

Intel® Xeon Phi™ Coprocessor System Software Developers Guide

Revision: 2.03
Last Modified: November 8, 2012
IBL Doc ID: 488596

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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 information here is subject to change without notice. Do not finalize a design with this information.

The Intel® MIC Architecture coprocessors described in this document 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.

The code names Knights Ferry, Aubrey Isle, and Knights Corner presented in this document are only for use by Intel to identify products, technologies, or services in development, that have not been made commercially available to the public, i.e., announced, launched or shipped. They are not "commercial" names for products or services and are not intended to function as trademarks.

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 order number and are referenced in this document, Intel literature may be obtained by calling 1-800-548-4725, or go to:
<http://www.intel.com/design/literature.htm>

Intel, the Intel logo, Intel® Pentium®, Intel® Pentium® Pro, Xeon®, Intel® Xeon Phi™, Intel® Pentium® 4 Processor, Intel Core™ Solo, Intel® Core™ Duo, Intel Core™ 2 Duo, Intel Atom™, MMX™, Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX), Intel® VTune™ Amplifier XE 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.

Copyright 2011-2012 Intel Corporation. All rights reserved.

Table of Contents

Table of Contents.....	3
List of Figures.....	7
List of Tables.....	8
1 Introduction	10
1.1 Programming Model	10
1.1.1 Application Programming.....	10
1.1.2 System Programming	10
1.2 Section Overview	11
1.3 Related Technologies and Documents	11
2 Intel® Xeon Phi™ Coprocessor Architecture	13
2.1 Intel® Xeon Phi™ Coprocessor Architecture	13
2.1.1 Core	17
2.1.2 Instruction Decoder.....	19
2.1.3 Cache Organization and Hierarchy.....	19
2.1.4 Page Tables.....	22
2.1.5 Hardware Threads and Multithreading.....	23
2.1.6 Faults and Breakpoints	24
2.1.7 Performance Monitoring Unit and Events Monitor	24
2.1.8 System Interface.....	26
2.1.9 VPU and Vector Architecture	32
2.1.10 Intel® Xeon Phi™ Coprocessor Instructions	33
2.1.11 Multi-Card	33
2.1.12 Host and Intel® MIC Architecture Physical Memory Map.....	34
2.1.13 Power Management.....	35
2.2 Intel® Xeon Phi™ Coprocessor Software Architecture.....	36
2.2.1 Architectural Overview.....	36
2.2.2 Intel® Manycore Platform Software Stack (MPSS).....	38
2.2.3 Bootstrap.....	40
2.2.4 Linux* Loader	41
2.2.5 The Coprocessor Operating System (coprocessor OS).....	42

2.2.6	Symmetric Communication Interface (SCIF)	44
2.2.7	Host Driver.....	44
2.2.8	Sysfs Nodes.....	50
2.2.9	Intel® Xeon Phi™ Coprocessor Software Stack for MPI Applications.....	52
2.2.10	Application Programming Interfaces.....	63
3	Power Management, Virtualization, RAS	64
3.1	Power Management (PM).....	64
3.1.1	Coprocessor OS Role in Power Management	65
3.1.2	Bootloader Role in Power Management.....	66
3.1.3	Host Driver Role in Power Management.....	66
3.1.4	Power Reduction	67
3.1.5	PM Software Event Handling Function	84
3.1.6	Power Management in the Intel® MPSS Host Driver	87
3.2	Virtualization.....	91
3.2.1	Hardware Assisted DMA Remapping	91
3.2.2	Hardware Assisted Interrupt Remapping.....	91
3.2.3	Shared Device Virtualization	91
3.3	Reliability Availability Serviceability (RAS).....	91
3.3.1	Check Pointing.....	92
3.3.2	Berkeley Labs Check point and Restore (BLCR).....	93
3.3.3	Machine Check Architecture (MCA).....	95
3.3.4	Cache Line Disable.....	108
3.3.5	Core Disable	109
3.3.6	Machine Check Flows	109
3.3.7	Machine Check Handler	111
3.3.8	Error Injection.....	111
4	Operating System Support and Driver Writer's Guide.....	113
4.1	Third Party OS Support	113
4.2	Intel® Xeon Phi™ Coprocessor Limitations for Shrink-Wrapped Operating Systems.....	113
4.2.1	Intel x86 and Intel 64 ABI	113
4.2.2	PC-AT / I/O Devices	113
4.2.3	Long Mode Support.....	113

4.2.4	Custom Local APIC.....	113
4.2.5	Custom I/O APIC.....	114
4.2.6	Timer Hardware.....	114
4.2.7	Debug Store.....	114
4.2.8	Power and Thermal Management	114
4.2.9	Pending Break Enable.....	115
4.2.10	Global Page Tables	115
4.2.11	CNXT-ID – L1 Context ID.....	115
4.2.12	Prefetch Instructions.....	115
4.2.13	PSE-36.....	115
4.2.14	PSN (Processor Serial Number)	116
4.2.15	Machine Check Architecture	116
4.2.16	Virtual Memory Extensions (VMX).....	116
4.2.17	CPUID.....	116
4.2.18	Unsupported Instructions	116
5	Application Programming Interfaces.....	119
5.1	The SCIF APIs.....	119
5.2	MicAccessAPI	122
5.3	Support for Industry Standards	125
5.3.1	TCP/IP Emulation.....	126
5.4	Intel® Xeon Phi™ Coprocessor Command Utilities	126
5.5	NetDev Virtual Networking.....	126
5.5.1	Introduction.....	126
5.5.2	Implementation.....	127
6	Compute Modes and Usage Models	129
6.1	Usage Models.....	129
6.2	MPI Programming Models	130
6.2.1	Offload Model	131
6.2.2	Coprocessor-Only Model.....	132
6.2.3	Symmetric Model	132
6.2.4	Feature Summary	133
6.2.5	MPI Application Compilation and Execution.....	134

7	Intel® Xeon Phi™ Coprocessor Vector Architecture	135
7.1	Overview	135
7.2	Vector State	135
7.3	VPU Basic Functions	135
7.4	VPU Data Types	136
7.5	Extended Math Unit	137
7.6	SP FP Operations	137
7.7	DP FP Operations	138
7.8	Vector ISA Overview	138
7.9	Vector Nomenclature	141
7.10	Write Masking	142
7.11	Swizzling	147
7.11.1	Swizzle Limitations	148
7.11.2	Register-Register Swizzle Form	149
7.11.3	Register-Memory Swizzle Form	151
7.11.4	Swizzle Usage Examples	153
7.12	The Shuffle	156
7.13	Memory and Vectors	157
7.13.1	Load and Store Operations	158
7.13.2	Alignment	160
7.13.3	Packing and Unpacking Operations	161
7.13.4	Non-Temporal Data	163
7.13.5	Prefetching	164
7.14	New Instructions	165
7.14.1	Mask Manipulation Instructions	165
7.14.2	Packed Typeless Instructions	166
7.14.3	New Packed SP FP Instructions	167
7.14.4	New Packed Double-Precision Floating-Point Instructions	167
7.14.5	New Packed Int32 Instructions	168

8	Glossary and Abbreviations	169
9	References	173
	Appendix: SBOX Control Register List	174

List of Figures

Figure 2-1. Basic building blocks of the Intel® Xeon Phi™ Coprocessor	14
Figure 2-2: Core Pipeline Components	18
Figure 2-3: Intel® Xeon Phi™ Coprocessor Core Architecture	18
Figure 2-4: MESI Protocol	20
Figure 2-5 Globally Owned Locally Shared (GOLS) Diagram	21
Figure 2-6. Multithreading Architectural Support in the Intel® Xeon Phi™ Coprocessor	23
Figure 2-7. DMA Channel Descriptor Ring plus Local Descriptor Queue	29
Figure 2-8. Descriptor Ring Attributes	30
Figure 2-9. Intel® Xeon Phi™ Coprocessor Address Format.....	30
Figure 2-10. Base Address Width Variations.....	31
Figure 2-11 Head and Tail Pointer Index Registers	31
Figure 2-12. Host and Intel® MIC Architecture Physical Memory Map	34
Figure 2-13. Intel® Xeon Phi™ Coprocessor Software Architecture	37
Figure 2-14. Intel® Xeon Phi™ Coprocessor Software Stack	39
Figure 2-15. The Linux* Coprocessor OS Block Diagram	43
Figure 2-16. Intel® Xeon Phi™ Coprocessor Host Driver Software Architecture Components.....	45
Figure 2-17. Control Panel Software Architecture.....	46
Figure 2-18. Ganglia* Monitoring System Data Flow Diagram	47
Figure 2-19: Ganglia* Monitoring System for a Cluster.....	47
Figure 2-20. Intel® Xeon Phi™ Coprocessor Ganglia* Support Diagram.....	48
Figure 2-21: MPSS Ganglia* Support	51
Figure 2-22 RDMA Transfer with CCL.....	53
Figure 2-23 MPI Application on CCL.....	54
Figure 2-24: OFED*/SCIF Modules	60
Figure 2-25. Supported Communication Fabrics	61
Figure 2-26. Extended SHM Fabric Structure.....	62
Figure 3-1. Intel® Xeon Phi™ Coprocessor Power Management Software Architecture.....	65
Figure 3-2. Power Reduction Flow	68
Figure 3-3. Core C6 Selection	72
Figure 3-4. Package C-state Selection Flow	75
Figure 3-5 CPU Idle State Transitions.....	78
Figure 3-6. Package C-state Transitions	81
Figure 3-7 Package C6 Entry and Exit Flow	83
Figure 3-8 Intel® MPSS Host Driver to Coprocessor OS Package State Interactions	90
Figure 5-1. SCIP Architectural Model	120
Figure 5-2 Intel® Xeon Phi™ Coprocessor SysMgmt MicAccessAPI Architecture Components Diagram	123
Figure 5-3 MicAccessAPI Flash Update Procedure	124
Figure 5-4 Linux* Network Stack	127
Figure 6-1 : A Scalar/Parallel Code Viewpoint of the Intel® MIC Architecture Enabled Compute Continuum	129
Figure 6-2: A Process Viewpoint of the Intel® MIC Architecture Enabled Compute Continuum	129
Figure 6-3: MPI Programming Models for the Intel® Xeon Phi™ Coprocessor	130
Figure 6-4. MPI on Host Devices with Offload to Coprocessors	131
Figure 6-5: MPI on the Intel® Xeon Phi™ coprocessors Only.....	132
Figure 6-6: MPI Processes on Both the Intel® Xeon® Nodes and the Intel® MIC Architecture Devices	133

Figure 6-7. Compiling and Executing a MPI Application	134
Figure 7-1: VPU Registers.....	135
Figure 7-2. Vector Organization When Operating on 16 Elements of 32-bit Data	139
Figure 7-3. Vector Register Lanes 3...0	141
Figure 7-4. Vector Elements D...A Within a Lane	142
Figure 7-5. Vector Elements P...A Across the Entire Vector Register	142
Figure 7-6. Basic Vector Operation Without Write Masking Specified.....	143
Figure 7-7. Basic Vector Operation With a Write Masking Specified	143
Figure 7-8. Effect of Write Mask Values on Result	144
Figure 7-9. A Native (Non-Optimal) Newton-Raphson Approximation Using a Write-Mask to Determine a Square Root.....	146
Figure 7-10. Partial Microarchitecture Design for Swizzle Support	147
Figure 7-11. The Complete Microarchitecture Design for Swizzle Support.....	148
Figure 7-12. Register-Register Swizzle Operation: Source Selection	149
Figure 7-13. Register-Register Swizzle Operation: Element Muxes.....	149
Figure 7-15. Register-Register Swizzle Operation: Complete Swizzle Result.....	150
Figure 7-14. Register-Register Swizzle Operation: First Lane Completion	150
Figure 7-16. The 1-to-16 Register Memory Swizzle Operation.....	151
Figure 7-17. The 4-to-16 Register-Memory Swizzle Operation	152
Figure 7-18. The uint8 Data Conversion Register-Memory Swizzle Operation	153
Figure 7-19. Trivial Implementation of a Horizontal Add Operation Within One Vector Lane	155
Figure 7-20. Trivial Implementation of a 3x3 Matrix Cross-Product Within One Vector Lane	156
Figure 7-21. The Behavior of the Vector Load Instruction With a Write Mask	159
Figure 7-22. The Behavior of the Vector Store Instruction With a Write Mask.....	159
Figure 7-23. Compiler Extension to Force Memory Alignment Boundaries in C or C++ Tools	161
Figure 7-24. The Behavior of the Vector Unpack Instruction	161
Figure 7-25: The Behavior of the Vector Pack Instruction	162

List of Tables

Table 1-1. Related Industry Standards.....	11
Table 1-2. Related Documents.....	11
Table 2-1. Description of Coprocessor Components	16
Table 2-2. L2 Cache States	20
Table 2-3. Tag Directory States	21
Table 2-4. Cache Hierarchy	22
Table 2-5. L1 and L2 Caches Characteristics	23
Table 2-6. Supported and Unsupported Faults on Intel® Xeon Phi™ Coprocessor	24
Table 2-7: Core PMU Instructions	25
Table 2-8. Core PMU Control Registers	25
Table 2-9. Examples of Base Address Ranges Based on Descriptor Ring Size	31
Table 2-10. Coprocessor Memory Map	32
Table 2-11. LSB Core Libraries.....	43
Table 2-12. Intel® MIC Architecture commands	50
Table 2-13. Kernel to User Space Mappings	50
Table 2-14. SYSFS Nodes	51
Table 2-15: Vendor Drivers Bypassing IB* Core for User-Mode Access	55
Table 2-16: Summary of Vendor Driver Characteristics.....	56
Table 3-1. Routines Common to All Package Idle States	77
Table 3-2 Package Idle State Behavior in the Intel® Xeon Phi™ Coprocessor	78
Table 3-3. Events and Conditions Handled by the Coprocessor OS.....	86
Table 3-4. Power Management Messages.....	88

Table 3-5. Control and Error Reporting Registers	98
Table 3-6. MCI_CTL Register Description	98
Table 3-7. MCI_STATUS Register Description	99
Table 3-8. MCI_ADDR Register Description	100
Table 3-9. Machine Check Registers	100
Table 3-10. Sources of Uncore Machine-Check Events	103
Table 3-11. SBox Machine Check Registers	103
Table 3-12. SBox Error Descriptions.....	104
Table 3-13. Correctable PCIe Fabric Error Signal	106
Table 3-14. Uncorrectable PCIe Fabric Error Signal	106
Table 3-15. GBox Errors	106
Table 3-16. TD Errors	107
Table 3-17. GBox Error Registers	108
Table 5-1 Summary of SCIF Functions.....	121
Table 5-2. MicAccessAPI Library APIs	125
Table 5-3. Intel® Xeon Phi™ Coprocessor Command Utilities	126
Table 7-1 Bidirectional Up/Down Conversion Table.....	137
Table 7-2 Throughput Cycle of Transcendental Functions	137
Table 7-3. The Scale-and-Bias Instruction vfmadd233ps on 32-bit Data.....	140
Table 7-4. The vmovaps Instruction Support for Data Conversion and Data Replication Modifiers.....	158
Table 7-5. The Floating-Point Data Type Encodings Supported.....	159
Table 7-6: Memory Alignment Requirements for Load and Store Operations.....	160
Table 7-7. L1 Prefetch Instructions	164
Table 7-8. L2 Prefetch Instructions	165
Table 7-9. Mask Manipulation Instructions	165
Table 7-10. Packed Typeless Instructions	166
Table 7-11. New Packed DP FP Instructions	167
Table 7-12. Packed Int32 Instructions.....	168

1 Introduction

1.1 Programming Model

As with most computing systems, the Intel® Many Integrated Core (Intel® MIC) Architecture programming model can be divided into two categories: application programming and system programming.

1.1.1 Application Programming

In this guide, application programming refers to developing user applications or codes using either the Intel® Composer XE 2013 or 3rd party software development tools. These tools typically contain a development environment that includes compilers, libraries, and assorted other tools.

Application programming will not be covered here; consult the Intel® Xeon Phi™ Coprocessor DEVELOPER'S QUICK START GUIDE for information on how to quickly write application code and run applications on a development platform including the Intel® Many Integrated Core Architecture (Intel® MIC Architecture). It also describes the available tools and gives some simple examples to show how to get C/C++ and Fortran-based programs up and running.

The development environment includes the following compilers and libraries, which are available at <https://registrationcenter.intel.com>:

- Intel® C/C++ Compiler XE 2013 including Intel® MIC Architecture for building applications that run on Intel® 64 and Intel® MIC Architectures
- Intel® Fortran Compiler XE 2013 including Intel® MIC Architecture for building applications that run on Intel® 64 and Intel® MIC Architectures

Libraries for use with the offload compiler include:

- Intel® Math Kernel Library (Intel® MKL) optimized for Intel® MIC Architecture
- Intel® Threading Building Blocks

The development environment includes the following tools:

- Debugger
 - Intel® Debugger for applications including Intel® MIC Architecture
 - Intel® Debugger for applications running on Intel® Architecture (IA)
 -
- Profiling
 - SEP enables performance data collection from the Intel® Xeon Phi™ coprocessor. This feature is included as part of the VTune™ Amplifier XE 2013 tool.
 - Performance data can be analyzed using VTune™ Amplifier XE 2013

1.1.2 System Programming

System programming here explains how to use the Intel® MIC Architecture, its low level APIs (e.g. SCIF), and the contents of the Intel® Many Integrated Core Architecture Platform Software Stack (MPSS). Detailed information on these low-level APIs can be found in Section 5 of this document.

1.2 Section Overview

The information in this guide is organized as follows:

- Section 2 contains a high-level description of the Intel® Xeon Phi™ coprocessor hardware and software architecture.
- Section **Error! Reference source not found.** covers power management from the software perspective. It also covers virtualization support in the Intel® Xeon Phi™ coprocessor and some Reliability Accessibility and Serviceability (RAS) features such as BLCR* and MCA.
- Section 4 covers Operating System support.
- Section 5 covers the low level APIs (e.g. SCIF) available with the Intel® Xeon Phi™ coprocessor software stack.
- Section 6 illustrates the usage models and the various operating modes for platforms with the Intel® Xeon Phi™ coprocessors in the compute continuum.
- Section 7 provides in-depth details of the Intel® Xeon Phi™ coprocessor Vector Processing Unit architecture.
- Glossary of terms and abbreviations used can be found in Section 8.
- References are collated in Section 9.

1.3 Related Technologies and Documents

This section lists some of the related documentation that you might find useful for finding information not covered here.

Industry specification for standards (i.e., OpenMP*, OpenCL*, MPI, OFED*, and POSIX* threads) are not covered in this document. For this information, consult relevant specifications published by their respective owning organizations:

Table 1-1. Related Industry Standards

Technology	Location
OpenMP*	http://openmp.org/
OpenCL*	http://www.khronos.org/opencv/
MPI	http://www.mpi-forum.org/
OFED* Overview	http://www.openfabrics.org/

You should also consult relevant published documents which cover the Intel® software development tools not covered here:

Table 1-2. Related Documents

Document	Location
Intel® Xeon Phi™ Coprocessor DEVELOPER'S QUICK START GUIDE	http://software.intel.com/en-us/mic-developer
Intel® Many Integrated Core Platform Software Stack	http://software.intel.com/en-us/mic-developer
Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual	http://software.intel.com/en-us/mic-developer
An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors	http://software.intel.com/en-us/mic-developer
Debugging Intel® Xeon Phi™ Coprocessor: Command-Line Debugging	http://software.intel.com/en-us/mic-developer
Building Native Applications for Intel® Xeon Phi™ Coprocessor	http://software.intel.com/en-us/mic-developer
Programming and Compiling for Intel® Many Integrated Core Architecture	http://software.intel.com/en-us/mic-developer

Document	Location
Intel® Xeon Phi™ coprocessor Micro-architecture Software Stack	http://software.intel.com/en-us/mic-developer
Intel® Xeon Phi™ coprocessor Micro-architecture Overview	http://software.intel.com/en-us/mic-developer
Intel® MPI Library	http://www.intel.com/go/mpi
Intel® MIC SCIF API Reference Manual for Kernel Mode Linux*	http://intel.com/software/mic
Intel® MIC SCIF API Reference Manual for User Mode Linux*	http://intel.com/software/mic

2 Intel® Xeon Phi™ Coprocessor Architecture

This Section explains both the hardware and the software architecture of the Intel® Xeon Phi™ coprocessor. It covers the major micro-architectural features such as the core, the vector processing unit (VPU), the high-performance on-die bidirectional interconnect, fully coherent L2 caches, and how the various units interact. Particular emphasis is placed on the key parameters necessary to understand program optimization, such as cache organization and memory bandwidth.

2.1 Intel® Xeon Phi™ Coprocessor Architecture

The Intel® Xeon Phi™ coprocessor comprises of up to sixty-one (61) processor cores connected by a high performance on-die bidirectional interconnect. In addition to the IA cores, there are 8 memory controllers supporting up to 16 GDDR5 channels delivering up to 5.5 GT/s, and special function devices such as the PCI Express* system interface.

Each core is a fully functional, in-order core, which supports fetch and decode instructions from four hardware thread execution contexts. In order to reduce hot-spot contention for data among the cores, a distributed tag directory is implemented so that every physical address the coprocessor can reach is uniquely mapped through a reversible one-to-one address hashing function. This hashing function not only maps each physical address to a tag directory, but also provides a framework for more elaborate coherence protocol mechanisms than the individual cores could provide.

Each memory controller is based on the GDDR5 specification, and supports two channels per memory controller. At up to 5.5 GT/s transfer speed, this provides a theoretical aggregate bandwidth of 352 GB/s (gigabytes per second) directly connected to the Intel® Xeon Phi™ coprocessor.

At a high level, Intel® Xeon Phi™ coprocessor silicon consists of up to 61 dual-issue in-order cores, where each core includes:

- 512 bit wide vector processor unit (VPU)
- The Core Ring Interface (CRI)
- Interfaces to the Core and the Ring Interconnect
- The L2 Cache (including the tag, state, data and LRU arrays) and the L2 pipeline and associated arbitration logic
- The Tag Directory (TD) which is a portion of the distributed duplicate tag directory infrastructure
- Asynchronous Processor Interrupt Controller (APIC) which receives interrupts (IPIs, or externally generated) and must redirect the core to respond in a timely manner.
 - Memory controllers (GBOX), which access external memory devices (local physical memory on the coprocessor card) to read and write data. Each memory controller has 2 channel controllers, which together can operate two 32-bit memory channels.
 - A Gen2 PCI Express* client logic (SBOX), which is the system interface to the host CPU or PCI Express* switch, supporting x8 and x16 configurations.
 - The Ring Interconnect connecting all of the aforementioned components together on the chip.

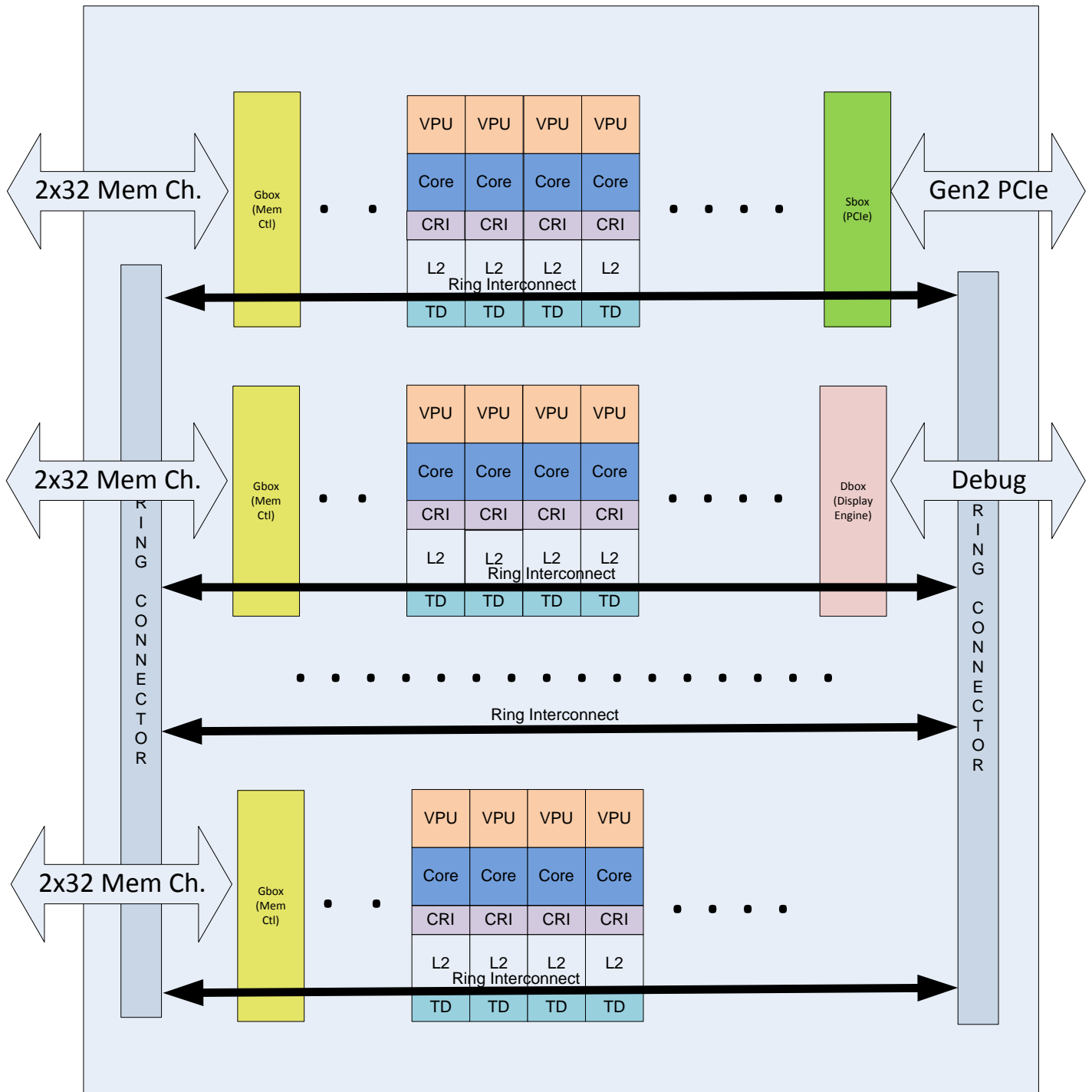


Figure 2-1. Basic building blocks of the Intel® Xeon Phi™ Coprocessor

Table 2-1 gives a high-level description of each component.

Table 2-1. Description of Coprocessor Components

Name	Description
Core	The processor core. It fetches and decodes instructions from four hardware thread execution contexts. It supports a 32-bit and 64-bit execution environment similar to those found in the Intel64® Intel® Architecture Software Developer’s Manual, along with the Intel Initial Many Core Instructions. . It contains a 32KB, 8-Way set associative L1 Icache and Dcache, and interfaces with the CRI/L2 block to request access to memory. The core can execute 2 instructions per clock cycle, one on the U-pipe, and one on the V-pipe. The V-pipe cannot execute all instruction types, and simultaneous execution is governed by pairing rules. The core does not support Intel® Streaming SIMD Extensions (Intel® SSE) or MMX™ instruction execution.
VPU	The Vector Processor Unit includes the EMU (extended math unit) and executes 16 single-precision floating point, 16 32bit integer operations per clock cycle, or 8 double-precision floating-point operations per cycle. Each operation can be a floating-point multiply-add, giving 32 single precision floating-point operations per cycle. The VPU contains the vector register file (32 registers per thread context), and can read one of its operands directly from memory, including data format conversion on the fly. Broadcast and swizzle instructions are also available. The EMU can perform base-2 exponential, base-2 logarithm, reciprocal, and reciprocal square root of single precision floating-point values.
L2/CRI	The Core-Ring Interface hosts the 512KB, 8-way, L2 cache and connects each core to an Intel® Xeon Phi™ coprocessor Ring Stop. Primarily, it comprises the core-private L2 cache itself plus all of the off-core transaction tracking queues and transaction / data routing logic. Two other major blocks also live in the CRI: the R-Unit (APIC) and the Tag Directory (TD).
TD	Distributed duplicate tag directory for cross-snooping L2 caches in all cores. The CPU L2 caches are kept fully coherent with each other by the TDs, which are referenced after an L2 cache miss. A TD tag contains the address, state, and an ID for the owner (one of the L2 caches) of the cache line. The TD that is referenced is not necessarily the one co-located with the core that generated the miss, but is based upon address (each TD gets an equal portion of the address space). A request is sent from the core that suffered the memory miss to the correct TD via the ring interconnect.
GBOX	The Intel® Xeon Phi™ coprocessor memory controller comprises three main units: the FBOX (interface to the ring interconnect), the MBOX (request scheduler) and the PBOX (physical layer that interfaces with the GDDR devices). The MBOX comprises two CMCs (or Channel Memory Controllers) that are completely independent from each other. The MBOX provides the connection between agents in the system and the DRAM I/O block. It is connected to the PBOX and to the FBOX. Each CMC operates independently from the other CMCs in the system.
SBOX	PCI Express* client logic: DMA engine, limited power management capabilities
Ring	Ring Interconnect, including component interfaces, ring stops, ring turns, addressing, and flow control. Intel® Xeon Phi™ coprocessor has 2 each of these rings – one travelling each direction. There is no queuing on the ring or in the ring turns; once a message is on the ring it will continue deterministically to its destination. In some cases, the destination does not have room to accept the message and may leave it on the ring and pick it up the next time it goes by. This is known as bouncing.

Name	Description
PBOX	The PBOX is the analog interface component of the GBOX that communicates with the GDDR memory device. Besides the analog blocks, the PBOX contains the input/output FIFO buffers, part of the training state machines and mode registers to trim the analog interface. The analog interface consists of the actual I/O pads for DQs, Address and Command and the clocking structure. The PBOX also includes the GPLL which defines the clock domain for each PBOX and the respective MBOX/CBOX.
PMU	Performance Monitoring Unit. This performance monitoring feature allows data to be collected from all units in the architecture, utilizing a P6-style programming interface to configure and access performance counters. Implements an Intel® Xeon Phi™ coprocessor SPFLT which allows user-level code to filter the core events that its thread generates. Does not implement some advanced features found in mainline IA cores (e.g. precise event-based sampling, etc.).
Clock	The clock generation on Intel® Xeon Phi™ coprocessor supplies clocks to each of the four main clock domains. The core domain supports from 600 MHz to the part's maximum frequency in steps of 25 MHz. Ratio changes in the core happen seamlessly and can be controlled through both software and internal hardware (using information from the thermal and current sensors on the card.) The GDDR supports frequencies that enable between 2.8 GT/s and the part's maximum frequency with a minimum step size of 50 MT/s. Intel® Xeon Phi™ coprocessors support frequency changes without requiring a reset. PCI Express* clock modes support both Gen1 and Gen2 operation. The external clock buffer has been incorporated into the Intel® Xeon Phi™ coprocessor die, and the clocks are sourced from two 100 MHz PCI Express* reference clocks.

2.1.1 Core

Each in-order execution core provides a 64 bit execution environment similar to that found in the Intel64® Intel® Architecture Software Developer's Guide, in addition to introducing support for Intel Initial Many Core Instructions. There is no support for MMX™ instructions, Intel Advanced Vector Extensions (Intel® AVX), or any of the Intel® Streaming SIMD Extensions (Intel® SSE). A full list of the instructions supported by the Intel® Xeon Phi™ coprocessor can be found in the following document (Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual (Reference Number: 327364)). New vector instructions provided by the Intel® Xeon Phi™ Coprocessor Instruction Set utilize a dedicated 512-bit wide vector floating-point unit (VPU) that is provided for each of the cores.

Each core is connected to a Ring Interconnect via the Core Ring Interface (CRI), which is comprised of the L2 cache control and the Tag Directory (TD). The Tag Directory contains the tags for a portion of the overall L2 cache. The Core and L2 Slices are interconnected on a ring based interconnect along with additional ring agents on the die. Each agent on the ring, whether a core/L2 Slice, memory controller, or the system (SBOX), implements a ring stop that enables requests and responses to be sent on the ring bus.

The core can execute 2 instructions per clock cycle, one on the U-pipe and one on the V-pipe. The V-pipe cannot execute all instruction types, and simultaneous execution is governed by pairing rules. Vector instructions can only be executed on the U-pipe.

Core Pipeline



PPF	Thread picker	D2	Microcode control execution Address generation Data cache lookup Register file read
PF	Instruction cache lookup Prefetch buffer write	E	Integer ALU execution Retire/stall/exception determination
D0	Thread picker Instruction rotate Decode of 0f, 62, D6, REX prefixes	WB	Integer register file write Condition code (flag) evaluation
D1	Instruction Decode CROM lookup Sunit register file read		

Figure 2-2: Core Pipeline Components

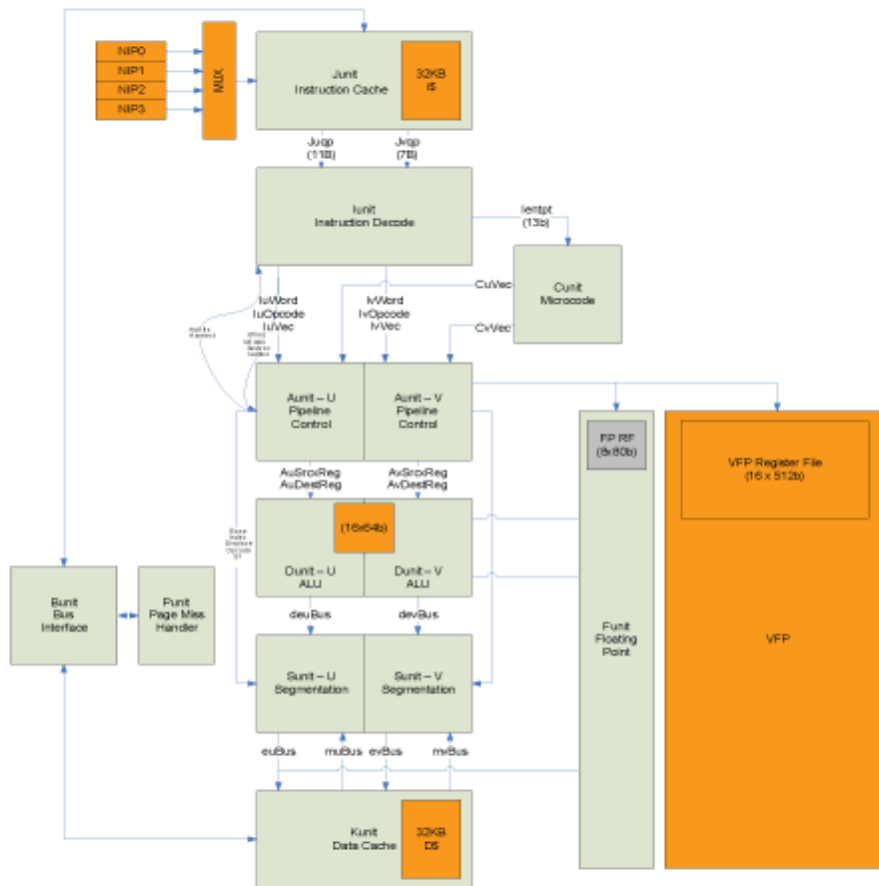


Figure 2-3: Intel® Xeon Phi™ Coprocessor Core Architecture

Most integer and mask instructions have a 1-clock latency, while most vector instructions have 4-clock latency with a 1 clock throughput. Dependent store- to-load latency is 4 clocks for simple vector operations. “Shuffles” and “Swizzles” increase this latency. The store-to-load penalty for the L1 is approximately 12 clocks. Kunit (data cache) bounces cause 2 dead clocks (bank conflicts, U-pipe/V-pipe conflicts with higher-priority replacements, invalidations). Prefix decodes are available with 0-cycle “fast”: 62, c4, c5, REX, 0f, and a 2-cycle “slow”: operand size 66, address size 67, lock, segment, REP.

2.1.2 Instruction Decoder

One of the changes made to simplify the core was to modify the instruction decoder to be a two-cycle unit. While fully pipelined, the result of this change is that the core cannot issue instructions from the same hardware context in back-to-back cycles. That is, if in cycle N the core issued instructions from context 1, then in cycle N +1 the core can issue instructions from any context except context 1. This allows for a significant increase in the maximum core frequency, resulting in a net performance gain even for single-threaded SPEC* benchmarks.

For maximum chip utilization, at least two hardware contexts or threads must be run on each core. Since the scheduler cannot issue instructions in back-to-back cycles from the same hardware context, running one thread on a core will result in, at best, 50% utilization of the core potential.

2.1.3 Cache Organization and Hierarchy

The Level One (L1) cache accommodates higher working set requirements for four hardware contexts per core. It has a 32 KB L1 instruction cache and 32 KB L1 data cache. Associativity was increased to 8-way, with a 64 byte cache line. The bank width is 8 bytes. Data return can now be out-of-order. The L1 cache has a load-to-use latency of 1 cycle -- an integer value loaded from the cache can be used in the next clock by an integer instruction. Note, however, that vector instructions experience different latencies than integer instructions. The L1 cache has an address generation interlock with at least a 3-clock cycle latency. A GPR register must be produced three or more clocks prior to being used as a base or index register in an address computation. The register set-up time for base and index has the same 3-clock cycle latency.

Another new feature is the 512 KB unified Level Two (L2) cache unit. The L2 organization comprises 64 bytes per way with 8-way associativity, 1024 sets, 2 banks, 32GB (35 bits) of cacheable address range and a raw latency of 11 clocks. The expected idle access time is approximately 80 cycles. The L2 cache has a streaming hardware prefetcher that can selectively prefetch code, read, and RFO (Read-For-Ownership) cache lines into the L2 cache. There are 16 streams that can bring in up to a 4-KB page of data. Once a stream direction is detected, the prefetcher can issue up to 4 multiple prefetch requests. The L2 in Intel® Xeon Phi™ coprocessor supports ECC, and power states such as the core C1 (shuts off clocks to the core and the VPU), C6 (shuts off clocks and power to the core and the VPU), and the package C3 states. The replacement algorithm for both the L1 and L2 caches is based on a pseudo-LRU implementation.

The L2 cache is part of the Core-Ring Interface block. This block also houses the tag directory (TD) and the Ring Stop (RS), which connects to the interprocessor core network. Within these sub-blocks is the Transaction Protocol Engine which is an interface to the RS and is equivalent to a front side bus unit. The RS handles all traffic coming on and off the ring. The TDs, which are physically distributed, filter and forward requests to appropriate agents on the ring. They are also responsible for initiating communications with the GDDR5 memory via the on-die memory controllers.

In the in-order Intel® Pentium® processor design, any miss to the cache hierarchy would be a core-stalling event such that the program would not continue executing until the missing data were fetched and ready for processing. In the Intel® Xeon Phi™ coprocessor cores, a miss in the L1 or L2 cache does not stall the entire core. Misses to the cache will not stall the requesting hardware context of a core unless it is a load miss. Upon encountering a load miss, the hardware context with the instruction triggering the miss will be suspended until the data are brought into the cache for processing. This allows the other hardware contexts in the core to continue execution. Both the L1 and L2 caches can also support up to about 38 outstanding requests per core (combined read and write). The system agent (containing the

PCI Express* agent and the DMA controller) can also generate 128 outstanding requests (read and write) for a total of 38*(number of cores) + 128. This allows software to prefetch data aggressively and avoids triggering a dependent stall condition in the cache. When all possible access routes to the cache are in use, new requests may cause a core stall until a slot becomes available.

Both the L1 and L2 caches use the standard MESI protocol for maintaining the shared state among cores. The normal MESI state diagram is shown in Figure 2-4 and the cache states are listed in Table 2-2. L2 Cache States.

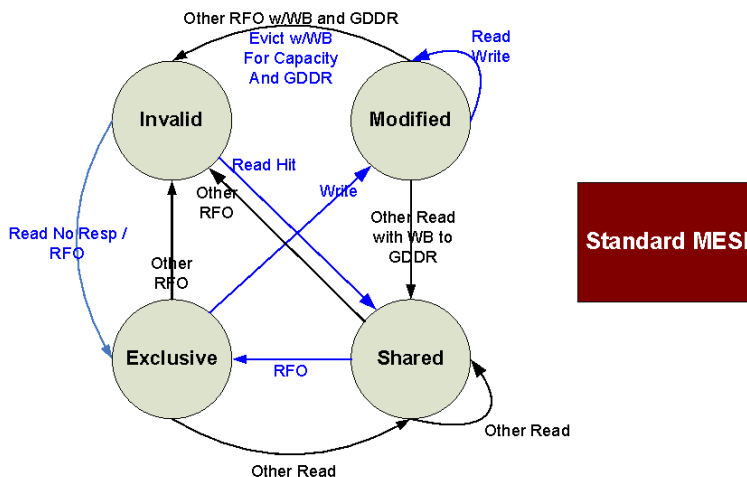


Figure 2-4: MESI Protocol

Table 2-2. L2 Cache States

L2 Cache State	State Definition
M	Modified. Cacheline is updated relative to memory (GDDR). Only one core may have a given line in M-state at a time.
E	Exclusive. Cacheline is consistent with memory. Only one core may have a given line in E-state at a time.
S	Shared. Cacheline is shared and consistent with other cores, but may not be consistent with memory. Multiple cores may have a given line in S-state at the same time.
I	Invalid. Cacheline is not present in this core’s L2 or L1.

To address potential performance limitations resulting from the lack of an O (Owner) state found in the MOESI protocol, the Intel® Xeon Phi™ coprocessor coherence system has an ownership tag directory (TD) similar to that implemented in many multiprocessor systems. The tag directory implements the GOLS3 protocol. By supplementing the individual core MESI protocols with the TD’s GOLS protocol, it becomes possible to emulate the missing O-state and to achieve the benefits of the full MOESI protocol without the cost of redesigning the local cache blocks. The TD is also useful for controlling other behaviors in the Intel® Xeon Phi™ coprocessor design and is used for more than this emulation behavior. The modified coherence diagrams for the core MESI protocol and the tag directory GOLS protocol are shown in Figure 2-5.

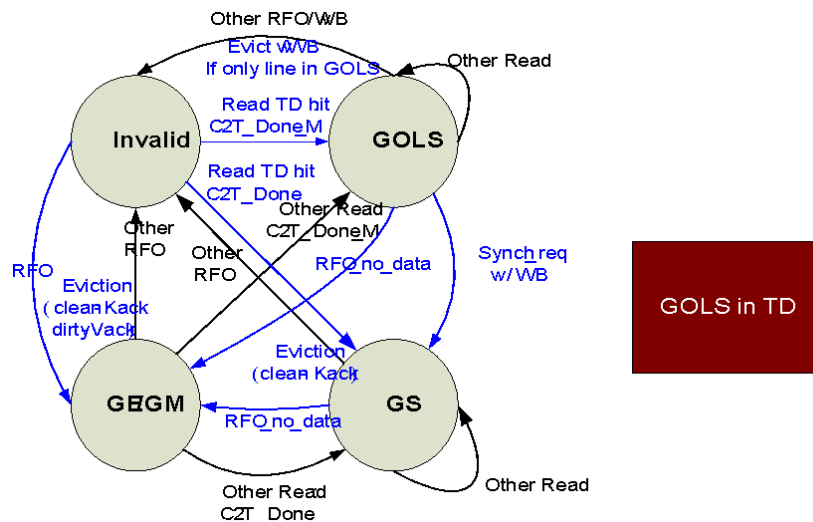
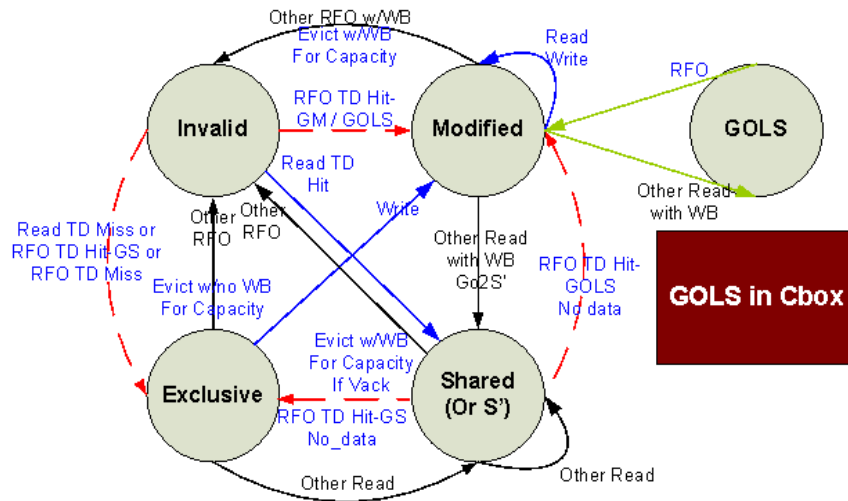


Figure 2-5 Globally Owned Locally Shared (GOLS) Diagram

Table 2-3. Tag Directory States

Tag Directory State	State Definition
GOLS	Globally Owned, Locally Shared. Cacheline is present in one or more cores, but is not consistent with memory.
GS	Globally Shared. Cacheline is present in one or more cores and consistent with memory.
GE/GM	Globally Exclusive/Modified. Cacheline is owned by one and only one core and may or may not be consistent with memory. The Tag Directory does not know whether the core has actually modified the line.
GI	Globally Invalid. Cacheline is not present in any core.

The tag directory is not centralized but is broken up into 64 distributed tag directories (DTDs). Each DTD is responsible for maintaining the global coherence state in the chip for its assigned cache lines. The basic L1 and L2 cache parameters are summarized in Table 2-4. Two unusual fields in this table are the Duty Cycle and Ports designations, which are specific only to the Intel® Xeon Phi™ coprocessor design. The L1 cache can be accessed each clock, whereas the L2 can only be accessed every other clock. Additionally, on any given clock software can either read or write the L1 or L2, but it cannot read and write in the same clock. This design artifact has implications when software is trying to access a cache while evictions are taking place.

Table 2-4. Cache Hierarchy

Parameter	L1	L2
Coherence	MESI	MESI
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line Size	64 bytes	64 bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU
Duty Cycle	1 per clock	1 per clock
Ports	Read or Write	Read or Write

The L2 cache organization per core is inclusive of the L1 data and instruction caches. How all cores work together to make a large, shared, L2 global cache (up to 31 MB) may not be clear at first glance. Since each core contributes 512 KB of L2 to the total shared cache storage, it may appear as though a maximum of 31 MB of common L2 cache is available. However, if two or more cores are sharing data, the shared data is replicated among the individual cores' various L2 caches. That is, if no cores share any data or code, then the effective total L2 size of the chip is 31 MB. Whereas, if every core shares exactly the same code and data in perfect synchronization, then the effective total L2 size of the chip is only 512 KB. The actual size of the workload-perceived L2 storage is a function of the degree of code and data sharing among cores and thread.

A simplified way to view the many cores in Intel® Xeon Phi™ coprocessor is as a chip-level symmetric multiprocessor (SMP). Each core acts as a stand-alone core with 512 KB of total cache space, and up to 62 such cores share a high-speed interconnect on-die. While not particularly accurate compared to a real SMP implementation, this simple mental model is useful when considering the question of how much total L2 capacity may be used by a given workload on the Intel® Xeon Phi™ coprocessor card.

2.1.4 Page Tables

The Intel® Xeon Phi™ coprocessor supports 32-bit physical addresses in 32-bit mode, 36-bit physical address extension (PAE) in 32-bit mode, and 40-bit physical address in 64-bit mode.

It supports 4-KB and 2-MB page sizes. It also supports the Execute Disable (NX) bit. But there is no support for the Global Page bit, unlike other Intel® Architecture microprocessors. On a TLB miss, a four-level page table walk is performed as usual, and the INVLPG instruction works as expected. The advantage of this approach is that there are no restrictions for mixing the page sizes (4 KB, 2MB) within a single address block (2MB). However, undefined behavior will occur if the 16 underlying 4-KB page-table entries are not consistent.

Each L1 data TLB (dTLB) has 64 entries for 4 KB pages and 8 entries for 2MB pages. Each core also has one instruction TLB (iTLB), which only has 32 entries for 4 KB pages. No support for larger page sizes is present in the instruction TLB. For L2, the 4-way dTLB has 64 entries, usable as second-level TLB for 2M pages or as a page directory entry (PDE) cache for

4K. TLBs can share entries among threads that have the same values for the following registers: CR3, CR0.PG, CR4.PAE, CR4.PSE, EFER.LMA.

Table 2-5. L1 and L2 Caches Characteristics

	Page Size	Entries	Associativity	Maps
L1 Data TLB	4K	64	4-way	256K
	2M	8	4-way	16M
L1 Instruction TLB	4K	32	4-way	128K
L2 TLB	4K, 2M	64	4-way	128M

The Intel® Xeon Phi™ coprocessor core implements two types of memory: uncacheable (UC) and write-back (WB). The other three memory forms [write-through (WT), write-combining (WC), and write-protect (WP)] are mapped internally to microcontroller behavior. No other memory type is legal or supported.

2.1.5 Hardware Threads and Multithreading

Figure 2-6 presents a high-level view of the major impacts for hardware multithreading support, such as architectural, pipeline, and cache interactions. This includes replicating complete architectural state 4 times: the GPRs, ST0-7, segment registers, CR, DR, EFLAGS, and EIP. Certain micro-architectural states are also replicated four times like the prefetch buffers, the instruction pointers, the segment descriptors, and the exception logic. “Thread specific” changes include adding thread ID bits to shared structures (iTLB, dTLB, BTB), converting memory stall to thread-specific flush, and the introduction of thread wakeup/sleep mechanisms through microcode and hardware support. Finally, the Intel® Xeon Phi™ coprocessor implements a “smart” round-robin multithreading.

Multithreading

Time-multiplexed multithreading

Add two thread pickers (PPF, PF)

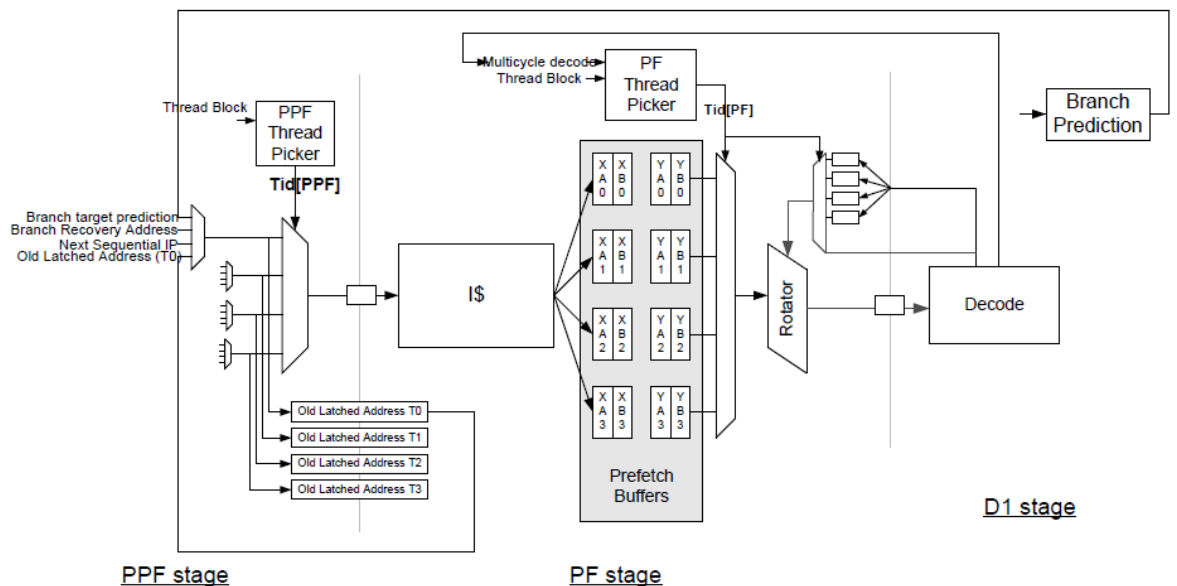


Figure 2-6. Multithreading Architectural Support in the Intel® Xeon Phi™ Coprocessor

Each of four hardware threads shown above in the grey shaded region has a “ready to run” buffer consisting of two instruction bundles. Since each core is capable of issuing two instructions per clock cycle, each bundle represents two

instructions. If the executing thread has a control transfer to a target that is not contained in this buffer, it will trigger a miss to the instruction cache, which flushes the context buffer and loads the appropriate target instructions. If the instruction cache does not have the control transfer point, a core stall will be initiated, which may result in performance penalties. In general, whichever hardware context issues instructions in a given clock cycle has priority for fetching the next instruction(s) from the instruction cache. Another significant function is the picker function (PF) that chooses the next hardware context to execute. The PF behaves in a round-robin manner, issuing instructions during any one clock cycle from the same hardware context only. In cycle N, if the PF issues instruction(s) from Context 3, then in cycle N + 1 the PF will try to issue instructions from Context 0, Context 1, or Context 2 – in that order. As previously noted it is not possible to issue instructions from the same context (Context 3 in this example) in back-to-back cycles.

2.1.6 Faults and Breakpoints

The Intel® Xeon Phi™ coprocessor supports the fault types shown in Table 2-6 below. For complete details of fault behavior, please consult the (Intel® 64 and IA-32 Architectures Software Developer Manuals).

Breakpoint support required the widening of DR0-DR3 for Intel® 64 instruction compatibility and is now for 1, 2, 4, or 8 bytes. The length was not extended to support 16, 32, or 64 bytes. Also, breakpoints in the Intel® Xeon Phi™ coprocessor instructions occur regardless of any conditional execution status indicated by mask registers.

Table 2-6. Supported and Unsupported Faults on Intel® Xeon Phi™ Coprocessor

Fault Type	Supported	Comments
#PF	Yes	Page Fault
#SS	Yes	For non-canonical and referencing SS segment
#GP	Yes	Address is not canonical or not aligned to operand size
#UD	Yes	If CR0.EM[2] = 1, or LOCK or REX prefix used; also triggered on IN or OUT instructions
#XF	No	No unmasked exceptions in SIMD
#AC	No	GP fault always takes priority
#NM	No	CR0.TS[3] = 1

2.1.7 Performance Monitoring Unit and Events Monitor

The Intel® Xeon Phi™ coprocessor includes a performance monitoring unit (abbreviated as PMU) like the original Intel® Pentium® processor core. Most of the 42 event types from the original Intel® Pentium® processor exist, although the PMU interface has been updated to reflect more recent programming interfaces. Particular Intel® Xeon Phi™ coprocessor-centric events have been added to measure memory controller events, vector processing unit utilization and statistics, local and remote cache read/write statistics, and more.

The Intel® Xeon Phi™ coprocessor comes with support for performance monitoring at the individual core level. Each core has four performance counters, four filtered counters, and four event select registers. The events supported for performance monitoring are a combination of the legacy Intel® Pentium® processor events and new Intel® Xeon Phi™ coprocessor-centric events. Each core PMU is shared amongst all four hardware threads in that core. The PMU in each core is responsible for maintaining the time stamp counter (TSC) and counts hardware events generated by the core or triggered by events arriving at the core. By default, events are counted for all hardware contexts in the core, but the filters may be set to count only specific hardware context events. The core PMU also receives events from co-located units, including the ring stop, the distributed tag directory, and the core-ring interface.

The Intel® Xeon Phi™ coprocessor switched to the Intel® Pentium® Pro processor style of PMU interface, which allows user-space (ring three) applications to directly interface with and use the PMU features via specialized instructions such as RDPMC4. In this model, Ring 0 still controls the PMU but Ring 3 is capable of interacting with exposed features for optimization.

Table 2-7 lists the instructions used by Ring 0 and Ring 3 code used to control and query the core PMU.

Table 2-7: Core PMU Instructions

Instruction Name	Description	Privilege Mode (CPL)	Thread-Specific	Input	Output
RDMSR	Read model specific register. Used by Ring 0 code to read any core PMU register.	Ring 0	Yes	ECX: Address of MSR	EDX:EAX = 64-bit MSR value
WRMSR	Write model specific register. Used by Ring 0 code to write to any core PMU register.	Ring 0	Yes	EDX:EAX = 64-bit MSR value ECX: Address of MSR	None
RDTSC	Read timestamp counter. Reads the current timestamp counter value.	Ring 0-3	No	None	EDX:EAX = 64-bit timestamp value
RDPMC	Read performance-monitoring counter. Reads the counts of any of the performance monitoring counters, including the PMU filtered counters.	Ring 0-3	Yes	ECX: Counter # 0x0: IA32_PerfCntr0 0x1: IA32_PerfCntr1	EDX:EAX = Zero-extended 40-bit counter value
SPFLT	Set user preference flag to indicate counter enable/disable.	Ring 0-3	Yes	Any GPR[0]: 0x0: Clear (disable) 0x1: Set (enable)	Set/clear USER_PREF bit in PERF_SPFLT_CONTROL.

