push ID/Data Q |
| 53 | FIFO Full – Push ID/Data Q | S_CLK | long | separate | FIFO Full – Push ID/Data Q |

11.4.6.19 IXP2400 DRAM Events Target ID(010000) / Design Block #(0011)

Table 187. IXP2400 DRAM PMU Event List (Sheet 1 of 3)

| Event Number | Event Name | Clock Domain | Pulse/Level | Burst | Description |
|--------------|---------------------|--------------|-------------|----------|-------------------------------|
| 0 | inlet_cmdq_enq | P_CLK | Single | Separate | Inlet Command Queue Enqueue |
| 1 | inlet_cmdq_deq | d2_CLK | Single | Separate | Inlet Command Queue Dequeue |
| 2 | inlet_cmdq_notempty | d2_CLK | Single | Separate | Inlet Command Queue Not Empty |
| 3 | inlet_cmdq_full | P_CLK | Single | Separate | Inlet Command Queue Full |
| 4 | bnk0_opq_enq | d2_CLK | Single | Separate | Bank0 Op Queue Enqueue |
| 5 | bnk0_opq_deq | d2_CLK | Single | Separate | Bank0 Op Queue Dequeue |
| 6 | bnk0_opq_notempty | d2_CLK | Single | Separate | Bank0 Op Queue Not Empty |
| 7 | bnk0_opq_full | d2_CLK | Single | Separate | Bank0 Op Queue Full |
| 8 | bnk1_opq_enq | d2_CLK | Single | Separate | Bank1 Op Queue Enqueue |
| 9 | bnk1_opq_deq | d2_CLK | Single | Separate | Bank1 Op Queue Dequeue |
| 10 | bnk1_opq_notempty | d2_CLK | Single | Separate | Bank1 Op Queue Not Empty |
| 11 | bnk1_opq_full | d2_CLK | Single | Separate | Bank1 Op Queue Full |
| 12 | bnk2_opq_enq | d2_CLK | Single | Separate | Bank2 Op Queue Enqueue |
| 13 | bnk2_opq_deq | d2_CLK | Single | Separate | Bank2 Op Queue Dequeue |
| 14 | bnk2_opq_notempty | d2_CLK | Single | Separate | Bank2 Op Queue Not Empty |
| 15 | bnk2_opq_full | d2_CLK | Single | Separate | Bank2 Op Queue Full |
| 16 | bnk3_opq_enq | d2_CLK | Single | Separate | Bank3 Op Queue Enqueue |
| 17 | bnk3_opq_deq | d2_CLK | Single | Separate | Bank3 Op Queue Dequeue |
| 18 | bnk3_opq_notempty | d2_CLK | Single | Separate | Bank3 Op Queue Not Empty |
| 19 | bnk3_opq_full | d2_CLK | Single | Separate | Bank3 Op Queue Full |
| 20 | push_cmdq_enq | d2_CLK | Single | Separate | Push Command Queue Enqueue |
| 21 | push_cmdq_deq | P_CLK | Single | Separate | Push Command Queue Dequeue |
| 22 | push_cmdq_notempty | P_CLK | Single | Separate | Push Command Queue Not Empty |
| 23 | push_dataq_enq | d2_CLK | Single | Separate | Push Data Queue Enqueue |
| 24 | push_dataq_deq | P_CLK | Single | Separate | Push Data Queue Dequeue |
| 25 | push_dataq_notempty | P_CLK | Single | Separate | Push Data Queue Not Empty |
| 26 | pull_cmdq_enq | d2_CLK | Single | Separate | Pull Command Queue Enqueue |

Table 187. IXP2400 DRAM PMU Event List (Sheet 2 of 3)

| | | | | | |
|----|-------------------------|--------|--------|----------|--------------------------------------|
| 27 | pull_cmdq_deq | P_CLK | Single | Separate | Pull Command Queue Dequeue |
| 28 | pull_cmdq_notempty | P_CLK | Single | Separate | SeparatePull Command Queue Not Empty |
| 29 | pull_cmdq_full | d2_CLK | Single | Separate | Pull Command Queue Full |
| 30 | pull_dataq_enq | P_CLK | Single | Separate | Pull Data Queue Enqueue |
| 31 | pull_dataq_deq | d2_CLK | Single | Separate | Pull Data Queue Dequeue |
| 32 | pull_dataq_notempty | d2_CLK | Single | Separate | Pull Data Queue Not Empty |
| 33 | pull_dataq_full | P_CLK | Single | Separate | Pull Data Queue Full |
| 34 | bnk0_dataq_enq | d2_CLK | Single | Separate | Bank0 Data Queue Enqueue |
| 35 | bnk0_dataq_deq | d2_CLK | Single | Separate | Bank0 Data Queue Dequeue |
| 36 | bnk0_dataq_notempty | d2_CLK | Single | Separate | Bank0 Data Queue Not Empty |
| 37 | bnk0_dataq_full | d2_CLK | Single | Separate | Bank0 Data Queue Full |
| 38 | bnk1_dataq_enq | d2_CLK | Single | Separate | Bank1 Data Queue Enqueue |
| 39 | bnk1_dataq_deq | d2_CLK | Single | Separate | Bank1 Data Queue Dequeue |
| 40 | bnk1_dataq_notempty | d2_CLK | Single | Separate | Bank1 Data Queue Not Empty |
| 41 | bnk1_dataq_full | d2_CLK | Single | Separate | Bank1 Data Queue Full |
| 42 | bnk2_dataq_enq | d2_CLK | Single | Separate | Bank2 Data Queue Enqueue |
| 43 | bnk2_dataq_deq | d2_CLK | Single | Separate | Bank2 Data Queue Dequeue |
| 44 | bnk2_dataq_notempty | d2_CLK | Single | Separate | Bank2 Data Queue Not Empty |
| 45 | bnk2_dataq_full | d2_CLK | Single | Separate | Bank2 Data Queue Full |
| 46 | bnk3_dataq_enq | d2_CLK | Single | Separate | Bank3 Data Queue Enqueue |
| 47 | bnk3_dataq_deq | d2_CLK | Single | Separate | SeparateBank3 Data Queue Dequeue |
| 48 | bnk3_dataq_notempty | d2_CLK | Single | Separate | Bank3 Data Queue Not Empty |
| 49 | bnk3_dataq_full | d2_CLK | Single | Separate | Bank3 Data Queue Full |
| 50 | pending_pullq_enq | d2_CLK | Single | Separate | Pending Pull Queue Enqueue |
| 51 | pending_pullq_deq | d2_CLK | Single | Separate | Pending Pull Queue Dequeue |
| 52 | pending_pullq_notempty | d2_CLK | Single | Separate | Pending Pull Queue Not Empty |
| 53 | dram_activate_bnk0side0 | d2_CLK | Single | Separate | DRAM Activate Bank0side0 |
| 54 | dram_activate_bnk1side0 | d2_CLK | Single | Separate | DRAM Activate Bank1side0 |
| 55 | dram_activate_bnk2side0 | d2_CLK | Single | Separate | DRAM Activate Bank2side0 |
| 56 | dram_activate_bnk3side0 | d2_CLK | Single | Separate | DRAM Activate Bank3side0 |