The instructions RDMSR, WRMSR, RDTSC, and RDPMC are well-documented in the (Intel® 64 and IA-32 Architectures Software Developer Manuals). The only Intel® MIC Architecture-specific notes are that RDTSC has been enhanced to execute in 4-5 clock cycles and that a mechanism has been implemented to synchronize timestamp counters across the chip.

SPFLT is unique because it allows software threads fine-grained control in enabling/disabling the performance counters. The anticipated usage model for this instruction is for instrumented code to enable/disable counters around desired portions of code. Note that software can only specify its preference for enabling/disabling counters and does not have control over which specific counters are affected (this behavior supports virtualization). The SPFLT instruction can only be executed while the processor is in Intel® 64-bit mode.

Table 2-8 lists the model-specific registers used to program the operation of the core PMU.

Table 2-8. Core PMU Control Registers

Register Address		Name	Description	Threaded?	Width
Hex	Dec				
0x10	16	MSR_TIME_STAMP_COUNTER	Timestamp Counter	No	64
0x20	32	MSR_PerfCntr0	Events Counted, core PMU counter 0	Yes	40
0x21	33	MSR_PerfCntr1	Events Counted, core PMU counter 1	Yes	40
0x28	40	MSR_PerfEvtSel0	Performance Event Selection and configuration register for IA32_PerfCntr0.	Yes	32
0x29	41	MSR_PerfEvtSel1	Performance Event Selection and configuration register for IA32_PerfCntr1.	Yes	32
0x2C	44	MSR_PERF_SPFLT_CONTROL	SPFLT Control Register. This MSR controls the effect of the SPFLT instruction and whether it will allow software fine-grained control to enable/disable IA32_PerfCntrN.	Yes	64

Register Address		Name	Description	Threaded?	Width
0x2D	45	MSR_PERF_GLOBAL_STATUS	Counter Overflow Status. This read-only MSR displays the overflow status of all the counters. Each bit is implemented as a sticky bit, set by a counter overflow.	Yes	32
0x2E	46	MSR_PERF_GLOBAL_OVF_CTRL	Counter Overflow Control. This write-only MSR clears the overflow indications in the Counter Overflow Status register. For each bit that is set, the corresponding overflow status is cleared.	Yes	32
0x2F	47	MSR_PERF_GLOBAL_CTRL	Master PMU Enable. Global PMU enable / disable. When these bits are set, the core PMU is permitted to count events as configured by each of the Performance Event Selection registers (which can each be independently enabled or disabled). When these bits are cleared, performance monitoring is disabled. The operation of the Timestamp Counter is not affected by this register.	Yes	32

For a description of the complete set of Intel® Xeon Phi™ coprocessor PMU and EMON registers and its performance monitoring facilities, please see the document (Intel® Xeon Phi™ Coprocessor Performance Monitoring Units, Document Number: 327357-001, 2012).

2.1.7.1 Timestamp Counter (TSC)

The RDTSC instruction that is used to access IA32_TIMESTAMP_COUNTER can be enabled for Ring 3 (user code) by setting CR4[2].

This behavior enables software (including user code) to use IA32_TIMESTAMP_COUNTER as a wall clock timer. The Intel® Xeon Phi™ coprocessor only supports this behavior in a limited configuration (P1 only) and not across different P-states. The Intel® Xeon Phi™ coprocessor will increment IA32_TIMESTAMP_COUNTER based on the current core frequency but the Intel® Xeon Phi™ coprocessor will not scale such MSRs across package C-states.

For Intel® Xeon Phi™ coprocessor performance analysis, the IA32_TIMESTAMP_COUNTER feature always works on P1 and standard behavior is expected so that any new or pre-existing code using RDTSC will obtain consistent results. However, P-states and package C-states must be disabled during fine-grained performance analysis.

2.1.8 System Interface

The System Interface consists of two major units: the Intel® Xeon Phi™ coprocessor System Interface and the Transaction Control Unit (TCU). The SI contains all of the PCI Express* logic, which includes the PCI Express* protocol engine, SPI for flash and coprocessor OS loading, I²C for fan control, and the APIC logic. The TCU bridges the coprocessor SI to the Intel® Xeon Phi™ coprocessor internal ring, and contains the hardware support for DMA and buffering with transaction control flow. This block includes the DMA controllers, the encryption/decryption engine, MMIO registers, and various flow-control queuing instructions that allow internal interface to the ring transaction protocol engine.

2.1.8.1 PCI Express

The Intel® Xeon Phi™ coprocessor card complies with the Gen2x16 PCI Express* and supports 64 to 256 byte packet. PCI Express* peer-to-peer writes and reads are also supported.

The following registers show the Intel® Xeon Phi™ coprocessor PCI Express configuration setting:

- PCIE_PCIE_CAPABILITY Register (SBOX MMIO offset 0x584C)

Bits	Type	Reset	Description
23:20	RO	0x0	Device/Port Type
other bits unmodified			

- PCIE_BAR_ENABLE Register (SBOX MMIO offset 0x5CD4)

Bits	Type	Reset	Description
0	RW	1	MEMBAR0 (Aperture) Enable
1	RW	1	MEMBAR1 (MMIO Registers) Enable
2	RW	0	I/O BAR Enable
3	RW	0	EXPROM BAR Enable
31:4	Rsvd	0	

2.1.8.2 Memory Controller

There are 8 on-die GDDR5-based memory controllers in the Intel® Xeon Phi™ coprocessor. Each can operate two 32-bit channels for a total of 16 memory channels that are capable of delivering up to 5.5 GT/s per channel. The memory controllers directly interface to the ring interconnect at full speed, receiving complete physical addresses with each request. It is responsible for reading data from and writing data to GDDR memory, translating the memory read and write requests into GDDR commands. All the requests coming from the ring interface are scheduled by taking into account the timing restrictions of the GDDR memory and its physical organization to maximize the effective bandwidth that can be obtained from the GDDR memory. The memory controller guarantees a bounded latency for special requests arriving from the SBOX. The bandwidth guaranteed to the SBOX is 2 GB/s. The MBOX communicates to the FBOX (the ring interface) and the PBOX (the physical interface to the GDDR). The MBOX is also responsible for issuing all the refresh commands to the GDDR.

The GDDR5 interface supports two major data integrity features: Parity on the Command/Address interface and an optional software-based ECC for data.

2.1.8.2.1 DMA Capabilities

Direct Memory Access (DMA) is a common hardware function within a computer system that is used to relieve the CPU from the burden of copying large blocks of data. To move a block of data, the CPU constructs and fills a buffer, if one doesn't already exist, and then writes a descriptor into the DMA Channel's Descriptor Ring. A descriptor describes details such as the source and target memory addresses and the length of data in cache lines. The following data transfers are supported:

- Intel® Xeon Phi™ coprocessor to Intel® Xeon Phi™ coprocessor GDDR5 space (aperture)
- Intel® Xeon Phi™ coprocessor GDDR5 to host System Memory
- Host System Memory to Intel® Xeon Phi™ coprocessor GDDR5 (aperture or non-aperture)
- Intra-GDDR5 Block Transfers within Intel® Xeon Phi™ coprocessor

A DMA Descriptor Ring is programmed by either the coprocessor OS or the Host Driver. Up to eight Descriptor Rings can be opened by software; each being referred to as a DMA Channel. The coprocessor OS or Host Driver can open a DMA Channel in either system or GDDR5 memory respectively; that is, all descriptor rings owned by the host driver must exist in system memory while rings owned by the coprocessor OS must exist in GDDR5 memory. A programmable arbitration

scheme resolves access conflicts when multiple DMA Channels vie for system or Intel® Xeon Phi™ coprocessor resources.

The Intel® Xeon Phi™ coprocessor supports host-initiated or device-initiated PCI Express* Gen2/Gen1 memory, I/O, and configuration transactions. The Intel® Xeon Phi™ coprocessor device-initiated memory transactions can be generated either from execution cores directly or by using the DMA engine in the SBOX.

In summary, the DMA controller has the following capabilities:

- 8 DMA channels operating simultaneously, each with its own independent hardware ring buffer that can live in either local or system memory
- Supports transfers in either direction (host / Intel® Xeon Phi™ coprocessor devices)
- Supports transfers initiated by either side
- Always transfers using physical addresses
- Interrupt generation upon completion
- 64-byte granularity for alignment and size
- Writing completion tags to either local or system memory

The DMA block operates at the core clock frequency. There are 8 independent channels which can move data:

- From GDDR5 Memory to System Memory
- From System Memory to GDDR5 Memory
- From GDDR5 Memory to GDDR5 Memory

The Intel® Xeon Phi™ coprocessor not only supports 64-bytes (1 cache line) per PCI Express* transaction, but up to a maximum of 256 bytes for each DMA-initiated transaction. This requires that the Root-Complex support 256 byte transactions. Programming the MAX_PAYLOAD_SIZE in the PCI_COMMAND_STATUS register sets the actual size of each transaction.

2.1.8.2.1.1 DMA Channel Arbitration

There is no notion of priority between descriptors within a DMA Channel; descriptors are fetched, and operated on, in a sequential order. Priority between descriptors is resolved by opening multiple DMA channels and performing arbitration between DMA channels in a round-robin fashion.

2.1.8.2.1.2 Descriptor Ring Overview

A Descriptor Ring is a circular buffer as shown in Figure 2-7. The length of a Descriptor Ring can be up to 128K entries, and must align to the nearest cache line boundary. Software manages the ring by advancing a Head Pointer as it fills the ring with descriptors. When the descriptors have been copied, it writes this updated Header Pointer into the DMA Head Pointer Register (DHPR0 – DHPR7) for the appropriate DMA Channel. Each DMA Channel contains a Tail Pointer that advances as descriptors are fetched into a channel's Local Descriptor Queue. The Descriptor Queue is 64 entries, and can be thought of as a sliding window over the Descriptor Ring. The Tail Pointer is periodically written back to memory so that software can track its progress. Upon initialization, software sets both the Head Pointer and Tail Pointer to point to the base of the Descriptor Ring. From the DMA Channel perspective, an empty state is approached when the Tail Pointer approaches the Head Pointer. From a software perspective, a full condition is approached when the Head Pointer approaches the Tail Pointer.

The Head and Tail Pointers are 40-bit Intel® Xeon Phi™ coprocessor addresses. If the high-order bit is a 1, the descriptors reside in system memory; otherwise they reside in the Intel® Xeon Phi™ coprocessor memory. Descriptors come in five different formats and are 16 bytes in length. There are no alignment restrictions when writing descriptors into the ring. However, performance is optimized when descriptors start and end on cache line boundaries because memory accesses are performed on cache line granularities, four descriptors at a time.

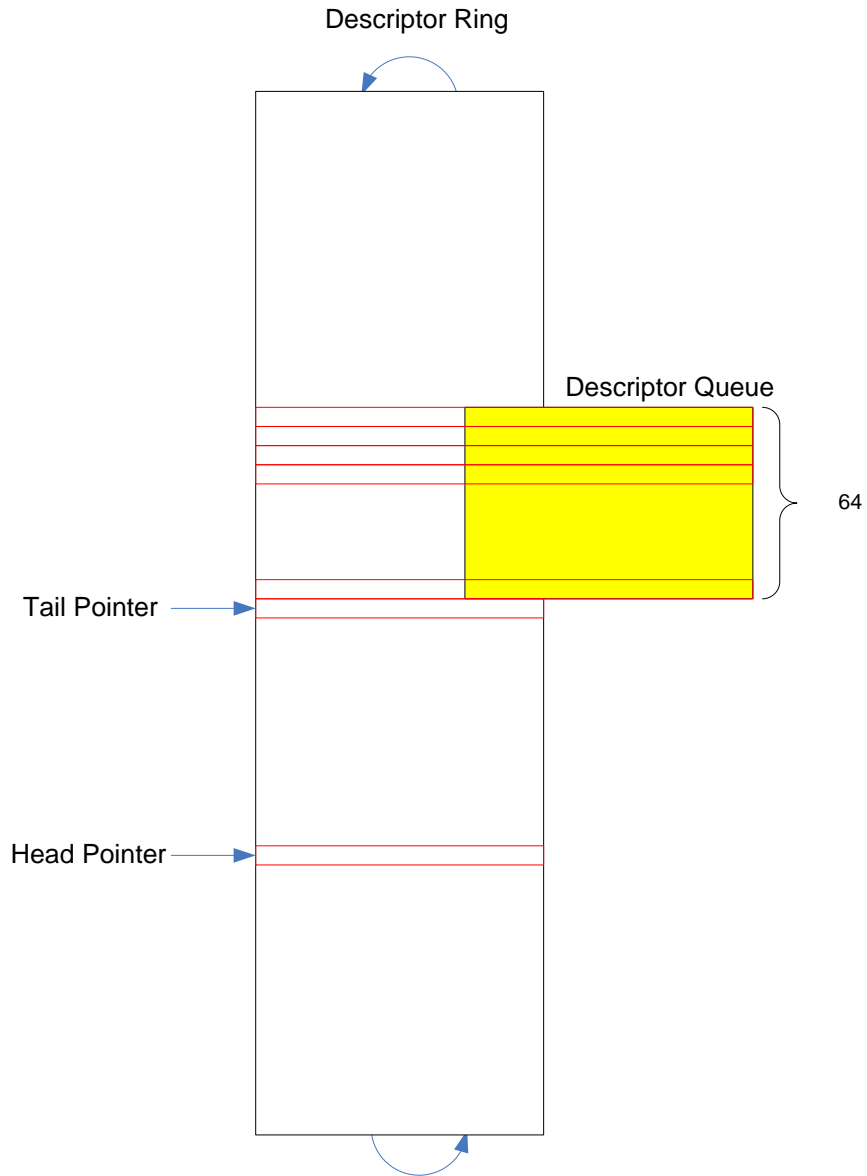


Figure 2-7. DMA Channel Descriptor Ring plus Local Descriptor Queue

2.1.8.2.1.3 Descriptor Ring Setup

Figure 2-8 shows how the Descriptor Ring Attributes Register or DRAR sets up the Descriptor Ring in each DMA channel. Because a descriptor ring can vary in size, the Base Address (BA) represents a 36-bit index. The Tail Pointer Index is concatenated to the BA field to form up a Tail Pointer to the GDDR space. If the descriptor ring resides in system memory, BA[35] and BA[34] will be truncated to correspond with the 16GB system-memory page as shown in Figure 2-9. The Sys bit must be set along with a valid system-memory page number.

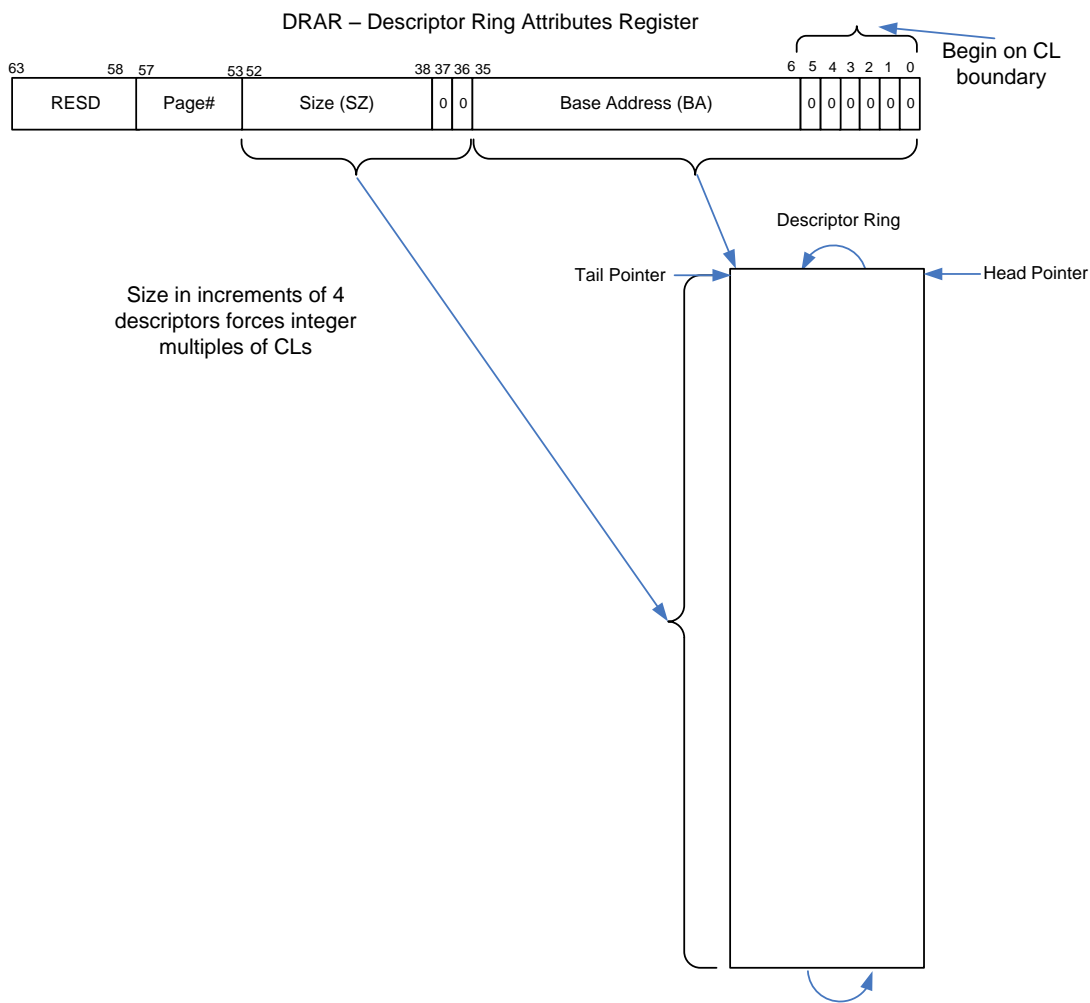


Figure 2-8. Descriptor Ring Attributes

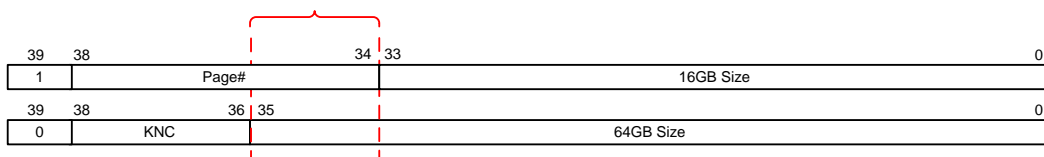


Figure 2-9. Intel® Xeon Phi™ Coprocessor Address Format

Because the size of the Descriptor Ring can vary, the Base Address must provide adequate space for concatenation of the Tail Pointer Index by zeroing out all the low-order bits that correspond to the size as shown in Figure 2-9. Table 2-9 gives some examples of the base address ranges based on the size of the descriptor ring.

Because the Head Pointer Index is updated by software, checks are made to determine if the index falls within the range specified by the size. An error will be generated if the range is exceeded.

Table 2-9. Examples of Base Address Ranges Based on Descriptor Ring Size

Size	Base Address Range	Tail Pointer Range
0x0004 (4)	0x0_0000_0000 : 0xF_FFFF_FFFC	0x0_0000 : 0x0003
0x0008 (8)	0x0_0000_0000 : 0xF_FFFF_FFF8	0x0_0000 : 0x0007
0x000C (12)	0x0_0000_0000 : 0xF_FFFF_FFF0	0x0_0000 : 0x000B
0x0010 (16)	0x0_0000_0000 : 0xF_FFFF_FFF0	0x0_0000 : 0x000F
0x0018 (24)	0x0_0000_0000 : 0xF_FFFF_FFE0	0x0_0000 : 0x0017
0x0100 (256)	0x0_0000_0000 : 0xF_FFFF_FF00	0x0_0000 : 0x00FF
0x0400 (1024)	0x0_0000_0000 : 0xF_FFFF_FC00	0x0_0000 : 0x03FF
0x1000 (4096)	0x0_0000_0000 : 0xF_FFFF_F000	0x0_0000 : 0x0FFF

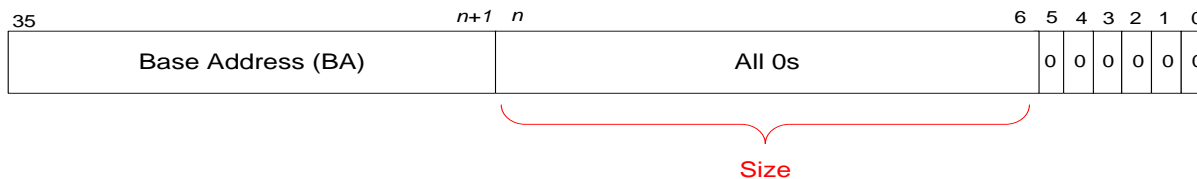


Figure 2-10. Base Address Width Variations

Figure 2-11 shows the Head and Tail Pointer index registers used to access the descriptor ring. Both pointers are indexes into the descriptor ring relative to the base, not to Intel® Xeon Phi™ coprocessor addresses. Both indexes are on descriptor boundaries and are the same width as the *Size* field in the DRAR. For the Tail Pointer Address, the DMA uses the *TPI* along with the *Sys* bit, *Page*, and *Base Address* in the DRAR.

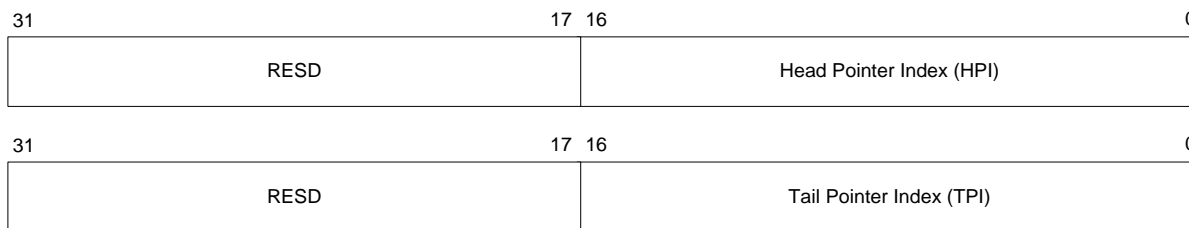


Figure 2-11 Head and Tail Pointer Index Registers

2.1.8.2.2 Interrupt Handling

There are three different types of interrupt flows that are supported in the Intel® Xeon Phi™ coprocessor:

- Local Interrupts** – These are the interrupts that are destined for one (or more) of the Intel® Xeon Phi™ coprocessor cores located on the originating device. They appear in the form of APIC messages on the APIC serial bus.
- Remote Interrupts** – These are the interrupts which are destined for one (or more) of the Intel® Xeon Phi™ coprocessor cores in other Intel® Xeon Phi™ coprocessor devices. They appear as MMIO accesses on the PEG port.
- System Interrupts** – These are the interrupts which are destined for the host processor(s). They appear as INTx/MSI/MSI-X messages on the PEG port, depending upon the PCI configuration settings.

2.1.8.2.3 Intel® Xeon Phi™ Coprocessor Memory Space

Table 2-10 lists the starting addresses assigned to specific functions.

Table 2-10. Coprocessor Memory Map

Function	Starting Address	Size (Bytes)	Comment
GDDR5 Memory	00_0000_0000	Variable	
System Memory		Variable	Addresses translated through SMPT
Flash Memory	00_FFF8_5000	364K	Actual size of flash varies, Some parts are not accessible through the normal memory path
MMIO Registers	00_007D_0000	64K	Accessibility from the host is limited
Boot ROM	00_FFFF_0000	64K	New for Intel® Xeon Phi™ coprocessor. Overlays FBOOT0 image in flash
Fuse Block	00_FFF8_4000	4K	New for Intel® Xeon Phi™ coprocessor memory space

2.1.8.2.3.1 Host-Visible Intel® Xeon Phi™ Coprocessor Memory Space

After Reset, all GDDR5 memory sits inside “stolen memory” (that is, memory not accessible by the Host). Stolen memory (CP_MEM_BASE/TOP) has precedence over the PCI Express* aperture. FBOOT1 code will typically shrink stolen memory or remove it. The aperture is programmed by the host or the coprocessor OS to create a flat memory space.

2.1.8.2.3.2 Intel® Xeon Phi™ Coprocessor Boot ROM

The Intel® Xeon Phi™ coprocessor software boot process is summarized below:

1. After Reset: Boot-Strap Processor (BSP) executes code directly from the 1st-Stage Boot-Loader Image (FBOOT0).
2. FBOOT0 authenticates 2nd-Stage Boot-Loader (FBOOT1) and jumps to FBOOT1.
3. FBOOT1 sets up/trains GDDR5 and basic memory map.
4. FBOOT1 tells host to upload coprocessor OS image to GDDR5.
5. FBOOT1 authenticates coprocessor OS image. If authentication fails, FBOOT1 locks out specific features.
6. FBOOT1 jumps to coprocessor OS.

2.1.8.2.3.3 SBOX MMIO Register Space

The SBOX contains 666 MMIO (Memory-Mapped I/O) registers (12 K bytes) that are used for configuration, status and debug of the SBOX and other parts of the rest of Intel® Xeon Phi™ coprocessor. These are sometimes referred to as CSR's and are not part of the PCI Express* configuration space. The SBOX MMIO space is located at 08_007D_0000h-08_007D_FFFFh in the Intel® Xeon Phi™ coprocessor memory space. These MMIO registers are not contiguous, but are split between various functional blocks within the SBOX. Accessibility is always allowed to the coprocessor OS while accessibility by the host is limited to a subset for security.

2.1.9 VPU and Vector Architecture

The Intel® Xeon Phi™ coprocessor has a new SIMD 512-bit wide VPU with a corresponding vector instruction set. The VPU can be used to process 16 single precision or 8 double precision elements. There are 32 vector registers (8 mask registers with per lane predicated execution). Prime (hint) instructions for scatter/gather are available. Load operation comes from 2-3 sources to 1 destination. There are new SP transcendental instructions supported in hardware for exponent, logarithm, reciprocal, and square root operations. The VPUs are mostly IEEE 754 2008 floating-point compliant with added SP, DP-denorm, and SAE support for IEEE compliance and improved performance on fdiv/sqrt. Streaming stores (no read for ownership before write) are available with the vmovaps/pd.nr and vmovaps/pd.ngo instructions.

Section 7 contains more detailed information on the vector architecture.

2.1.10 Intel® Xeon Phi™ Coprocessor Instructions

The Intel® Xeon Phi™ coprocessor instruction set includes new vector instructions that are an extension of the existing Intel® 64 ISA. However, they do not support the Intel Architecture family of vector architecture models (MMX™ instructions, Intel® Streaming SIMD Extensions, or Intel® Advanced Vector Extensions).

The major features of the Intel® Xeon Phi™ coprocessor vector ISA extensions are:

- A new instruction repertoire specifically tailored to boost the performance of High Performance Computing (HPC) applications. The instructions provide native support for both float32 and int32 operations while providing a rich set of conversions for common high performance computing native data types. Additionally, the Intel® Xeon Phi™ coprocessor ISA supports float64 arithmetic and int64 logic operations.
- There are 32 new vector registers. Each is 512 bits wide, capable of packing 16 32-bit elements or 8 64-bit elements of floating point or integer values. A large and uniform vector register file helps in generating high performance code and covering longer latencies.
- Ternary instructions with two sources and different destinations. There are also Fused Multiply and Add (FMA) instructions which are ternary with three sources, one of which is also the destination.
- Intel® Xeon Phi™ coprocessor instructions introduce 8 vector mask registers that allow conditional execution over the 16 elements in a vector instruction and merged results to the original destination. Masks allow vectorizing loops that contain conditional statements. Additionally, support is provided for updating the value of the vector masks with special vector instructions such as vcmpps.
- The vector architecture supports a coherent memory model wherein the new set of instructions operates in the same memory address space as the standard Intel® 64 instructions. This feature eases the process of developing vector code.
- Specific gather/scatter instructions manipulate irregular data patterns in memory (by fetching sparse locations of memory into a dense vector register or vice-versa) thus enabling vectorization of algorithms with complex data structures.

Consult the (Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual (Reference Number: 327364)) for complete details on the Intel® Xeon Phi™ coprocessor instructions.

2.1.11 Multi-Card

Each Intel® Xeon Phi™ coprocessor device is treated as an independent computing environment. The host OS enumerates all the cards in the system at boot time and launches separate instances of the coprocessor OS and the SCIF driver. See the SCIF documentation for more details about intercard communication.

2.1.12 Host and Intel® MIC Architecture Physical Memory Map

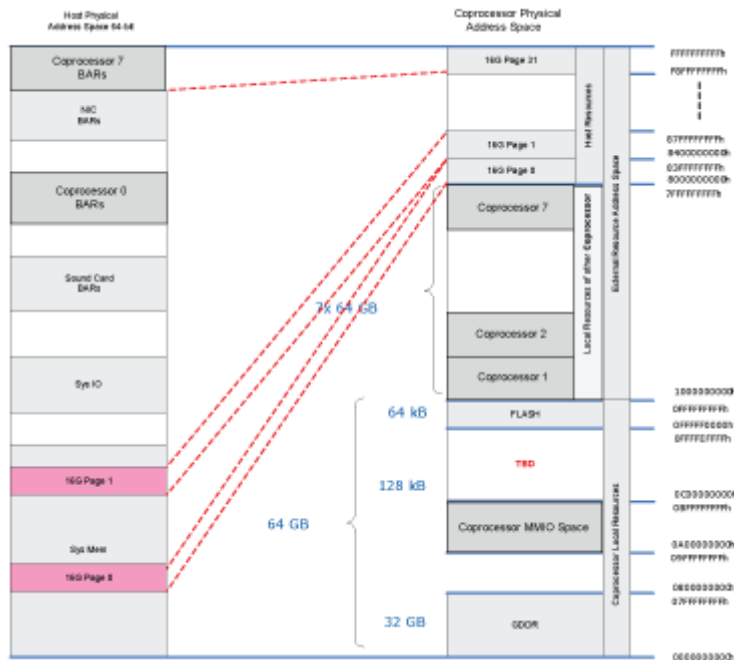


Figure 2-12. Host and Intel® MIC Architecture Physical Memory Map

The Intel® Xeon Phi™ coprocessor memory space supports 40-bit physical addresses, which translates into 1024 GiB of addressable memory space that is split into 3 high-level ranges:

- Local address range: 0x00_0000_0000 to 0x0F_FFFF_FFFF (64 GiB)
- Reserved: 0x10_0000_0000 to 0x7F_FFFF_FFFF (448 GiB)
- System (Host) address range 0x80_0000_0000 to 0xFF_FFFF_FFFF (512 GiB)

The Local Address Range 0x00_0000_0000 to 0x0F_FFFF_FFFF (64 GiB) is further divided into 4 equal size ranges:

- 0x00_0000_0000 to 0x03_FFFF_FFFF (16 GiB)
 - GDDR (Low) Memory
 - Local APIC Range (relocatable) 0x00_FEE0_0000 to 0x00_FEE0_OFFF (4 kB)
 - Boot Code (Flash) and Fuse (via SBOX) 0x00_FF00_0000 to 0x00_FFFF_FFFF (16 MB)
 - 0x04_0000_0000 to 0x07_FFFF_FFFF (16 GB)
- GDDR Memory (up to PHY_GDDR_TOP)
 - 0x08_0000_0000 to 0x0B_FFFF_FFFF (16 GB)
- Memory mapped registers
- DBOX registers 0x08_007C_0000 to 0x08_007C_FFFF (64 kB)
- SBOX registers 0x08_007D_0000 to 0x08_007D_FFFF (64 kB)
- Reserved 0x0C_0000_0000 to 0x0F_FFFF_FFFF (16 GB)

The System address range 0x80_0000_0000 to 0xFF_FFFF_FFFF (512 GB) contains 32 pages of 16 GB each:

- Sys 0: 0x80_0000_0000 to 0x83_FFFF_FFFF (16 GB)
- Sys 1: 0x84_0000_0000 to 0x87_FFFF_FFFF (16 GB)
- ...
- Sys 31: 0xFC_0000_0000 to 0xFF_FFFF_FFFF (16 GB)

These are used to access System Physical Memory addresses and can access up to 512 GiB at any given time. Remote Intel® Xeon Phi™ coprocessor devices are also accessed through System addresses. All requests over PCI Express* to

Host are generated through this range. A System Memory Page Table (SMPT) expands the 40-bit local address to 64-bit System address.

Accesses to host memory are snooped by the host if the No-snoop bit in the SMPT register is not set. The SCIF driver (see Sections 2.2.5.1 and 5.1) does not set this bit so host accesses are always snooped. Host accesses to Intel® Xeon Phi™ coprocessor memory are snooped if to cacheable memory.

The System (Host) address map of Intel® Xeon Phi™ coprocessor memory is represented by two base address registers:

- MEMBAR0
 - Relocatable in 64-bit System Physical Memory Address space
 - Prefetchable
 - 32 GiB (max) down to 256 MiB (min)
 - Programmable in Flash
 - Offset into Intel® Xeon Phi™ coprocessor Physical Memory Address space
 - Programmable in APR_PHY_BASE register
 - Default is 0
- MEMBAR1
 - Relocatable in 64b System Physical Memory Address space
 - Non-prefetchable
 - 128 KiB
 - Covers DBOX0 & SBOX Memory-mapped registers
 - DBOX at offset 0x0_0000
 - SBOX at offset 0x1_0000

2.1.13 Power Management

Intel® Xeon Phi™ coprocessor power management supports Turbo Mode and other P-states. Turbo mode is an opportunistic capability that allows the CPU to take advantage of thermal and power delivery headroom to increase the operating frequency and voltage, depending on the number of active cores. Unlike the multicore family of Intel® Xeon® processors, there is no hardware-level power control unit (PCU); power management is controlled by the coprocessor OS. Please see [Section 3.1](#) for more information on the power management scheme.

Below is a short description of the different operating modes and power states. For additional details, see the “Intel® Xeon Phi™ Coprocessor Datasheet,” Document Number 488073.

- Core C1 State – Core and VPU are clock gated (all 4 threads have halted)
- Core C6 State– Core and VPU are power gated (C1 + time threshold)
- Package C3 State
- All Cores Clock or Power Gated
- The Ring and Uncore are Clock Gated (MCLK gated (auto), VccP reduced (Deep))
 - Package C6 State – The VccP is Off (Cores/Ring/Uncore Off)
 - Memory States
- M1 – Clock Gating
- M2 – GDDR in Self Refresh
- M3 – M2 + Shut off
 - GMCIk PLL
 - SBOX States– L1 (PCI Express* Link States), SBOX Clock Gating

2.2 Intel® Xeon Phi™ Coprocessor Software Architecture

The software architecture for Intel® Xeon Phi™ coprocessor products accelerates highly parallel applications that take advantage of hundreds of independent hardware threads and large local memory. Intel® Xeon Phi™ coprocessor product software enables easy integration into system platforms that support the PCI Express* interconnect and running either a Linux* or Windows* operating system.

2.2.1 Architectural Overview

Intel® Xeon Phi™ coprocessor products are implemented as a tightly integrated, large collection of processor cores (Intel® Many Integrated Core (MIC) Architecture) on a PCI Express* form-factor add-in card. As such, Intel® Xeon Phi™ coprocessor products comply as a PCI Express* endpoint, as described in the PCI Express* specification. Therefore, each Intel® Xeon Phi™ coprocessor card implements the three required address spaces (configuration, memory, and I/O) and responds to requests from the host to enumerate and configure the card. The host OS loads a device driver that must conform to the OS driver architecture and behavior customary for the host operating system running on the platform (e.g., interrupt handling, thread safe, security, ACPI power states, etc.).

From the software prospective, each Intel® Xeon Phi™ coprocessor add-in card represents a separate Symmetric Multi-Processing (SMP) computing domain that is loosely-coupled to the computing domain represented by the OS running on the host platform. Because the Intel® Xeon Phi™ coprocessor cards appear as local resources attached to PCI Express*, it is possible to support several different programming models using the same hardware implementation. For example, a programming model requiring shared memory can be implemented using SCIF messaging for communication. Highly parallel applications utilize a range of programming models, so it is advantageous to offer flexibility in choosing a programming model.

In order to support a wide range of tools and applications for High-Performance Computing (HPC), several Application Programming Interfaces (APIs) are provided. The standard APIs provided are sockets over TCP/IP*, MPI, and OpenCL*. Some Intel proprietary interfaces are also provided to create a suitable abstraction layer for internal tools and applications. The SCIF APIs provide a common transport over which the other APIs communicate between host and Intel® Xeon Phi™ coprocessor devices across the platform's PCI Express hardware. **Error! Reference source not found.** illustrates the relative relationship of each of these APIs in the overall Intel® MIC Architecture Manycore Platform Software Stack (MPSS).

As shown, the executable files and runtimes of a set of software development tools targeted at highly parallel programming are layered on top of and utilize various subsets of the proprietary APIs.

2.2.2 Intel® Manycore Platform Software Stack (MPSS)

Intel® Xeon Phi™ Board

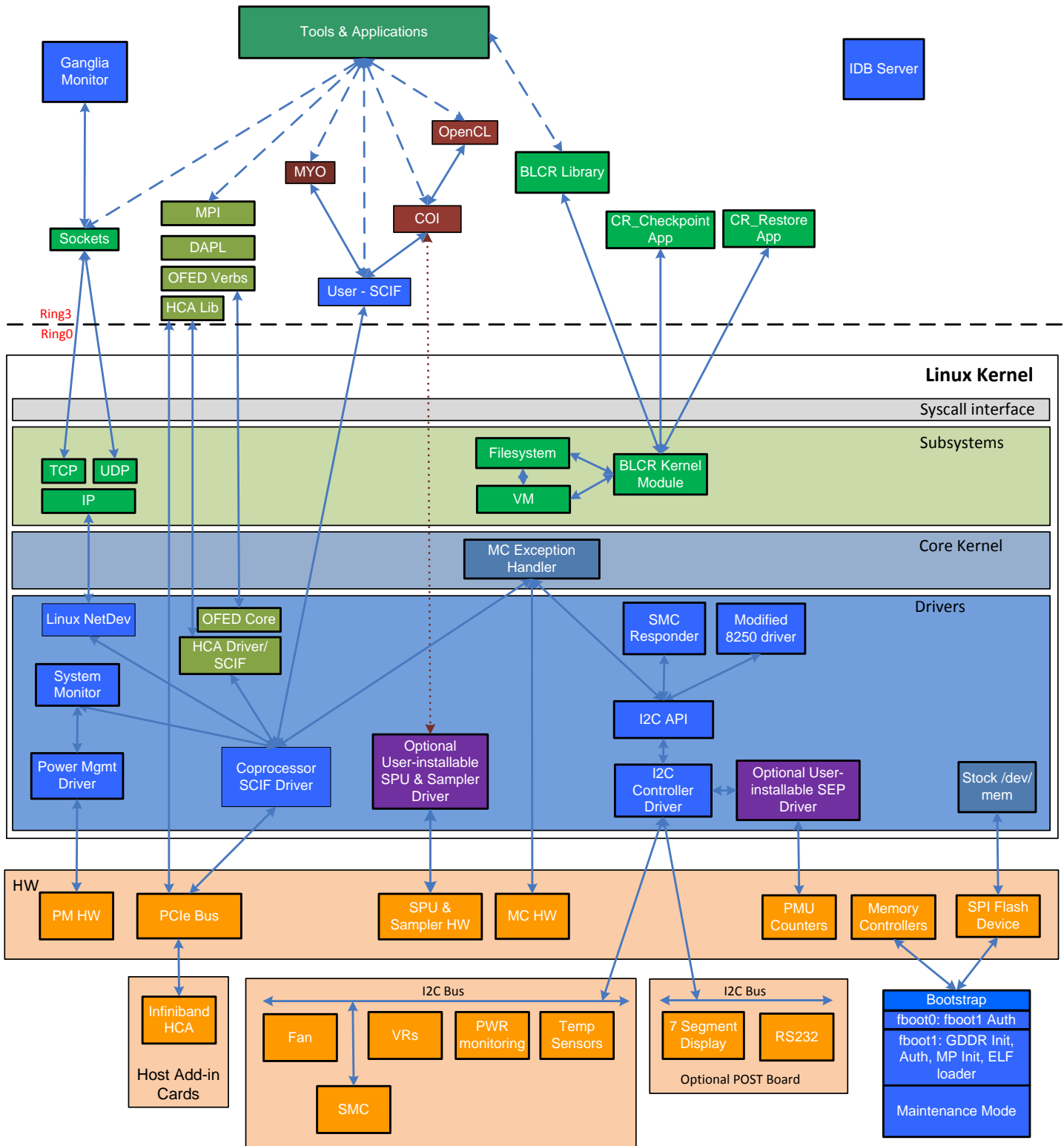


Figure 2-14 outlines the high level pieces that comprise the Intel® Manycore Platform Software Stack, or MPSS.

Intel® Xeon Phi™ Board

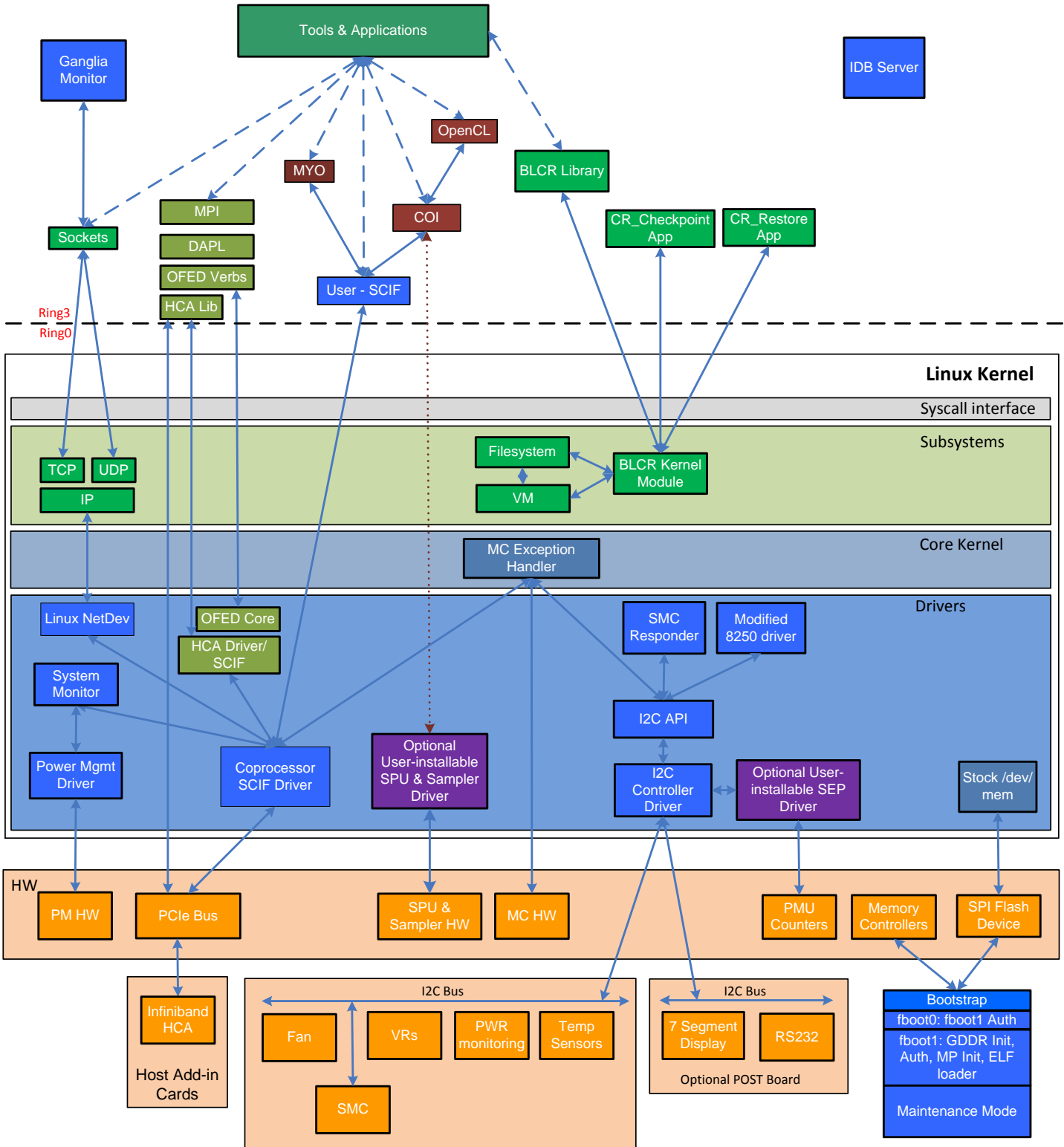


Figure 2-14. Intel® Xeon Phi™ Coprocessor Software Stack

2.2.3 Bootstrap

Since the Intel® Xeon Phi™ coprocessor cores are x86 Intel Architecture cores, the bootstrap resembles a System BIOS at POST. The bootstrap runs when the board first gets power, but can also run when reset by the host due to a catastrophic failure. The bootstrap is responsible for card initialization and booting the coprocessor OS.

The bootstrap consists of two separate blocks of code, called fboot0 and fboot1. The fboot0 block resides in ROM memory on the die and cannot be upgraded, while fboot1 is upgradeable in the field and resides in the flash memory.

2.2.3.1 fboot0

When the card comes out of reset, the fboot0 instruction is executed first. This block of code is the root of trust because it cannot be modified in the field. Its purpose is to authenticate the second stage, fboot1, by passing the root of trust to fboot1. If authentication fails, fboot0 will remove power from the ring and cores, preventing any further action. The only recovery mechanism from this state is to put the card into “zombie mode” by manually changing a jumper on the card. Zombie mode allows the host to reprogram the flash chip, recovering from a bad fboot1 block.

The fboot0 execution flow is as follows:

1. Setup CAR mode to reduce execution time.
2. Transition to 64-bit protected mode.
3. Authenticate fboot1.
4. If authentication fails, shut down card.
5. If authentication passes, hand control to fboot1.

2.2.3.2 fboot1

Fboot1 is responsible for configuring the card and booting the coprocessor OS. The card configuration involves initializing all of the cores, uncore units, and memory. This includes implementing any silicon workarounds since the hardware does not support microcode patching like a typical x86 core. The cores must be booted into 64-bit protected mode to be able to access the necessary configuration registers.

When booting a 3rd party coprocessor OS, including the MPSS Linux*-based coprocessor OS, the root of trust is not passed any further. The root of trust is only passed when booting into maintenance mode since privileged operations are performed while in maintenance mode. Maintenance mode is where some locked registers are re-written for hardware failure recovery.

Authentication determines which coprocessor OS type is booting (3rd party or maintenance). Fboot1 calls back into fboot0 to run the authentication routine using the public key also embedded in fboot0. Only the maintenance coprocessor OS is signed with a private key, and all other images must remain unsigned. If authentication passes, the maintenance coprocessor OS boots. If authentication fails, the process is assumed to be a 3rd party coprocessor OS and the Linux* boot protocol is followed, locking out access to sensitive registers, protects intellectual property.

The fboot1 execution flow is as follows:

1. Set memory frequency then reset the card.
2. Perform core initialization.
3. Initialize GDDR5 memory
 - a. Use training parameters stored in the flash if memory has been trained earlier.
 - b. If no training parameters are stored, or these parameters do not match the current configuration, perform the normal training routine and store training values in the flash.
4. Shadow fboot1 into GDDR5 to improve execution time.
5. Perform uncore initialization.
6. Perform CC6 register initialization.

7. Boot APs.
8. AP's transition to 64-bit protected mode.
9. AP's perform core initialization.
10. AP's perform CC6 register initialization.
11. AP's reach the end of the AP flow and wait for further instructions.
12. Wait for coprocessor OS download from host.
13. Authenticate coprocessor OS. All cores participate in authentication to minimize execution time.
14. If authentication passes, it is a maintenance coprocessor OS. Boot maintenance coprocessor OS.
15. If authentication fails, it is a 3rd party coprocessor OS (see Linux* loader section below).
 - a. Lock out register access.
 - b. Create boot parameter structure.
 - c. Transition to 32-bit protected mode with paging disabled.
 - d. Hand control to the coprocessor OS.

2.2.4 Linux* Loader

The Intel® Xeon Phi™ coprocessor boots Linux*-based coprocessor OS images. It is capable of booting any 3rd party OS developed for the Intel® Xeon Phi™ coprocessor. Previously, an untrusted coprocessor OS would result in a card shutdown; however, the Intel® Xeon Phi™ coprocessor considers the Intel developed Linux*-based coprocessor OS to be untrusted. For this reason, it becomes simple to support 3rd party coprocessor OS images.

To boot a Linux* OS, the bootstrap has to conform to a certain configuration as documented in the Linux* kernel. There are 3 potential entry points into the kernel: 16-bit, 32-bit, and 64-bit entry points. Each entry point requires increasingly more data structures to be configured. The Intel® Xeon Phi™ coprocessor uses the 32-bit mode entry point.

2.2.4.1 16-bit Entry Point

The 16-bit entry point does not require any data structures to be created prior to entering the kernel; however it requires that there be support for system BIOS callbacks. The Intel® Xeon Phi™ coprocessor does not support this mode.

2.2.4.2 32-bit Entry Point

The 32-bit entry point requires a boot parameter (or zero page) structure and a structure defining the number of cores and other hardware (either an MP Table or SFI – Simple Firmware Interface - table). The Linux* documentation in boot.txt states “the CPU must be in 32-bit protected mode with paging disabled; a GDT must be loaded with the descriptors for selectors __BOOT_CS(0x10) and __BOOT_DS(0x18); both descriptors must be 4G flat segment; __BOOT_CS must have execute/read permission, and __BOOT_DS must have read/write permission; CS must be __BOOT_CS and DS, ES, SS must be __BOOT_DS; interrupt must be disabled; %esi must hold the base address of the struct boot_params; %ebp, %edi and %ebx must be zero.”

There exists a field in the boot parameter structure (load flags) that tells the kernel whether it should use the segments setup by the bootstrap or to load new ones. If the kernel loads new ones, it uses the above settings. The bootstrap, however, does not have the segment descriptors in the same order as required by this documentation; and therefore sets the boot parameter flag to tell the kernel to continue using the segments already setup by the bootstrap. Everything about the bootstrap descriptors matches the documentation except for the offset location in the GDT, so it is safe to continue using them.

The bootstrap also uses the SFI tables to report the number of cores, memory map, and other hardware configurations. This is a relatively new format designed by Intel and adheres to SFI version 0.7 (<http://simplefirmware.org>). SFI support was initially added to the Linux* kernel in version 2.6.32. The Intel® Xeon Phi™ coprocessor supports booting a Linux* kernel by using the 32-bit entry point.

2.2.4.3 64-bit Entry Point

The Intel® Xeon Phi™ coprocessor does not support this mode.

2.2.5 The Coprocessor Operating System (coprocessor OS)

The Intel® Xeon Phi™ coprocessor establishes the basic execution foundation that the remaining elements of the Intel® Xeon Phi™ coprocessor card's software stack rest upon. The Intel® Xeon Phi™ coprocessor OS is based on a standard Linux* kernel source code (from kernel.org) with as few changes to the standard kernel as possible. While some areas of the kernel are designed, by the Linux* development community, to be tailored for specific architectures, this is not the general case. Therefore, additional modifications to the kernel have been made to compensate for hardware normally found on PC platforms, but missing from Intel® Xeon Phi™ coprocessor cards.

The coprocessor OS provides typical capabilities such as process/task creation, scheduling, and memory management. It also provides configuration, power, and server management. Intel® Xeon Phi™ coprocessor-specific hardware is only accessible through a device driver written for the coprocessor OS environment.

The Intel® Xeon Phi™ coprocessor Linux* kernel can be extended with loadable kernel modules (LKMs); LKMs may be added or removed with modprobe. These modules may include both Intel supplied modules, such as the idb server and SEP sampling collector, and end-user supplied modules.

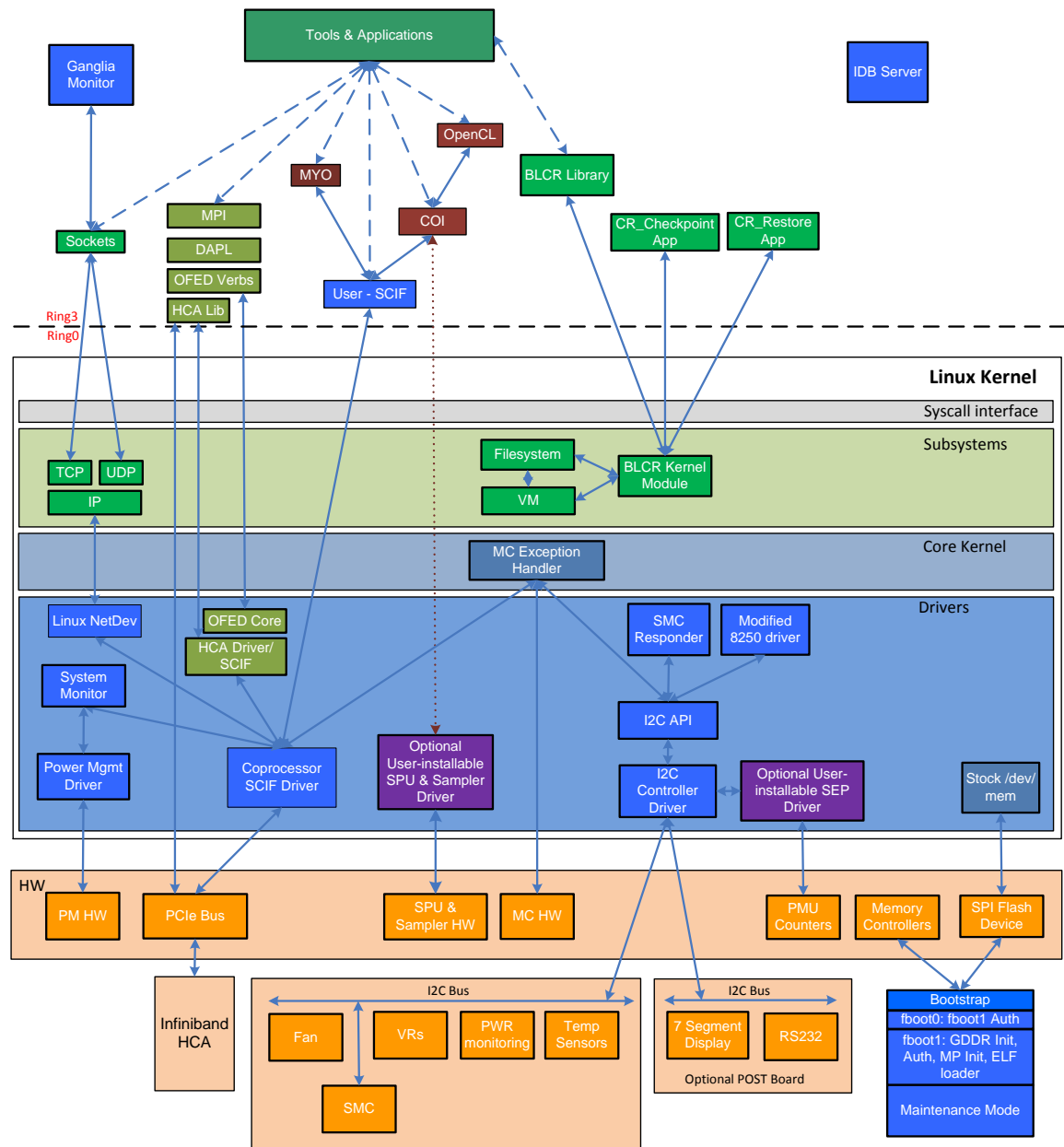


Figure 2-15. The Linux* Coprocessor OS Block Diagram

The Intel® Xeon Phi™ coprocessor Linux*-based coprocessor OS is a minimal, embedded Linux* environment ported to the Intel® MIC Architecture with the Linux* Standard Base (LSB) Core libraries. It is also an unsigned OS. It implements the Busybox* minimal shell environment. Table 2-11 lists the LSB components.

Table 2-11. LSB Core Libraries

Component	Description
glibc	the GNU C standard library
libc	the C standard library
libm	the math library
libdl	programmatic interface to the dynamic linking loader
librt	POSIX real-time library (POSIX shared memory, clock and time functions, timers)
libcrypt	password and data encryption library
libutil	library of utility functions

Component	Description
libstdc++	the GNU C++ standard library
libgcc_s	a low-level runtime library
libz	a lossless data compression library
libcurses	a terminal-independent method of updating character screens
libpam	the Pluggable Authentication Module (PAM) interfaces allow applications to request authentication via a system administrator defined mechanism

2.2.5.1 CPUID Enumeration

CPUID enumeration can be obtained via the Linux* OS APIs that report information about the topology as listed in `/sys/devices/system/cpu/cpu*/topology/*`.

2.2.6 Symmetric Communication Interface (SCIF)

SCIF is the communication backbone between the host processors and the Intel® Xeon Phi™ coprocessors in a heterogeneous computing environment. It provides communication capabilities within a single platform. SCIF enables communications between host and Intel® Xeon Phi™ coprocessor cards, and between Intel® Xeon Phi™ coprocessor cards within the platform. It provides a uniform API for communicating across the platform's PCI Express* system busses while delivering the full capabilities of the PCI Express* transport hardware. SCIF directly exposes the DMA capabilities of Intel® Xeon Phi™ coprocessor for high bandwidth transfer of large data segments, as well as the ability to map memory of the host or an Intel® Xeon Phi™ coprocessor device into the address space of a process running on the host or on any Intel® Xeon Phi™ coprocessor device.

Communication between SCIF node pairs is based on direct peer-to-peer access of the physical memory of the peer node. In particular, SCIF communication is not reflected through system memory when both nodes are Intel® Xeon Phi™ coprocessor cards.

SCIF's messaging layers take advantage of the PCI Express*'s inherent reliability, and operates as a simple data-only network without the need for any intermediate packet inspection. Messages are not numbered, nor is error checking performed. Due to the data-only nature of the interface, it is not a direct replacement for higher level communication APIs, but rather provides a level of abstraction from the system hardware for these other APIs. Each API that wishes to take advantage of SCIF will need to adapt to this new proprietary interface directly or through the use of a shim layer.

A more detailed description of the SCIF API can be found in Section [5.3](#).

2.2.7 Host Driver

The host driver is a collection of host-side drivers and servers including SCIF, power management, and RAS and server management. The primary job of the host driver is to initialize the Intel® Xeon Phi™ coprocessor card(s); this includes loading the coprocessor OS and its required boot parameters for each of the cards. Following successful booting, the primary responsibility of the host driver is to serve as the root of the SCIF network. Additional responsibilities revolve around serving as the host-side interface for power management, device management, and configuration. However, the host driver does not directly support any type of user interface or remote process API. These are implemented by other user-level programs or by communication protocols built on top of the driver or SCIF (e.g. Sockets, MPI, etc.).

DMA support is an asynchronous operation. Host initiated DMA is expected to have less latency compared to the proxy DMA from the card. Applications have the option to pick between memory copy and DMA, or to let the driver choose the best method. Memory copy is optimized to be multiple threaded, which makes use of the multi-core to parallelize the operation at the limit of the PCI Express* bandwidth. When there is a need to lower the host CPU load, or when the transfer size is above threshold, DMA is the preferred method.

Interrupts based on MSI/x (Message Signaled Interrupts) are supported by the host driver with these benefits:

- Eliminates dedicated hardware interrupt line connection
- No interrupt sharing with other device(s)
- With optimized hardware design, no need for the interrupt routine to read back from hardware which will improve the efficiency of the interrupt handling
- The device can target different CPU cores when triggering, thus making full use of the multicore for interrupt handling.

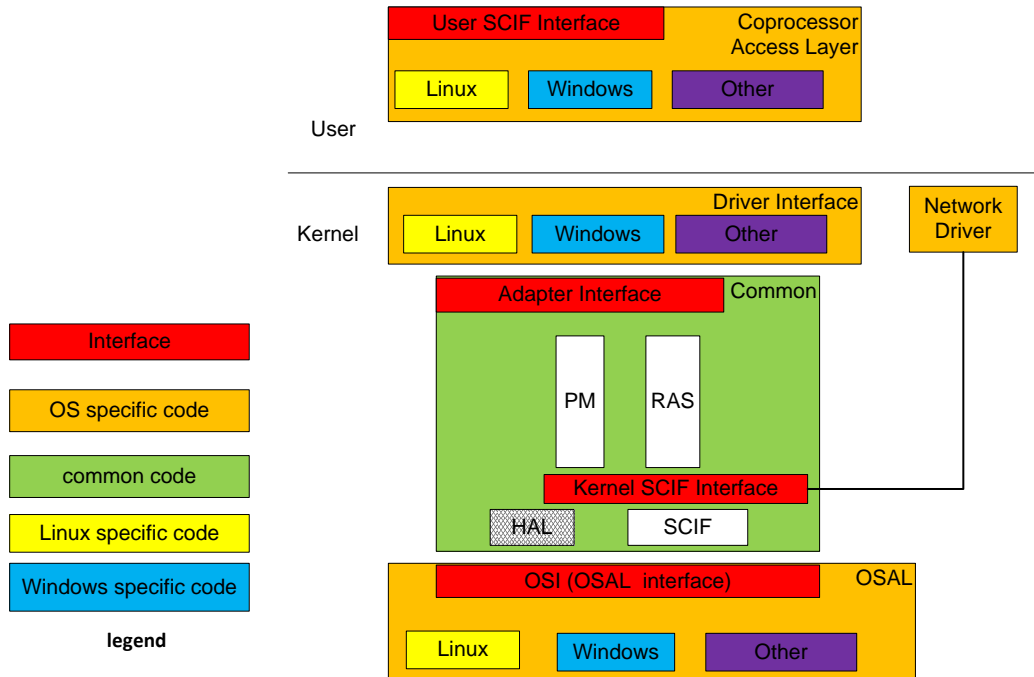


Figure 2-16. Intel® Xeon Phi™ Coprocessor Host Driver Software Architecture Components

2.2.7.1 Intel® Xeon Phi™ Coprocessor SMC Control Panel

The SMC Control Panel (micsmc), located in /opt/intel/mic/bin after installing Intel® MPSS, is the local host-side user interface for system management. The Control Panel is more practical for smaller setups like a workstation environment rather than for a large-scale cluster deployment. The Control Panel is mainly responsible for:

- Monitoring Intel® Xeon Phi™ coprocessor card status, parameters, power, thermal, etc.
- Monitoring system performance, core usage, memory usage, process information
- Monitoring overall system health, critical errors, or events
- Hardware configuration and setting, ECC, turbo mode, power plan setting, etc.

Control Panel applications rely on the MicAccessSDK to access card parameters. The MicAccessSDK exposes a set of APIs enabling applications to access the Intel® MIC Architecture hardware. The Ring 3 system management agent running on the card handles the queries from the host and returns results to the host through the SCIF interface.

The host RAS agent captures the MCA error report from the card and takes proper action for different error categories. The host RAS agent determines the error exposed to the end-user based on the error filter and Maintenance mode test/repair result. Then the error/failure is shown to end users on the Control Panel.

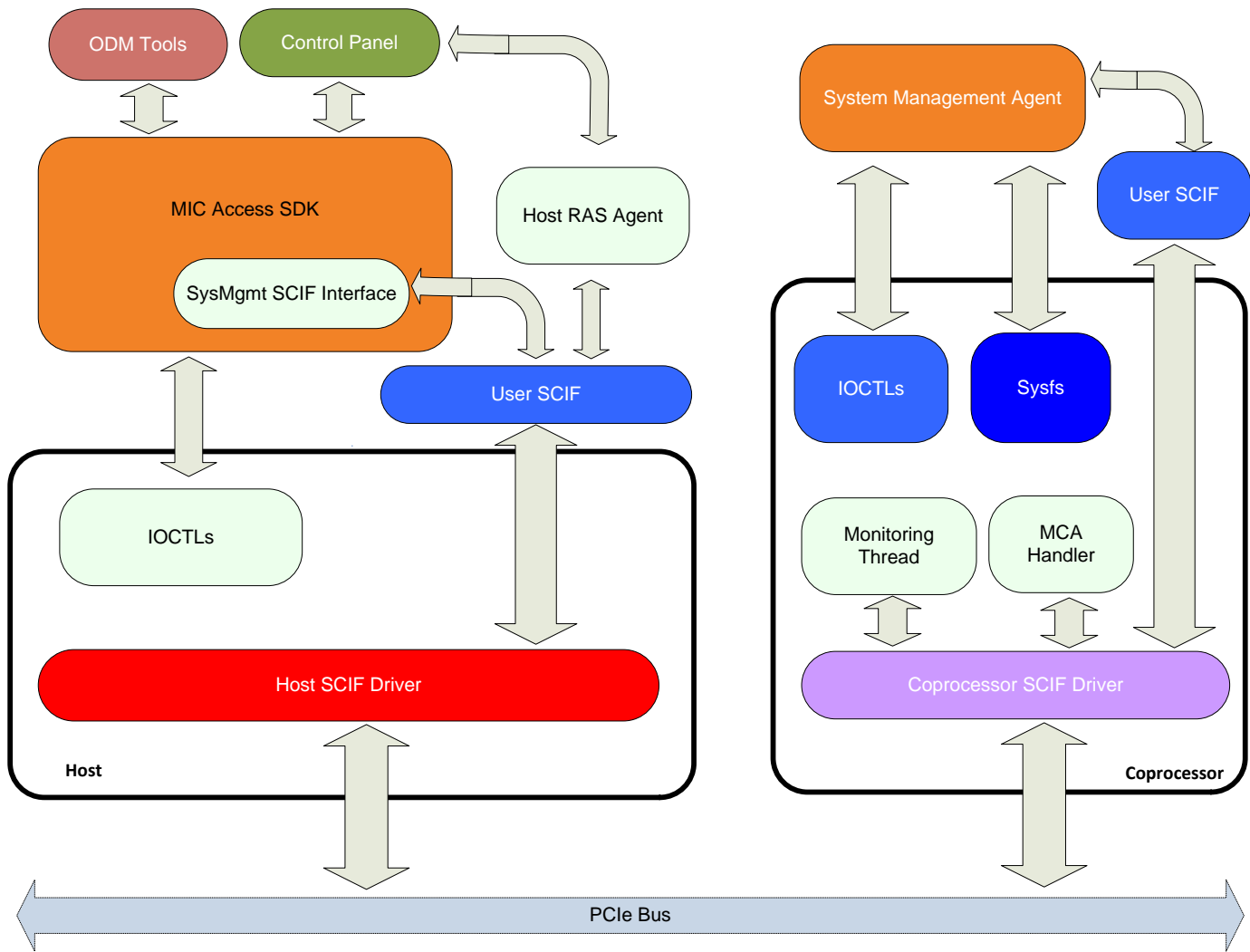


Figure 2-17. Control Panel Software Architecture

2.2.7.2 Ganglia* Support

Ganglia* is a scalable, distributed monitoring system for high-performance computing systems such as clusters and grids. The implementation of Ganglia* is robust, has been ported to an extensive set of operating systems and processor architectures, and is currently in use on thousands of clusters around the world.

Briefly, the Ganglia* system has a daemon running on each computing node or machine. The data from these daemons is collected by another daemon and placed in an rrdtool database. Ganglia* then uses PHP scripts on a web server to generate graphs as directed by the user. The typical Ganglia* data flow is illustrated in Figure 2-18.

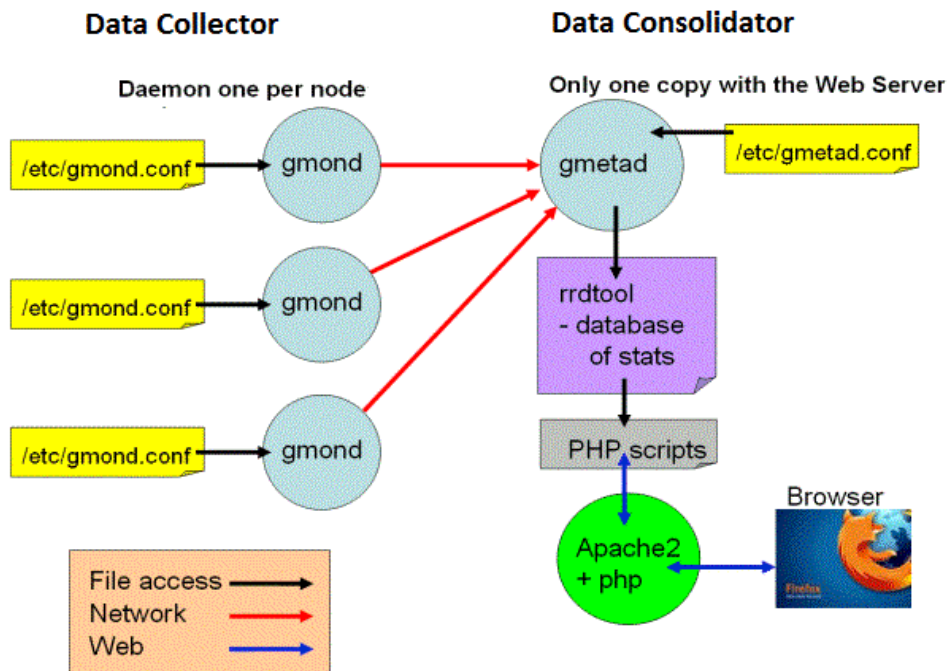


Figure 2-18. Ganglia* Monitoring System Data Flow Diagram

The cluster level deployment of Ganglia* is illustrated in Figure 2-19.

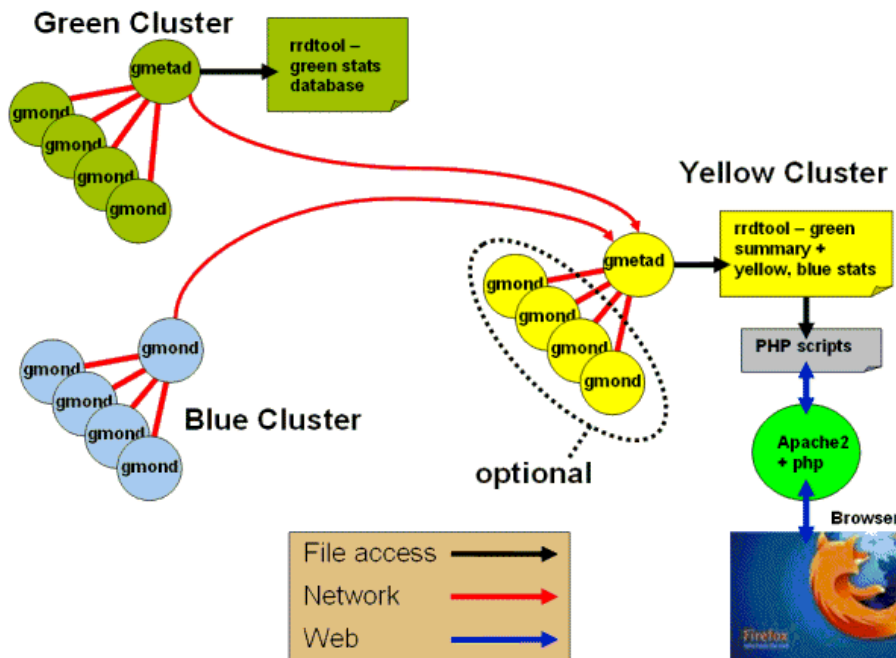


Figure 2-19: Ganglia* Monitoring System for a Cluster

For integration with system management and monitoring systems like Ganglia*, the Manycore Platform Software Stack (MPSS) :

- Provides an interface for the Ganglia* monitoring agent to collect monitoring state or data: sysfs or /proc virtual file system exposed by the Linux*-based coprocessor OS on each Intel® Xeon Phi™ coprocessor device.

- Provides a plug-in for custom made metrics about the nodes (that is, Intel® Xeon Phi™ coprocessor cards) that are being monitored by Ganglia*.
- Serves as a reference implementation for the whole Ganglia* monitoring environment setup.

In the Ganglia* reference implementation shown in Figure 2-20, each Intel® Xeon Phi™ coprocessor card can be treated as an independent computing node. Because Intel® Xeon Phi™ coprocessor is running a Linux*-based OS on the card, one can run gmond monitoring agent on the card as-is. Gmond supports configuration files and plug-ins so it is easy to add customized metrics.

For workstation configuration or for a remote server in a cluster environment, gmetad can be run on the host. For gmetad, no customization is needed. All the front-end tools like rrdtool, scripts should be standard Ganglia* configuration.

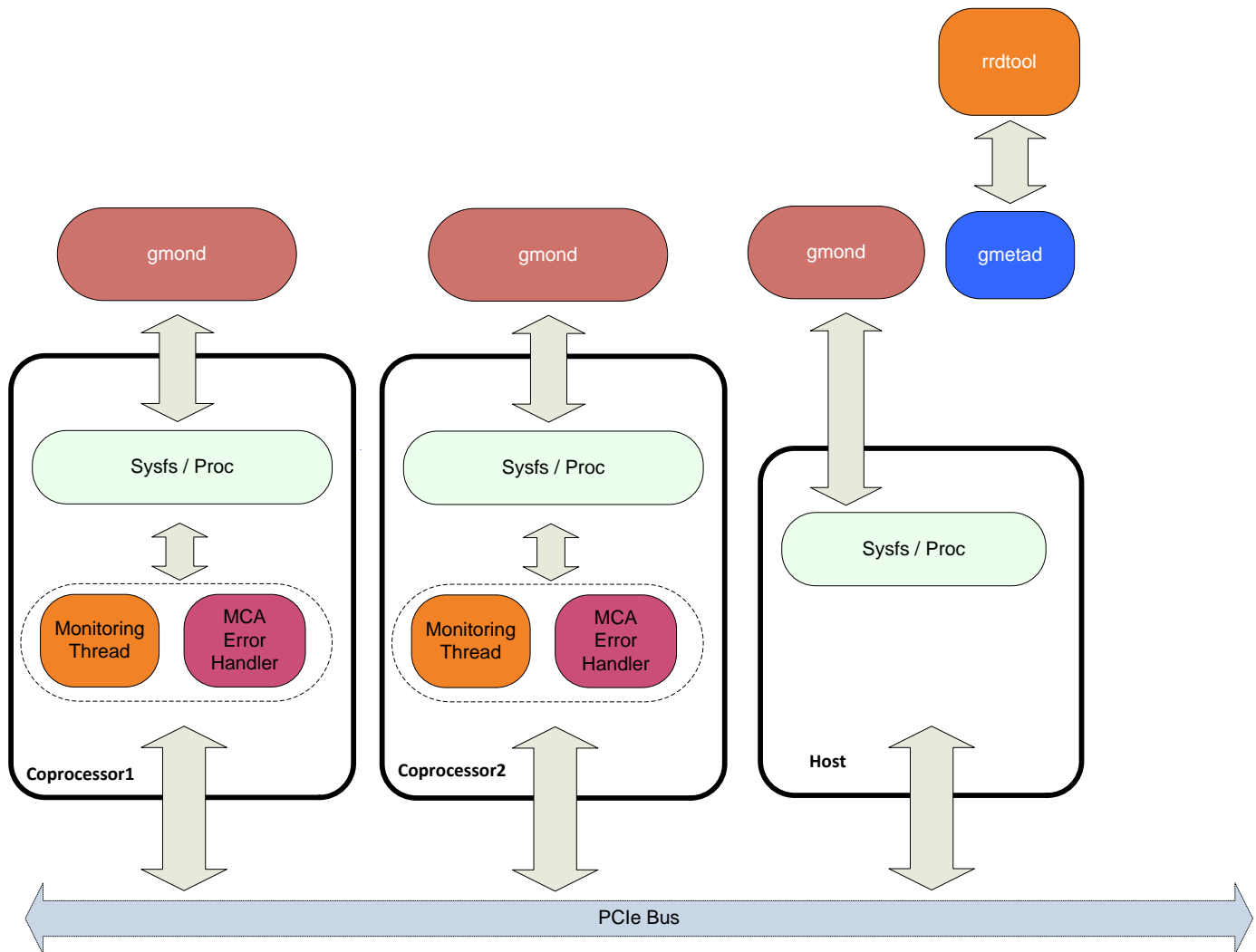


Figure 2-20. Intel® Xeon Phi™ Coprocessor Ganglia* Support Diagram

All of the daemons in Ganglia* talk to each other over TCP/IP. Intel® Xeon Phi™ coprocessor devices are accessible via a TCP/IP subnet off the host, in which the IP component is layered on SCIF.

By default, Ganglia* collects the following metrics:

- cpu_num
- cpu_speed

- mem_total
- swap_total
- boottime
- machine_type
- os_name
- os_release
- location
- gexec
- cpu_user
- cpu_system
- cpu_idle
- cpu_nice
- cpu_aidle
- cpu_wio
- cpu_intr
- cpu_sintr
- load_one
- load_five
- load_fifteen
- proc_run
- proc_total
- mem_free
- mem_shared
- mem_buffers
- mem_cached
- swap_free
- bytes_out
- bytes_in
- pkts_in
- pkts_out
- disk_total
- disk_free
- part_max_used

In addition to these default metrics, the following metrics can be collected on the Intel® Xeon Phi™ coprocessor:

- Intel® Xeon Phi™ coprocessor device utilization
- Memory utilization
- Core utilization
- Die temperature
- Board temperature
- Core frequency
- Memory frequency
- Core voltage
- Memory voltage
- Power consumption
- Fan speed
- Active core number (CPU number is standard)

To collect additional metrics follow these steps:

1. Write a script or C/C++ program which retrieves the information. The script can be written in any scripting language. Python is used to retrieve default metrics. In case of a C/C++ program, the .so files are needed.

2. Register the program with the Ganglia* daemon (gmond) by issuing the Ganglia* command gmetric.
3. Make the registration persistent by adding the modification to the configuration file: /etc/ganglia/gmond.conf.

2.2.7.3 Intel® Manycore Platform Software Stack (MPSS) Service

The Linux* mechanism for controlling system services is used to boot and shut down Intel® Xeon Phi™ coprocessor cards. This service will start (load) and stop (unload) the MPSS to and from the card (e.g. “service mpss start/stop”). This replaces the micstart command utility described in the next section. Please see the README file included in the MPSS tar packages for instructions on how to use this service.

2.2.7.4 Intel® MIC Architecture Commands

This section provides a short summary of available Intel® MIC Architecture commands. More detailed information of each command can be obtained by issuing the ‘-help’ option with each command.

Table 2-12. Intel® MIC Architecture commands

Command	Description
micflash	A command utility normally used to update the Intel® Xeon Phi™ coprocessor PCI Express* card on-board flash. It can also be used to list the various device characteristics.
micinfo	Displays the physical settings and parameters of the card including the driver versions.
micsmc	The Control Panel that displays the card thermal, electrical, and usage parameters. Examples include Core Temperature, Core Usage, Memory Usage, etc. An API for this utility is also available to OEMs under the MicAccess SDK as mentioned previously in the section on the Control Panel.
miccheck	A utility that performs a set of basic checks to confirm that MPSS is correctly installed, all communications links between the host and coprocessor(s), and between coprocessors are functional.

2.2.8 Sysfs Nodes

Sysfs is a Linux* 2.6 virtual file system. It exports information about devices and drivers from the kernel device model to user space; and is similar to the sysctl mechanism found in BSD systems, albeit implemented as a file system. As such, some Intel® Xeon Phi™ coprocessor device characteristics can be obtained from sysfs. Characteristics such as core/cpu utilization, process/thread details and system memory usage are better presented from standard /proc interfaces. The purpose of these sysfs nodes is to present information not otherwise available. The organization of the file system directory hierarchy is strict and is based on the internal organization of kernel data structures.

Sysfs is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It provides two components: a kernel programming interface for exporting these items via sysfs, and a user interface to view and manipulate these items that maps back to the kernel objects they represent. Table 2-13 shows the mapping between internal (kernel) constructs and their external (user space) Sysfs mappings.

Table 2-13. Kernel to User Space Mappings

Internal	External
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

The currently enabled sysfs nodes are listed in Table 2-14.

Table 2-14. SYSFS Nodes

Node	Description
clst	Number of known cores
fan	Fan state
freq	Core frequencies
gddr	GDDR device info
gfreq	GDDR frequency
gvolt	GDDR voltage
hwinf	hardware info (revision, stepping, ...)
temp	Temperature sensor readings
vers	Version string
volt	Core voltage

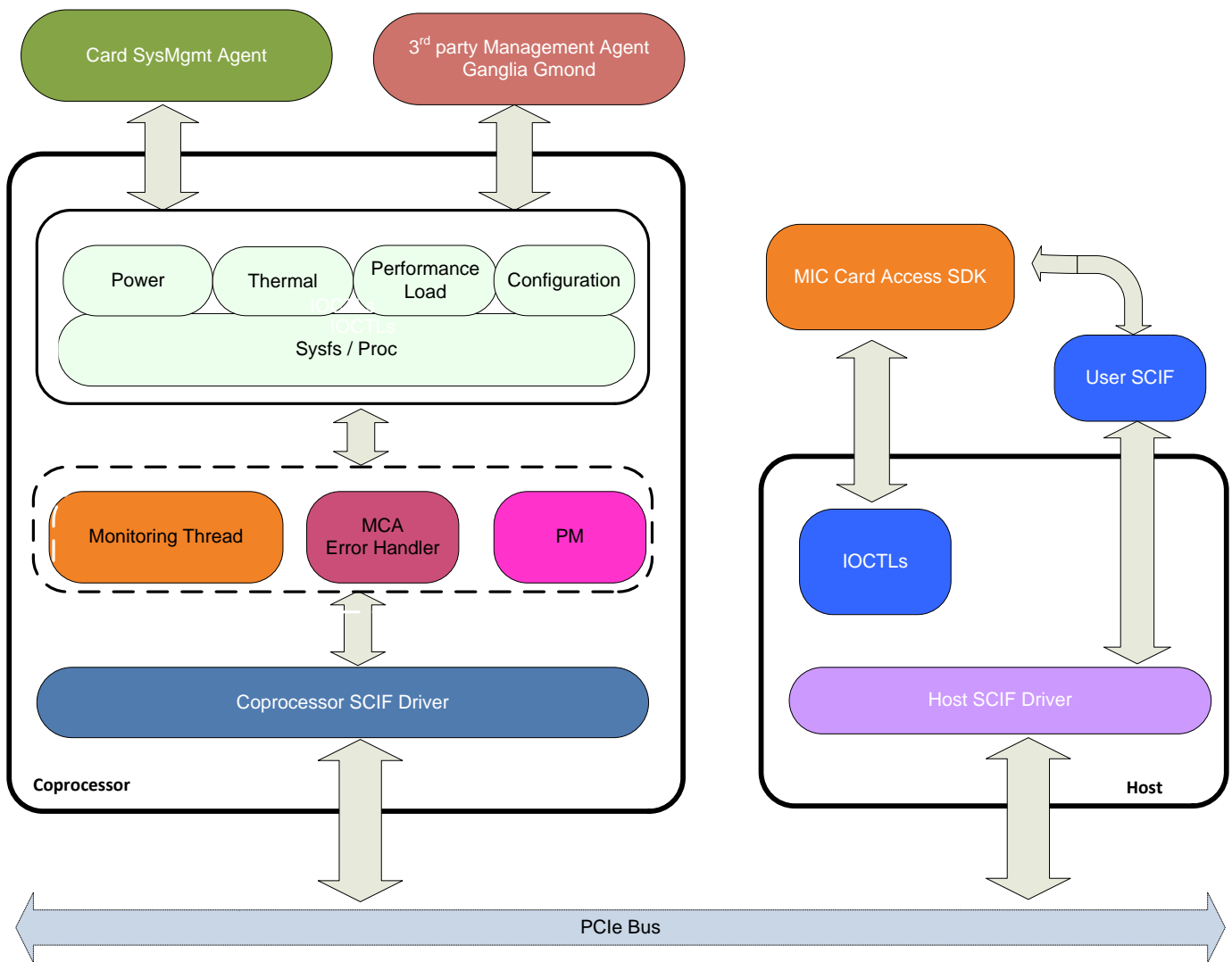


Figure 2-21: MPSS Ganglia* Support

Sysfs is a core piece of the kernel infrastructure that provides a relatively simple interface to perform a simple task. Some popular system monitoring software like Ganglia* uses /proc or the sysfs interface to fetch system status

information. Since the Intel® Xeon Phi™ coprocessor can expose card information through sysfs, a single interface can be maintained for both local and server management.

2.2.9 Intel® Xeon Phi™ Coprocessor Software Stack for MPI Applications

This section covers the architecture of the Intel® Xeon Phi™ coprocessor software stack components to enable μ DAPL and IB verbs support for MPI. Given the significant role of MPI in high-performance computing, the Intel® Xeon Phi™ coprocessor has built-in support for OFED* (Open Fabrics Enterprise Edition) which is widely used in high performance computing for applications that require high efficiency computing, wire-speed messaging, and microsecond latencies. OFED* is also the preferred communications stack for the Intel® MPI Library, allowing Intel® MIC Architecture to take advantage of remote direct memory access (RDMA) capable transport that it exposes. The Intel® MPI Library for Intel® MIC Architecture on OFED* can use SCIF or physical InfiniBand* HCA (Host Channel Adapter) for communications between Intel® Xeon Phi™ coprocessor devices and between an Intel® Xeon Phi™ coprocessor and the host; in this way, Intel® Xeon Phi™ coprocessor devices are treated as stand-alone nodes in an MPI network.

There are two implementations that cover internode and intranode communications through the InfiniBand* HCA:

- CCL (Coprocessor Communication Link). A proxy driver that allows access to a hardware InfiniBand* HCA from the Intel® Xeon Phi™ coprocessor.
- OFED*/SCIF. A software-based InfiniBand*-like device that allows communication within the box.

This guide only covers the first level decomposition of the software into its major components and describes how these components are used. This information is based on the OpenFabrics Alliance* (OFA*) development effort. Because open source code is constantly changing and evolving, developers are responsible for monitoring the OpenFabrics Alliance* to ensure compatibility.

2.2.9.1 Coprocessor Communication Link (CCL)

To efficiently communicate with remote systems, applications running on Intel® Many Integrated Core Architecture (Intel® MIC Architecture) coprocessors require direct access to RDMA devices in the host platform. This section describes an architecture providing this capability (called CCL) that is targeted for internode communication.

In a heterogeneous computing environment, it is desirable to have efficient communication mechanisms from all processors, whether they are the host system CPUs or Intel® Xeon Phi™ coprocessor cores. Providing a common, standards-based, programming and communication model, especially for clustered system applications is an important goal of the Intel® Xeon Phi™ coprocessor software. A consistent model not only simplifies development and maintenance of applications, but allows greater flexibility for using a system to take full advantage of its performance.

RDMA architectures such as InfiniBand* have been highly successful in improving performance of HPC cluster applications by reducing latency and increasing the bandwidth of message passing operations. RDMA architectures improve performance by moving the network interface closer to the application, allowing kernel bypass, direct data placement, and greater control of I/O operations to match application requirements. RDMA architectures allow process isolation, protection, and address translation to be implemented in hardware. These features are well-suited to the Intel® Xeon Phi™ coprocessor environment where host and coprocessor applications execute in separate address domains.

CCL brings the benefits of RDMA architecture to the Intel® Xeon Phi™ coprocessor. In contrast, without CCL, communications into and out of attached processors must incur an additional data copy into host memory, substantially impacting both message latency and achievable bandwidth. Figure 2-22 illustrates the operation of an RDMA transfer with CCL and an Intel® Xeon Phi™ coprocessor add-in PCI Express* card.

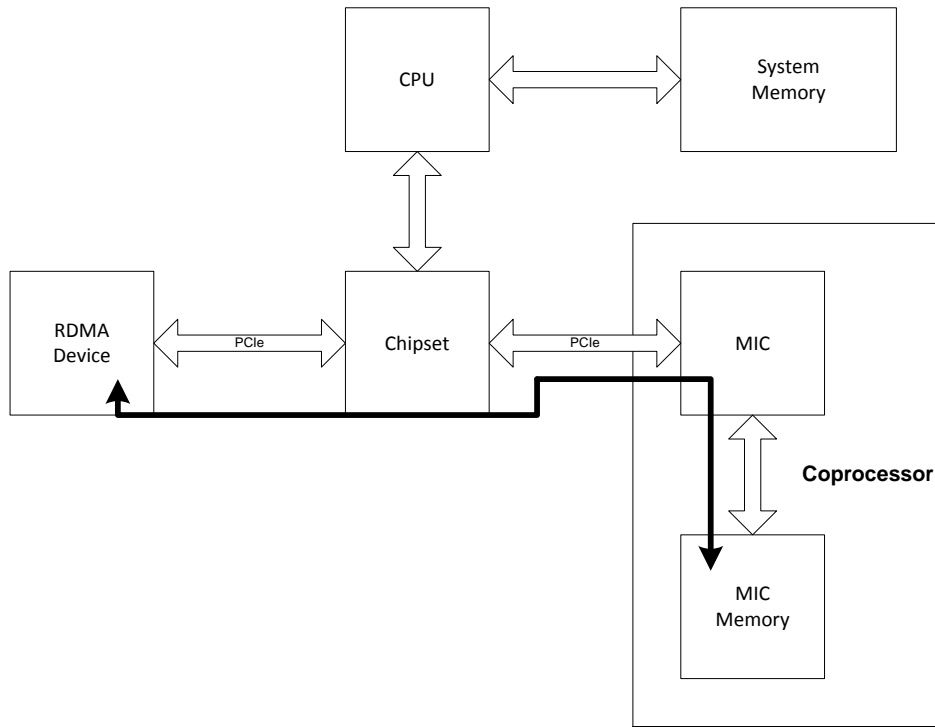


Figure 2-22 RDMA Transfer with CCL

CCL allows RDMA device hardware to be shared between Linux*-based host and Intel® Xeon Phi™ coprocessor applications. Figure 2-23 illustrates an MPI application using CCL.

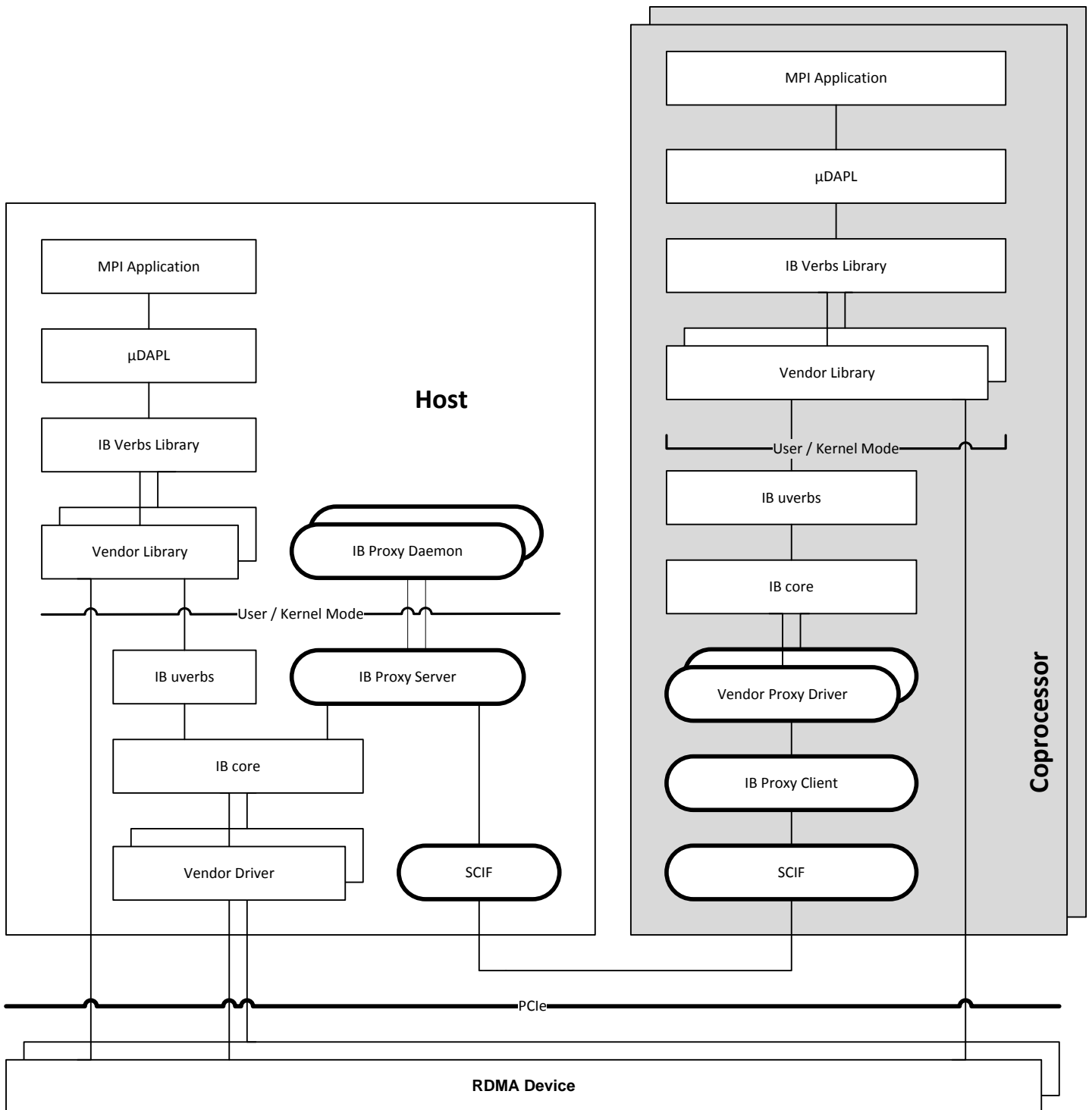


Figure 2-23 MPI Application on CCL

Figure 2-23 highlights the primary software modules (bolded rounded components) responsible for CCL. The host system contains a PCI Express* interface with one or more RDMA devices and one or more Intel® Xeon Phi™ coprocessor add-in cards. Software modules on the host and Intel® Xeon Phi™ coprocessor communicate with each other and access RDMA devices across the PCI Express* bus. The software uses a split-driver model to proxy operations across PCI Express* to manage RDMA device resources allocated by the Vendor Driver on the host. These modules include the IB* Proxy Daemon, the IB* Proxy Server, the IB* Proxy Client, the Vendor Proxy Drivers, and SCIF.

RDMA operations are performed by a programming interface known as verbs. Verbs are categorized into privileged and non-privileged classes. Privileged verbs are used to allocate and manage RDMA resources. Once these resources have

been initialized, non-privileged verbs are used to perform I/O operations. I/O operations can be executed directly to and from user-mode applications on the Intel® Xeon Phi™ coprocessor concurrently with host I/O operations, with kernel-mode bypass, and with direct data placement. The RDMA device provides process isolation and performs address translation needed for I/O operations. CCL proxies privileged verb operations between host and Intel® Xeon Phi™ coprocessor systems such that each Intel® Xeon Phi™ coprocessor PCI Express* card appears as if it were another “user-mode” process above the host IB* core stack.

2.2.9.1.1 IB* Core Modifications

The IB* core module defines the kernel-mode verbs interface layer and various support functions. Support functions that allow vendor drivers to access user-mode data are:

- `ib_copy_to_udata()`
- `ib_copy_from_udata()`
- `ib_umem_get()`
- `ib_umem_page_count()`
- `ib_umem_release()`

These functions may be used by vendor drivers for privileged verb operations. Since the implementation of these functions assumes that data is always in host system user-space, modifications allowed redirection of these functions for CCL. The IB* Proxy Server overrides the default implementation of these functions to transfer data to or from the Intel® Xeon Phi™ coprocessor as needed. To be effective, vendor drivers must use the support functions provided by IB* core.

2.2.9.1.2 Vendor Driver Requirements

The IB* core module provides support functions that allow Vendor Drivers to access user-mode data. Instead of using the IB* core support functions, however, some Vendor Driver implementations call user-mode access routines directly. Table 2-15 lists drivers that require modification to work with CCL. Currently, only the Mellanox HCAs are supported.

Table 2-15: Vendor Drivers Bypassing IB* Core for User-Mode Access

	amso1100*	cxgb3*	cxgb4*	ehca*	ipath*	mlx4*	mthca*	nes*	qib*
copy_to_user					X				X
copy_from_user					X				X
get_user_pages					X		X		X

Beyond utilizing the IB* core interface support functions, there are additional requirements for enabling Vendor Drivers to take full advantage of CCL. Table 2-16 shows that RDMA is divided into two distinct architectures, InfiniBand* and iWARP*.

The underlying process for establishing a connection differs greatly between InfiniBand* and iWARP* architectures. Although InfiniBand* architecture defines a connection management protocol, it is possible to exchange information out-of-band and directly modify a queue pair to the connected state. μDAPL implements a socket CM (SCM) protocol that utilizes this technique and only requires user-mode verbs access through CCL. For iWARP* architecture, however, this requires the `rdma_cm` kernel module to invoke special iWARP* CM verbs. Therefore, to support iWARP* devices, CCL must proxy `rdma_cm` calls between the host and the Intel® Xeon Phi™ coprocessor.

As shown in Table 2-16, the IBM* eHCA* device is not supported on x86 architecture; it requires a PowerPC* system architecture, which is not supported by Intel® Xeon Phi™ coprocessor products.

QLogic* provides `ipath*` and `qib*` drivers, which are hybrid hardware/software implementations of InfiniBand* that in some cases use `memcpy()` to transfer data and that do not provide full kernel bypass.

Table 2-16: Summary of Vendor Driver Characteristics

Driver	Vendor	RDMA Type	x86 Support	Kernel Bypass
cxgb3*	Chelsio Communications*	iWARP*	yes	yes
cxgb4*	Chelsio Communications*	iWARP*	yes	yes
ehca*	IBM Corporation*	InfiniBand*	no	yes
ipath*	QLogic*	InfiniBand*	yes	no
mlx4*	Mellanox Technologies*	InfiniBand*	yes	yes
mthca*	Mellanox Technologies*	InfiniBand*	yes	yes
nes*	Intel Corporation	iWARP*	yes	yes
qib*	QLogic*	InfiniBand*	yes	no

2.2.9.1.3 IB* Proxy Daemon

The IB* Proxy Daemon is a host user-mode application. It provides a user-mode process context for IB* Proxy Server calls (through the IB* core) to the underlying vendor drivers. A user-mode process context is needed to perform memory mappings without modifying the existing vendor drivers. Vendor drivers typically map RDMA device MMIO memory into the calling user-mode process virtual address space with `ioremap()`, which requires a valid user-mode `current->mm` structure pointer.

An instance of the IB* Proxy Daemon is started via a `udev` “run” rule for each Intel® Xeon Phi™ coprocessor device added by the IB* Proxy Server. The IB* Proxy Daemon is straightforward. It immediately forks to avoid blocking the `udev` device manager thread. The parent process exits while the child examines the action type for device add notifications; all other notifications are ignored and the daemon simply exits. If a device add notification is received, the device is opened followed by zero byte write. It is this call to write that provides the user-mode process context used by the IB* Proxy Server. When the IB* Proxy Server relinquishes the thread, the write completes, and the IB* Proxy Daemon closes the device and exits.

2.2.9.1.4 IB* Proxy Server

The IB* Proxy Server is a host kernel module. It provides communication and command services for Intel® Xeon Phi™ coprocessor IB* Proxy Clients. The IB* Proxy Server listens for client connections and relays RDMA device add, remove, and event notification messages. The IB* Proxy Server initiates kernel-mode IB* verbs calls to the host IB* core layer on behalf of Intel® Xeon Phi™ coprocessor IB* Proxy Clients and returns their results.

Upon initialization, the IB* Proxy Server registers with the host IB* core for RDMA device add and remove callbacks, and creates a kernel thread that listens for Intel® Xeon Phi™ coprocessor connections through SCIF. The IB* Proxy Server maintains a list of data structures for each side of its interface. One list maintains RDMA device information from IB* core add and remove callbacks, while another list maintains connections to IB* Proxy Clients running on the Intel® Xeon Phi™ coprocessor. Together these lists preserve the state of the system so that RDMA device add and remove messages are forwarded to IB* Proxy Clients.

When an IB* Proxy Client connection is established through SCIF, the IB* Proxy Server creates a device that represents the interface. The device exists until the SCIF connection is lost or is destroyed by unloading the driver. The Linux* device manager generates `udev` events for the device to launch the IB* Proxy Daemon. The IB* Proxy Server uses the IB* Proxy Daemon device write thread to send add messages for existing RDMA devices to the IB* Proxy Client, and enters a loop to receive and process client messages. Any RDMA device add or remove notifications that occur after the IB* Proxy Client SCIF connections are established are sent from the IB* core callback thread. In addition, the IB* Proxy

Server forwards asynchronous event and completion queue notification messages from IB* core to IB* Proxy Clients. These messages are also sent from the IB* core callback thread.

The IB* Proxy Server performs verbs on behalf of IB* Proxy Clients. Received messages are dispatched to an appropriate verb handler where they are processed to generate a verb response message. Verbs are synchronous calls directed to specific Vendor Drivers through the IB* core interface. The IB* Proxy Server performs pre- and post-processing operations as required for each verb, and maintains the state required to teardown resources should a SCIF connection abruptly terminate. Privileged verbs provide access to user-mode data to Vendor Drivers through IB* core support functions. The IB* Proxy Server overrides the default implementation of these functions to transfer data to or from Intel® Xeon Phi™ coprocessors as needed.

2.2.9.1.5 IB* Proxy Client

The IB* Proxy Client is an Intel® Xeon Phi™ coprocessor kernel module. The IB* Proxy Client provides a programming interface to vendor proxy drivers to perform IB* verbs calls on the host. The interface abstracts the details of formatting commands and performing the communication. The IB* Proxy Client invokes callbacks for device add, remove, and event notifications to registered Intel® Xeon Phi™ coprocessor Vendor Proxy Drivers.

Upon initialization, the IB* Proxy Client creates a kernel thread to establish a connection to the IB* Proxy Server through SCIF. The IB* Proxy Client maintains a list of data structures for each side of its interface. One list maintains RDMA device information received from IB* Server add and remove messages, while another list maintains Vendor Proxy Drivers that have registered with the IB* Proxy Client. Together, these lists preserve the state of the system so that RDMA device add and remove callbacks are forwarded to Vendor Proxy Drivers as required.

When a connection to the IB* Proxy Server is established through SCIF, the IB* Proxy Client enters a loop to receive and process server messages. With the exception of verb response messages, all device add, remove, asynchronous event, and completion queue notification messages are queued for processing on a Linux work queue. Processing these messages on a separate thread is required to avoid a potential communication deadlock with the receive thread. Device add and remove message callbacks are matched to registered Vendor Proxy Drivers using PCI vendor and device ID information. Asynchronous event and completion queue notifications are dispatched to callback handlers provided upon resource creation or to the Intel® Xeon Phi™ coprocessor IB* core layer.

The IB* Proxy Client provides a verbs command interface for use by Vendor Proxy Drivers. This interface is modeled after the IB* Verbs Library command interface provided for user-mode Vendor Libraries. A Vendor Proxy Driver uses this interface to perform IB* verbs calls to the Vendor Driver on the host. The interface abstracts the details of formatting commands and performing the communication through SCIF. Verbs are synchronous calls; the calling thread will block until the corresponding verb response message is received to complete the operation.

2.2.9.1.6 Vendor Proxy Driver

A vendor proxy driver is an Intel® Xeon Phi™ coprocessor kernel module. Different vendor proxy drivers may be installed to support specific RDMA devices. Upon initialization, each Vendor Proxy Driver registers with the IB* Proxy Client for RDMA device add and remove notifications for the PCI vendor and device IDs that it supports. The Vendor Proxy Driver uses the programming interface provided by the IB* Proxy Client to perform kernel-mode IB* verbs calls. The Vendor Proxy Driver handles the transfer and interpretation of any private data shared between the vendor library on the Intel® Xeon Phi™ coprocessor and vendor driver on the host.

A vendor proxy driver announces that a device is ready for use when it calls the IB* core `ib_register_device()` function. All initialization must be complete before this call. The device must remain usable until the call to `ib_unregister_device()` has returned, which removes the device from the IB* core layer. The Vendor Proxy Driver must call `ib_register_device()` and `ib_unregister_device()` from process context. It must not hold any semaphores that could cause deadlock if a consumer calls back into the driver across these calls.

Upper level protocol consumers registered with the IB* core layer receive an add method callback indicating that a new device is available. Upper-level protocols may begin using a device as soon as the add method is called for the device. When a remove method callback is received, consumers must clean up and free all resources relating to a device before returning from the remove method. A consumer is permitted to sleep in the add and remove methods. When a Vendor Proxy Driver call to `ib_unregister_device()` has returned, all consumer allocated resources have been freed.

Each vendor proxy driver provides verb entry points through an `ib_device` structure pointer in the `ib_register_device()` call. All of the methods in the `ib_device` structure exported by drivers must be fully reentrant. Drivers are required to perform all synchronization necessary to maintain consistency, even if multiple function calls using the same object are run simultaneously. The IB* core layer does not perform any serialization of verb function calls.

The vendor proxy drivers use the programming interface provided by the IB* Proxy Client to perform IB* verbs calls to the vendor driver on the host. Each vendor proxy driver is responsible for the transfer and interpretation of any private data shared between the vendor library on the Intel® Xeon Phi™ coprocessor and the vendor driver on the host. Privileged verb operations use the default IB* core support functions to transfer data to or from user-mode as needed. The interpretation of this data is vendor specific.

2.2.9.2 OFED*/SCIF

The Symmetric Communications Interface (SCIF) provides the mechanism for internode communication within a single platform, where a node is an Intel® Xeon Phi™ coprocessor device or a host processor complex. SCIF abstracts the details of communicating over PCI Express* (and controlling related coprocessor hardware) while providing an API that is symmetric between the host and the Intel® Xeon Phi™ coprocessor.

MPI (<http://www.mpi-forum.org>) (Message-Passing Interface) on the Intel® Xeon Phi™ coprocessor can use either the TCP/IP or the OFED* stack to communicate with other MPI nodes. The OFED*/SCIF driver enables a hardware InfiniBand* Host Communications Adapter (IBHCA) on the PCI Express* bus to access physical memory on an Intel® Xeon Phi™ coprocessor device. When there is no IBHCA in the platform, the OFED*/SCIF driver emulates an IBHCA, enabling MPI applications on the Intel® Xeon Phi™ coprocessor devices in the platform.

OFED*/SCIF implements a software-emulated InfiniBand* HCA to allow OFED*-based applications, such as the Intel® MPI Library for Intel® MIC Architecture, to run on Intel® MIC Architecture without the presence of a physical HCA. OFED*/SCIF is only used for intranode communication whereas CCL is used for internode communication.

OFED* provides an industrial standard low-latency, high-bandwidth communication package for HPC applications, leveraging the RDMA-based high performance communication capabilities of modern fabrics such as InfiniBand*. SCIF is a communication API (sections 2.2.5.1 and 5.1) for the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) device that defines an efficient and consistent interface for point-to-point communication between Intel® Xeon Phi™ coprocessor nodes, as well as between it and the host. By layering OFED* on top of SCIF, many OFED*-based HPC applications become readily available to Intel® MIC Architecture.

The OFED* software stack consists of multiple layers, from user-space applications and libraries to kernel drivers. Most of the layers are common code shared across hardware from different vendors. Vendor dependent code is confined in the vendor-specific hardware driver and the corresponding user-space library (to allow kernel bypass). Figure 2-24 shows the architecture of the OFED*/SCIF stack. Since SCIF provides the same API for Intel® Xeon Phi™ coprocessor and the host, the architecture applies to both cases.

The rounded bold blocks in Figure 2-24 are the modules specific to OFED*/SCIF. These modules include the IB-SCIF Library, IB-SCIF Driver, and SCIF (the kernel space driver only).

2.2.9.2.1 IB-SCIF Library

The IB-SCIF Library is a user-space library that is required by the IB Verbs Library to work with the IB-SCIF Driver. It defines a set of routines that the IB Verbs Library calls to complete the corresponding functions defined by the user-mode IB Verbs API. This allows vendor specific optimization (including kernel bypass) to be implemented in user space. The IB-SCIF Library, however, does not provide kernel bypass; it relays user-mode requests to the kernel-mode driver through the interface exposed by the IB uverbs driver.

2.2.9.2.2 IB-SCIF Driver

The IB-SCIF Driver is a kernel module that implements a software-based RDMA device. At initialization, it sets up one connection between each pair of SCIF nodes, and registers to the IB core driver as an “iWARP” device (to avoid MAD related functions being used). For certain OFED* operations (plain RDMA read/write), data is transmitted directly using the SCIF RMA functions. For all other OFED* operations, data is transmitted as packets, with headers that identify the communication context so that a single connection between two SCIF nodes is sufficient to support an arbitrary number of logical connections. Under the packet protocol, small-sized data is transmitted with the `scif_send()` and `scif_recv()` functions; and large-sized data is transmitted with the SCIF RMA functions after a hand shaking. When both ends of the logical connection are on the same SCIF node (i.e. loopback), data is copied directly from the source to the destination without involving SCIF.

2.2.9.2.3 SCIF (See also Section 5.1)

The SCIF kernel module provides a communication API between Intel® Xeon Phi™ coprocessors and between an Intel® Xeon Phi™ coprocessor and the host. SCIF itself is not part of OFED*/SCIF. OFED*/SCIF uses SCIF as the only internode communication channel (in SCIF terminology, the host is a node, and each Intel® Xeon Phi™ coprocessor card is a separate node). Although there is a SCIF library that provides similar API in the user space, that library is not used by OFED*/SCIF.

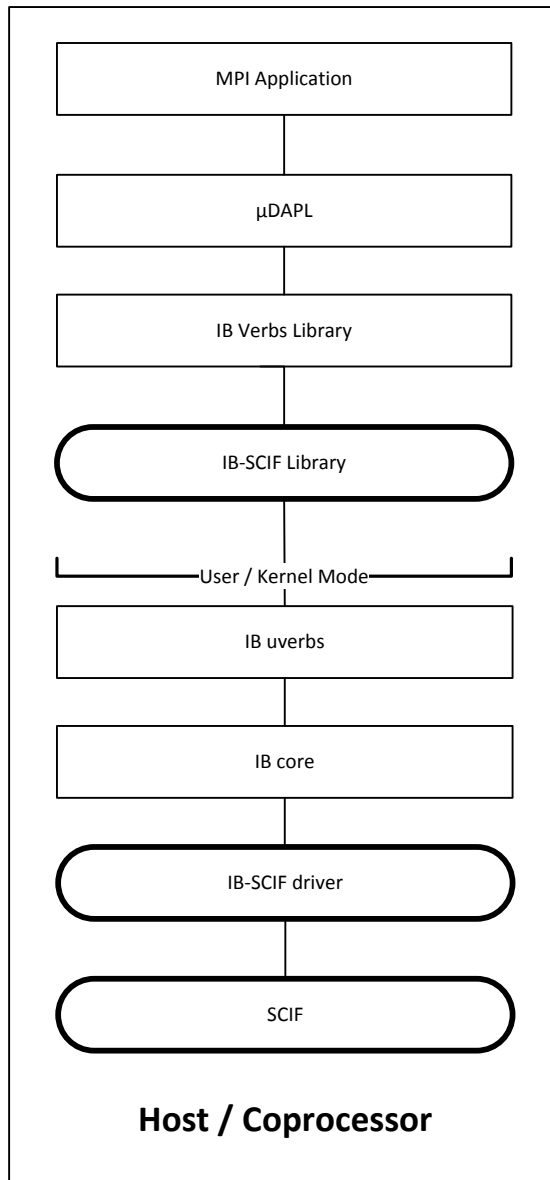


Figure 2-24: OFED*/SCIF Modules

2.2.9.3 Intel® MPI Library for Intel® MIC Architecture

The Intel® MPI Library for Intel® MIC Architecture provides only the Hydra process manager (PM). Each node and each coprocessor are identified using their unique symbolic or IP addresses. Both external (e.g., command line) and internal (e.g., MPI_Comm_Spawn) methods of process creation and addressing capabilities to place executables explicitly on the nodes and the coprocessors are available. This enables you to match the target architecture and the respective executables.