Table 187. IXP2400 DRAM PMU Event List (Sheet 3 of 3)

| | | | | | |
|----|-------------------------|--------|--------|----------|-------------------------------|
| 57 | dram_activate_bnk0side1 | d2_CLK | Single | Separate | DRAM Activate Bank0side1 |
| 58 | dram_activate_bnk1side1 | d2_CLK | Single | Separate | DRAM Activate Bank1side1 |
| 59 | dram_activate_bnk2side1 | d2_CLK | Single | Separate | DRAM Activate Bank2side1 |
| 60 | dram_activate_bnk3side1 | d2_CLK | Single | Separate | DRAM Activate Bank3side1 |
| 61 | dram_read_bnk0side0 | d2_CLK | Single | Separate | DRAM Read Bank0side0 |
| 62 | dram_read_bnk1side0 | d2_CLK | Single | Separate | DRAM Read Bank1side0 |
| 63 | dram_read_bnk2side0 | d2_CLK | Single | Separate | DRAM Read Bank2side0 |
| 64 | dram_read_bnk3side0 | d2_CLK | Single | Separate | DRAM Read Bank3side0 |
| 65 | dram_read_bnk0side1 | d2_CLK | Single | Separate | DRAM Read Bank0side1 |
| 66 | dram_read_bnk1side1 | d2_CLK | Single | Separate | DRAM Read Bank1side1 |
| 67 | dram_read_bnk2side1 | d2_CLK | Single | Separate | DRAM Read Bank2side1 |
| 68 | dram_read_bnk3side1 | d2_CLK | Single | Separate | DRAM Read Bank3side1 |
| 69 | dram_write_bnk0side0 | d2_CLK | Single | Separate | DRAM Write Bank0side0 |
| 70 | dram_write_bnk1side0 | d2_CLK | Single | Separate | DRAM Write Bank1side0 |
| 71 | dram_write_bnk2side0 | d2_CLK | Single | Separate | DRAM Write Bank2side0 |
| 72 | dram_write_bnk3side0 | d2_CLK | Single | Separate | DRAM Write Bank3side0 |
| 73 | dram_write_bnk0side1 | d2_CLK | Single | Separate | DRAM Write Bank0side1 |
| 74 | dram_write_bnk1side1 | d2_CLK | Single | Separate | DRAM Write Bank1side1 |
| 75 | dram_write_bnk2side1 | d2_CLK | Single | Separate | DRAM Write Bank2side1 |
| 76 | dram_write_bnk3side1 | d2_CLK | Single | Separate | DRAM Write Bank3side1 |
| 77 | dram_partialwrite | d2_CLK | Single | Separate | DRAM Partial Write |
| 78 | dram_rd_data_vld | d2_CLK | Single | Separate | DRAM Read Data Valid |
| 79 | dram_wr_data_vld | d2_CLK | Single | Separate | SeparateDRAM Write Data Valid |

11.5 Software Support

11.5.1 Introduction

An important part of a performance monitoring architecture is the software that allows the counters to be programmed, data to be collected and analyzed. Just as the hardware aspect of the performance monitoring architecture is expected to span generations, the software tools must span generations of product as well. A tool being designed to handle this is known as PLATUNE. PLATUNE is a stand-alone utility that is just starting to be tested. PLATUNE is also designed to be a VTUNE plug in and is scheduled to be included in VTUNE 6.0 (VTUNE 5.0 was released in early 2001).

PLATUNE runs on an IA32 PC under a Microsoft* Windows* operating system, and is appropriate for performance monitoring of PCs. The Intel® IXP2400 network processor is unique in the following respects:

- No Windows operating system; there may be no operating system.
- No disk for recording data.
- No console or GUI.
- No IA32 processor.
- Events monitored that are specific to network processors.

11.5.2 Mode of Operation

One possible way to make use of PLATUNE on network processors is to run PLATUNE standalone on the XScale core. PLATUNE has a script-driven mode that can be used on network processors. The data collected can be kept in the XScale memory or dumped over the PCI interface. Once the data has been collected, VTUNE can transfer it to a PC for analysis with the PLATUNE plug-in.



This page intentionally left blank.