Within the respective units (host nodes and coprocessors), the MPI processes are placed and pinned according to the default and eventual explicit settings as described in the Intel® MPI Library documentation. The application should be able to identify the platform it is running on (host or coprocessor) at runtime.

The Intel® MPI Library for Intel® MIC Architecture supports the communication fabrics shown in Figure 2-25.

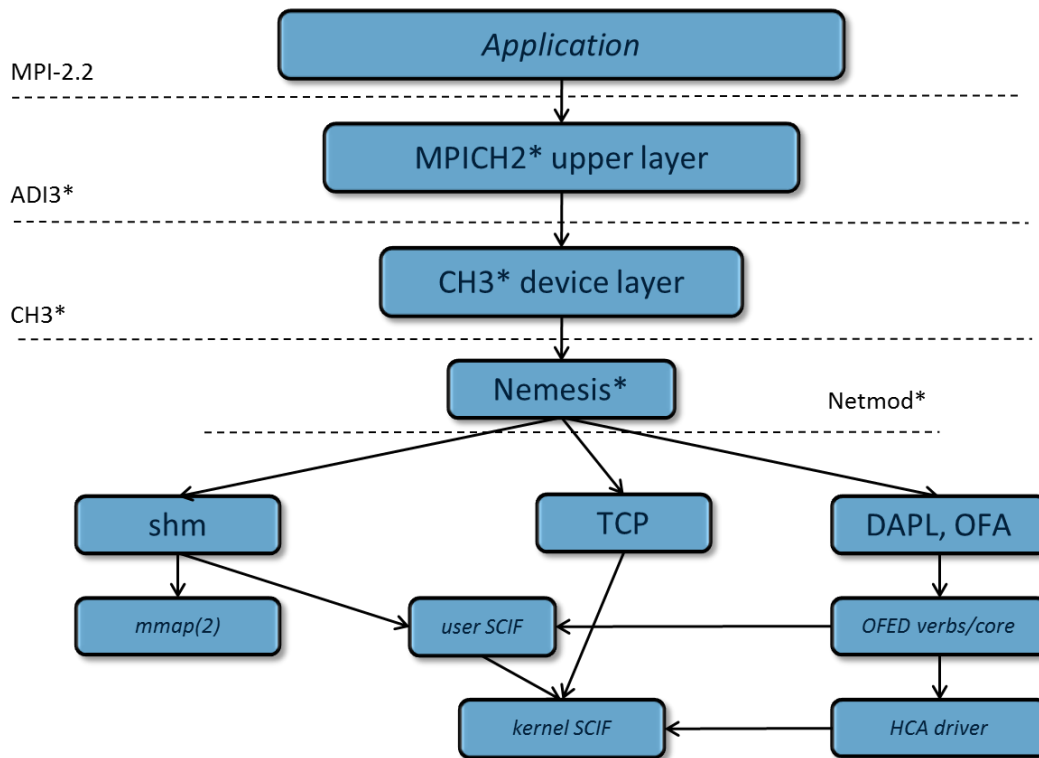


Figure 2-25. Supported Communication Fabrics

2.2.9.3.1 Shared Memory

This fabric can be used within any coprocessor, between the coprocessors attached to the same node, and between a specific coprocessor and the host CPUs on the node that the coprocessor is attached to. The intracoprocessor communication is performed using the normal mmap(2) system call (shared memory approach). All other communication is performed in a similar way based on the scif_mmap(2) system call of the Symmetric Communication Interface (SCIF). This fabric can be used exclusively or combined with any other fabric, typically for higher performance.

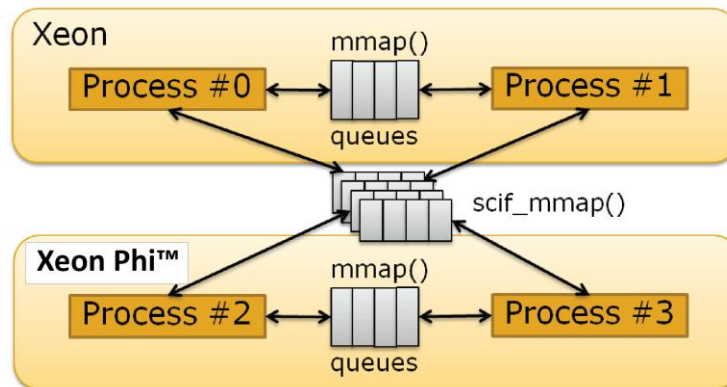


Figure 2-26. Extended SHM Fabric Structure

The overall structure of the extended SHM fabric is illustrated in Figure 2-26. The usual shared memory (SHM) communication complements the SCIF SHM extension that supports multisocket platforms, each socketed processor having a PCI Express* interface. SCIF-based SHM extensions can be used between any host processor and any Intel® Xeon Phi™ coprocessor, and between any two such coprocessors connected to separate PCI Express* buses.

2.2.9.3.2 DAPL/OFA*

This fabric is accessible thru two distinct interfaces inside the Intel® MPI Library: the Direct Application Programming Library (DAPL*) and the Open Fabrics Association (OFA*) verbs [also known as Open Fabrics Association Enterprise Distribution (OFED*) verbs] of the respective Host Channel Adaptor (HCA). In both cases, the typical Remote Memory Access (RMA) protocols are mapped upon the appropriate parts of the underlying system software layers; in this case, `scif_writeto(2)` and `scif_readfrom(2)` SCIF system calls.

2.2.9.3.3 TCP

This fabric is normally the slowest of all fabrics available. This fabric is normally used as a fallback communication channel when the higher performance fabrics mentioned previously cannot be used for some reason.

2.2.9.3.4 Mixed Fabrics

All these fabrics can be used in reasonable combinations for the sake of better performance; for example, `shm:dapl`, `shm:OFA*`, and `shm:tcp`. All the default and eventual explicit settings described in the Intel® MPI Library documentation are inherited by the Intel® MPI Library for Intel® MIC Architecture. This also holds for the possibility of intranode use of both the shared memory and RDMA interfaces such as DAPL or OFA*.

2.2.9.3.5 Standard Input and Output

Finally, the Intel® MPI Library for Intel® MIC Architecture supports the following types of input/output (I/O):

- **Standard file I/O.** The usual standard I/O streams (`stdin`, `stdout`, `stderr`) are supported through the Hydra PM as usual. All typical features work as expected within the respective programming model. The same is true for the file I/O.
- **MPI I/O.** All MPI I/O features specified by the MPI standard are available to all processes if the underlying file system(s) support it.

Please consult the (Intel® MPI Library for Intel® MIC Architecture, 2011-2012) user guide for details on how to set up and get MPI applications running on systems with Intel® Xeon Phi™ coprocessors.

2.2.10 Application Programming Interfaces

Several application programming interfaces (APIs) aid in porting applications to the Intel® Xeon Phi™ coprocessor system. They are the sockets networking interface, the Message Passing Interface (MPI), and the Open Computing Language (OpenCL*), and are industry standards that can be found in multiple execution environments. Additionally, the SCIF APIs have been developed for the Intel® Xeon Phi™ coprocessor.

2.2.10.1 SCIF API

SCIF serves as the backbone for intraplatform communication and exposes low-level APIs that developers can program to. A more detailed description of the SCIF API can be found in Section 5.

2.2.10.2 NetDev Virtual Networking

The virtual network driver provides a network stack connection across the PCI Express* bus. The NetDev device driver emulates a hardware network driver and provides a TCP/IP network stack across the PCI Express* bus. The Sockets API and library provide parallel applications with a means of end-to-end communication between computing agents (nodes) that is based on a ubiquitous industry standard. This API implemented upon the TCP/IP protocol stack simplifies application portability and scalability. Other standard networking services, such as NFS, can be supported through this networking stack. See Section 5 for more details.

3 Power Management, Virtualization, RAS

The server management and control panel component of the Intel® Xeon Phi™ coprocessor software architecture provides the system administrator with the runtime status of the Intel® Xeon Phi™ coprocessor card(s) installed into a given system. There are two use cases that are of interest. The first is the rack-mounted server that is managed remotely and that relies on 3rd-party management software. The second is a stand-alone pedestal or workstation system that uses a local control panel application to access information stored on the system. Applications of this type are designed to execute in a specific OS environment, and solutions for both the Linux* and the Windows operating systems are available. Although these implementations may utilize common modules, each must address the particular requirements of the target host OS.

There are two access methods by which the System Management (SM)/control panel component may obtain status information from the Intel® Xeon Phi™ coprocessor devices. The “in-band” method uses the SCIF network and the capabilities designed into the coprocessor OS and the host driver; delivers Intel® Xeon Phi™ coprocessor card status to the user; and provides a limited ability to control hardware behavior. The same information can be obtained using the “out-of-band” method. This method starts with the same capabilities in the coprocessors, but sends the information to the Intel® Xeon Phi™ coprocessor card’s System Management Controller (SMC). The SMC can then respond to queries from the platform’s BMC using the IPMB protocol to pass the information upstream to the user.

3.1 Power Management (PM)

Today’s power management implementations increasingly rely on multiple software pieces working cooperatively with hardware to improve the power and performance of the platform, while minimizing the impact on performance. Intel® MIC Architecture based platforms are no exception; power management for Intel® Xeon Phi™ coprocessors involves multiple software levels.

Power management for the Intel® Xeon Phi™ coprocessor is predominantly performed in the background. The power management infrastructure collects the necessary data to select performance states and target idle states, while the rest of the Intel® Manycore Platform Software Stack (MPSS) goes about the business of processing tasks for the host OS. In periods of idleness, the PM software places Intel® Xeon Phi™ coprocessor hardware into one of the low-power idle states to reduce the average power consumption.

Intel® Xeon Phi™ coprocessor power management software is organized into two major blocks. One is integrated into the coprocessor OS running locally on the Intel® Xeon Phi™ coprocessor hardware. The other is part of the host driver running on the host. Each contributes uniquely to the overall PM solution.

MIC Power Management Software Architecture

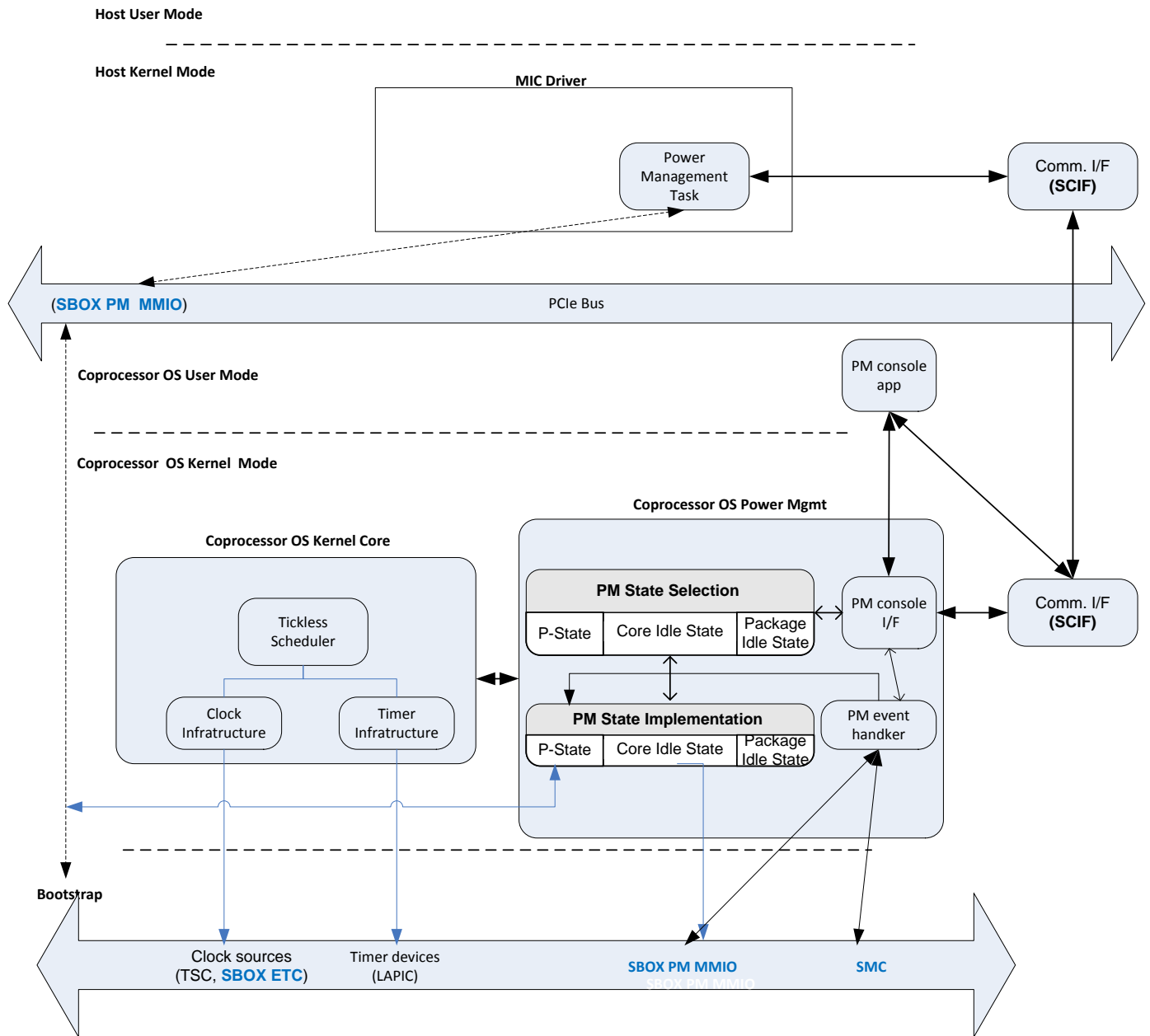


Figure 3-1. Intel® Xeon Phi™ Coprocessor Power Management Software Architecture

3.1.1 Coprocessor OS Role in Power Management

Because this code controls critical power and thermal management safeguards, modification of this code may void the warranty for Intel® Xeon Phi™ coprocessor devices used with the modified code.

Power management capabilities within the coprocessor OS are performed in the kernel at ring0. The one exception is that during PC6 exit, the bootloader plays an important role after loss of core power.

Primarily the coprocessor OS kernel code is responsible for managing:

- Selection and setting of the hardware's performance level (P-states) including any "Turbo Mode" capability that may be present.
- Data collection used to assess the level of device utilization; device thermal and power consumption readings must be collected to support the P-state selection process.
- Modified P-state selection, which is based on externally imposed limits on card power consumption.
- Selection and setting of core idle states (C-states).
- Data collection to assess the level of device utilization that will be used to support core C-state selection.
- Save and restore CPU context on core C6 entry and exit.
- Orchestrate the entry and exit of the package to Auto C3 package state in order to ensure that the coprocessor OS is able to meet the scheduled timer deadlines.
- Prepare the Intel® Xeon Phi™ coprocessor for entry into the PC6-state (that is, to make sure all work items are completed before entering PC6-state), save and restore machine context before PC6 entry and after PC6 exit, and return to full operation after PC6-state exit. The Bootloader then performs reset initialization and passes control to the GDDR resident coprocessor OS kernel.

PM services executing at Ring0 provide the means to carry out many of the PM operations required by the coprocessor OS. These services are invoked at key event boundaries in the OS kernel to manage power on the card. The active-to-idle transition of the CPU in the kernel is one such event boundary that provides an opportunity for PM services in the kernel to capture data critical for calculating processor utilization. In addition, idle routines use restricted instructions (e.g. HLT or MWAIT) enabling processors to take advantage of hardware C-states. Other services perform or assist in evaluating hardware utilization, selection, and execution of target P- and C-states. Finally, there are services that directly support entry and exit from a particular C-state.

PC-state entry/exit refers to the dedicated execution paths or specific functions used during transition from a C0 state of operation to a specific PC-state (entry) or from a specific PC-state back to the C0 state (exit). To minimize the time required in transition, these dedicated execution paths must be tailored to the specific hardware need of the target PC-state. Minimizing transition times enables PC-states to be used more frequently, thus reducing lower average power consumption without any user-perceived impact on performance.

3.1.2 Bootloader Role in Power Management

The BootLoader is put into service during PC6 exit. This PC6-state lowers the VccP voltage to zero. As a result, the Intel® Xeon Phi™ coprocessor cores begin code execution at the reset vector (i.e. FFFF_FFF0h) when the voltage and clocks are restored to operational levels. However, unlike cycling power at the platform level or at a cold reset, an abbreviated execution path designed specifically for PC6 state exit can be executed. This helps minimize the time required in returning Intel® Xeon Phi™ coprocessor to full operation and prevents a full-scale boot process from destroying GDDR contents that are retained under self-refresh. These shortened execution paths are enabled in part by hardware state retention on sections that remain powered and through the use of a self-refresh mechanism for GDDR memory devices.

3.1.3 Host Driver Role in Power Management

The Host driver plays a central role in power management. Its primary power management responsibilities are:

- To monitor and manage the Intel® Xeon Phi™ coprocessor package idle states.
- To address server management queries.
- To drive the power management command/status interface between the host and the coprocessor OS.
- To interface with the host communication layer.

3.1.4 Power Reduction

The PM software reduces card power consumption by leveraging the Intel® Xeon Phi™ coprocessor hardware features for voltage/core frequency scaling (P-states), core idle states, and package idle states. By careful selection of the available P-states and idle states, the PM software opportunistically reduces power consumption without impacting the application performance. For all the idle and P-states, software follows a two-step approach: state selection followed by state control or setting. The architecture reflects this by grouping the modules as either state-selection or state-setting modules.

3.1.4.1 P-State Selection

The PM software uses Demand Based Scaling (DBS) to select the P-state under which the cores operate. “Demand” refers to the utilization of the CPUs over a periodic interval. An increase in CPU utilization is seen as a signal to raise the core frequency (or to reduce the P-state) in order to meet the increase in demand. Conversely, a drop in utilization is seen as an opportunity to reduce the core frequency and hence save power. The primary requirement of the P-state selection algorithm is to be responsive to changes in workload conditions so that P-states track the workload fluctuations and hence reduce power consumption with little or no performance impact. Given this sensitivity of the P-state selection algorithm to workload characteristics, the algorithm undergoes extensive tuning to arrive at an optimum set of parameters. The software architecture allows for extensive parameterization of the algorithms and even the ability to switch algorithms on the fly. Some of the parameters that can be changed to affect the P-state selection are:

- Evaluation period over which utilization values are calculated.
- Utilization step size over which a P-state selection is effective.
- P-state step size that controls the P-state gradient between subsequent selections.
- Guard bands around utilization thresholds to create a hysteresis in the way the P-states are increased and decreased. This prevents detrimental ping-pong behavior of the P-states.

The architecture supports user-supplied power policy choices that can map to a set of predefined parameters from the list above. Other variables such as power budget for the Intel® Xeon Phi™ coprocessor hardware, current reading, and thermal thresholds can factor into the P-state selection either as individual upper limits that cause the P-states to be throttled automatically, or can be combined in more complex ways to feed into the selection algorithm.

The coprocessor OS has exclusive responsibility for P-state selection. The P-state selection module contains the following routines:

- Initialization
- Evaluation task
- Notification handler

The P-state selection module has interfaces to the core coprocessor OS kernel, the P-state setting module, and the PM Event Handler. The architecture keeps this module independent of the underlying hardware mechanisms for setting P-states (i.e., detecting over-current or thermal conditions, etc.).

The P-state selection module registers a periodic timer task with the coprocessor OS core kernel. The “evaluation period” parameter decides the interval between consecutive invocations of the evaluation task. Modern operating system kernels maintain per-CPU running counters that keep track of the cumulative time that the CPU is idle, that the CPU executes interrupt code, that the CPU executes kernel code, and so on. The evaluation task wakes up every evaluation time period, reads from these per-CPU counters the total time that that CPU was idle during the last evaluation window, and calculates the utilization for that CPU. For the purpose of calculating the target P-state, the maximum utilization value across all CPUs is taken. Since the evaluation task runs in the background while the CPUs are executing application code, it is important that software employs suitable methods to read an internally consistent value for the per-CPU idle time counters without any interference to code execution on the CPUs.

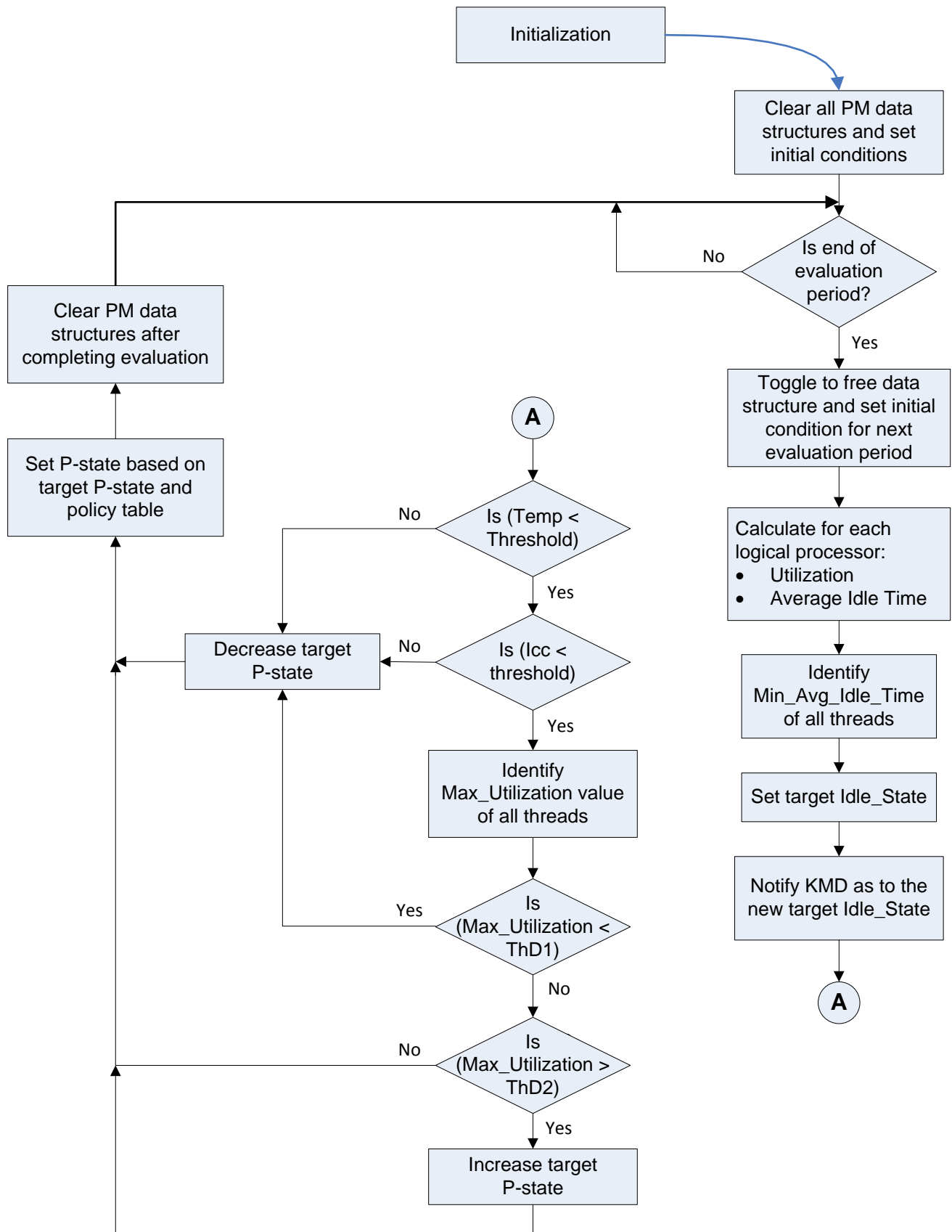


Figure 3-2. Power Reduction Flow

Once the maximum utilization value (as a percentage of evaluation period) across all CPUs is computed, the evaluation task has to map this value to a target P-state. There are a number of ways this can be accomplished. Figure 3-2 shows one way this can be done. Thresholds ThD1 and ThD2 provide the hysteresis guard band within which the P-state remains the same. The goal of this algorithm is to raise P-states (that is, lower core frequency) progressively till the maximum utilization value is increased to a configurable threshold value (THD2) value. As workload demand increases and the maximum utilization increases beyond this threshold, the algorithm decreases the target P-state (increase core frequency) still keeping within power and thermal limits. The threshold values, P-state increase and decrease step size, are all parameters that either map to a policy or set explicitly.

The P-state selection module has to handle notifications from the rest of the system and modify its algorithm accordingly. The notifications are:

- Start and stop thermal throttling due to conditions such as CPUHOT.
- Changes to the card power budget.
- Thermal threshold crossings.
- Changes to power policy and P-state selection parameters.

The notifications bubble up from the PM Event Handler and can be totally asynchronous to the evaluation task. The effect of these notifications can range from modifications to P-state selection to a complete pause or reset of the evaluation task.

The host driver generally does not play an active role in the P-state selection process. However, the host driver interfaces with the coprocessor OS P-state selection module to get P-state information, to set or get policy, and to set or get parameters related to P-state selection.

3.1.4.2 P-State Control

The P-state control module implements the P-states on the target Intel® Xeon Phi™ coprocessor hardware. The process of setting P-states in the hardware can vary between Intel® Xeon Phi™ coprocessor products. Hence the P-state module, by hiding the details of this process from other elements of the PM software stack, makes it possible to reuse large parts of the software between different generations of Intel® Xeon Phi™ coprocessors.

P-state control operations take place entirely within the coprocessor OS. The P-state control module has the following main routines:

- P-state table generation routine
- P-state set/get routine
- SVID programming routine
- Notifier routine

The P-state control module exports:

- Get/set P-state
- Register notification
- Read current value
- Set core frequency/voltage fuse values

On Intel® Xeon Phi™ coprocessor devices (which do not have an architecturally defined mechanism to set P-states, like an MSR write), the mapping of P-states to core frequency and voltage has to be generated explicitly by software and stored in a table. The table generation routine takes as parameters:

- Core frequency and voltage pairs for a minimal set of guaranteed P-states (P_n, P₁ and P₀) from which other pairs can be generated using linear interpolation.
- Core frequency step sizes for different ranges of the core frequency.

- Mapping between core frequency value and corresponding MCLK code.
- Mapping between voltage values and SVID codes.

There are hardware-specific mechanisms by which these P-states are made available to the coprocessor OS. In the Intel® Xeon Phi™ coprocessor, these values are part of a preset configuration space that is read by the bootloader and copied to flash MMIO registers and read by the P-state control module. This routine exports the “Set core frequency/voltage fuse configuration” so that the coprocessor OS flash driver that initializes the MMIO registers containing the fuse configuration can store them before they get initialized.

The P-state Get/Set routine uses the generated P-state table to convert P-states to core frequency and voltage pairs and vice versa.

Other parts of the coprocessor OS may need to be notified of changes to core frequency. For example parts of the coprocessor OS that use the Timestamp Counter (TSC) as a clock source to calculate time intervals must be notified of core frequency changes so that the TSC can be recalibrated. The notifier routine exports a “register notification” interface so that other routines in the coprocessor OS can call-in to register for notification. The routine sends a notification any time a core frequency change occurs as a result of a P-state setting.

3.1.4.3 Idle State Selection

Prudent use of the core and package idle states enables the Intel® Xeon Phi™ coprocessor PM software to further reduce card power consumption without incurring a performance penalty. The algorithm for idle state selection considers two main factors: the expected idle residency and the idle state latency. In general, the deeper the idle state (and hence the greater the power saving), the higher the latency. The formula for deciding the particular idle state to enter is of the form:

$$\text{Expected idle residency} \geq C * (\text{ENTRY_LATENCY}_{Cx} + \text{EXIT_LATENCY}_{Cx})$$

Where:

- C is a constant that is always greater than one and determined by power policy. It can also be set explicitly.
- ENTRY_LATENCY_{Cx} is the time required to enter the Cx idle state.
- Exit_LATENCY_{Cx} is the time required to exit the Cx idle state.

The comparison is performed for each of the supported idle states (Cx) and the deepest idle state that satisfies this comparison is selected as the target idle state. If none of the comparisons are successful, then the target idle state is set to C0 (no idle state).

The expected idle residency for a CPU is a function of several factors; some of which are deterministic such as synchronous events like timers scheduled to happen on the CPU at certain times in the future (that will force the CPU out of its idleness) and some of which are nondeterministic such as interprocessor interrupts.

In order to keep the idle-state selection module independent of the specific Intel® Xeon Phi™ coprocessor, the PM software architecture includes data structures that are used to exchange information between the idle-state selection and hardware-specific idle state control modules, such as the:

- Number of core idle states supported by the hardware
- Number of package idle states supported for each core and package idle state
- Name of the state (for user-mode interfaces)
- Entry and exit latency
- Entry point of the routine to call to set state
- Average historical residency of state

- TSC and LAPIC behavior in this idle state
- Bitmasks marking core CPUs that have selected this idle state

The idle-state control module fills in most of the information in these data structures.

3.1.4.3.1 Core Idle State Selection

The Intel® Xeon Phi™ coprocessor supports a Core C1 idle state and a deeper Core C6 idle state. Both core idle states are a logical AND operations of the individual idle states of the CPUs that make up the core. While entry and exit into the core C1 state needs no software intervention (except the individual CPUs executing a HALT), Core C6 entry and exit require the CPU state to be saved/restored by software. Hence a deliberate choice has to be made by software running on the CPU whether to allow the core (of which the CPU is part) transition to Core C6 state.

3.1.4.3.2 The coprocessor OS Role in Core Idle State Selection

Core idle state selection happens entirely in the coprocessor OS. As mentioned before, modern operating systems have an architecturally defined CPU idle routine. Entry to and exit from idleness occurs within this routine. The core idle-selection module interfaces with this routine to select the core idle state on entry and to collect idleness statistics on exit (to be used for subsequent idle state selections). The core idle state selection module has the following main routines:

- Core idle select
- Core idle update
- Core idle get and set parameter

Figure 3-2 shows the Core C6 selection process in the Intel® Xeon Phi™ coprocessor.

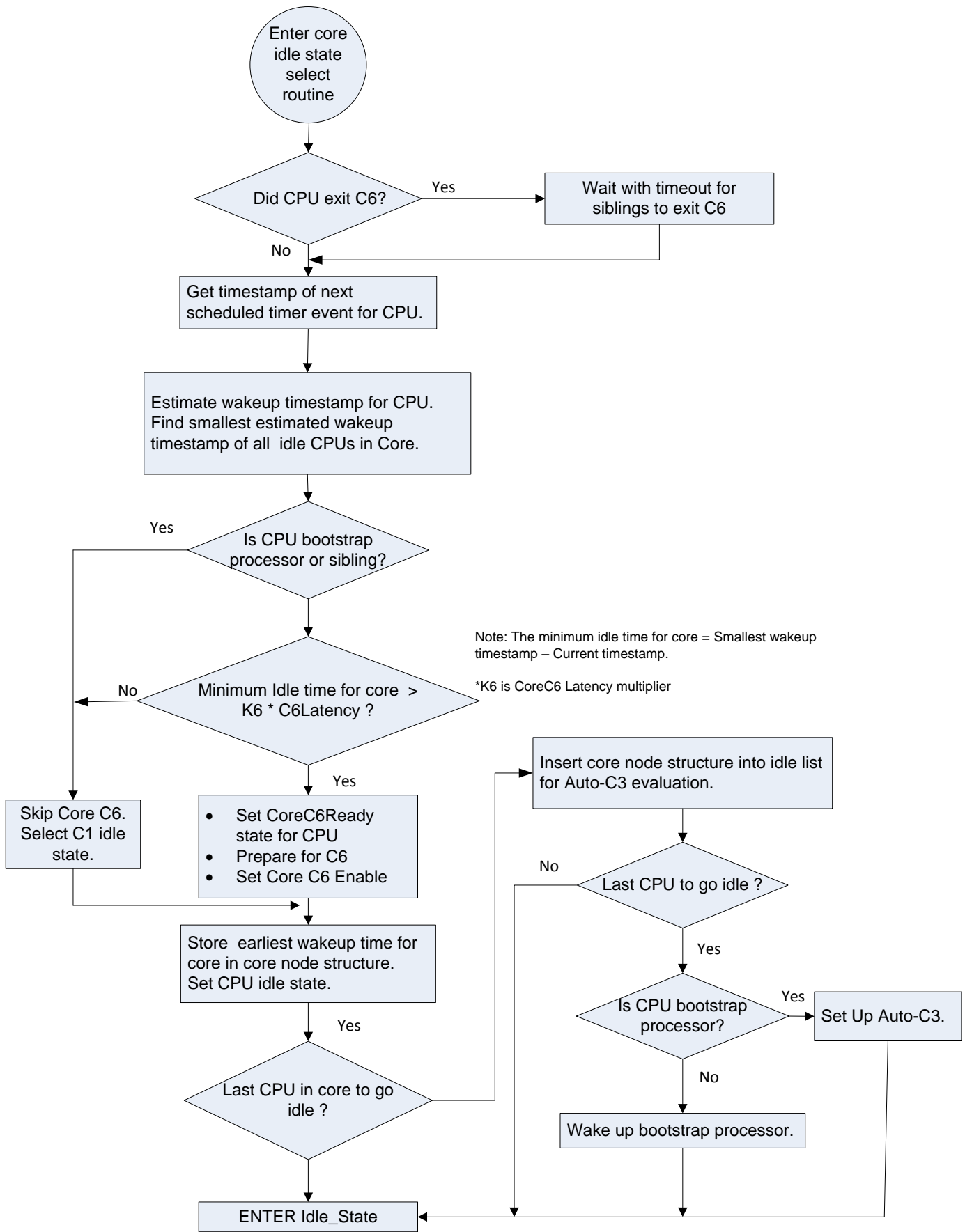


Figure 3-3. Core C6 Selection

3.1.4.3.2.1 Core Idle Select

This routine interfaces to the coprocessor OS CPU idle routine and gets control before the CPU executes the idle instruction (HALT in the case of Intel® Xeon Phi™ coprocessor). The core idle select routine runs the algorithm to compute the expected idle residency of the CPU. The main components in the idle residency calculation are the next timer event time for the CPU and the historic idle residency values for the CPU.

In the case of core C6 for the Intel® Xeon Phi™ coprocessor, the algorithm running on the last CPU in the core to go idle can optionally estimate the idle residency of the core by taking into account the expected idle residency of other idle CPUs in the core and the time elapsed since the other CPUs went idle.

3.1.4.3.2.2 Core Idle Update

This routine interfaces to the coprocessor OS CPU idle routine and gets control after the CPU wakes up from idle. It records the actual residency of the CPU in the idle state for use in the computation of the historic idle residency component in the core idle selection.

3.1.4.3.2.3 Core Idle Get/Set Parameter

This routine provides interfaces to user-mode programs that allow them to get and set core idle state parameters such as the latency constant C used in the equation to determine target core idle state.

3.1.4.3.3 Package Idle State Selection

The Intel® Xeon Phi™ coprocessor supports package idle states™ such as Auto-C3 (wherein all cores and other agents on the ring are clock gated), Deeper-C3 (which further reduces the voltage to the package), and Package C6 (which completely shuts off power to the package while keeping card memory in self-refresh). Some of the key differences between the package idle states and the core (CPU) idle states are:

- One of the preconditions for all package idle states is that all the cores be idle.
- Unlike P-states and core idle states, package state entry and exit are controlled by the Intel® Xeon Phi™ coprocessor host driver (except in Intel® Xeon Phi™ coprocessor Auto-C3 where it is possible to enter and exit the idle state without host driver intervention).
- Wake up from package idle states requires an external event such as PCI Express* traffic, external interrupts, or active intervention by the Intel® Xeon Phi™ coprocessor driver.
- Idle residency calculations for the package states take into account the idle residency values of all the cores.
- Since the package idle states cause the Timestamp counter (TSC) and the local APIC timer to freeze, an external reference timer like the SBox Elapsed Time Counter (ETC) on the Intel® Xeon Phi™ coprocessor can be used, on wake up from idle, to synchronize any software timers that are based on the TSC or local APIC.

3.1.4.3.4 The coprocessor OS Role in Package Idle State Selection

The coprocessor OS plays a central role in selecting package idle states. The package idle state selection is facilitated in the coprocessor OS by three main routines:

- package idle select
- package idle update
- get/set package idle parameter

3.1.4.3.4.1 Package Idle Select

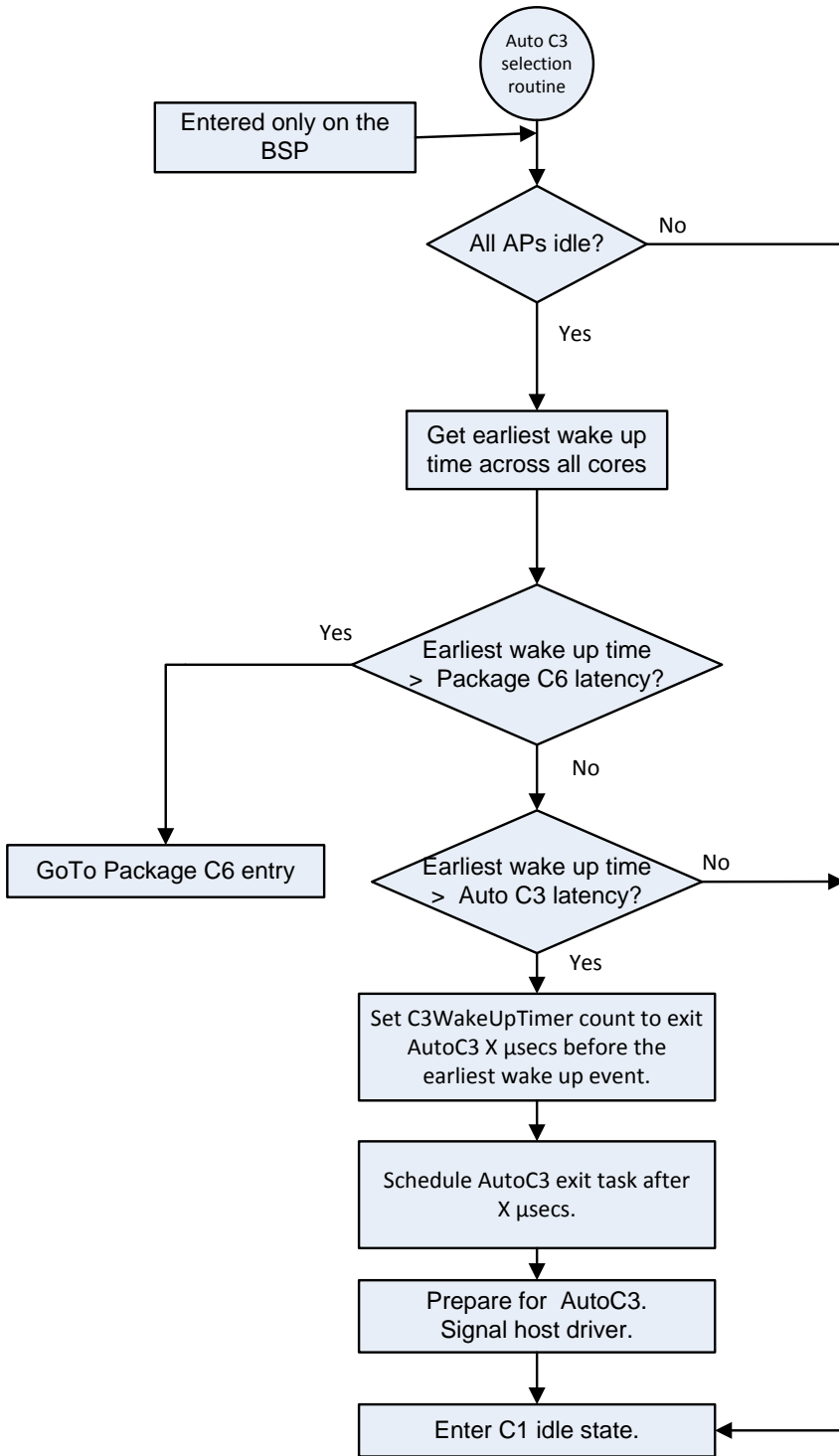
The last CPU that is ready to go idle invokes the package idle-select routine. As with the core idle state selection algorithm, the package idle-select algorithm bases its selection on the expected idle residency of the package and the

latency of the package idle state. The expected idle residency is calculated using the earliest scheduled timer event across all cores and the historical data on package idleness.

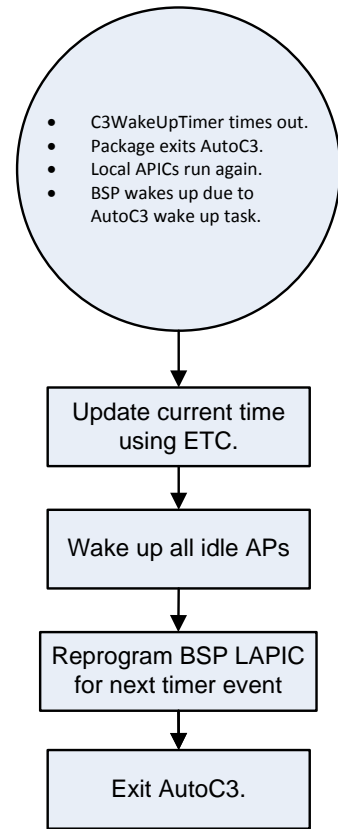
On the Intel® Xeon Phi™ coprocessor, the coprocessor OS selects the PC3 and PC6 package states. Figure 3-4 shows the software flow for package idle-state selection.

While selecting a package idle state, the coprocessor OS PM software can choose to disregard certain scheduled timer events that are set up to accomplish housekeeping tasks in the OS. This ensures that such events do not completely disallow deeper package idle states from consideration. It is also possible for the coprocessor OS package idle-state selection algorithm to choose a deeper idle state (such as PC6), and still require that the package exit the deep idle state in order to service a timer event. In such cases, the coprocessor OS informs the host PM software not only the target package idle-state selected but also the desired wake up time from the idle state.

Auto C3 Entry Algorithm



Auto C3 Exit Algorithm (BSP)



Auto C3 Exit Algorithm (AP)

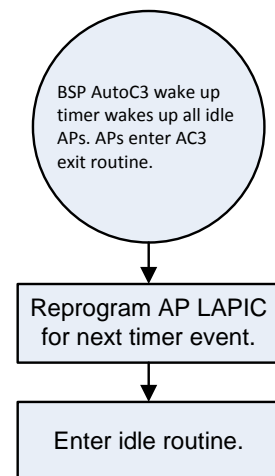


Figure 3-4. Package C-state Selection Flow

3.1.4.3.4.2 Package Idle Update

This routine is invoked upon wake up from a package idle state. It records the actual time that the package was idle, which is then used in the idle residency calculation. Since the TSC and the local APIC timers freeze during a package idle state, this routine uses an external clock (such as the SBox ETC) on Intel® Xeon Phi™ coprocessor cards to measure the package idle time.

3.1.4.3.5 Host Driver Role in Package Idle State Selection

The PM task in the host driver plays a key role in the package idle-state selection process. Though the coprocessor OS selects the package idle state based on its assessment of the expected idle residency, there are other reasons that might cause the host PM task to modify this selection. Some of these are:

- The coprocessor OS selects PC3 based on the expected residency of the cores. However, PC3 depends on the idleness of both the core and the uncore parts of the package. So, it is possible for a PC3 selection by the coprocessor OS to be overridden by the host driver if it determines that some part of the uncore chain is busy.
- If the idle residency estimate by the coprocessor OS for a certain package idle state turns out to be too conservative and the package stays in the selected idle state longer than the estimated time, the host driver can decide to select a deeper idle state than the one chosen by the coprocessor OS.
- Package idle states, such as DeepC3 and PC6 on the Intel® Xeon Phi™ coprocessor, require the active intervention of the host driver to wake up the package so that it can respond to PCI Express* traffic from the host. Therefore, these deeper idle states might be unsuitable in scenarios where the card memory is being accessed directly by a host application that bypasses the host driver. The host driver should detect such situations and override the deeper idle-state selections.

3.1.4.3.6 Coprocessor OS-to-Host Driver Interface for Package Idle Selection

The coprocessor OS and the host driver use two main interfaces to communicate their package idle state selections:

- The coprocessor OS-host communication interface through SCIF message.
- The PM state flags such as the μ OSPMState and hostPMState. In the Intel® Xeon Phi™ coprocessor, these flags are implemented as registers in the MMIO space. The μ OSPMState is written by the coprocessor OS to indicate its state selection, and read by the host driver and vice versa for the hostPMState flag.

The SCIF API and the package idle control API are implemented so as to be hardware independent.

3.1.4.4 Idle State Control

The idle state control function sets the cores (or the package) to the selected idle state. While controlling the core's idle state is primarily handled by the coprocessor OS, controlling the package idle state requires co-ordination between the host driver and the bootstrap software.

3.1.4.4.1 Coprocessor OS Role in Idle State Control

The idle-state control module in the coprocessor OS implements the selected core or package idle state on the target Intel® Xeon Phi™ coprocessor. It hides all the hardware details from the selection module. It initializes the data structures that it shares with the idle-state selection module with information on idle states specific to the Intel® Xeon Phi™ coprocessor. The interface to the selection module is mainly through these data structures. Table 3-1 lists some low-level routines in this module that are common to all idle states.

Table 3-1. Routines Common to All Package Idle States

Routine	Description
Save_CPU_State	Saves the register state of the selected logical processor. The CPU state includes basic program execution registers, x87 FPU registers, control registers, memory management registers, debug registers, memory type range registers (MTRR), and machine specific registers (MSR). The VPU register context is also saved.
Restore_CPU_State	Restores the register state that was saved by the Save_CPU_State routine.
Save_Uncore_State	Saves the Intel® Xeon Phi™ coprocessor hardware states that are not associated with CPUs (e.g. SBox). This function is used to preserve the uncore context in preparation for or during the PC6 entry sequence.
Restore_Uncore_State	Restores the Intel® Xeon Phi™ coprocessor hardware state that was saved by the Save_Uncore_State routine.

3.1.4.4.2 Core Idle State Control in the Coprocessor OS

There are two routines that control the idle state of the core (Core C6): CC6_Enter and CC6_Exit.

3.1.4.4.2.1 CC6_Enter

The CC6_Enter routine starts when Core C6 is selected to prepare the CPU for a CC6 entry. However, if one or more other CPUs either are non-idle or did not enable C6, then the core might not enter the C6 idle state. The return from this routine to the caller (that is, to the CPU idle routine) looks exactly the same as a return from a Core C1 (return from HALT). The only way software using an Intel® Xeon Phi™ coprocessor can figure out that a CPU entered Core C6 is when the CPU exits Core C6 and executes its designated CC6 exit routine. The essential sequence of actions in this routine is as follows:

1. Start CC6_Enter.
2. Reset the count of CPUs that have exited CC6 (only for last CPU in core going idle).
3. Save CR3.
4. Switch page tables to the identity map of the lower 1MB memory region.
5. Run Save_CPU_State.
6. Enable CC6 for the selected CPU.
7. Enable interrupt and HALT.

The real mode trampoline code runs in lower memory (first MB of memory), and the CC6_Enter entry point is an address in this memory range. The idle-state control driver copies the trampoline code to this memory area during its initialization. It is also important to make sure that this memory range is not used by the bootloader program.

3.1.4.4.2.2 CC6_Exit

When cores exit from CC6 (as a result of an interrupt to one or more CPUs in the core), they come back from reset in real mode and start executing code from an entry point that is programmed by the Enable_CC6 routine. The essential sequence of actions in the CC6 exit routine is as follows:

1. Start CC6_Exit.
2. Run trampoline code to set up for 64-bit operation.
3. Detect the CPU number from the APIC identification number.
4. Restore the CPU state.
5. Restore CR3.
6. Increment the count of CPUs in the core that have exited CC6.
7. Enable interrupt and HALT.

As shown in Figure 3-5, it is possible for a CPU to exit CC6 while remaining HALTED and to go back to CC6 when the CC6 conditions are met again. If a CPU stays HALTED between entry and exit from CC6, it is not required that the CPU state be saved every time it transitions to CC6.

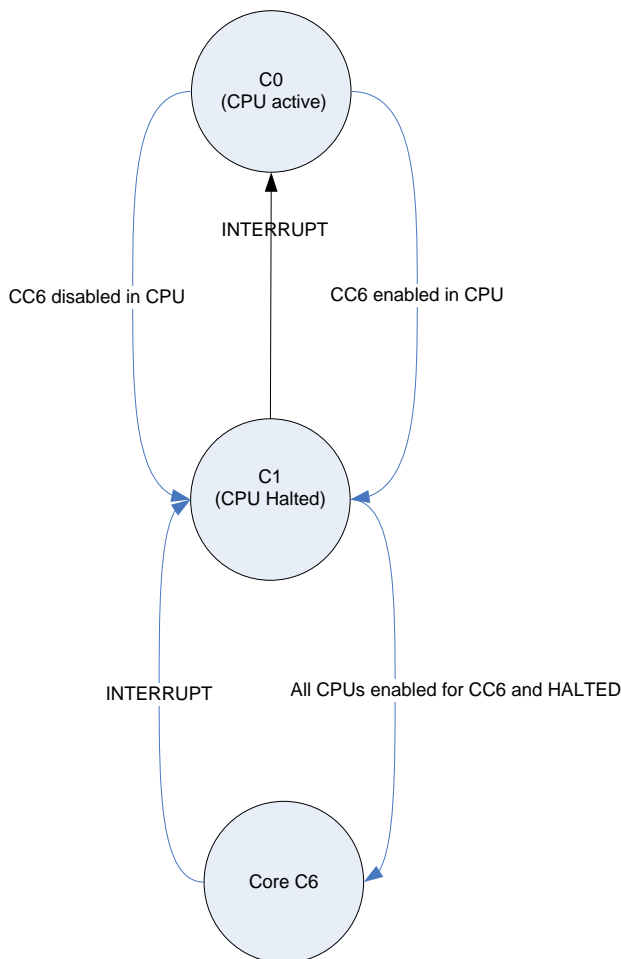


Figure 3-5 CPU Idle State Transitions

3.1.4.4.3 Package Idle State Control

Table 3-2 Package Idle State Behavior in the Intel® Xeon Phi™ Coprocessor

Package Idle State	Core State	Uncore State	TSC/LAPIC	C3WakeupTimer	PCI Express* Traffic
PC3	Preserved	Preserved	Frozen	On expiration, package exits PC3	Package exits PC3
Deep C3	Preserved	Preserved	Frozen	No effect	Times out
PC6	Lost	Lost	Reset	No effect	Time out

As shown in Table 3-2, the package idle states behave differently in ways that impact the PM software running both on the card as well as on the host. The idle-state control driver handles the following key architectural issues:

- LAPIC behavior:** The LAPIC timer stops counting forward when the package is in any idle state. Modern operating systems support software timers (like the POSIX timer) that enable application and system programs to schedule execution in terms of microseconds or ticks from the current time. On the Intel® Xeon Phi™ coprocessor, due to the absence of platform hardware timers, the LAPIC timer is used to schedule timer interrupts that wake up the CPU to

service the software timer requests. When the LAPIC timer stops making forward progress during package idle states, timer interrupts from the LAPIC are suspended. So, the software timers cannot be serviced when the package is in an idle state. In order for the operating system to honor such software timer requests, the package idle state control software enlists the services of hardware timers, such as the C3WakeupTimer in the Intel® Xeon Phi™ coprocessor, or the host driver to wake up the card in time to service the scheduled timers.

- **TSC behavior:** On the Intel® Xeon Phi™ coprocessor, the TSC is used as the main clock source to maintain a running clock of ticks in the system. When the TSC freezes during package idle states, the software must be able to rely on an external reference clock to resynchronize the TSC based clock upon exit from the package idle state. On the Intel® Xeon Phi™ coprocessor, the SBox Elapsed Time Counter can be used for this purpose.
- **Effect of PCI Express* traffic:** While PCI Express* traffic brings the card out of a Package C3 idle state, it does not do so for deeper idle states such as DeepC3 or PC6. Also, the transition to DeepC3 or PC6 from PC3 does not happen automatically but requires active intervention from host software. Consequently, when the host driver places the card in one of these deep package idle states, it has to ensure that all subsequent PCI Express* traffic to the card be directed through the host driver. This makes it possible for the host driver to bring the card out of one of these deeper package idle states so that the card can respond to the subsequent PCI Express* traffic.
- **Core and uncore states:** While the core and uncore states are preserved across PC3 and DeeperC3 idle states entry and exit, they are not preserved for PC6. So, when the host driver transitions the package to PC6 from PC3 or DeepC3, it has to wake up the card and give the coprocessor OS a chance to save the CPU state as well as to flush the L2 cache before it puts the package in PC6 idle state.

Package idle state control is implemented both in the coprocessor OS and in the host driver.

3.1.4.4.3.1 Package Idle State Control in the Coprocessor OS

The coprocessor OS role in package idle-state control is limited to the PC3 and PC6 idle states. DeepPC3 is controlled by the host driver, and the coprocessor OS has no knowledge of it. Coprocessor OS package idle state control mainly consists of the following activities:

- Prepare the coprocessor OS and the hardware to wake up from idle state in order to service timer interrupts.
- Save the core/uncore state and flush L2 cache, when necessary.
- On exit from package idle state reprogram LAPIC timers and synchronize timekeeping using an external reference clock such as the ETC on the Intel® Xeon Phi™ coprocessor.
- Send and receive messages to the host driver, and update the μ OSPMstate flag with package idle state as seen from the coprocessor OS.

3.1.4.4.3.2 PC3_Entry

This function handles the package C3 idle state entry. As shown in Figure 3-6, this function is called from the core idle-state control entry function of the last CPU in the system to go idle. The core idle-selection module selects the package idle state in addition to the CPU idle state for the last CPU going idle and calls the core idle-state control entry function. The sequence of actions this function executes is:

1. Start PC3_Entry.
2. The last CPU going idle sets up the C3WakeupTimer so that the package will exit PC3 in time to service the earliest scheduled timer event across all CPUs.
3. Record current tick count and reference clock (ETC) time.
4. Set μ OSPMState flag to PC3.
5. Send message to host driver with target state and wake up time.
6. CPU HALTS.

There might be conditions under which the time interval to the earliest scheduled timer event for the package is larger than what can be programmed into the C3WakeupTimer. In such cases the coprocessor OS relies on the host driver to

wake up the package. The package idle-state readiness message that the coprocessor OS sends to the host PM software could optionally include wake up time. The host driver will wake up the package at the requested time.

3.1.4.4.3.3 PC3_Exit

An exit from the package C3 idle state happens when the C3WakeupTimer expires and exits from PC3 or when PCI Express* traffic arrives and causes the package to exit PC3. Figure 3-6 illustrates the former. It is important to remember that in either case, when the package exits PC3, it triggers the GoalReached interrupt when the core frequency reaches the set value. One possible sequence of events that can happen in this case is as follows:

1. The C3WakeupTimer expires and the package exits PC3.
2. The GoalReached interrupt wakes up BSP.
3. The BSP processes PC3 exit.

Although the package is set up for PC3 and all the CPUs are HALTED, there is no guarantee that the package actually transitioned to PC3 idle. So, any CPU that wakes up after PC3_Entry is executed, must check to make sure that a transition to PC3 idle did indeed take place. One way that this can be done is through the hostPMState flag that is set by the host when it confirms that the package is in PC3 idle.

The sequence of steps taken by the PC3_Exit routine is as follows:

1. Start PC3_Exit.
2. Check the hostPMState flag to confirm transition to PC3.
3. If the hostPMState flag is not set, then set the μ OSPMState flag to PC0.
4. Send UOS_PM_PC3_ABORT message to the host driver.
5. Return .
6. Read the ETC and calculate package residency in AutoC3.
7. Update kernel time counters.
8. Send AutoC3_wakeup IPI to all APs.
9. Reprogram the Boot Strap Processor (BSP) LAPIC timer for earliest timer event on BSP.
10. Set the μ OSPMState flag to PC0.
11. Send UOS_PM_PC3_WAKEUP message to the host driver.
12. Return.

The sequence of steps taken by the AC3_wakeup_IPI_handler (on all Application Processors (APs)) is:

1. Reprogram LAPIC timer for earliest timer event on CPU
2. Return

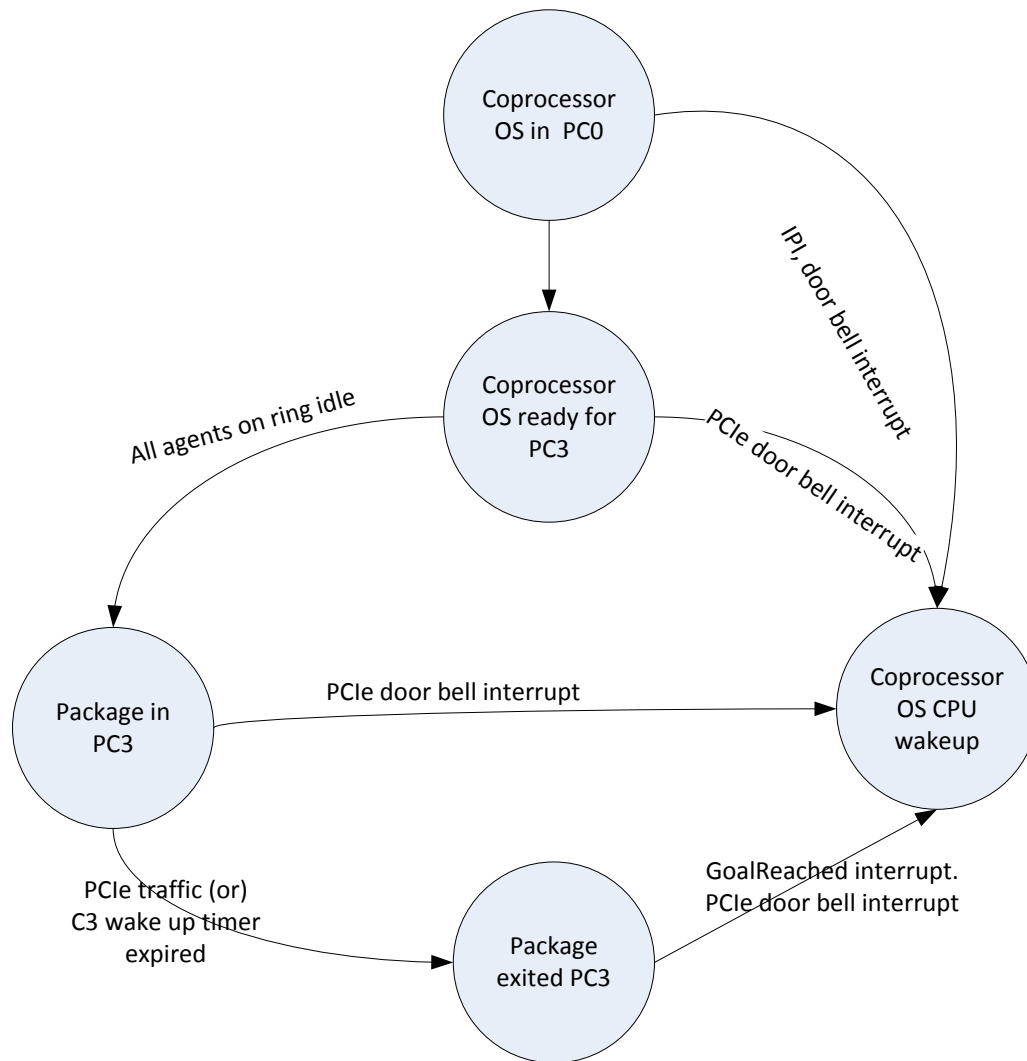


Figure 3-6. Package C-state Transitions

3.1.4.4.3.4 PC6_Entry

The coprocessor OS runs the PC6_Entry routine either when the coprocessor OS idle-state selection module selects PC6 as the target package idle state or when the host PM software decides that the package has been in PC3 long enough to warrant a deeper idle state like PC6. In the latter case, the host software sends a PC6_Request message to the coprocessor OS that invokes the PC6_Entry routine. Architecturally, the PC6 idle state is similar to the APCI S3 suspend state, wherein the memory is in self refresh while the rest of the package is powered down. The sequence of actions this routine executes consists of:

1. PC6_Entry (on BSP)
2. Save CR3.
3. Switch page tables (to identity map for lower 1MB memory region).
4. Send C6_Entry IPI to all APs.
5. Wait for APs to finish PC6 Entry preparation.
6. Save uncore context to memory.
7. Record the ETC value and current tick count.
8. Save BSP context to memory.
9. Flush cache.
10. Set the μ OSPMState flag to PC6.

11. Send PC6 ready message to host.
12. HALT BootStrap Processor (BSP).

Or

13. PC6 Entry (on AP).
14. Save CR3.
15. Switch page tables (to identity map for lower 1MB memory region).
16. Save AP context to memory.
17. Set flag to mark PC6 Entry completion.
18. Flush cache.
19. HALT AP.

The PC6 entry implementation takes advantage of the fact that when the PC6 selection is made, it is more than likely that most of the cores are already in Core C6, and therefore have already saved the CPU context. If the L2 cache is flushed before the last CPU in every core prepares to go to Core C6, then the PC6 Entry algorithm might not need to wake up CPUs (from core C6) only to flush the cache. This reduces the PC6 entry latencies and simplifies the design, but the cost of doing a L2 cache flush every time a core is ready for CC6 has to be factored in.

3.1.4.4.3.5 PC6 Exit

The host driver PM software is responsible for bringing the package out of a PC6 idle state when the host software attempts to communicate with the card. The implicit assumption in any host-initiated package idle-state exit is that after the card enters a deep idle state, any further communication with the card has to be mediated through the host PM software. Alternatively, the host PM software can bring the card out of a package idle state if the coprocessor OS on the card has requested (as part of its idle entry process) that it be awakened after a certain time interval.

The sequence of actions this routine executes consists of:

1. PC6_Exit (BSP).
2. Begin BSP execution from the reset vector because of the VccP transition from 0 to minimum operational voltage and the enabling of MCLK.
3. BootLoader determines that this is a PC6 Exit (as opposed to a cold reset).
4. BootLoader begins execution of specific PC6_Exit sequence.
5. Bootstrap passes control to _PC6_Exit_ entry point in GDDR resident coprocessor OS.
6. BSP restores processor context.
7. BSP restores uncore context.
8. BSP reads the SBox ETC and updates kernel time counters.
9. BSP wakes up APs.
10. BSP sets μ OSPMState to PC0.
11. BSP sends coprocessor OS_Ready message to host driver .

Or

1. PC6_Exit (AP).
2. AP begins execution of trampoline code and switches to 64 bit mode.
3. AP restores processor state.
4. Signals PC6_Exit complete to BSP.

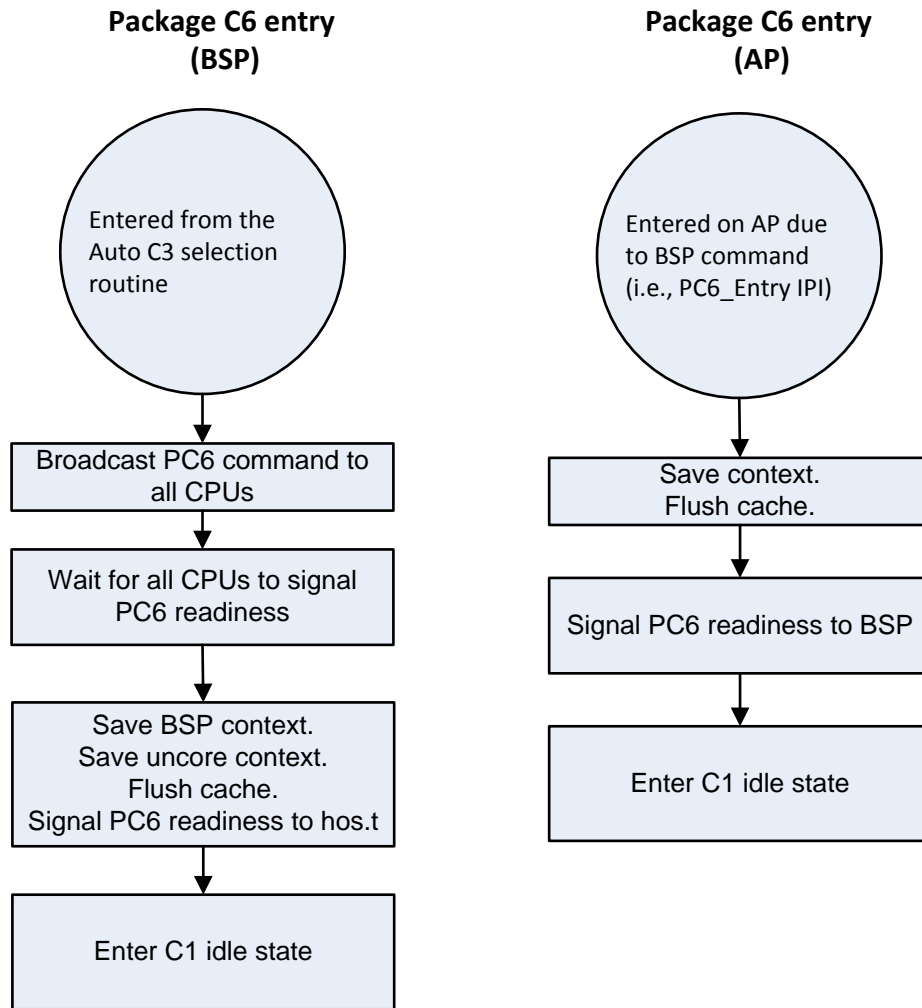


Figure 3-7 Package C6 Entry and Exit Flow

3.1.4.4.3.6 Bootloader Role in Idle State Control

The bootloader program co-ordinates the exit from PC6 as well as facilitating the waking up of cores from CC6. The Bootloader interfaces with both the coprocessor OS and the host Intel® MPSS driver to enable these transitions. The main interfaces are:

- Interface to reserve memory in the first megabyte of GDDR to install Core C6 wake up code
- Interface with host Intel® MPSS driver to obtain PC6 entry point into the coprocessor OS kernel.
- Interface with the host Intel® MPSS driver to detect a PC6 exit as against a cold reset.

One Intel® Xeon Phi™ coprocessor implementation option is for the host Intel® MPSS driver to send the PC6 exit entry point as part of a BootParam structure that is located in a region of GDDR memory at a well-known address between the host Intel® MPSS driver and the Bootloader.

The hostPMState MMIO register could be used by the Bootloader to distinguish a PC6 exit from cold reset.

Every Intel® Xeon Phi™ coprocessor core has a block of registers that is initialized by the Bootloader, and then locked against subsequent write access for security reasons. However, since these register contents are lost during CC6, the

Intel® Xeon Phi™ coprocessor reserves a block of SBox MMIO registers that are used to maintain a copy of these secure register contents. It is the Bootloader's responsibility to initialize this block with the contents of the control registers during the boot up process. Subsequently, when a core wakes up from CC6, the µcode copies the contents of the SBox register block back into the core registers.

3.1.5 PM Software Event Handling Function

One of the key roles for the Intel® MIC Architecture PM software is the handling of power and thermal events and conditions that occur during the operation of the Intel® Xeon Phi™ coprocessor. These events and conditions are handled primarily by the coprocessor OS PM Event Handler module. The number and priority of these events are hardware dependent and implementation specific. However, these events fall into two basic categories: proactive and reactive.

For example, the Intel® Xeon Phi™ coprocessor has the ability to notify the coprocessor OS when the die temperature exceeds programmed thresholds, which allows the software to act proactively. On the other hand, the coprocessor OS software acts reactively when an OverThermal condition occurs in the die by automatically throttling the core frequency to a predetermined lower value and interrupting the CPU.

Table 2-1 lists the events and conditions that the coprocessor OS should handle for the Intel® Xeon Phi™ coprocessor, their source, indications, and suggested software response.

Table 3-3. Events and Conditions Handled by the Coprocessor OS

Event or Condition	Source	Indication	Suggested Coprocessor OS Action	Remarks
CPUHOT	Raised either by the sensors in the die, the VR, or the SMC	TMU interrupt and MMIO status register	Hardware automatically throttles core frequency to a low value. Coprocessor OS resets its P-state evaluation algorithm, programs frequency and voltage to correspond to configurable values and enables the GoalReached interrupt.	When the hardware exits the CPUHOT condition, it locks on to the frequency programmed by the coprocessor OS, and raises the GoalReached interrupt. Coprocessor OS restarts the P-state evaluation algorithm.
SW Thermal threshold 1 crossed on the way up.	TMU	TMU interrupt and MMIO status register	Coprocessor OS sets max P-state to P1. The new max P-state takes effect during the next P-state selection pass.	
SW Thermal threshold 2 crossed on the way up.	TMU	TMU interrupt and MMIO status register	Coprocessor OS sets max P-state to a configurable value between P1 and Pn. Affects P-state change immediately.	
SW Thermal threshold 1 crossed on the way down.	TMU	TMU interrupt and MMIO status register	Coprocessor OS sets max P-state to P0 (turbo). The new max P-state takes effect during the next P-state selection pass.	
PWRLIMIT	SMC	I2C interrupt	Coprocessor OS reads SMC power limit value and sets low and high water mark thresholds for power limit alerting.	SMC will interrupt the coprocessor OS when it has a new power limit setting from the platform.
PWRALERT	SMC	TMU interrupt, MMIO status register	Raised when the card power consumption crosses either the low or the high threshold set by the coprocessor OS. The coprocessor OS adjusts P-state accordingly.	
Over current limit	SVID		Coprocessor OS P-state evaluation algorithm reads SVID current output and compares it to preset limits for modifying the P-state.	
Fan speed	SMC	MMIO register	Coprocessor OS P-state evaluation algorithm reads fan speed and compares it to preset limits for modifying the P-state.	

3.1.6 Power Management in the Intel® MPSS Host Driver

The host driver power management (PM) component is responsible for performing PM activities in cooperation with the coprocessor OS on an Intel® Xeon Phi™ coprocessor. These activities are performed after receiving events or notifications from the control panel, the coprocessor OS, or the host operating system. The PM component in the host driver and the PM component in the coprocessor OS communicate using the SCIF.

The Power Management for the host driver falls into four functional categories:

- Control panel (Ring3 module) interface
- Host OS power management
- Host-to-coprocessor OS communication and commands
- Package states handling

3.1.6.1 PM Interface to the Control Panel

The Host driver implements services to collect user inputs. It is an interface (e.g., Sysfs on Linux*) by which the control panel reads PM status variables such as core frequency, VID, number of idle CPUs, power consumption, etc. The interface can also be used by other PM tools and monitoring applications to set or get PM variables.

3.1.6.2 Host OS Power Management

Power management works on two levels. It can be applied to the system as a whole or to individual devices. The operating system provides a power management interface to drivers in the form of entry points, support routines, and I/O requests. The Intel® MPSS host drivers conform to operating system requirements and cooperate to manage power for its devices. This allows the operating system to manage power events on a system wide. For example, when the OS sets the system to state S3; it relies upon the Intel® MPSS host driver to put the device in the corresponding device power state (D-state) and to return to the working state in a predictable fashion. Even if the Intel® MPSS host driver can manage the Intel® Xeon Phi™ coprocessor's sleep and wake cycles, it uses the operating system's power management capabilities to put the system as a whole into a sleep state.

The Intel® MPSS host driver interfaces with the host operating system for power management by doing the following:

- Reporting device power capabilities during PnP enumeration.
- Handling power I/O requests sent by the host OS or by another driver in the device stack (applicable to Windows environment).
- Powering up the Intel® Xeon Phi™ coprocessor(s) as soon as it is needed after system startup or idle shutdown.
- Powering down the Intel® Xeon Phi™ coprocessor at system at shutdown or putting system to sleep when idle.

Most of the power management operations are associated with installing and removing Intel® Xeon Phi™ coprocessors. Hence, the Intel® MPSS host driver supports Plug and Play (PnP) to get power-management notifications.

3.1.6.2.1 Power Policies (applicable to Windows)

You can use the Windows control panel to set system power options. The Intel® MPSS host driver registers a callback routine with the operating system to receive notification. As soon as a callback is registered by the driver during load, the OS immediately calls the callback routine and passes the current value of the power policy. Later, the OS notifies the host driver of the changes to the active power policy that were made through this callback. The driver then forwards the policy change request and associated power settings to the coprocessor OS.

3.1.6.3 PM Communication with the coprocessor OS

A set of commands specifically for power management facilitate communication between the host driver and the coprocessor OS. These commands initiate specific PM functions or tasks, and coordinate the exchange of PM information.

The Intel® MPSS host driver uses the symmetric communication interface (SCIF) layer to create a channel to send messages to the coprocessor OS PM component. SCIF provides networking and communication capabilities within a single platform. In the SCIF context, the host driver and the coprocessor OS PM components are on different SCIF nodes. The Intel® MPSS host driver creates a Ring0-to-Ring0 communication queue from its own node to a “known” SCIF port (logical destination) on the coprocessor OS node. The message types are summarized in Table 3-4.

Table 3-4. Power Management Messages

Message Type	Description
Status queries	Messages passed to inquire about the current PM status; for example, core voltage, frequency, power budget, etc. Most of this data is supplied to the control panel.
Policy control	Messages that control PM policies in the coprocessor OS. For example, enable/disable turbo, enable/disable idle package states, etc.
Package state commands	Messages used to monitor and handle package states. For example, get/set vccp, get entry/exit latencies, etc.
Notifications from the coprocessor OS	The coprocessor OS notifies the host driver when it is going to enter an idle state because all the cores are idle.

3.1.6.4 Package States (PC States) Handling

One of the main PM responsibilities of the Intel® MPSS host driver is to monitor idle states. The host driver monitors the amount of time that the coprocessor OS spends idle and makes decisions based on the timer’s expiration. When all the CPUs in the Intel® Xeon Phi™ coprocessors are in core state (C1), the coprocessor OS notifies the host driver that the devices are ready to enter package sleep states. At this stage, the coprocessor OS goes to auto PC3 state. The coprocessor OS, on its own, cannot select the deeper idle states (deep PC3 and PC6). It is the responsibility of the host driver to request that the coprocessor OS enter a deeper idle state when it believes that the coprocessor OS has spent enough idle time in the current idle state (PC6 is the deepest possible idle state).

3.1.6.4.1 Power Control State Entry and Exit Sequences

This section summarizes the steps followed when the package enters the PC3 or the PC6 idle state.

_PC3_auto Entry_:

1. Receive idle state notification for auto PC3 entry from coprocessor OS.
2. Wait for Intel® Xeon Phi™ coprocessor Idle/Resume flag = PC3 code.
3. Verify hardware idle status.
4. Set HOST Idle/Resume flag = auto PC3 code.
5. Start host driver timer for auto PC3 state.

_PC3_deep Entry_:

1. Make sure that the host driver auto PC3 timer has expired.
2. Verify hardware idle status.
3. Set VccP to minimum the retention voltage value.
4. Set HOST Idle/Resume flag = deep PC3 code.
5. Start the host driver timer for PC6 state.

_PC6_Entry_:

1. Make sure that the host driver PC6 timer has expired.
2. Executethw _PC3_deep_Exit_ algorithm.
3. Request that the coprocessor OS to enter PC6 state.
4. Receive readiness notification for PC6 entry from the coprocessor OS.
5. Wait for Intel® Xeon Phi™ coprocessor Idle/Resume flag = PC6 code.
6. Verify hardware idle status.
7. Set VccP to zero (0) volts.
8. Set HOST Idle/Resume flag = PC6 code.

_PC3_deep Exit_:

1. Set VccP to the minimum operating voltage.
2. Wait for Intel® Xeon Phi™ coprocessor Idle/Resume flag = C0 code.
3. Set HOST Idle/Resume flag = C0 code.

_PC6_Exit_:

1. Set VccP to the minimum operating voltage.
2. Wait for LRB Idle/Resume flag = C0 code.
3. Set HOST Idle/Resume flag = C0 code.

3.1.6.4.2 Package State Handling and SCIF

SCIF is the interface used for communication between the host software and the coprocessor OS software running on one or more Intel® Xeon Phi™ coprocessors. SCIF is also used for peer-to-peer communication between Intel® Xeon Phi™ coprocessors. This interface could potentially (for speed and efficiency reasons) be based on a distributed shared memory architecture where peer entities on the host and the Intel® Xeon Phi™ coprocessor share messages by directly writing to each other's local memory (Remote Memory Access). The host driver takes into account the SCIF communication channels that are open on an Intel® Xeon Phi™ coprocessor when deciding to put it into a deeper package idle state.

3.1.6.4.3 Boot Loader to Host Driver Power Management Interface

The boot loader executes when power is first applied to the device, but can also run when exiting from PC6 idle states due to the removal of the VccP power rail. The boot-loader component for Intel® Xeon Phi™ coprocessors has a PM-aware abbreviated execution path designed specifically for exiting D3 and PC6 states, minimizing the time required to return the Intel® Xeon Phi™ coprocessor to full operation from D3 and PC6. To support PC6 exit, the host driver interacts with the boot loader via the scratchpad registers.

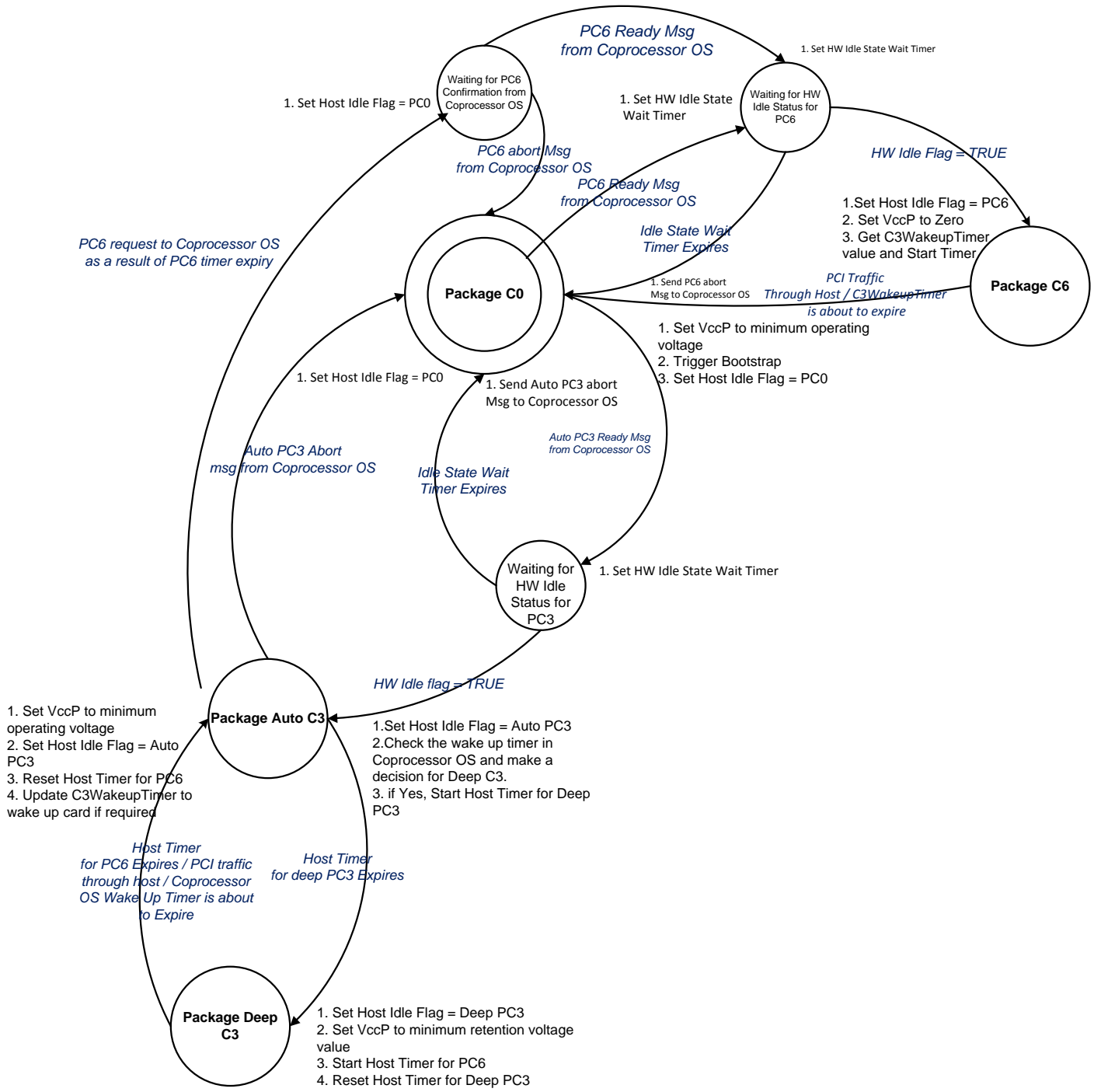


Figure 3-8 Intel® MPSS Host Driver to Coprocessor OS Package State Interactions

3.2 Virtualization

A platform that supports virtualization typically has a Virtual Machine Manager (VMM) that hosts multiple Virtual Machines. Each virtual machine runs an OS (Guest OS) and application software. Different models exist for supporting I/O devices in virtualized environments, and the Intel® Xeon Phi™ coprocessor supports the direct assignment model wherein the VMM directly assigns the Intel® Xeon Phi™ coprocessor device to a particular VM and the driver within the VM has full control with minimal intervention from the VMM. The coprocessor OS does not require any modifications to support this model; however, the chipset and VMM are required to support the following Intel VT-d (Intel Virtualization Technology for Direct I/O) features:

- Hardware-assisted DMA remapping
- Hardware-assisted interrupt remapping
- Shared device virtualization

3.2.1 Hardware Assisted DMA Remapping

In virtualized environments, guests have their own view of physical memory (guest physical addresses) that is distinct from the host's physical view of memory. The guest OS Intel® Xeon Phi™ coprocessor device driver (and thus the coprocessor OS on the Intel® Xeon Phi™ coprocessor dedicated to the guest) only knows about guest physical addresses that must be translated to host physical addresses before any system memory access. Intel VT-d (implemented in the chipset) supports this translation for transactions that are initiated by an I/O device in a manner that is transparent to the I/O device (i.e., the Intel® Xeon Phi™ coprocessor). It is the VMM's responsibility to configure the VT-d hardware in the chipset with the mappings from guest physical to host physical addresses when creating the VM. For details refer to the Intel VT for Direct I/O Specification (Intel® Virtualization Technology for Directed I/O, 2011).

3.2.2 Hardware Assisted Interrupt Remapping

In a virtualized environment with direct access, it is the guest and not the host VMM that should handle an interrupt from an I/O device. Without hardware support, interrupts would have to be routed to the host VMM first which then injects the interrupt into the guest OS. Intel VT-d provides Interrupt remapping support in the chipset which the VMM can use to route interrupts (either I/O APIC generated or MSIs) from specific devices to guest VMs. For details refer to the Intel VT for Direct I/O specification.

3.2.3 Shared Device Virtualization

Each card in the system can be either dedicated to a guest OS or shared among multiple guest operating systems. This option requires the highest level of support in the coprocessor OS as it can service multiple host operating systems simultaneously.

3.3 Reliability Availability Serviceability (RAS)

RAS stands for reliability, availability, and serviceability. Specifically, *reliability* is defined as the ability of the system to perform its actions correctly. *Availability* is the ability of the system to perform useful work. *Serviceability* is the ability of the system to be repaired when failures occur. Given that HPC computing tasks may require large amounts of resources both in processing power (count of processing entities or *nodes*) and in processing time, node reliability becomes a limiting factor if not addressed by RAS strategies and policies. This section covers RAS strategies available in software on Intel® Xeon Phi™ coprocessor and its host-side server.

In HPC compute clusters, reliability and availability are traditionally handled in a two-pronged approach: by deploying hardware with advanced RAS features to reduce error rates (as exemplified in the Intel® Xeon® processors) and by adapting fault tolerance in high-end system software or hardware. Common software-based methods of fault tolerance

are to deploy redundant cluster nodes or to implement snapshot and restore (check pointing) mechanisms that allow a cluster manager to reduce data loss when a compute node fails by setting it to the state of last successful snapshot. Fault tolerance, in this context, is about resuming from a failure with as much of the machine state intact as possible. It does not imply that a cluster or individual compute nodes can absorb or handle failures without interrupting the task at hand.

The Intel® Xeon Phi™ coprocessor addresses reliability and availability the same two ways. Hardware features have been added that improve reliability; for example, ECC on GDDR and internal memory arrays that reduce error rates. Fault tolerance on Intel® Xeon Phi™ coprocessor hardware improves failure detection (extended machine check architecture, or MCA). Managed properly, the result is a controlled and limited degradation allowing a node to stay in service after certain anticipated hardware failure modes manifest themselves. Fault tolerance in Intel® Xeon Phi™ coprocessor software is assisted by the Linux* coprocessor OS, which supports application-level snapshot and restore features that are based on BLCR (Berkeley Labs Checkpoint Restart).

Intel® Xeon Phi™ coprocessor approach to serviceability is through software redundancy (that is, node management removes failing compute nodes from the cluster), and has no true hardware redundancy. Instead software and firmware features allow a compute node to reenter operation after failures at reduced capacity until the card can be replaced. The rationale behind this ‘graceful’ degradation strategy is the assumption that an Intel® Xeon Phi™ coprocessor unit with, say one less core, will be able to resume application snapshots and therefore is a better proposition to the cluster than removing the node entirely.

A hardware failure requires the failing card to be temporarily removed from the compute cluster it is participating in. After a reboot, the card may rejoin the cluster if cluster management policies allow for it.

The Intel® Xeon Phi™ coprocessor implements extended machine check architecture (MCA) features that allow software to detect and act on detected hardware failures in a manner allowing a ‘graceful’ degradation of service when certain components fail. Intel® Xeon Phi™ coprocessor hardware reads bits from programmable FLASH at boot time, which may disable processor cores, cache lines, and tag directories that the MCA has reported as failing.

3.3.1 Check Pointing

In the context of RAS, check pointing is a mechanism to add fault tolerance to a system by saving its state at certain intervals during execution of a task. If a non-recoverable error occurs on that system, the task can be resumed from the last saved checkpoint, thereby reducing the loss caused by the failure to the work done since the last checkpoint. In HPC, the system is the entire *cluster*, which is defined as all the compute nodes participating in a given HPC application. Cluster management controls where and when checkpoints occur and locks down its compute nodes prior to the checkpoint. The usual mode of operation is for checkpoints to occur at regular intervals or if system monitoring determines that reinstating a checkpoint is the proper course of action. Individual compute nodes are responsible for handling local checkpoint and restore (C/R) events, which have to be coordinated in order to establish a cluster-wide coherent C/R. Conceptually check pointing can be handled in two ways:

- a checkpoint contains the state of the entire compute node, which includes all applications running on it (similar to hibernate)
- or a checkpoint contains the state of a single program running on the compute node, which is referred to as system or application checkpoints.

Application check pointing is by far the most widespread method; it is simpler to implement, produces smaller snapshot images, and may have uses beyond fault tolerance, such as task migration (create snapshot of one system, terminate the application, and restart it on another system) and gang scheduling. These alternate uses are limited to cluster nodes running the same OS and running on similar hardware. System checkpoints are, for all practical purposes, locked to the system it was taken on.

The remainder of this section addresses the basics of BLCR and its integration into the Intel® Xeon Phi™ coprocessor. BLCR details are available at the following links:

- <http://crd.lbl.gov/~jcduell/papers/blcr.pdf>
- <https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html#batch>
- https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html
- https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html

3.3.2 Berkeley Labs Check point and Restore (BLCR)

Due to the altered ABI required for the Linux* coprocessor OS, BLCR is recompiled specifically for the Intel® Xeon Phi™ coprocessor, but otherwise no changes are required for BLCR except for the kernel module. The kernel module incorporates additional process states provided by Intel® Xeon Phi™ coprocessor hardware (the vector registers).

Beyond the enhanced register set, the BLCR kernel module is not different. A patch set for BLCR version 0.8.2 (the latest) exists for the Linux* kernel 2.6.34 and has been shown to build correctly on a standard Linux* system.

BLCR software is, by design, limited to creating a checkpoint for a process (or process group) running under a single operating system. In larger clusters, where the compute workload is spread over several cooperating systems, a checkpoint of a single process does not result in any fault tolerance because the state of that process would soon be out of synchronization with the rest of the cluster (due to inter process messaging). Therefore, a checkpoint within a cluster must be coordinated carefully; e.g., by creating checkpoints of all participants in compute task simultaneously during a lock-down of interprocess communications. Cluster management software must support C/R and implement a method either for putting all participants into a quiescent state during the checkpoint (and to restore all if a participant fails to create one) or for providing a protocol to put each node into a restorable state before the checkpoint occurs.

MPI stacks supporting BLCR have built-in protocols to shut down the IPC between compute nodes and to request a checkpoint to be created on all participants of a 'job'.

Locally, BLCR offers either a cooperative approach or a non-cooperative approach for very simple applications. With the cooperative approach, the application is notified before and after a checkpoint is created. The cooperative approach is intended to give checkpoint-aware applications a way to save the state of features known not to be preserved across a C/R event. The design of BLCR deliberately leaves out the handling of process states that cannot be implemented well (to avoid instability), such as TCP/IP sockets, System-V IPC, and asynchronous I/O. If any of these features are used by the application, they must be brought into a state that allows the application to recreate them after a restore event.

BLCR relies on kernel-assisted (kernel module required) methods to retrieve a useful process state. A BLCR library must be linked to the application in order to establish communication between the application and the kernel module, and to run a private thread within the application that handles call-outs before and after C/R events.

An application process gets notification from the BLCR kernel module through a real time signal so that it can protect its critical regions by registering callbacks to clean house before the checkpoint data is written to file. Upon restart, the same callbacks allow the process to restore internal settings before resuming operations.

The result of a BLCR checkpoint is an image file containing all process state information necessary to restart it. A checkpoint image can be quite large, potentially as large as the node's available memory (swap plus RAM). The Intel® Xeon Phi™ coprocessor does not have local persistent storage to hold checkpoint images, which means they must be shipped to the host (or possibly beyond) over a networked file system to a disk device.

Analysis of BLCR implementations shows that I/O to the disk device is the most time consuming part of check pointing. Assuming the checkpoint images go to the local host's file system, the choice of file system and disk subsystem on the

host become the key factors on checkpoint performance. Alternatives to spinning disks must be considered carefully, though it does not impact the C/R capability and is outside the scope of BLCR.

The BLCR package provides three application programs and a library (plus includes) for building check pointing applications. The BLCR library contains local threads that allow the application some control over when a checkpoint can take place. A simple API lets parts of the application prepare for a checkpoint independently. The mechanism is to register functions like the following with the BLCR library during process initialization:

```
Void my_callback(void *data_ptr)
{
    struct my_data *pdata = (struct my_data*) data_ptr;
    int did_restart;
    // do checkpoint-time shutdown logic
    // tell system to do the checkpoint
    did_restart = cr_checkpoint();
    if (did_restart)
        // we've been restarted from a checkpoint
    else
        // we're continuing after being backed up
}
```

The local BLCR thread calls all registered callbacks before the kernel module checkpoints the application from a local thread. Once all callbacks have called with `cr_checkpoint()`, the local BLCR thread signals the kernel module to proceed with the checkpoint. After the checkpoint, `cr_checkpoint()` returns to the callback routines with information on whether a restart or checkpoint took place.

3.3.2.1 BLCR and SCIF

SCIF is a new feature in the Linux* based coprocessor OS and so has no support in the current BLCR implementation. SCIF has many features in common with sockets. Therefore, BLCR handling of open SCIF connections is treated the same way as open sockets; that is, not preserved across C/R events.

The problem area for sockets is that part of the socket state might come from data present only in the kernel's network stack at the time of checkpoint. It is not feasible for the BLCR kernel module to retrieve this data and stuff it back during a later restore.

The problems for SCIF are the distribution of data in the queue pair and the heavy use of references to physical addresses in the PCI-Express* domain. It is not feasible to rely on physical locations of queue pairs being consistent across a Linux* coprocessor OS reboot, and SCIF is not designed to be informed of the location of queue pairs.

3.3.2.2 Miscellaneous Options

Some aspects of BLCR on the Intel® Xeon Phi™ coprocessor are linked to the applied usage model. In the Intel® MIC Architecture coprocessing mode, this requires a decision as to what a checkpoint covers. In this mode, only the host participates (by definition) as a node in a compute cluster. If it is compatible with compute clusters and C/R is used within the cluster, then only the host can be asked to create a checkpoint. The host must act as a proxy and delegate BLCR checkpoints to the Intel® Xeon Phi™ coprocessor cards as appropriate and manage the checkpoint images from Intel® Xeon Phi™ coprocessors in parallel with its own checkpoint file.

Another, and less complicated approach, is to terminate tasks on all Intel® Xeon Phi™ coprocessors before creating a check point on itself. The tradeoff is between complexities vs. compute time to be redone, depending on the average task length, as part of resuming from check pointing.

Intel® Xeon Phi™ coprocessors used in an Intel® Xeon® offload or autonomous mode do not face this problem because each card is known to the cluster manager that dispatches check point requests to cards individually. The host is a shared resource to the Intel® Xeon Phi™ coprocessors and is not likely to be part of the check pointing mechanism.

Check pointing speed has been identified as a potential problem, mostly because the kernel module that performs the bulk of the state dump is single threaded. Work has been done in the MPI community to speed this up, but the bottleneck appears to be the disk driver and disk I/O, not the single threading itself. Several references point to PCI-Express*-based battery backed memory cards or to PCI-Express*-based Solid State Drive (SSD) disks as a faster medium for storing checkpoint images. It is trivial to make the host use these devices to backup networked file systems used by the Linux* coprocessor OS, but access still has to go through the host. It may be more effective to let the Intel® Xeon Phi™ coprocessors access these devices directly over PCI-Express*, but that approach requires that the device be independently accessible from multiple peer Intel® Xeon Phi™ coprocessors and that device space be divided persistently between Intel® Xeon Phi™ coprocessors such that each has its own fast-access file system dedicated to checkpoint images.

3.3.3 Machine Check Architecture (MCA)

Machine Check Architecture is a hardware feature enabling an Intel® Xeon Phi™ coprocessor card to report failures to software by means of interrupts or exceptions. Failures in this context are conditions where logic circuits have detected something out of order, which may have corrupted processor context or memory content. Failures are categorized by severity as either DUEs or CEs:

- DUEs (Detected Unrecoverable Errors) are errors captured by the MC logic but the corruption cannot be repaired and the system as a whole is compromised; for example, errors in L1 cache.
- CEs (Corrected Errors) are errors that have occurred and been corrected by the hardware, such as single bit errors in L2 ECC memory.

3.3.3.1 MCA Hardware Design Overview

Standard IA systems implement MCA by providing two mechanisms to report MC events to software: MC exceptions (#18) for events detected in the CPU core and NMI (#2) interrupts for events detected outside of the CPU core (uncore).

Specifics on occurred MC exceptions are presented in MSR banks, each representing up to 32 events. The processor capability MSRs specify how many banks are supported by a given processor. The interpretation of data in MSR banks is semi-standardized; that is, acquiring detailed raw data on an event is standardized but the interpretation of acquired raw data is not. The Intel® Xeon Phi™ coprocessor provides three MC MSR banks.

MC events signaled through the NMI interrupt on standard IA systems come from the chipsets and represent failures in memory or I/O paths. Newer CPUs with built-in memory controllers also provide a separate interrupt for CEs (CMCIs) that have built-in counter dividers to throttle interrupt rates. This capability is not provided on the Intel® Xeon Phi™ coprocessor. Instead, the Intel® Xeon Phi™ coprocessor delivers both uncorrected and corrected errors that are detected in the core domain via the standard MCA interrupt (#18). Machine check events that occur in the uncore domain are delivered via the SBox, which can be programmed to generate an NMI interrupt targeted at one or all threads. The Uncore Interrupt includes MC events related to the PCI-Express interface, Memory Controller (ECC and link training errors), or other uncore units. There is no CE error rate throttle in the Intel® Xeon Phi™ coprocessor. The only remedy against high error frequencies is to disable the interrupt at the source of the initiating unit (L2/L1 Cache, Tag Directory, or GBox).

The NMI interrupt handler software must handle a diverse range of error types on Intel® Xeon Phi™ coprocessor. Registers to control and report uncore MC events on Intel® Xeon Phi™ coprocessor differ significantly from registers on standard IA chipsets, which means that stock operating systems have no support for uncore MC events on an Intel® Xeon Phi™ coprocessor.

3.3.3.2 MCA Software Design Overview

Intel® Xeon Phi™ coprocessor RAS demands that the software perform MC event handling in two stages, event data gathering and event post processing.

The first stage (which takes place in the Linux* coprocessor OS) receives MC event notifications, collects raw data, and dispatches it to interested parties (i.e., an MCA agent running on the host and the on-card SMC controller). If the coprocessor OS can resume operation, then its event handling is completed. Otherwise, the MC event handler notifies the host separately that its internal state has been corrupted and a reboot is required.

An unrelated service for host-side monitoring of the Intel® Xeon Phi™ coprocessor card state will be added to the MCA handling routines. This service will act as a gateway between host side 'in-band' platform management and the SMC sub-system and respond to system state queries, such as memory statistics, free memory, temperatures, CPU states etc. Host queries of the coprocessor OS MCA log is a part of the service too.

3.3.3.3 MC Event Capture in Linux* 2.6.34

The stock Linux* kernel has support for core MCs in a single dedicated exception handler. The handler expects MCA exceptions to be broadcast to all processors in the system, and it will wait for all CPUs to line up at a rendezvous point before every CPU inspects its own MCA banks and stores flagged events in a global MC event log (consisting of 32 entries). Then the handler on all CPUs lines up at a rendezvous point again and one CPU (the monarch, which is selected as the first entering the MCA event handler) gets to grade the MCA events collected in the global MC event log and to determine whether to panic or resume operation. This takes place in function `monarch_reign()`. If resumed, the MCA handler may send BUS-ERROR signals to the processes affected by the error. Linux* has several kernel variables that control sensitivity to MCA exceptions, ranging from always panic to always ignore them.

Linux* expects MC events to be broadcast to all CPUs. The rendezvous point uses CPU count versus event handler entries as wait criteria. The wait loop is implemented as a spinlock with timeout, such that a defunct CPU cannot prevent the handler from completing.

NMI interrupts on Linux* are treated one way for the boot processor (BP) and differently on the application processors (AP). Signals from the chipset are expected to be routed only to the BP and only the BP will check chipset registers to determine the NMI source. If chipset flags `SERR#` or `IOCHK` are set the BP NMI handler consults configurable control variables to select panic or ignore the MC event. Otherwise, and on APs, the NMI handler will check for software watchdog timers, call registered NMI handlers, or if not serviced then a configurable control variables to select panic or ignore the unknown NMI.

3.3.3.4 MC Handling in the Intel® Xeon Phi™ Coprocessor Linux*-based coprocessor OS

The Linux* coprocessor OS MCA logic handles capture of core MC events on the Intel® Xeon Phi™ coprocessor without modifications if events are broadcast to all CPUs the same way as on standard IA systems. A callout is required from `monarch_reign()` to a new module for distribution of MC event reports to other interested parties (such as the SMC and the host side MC agent). After distributing the MC events, the Linux* coprocessor OS uses the grading result to select between CEs that resume operation immediately and DUEs that must request a reboot to maintenance mode and then cease operation. Another callout from `monarch_reign()` is required for this part.

Handling of NMIs in the Linux* coprocessor OS requires new code because uncore MCA registers are completely different from those of chipset MCA; for example, MMIO register banks vs. I/O registers. Uncore MCA registers are organized similarly to core MCA banks, but the access method for 32-bit MMIO vs. 64-bit MSRs differs sufficiently to make a merge into the MCA exception handler code unfeasible. However, the global MC event log, the use of `monarch_reign()`, and the event signaling to the host side MCA agent should be the same for the NMI handler as it is for the MC exception handler.

3.3.3.5 MCA Event Sources and Causes

MCA events are received from three sources on the ring: the CPU box, the GBox, and the SBox. For more information on the encoding and controls available on the MCA features, refer to Section 3.3.3.8.

3.3.3.6 MCA Event Post-Processing (coprocessor OS Side Handling)

Once the MC event(s) has been collected into the global MC event log and graded, the failure has been classified as either a DUE or CE. Both event types are distributed to the host and the SMC, potentially with some form of throttling or discrimination based on user configurable settings (via the kernel command line as a boot parameter or at runtime through the control panel).

On CE type failures, the Intel® Xeon Phi™ coprocessor will resume operation because the hardware state is intact. DUE failures cannot be ignored and the next action is to signal the host for a reboot into maintenance mode.

These activities are initiated by callbacks from a special routine and the NMI exception handler. The processing context is exception or interrupt. Both of these require careful coding because locking cannot be relied on for synchronization, even to a separate handler thread. The stock Linux* reaction to a DUE is simply to panic. On the Intel® Xeon Phi™ coprocessor, the recorded events must be distributed to at least two parties, both of which are based on non-trivial APIs (the I2C driver for reporting to the SMC and the SCIF driver for reporting to the host-side MC agent).

3.3.3.7 MCA Event Post-Processing (Host Side Handling)

There are several active proposals on what sort of processing is required for MC events. The Linux* coprocessor OS will capture events in raw form and pass them to an agent on the host for further processing.

The host side MCA agent is a user space application using dedicated SCIF connections to communicate with the Intel® Xeon Phi™ coprocessor Linux* coprocessor OS MCA kernel module. The agent is responsible for the following:

- Maintaining and providing access to a permanent MC event log on the host, preferably as a file on the host's local file system. This agent also handles the distribution of events beyond the host.
- Providing a means to reset (or to trigger a reset) of an Intel® Xeon Phi™ coprocessor card into maintenance mode and passing the latest MC event. The card reset needs support by the host side Intel® Xeon Phi™ coprocessor driver since ring0 access is required.
- Optionally providing access to the Intel® Xeon Phi™ coprocessors global MC event log
- Acting as the host side application; that is, the RAS defect analyzer providing an interface to dump MCA error records from the EEPROM.

The design of the host side MCA agent is beyond the scope of this document. It must place as much content as possible as a user mode application in order to keep the host side drivers as simple and portable as possible. It shall be noted that it has been requested to have sysfs nodes on Linux* hosts present Intel® Xeon Phi™ coprocessor card properties, including MC event logs and states. This may require a kernel agent on the host side to provide the sysfs nodes.

Beyond the overlap of features between driver and user mode agent, this also has issues with SCIF because only one party can own a SCIF queue pair. Having separate SCIF links for the kernel driver and user space agent is not feasible. The host side MCA agent may split into a kernel driver to provide the sysfs nodes and a user space application using the sysfs nodes, where only the kernel driver use SCIF.

3.3.3.8 Core CPU MCA Registers (Encoding and Controls)

While the Intel® Xeon Phi™ coprocessor does support MCA and MCE capabilities, the CPUID feature bits used to identify the processor supports for these features are not set on the Intel® Xeon Phi™ coprocessor.

The Intel® Xeon Phi™ coprocessor implements a mechanism for detecting and reporting hardware errors. Examples of these errors include on-chip cache errors, memory CRC errors, and I/O (PCI Express) link errors. The Intel® Xeon Phi™ coprocessor uses sets of MSR registers to setup machine checking as well as to log detected errors.

Machine checks on the Intel® Xeon Phi™ coprocessor are broken down into two domains:

- Core machine check events, which are handled in a similar fashion to the IA MCA architecture definition
- System machine check events, which are handled in a similar fashion to chipset machine check events

Machine-check event delivery on the Intel® Xeon Phi™ coprocessor is not guaranteed to be synchronous with the instruction execution that may have caused the event. Therefore, recovery from a machine check is not always possible. Software is required to determine if recovery is possible, based on the information stored in the machine-check registers.

The Intel® Xeon Phi™ coprocessor MCA implements one set of MC general registers per CPU (core control registers). There are three banks of MCx registers per core. All hardware threads running on a core share the same set of registers. These registers are for the L1 cache, the L2 cache and the Tag Directories. For the uncore sections, there is one bank of registers per box (GBox, SBox, etc.), each of which is composed of eight 32-bit registers. All uncore events are sent over a serial link to the SBox's I/O APIC. From the I/O APIC, an interrupt is sent to a core, after which normal interrupt processing occurs.

The machine check registers on the Intel® Xeon Phi™ coprocessor consist of a set of core control registers, error reporting MSR register banks, and global system error reporting banks containing error status for the RAS agents. Most core machine-check registers are shared amongst all the cores. The machine-check error reporting registers are listed in Table 3-5.

Table 3-5. Control and Error Reporting Registers

Intel® Xeon Phi™ Coprocessor Machine Check Control Registers		
Register Name	Size (bits)	Description
MCG_CAP	64	Core machine check capability register
MCG_STATUS	64	Core machine check status register
MCG_CTL	64	Core machine check control register (per thread register)
Intel® Xeon Phi™ Coprocessor Machine Error Reporting Registers		
Register Name	Register Name	Register Name
MCI_CTL	64	Machine check control register
MCI_STATUS	64	Machine check status register
MCI_ADDR	64	Machine check address register
MCI_MISC	32	Not Implemented in every MC bank

3.3.3.8.1 MCI_CTL MSR

The MCI_CTL MSR controls error reporting for specific errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EEj) represents a potential error. Setting an EEj flag enables reporting of the associated error, and clearing it disables reporting of the error. Writing the 64-bit value FFFFFFFFFFFFFFFFH to an MCI_CTL register enables the logging of all errors. The coprocessor does not write changes to bits that are not implemented.

Table 3-6. MCI_CTL Register Description

Field Name	Bit Range	Description	Type
EEj	63:0	Error reporting enable flag (where j is 00 through 63)	R/W

3.3.3.8.2 MCI_STATUS MSR

The MCI_STATUS MSR contains information related to a machine check error if its VAL (valid) flag is set. Software is responsible for clearing the MCI_STATUS register by writing it with all 0's; writing 1's to this register will cause a general-protection exception to be generated. The fields in this register are as follows (see also Table 3-7):

- **MCA (machine-check architecture) error code field, bits 0 through 15**
Specifies the machine-check architecture defined error code for the machine-check error condition detected.
- **Model-specific error code field, bits 16 through 31**
Specifies the model-specific error code that uniquely identifies the machine-check error condition detected.
- **Other information field, bits 32 through 56**
The functions of the bits in this field are implementation specific and are not part of the machine-check architecture.
- **PCC (processor context corrupt) flag, bit 57**
Indicates (when set) that the state of the processor might have been corrupted by the detected error condition and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor's state.
- **ADDRV (MCI_ADDR register valid) flag, bit 58**
Indicates (when set) that the MCI_ADDR register contains the address where the error occurred. When clear, this flag indicates that the MCI_ADDR register does not contain the address where the error occurred.
- **MISCV (MCI_MISC register valid) flag, bit 59**
Indicates (when set) that the MCI_MISC register contains additional information regarding the error. When clear, this flag indicates that the MCI_MISC register does not contain additional information regarding the error.
- **EN (error enabled) flag, bit 60**
Indicates (when set) that the error was enabled by the associated EEj bit of the MCI_CTL register.
- **UC (error uncorrected) flag, bit 61**
Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.
- **OVER (machine check overflow) flag, bit 62**
Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the MCI_STATUS register). The processor sets the OVER flag and software is responsible for clearing it.
- **VAL (MCI_STATUS register valid) flag, bit 63**
Indicates (when set) that the information within the MCI_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the MCI_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

The VAL bit is only set by hardware when an MC event is detected and the respective MC enable bit in the MCI_CTL register is set as well. Software should clear the MCI_STATUS.VAL bit by writing all 0's to the MCI_STATUS register.

Table 3-7. MCI_STATUS Register Description

Field Name	Bit Range	Description	Type
MCA Code	15:0	MCA error code field	R/W
Model Code	31:16	Model specific error code	R/W
Info	56:32	Other information field	R/W
PCC	57	Processor context corrupt	R/W
ADDRV	58	MCI_ADDR register valid	R/W
MISCV	59	MCI_MISC register valid	R/W
EN	60	Error enabled	R/W

Field Name	Bit Range	Description	Type
UC	61	Uncorrected error	R/W
OVER	62	Error overflow	R/W
VAL	63	MCI_STATUS register valid	R/W

3.3.3.8.3 MCI_ADDR MSR

The MCI_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDR_V flag in the MCI_STATUS register is set. The address returned is a physical address on the Intel® Xeon Phi™ coprocessor.

Table 3-8. MCI_ADDR Register Description

Field Name	Bit Range	Description	Type
Address	n:0	Address associated with error event	R/W
Reserved	63:n	Reserved (where n is implementation specific)	R/W

3.3.3.8.4 MCI_MISC MSR

The MCI_MISC MSR contains additional information describing the machine-check error if the MISCV flag in the MCI_STATUS register is set. This register is not implemented in the MCO error-reporting register banks of the Intel® Xeon Phi™ coprocessor.

3.3.3.8.5 Summary of Machine Check Registers

Table 3-9 describes the Intel® Xeon Phi™ Coprocessor MCA registers.

Table 3-9. Machine Check Registers

MSR/MMIO Address	Register Name	Size (bits)	Description
Core MCA Registers			
179H	MCG_CAP	64	Core machine check capability register
17AH	MCG_STATUS	64	Core machine check Status register
17BH	MCG_CTL	64	Core machine check control register
400H	MCO_CTL	64	Core machine check control register
401H	MCO_STATUS	64	Core machine check status register
402H	MCO_ADDR	64	Core machine check address register (Not Implemented)
403H	MCO_MISC	32	Core machine check miscellaneous register (Not Implemented)
Intel® Xeon Phi™ coprocessor MCA Registers			
404H	MC1_CTL	32	L2 Cache machine check control register
405H	MC1_STATUS	64	L2 Cache machine check status register
406H	MC1_ADDR	64	L2 Cache machine check address register
407H	MC1_MISC	32	L2 Cache machine check Misc register
TAG Directory MCA Registers			
408H	MC2_CTL	32	TAG Directory machine check control register
409H	MC2_STATUS	64	TAG Directory machine check status register
40AH	MC2_ADDR	64	TAG Directory machine check address register
40BH	MC2_MISC	32	TAG Directory (Not Implemented)
Uncore MCA Registers (#18 MCA interrupt not generated. Signalling via local interrupt controller)			

MSR/MMIO Address	Register Name	Size (bits)	Description
SBox MCA Registers			
0x8007D3090	MCX_CTL_LO	32	SBox machine check control register
0x8007D3094	MCX_CTL_HI	32	SBox machine check control register (Not implemented)
0x8007D3098	MCX_STATUS_LO	32	SBox machine check status register
0x8007D309C	MCX_STATUS_HI	32	SBox machine check status register
0x8007D30A0	MCX_ADDR_LO	32	SBox machine check address register
0x8007D30A4	MCX_ADDR_HI	32	SBox machine check address register
0x8007D30A8	MCX_MISC	32	SBox Misc (timeout TID register)
0x8007D30AC	MCX_MISC2	32	SBox Misc (timeout address register)
0x8007DAB00	MCA_INT_STAT	32	SBox MCA Interrupt Status Register (Not retained on warm reset)
0x8007DAB04	MCA_INT_EN	32	SBox MCA Interrupt Enable Register (Not retained on warm reset)
Standalone TAG Directory 0 MCA Registers			
0x8007C0340	RTD_MCX_CTL	32	TAG Directory machine check control register
0x8007C0348	RTD_MCX_STATUS	64	TAG Directory machine check status register
0x8007C0350	RTD_MCX_ADDR	64	TAG Directory machine check address register
Standalone TAG Directory 1 MCA Registers			
0x800620340	RTD_MCX_CTL	32	TAG Directory machine check control register
0x800620348	RTD_MCX_STATUS	64	TAG Directory machine check status register
0x800620350	RTD_MCX_ADDR	64	TAG Directory machine check address register
Memory Controller (Gbox0) MCA Registers			
0x8007A005C	MCX_CTL_LO	32	Gbox0 Fbox machine check control register
0x8007A0060	MCX_CTL_HI	32	Gbox0 Fbox machine check control register
0x8007A0064	MCX_STATUS_LO	32	Gbox0 Fbox machine check status register
0x8007A0068	MCX_STATUS_HI	32	Gbox0 Fbox machine check status register
0x8007A006C	MCX_ADDR_LO	32	Gbox0 Fbox machine check address register
0x8007A0070	MCX_ADDR_HI	32	Gbox0 Fbox machine check address register
0x8007A0074	MCX_MISC	32	Gbox0 Fbox Misc (Transaction timeout register)
0x8007A017C	MCA_CRC0_ADDR	32	Gbox0 Mbox0 CRC address capture register
0x8007A097C	MCA_CRC1_ADDR	32	Gbox0 Mbox1 CRC address capture register
Memory Controller (Gbox1) MCA Registers			
0x80079005C	MCX_CTL_LO	32	Gbox1 Fbox machine check control register
0x800790060	MCX_CTL_HI	32	Gbox1 Fbox machine check control register
0x800790064	MCX_STATUS_LO	32	Gbox1 Fbox machine check status register
0x800790068	MCX_STATUS_HI	32	Gbox1 Fbox machine check status register
0x80079006C	MCX_ADDR_LO	32	Gbox1 Fbox machine check address register
0x800790070	MCX_ADDR_HI	32	Gbox1 Fbox machine check address register
0x800790074	MCX_MISC	32	Gbox1 Fbox Misc (Transaction timeout register)
0x80079017C	MCA_CRC0_ADDR	32	Gbox1 Mbox0 CRC address capture register
0x80079097C	MCA_CRC1_ADDR	32	Gbox1 Mbox1 CRC address capture register
Memory Controller (Gbox2) MCA Registers			
0x80070005C	MCX_CTL_LO	32	Gbox2 Fbox machine check control register
0x800700060	MCX_CTL_HI	32	Gbox2 Fbox machine check control register
0x800700064	MCX_STATUS_LO	32	Gbox2 Fbox machine check status register
0x800700068	MCX_STATUS_HI	32	Gbox2 Fbox machine check status register
0x80070006C	MCX_ADDR_LO	32	Gbox2 Fbox machine check address register
0x800700070	MCX_ADDR_HI	32	Gbox2 Fbox machine check address register
0x800700074	MCX_MISC	32	Gbox2 Fbox Misc (Transaction timeout register)

MSR/MMIO Address	Register Name	Size (bits)	Description
0x80070017C	MCA_CRC0_ADDR	32	Gbox2 Mbox0 CRC address capture register
0x80070097C	MCA_CRC1_ADDR	32	Gbox2 Mbox1 CRC address capture register
Memory Controller (Gbox3) MCA Registers			
0x8006F005C	MCX_CTL_LO	32	Gbox3 Fbox machine check control register
0x8006F0060	MCX_CTL_HI	32	Gbox3 Fbox machine check control register
0x8006F0064	MCX_STATUS_LO	32	Gbox3 Fbox machine check status register
0x8006F0068	MCX_STATUS_HI	32	Gbox3 Fbox machine check status register
0x8006F006C	MCX_ADDR_LO	32	Gbox3 Fbox machine check address register
0x8006F0070	MCX_ADDR_HI	32	Gbox3 Fbox machine check address register
0x8006F0074	MCX_MISC	32	Gbox3 Fbox Misc (Transaction timeout register)
0x8006F017C	MCA_CRC0_ADDR	32	Gbox3 Mbox0 CRC address capture register
0x8006F097C	MCA_CRC1_ADDR	32	Gbox3 Mbox1 CRC address capture register
Memory Controller (Gbox4) MCA Registers			
0x8006D005C	MCX_CTL_LO	32	Gbox4 Fbox machine check control register
0x8006D0060	MCX_CTL_HI	32	Gbox4 Fbox machine check control register
0x8006D0064	MCX_STATUS_LO	32	Gbox4 Fbox machine check status register
0x8006D0068	MCX_STATUS_HI	32	Gbox4 Fbox machine check status register
0x8006D006C	MCX_ADDR_LO	32	Gbox4 Fbox machine check address register
0x8006D0070	MCX_ADDR_HI	32	Gbox4 Fbox machine check address register
0x8006D0074	MCX_MISC	32	Gbox4 Fbox Misc (Transaction timeout register)
0x8006D017C	MCA_CRC0_ADDR	32	Gbox4 Mbox0 CRC address capture register
0x8006D097C	MCA_CRC1_ADDR	32	Gbox4 Mbox1 CRC address capture register
Memory Controller (Gbox5) MCA Registers			
0x8006C005C	MCX_CTL_LO	32	Gbox5 Fbox machine check control register
0x8006C0060	MCX_CTL_HI	32	Gbox5 Fbox machine check control register
0x8006C0064	MCX_STATUS_LO	32	Gbox5 Fbox machine check status register
0x8006C0068	MCX_STATUS_HI	32	Gbox5 Fbox machine check status register
0x8006C006C	MCX_ADDR_LO	32	Gbox5 Fbox machine check address register
0x8006C0070	MCX_ADDR_HI	32	Gbox5 Fbox machine check address register
0x8006C0074	MCX_MISC	32	Gbox5 Fbox Misc (Transaction timeout register)
0x8006C017C	MCA_CRC0_ADDR	32	Gbox5 Mbox0 CRC address capture register
0x8006C097C	MCA_CRC1_ADDR	32	Gbox5 Mbox1 CRC address capture register
Memory Controller (Gbox6) MCA Registers			
0x8006B005C	MCX_CTL_LO	32	Gbox6 Fbox machine check control register
0x8006B0060	MCX_CTL_HI	32	Gbox6 Fbox machine check control register
0x8006B0064	MCX_STATUS_LO	32	Gbox6 Fbox machine check status register
0x8006B0068	MCX_STATUS_HI	32	Gbox6 Fbox machine check status register
0x8006B006C	MCX_ADDR_LO	32	Gbox6 Fbox machine check address register
0x8006B0070	MCX_ADDR_HI	32	Gbox6 Fbox machine check address register
0x8006B0074	MCX_MISC	32	Gbox6 Fbox Misc (Transaction timeout register)
0x8006B017C	MCA_CRC0_ADDR	32	Gbox6 Mbox0 CRC address capture register
0x8006B097C	MCA_CRC1_ADDR	32	Gbox6 Mbox1 CRC address capture register
Memory Controller (Gbox7) MCA Registers			
0x8006A005C	MCX_CTL_LO	32	Gbox7 Fbox machine check control register
0x8006A0060	MCX_CTL_HI	32	Gbox7 Fbox machine check control register
0x8006A0064	MCX_STATUS_LO	32	Gbox7 Fbox machine check status register
0x8006A0068	MCX_STATUS_HI	32	Gbox7 Fbox machine check status register

MSR/MMIO Address	Register Name	Size (bits)	Description
0x8006A006C	MCX_ADDR_LO	32	Gbox7 Fbox machine check address register
0x8006A0070	MCX_ADDR_HI	32	Gbox7 Fbox machine check address register
0x8006A0074	MCX_MISC	32	Gbox7 Fbox Misc (Transaction timeout register)
0x8006A017C	MCA_CRC0_ADDR	32	Gbox7 Mbox0 CRC address capture register
0x8006A097C	MCA_CRC1_ADDR	32	Gbox7 Mbox1 CRC address capture register

3.3.3.9 Uncore MCA Registers (Encoding and Controls)

The Intel® Xeon Phi™ coprocessor’s uncore agents (which are not part of the core CPU) signal their machine-check events via the I/O APIC, and log error events via agent-specific error control and logging registers. These registers are implemented as registers in Intel® Xeon Phi™ coprocessor MMIO space associated with each uncore agent that is capable of generating machine-check events.

Once an error is detected by an uncore agent, it signals the interrupt controller located in the uncore system box (SBox). The SBox logs the source of the error and generates an interrupt to the specified LRB programmed in the APIC redirection tables.

Software must check all the uncore machine-check banks to identify the source of the uncore machine-check event. To enable the generation of a machine-check event from a given source, the software should set the corresponding bit in the SBox MCA Interrupt Enable Register (MCA_INT_EN). To disable the generation of machine-check events from a given source, the software should clear the corresponding bit of the SBox MCA_INT_EN register.

Sources of uncore machine-check events in the Intel® Xeon Phi™ coprocessor uncore are listed in Table 3-10.

Table 3-10. Sources of Uncore Machine-Check Events

Uncore Agent Name	number of instances	Description
SBox	1	System agent
Tag Directory	2	Tag Directories not collocated with CPU slice
GBox	8	Memory controller

Each uncore agent capable of generating machine-check events contains event control and logging registers to facilitate event detection and delivery.

3.3.3.9.1 System Agent (SBox) Error Logging Registers

The SBox contains a set of machine check registers similar to core bank registers, but implemented in MMIO (USR register) for reporting errors. Machine check events from the SBox are routed to the OS running on a specified thread via the local APIC in the SBox. The SBox local APIC redirection table assigned to MCA interrupts must be programmed with a specific thread in order to service SBox (and other uncore) machine-check events. Errors related to DMA requests are handled directly by the affected DMA Channel itself and are reported to the DMA SW Driver via the local I/O APIC or by the System Interrupt logic, depending on the assigned ownership of the channel. All MCA errors detected by the SBox are logged in the SBox MCA logging registers (MCx.STATUSx, MCx.MISCx, and MCx.ADDR) regardless of whether the corresponding MCA_CTL bit is set, the exception being when the MCA_STATUS.EN bit is already set. Only errors with their corresponding bit set in the MCx.CTL register can signal an error.

Table 3-11. SBox Machine Check Registers

Register Name	Size (bits)	Description
MCX_CTL_LO	32	Machine check control register

MCX_CTL_HI	32	Machine check control register (Reads 0, Writes Dropped, Not implemented on coprocessor)
MCX_STATUS_LO	32	Machine check status register
MCX_STATUS_HI	32	Machine check status register
MCX_ADDR_LO	32	Machine check address register
MCX_ADDR_HI	32	Machine check address register
MCX_MISC	32	Misc (timeout TID register)
MCX_MISC2	32	Misc (timeout address register)

3.3.3.9.2 Multiple Errors and Errors Over Multiple Cycles

There are two cases in which the SBox may receive two or more errors before the software has a chance to process each individual error:

5. Multiple errors (errors occurring simultaneously). This occurs when multiple error events are detected in the same cycle. Essentially, this allows the hardware not to try and decode and prioritize multiple errors that occur in the same cycle.
6. Errors occurring one after another over multiple cycles. This occurs when an existing error is already logged in the MCx register and another error is received in a subsequent cycle.

3.3.3.9.3 SBox Error Events

Table 3-12 lists the value of the Model code associated with each individual error. The sections following the table provide some additional information on a select set of these errors.

Table 3-12. SBox Error Descriptions

Error Class	MCX_CTL Bit	Error Name	Model Code	Description	SBOX Behaviour
Unsuccessful Completion	9	Received Configuration Request Retry Status (CRS)	0x0006h	A Completion with Configuration Request Retry Status was received for a Request from a Ring Agent	All 1's for data returned to the Ring
	1	Received Completer Abort (CA)	0x0007h	A Completion with Completer Abort status was received for a Request from a Ring Agent	All 1's for data returned to the Ring
	2	Received Unsupported Request (UR)	0x0008h	A Completion with Unsupported Request status was received for a Request from a Ring Agent	All 1's for data returned to the Ring
Poisoned Data	7	Received Poisoned Data in Completion (PD)	0x0040h	A Successful Completion (SC) with Poisoned Data was received for a Request from a Ring Agent	Data payload with error is returned to the Ring
Timeout	6	Upstream Request terminated by Completion Timeout (CTO)	0x0009h	A Completion Timeout was detected for a Request from a Ring Agent	All 1's for data returned to the Ring.

Error Class	MCX_CTL Bit	Error Name	Model Code	Description	SBOX Behaviour
Illegal Access	3	Downstream Address outside of User accessible Range	0x0020h	PCIe downstream attempt to use indirect registers to access illegal address ranges via the I/O space	RD: Successful Completion (SC) with all 0's for data returned to PCIe WR: Discard transaction. Successful Completion (SC) with no data returned to PCIe.
	8	Unclaimed Address (UA)	0x0021h	A Ring Agent Request to an unclaimed address was terminated by subtractive decode	RD: All 1's for data returned to the Ring. WR: Request is discarded.
PCIe Error	4	PCIe Correctable Error	0x0030h	A PCIe correctable error was logged by the Endpoint	ERR_COR Message transmitted on PCIe
	5	PCIe Uncorrectable Error	0x0031h	A PCIe Uncorrectable error was logged by the Endpoint	ERR_NONFATAL or ERR_FATAL Message transmitted on PCIe

3.3.3.9.3.1 Timeout Error

An upstream timeout occurs when the target fails to respond to an Intel® Xeon Phi™ coprocessor-initiated read request within a programmable amount of time. The PCIe endpoint keeps track of these outstanding completions and will signal the GHOST unit when it is okay to free up the buffers allocated to hold the timed out completion. To ensure that the core subsystem within the Intel® Xeon Phi™ coprocessor doesn't hang while waiting for a read that will never return, the SBox generates a dummy completion back to the requesting thread. The payload of this completion (BAD66BAD) clearly indicates that the completion is fake. As well as generating an MCA event that is logged in the MCX_STATUS register, a portion of the completion header associated with the failing PCIe transaction is logged in the MCX_MISC register.

3.3.3.9.3.2 Unrecognized Transaction Error

This type of error indicates that a transaction was dropped by the SBox because it was of a type that is not handled by Intel® Xeon Phi™ coprocessor. Transactions that fall into this category are usually vendor-specific messages that are not recognized by the Intel® Xeon Phi™ coprocessor.

3.3.3.9.3.3 Illegal Access Error

An illegal access error indicates that the SBOX was unable to complete the transaction because the destination address was not within the legal range. For inbound transactions initiated by PCIe, this can only happen via I/O read and write cycles to the indirect address and data ports. If the user specifies an address above or below the range set aside for MMIO host visibility, a machine check exception will be generated and key debug information will be logged for software inspection.

Ring-initiated transactions can also result in an illegal access error if the coprocessor OS or Tag Directory contains flaws in the coding or logic. The SBox microarchitecture will ensure that all EXT_RD and EXT_WR transactions are delivered to the endpoint scheduler for inspection. If the destination address (an internal Intel® Xeon Phi™ coprocessor address) does not match one of the direct-access ranges set aside for the Flash device or does not match one of the 32 available system memory ranges, it will be terminated and a default value returned to the requester. If the ring traffic was routed to the SBox in error, this will likely fail all built-in address range checks and will overload the platform as a result. To guard against this possibility, the endpoint scheduling logic *must explicitly match one of its valid address ranges before driving PCI Express link*. Outbound traffic that fails this check will result in the following explicit actions:

- EXT_WR transactions will be discarded, key debug information will be logged and an MCA exception will be generated.
- EXT_RD transactions will complete and return data back to the requestor, key debug information will be logged, and an MCA exception will be generated.

3.3.3.9.3.4 Correctable PCIe Fabric Error

Errors detected in the PCIe fabric will generate an MCA error and be logged in the MCX_STATUS register as an event. It is the responsibility of the software handler to extract the error event from the PCIe standalone agent status registers as well as from communication with the PCIe host. The SBox does not log any more information on this error than what is contained in the MCX status register. These errors are signaled by the assertion of this endpoint interface signal.

Table 3-13. Correctable PCIe Fabric Error Signal

Signal Name	Width	Description
func0_rep_cor_err	Scalar	The end point has sent a correctable error message to the root complex

3.3.3.9.3.5 Uncorrectable PCIe Fabric Error

Errors detected in the PCIe fabric (GDA) will generate an MCA error and be logged in the MCX_STATUS register as an event. It is the responsibility of the software handler to extract the error event from the PCIe standalone agent status registers as well as from communication with the PCIe host. The SBox does not log any more information on this error than is contained in the MCX status register. These errors are signaled by the assertion of this endpoint interface signal.

Table 3-14. Uncorrectable PCIe Fabric Error Signal

Signal Name	Width	Description
func0_rep_uncor_err	Scalar	The end point has sent an uncorrectable error message (fatal or nonfatal) to the root complex

3.3.3.9.4 GBox Error Events

Table 3-15. GBox Errors

Error Category	MCX_CTL Bit	Error Name	Model Code	Description	GBOX Behaviour
ECC	2	Correctable ECC Error Detected Ch 0	0x00000000 4h	Single bit ECC error on channel 0	Log/Signal Event
	3	Correctable ECC Error Detected Ch 1	0x00000000 8h	Single bit ECC error on channel 1	
	31	Uncorrectable ECC Error Detected Ch 0	0x08000000 0h	Double bit ECC error on channel 0	Log/Signal Event "Corrupted" Data may be returned to consumer.
	32	Uncorrectable ECC Error Detected Ch	0x10000000	Double bit ECC error	

Error Category	MCX_CTL Bit	Error Name	Model Code	Description	GBOX Behaviour
		1	0h	on channel 1	
	33	Illegal Access to Reserved ECC Memory Space	0x20000000 0h	Access to reserved ECC memory	
CAPE	4	CAPE Exceeded Threshold Ch 0	0x00000001 0h	Memory Cape threshold Exceeded on Ch0	Log/Signal Event
	5	CAPE Exceeded Threshold Ch 1	0x00000002 0h	Memory Cape threshold Exceeded on Ch0	
Training	0	Channel 0 retraining	0x00000000 1h	Channel 0 retraining event	Log/Signal Event
	1	Channel 1 retraining	0x00000000 2h	Channel 0 retraining event	
	29	Training failure after DM request Ch 0	0x02000000 0h	Training failure after DM request Ch 0	Log/Signal Event Transaction halted
	30	Training failure after DM request Ch 1	0x04000000 0h	Training failure after DM request Ch 1	
Proxy MCA	8	Standalone tag directory Proxy MCA event	0x00000010 0h	MCA event In Standalone Tag Directory	Log/Signal Event
Miscellaneous	6	Transaction Received to an Invalid Channel	0x00000004 0h	Memory transaction with invalid channel encountered	Log/Signal Event "Corrupted" Data may be returned to consumer/Transaction halted
	23	Channel 0 Write Queue overflow	0x00080000 0h	Channel 0 Write Queue overflow	Log/Signal Event Unspecified behaviour
	24	Channel 1 Write Queue overflow	0x00100000 0h	Channel 1 Write Queue overflow	

3.3.3.9.5 Tag Directory Error Events

Table 3-16. TD Errors

Error Category	MXC_CTL Bit	Error Name	Model Code	Description	Logging Register
Tag-State UNCRR Error	0	External Transaction	0x0001h	A tag error occurred on an external TD transaction	MC2_STATUS MC2_ADDR
	0	Internal Transaction	0x0002h	A tag error occurred on an internal TD transaction (i.e. Victim)	MC2_STATUS MC2_ADDR
Core-Valid UNCRR Error	1	External Transaction	0x0010h	A state error occurred on an external TD transaction	MC2_STATUS MC2_ADDR
	1	Internal Transaction	0x0011h	A State error occurred on an internal TD transaction (i.e. Victim)	MC2_STATUS MC2_ADDR
Tag-State CORR	0	External	0x0100h	A tag error occurred on an external	MC2_STATUS

Error Category	MXC_CTL Bit	Error Name	Model Code	Description	Logging Register
Error		Transaction		TD transaction	MC2_ADDR
	0	Internal Transaction	0x0101h	A tag error occurred on an internal TD transaction (i.e. Victim)	MC2_STATUS MC2_ADDR
Core-Valid CORR Error	1	External Transaction	0x0110h	A state error occurred on an external TD transaction	MC2_STATUS MC2_ADDR
	1	Internal Transaction	0x0111h	A State error occurred on an internal TD transaction (i.e. Victim)	MC2_STATUS MC2_ADDR

3.3.3.9.6 Spare Tag Directory (TD) Logging Registers

The Spare Tag Directory contains a set of registers similar to core bank registers but implemented in MMIO (USR register) space instead of the MSR space that co-located TD's are assigned to.

3.3.3.9.7 Memory Controller (GBox) Error Logging Registers

The GBox contains a set of registers similar to core bank registers but implemented in MMIO (USR register) space instead of as MSRs. The GBox signals two classes of events, CRC retry and Training failure. CRC retry is signaled when the GBox attempts a predefined number of retries for a transaction (before initiating retraining). Training failure is signaled when the GBox training logic fails or when a transaction incurs a CRC failure after retraining was initiated.

Table 3-17. GBox Error Registers

Register Name	Size (bits)	Description
MCX_CTL_LO	32	Machine Check control register
MCX_CTL_HI	32	Machine Check control register
MCX_STATUS_LO	32	Machine Check status register
MCX_STATUS_HI	32	Machine Check status register
MCX_ADDR_LO	32	Machine Check address register
MCX_ADDR_HI	32	Machine Check address register
MCX_MISC	32	MISC (Transaction timeout register)

3.3.3.10 Uncore MCA Signaling

Once a machine-check event has occurred in an uncore agent and has been logged in the error reporting register (MCX_STATUS), the MCX_STATUS.VAL bit is sent from each agent to the SBox interrupt controller, which captures this bit in the SBox MCA_INT_STAT register. Each bit of the SBox MCA_INT_STAT register represents an MCA/EMON event of an uncore agent. When the corresponding bit of MCA_INT_EN is also set, then the SBox will generate an interrupt to the specified Intel® Xeon Phi™ coprocessor core with the interrupt vector specified in the SBox interrupt controller's redirection table.

3.3.4 Cache Line Disable

A statistically significant number of SRAM cells can develop erratic and sticky bit failures over time. Burn-in can be used to reduce these types of errors, but it is not sufficient to guarantee that there is statistically insignificant number of these errors as has been the case in the past. These array errors also manifest more readily as a result of the requirement for the product to run at low voltage in order to reduce power consumption. The Intel® Xeon Phi™ coprocessor operational voltage will need to find the right balance between power and reliable operation in this regard and it must be assumed that SRAM arrays on Intel® Xeon Phi™ coprocessor can develop erratic and sticky bit failures.

As a result of the statistically significant SRAM array error sources outlined above, the Intel® Xeon Phi™ coprocessor supports a mechanism known as Cache Line Disable (CLD) that is used to disable cache lines that develop erratic and sticky bit failures. Intel® Xeon Phi™ coprocessor hardware detects these array errors and signals a machine check exception to a machine check handler, which implements the error handling policy and which can (optionally) use CLD to preclude these errors from occurring in the future. Since the cache line in question will no longer be allowed to allocate a new line in the specific array that sourced the error, there may be a slight performance loss. Since the errors can be sticky, and therefore persistent, the Intel® Xeon Phi™ coprocessor remembers the CLDs between cold boots and reapplies the CLDs as part of the reset process before the cache is enabled. This is done through reset packets that are generated in the SBox and delivered to all units with CLD capability.

3.3.5 Core Disable

Similar to Cache Line Disable (CLD), core disable enables the software (OS) to disable a segment of the Intel® Xeon Phi™ coprocessor. Core disable allows the OS to disable a particular Intel® Xeon Phi™ coprocessor core.

Core disable is achieved by writing a segment of the flash room with a core disable mask, and then initiating a cold or warm reboot. The selected cores will not be enumerated.

Core Disable is intended to be used when it is determined that a particular core cannot function correctly due to specific error events. When this case is detected, the coprocessor OS sends information to the host RAS agent corresponding to the affected core. The RAS agent reboots the card into a special processing mode to disable the core, and then resets the Intel® Xeon Phi™ coprocessor card.

On the next reboot, the core disable flash record will be used to disable the selected cores and prevent them from becoming visible to the coprocessor OS for future scheduling. There will be no allocation into the CRI associated with the disabled core, but the co-located TD will still function to maintain Intel® Xeon Phi™ coprocessor coherency.

3.3.6 Machine Check Flows

This section describes the physical sources of machine check events on the Intel® Xeon Phi™ coprocessor and the hardware flows associated with them. It also suggests how software handlers should address these machine check events.

3.3.6.1 Intel® Xeon Phi™ Coprocessor Core

Sources for machine check events are the L1 instruction and L1 data caches and their associated TLB's as well as the microcode ROM.

The L1 instruction cache is protected by parity bits. There is no loss of machine state when a cache parity error occurs. MCA's generated due to parity errors are informational only and are corrected in hardware. The intent is for software to log the event.

Both TLB's are protected by parity and contain architectural state. Errors in the TLB's are uncorrected. It is up to the software handler to decide if execution should continue.

The L1 data cache is parity protected, but it does contain modified cache lines that make recovery impossible in every case. Also, it does not have any mechanism to convey its machine-check events in a synchronous fault. Hence, instructions that encounter parity errors will consume the bad data from the cache. Software must decide if execution should continue upon receiving a parity error. The reporting registers provided for this cache allow software to invalidate or flush the cache index and way that encountered the parity error.

The Cache Ring Interface (CRI) L2 data cache is protected by ECC. While machine checks are delivered asynchronously with respect to the instruction accessing the cache, single bit errors are corrected by hardware in-line with the data delivery. The L2 tags are protected by parity.

If data integrity is desired, software should consider a mode where all Intel® Xeon Phi™ coprocessor uncorrected errors are treated as fatal errors. To enable potential recovery from L2 cache errors, the address and way of the transaction that encounters an error is logged in the Cache Ring Interface. Software may use the address to terminate applications that use the affected memory addresses range and flush the affected line from cache. L2 cache errors may occur in the Tag array or the data array. Errors in the Tag or data array are typically not corrected and result in incorrect data being returned to the application.

In addition to the error reporting resources, the CRI also contains Cache Line Disable (CLD) registers. These registers are programmed on the accumulation of multiple errors to the same cache set and way. Once written, the cache will not allow allocations into the specified cache set and way.

The Intel® Xeon Phi™ coprocessor does not propagate a poison bit with cache-to-cache transfers. Hence the probability of a bad line in the L2 propagating without a machine check is significantly higher. On a cache-to-cache transfer for a line with bad parity, a machine check is going to be generated on the source L2's core but the data is going to be transferred and cached in the requesting L2 as a good line. As part of the MCA logging on a snoop operation, the destination of data is logged; this information can be used by the error handler to contain the effect of an L2 error.

There are two special cases for snoops. The first is a snoop that encounters a Tag/State error that causes a miss in the Tag. The second case is a snoop that misses in the tag without a Tag error (or a regular miss). In both cases, the CRI should complete the snoop transaction. For snoop types that need a data response, the CRI returns data that may be unrelated to the actual requested line. Snoops that incur a miss with a parity error report a TAG_MISS_UNCORR_ERR error, but coherency snoops (from TD) that miss generate a SNP_MISS_UNCORR_ERR error.

The TD Tag-State (TS) and Core-Valid (CV) arrays are protected by ECC. For the Intel® Xeon Phi™ coprocessor all errors detected in either the TS or CV arrays may generate a MCA event and are logged in the MCA logging register. Single bit errors by the TD are corrected inline and do not change any TD flows for the affected transaction.

Software must decide if and when to try and recover from a TD error. To remove the error from the TD, software must issue WBINVD instructions such that all cores evict all lines from all caches and then evict or flush all possible addresses to the same set as the error address to regain coherency in the TDs as it is not obvious which lines are tracked in a particular TD.

The TD allows one Cache-Line-Disable (CLD) register that software can program to disable allocation to a particular TD set and way.

3.3.6.2 Memory Controller (GBox)

The GBox detects link CRC failures between the PBox and the GDDR devices. These CRC failures are delivered as machine-check events to the SBox and are logged in the error reporting registers located in the GBox. In addition to CRC, ECC protection of GDDR contents has been added. The GBox can detect single and double bit errors and can correct single bit errors. Both single and double bit errors can be enabled to signal machine-check events.

For a read request that encounters a CRC training failure or a double bit ECC error, the GBox will generate a CRC training failure or a double bit ECC error. The GBox will generate a fake completion of the request. On a write the GBox should complete the transaction by dropping the write for failing link training or completing the write for a double bit error.

3.3.7 Machine Check Handler

A software machine check handler (implemented as a kernel module in the coprocessor OS) is required to resolve hardware machine check events triggered during Intel® Xeon Phi™ coprocessor operation. The machine check handler is responsible for logging hardware-corrected events (threshold controlled) and for communicating information to the host RAS agent about uncorrected events and logging these events. The host RAS agent determines the correct action to take on uncorrected events.

3.3.7.1 Basic Flow

Due to the reliability requirements on the Intel® Xeon Phi™ coprocessor and the unique nature of the hardware a generic handler will not suffice and an Intel® Xeon Phi™ coprocessor specific handler is required. A machine-check handler must perform following steps:

1. Stop all threads and divert them to the machine check handler via IPI.
2. Once all threads have reached, the machine check handler skips to step 4.
3. One or more threads may be hung. Trigger shutdown/checkpoint failure and jump to step 20.
4. Read MCA registers in each bank and log the information.
5. If uncorrected error (MCI.STATUS.UC || MCI.STATUS.PCC), then jump to step 9.
6. Write CLD field in flash, if necessary.
7. If the reliability threshold is met, then jump to step 9.
8. Exit handler.
9. Turn off all cache and disable MCA (MCI.CTL) for all MC banks.
10. Perform a WBINV to flush L2 contents.
11. Invalidate L1 instruction and data caches via test registers.
12. Turn on the caches, but not MCA.
13. On a selected thread/core, perform a read of the entire GDDR memory space.
14. Perform a WBINV to flush the contents of the selected core.
15. Clear MCI.STATUS registers for all MC banks.
16. If reliability testing is not enabled, jump to step 20.
17. Perform a targeted test of the caches.
18. Check the contents of the MCI_STATUS register for failure (note that MC.STATUS.VAL will not be set).
19. If failure is detected, then set CLD to disable affected lines, and then repeat steps 9-15.
20. Turn on MCA (enable MCI.CTL).
21. Asses the severity of the error and determine the action to be taken (i.e., shutdown application, if possible).
22. Clear the MCIP bit.
23. Exit handler.

3.3.8 Error Injection

There are three basic methods that system software can use to simulate machine-check events:

1. Via LDAT DFX register interface.
2. Dedicated Error Injection Registers.
3. Machine checks STATUS register.

3.3.8.1 Error Injection Using LDAT

Errors can be injected directly into protected arrays via the LDAT DFX interface registers.

To trigger an error, modify the contents of the targeted array, corrupting the ECC/parity/Data such that a check performed by hardware fails. This is the preferred method to fully test the array protection mechanism and the MC logic. The following arrays can be tested by this method:

- L1 Instruction and L1 data caches
- Both TLB's
- L2 data array
- L2 tag array
- TD state and CV arrays

3.3.8.2 Dedicated Error Injection Registers

Machine check events can be generated using dedicated error injection registers available for a limited number of protected arrays. For Intel® Xeon Phi™ coprocessor, this is limited to the MC0 and MC1 error reporting banks.

3.3.8.3 Error Injection via MCI_STATUS Register

The last method of injecting MC events into the machine is via the MCI_STATUS register. For the MC1, MC2 and Uncore MC Bank registers writing the MCx_STATUS.VAL bit will cause a machine check event to be generated from the targeted error reporting bank.

3.3.8.4 List of API's for RAS

The following interfaces provide communication between RAS features and other parts of the Intel® Xeon Phi™ coprocessor software:

- SCIF access from exception/interrupt context
- SCIF well known ports for the MCA agent and Linux* coprocessor OS MC event handlers
- SCIF message formats for MC events reported to host side agent
- Reboot to maintenance mode via IOCTL request
- SCIF message formats for Intel® Xeon Phi™ coprocessor system queries and controls
- Throttle mechanism for CEs
- I2C driver for the bus where SMC resides
- I2C identifiers for communicating with the SMC
- Data formats for MC events.
- Data formats for Intel® Xeon Phi™ coprocessor system queries (if any)
- Data formats for system environment changes (fan speeds, temp, etc.)
- Filter for which events to report to SMC
- Storage location in SMC for MC raw data
- Fuse override requests to maintenance mode
- Diagnostic mode entry to maintenance mode
- Data formats on the RAS log in Intel® Xeon Phi™ coprocessor EEPROM

Time reference in maintenance mode (Intel® Xeon Phi™ coprocessor cards have no time reference). If the RAS log includes the timestamp, the host must provide a time base or a reference to a time counter.

4 Operating System Support and Driver Writer's Guide

This section discusses the support features that the Intel® Xeon Phi™ coprocessor provides for the operating system and device drivers.

4.1 Third Party OS Support

The Intel® MIC Architecture products support 3rd party operating systems such as modified versions of Linux* or completely custom designs. The Linux* based coprocessor OS is treated like a 3rd party OS.

4.2 Intel® Xeon Phi™ Coprocessor Limitations for Shrink-Wrapped Operating Systems

This section is intended to help developers port an existing operating system that runs a platform built around an Intel 64 processor to Intel® Xeon Phi™ coprocessor hardware.

4.2.1 Intel x86 and Intel 64 ABI

The Intel x86 and Intel 64 -bit ABI uses the SSE2 XMM registers, which do not exist in the Intel® Xeon Phi™ coprocessor.

4.2.2 PC-AT / I/O Devices

Because the Intel® Xeon Phi™ coprocessor does not have a PCH southbridge, many of the devices generally assumed to exist on a PC platform do not exist. Intel® Xeon Phi™ coprocessor hardware supports a serial console using the serial port device on the SBOX I2C bus. It is also possible to export a standard device, like an Ethernet interface, to the OS by emulating it over system and GDDR memory shared with the host. This allows for higher level functionality, such as SSH or Telnet consoles for interactive and NFS for file access.

4.2.3 Long Mode Support

Intel 64 Processors that support Long mode also support a compatibility submode within Long mode to handle existing 32-bit x86 applications without recompilation. The Intel® Xeon Phi™ coprocessor does not support the compatibility submode.

4.2.4 Custom Local APIC

The local APIC registers have expanded fields for the APIC ID, Logical APIC ID, and APIC Destination ID. Refer to the *SDM Volume 3A System Programming Guide* for details.

There is a local APIC (LAPIC) per hardware thread in the Intel® Xeon Phi™ coprocessor. In addition, the SBox contains within it a LAPIC that has 8 Interrupt Command Registers (ICRs) to support host-to-coprocessor and inter-coprocessor interrupts. To initiate an interrupt from the host to an Intel® Xeon Phi™ coprocessor or from one Intel® Xeon Phi™ coprocessor to another, the initiator must write to an ICR on the target Intel® Xeon Phi™ coprocessor. Since there are 8 ICRs, the system can have up to 8 Intel® Xeon Phi™ coprocessors that can be organized in a mesh topology along with the host.

4.2.5 Custom I/O APIC

The Intel® Xeon Phi™ coprocessor I/O APIC has a fixed 64-bit base address. The base address of the I/O APIC on IA platforms is communicated to the OS by the BIOS (Bootstrap) via MP, ACPI, or SFI table entries. The MP and ACPI table entries use a 32-bit address for the base address of the I/O APIC, whereas the SFI table entry uses a 64-bit address. Operating systems that assume a 32-bit address for the I/O APIC will need to be modified.

The I/O APIC pins (commonly known as irq0, irq1 and so on) on a PC-compatible platform are connected to ISA and PCI device interrupts. None of these interrupt sources exist on the Intel® Xeon Phi™ coprocessor; instead the I/O APIC IRQs are connected to interrupts generated by the Intel® Xeon Phi™ coprocessor SBox (e.g., DMA channel interrupts, thermal interrupts, etc.).

4.2.6 Timer Hardware

Timer hardware devices like the programmable interval timer (PIT), the CMOS real time clock (RTC), the advanced configuration and power interface (ACPI) timer, and the high-precision event timer (HPET) commonly found on PC platforms are absent on the Intel® Xeon Phi™ coprocessor.

The lack of timer hardware means that the Intel® Xeon Phi™ coprocessor OS must use the LAPIC timer for all timekeeping and scheduling activities. It still needs a mechanism to calibrate the LAPIC timer which is otherwise calibrated using the PIT. It also needs an alternative solution to the continuously running time-of-day (TOD) clock, which keeps time in year/month/day hour:minute:second format. The Intel® Xeon Phi™ coprocessor has a SBox MMIO register that provides the current CPU frequency, which can be used to calibrate the LAPIC timer. The TOD clock has to be emulated in software to query the host OS for the time at bootup and then using the LAPIC timer interrupt to update it. Periodic synchronization with the host may be needed to compensate for timer drift.

4.2.7 Debug Store

The Intel® Xeon Phi™ coprocessor does not support the ability to write debug information to a memory resident buffer. This feature is used by Branch Trace Store (BTS) and Precise Event Based Sampling (PEBS) facilities.

4.2.8 Power and Thermal Management

4.2.8.1 Thermal Monitoring

Thermal Monitoring of the Intel® Xeon Phi™ coprocessor die is implemented by a Thermal Monitoring Unit (TMU). The TMU enforces throttling during thermal events by reducing core frequency ratio. Unlike TM2 thermal monitoring on other Intel processors (where thermal events result in throttling of both core frequency and voltage), the Intel® Xeon Phi™ coprocessor TMU does not automatically adjust the voltage. The Intel® Xeon Phi™ coprocessor TMU coordinates with a software-based mechanism to adjust processor performance states (P-states). The TMU software interface consists of a thermal interrupt routed through the SBox I/O APIC and SBox interrupt control and status MMIO registers. For more information on the TMU and its software interface refer to the section on Intel® Xeon Phi™ Coprocessor Power and Thermal Management.

4.2.8.2 ACPI Thermal Monitor and Software Controlled Clock Facilities

The processor implements internal MSRs (IA32_THERM_STATUS, IA32_THERM_INTERRUPT, IA32_CLOCK_MODULATION) that allow the processor temperature to be monitored and the processor performance to be modulated in predefined duty cycles under software control.

The Intel® Xeon Phi™ coprocessor supports non-ACPI based thermal monitoring through a dedicated TMU and a set of thermal sensors. Thermal throttling of the core clock occurs automatically in hardware during a thermal event. Additionally, OS power-management software is given an opportunity to modulate the core frequency and voltage in response to the thermal event. These core frequency and voltage settings take effect when the thermal event ends. In other words, Intel® Xeon Phi™ coprocessor hardware provides equivalent support for handling thermal events but through different mechanisms.

4.2.8.2.1 Enhanced SpeedStep (EST)

ACPI defines performance states (P-state) that are used to facilitate system software's ability to manage processor power consumption. EST allows the software to dynamically change the clock speed of the processor (to different P-states). The software makes P-state decisions based on P-state hardware coordination feedback provided by EST.

Again, the Intel® Xeon Phi™ coprocessor is not ACPI compliant. However, the hardware provides a means for the OS power-management software to set core frequency and voltage that corresponds to the setting of P-states in the ACPI domain. OS PM software in the Intel® Xeon Phi™ coprocessor (just as in the case of EST) dynamically changes the core frequency and voltage of the processor cores based on core utilization, thereby reducing power consumption. Additionally, the Intel® Xeon Phi™ coprocessor hardware provides feedback to the software when the changes in frequency and voltage take effect. This is roughly equivalent to what exists for EST; except that there is a greater burden on OS PM software to:

- generate a table of frequency/voltage pairs that correspond to P-states
- set core frequency and voltage to dynamically change P-states.

4.2.9 Pending Break Enable

The Intel® Xeon Phi™ coprocessor does not support this feature.

4.2.10 Global Page Tables

The Intel® Xeon Phi™ coprocessor does not support the global bit in Page Directory Entries (PDEs) and Page Table Entries (PTEs). Operating systems typically detect the presence of this feature using the CPUID instruction. This feature is enabled on processors that support it by writing to the PGE bit in CR4. On the Intel® Xeon Phi™ coprocessor, writing to this bit results in a GP fault.

4.2.11 CNXT-ID – L1 Context ID

Intel® Xeon Phi™ coprocessor does not support this feature.

4.2.12 Prefetch Instructions

The Intel® Xeon Phi™ coprocessor's prefetch instructions differ from those available on other Intel® processors that support the MMX™ instructions or the Intel® Streaming SIMD Extensions. As a result, the PREFETCH instruction is not supported. This set of instructions is replaced by VPREFETCH as described in the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)).

4.2.13 PSE-36

PSE-36 refers to an Intel processor feature (in 32-bit mode) that extends the physical memory addressing capabilities from 32 bits to 36 bits. The Intel® Xeon Phi™ coprocessor has 40 bits of physical address space but only supports 32 bits of physical address space in 32-bit mode. See also the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)).

4.2.14 PSN (Processor Serial Number)

The Intel® Xeon Phi™ coprocessor does not support this feature.

4.2.15 Machine Check Architecture

The Intel® Xeon Phi™ coprocessor does not support MCA as defined by the Intel® Pentium® Pro and later Intel processors. However, MCEs on the Intel® Xeon Phi™ coprocessor are compatible with the Intel® Pentium® processor.

4.2.16 Virtual Memory Extensions (VMX)

The Intel® Xeon Phi™ coprocessor does not support the virtualization technology (VT) extensions available on some Intel® 64 processors.

4.2.17 CPUID

The Intel® Xeon Phi™ coprocessor supports a highest-source operand value (also known as a CPUID leaf) of 4 for CPUID basic information, 0x80000008 for extended function information, and 0x20000001 for graphics function information.

4.2.17.1 Always Running LAPIC Timer

The LAPIC timer on the Intel® Xeon Phi™ coprocessor keeps ticking even when the Intel® Xeon Phi™ coprocessor core is in the C3 state. On other Intel processors, the OS detects the presence of this feature using the CPU ID leaf 6. The Intel® Xeon Phi™ coprocessor does not support this leaf so any existing OS code that detects this feature must be modified to support the Intel® Xeon Phi™ coprocessor.

4.2.18 Unsupported Instructions

For the details on supported and unsupported instructions, please consult the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)).

4.2.18.1 Memory Ordering Instructions

The Intel® Xeon Phi™ coprocessor memory model is the same as that of the Intel® Pentium processor. The reads and writes always appear in programmed order at the system bus (or the ring interconnect in the case of the Intel® Xeon Phi™ coprocessor); the exception being that read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cached hits and are, therefore, not directed to the same address being accessed by the read miss.

As a consequence of its stricter memory ordering model, the Intel® Xeon Phi™ coprocessor does not support the SFENCE, LFENCE, and MFENCE instructions that provide a more efficient way of controlling memory ordering on other Intel processors.

While reads and writes from an Intel® Xeon Phi™ coprocessor appear in program order on the system bus, the compiler can still reorder unrelated memory operations while maintaining program order on a single Intel® Xeon Phi™ coprocessor (hardware thread). If software running on an Intel® Xeon Phi™ coprocessor is dependent on the order of memory operations on another Intel® Xeon Phi™ coprocessor then a serializing instruction (e.g., CPUID, instruction with a LOCK prefix) between the memory operations is required to guarantee completion of all memory accesses issued prior to the serializing instruction before any subsequent memory operations are started.

4.2.18.2 Conditional Movs

Intel® Xeon Phi™ coprocessor does not support the Conditional Mov instructions. The OS can detect the lack of CMOVs using CPUID.

4.2.18.3 IN and OUT

The Intel® Xeon Phi™ coprocessor does not support IN (IN, INS, INSB, INSW, INSD) and OUT (OUT, OUTS, OUTSB, OUTSW, OUTSD) instructions. These instructions result in a GP fault. There is no use for these instructions on Intel® Xeon Phi™ coprocessors; all I/O devices are accessed through MMIO registers.

4.2.18.4 SYSENTER and SYSEXIT

The Intel® Xeon Phi™ coprocessor does not support the SYSENTER and SYSEXIT instructions that are used by 32-bit Intel processors (since the Pentium II) to implement system calls. However, the Intel® Xeon Phi™ coprocessor does support the SYSCALL and SYSRET instructions that are supported by Intel 64 processors. Using CPUID, the OS can detect the lack of SYSENTER and SYSEXIT and the presence of SYSCALL and SYSRET instructions.

4.2.18.5 MMX™ Technology and Streaming SIMD Extensions

The Intel® Xeon Phi™ coprocessor only supports SIMD vector registers that are 512 bits wide (zmm0-31) along with eight 16-bit wide vector mask registers.

The IA-32 architecture includes features by which an OS can avoid the time-consuming restoring of the floating-point state when activating a user process that does not use the floating-point unit. It does this by setting the TS bit in control register CR0. If a user process then tries to use the floating-point unit, a device-not-available fault (exception 7 = #NM) occurs. The OS can respond to this by restoring the floating-point state and by clearing CR0.TS, which prevents the fault from recurring.

The Intel® Xeon Phi™ coprocessor does not include any explicit instruction to perform context a save and restore of the Intel® Xeon Phi™ coprocessor state. To perform a context save and restore you can use:

- Vector loads and stores for vector registers
- A combination of vkmov plus scalar loads and stores for mask registers
- LDMXCSR and STMXCSR for MXCSR state register

4.2.18.6 Monitor and Mwait

The Intel® Xeon Phi™ coprocessor does not support the MONITOR and MWAIT instructions. The OS can use CPUID to detect lack of support for these.

MONITOR and MWAIT are provided to improve synchronization between multiple agents. In the implementation for the Intel® Pentium®4 processor with Streaming SIMD Extensions 3 (SSE3), MONITOR/MWAIT are targeted for use by system software to provide more efficient thread synchronization primitives. MONITOR defines an address range used to monitor write-back stores. MWAIT is used to indicate that the software thread is waiting for a write-back store to the address range defined by the MONITOR instruction.

FCOMI, FCOMIP, FUCOMI, FUCOM, FCMOVcc

The Intel® Xeon Phi™ coprocessor does not support these floating-point instructions, which were introduced after the Intel® Pentium® processor.

4.2.18.7 Pause

The Intel® Xeon Phi™ coprocessor does not support the pause instruction (introduced in the Intel® Pentium® 4 to improve its performance in spin loops and to reduce the power consumed). The Intel® Pentium® 4 and the Intel® Xeon® processors implement the PAUSE instruction as a pre-defined delay. The delay is finite and can be zero for some processors. The equivalent Intel® Xeon Phi™ coprocessor instruction is DELAY, which has a programmable delay. Refer to the programmer's manual for further details.

5 Application Programming Interfaces

5.1 The SCIF APIs

SCIF provides a mechanism for internode communication within a single platform, where a node is either an Intel® Xeon Phi™ coprocessor or the Xeon-based host processor complex. In particular, SCIF abstracts the details of communicating over the PCI Express* bus (and controlling related Intel® Xeon Phi™ coprocessors) while providing an API that is symmetric between the host and the Intel® Xeon Phi™ coprocessor. An important design objective for SCIF was to deliver the maximum possible performance given the communication abilities of the hardware.

The Intel® MPSS supports a computing model in which the workload is distributed across both the Intel® Xeon® host processors and the Intel® Xeon Phi™ coprocessor based add-in PCI Express* cards. An important property of SCIF is symmetry; SCIF drivers should present the same interface on both the host processor and the Intel® Xeon Phi™ coprocessor so that software written to SCIF can be executed wherever is most appropriate. SCIF architecture is operating system independent; that is, SCIF implementations on different operating systems are able to intercommunicate. SCIF is also the transport layer that supports MPI, OpenCL*, and networking (TCP/IP).

This section defines the architecture of the Intel® MIC Symmetric Communications Interface (SCIF). It identifies all external interfaces and each internal interface between the major system components.

The feature sets listed below are interdependent with SCIF.

- **Reliability Availability Serviceability (RAS) Support**
Because SCIF serves as the communication channel between the host and the Intel® Xeon Phi™ coprocessors, it is used for RAS communication.
- **Power Management**
SCIF must deal with power state events such as a node entering or leaving package C6.
- **Virtualization Considerations**
The Intel® Xeon Phi™ coprocessor product supports the direct assignment virtualization model. The host processor is virtualized, and each Intel® Xeon Phi™ coprocessor device is assigned exclusively to exactly one VM. Under this model, each VM and its assigned Intel® Xeon Phi™ coprocessor devices can operate as a SCIF network. Each SCIF network is separate from other SCIF networks in that no intercommunication is possible.
- **Multi-card Support**
The SCIF model, in principle, supports an arbitrary number of Intel® Xeon Phi™ coprocessor devices. The SCIF implementation is optimized for up to 8 Intel® Xeon Phi™ coprocessor devices.
- **Board Tools**
The Intel® MPSS ships with some software tools commonly referred to as “board tools”. Some of these board tools are layered on SCIF.

As SCIF provides the communication capability between host and the Intel® Xeon Phi™ coprocessors, there must be implementations of SCIF on both the host and the Intel® Xeon Phi™ coprocessor. Multisocket platforms are supported by providing each socketed processor with a physical PCI Express* interface. SCIF supports communication between any host processor and any Intel® Xeon Phi™ coprocessor, and between any two Intel® Xeon Phi™ coprocessors connected through separate physical PCI buses.

All of Intel® Xeon Phi™ coprocessor memory can be visible to the host or other Intel® Xeon Phi™ coprocessor devices. The upper 512GB of the Intel® Xeon Phi™ coprocessor’s physical address space is divided into 32 16-GB ranges that map through 32 corresponding SMPT registers to 16-GB ranges in host system address space. Each SMPT register can be programmed to any multiple of 16-GB in the host’s 64-bit address space. The Intel® Xeon Phi™ coprocessor accesses the host’s physical memory through these registers. It also uses these registers to access the memory space of other Intel® Xeon Phi™ coprocessor devices for peer-to-peer communication since Intel® Xeon Phi™ coprocessor memory is mapped

into the host address space. Thus, there is an upper limit of 512 GB to the host system memory space that can be addressed at any time. Up to seven SMPT registers (112 GB of this aperture) are needed to access the memory of seven other Intel® Xeon Phi™ coprocessor devices in a platform, for a maximum of 8 Intel® Xeon Phi™ coprocessor devices (assuming up to 16 GB per Intel® Xeon Phi™ coprocessor device). This leaves 25 SMPTs, which can map up to 400GB of host memory. Overall, as the number of Intel® Xeon Phi™ coprocessor devices within a platform increases, the amount of host memory that is visible to each Intel® Xeon Phi™ coprocessor device decreases.

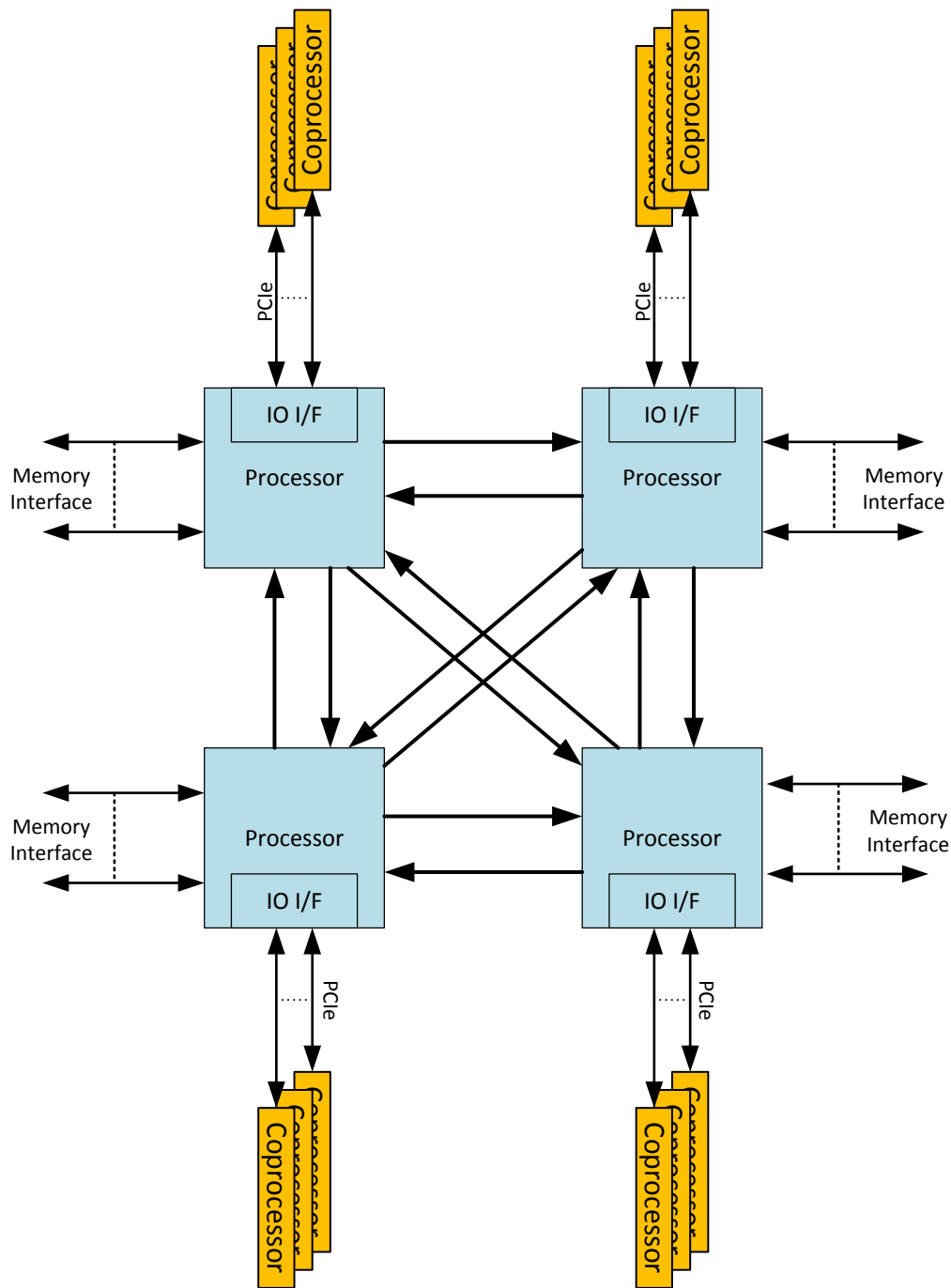


Figure 5-1. SCIP Architectural Model

Note that although SCIF supports peer-to-peer reads, the PCIe* root complex of some Intel client platforms do not.

The Intel® Xeon Phi™ coprocessor DMA engine begins DMAs on cache-line boundaries, and the DMA length is some multiple of the cache-line length (64B). Many applications need finer granularity. SCIF uses various software techniques to work compensate for this limitation. For example, when the source and destination base addresses are separated by a multiple of 64B, but do not begin on a cache-line boundary, the transfer is performed as unaligned “head” and “tail” read and write transfers (by the Intel® Xeon Phi™ coprocessor cores) and an aligned DMA “body” transfer. When the source and destination base addresses are not separated by a multiple of 64B, SCIF may first perform a local memory-to-memory copy of the buffer, followed by the head/body/tail transfer.

A SCIF implementation on a host or Intel® Xeon Phi™ coprocessor device includes both a user mode (Ring 3) library and kernel mode (Ring 0) driver. The user mode (Ring 3) library and kernel mode (Ring 0) driver implementations are designed to maximize portability across devices and operating systems. A kernel mode library facilitates accessing SCIF facilities from kernel mode. Subsequent subsections briefly describe the major components layered on SCIF.

The kernel-mode SCIF API is similar to the user mode API and is documented in the *Intel® MIC SCIF API Reference Manual for Kernel Mode Linux**. Table 5-1 is a snapshot summary of the SCIF APIs. In the table, μSCIF indicates a function in the user mode API, and kSCIF indicates a function in the kernel mode API. For complete details of the SCIF API and architectural design, please consult the *Intel® MIC SCIF API Reference Manual for User Mode Linux**.

Table 5-1 Summary of SCIF Functions

Group	Function	Mode
Connection	scif_open	μSCIF/kSCIF
	scif_close	μSCIF/kSCIF
	scif_bind	μSCIF/kSCIF
	scif_listen	μSCIF/kSCIF
	scif_connect	μSCIF/kSCIF
	scif_accept	μSCIF/kSCIF
Messaging	scif_send	μSCIF/kSCIF
	scif_recv	μSCIF/kSCIF
Registration and Mapping	scif_register	μSCIF/kSCIF
	scif_unregister	μSCIF/kSCIF
	scif_mmap	μSCIF
	scif_munmap	μSCIF
	scif_pin_pages	kSCIF
	scif_unpin_pages	kSCIF
	scif_register_pinned_pages	kSCIF
	scif_get_pages	kSCIF
	scif_put_pages	kSCIF
RMA	scif_readfrom	μSCIF/kSCIF
	scif_writeto	μSCIF/kSCIF
	scif_vreadfrom	μSCIF/kSCIF
	scif_vwriteto	μSCIF/kSCIF
	scif_fence_mark	μSCIF/kSCIF
	scif_fence_wait	μSCIF/kSCIF
	scif_fence_signal	μSCIF/kSCIF
Utility	scif_event_register	kSCIF
	scif_poll	μSCIF/kSCIF
	scif_get_nodeIDs	μSCIF/kSCIF
	scif_get_fd	μSCIF

The Connection API group enables establishing connections between processes on different nodes in the SCIF network, and employs a socket-like connection procedure. Such connections are point-to-point, connecting a pair of processes, and are the context in which communication between processes is performed.

The Messaging API group supports two-sided communication between connected processes and is intended for the exchange of short, latency-sensitive messages such as commands and synchronization operations.

The Registration API group enables controlled access to ranges of the memory of one process by a process to which it is connected. This group includes APIs for mapping the registered memory of a process in the address space of another process.

The RMA API group supports one-sided communication between the registered memories of connected processes, and is intended for the transfer of medium to large buffers. Both DMA and programmed I/O are supported by this group. The RMA API group also supports synchronization to the completion of previously initiated RMAs.

Utility APIs provide a number of utility services.

5.2 MicAccessAPI

The MicAccessAPI is a C/C++ library that exposes a set of APIs for applications to monitor and configure several metrics of the Intel® Xeon Phi™ coprocessor platform. It also allows communication with other agents, such as the System Management Controller if it is present on the card. This library is in turn dependent on *libscif.so*. This library is required in order to be able to connect to and communicate with the kernel components of the software stack. The *libscif.so* library is installed as part of Intel® MPSS. Several tools, including MicInfo, MicCheck, MicSmc & MicFlash all of which are located in `/opt/intel/mic/bin` after installing MPSS, rely heavily on MicAccessAPI.

Following a successful boot of the Intel® Xeon Phi™ coprocessor card(s), the primary responsibility of MicAccessAPI is to establish connections with the host driver and the coprocessor OS, and subsequently allow software to monitor/configure Intel® Xeon Phi™ coprocessor parameters. The host application and coprocessor OS communicate using messages, which are sent via the underlying SCIF architecture using the Sysfs mechanism as indicated in the figure below.

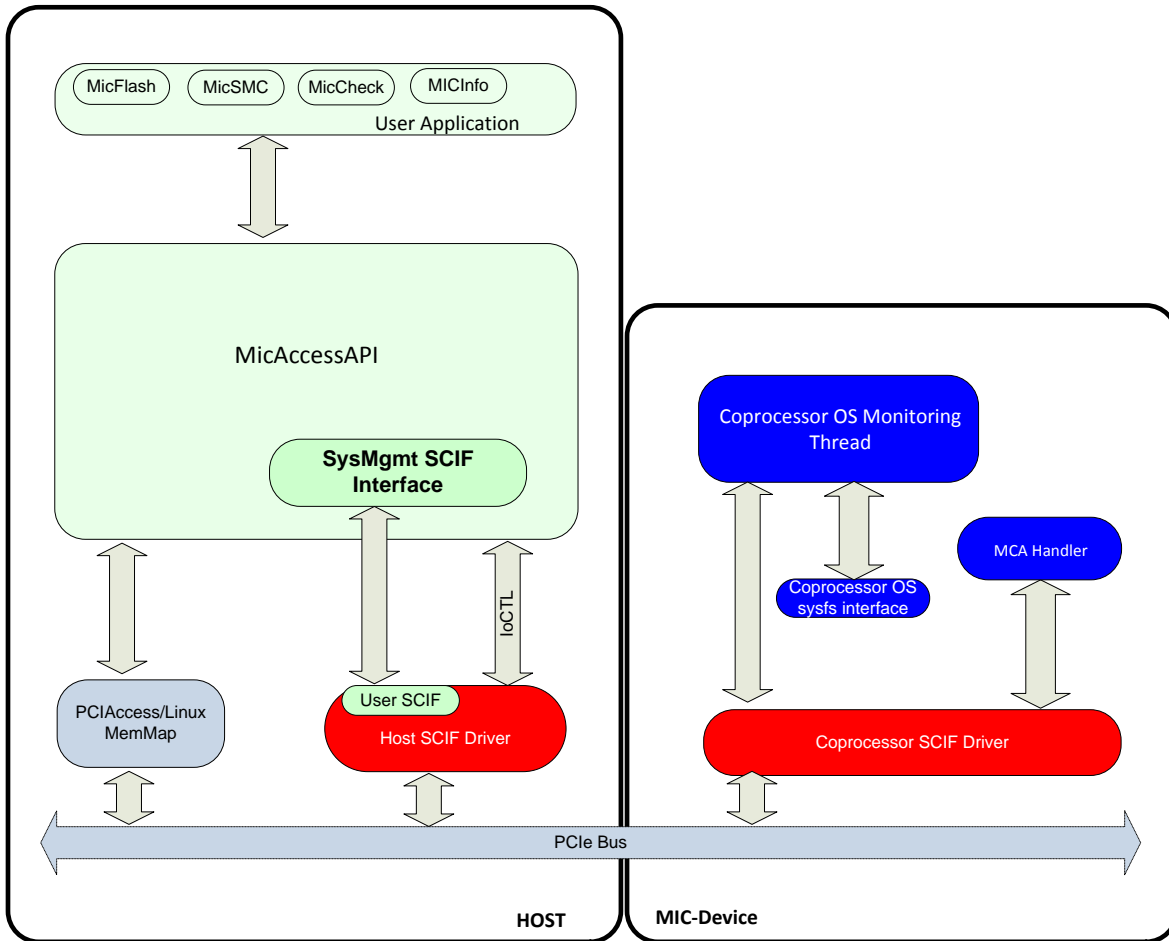


Figure 5-2 Intel® Xeon Phi™ Coprocessor SysMgmt MicAccessAPI Architecture Components Diagram

Another important responsibility of MicAccessAPI is to update the Flash & SMC. In order to be able to perform this update, the Intel® Xeon Phi™ coprocessor cards must be in the 'ready' mode. This can be accomplished using the 'micctrl' tool that comes with MPSS. The MicAccessAPI then enters into maintenance mode and interacts with the SPI Flash and the SMC's flash components via the maintenance mode handler to successfully complete the update process as shown in the figure below.

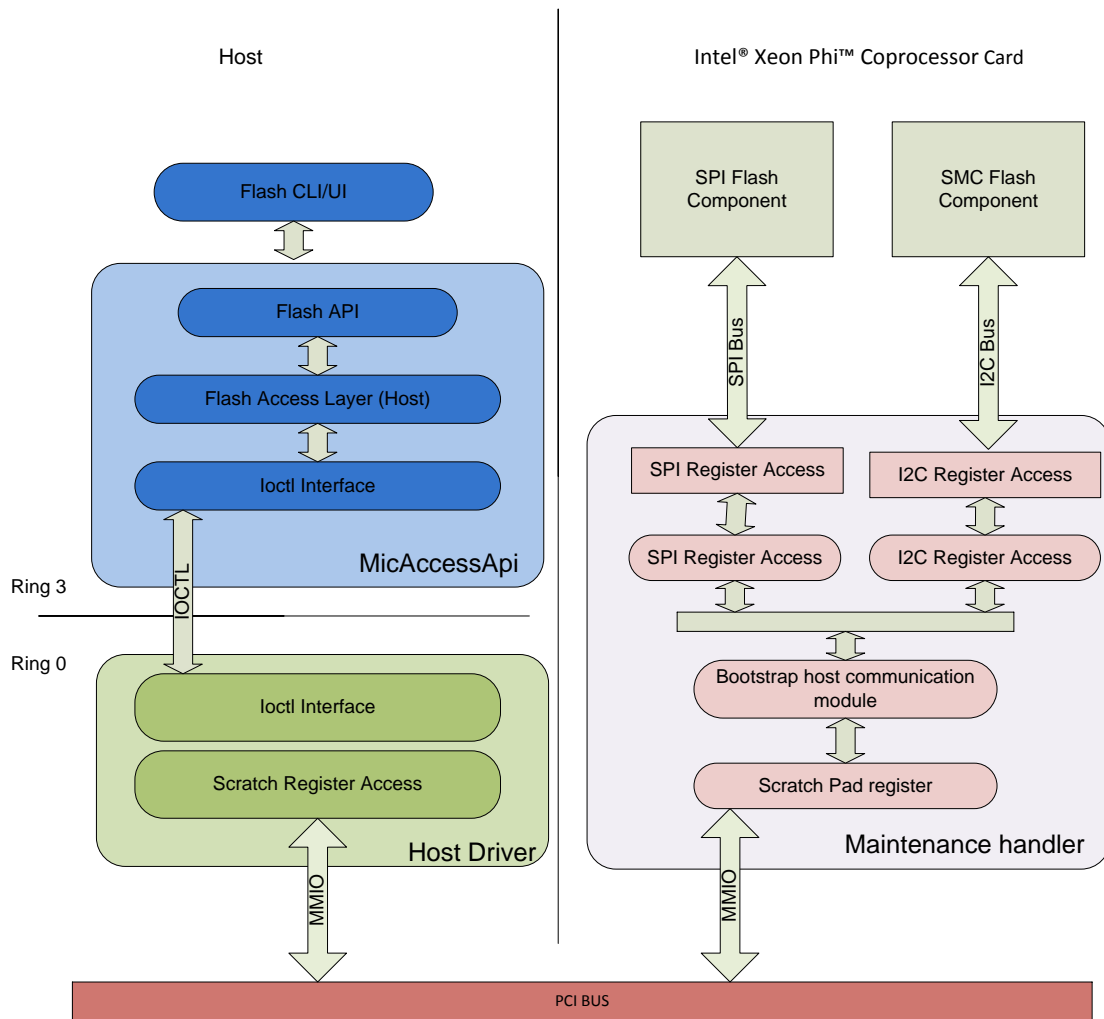


Figure 5-3 MicAccessAPI Flash Update Procedure

The various APIs included in the MicAccessAPI library can be classified into several broad categories as shown in Table 5-2.

Table 5-2. MicAccessAPI Library APIs

Group	API Name
Initialization	MicInitAPI, MicCloseAPI, MicInitAdapter, MicCloseAdapter
Flash	MicGetFlashDeviceInfo, MicGetFlashInfo, MicGetFlashVersion, MicUpdateFlash, MicSaveFlash, MicWriteFlash, MicAbortFlashUpdate, MicDiffFlash, MicFlashCompatibility, MicGetMicFlashStatus
Power management	MicPowerManagementStatus, MicGetPowerUsage, MicGetPowerLimit, MicPowerManagementEnable, MicResetMaxPower
SMC	MicGetSMCFWVersion, MicGetSMCHWRevision, MicGetUUID, MicLedAlert
Thermal	MicGetFanStatus, MicSetFanStatus, MicGetTemperature, MicGetFSCInfo, MicGetFrequency, MicGetVoltage
Memory	MicGetDevMemInfo, MicGetGDDRMemSize, MicGetMemoryUtilization, MicMapMemory, MicUnmapMemory, MicReadMem, MicWriteMem, MicReadMemPhysical, MicWriteMemPhysical, MicCopyGDDRTToFile
PCI	MicGetPcieLinkSpeed, MicGetPcieLinkWidth, MicGetPcieMaxPayload, MicGetPcieMaxReadReq
Core	MicGetCoreUtilization, MicGetNumCores
Turbo & ECC	MicGetTurboMode, MicDeviceSupportsTurboMode, MicEnableTurboMode, MicDisableTurboMode, MicGetEccMode, MicEnableEcc, MicDisableEcc
Exception	MicExceptionsEnableAPI, MicExceptionsDisableAPI, MicThrowException
General Card Information	MicGetDeviceID, MicGetPostCode, MicGetProcessorInfo, MicGetRevisionID, MicGetSiSKU, MicGetSteppingID, MicGetSubSystemID, MicCheckUOSDownloaded, MicGetMicVersion, MicGetUsageMode, MicSetUsageMode, MicCardReset

5.3 Support for Industry Standards

The Intel® MPSS supports industry standards like OpenMP™, OpenCL*, MPI, OFED*, and TCP/IP.

OpenMP™ is supported as part of the Intel® Composer XE software tools suite for the Intel® MIC Architecture.

MPI standards are supported through OFED* verbs development. See Section 2.2.9.2 for OFED* support offered in the Intel® Xeon Phi™ coprocessor.

The support for the OpenCL* standard for programming heterogeneous computers consists of three components:

- Platform APIs used by a host program to enumerate compute resources and their capabilities.
- A set of Runtime APIs to control compute resources in a platform independent manner. The Runtime APIs are responsible for memory allocations, copies, and launching kernels; and provide an event mechanism that allows the host to query the status of or wait for the completion of a given call.
- A C-based programming language for writing programs for the compute devices.

For more information, consult the relevant specification published by the respective owning organizations:

- OpenMP™ (<http://openmp.org/>)
- OpenCL (<http://www.khronos.org/opencv/>)
- MPI (<http://www.mpi-forum.org/>)
- OFED* Overview (<http://www.openfabrics.org>)

5.3.1 TCP/IP Emulation

The NetDev drivers emulate an Ethernet device to the next higher layer (IP layer) of the networking stack. Drivers have been developed specifically for the Linux* and Windows* operating systems. The host can be configured to bridge the TCP/IP network (created by the NetDev drivers) to other networks that the host is connected to. The availability of such a TCP/IP capability enables, among other things:

- remote access to Intel® Xeon Phi™ coprocessor devices via Telnet or SSH
- access to MPI on TCP/IP (as an alternative to MPI on OFED*)
- NFS access to the host or remote file systems (see Section 0).

5.4 Intel® Xeon Phi™ Coprocessor Command Utilities

Table 5-3 describes the utilities that are available to move data or execute commands or applications from the host to the Intel® Xeon Phi™ coprocessors.

Table 5-3. Intel® Xeon Phi™ Coprocessor Command Utilities

Utility	Description
micctrl	<ul style="list-style-type: none"> ▪ This utility administers various Intel® Xeon Phi™ duties including initialization, resetting and changing/setting the modes of any coprocessors installed on the platform. ▪ See the Intel® Xeon Phi™ Manycore Platform Software Stack (MPSS) Getting Started Guide (document number 513523) for details on how to use this tool.
micnativeloadex	<p>Uploads an executable and any dependent libraries:</p> <ul style="list-style-type: none"> ▪ from the host to a specified Intel® Xeon Phi™ coprocessor device ▪ from one Intel® Xeon Phi™ coprocessor device back to the host ▪ from one Intel® Xeon Phi™ coprocessor device to another Intel® Xeon Phi™ coprocessor device. <p>A process is created on the target device to execute the code. The application micnativeloadex can redirect (proxy) the process's file I/O to or from a device on the host. See the Intel® Xeon Phi™ Manycore Platform Software Stack (MPSS) Getting Started Guide (document number 513523) for details on how to use this tool.</p>

5.5 NetDev Virtual Networking

5.5.1 Introduction

The Linux* networking (see Figure 5-4) stack is made up of many layers. The application layer at the top consists of entities that typically run in ring3 (e.g., FTP client, Telnet, etc.) but can include support from components that run in ring0. The ring3 components typically use the services of the protocol layers via a system call interface like sockets. The device agnostic transport layer consists of several protocols including the two most common ones – TCP and UDP. The transport layer is responsible for maintaining peer-to-peer communications between two endpoints (commonly identified by ports) on the same or on different nodes. The Network layer (layer 3) includes protocols such as IP, ICMP, and ARP; and is responsible for maintaining communication between nodes, including making routing decisions. The Link layer (layer 2) consists of a number of protocol agnostic device drivers that provide access to the Physical layer for a

number of different mediums such as Ethernet or serial links. In the Linux* network driver model, the Network layer talks to the device drivers via an indirection level that provides a common interface for access to various mediums.

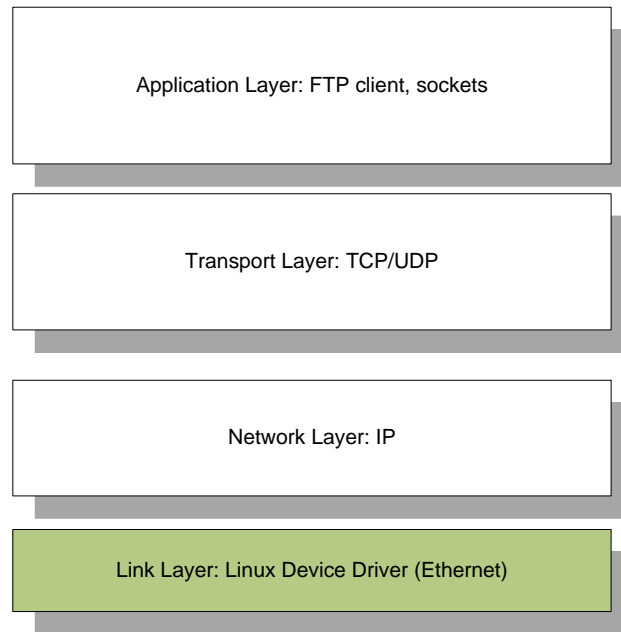


Figure 5-4 Linux* Network Stack

The focus of this section is to describe the virtual Ethernet driver that is used to communicate between various nodes in the system, including between cards. The virtual Ethernet driver sends and receives Ethernet frames across the PCI Express* bus and uses the DMA capability provided by the SBox on each card.

5.5.2 Implementation

A separate Linux* interface is created for each Intel® Xeon Phi™ coprocessor (mic0, mic1, and so on). It emulates a Linux* hardware network driver underneath the network stack on both ends. Currently, the connections are class C subnets local to the host system only. In the future, the class C subnets will be made available under the Linux* network bridging system for outside of host access.

During initialization, the following steps are followed:

1. Descriptor ring is created in host memory.
2. Host provides receive buffer space in the descriptor ring using Linux* skbuffs
3. Card maps to the host descriptor ring.
4. During host transmit, the host posts transmit skbuffs to the card OS in descriptor ring.
5. Card polls for changes in descriptor host transmit ring
6. Card allocates skbuff and copies host transmit data
7. Card sends new skbuff to card side TCP/IP stack.
8. At card transmit, card copies transmit skbuff to receive buffer provided at initialization.
9. Card increments descriptor pointer.
10. Host polls for changes in transmit ring.
11. Host sends receive buffer to TCP/IP stack.

In a future implementation, during initialization, Host will create a descriptor ring for controlling transfers, Host will allocate and post a number of receive buffers to the card, card will allocate and post a number of receive buffers to the host. At Host Transmit, Host DMA's data to receive skbuff posted by Intel® Xeon Phi™ coprocessor, Host interrupts card, Card interrupt routine sends skbuff to tcp/ip stack, card allocates and posts new empty buffer for host use. At Card

Transmit, Card DMAs data to receive skbuff posted by Host, Card interrupts host, Host interrupt routine sends skbuff to tcp/ip stack, Host allocates and posts new empty buffer for card use.

6 Compute Modes and Usage Models

The architecture of the Intel® Xeon Phi™ coprocessor enables a wide continuum of compute paradigms far beyond what is currently available. This flexibility allows a dynamic range of solution to address your computing needs – from highly scalar processing to highly parallel processing, and a combination of both in between. There are three general categories of compute modes supported with the Intel® Xeon Phi™ coprocessor, which can be combined to develop applications that are optimal for the problem at hand.

6.1 Usage Models

The following two diagrams illustrate the compute spectrum enabled and supported by the Intel® Xeon® processor-Intel® Xeon Phi™ coprocessor coupling. Depending on the application’s compute needs, a portion of its compute processes can either be processed by the Intel® Xeon® processor host CPUs or by the Intel® Xeon Phi™ coprocessor. The application can also be started or hosted by either the Intel® Xeon® processor host or by the Intel® Xeon Phi™ coprocessor. Depending on the computational load, an application will run within the range of this spectrum for optimal performance.

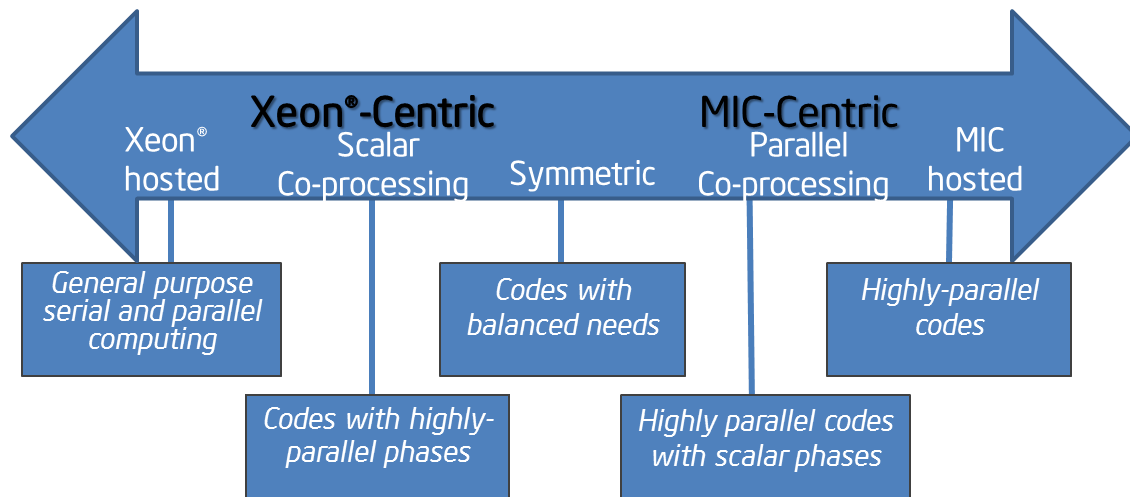


Figure 6-1 : A Scalar/Parallel Code Viewpoint of the Intel® MIC Architecture Enabled Compute Continuum

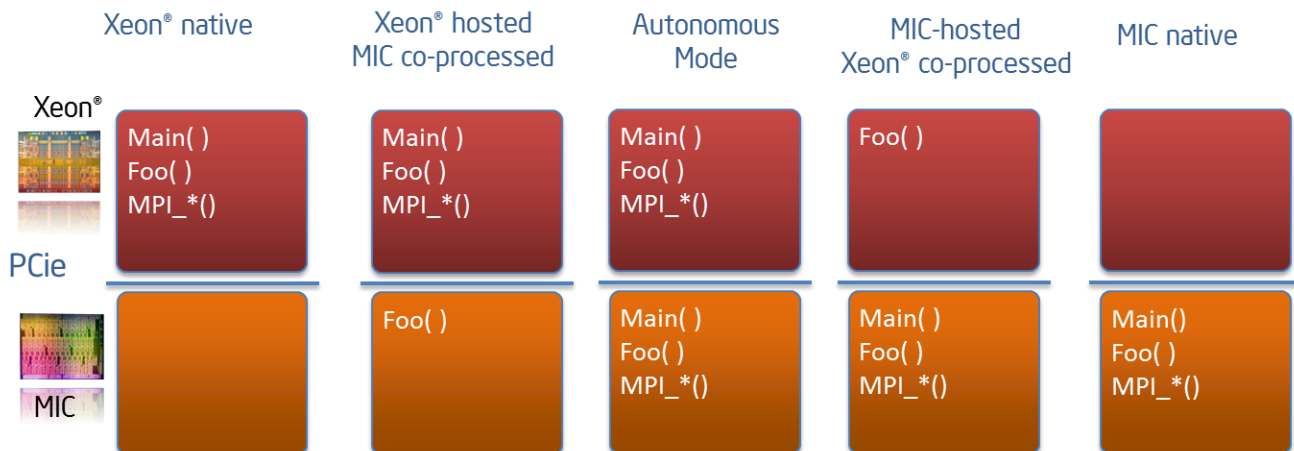


Figure 6-2: A Process Viewpoint of the Intel® MIC Architecture Enabled Compute Continuum

6.2 MPI Programming Models

The Intel® MPI Library for Intel® MIC Architecture plans to provide all of the traditional Intel® MPI Library features on any combination of the Intel® Xeon® and the Intel® Xeon Phi™ coprocessors. The intention is to extend the set of architectures supported by the Intel® MPI Library for the Linux* OS, thus providing a uniform program development and execution environment across all supported platforms.

The Intel® MPI Library for Linux* OS is a multi-fabric message-passing library based on ANL* MPICH2* and OSU* MVAPICH2*. The Intel® MPI Library for Linux* OS implements the Message Passing Interface, version 2.1* (MPI-2.1) specification.

The Intel® MPI Library for Intel® MIC Architecture supports the programming models shown in Figure 6-3.

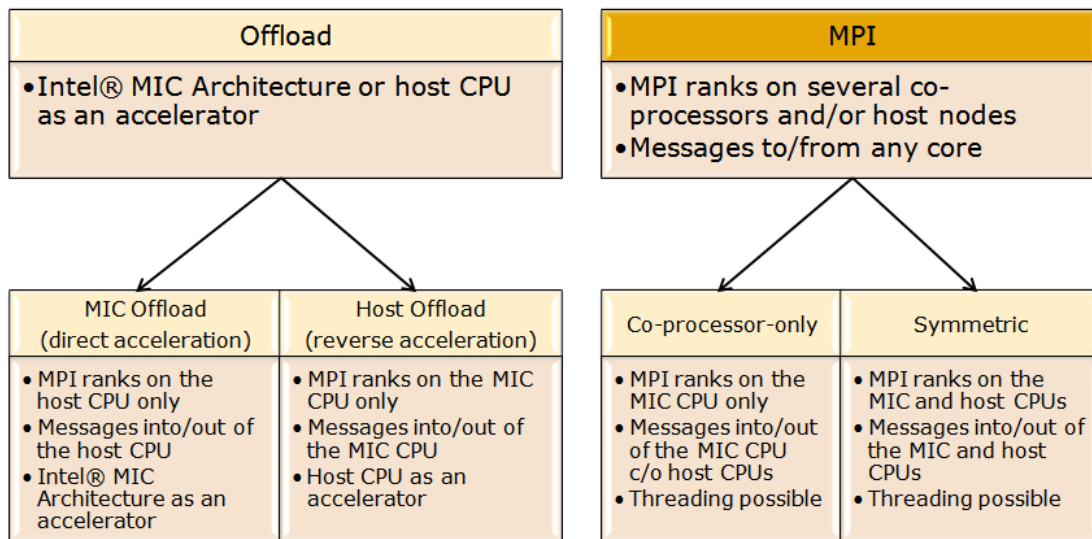


Figure 6-3: MPI Programming Models for the Intel® Xeon Phi™ Coprocessor

In the Offload mode, either the Intel® Xeon Phi™ coprocessors or the host CPUs are considered to be coprocessors. There are two possible scenarios:

1. Xeon® hosted with Intel® Xeon Phi™ coprocessors, where the MPI processors run on the host Xeon® CPUs, while the offload is directed towards the Intel® Xeon Phi™ coprocessors. This model is supported by the Intel® MPI Library for Linux* OS as of version 4.0. Update 3.
2. Intel® Xeon Phi™ coprocessor hosted with Xeon® coprocessing, where the MPI processes run on the Intel® Xeon Phi™ coprocessors while the offload is directed to the host Xeon® CPU.

Both models make use of the offload capabilities of the products like Intel® C, C++, Fortran Compiler for Intel® MIC Architecture, and Intel® Math Kernel Library (MKL). The second scenario is not supported currently due to absence of the respective offload capabilities in the aforementioned collateral products.

In the MPI mode, the host Xeon® CPUs and the Intel® Xeon Phi™ coprocessors are considered to be peer nodes, so that the MPI processes may reside on both or either of the host Xeon® CPUs and Intel® Xeon Phi™ coprocessors in any combination. There are three major models:

- **Symmetric model**
The MPI processes reside on both the host and the Intel® Xeon Phi™ coprocessors. This is the most general MPI view of an essentially heterogeneous cluster.

6.2.2 Coprocessor-Only Model

In this model (also known as the “Intel® MIC architecture native” model), the MPI processes reside solely inside the coprocessor. MPI libraries, the application, and other needed libraries are uploaded to the coprocessors. Then an application can be launched from the host or from the coprocessor.

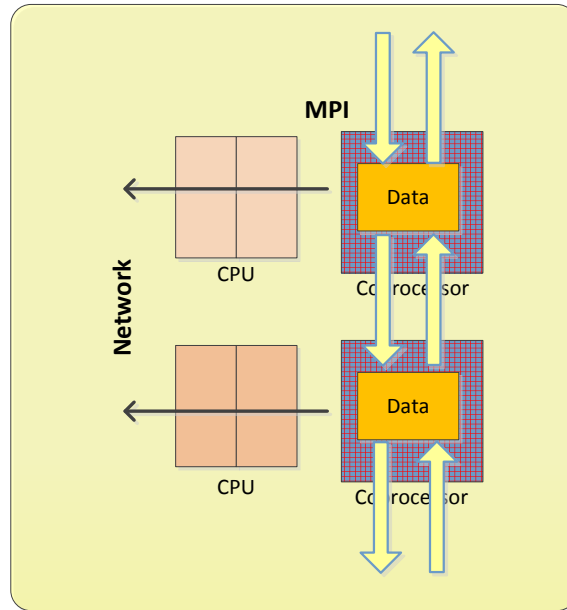


Figure 6-5: MPI on the Intel® Xeon Phi™ coprocessors Only

6.2.3 Symmetric Model

In this model, the host CPUs and the coprocessors are involved in the execution of the MPI processes and the related MPI communications. Message passing is supported inside the coprocessor, inside the host node, and between the coprocessor and the host via the shm and shm:tcp fabrics. The shm:tcp fabric is chosen by default; however, using shm for communication between the coprocessor and the host provides better MPI performance than TCP. To enable shm for internode communication, set the environment variable: `I_MPI_SSHM_SCIF={enable|yes|on|1}`.

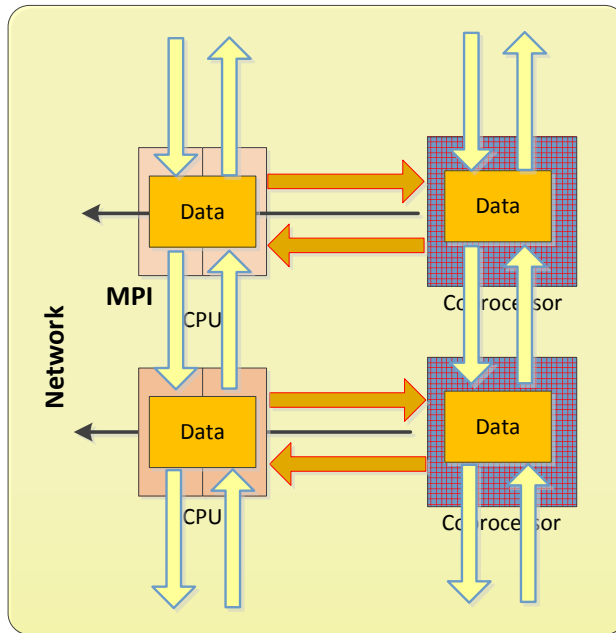


Figure 6-6: MPI Processes on Both the Intel® Xeon® Nodes and the Intel® MIC Architecture Devices

The following is an example of the symmetric model:

Symmetric model:

mpiexec.hydra is started on host,
 launches 4 processes on host with 4 threads in each process,
 and 2 processes on "mic0" coprocessor with 16 threads in each process

```
(host)$mpiexec.hydra -host $(hostname) -n 4 -env OMP_NUM_THREADS 4 ./test.exe.host: \
-host mic0 -n 2 -env OMP_NUM_THREADS 16 -wdir /tmp /tmp/test.exe.mic
```

6.2.4 Feature Summary

The Intel® MPI Library requires the presence of the /dev/shm device in the system. To avoid failures related to the inability to create a shared memory segment, the /dev/shm device must be set up correctly.

Message passing is supported inside the coprocessor, inside the host node, between the coprocessors, and between the coprocessor and the host via the shm and shm:tcp fabrics. The shm:tcp fabric is chosen by default.

The Intel® MPI Library pins processes automatically. The environment variable I_MPI_PIN and related variables are used to control process pinning. The number of the MPI processes is limited only by the available resources. The memory limitation may manifest itself as an 'lseek' or 'cannot register the bufs' error in an MPI application. The environment variable I_MPI_SSHM_BUFFER_SIZE set to a value smaller than 64 KB may work around this issue.

The current release of the Intel® MPI Library for Intel® MIC Architecture for Linux* OS does not support certain parts of the MPI-2.1 standard specification:

- Dynamic process management
- MPI file I/O
- Passive target one-sided communication when the target process does not call any MPI functions

The current release of the Intel® MPI Library for Intel® MIC Architecture for Linux* OS also does not support certain features of the Intel® MPI Library 4.0 Update 3 for Linux* OS:

- ILP64 mode
- gcc support
- IPM Statistic
- Automatic Tuning Utility
- Fault Tolerance
- mpiexec -perhost option

6.2.5 MPI Application Compilation and Execution

The typical steps of compiling an MPI application and executing it using mpiexec.hydra are canonically shown in Figure 6-7.

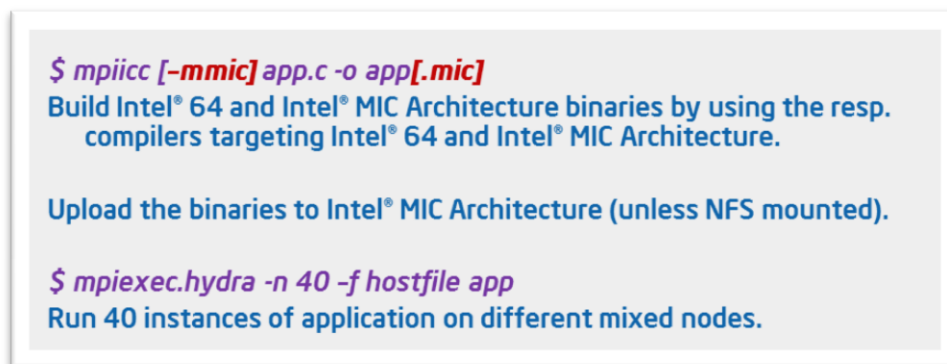


Figure 6-7. Compiling and Executing a MPI Application

For detailed information about installing and running Intel® MPI Library for Intel® MIC Architecture with the Intel® Xeon Phi™ coprocessors, please see the *Intel® Xeon Phi™ Coprocessor DEVELOPER'S QUICK START GUIDE*.

7 Intel® Xeon Phi™ Coprocessor Vector Architecture

7.1 Overview

The Intel® Xeon Phi™ coprocessor includes a new vector processing unit (VPU) with a new SIMD instruction set. These new instructions do not support prior vector architecture models like MMX™, Intel® SSE, or Intel® AVX.

The Intel® Xeon Phi™ coprocessor VPU is a 16-wide floating-point/integer SIMD engine. It is designed to operate efficiently on SOA (Structures of Array) data, i.e. [x0, x1, x2, x3, ..., x15], [y0, y1, y2, y3, ..., y15], [z0, z1, z2, z3, ..., z15], and [w0, w1, w2, w3, ..., w15] as opposed to [x0, y0, z0, w0], [x1, y1, z1, w1], [x2, y2, z2, w2], [x3, y3, z3, w3], ..., [x15, y15, z15, w15].

7.2 Vector State

The VPU brings with it a new architectural state, comprising vector general registers, vector mask registers, and a status register known as VXCSR, which mimics the Intel® SSE status register MXCSR in behavior. The new VPU architectural state is replicated four times, once for each hardware context in each core. The Intel® Xeon Phi™ coprocessor introduces 32 new vector registers, zmm0 through zmm31. Each vector register is sixty-four bytes wide (512 bits). The primary use of a vector register is to operate on a collection of sixteen 32-bit elements, or eight 64-bit elements. Figure 7-1 shows the new architectural state associated with the VPU.

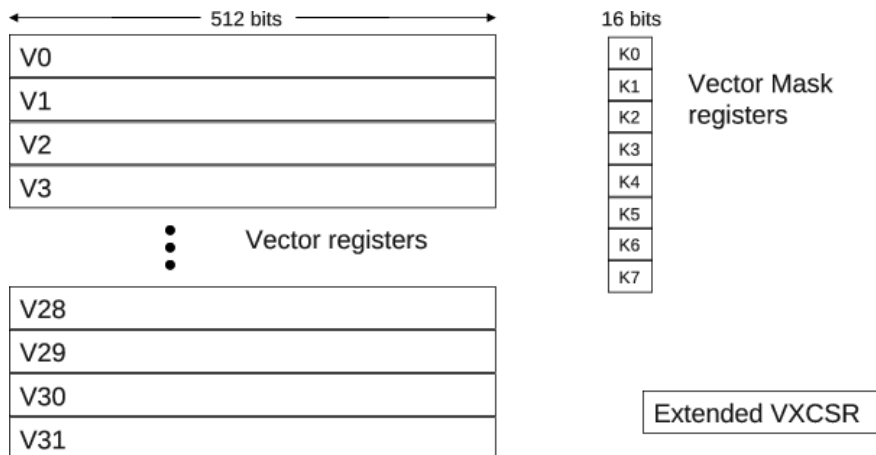


Figure 7-1: VPU Registers

The Intel® Xeon Phi™ coprocessor also introduces eight vector mask registers, denoted with a k prefix; that is, k0 through k7. Each mask register is sixteen bits wide and is used in a variety of ways, including write masking, carry/borrow flags, comparison results, and more.

7.3 VPU Basic Functions

The VPU is a SIMD vector engine that is fully pipelined and that executes Intel® Xeon Phi™ coprocessor vector instructions at a throughput of one instruction per cycle for most operations (If there are more than 2 threads working on a core, the decoder cannot take an instruction from one thread at each cycle because of back-to-back issues.). Most

VPU instructions use fully pipelined Multiply-Add (MADD) resources and get issued from the core U-pipe. Vector stores (except for scatter), mask operations, and a subset of integer instructions have duplicated or dedicated resources. Most instructions can also be issued from the core V-pipe to be paired with vector instructions issued down the U-pipe for execution in the VPU.

The VPU does not have any logic to handle stall or flush conditions, but it does contain logic to check for vector-register and mask-register dependencies. Dependency detection logic in the VPU sends a stall request to the core if the operands are not data-ready (freeze). Most vector instructions go through the vector pipeline exactly once, and its execution is completed in 6 cycles, namely VC1, VC2, V1, V2, V3, and V4.

The VPU reads/writes the DCache at cache-line granularity via a dedicated 512-bit bus; this allows the VPU to perform one read from and one write to memory per clock cycle. Reads from DCache go through load conversion and generic swizzling mux before getting sent to the ALUs. Writes go through store conversion and alignments before getting sent to the Store Commit Buffers which reside in the DCache. For register operands, the VPU uses data stored in the vector register file exclusively.

The major structures in the VPU are the Vector Register File (VRF), the Flag Registers (MRF), the Swizzle Muxes (SWZ), the SRGB Lookup Table (SLUT), and the Trans Lookup Table (TLUT).

7.4 VPU Data Types

The VPU contains 16 SP ALU and 8 DP ALU. The VPU contains a 32-bit mantissa data-path for SP ALU, a 54-bit mantissa data-path for DP ALU, and a 512-bit data-path for loads and stores. Data types are converted to and from 32-bit or 64-bit representation before and after execution respectively.

The VPU instructions support the following native data types:

- Packed 32-bit Integers (or dword)
- Packed 32-bit SP FP values
- Packed 64-bit Integers (or qword)
- Packed 64-bit DP FP values

The VPU instructions can be categorized into typeless 32-bit instructions (denoted by postfix “d”), typeless 64-bit instructions (denoted by postfix “q”), signed and unsigned int32 instructions (denoted by postfix “pi” and “pu” respectively), signed int64 instructions (denoted by postfix “pq”), and fp32 and fp64 instructions (denoted by postfix “ps” and “pd” respectively).

For arithmetic calculations, the VPU represents values internally using 32-bit or 64-bit two’s complement plus a sign bit (duplicate of the MSB) for signed integers and 32-bit or 64-bit plus a sign bit tied to zero for unsigned integers. This simplifies the integer datapath and eliminates the need to implement multiple paths for the integer arithmetic. The VPU represents floating-point values internally using signed-magnitude with exponent bias of 128 or 1024 to adhere to the IEEE basic single-precision or double-precision format.

The VPU supports the up-conversion/down-conversion of the data types to/from either 32-bit or 64-bit values in order to execute instructions in SP ALU or DP ALU (see Table 7-1).

Table 7-1 Bidirectional Up/Down Conversion Table

	Float32	Float64
SRGB8	Yes	No
Float10	Yes	No
Float11	Yes	No
Float16	Yes	No
Unorm2	Yes	No
Unorm10	Yes	No
Int32	Yes	Yes
UInt32	Yes	Yes
Float64	Yes	No

7.5 Extended Math Unit

The Extended Math Unit is added to provide hardware with transcendental fast approximations using Lookup Tables. Minimax quadratic polynomial approximation is used to compute single-precision FP transcendental functions. Lookup Tables are used to store coefficients C0, C1, and C2, to add a special truncated squarer, and to modify the existing Wallace tree in the multiplier to accommodate the calculation of each approximation. The goal of the EMU is to provide 1-cycle or 2-cycle throughput transcendental functions. Specifically, the hardware will provide elementary functions: reciprocal (1/X, recip), reciprocal square root (1/ \sqrt{X} , rsqrt), base 2 exponential (2^X, exp2), and logarithm base 2 (log2). Other transcendental functions can be derived from elementary functions: division (div) using recip and multiplication (mult), square root (sqrt) using rsqrt and mult, exponential base 10 (exp) using exp2 and mult, logarithm base B (logB) using log2 and mult, and power (pow) using log2, mult, and exp2. Table 7-2 shows the projected latency (in terms of number of instructions) of elementary and derived functions.

Table 7-2 Throughput Cycle of Transcendental Functions

Elementary Functions	Latency	Derived Functions	Latency
RECIP	1	DIV	2
RSQRT	1	SQRT	2
EXP2	2	POW	4
LOG2	1		

The EMU is a fully-pipelined design. Look-up Table access happens in parallel with squarer computation. Polynomial terms are computed and accumulate in a single pass through the Wallace tree, without resource contention. Exponent goes through special setup logic, and then is combined with mantissa path polynomial evaluation. The resulting exponent and mantissa then flow through rest of the logic for normalization and rounding.

7.6 SP FP Operations

IEEE754r requires basic arithmetic operations (add, subtract, multiply) to be accurate to 0.5 ULP (Unit in the Last Place). The Intel® Xeon Phi™ coprocessor SP FP hardware achieves 0.5 ULP for SP FP add, subtract, and multiply operations.

OpenCL* requires per-instruction rounding mode. The Intel® Xeon Phi™ coprocessor provides support for per-instruction rounding mode only for register-register arithmetic instructions. This is accomplished by setting the NT bit and using on-the-fly swizzle control bits as per-instruction rounding mode control bits. For details please see the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)).

DX10/11 requires basic arithmetic operations (add, subtract, multiply) to be accurate to 1 ULP and, as previously indicated, the Intel® Xeon Phi™ coprocessor SP FP hardware is capable of providing results that are accurate to 0.5 ULP for SP FP add, subtract, multiply, and FMA.

7.7 DP FP Operations

IEEE754r requires basic arithmetic operations (add, subtract, multiply) to be accurate to 0.5 ULP. Intel® Xeon Phi™ coprocessor DP FP hardware is capable of providing results that are accurate to 0.5 ULP for DP FP add, sub, and multiply. IEEE754r requires four rounding modes, namely, “Round to Nearest Even”, “Round toward 0”, “Round toward +∞”, and “Round toward -∞”. The Intel® Xeon Phi™ coprocessor supports “Round to Nearest Even”, “Round toward 0”, “Round toward +∞”, and “Round toward -∞” for DP FP operations via VXCSR. In addition, a new instruction VROUNDPD is added to allow you to set rounding mode on the fly.

To meet OpenCL* per-instruction rounding mode, support is provided for per-instruction rounding mode for register-register arithmetic instructions. This is accomplished by setting the NT bit and using on-the-fly swizzle control bits as per-instruction rounding mode control bits. For details please see the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)).

The Intel® Xeon Phi™ coprocessor also supports denormal numbers for DP FP in hardware for all float64 arithmetic instructions. Denormal number support includes input range checking and output data normalization.

7.8 Vector ISA Overview

Historically, SIMD implementations have a common set of semantic operations such as add, subtract, multiply, and so forth. Where most SIMD implementations differ lies in the specific number of operands to an operator, the nature of less common operations such as data permutations, and the treatment of individual elements contained inside a vector register.

Like Intel’s AVX extensions, the Intel® Xeon Phi™ coprocessor uses a three-operand form for its vector SIMD instruction set. For any generic instruction operator, denoted by `vop`, the corresponding Intel® Xeon Phi™ coprocessor instruction would commonly be:

$$vop :: zmm1, : zmm2, : zmm3$$

Where `zmm1`, `zmm2`, `zmm3` are vector registers, and `vop` is the operation (add, subtract, etc.) to perform on them. The resulting expression¹ would be:

$$zmm1 = zmm2 :: vop :: zmm3$$

Given that the Intel Architecture is a CISC design, the Intel® Xeon Phi™ coprocessor allows the second source operand to be a memory reference, thereby creating an *implicit memory load* operation in addition to the vector operation. The generic representation of using such a memory source is shown as:

$$vop :: zmm1, : zmm2, ; [ptr]$$

$$zmm1 = zmm2 :: vop :: MEM[ptr]$$

Any memory reference in the Intel® Xeon Phi™ coprocessor instruction set conforms to standard Intel Architecture conventions, so it can be a direct pointer reference (`[rax]`) or an indirect (`[rbp] + [rcx]`); and can include an immediate offset, scale, or both² in either direct or indirect addressing form.

¹ This is true for two-operand operators, such as arithmetic + or ×. For those operators that require additional operands, such as carry-propagate instructions or fused multiply-add, a different form is used.

² e.g., An address could be of the form `[rbp]+([rax]*2)+0x443C0000`.

While these basics are relatively straightforward and universal, the Intel® Xeon Phi™ coprocessor introduces new operations to the vector instruction set in the form of *modifiers*. The mask registers can be understood as one type of modifier, where most vector operations take a mask register to use as a write-mask of the result:

$$vop ::= zmm1: \{k1\}, : zmm2, : zmm3/ptr$$

In the above expression, the specifier *k1* indicates that vector mask register number one is an additional source to this operation. The mask register is specified inside curly brackets {}, which indicates that the mask register is used as a write-mask register. If the vector register has COUNT elements inside it, then the interpretation of the write-mask behavior could be considered as:

```

for (i=0; i<COUNT; i++           1
{
if (k1[i] == 1)
zmm1[i] = zmm2 vop zmm3/MEM[ptr]  4
}

```

The key observation here is that the write-mask is a non-destructive modification to the destination register; that is, where the write-mask has the value 0 no modification of the vector register’s corresponding element occurs in that position. Where the mask has the value 1, the corresponding element of the destination vector register is replaced with the result of the operation indicated by *vop*. Write-masking behavior is explored in more detail later Section 7.10.

Another modifier argument that may be specified on most SIMD vector operations is a *swizzle*, although the specific swizzle behavior is determined by whether the arguments are from registers or memory. The first type of swizzle is only permitted when all operands to the vector operation are registers:

$$vop ::= zmm1: [: \{k1\} :], : zmm2, : zmm3: [: \{swizzle\} :]$$

Here, square brackets [:] denote that the write-mask and the swizzle are optional modifiers of the instruction. The actual meaning of the swizzle, also denoted with curly brackets {} (just as write-masks are denoted), is explained in depth in Section 7.12. Conceptually, an optional swizzle modifier causes the second source argument to be modified via a data pattern shuffle *for the duration of this one instruction*. It does not modify the contents of the second source register, it only makes a temporary copy and modifies the temporary copy. The temporary copy is discarded at the end of the instruction.

The swizzle modifier that the Intel® Xeon Phi™ coprocessor supports has an alternate form when used with the *implicit load* form. In this form, the swizzle acts as a *broadcast* modifier of the value loaded from memory. This means that a subset of memory may be read and then replicated for the entire width of the vector architecture. This can be useful for vector expansion of a scalar, for repeating pixel values, or for common mathematical operations.

One subtle aspect of the Intel® Xeon Phi™ coprocessor design is that each vector register is treated as though it entirely contains either 32-bit or 64-bit elements. Figure 7-2 depicts the organization of the vector register when working with 32-bit data.

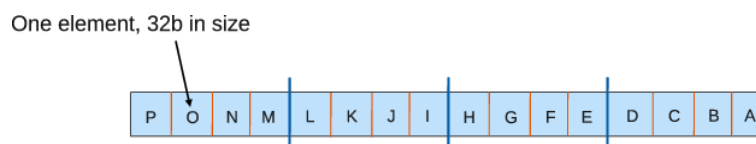


Figure 7-2. Vector Organization When Operating on 16 Elements of 32-bit Data

When executing an Intel® Xeon Phi™ coprocessor vector instruction, all arithmetic operations are carried out at either 32-bit or 64-bit granularity. This means that, when manipulating data of a different native size such as a two-byte float called `float16`, a different mathematical result might be obtained than if the operation were carried out with native

float16 hardware. This can cause bit-differences between an expected result and the actual result, triggering violations of commutatively or associativity rules.

Intel® Xeon Phi™ coprocessor includes IEEE 754-2008 [(Institute of Electrical and Electronics Engineers Standard for Floating Point Arithmetic, 2008)]-compliant, fused multiply-add (FMA) and fused multiply-subtract (FMS) instructions as well. These instructions produce results that are accurate to $0.5ulp^3$ (one-rounding) as compared to separate multiply and add instructions back-to-back as well as the “fused” multiply-add instructions of other architectures that produce results of $1.0ulp$ (two-rounding). In the case of Intel® Xeon Phi™ coprocessor’s fused instructions, the basic three-operand instruction form is interpreted slightly differently:

$$zmm1 = zmm1::vop1::zmm2::vop2::zmm3$$

FMA operations, for example, may have vop_1 set to \times and vop_2 set to $+$. The reality is richer than this. As mentioned previously, the ability to perform implicit memory loads for the second source argument $zmm3$, or to apply swizzles, conversions, or broadcasts to the second source argument, allows a wider range of instruction possibilities. In the presence of FMA and FMS, however, this restriction may lead to cumbersome workarounds to place the desired source field as the second source in the instruction.

Therefore, the Intel® Xeon Phi™ coprocessor instruction set provides a series of FMA and FMS operations, each one numbered in a sequence of three digits to the order field interpretation. This allows you to use the modifiers without knowing the particulars of the features. For example, the FMA operation for 32-bit floating-point data comes with these variants: $vmadd132ps$, $vmadd213ps$, and $vmadd231ps$. The logical interpretation⁴ is seen from the numeric string embedded in each mnemonic:

$$\begin{aligned} vfmadd132ps::zmm1,zmm2,zmm3 &: zmm1=zmm1\times zmm3+zmm2 \\ vfmadd213ps::zmm1,zmm2,zmm3 &: zmm1=zmm2\times zmm1+zmm3 \\ vfmadd231ps::zmm1,zmm2,zmm3 &: zmm1=zmm2\times zmm3+zmm1 \end{aligned}$$

Memory loads, modifiers such as swizzle, conversion, or broadcast, are only applicable to the $zmm3$ term. By selecting a mnemonic, you can apply the modifiers to different locations in the functional expression.

The Intel® Xeon Phi™ coprocessor also introduces a special fused multiply-add operation that acts as a *scale and bias* transformation in one instruction: $vfmadd233ps$. The interpretation of this instruction is best summarized in a series of equations. So, the $vfmadd233ps$ of the form $vfmadd233ps::\vec{z},\vec{u},\vec{v}$ generates the following:

$$\begin{aligned} \vec{z} [3..0] &= \vec{u} [3..0]\times\vec{v} [1]+\vec{v} [0] \\ \vec{z} [7..4] &= \vec{u} [7..4]\times\vec{v} [5]+\vec{v} [4] \\ \vec{z} [11..8] &= \vec{u} [11..8]\times\vec{v} [9]+\vec{v} [8] \\ \vec{z} [15..12] &= \vec{u} [15..12]\times\vec{v} [13]+\vec{v} [12] \end{aligned}$$

To make this example more concrete, the operation $vfmadd233ps zmm1, zmm2, zmm3$ results in the destination $zmm1$ values shown in Table 7-3.

Table 7-3. The Scale-and-Bias Instruction $vfmadd233ps$ on 32-bit Data

$zmm1[0] = zmm2[0] \times zmm3[1] + zmm3[0]$	$zmm1[1] = zmm2[1] \times zmm3[1] + zmm3[0]$
$zmm1[2] = zmm2[2] \times zmm3[1] + zmm3[0]$	$zmm1[3] = zmm2[3] \times zmm3[1] + zmm3[0]$

³ Unit in the Last Place (ulp), a measure of the accuracy of the least significant bit in a result.

⁴ For simplicity, the third operand is shown as vector register $v3$, although it could alternately be a memory reference.

$zmm1[4] = zmm2[4] \times zmm3[5] + zmm3[4]$	$zmm1[5] = zmm2[5] \times zmm3[5] + zmm3[4]$
$zmm1[6] = zmm2[6] \times zmm3[5] + zmm3[4]$	$zmm1[7] = zmm2[7] \times zmm3[5] + zmm3[4]$
$zmm1[8] = zmm2[8] \times zmm3[9] + zmm3[8]$	$zmm1[9] = zmm2[9] \times zmm3[9] + zmm3[8]$
$zmm1[10] = zmm2[10] \times zmm3[9] + zmm3[8]$	$zmm1[11] = zmm2[11] \times zmm3[9] + zmm3[8]$
$zmm1[12] = zmm2[12] \times zmm3[13] + zmm3[12]$	$zmm1[13] = zmm2[13] \times zmm3[13] + zmm3[12]$
$zmm1[14] = zmm2[14] \times zmm3[13] + zmm3[12]$	$zmm1[15] = zmm2[15] \times zmm3[13] + zmm3[12]$

The Intel® Xeon Phi™ coprocessor also introduces vector versions of the carry-propagate instructions (CPI). As with scalar Intel Architecture carry-propagate instructions, these can be combined together to support wider integer arithmetic than the hardware default. These are also building blocks for other forms of wide-arithmetic emulations. The challenge incurred in the vector version of these instructions (discussed in detail later on) is that a carry-out flag must be generated for each element in the vector. Similarly, on the propagation side, a carry-in flag must be added for each element in the vector. The Intel® Xeon Phi™ coprocessor uses the vector mask register for both of these: as a carry-out bit vector and as a carry-in bit vector.

There are many other additions to the Intel® Xeon Phi™ coprocessor instruction set, for use in both scalar and vector operations. Some are discussed later in this guide, and all may be found in (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364))

7.9 Vector Nomenclature

The microarchitecture design that implements the Intel® Xeon Phi™ coprocessor instruction set has certain artifacts that can affect how operations and source modifiers work. The remainder of this section explores these concepts in depth, focusing on the principles of supporting 32-bit data elements in vectors and pointing out microarchitecture limitations and their implications along the way.

Recall that each vector register is 64 bytes wide, or 512 bits. It is useful to consider the vector register as being subdivided into four *lanes*, numbered 3 ... 0, where each lane is 128 bits wide. This is shown graphically in Figure. Lanes are always referred to by their lane number.



Figure 7-3. Vector Register Lanes 3...0

While this bears a superficial resemblance to the prior Intel® SSE Architecture, which uses 128-bit vector registers, the actual implementation is quite different and the comparison is strongly discouraged. Viewing the Intel® Xeon Phi™ coprocessor from the perspective of SIMD Streaming Extensions will limit your comprehension of the capabilities that Intel® Xeon Phi™ coprocessor provides.

There are four 32-bit *elements*, in a 128-bit lane, identified by the letters *D...A*. Regardless of which lane, elements within a lane are always referred to by their element letter (as shown in Figure 7-4).

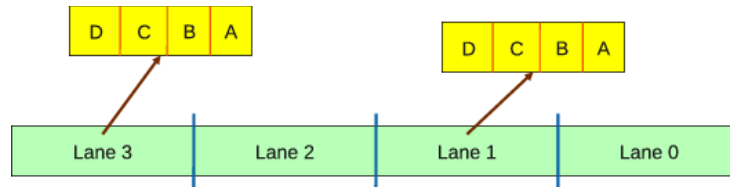


Figure 7-4. Vector Elements D...A Within a Lane

In contrast, when discussing all 16 elements in a vector, the elements are denoted by a letter from the sequence $P...A$, as shown in Figure 7-5..

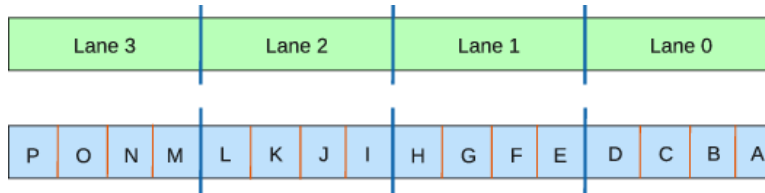


Figure 7-5. Vector Elements P...A Across the Entire Vector Register

The terms *lane* and *element*, as well as their enumerations, are standard usage in this guide.

In the memory storage form for vectors in the Intel® Xeon Phi™ coprocessor, the lowest address is always on the right-most side, and the terms are read right-to-left. Therefore, when loading a full vector from memory at location $0xA000$, the 32-bit vector elements $P...A$ are laid out accordingly:

Address	Element
$0xA000$	a
$0xA004$	b
\vdots	\vdots
$0xA03C$	p

7.10 Write Masking

The vector mask registers in Intel® Xeon Phi™ coprocessor are not general-purpose registers in the sense that they can be used for arbitrary operations. They support a small set of native operations (such as AND, OR, XOR) while disallowing many others (such as + or ×). This prevents the use of the vector mask registers as a form of 16-bit general-purpose register or as a scratch pad for temporary calculations. Although the Intel® Xeon Phi™ coprocessor allows load and store v-mask to general purpose register, general-purpose mathematical operations do not exist for the vector mask registers.

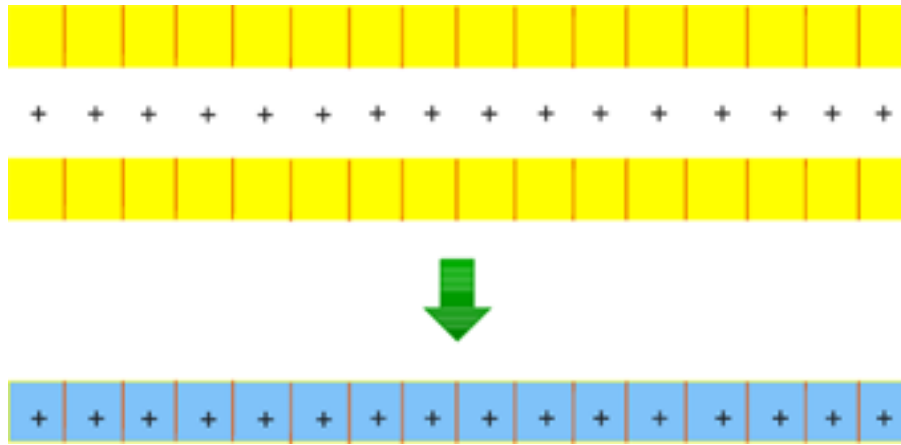


Figure 7-6. Basic Vector Operation Without Write Masking Specified

The vector mask registers are intended to control the update for vector registers inside a calculation. In a typical vector operation, such as a vector add, the destination vector is completely overwritten with the results of the operation. This is depicted in **Error! Reference source not found.**

The two source vectors (shown in yellow) combine to overwrite the destination vector (shown in blue). The write-mask makes the destination overwrite for each element *conditional*⁵, depending on the bit vector contents of a vector mask register. This is depicted in **Error! Reference source not found.**

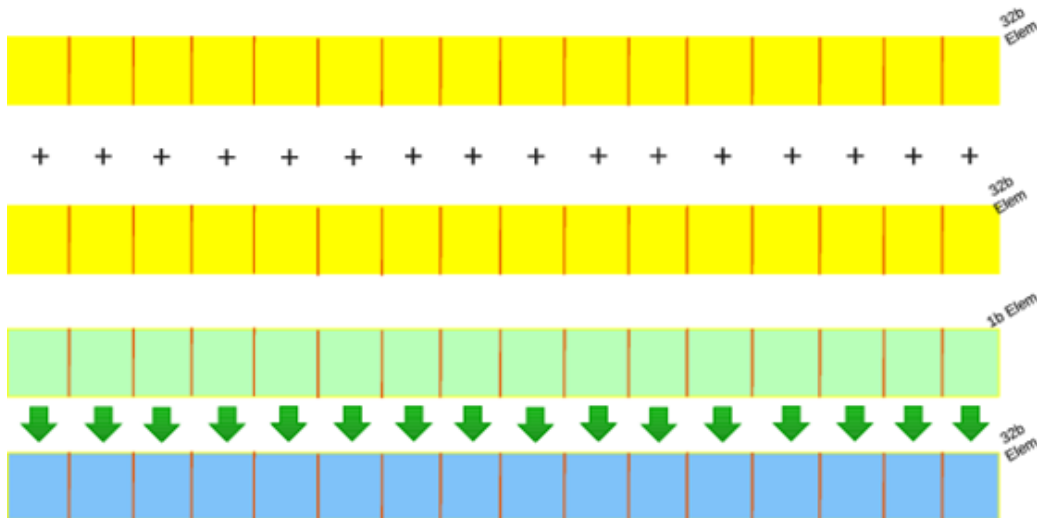


Figure 7-7. Basic Vector Operation With a Write Masking Specified

⁵ Alternately, think of making the per-element destination update *predicated* by the write mask.

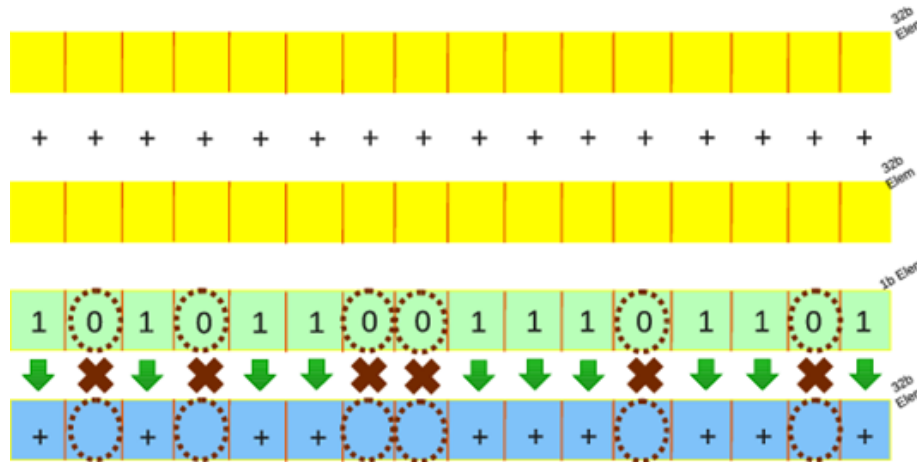


Figure 7-8. Effect of Write Mask Values on Result

Here, the write mask register (shown in green) has been added. Now each element in the destination vector is conditionally updated with the results of the vector operation, contingent on the corresponding element position bit in the mask register. Figure 7-8 shows how the write-mask values affect the results.

It is important to note that, unlike other Intel® Advanced Vector Extensions⁶, the write-mask behavior in the Intel® Xeon Phi™ coprocessor is *non-destructive* to the destination register. As shown in **Error! Reference source not found.**, where the write-mask register has the value zero, the destination register is not modified for that element. Prior Intel Architecture-based vector extensions were *destructive*; that is, when a mask value was set to zero, the destination element was cleared (all bits reset to zero).

Write mask support is present on *every* Intel® Xeon Phi™ coprocessor vector instruction; however, the specification is entirely optional. The reason is that for instructions which do *not* specify a mask register, a default value of $0 \times \text{FFFF}$ is implied and used.

Of special note is the mask register k0. The instruction encoding pattern that corresponds to mask register k0 is used to represent the default behavior when write masks are not specified (implying a default mask of $0 \times \text{FFFF}$). So specifying the “real” k0 as a write-mask register (which may not be equal to $0 \times \text{FFFF}$) is not allowed. Though this mask register *cannot* be used as a write mask register for Intel® Xeon Phi™ coprocessor vector instructions, it *can* be used for any of the other mask register purposes (i.e., carry propagate, comparison results, etc.). A simple rule of thumb to determine where k0 cannot be used is to examine the Intel® Xeon Phi™ coprocessor instruction set closely – any mask register specified inside of the curly braces { } cannot be designated as k0, while any other use can be k0.

Write masking can introduce false dependencies from a hardware scheduler perspective. Consider two different mask registers, k1 set to the value $0 \times \text{FF}00$ and k2 set to the value $0 \times 00\text{FF}$. If two vector operations are issued back-to-back with these mask registers, such as:

```
vaddps    zmm1:{k1},zmm2,zmm3
vaddps    zmm4:{k2},zmm5,zmm1
```

Then, for these instructions and assumed mask values, it is apparent that no true dependency exists in the register set. The dependency through zmm1 is false because the half of the vector register that is not overwritten by the first instruction is used as an input to the second instruction. The Intel® Xeon Phi™ coprocessor’s hardware schedule cannot examine the masks early enough in the pipeline to recognize this false dependency. From the scheduler’s perspective, *all write masks are considered to have the value $0 \times \text{FFFF}$* ⁷.

⁶ Such as Intel® Advanced Vector Extension (AVX).

⁷ The write mask is also not consulted for memory alignment issues.

Consider applying the simple Newton-Raphson root-finding approximation algorithm to a problem $f(x)$ as an example for how write masking is used in general:

$$x_{n+1} = x_n + \frac{f(x)}{f'(x)}$$

When this is implemented, the calculation kernel is an iterative algorithm that runs in a loop until the error term falls below a preselected target threshold. For whatever function is being evaluated, each successive iteration tests the value x_n to determine whether $|f(x_n) - f(x)| \leq \epsilon$.

Translating this behavior into use of the write mask is straightforward. Assuming 32-bit data values, the kernel would operate on 16 different versions of the algorithm concurrently. That is, if you are trying to compute $f(x)$ for the values a, b, \dots, p , then you start with the write mask set to a default value of all ones. At the end of each loop body, a vector comparison is done to test whether the computed values a_n, b_n, \dots, p_n are individually satisfying the constraint

$|f(x_n) - f(x)| > \epsilon$. The result of this vector comparison along with a reversal of the original comparison test ($\leq \epsilon$) will rewrite the mask register to have the value one only for those elements that have not converged to meet the desired threshold ϵ . An Intel® Xeon Phi™ coprocessor code sequence implementing a native version of the algorithm for approximating the square root of a number is shown in Figure 7-9. The modifier `{1to16}` used on source lines 8, 16 and 31 is explained in Section 7.11.3.1.

```

; reset mask k1 to 0xFFFF                                1
kxnor k1, k1

; load next 16 values                                    4
vmovaps zmm0, [data]

; load initial guess                                    7
vbroadcastss zmm1, [guess]

loop:                                                    10
; Generate a new guess for the square root

; compute (guess^2)                                     13
vmulps zmm2 {k1}, zmm1, zmm1
; compute (2.0 * guess)
vmulps zmm3 {k1}, zmm1, [two]{1to16}                  16
; compute (guess^2 - value)
vsubps zmm4 {k1}, zmm2, zmm0
; compute (zmm4/zmm3)                                   19
__vdivps zmm5 {k1}, zmm4, zmm3
; new guess = (old guess - zmm5)
vsubps zmm1 {k1}, zmm1, zmm5                           22

; find the amount of error in this new guess            25

; square of the new guess
vmulps zmm6 {k1}, zmm1, zmm1
; (guess^2 - value)                                     28
vsubps zmm7 {k1}, zmm6, zmm0
; -1.0f * (guess^2 - value)
vmulps zmm8 {k1}, zmm7, [neg1]{1to16}                 31
; error = abs| guess^2 - value |
vgmaxabsp zmm9 {k1}, zmm7, zmm8                        34

; check against epsilon, and loop if necessary

; k1 = error > epsilon                                  37
vcmpss k1 {k1}, zmm9, [epsilon]{1to16}, gt
; any elements to guess again?
kortest k1, k1                                         40
; if so, repeat
jnz loop

```

Figure 7-9. A Native (Non-Optimal) Newton-Raphson Approximation Using a Write-Mask to Determine a Square Root

The variables `data` and `guess` are symbolic, meaning that they would normally be an address calculation using architectural registers and displacements. The instruction `vdivps` is not a real instruction, but instead a library call using a template sequence as described later.

The vector mask test instruction `kortest` (line 40) reports whether any bits in the bit-wise OR of the operand mask registers are set to the value one, updating the ZF and CF bits in the EFLAG register based on the outcome. This instruction is typically followed by a conditional branch – here, the branch will re-execute the loop to compute x_{n+1} for those elements that have not yet converged. This use of the write mask may also be used as a mechanism for reducing total power consumption, as those elements which are masked off are (in general) not computed. In the case of the Newton-Raphson approximation; if half the vector elements have converged, then only half the vector ALUs will be used in the next iteration of the kernel. The remainder will be idle and will thereby consume less power. Other uses for the vector mask registers will be discussed in later.

7.11 Swizzling

Swizzles modify a source operand for the duration of a single instruction. Swizzles do not permanently alter the state of the source operand they modify. The proper conceptual model is that a swizzle makes a temporary copy of one input to an instruction, performs a data pattern combination on the temporary value, then feeds this altered temporary value to the ALU as a source term for one instruction. After the instruction is completed, the temporary value generated by the swizzle is discarded.

In effect, swizzles are a way to create data pattern combinations without requiring extra registers or instructions – as long as the desired combination is possible with the swizzle support at the microarchitectural level. Swizzles, like write masks, are optional arguments to instructions:

```
vop::zmm1[:{kl}:],:zmm2,:zmm3/ptr[:{swizzle}:]
```

While swizzles are powerful and useful, there are restrictions on the types of swizzles that may be used for a given instruction. These restrictions hinge upon the actual form of the instruction, but they also carry microarchitectural implications with them. In order to understand why swizzles work the way they do, it is useful to understand how the hardware support for swizzles works. A conceptual diagram of the swizzle logic is shown in Figure 7-10.

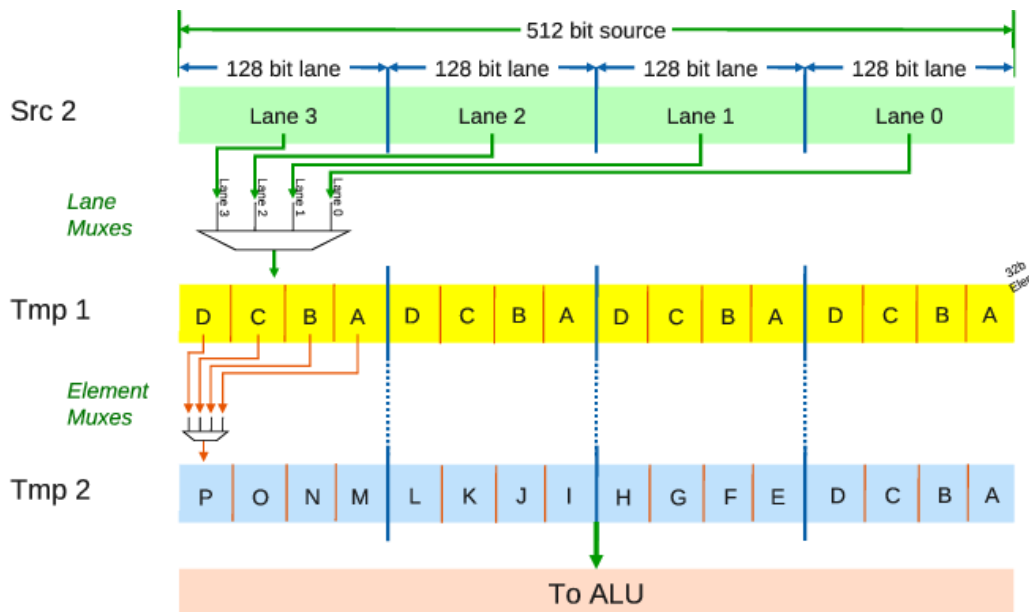


Figure 7-10. Partial Microarchitecture Design for Swizzle Support

The second source operand (shown in green) for a given instruction is the basis for all swizzle modifiers. This source is conceptually broken into the same 128-bit lanes as previously described. Each of the four lanes from the second source is presented to a multiplexor (mux) that is 128 bits wide.

Recall that the swizzle changes a source for the duration of the instruction by making a temporary copy (shown in yellow) of the source. The *lane mux* is capable of choosing which source lane is fed into which temporary copy lane. The lane multiplexors (muxes) allow for the remapping of any valid combination of source lanes into the temporary value lanes, such as taking the source lane pattern {3210} and creating a new lane pattern {3302}.

The first temporary value is then fed through a second set of data muxes to generate a second temporary value (shown in blue). The second set of muxes operates on the data elements within a lane from the first temporary value; and are,

therefore, called the *element muxes*. Element muxes are 32 bits wide and allow data pattern combinations within the lane. Taking a lane's source pattern in the first temporary value of the form {dcba}, the element muxes can be used to make a new combination such as {adbb}. The full implementation is shown in Figure 7-11 and depicts all the muxes in the swizzle logic.

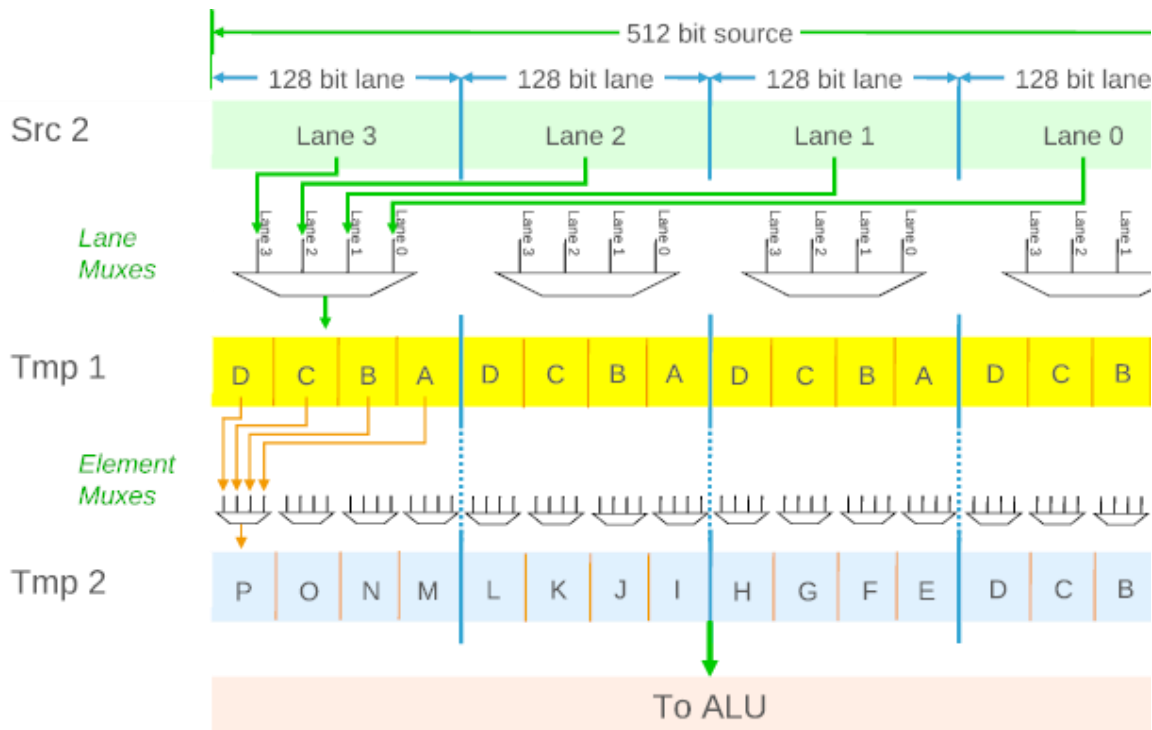


Figure 7-11. The Complete Microarchitecture Design for Swizzle Support

The second temporary value computed reflects the results of both lane muxes and the element muxes acting on the source value. This result is, in turn, fed to the ALU for the requested instruction. There are several important implications to be drawn from the mux network design that impact how the software should be written and that limit of the use of swizzles.

7.11.1 Swizzle Limitations

While all of the hardware capabilities of the swizzle logic previously described are fully supported by the shuffle instruction and the lane muxes are driven by some memory-form swizzles, general swizzles do not allow you to specify a lane mux pattern. In those cases, the default behavior is that the source lane is fed directly to the same lane in the first temporary value copy. The implication of this rule is that the *lane boundaries* are hard boundaries. The hardware cannot combine elements from different source lanes into the same lane at the conclusion of the swizzle. The only way to cross a lane boundary is through load or store operations, or through the vector shuffle instruction.

Second, to drive the element muxes completely would require 32 bits of immediate encoding on every instruction. This is clearly impractical, so the Intel® Xeon Phi™ coprocessor presents a choice of eight predefined swizzle patterns. This allows all swizzles to be represented in a three-bit field in every instruction's encoding. Whichever pattern is picked in a swizzle operation, *that pattern is applied across all lanes*.

When all operands to an instruction come from the vector register file, the choice of predefined swizzles to be applied are made from the *Register-Register* form, indicating that they operate on a source register. When the second source

argument comes from memory instead of from the vector register file, the available swizzles are from the *Register-Memory* form. Each type of swizzle behavior is explored below.

7.11.2 Register-Register Swizzle Form

The register-register form of the swizzle is only available when all operands to an instruction have vector register sources, conforming to the syntax:

$$vop::zmm1[:\{k1\}:],:zmm2,:zmm3[:\{swizzle\}:]$$

To illustrate the interpretation of the register-register swizzle form, we will walk through an example vector instruction: `vorpd zmm0 {k1}, zmm1, zmm2 {aaaa}`. This instruction is a bit-wise OR operation, using signed 32-bit integers. The destination `zmm0` will be modified on a per-element basis, with that control specified by the write mask register `k1` values. The two sources to the OR operator are the vector registers `zmm1` and `zmm2`. Figure 7-12 shows the initial evaluation of this instruction and the swizzle logic.

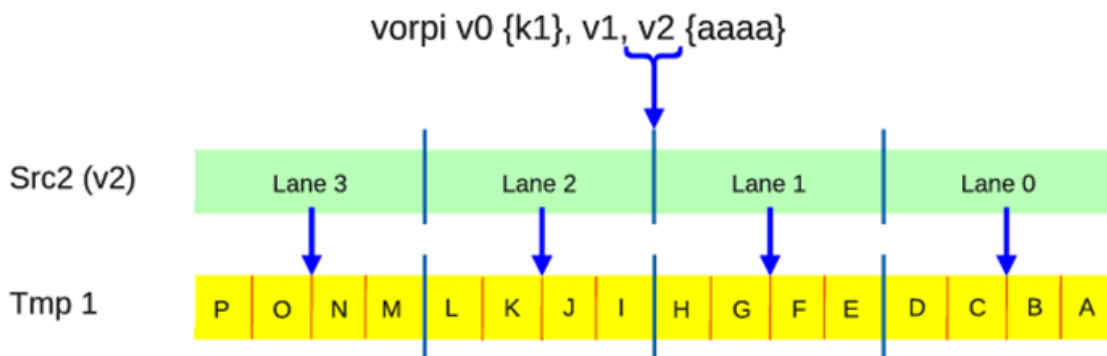


Figure 7-12. Register-Register Swizzle Operation: Source Selection

As depicted, the second source vector `zmm2` becomes the input to the swizzle logic. Since swizzles cannot drive the lane muxes, the entire source is pushed straight through the first temporary value stage in preparation for driving the element muxes. The swizzle field, denoted by `{aaaa}` will directly drive the element muxes. Figure 7-13 shows the element muxes that will be configured.

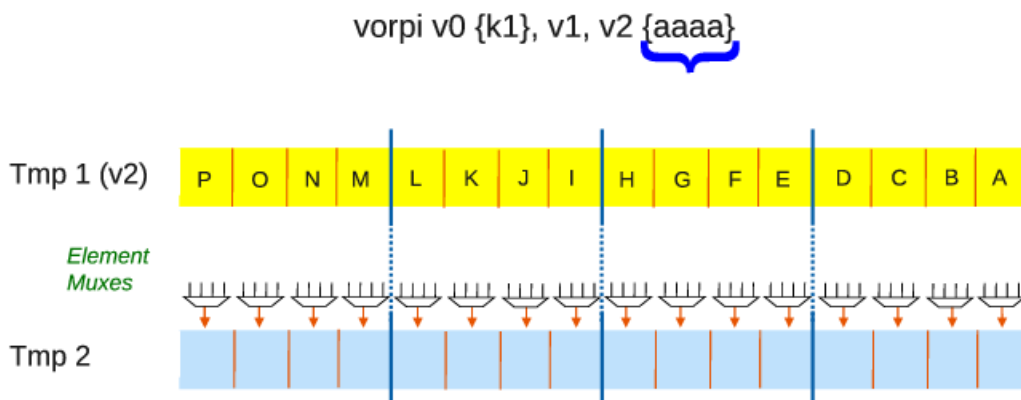


Figure 7-13. Register-Register Swizzle Operation: Element Muxes

As with the vector element enumeration, the swizzle is read right-to-left in order to drive the element muxes. The right-most designator of the swizzle (`a` in this case) indicates that the first element in the final destination should be set equal to the designated swizzle selection (`a`) element source of the lane. The second-most designator of the swizzle (`a` again)

indicates that the second element in the lane should be set to the designated swizzle element (a). This is repeated across the entire lane, as shown in Figure 7-14.

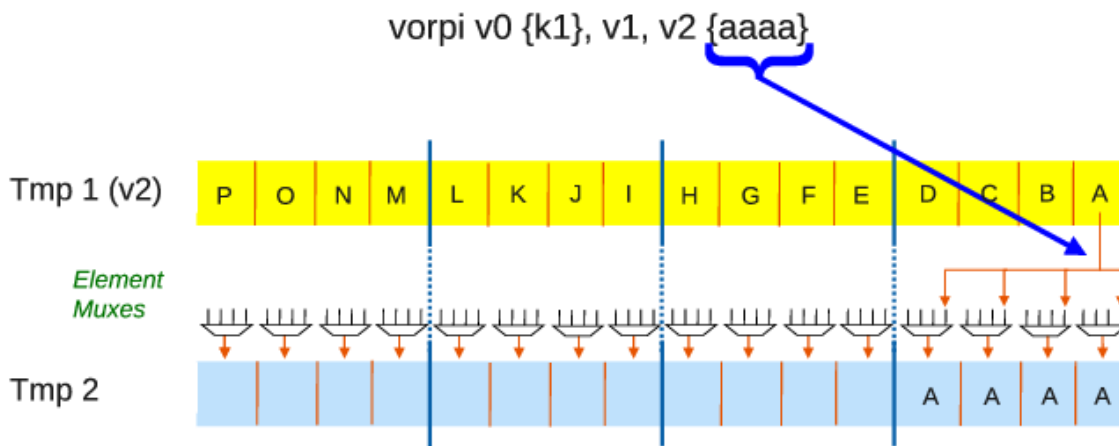


Figure 7-14. Register-Register Swizzle Operation: First Lane Completion

Remember, whatever swizzle pattern the instruction specifies drives the element muxes *across all lanes* in the same pattern. So, each lane driven has the same result, although the actual per-lane data values differ. Figure 7-15 shows the final result of the swizzle applied to the second source argument, with the final value driven to the ALU for the full instruction to operate on – in this example, the `vorpd` instruction.

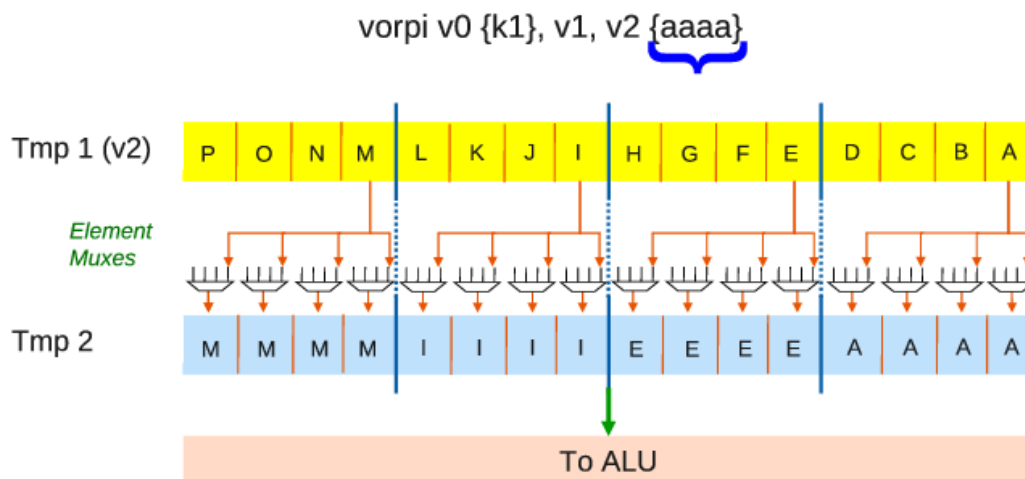


Figure 7-15. Register-Register Swizzle Operation: Complete Swizzle Result

There are eight predefined swizzle patterns that operate in register-register mode: `aaaa`, `bbbb`, `cccc`, `dddd`, `dacb`, `badc`, `cdab`, and `dcba`. The first four patterns correspond to repeat element selections, and are useful in many operations such as scalar-vector arithmetic. The next three patterns are common interleaving patterns, corresponding loosely to interleave pairs, swap pairs, and rotate pairs. These three are applicable to a wide range of arithmetic manipulations, including cross products (matrix determinant) and horizontal operations (such as horizontal add). The last pattern (`dcba`) corresponds to a no-change pattern, where the data is not reordered. The no-change pattern is the default pattern if no swizzle argument is specified in an instruction.

7.11.3 Register-Memory Swizzle Form

The register-memory form of the swizzle is available for all implicit load operations, conforming to the syntax:

```
vop:::zmm1[:{k1}:],:zmm2,:ptr[:{swizzle}:]
```

In register-register swizzles, you specify data pattern combinations that are applied to a temporary copy of the second source argument. In register-memory swizzles, you specify either a data broadcast operation (data replication), or else a data conversion.

With the implicit in-line load operation form, it is not possible to specify *both* a conversion and a broadcast on the same instruction. In order to have both broadcast and conversion, as well as a larger range of conversion choices, a true vector load operation must be used.

7.11.3.1 Data Broadcast

The purpose of the data broadcast swizzle is to perform data replication. Instead of loading a full 64-byte vector width of data from the cache hierarchy, a subset of that data is loaded. This subset will then replicate itself an integral number of times until the full 64-byte width is achieved. This is useful for optimizing memory utilization into the cache hierarchy.

There are three predefined swizzle patterns that operate in register-memory mode when performing data broadcast operations: 1to16, 4to16, and 16to16. The last pattern (16to16) corresponds to a load all pattern, where the data is not broadcast. This is the default pattern if no swizzle argument is specified for an instruction that takes this form.

The interpretation of the swizzle designator is to read the two numbers involved directly. The first is the number of 32-bit elements to read from memory, while the second is the total number of 32-bit elements being populated. In the case of 1to16, exactly one 32-bit value is being read from memory, and it is being replicated 15 times to make a total of 16 values. This is depicted in Figure 7-16.

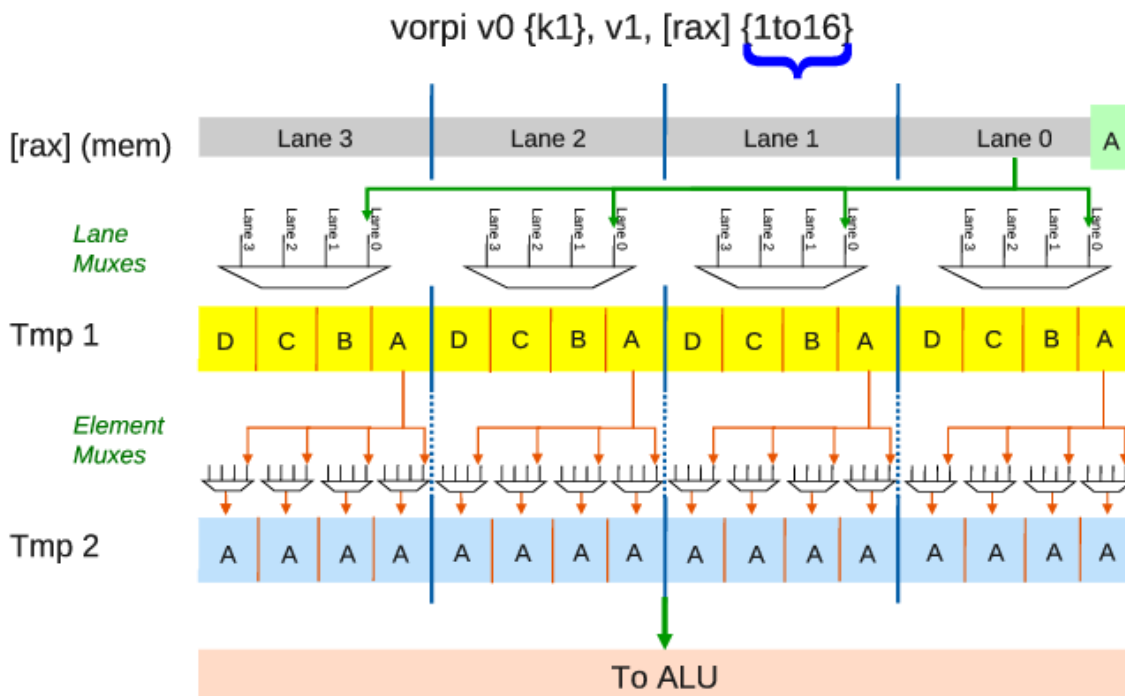


Figure 7-16. The 1-to-16 Register Memory Swizzle Operation

Note that, while you cannot choose lanes with the swizzles, the register-memory swizzle modifiers *conceptually* have the ability to modify lane mappings. The modifier `1to16` does just that in replicating the data loaded from memory. Thus lane zero is replicated to all lanes in the figure. Two common use cases for the `1to16` modifier are: performing a scalar-vector multiply and adding a constant to each element in a vector.

The modifier `4to16` is depicted in Figure 7-17. All the lane muxes have been driven to point to the source lane zero, but a full 128 bits have been loaded from memory corresponding to the elements `d, c, b, a`. The four loaded values from memory are replicated across each lane.

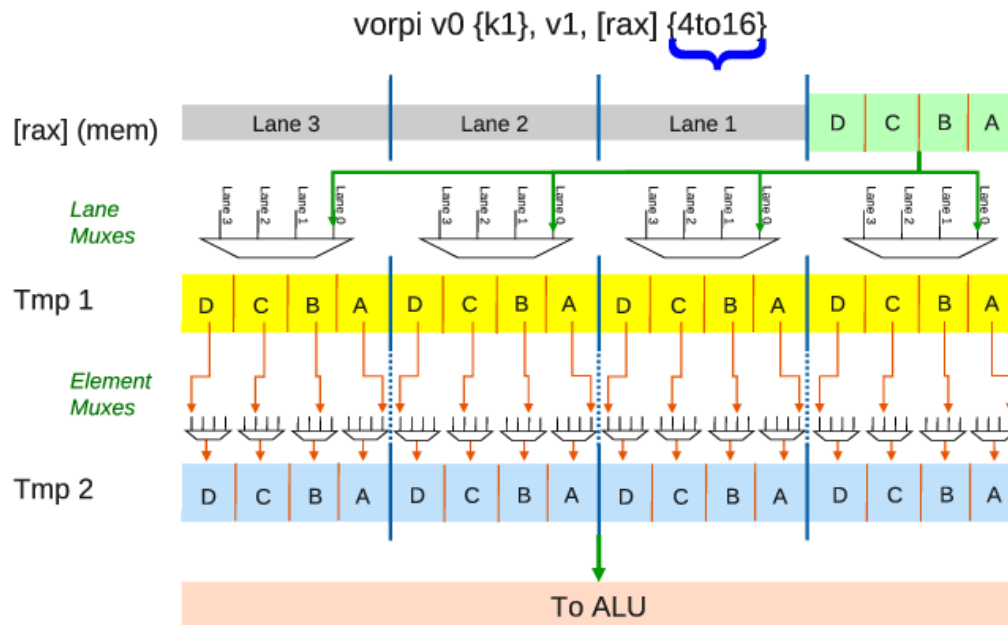


Figure 7-17. The 4-to-16 Register-Memory Swizzle Operation

The default (implied) `16to16` operator loads the full 16 elements from memory, driving the lane and element muxes in the default pass-through configuration. The `16to16` operator is optional.

7.11.3.2 Data Conversion

The purpose of the data conversion modifier is to use the swizzle field to read data from memory that is in a convenient memory storage form, and to convert it to match the 32-bit wide native Intel® Xeon Phi™ coprocessor data type of vector elements. As stated previously, the Intel® Xeon Phi™ coprocessor performs all operations assuming a collection of 16 elements, each 32-bits in size. There are four predefined swizzle patterns that operate in register-memory mode when performing data-conversion operations. A different set of four conversions is available depending on whether the conversion is to a destination floating-point or to an integer operation.

For floating-point instructions, conversions from memory are supported via inline swizzles: `sint16` and `uint8`. For integer (signed or unsigned) instructions, conversions from memory are supported via inline swizzles: `uint8`, `sint8`, `uint16`, and `sint16`. If no swizzle conversion is specified, then the load from memory undergoes no conversion, meaning that the data is assumed to be in the appropriate native format in the memory storage form (`float32` or signed/unsigned `int32`). Figure 7-18 depicts the conversion process for `vorpi` from a memory storage form of `uint8` to the `sint32` expected for the instruction.

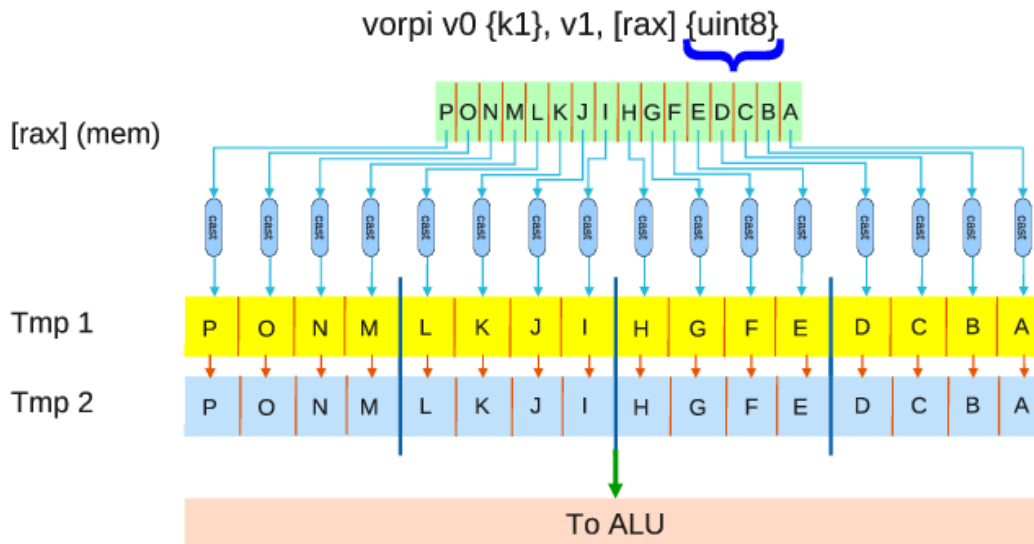


Figure 7-18. The uint8 Data Conversion Register-Memory Swizzle Operation

For this particular `vorpd` instruction, the data in memory consists of a series of one-byte items. The conversion process reads the `uint8` data from memory, and converts it to a signed 32-bit integer. The 16 converted values are then fed to the ALU for the OR operation to use. For this operation, the floating-point conversion types are disallowed because `vorpd` requires *signed integers* – thus, `float16` is an illegal conversion specifier in this example.

7.11.4 Swizzle Usage Examples

In order to demonstrate scenarios where swizzles are useful, three examples are briefly explored: scalar-vector multiplication, horizontal addition, and 3x3 matrix determinants.

7.11.4.1 Scalar-Vector Multiplication

There are two readily usable forms for scalar-vector multiplication available. The conceptual difference between the two forms hinges on whether you are using lane semantics (where each vector is broken into four lanes) or total vector semantics (where each vector is considered a collection of 16 elements without additional abstraction levels).

For the lane semantic case, scalar-vector products expect each lane to have a different scalar multiplier applied to the destination vector. For example, consider the following vectors:

$$\vec{u} = \langle x, x, x, S_3, x, x, x, S_2, x, x, x, S_1, x, x, x, S_0 \rangle \quad \text{and} \quad \vec{v} = \langle p, o, n, m \rangle$$

where the value `x` represents a value that is irrelevant to the operation. A lane-based scalar-vector product would result in the vector:

$$\begin{aligned} \vec{z} = & \langle S_3 \langle p, o, n, m \rangle, S_2 \langle l, k, j, i \rangle, S_1 \langle h, g, f, e \rangle, S_0 \langle d, c, b, a \rangle \rangle \\ & \vec{z} = \langle S_3 \cdot p, S_3 \cdot o, S_3 \cdot n, S_3 \cdot m, S_2 \cdot l, S_2 \cdot k, S_2 \cdot j, S_2 \cdot i, \end{aligned}$$

$$S_1 \cdot h, S_1 \cdot g, S_1 \cdot f, S_1 \cdot e, S_0 \cdot d, S_0 \cdot c, S_0 \cdot b, S_0 \cdot a$$

>

In terms of Intel® Xeon Phi™ coprocessor programming, this operation is easily conveyed in one instruction, as long as the scalar multipliers reside in a vector register and each scalar multiplier resides in the same per-lane element field (e.g., each multiplier is in the position of element *a* for every lane). Let $zmm1 = \vec{u}$ and $zmm2 = \vec{v}$. The scalar-vector multiplication for lane semantics, assuming floating-point data with a destination of $zmm0$ and a write mask of $k1$, is simply:

```
vmulps:::zmm0:{k1},:zmm2,:zmm1:{aaaa}
```

This single instruction will obtain the desired result, however using it frequently may not be optimal. Use of the register-swizzle modifier incurs a +1 clock penalty on the latency⁸ of an instruction using the swizzle. Therefore, if the same data value (here, $zmm1$) is frequently reused with the same swizzle, it might be better to generate a temporary register with the desired data pattern combination and then refer to that temporary register.

For total vector semantics, the operation follows traditional mathematic rules. For some constant, k , the scalar-vector multiplication against a vector $\vec{u} = \langle p, n, a \rangle$ would result in $\vec{z} = k \langle p, o, a \rangle$. This, too, can be achieved with a single Intel® Xeon Phi™ coprocessor instruction, assuming the constant k resides *in memory*:

```
vmulps:::zmm0:{k1},:zmm2,:[k]:{1to16}
```

In this case, the memory storage form must match the expected input to the instruction, which for `vmulps` is a 32-bit floating-point value. While this operation will get the desired result, it consumes cache bandwidth. In a situation where this constant is used repeatedly, it might be better to load the constant (with broadcast replication) into a temporary vector register, and then use the temporary register to preserve cache bandwidth.

7.11.4.2 Horizontal Addition

From the viewpoint of SIMD architecture, the Intel® Xeon Phi™ coprocessor implements wide vectors. Horizontal operations (also known as reduction operations) do not scale well in microarchitecture implementations due to the inherent serial nature of the operation. For any given vector $\vec{v} = \langle v_{n-1}, v_{n-2}, \dots, v_0 \rangle$, a horizontal operation on \vec{v} , such as addition, requires the computation of:

$$\sum_{i=0}^{n-1} v_i$$

This computed value is commonly stored in either the beginning or ending element of the resulting vector, or else in every element position of the resulting vector. Some forms of horizontal operations are defined to leave partial results in each element of the vector. That is, if the summation were redefined to be:

$$f(n, \vec{v}) = \sum_{i=0}^{n-1} v_i$$

then some horizontal operations would require computing the resultant vector. Where n is the number of elements in the vector, the resultant vector becomes:

$$\vec{z} = \langle f(n, \vec{v}), f(n-1, \vec{v}), \dots, f(1, \vec{v}) \rangle$$

⁸ This is an observed extension of the pipeline depth; it does not trigger a core stall or otherwise impede the thread as long as excepting standard dependency rules.

The Intel® Xeon Phi™ coprocessor does not provide direct support for horizontal operations (They cannot be scaled efficiently in microarchitectures as the SIMD vector width lengthens.). As a general observation, most needs for horizontal operations stem from *inner-loop unrolling*, where data parallelization is applied to the data set of the inner-most loop body. Horizontal or reduction operations are common in this type of parallelization. In contrast, *outer-loop unrolling* achieves a different effect by parallelizing instances through each element in the vector. There is rarely a need for horizontal operations in outer-loop unrolling. An extreme form of outer-loop unrolling would be the typical programming model of High-Level Shader Language (HLSL), where operations are defined only on scalar data types. In the HLSL case, parallel threads are run so that thread t_i is executed in vector element v_i . In this case, execution of parallel threads occurs where the number of threads being executed equals the width of vector elements supported by the underlying microarchitecture.

The Intel® Xeon Phi™ coprocessor programming uses outer-loop unrolling and a more aggressive form of thread parallelism through the vector elements. From this perspective, the lack of horizontal operations does not impede highly efficient programming constructs. When situations arise where horizontal operations are needed, they tend to happen at the final stage. When combining work from multiple virtual threads in the vector elements, a few extra cycles in the calculation to emulate horizontal operations has a negligible effect on overall program performance.

Keeping this in mind, it is still possible to perform horizontal operations efficiently even though the Intel® Xeon Phi™ coprocessor lacks single instruction versions of common horizontal operations. For example, to compute the dot-product of two vectors, use a vector multiply followed by a horizontal add. To continue with the example of horizontal add, let vector $\vec{v} = \langle d, c, b, a \rangle$. The horizontal add applied to \vec{v} will return a vector:

$$\vec{z} = \langle d + c + b + a, d + c + b + a, d + c + b + a, d + c + b + a \rangle$$

The sequence of Intel® Xeon Phi™ coprocessor instructions shown in Figure 7-19 can easily be constructed, assuming an input vector `zmm0` with the contents $\langle x, x, \dots, x, d, c, b, a \rangle$, where x is a data value that is uninteresting.

```

; zmm0 = (.....,d,c,b,a)
; Goal: zmm1 = (... , d+c+b+a, d+c+b+a, d+c+b+a, d+c+
b+a)
;
; value in Lane 0 of zmm1
; (d+c), (c+d), (b+a), (a+b)
;
; value in Lane 0 of zmm1
; (d+c)+(b+a), (c+d)+(a+b), (b+a)+(d+c), (a+b)+(c+d)

```

Figure 7-19. Trivial Implementation of a Horizontal Add Operation Within One Vector Lane

At the end of the addition, with the alternating swizzles `cdab` and `badc`, it is clear that the horizontal addition has completed within each lane of the vector `zmm1`. Using floating point commutativity⁹ rules, the actual result of the destination vector in lane zero is:

$$\langle (d+c) + (b+a), (d+c) + (b+a), (d+c) + (b+a), (d+c) + (b+a) \rangle$$

⁹ Commutativity is limited when the x87 or up-conversion of a data type is involved.

If the horizontal operation uses floating-point data, be very careful when using the results as all four values violate the associativity rules set as IEEE industry standards. Unsafe compiler optimizations may produce these results, or programmers who are aware of the implications of violation may choose to use this technique regardless. Three Intel® Xeon Phi™ coprocessor instructions are required to get a correct horizontal add result with respect to floating point associativity, but the programmer must know whether to evaluate the terms left-to-right or right-to-left (e.g., $((a+b)+c)+d$ versus $((d+c)+b)+a$).

The two instructions used in Figure 7-19 to determine the horizontal add only compute that horizontal add result within *each lane* of the vector concurrently. If the horizontal add only applies to a single lane, then the previously mentioned instructions generate four horizontal add results.

7.11.4.3 3x3 Matrix Cross Product (Determinant)

A common vector operation computes a 3x3 cross-product (matrix determinant) to find a normal vector for two input vectors or the area of a parallelogram defined by two input vectors. For two vectors $\vec{u} = \langle c, b, a \rangle$ and $\vec{v} = \langle o, n, m \rangle$, the cross product is $\vec{u} \times \vec{v} = \langle an - bm, cm - ao, bo - cn \rangle$. While the cross product is only defined for either three- or seven-dimensional vectors, the calculation for the cross product is equal to calculating the determinant of a matrix.

Using the swizzle modifiers carefully makes this computation easy and can attain four products for every three instructions. A simple code sequence that calculates the determinant in a lane is shown in Figure. It is actually possible to accumulate multiple determinants by adding the incremental results in the vector `zmm2` (not shown). The final operation on line 3 is to correct the order of the terms in `zmm3`, but it follows that, if `zmm2` is non-zero, then the result will be added to any prior data.

```

; zmm0 = (....., c, b, a)
; zmm1 = (....., o, n, m)
; zmm2 = <zero vector>                                     3
; Goal: zmm3 = (... , an-bm, cm-ao, bo-cn)
;
; operation value in Lane 0 of zmm3                         6
;
vmulps zmm3, zmm0, zmm1 {dacb} ; (cm, bo, an)
vfnadd231ps zmm3, zmm1, zmm0 {dacb} ; (cm-ao, bo-cn, an-  9
bm)
vaddps zmm3, zmm2, zmm3 ; (an-bm, cm-ao, bo-cn)

```

Figure 7-20. Trivial Implementation of a 3x3 Matrix Cross-Product Within One Vector Lane

As with the horizontal add, the three instruction sequence shown is computing a 3x3 matrix determinant in each lane of the vector. Therefore, four determinants are generated for every three instructions. This code sample can also be extended to higher-order square matrix determinant calculations. Higher-order determinants are calculated by iterative cofactor expansion through row or column reductions.

7.12 The Shuffle

Fundamentally, the power and ease of using swizzles on almost any instruction in the Intel® Xeon Phi™ coprocessor architecture makes manipulating data pattern combinations remarkably easy. There remain cases, however, where a more powerful data-pattern combination function is needed – these cases generally hinge on the need to cross lane boundaries, a feature that swizzles cannot provide. One option to cross a lane boundary is to use load-store operations to memory, but this has the undesirable side effect of increasing cache pressure and wasting the level one caches, which

are a limited resource. Using load-store operations also interferes with other scheduled activities, such as prefetching, gather-scatter operations, and so on.

For generalized data pattern combinations and to cross lane boundaries, the proper instruction to use is the shuffle. The Intel® Xeon Phi™ coprocessor's shuffle instructions are a more advanced form of those found in Intel's SSE or AVX instruction sets. In the Intel® Xeon Phi™ coprocessor, the shuffle instructions are not fully generalized; that is, there are limitations to how you can combine the data pattern in one instruction.

Unlike swizzle, the shuffle instruction allows you to specify the controls for the lane muxes. Any valid combination of lanes may be routed from the source to the first temporary value. The first restriction of the vector shuffle in the Intel® Xeon Phi™ coprocessor is that you can rearrange lanes, but you cannot combine elements from different lanes in one instruction. Eight bits of immediate encoding are required in the shuffle instruction in order to fully specify the lane mux selections.

Once the lane sources are mapped, it becomes possible to drive the element muxes. To support fully arbitrary control sequences across all of the element muxes, however, would require 32 bits of immediate encoding on the shuffle instruction. Support for this is impractical in a pipeline implementation. To reduce the encoding space required, the vector shuffle instruction of Intel® Xeon Phi™ coprocessor allows you to specify an arbitrary element combination inside of a single lane. Once this combination is selected, the same pattern is applied across all the lanes. This is similar to how the swizzle works, in that the pattern is applied across all lanes; but unlike the swizzle any arbitrary combination may be chosen for the pattern. The shuffle instruction does not restrict you to eight possible combinations.

The Intel® Xeon Phi™ coprocessor supports the following shuffle instructions:

- **VPSHUFD**
vpsshufd zmm1 {k1}, zmm2/mt, imm8
Shuffles 32-bit blocks of the vector read from memory or vector zmm2/mem using index bits in immediate. The result of the shuffle is written into vector zmm1. No swizzle, broadcast, or conversion is performed by this instruction.
- **VPERMF32X4**
vpermf32x4 zmm1 {k1}, zmm2/mt, imm8
Shuffles 128-bit blocks of the vector read from memory or vector zmm2/mem using index bits in immediate. The result of the shuffle is written into vector zmm1. No swizzle, broadcast, or conversion is performed by this instruction.

Do not confuse the swizzle and the shuffle element patterns; the shuffle can take an arbitrary pattern, whereas the swizzle is limited to eight predefined patterns. If you receive compiler or assembler errors regarding a *data pattern swizzle*, the two immediate checks you should perform are:

- Verify that you have all register operands
- Verify that your swizzle combination pattern is legal

7.13 Memory and Vectors

Section 2 introduced the major components of Intel® Xeon Phi™ coprocessor's new vector architecture, including the general form of the vector instructions. That section also introduced the difference between all-register operands to an instruction and those instructions that include an *implicit load* by replacing the second source register with a memory reference. This type of inline load is common among CISC architectures, and can allow for better code density than other approaches. The material previously covered also introduced the idea of modifiers to the source arguments, where an implicit load can do either a limited form of data conversion through swizzles *or* data replication through broadcasts.

This section expands upon the prior foundation by looking at the true vector load and store instructions, with their extended capabilities beyond what the implicit load modifiers can support. Unusual vector load and store instructions, such as pack and unpack, are presented with their exact behaviors. This section also details the memory alignment restrictions that the Intel® Xeon Phi™ coprocessor places upon vector loads and stores of any form, as well as ways to work around the alignment restrictions.

7.13.1 Load and Store Operations

Implicit load operations on routine vector instructions, such as `vaddps`, allow you to select from a small set of data conversions or data replications. In contrast, the vector load instruction `vmovaps` allows specification of *both*, with a far larger set of data conversions supported. The basic form of the vector load and store instruction is:

Store instruction: `vmovaps mt {k1}, Df32(zmm1)`
 Load instruction: `vmovaps zmm1 {k1}, Uf32(mt)`

Table 7-4 summarizes the range of choices in the optional *{conversion}* and *{broadcast}* fields.

Table 7-4. The `vmovaps` Instruction Support for Data Conversion and Data Replication Modifiers

Broadcasts Supported				
1to16	4to16	16to16		
Conversions Supported				
To	From			
float32	srgb8	sint8	uint8	snorm8
	unorm8	uint16	sint16	unorm16
	snorm16	float16	unorm10A	unorm1B
	float11C			
sint32	sint8i	sint16i		
uint32	uint8i	uint16i		

More information about the forms can be found in various public specifications, as well as the Intel Architecture manuals [(Intel® 64 and IA-32 Architectures Software Developer Manuals)].

Table 7-4 also shows that the memory storage form of `sint16` can be converted to either `float32` or `sint32` destination types. Note that if the conversion target is a signed or unsigned integer, the conversion type has an `I` postfix; therefore, `uint8i` and `uint16i` will convert from 8- or 16-bit unsigned integers into a 32-bit unsigned integer. Similarly, `sint8i` and `sint16i` will converted from 8- or 16-bit signed integers into a 32-bit signed integer. There is no direct conversion in the `vmovaps` instruction from unsigned to signed integer.

The `float32` target supports a wide variety of conversion types, including the half-precision `float16`. Most of the conversion types supported by the vector load instruction are in direct support of the various graphics standards. The

distinction between `unorm` conversions in the range 10A,10B,10C,2D has to do with which field of the packed data format is converted (A, B, C, or D, respectively). Similarly, the `float` types including 11A,11B, and 10C center on which field is converted from the packed memory storage form (A, B, or C). The specific encoding rules for the various types are summarized in Table 7-5.

Table 7-5. The Floating-Point Data Type Encodings Supported

Type	Encoding
float16	s10e5
float11	s6e5
float10	s5e5

When a write mask is used in conjunction with a vector load, only those elements that have their mask bit set to one are loaded. This is shown in Figure 7-21.

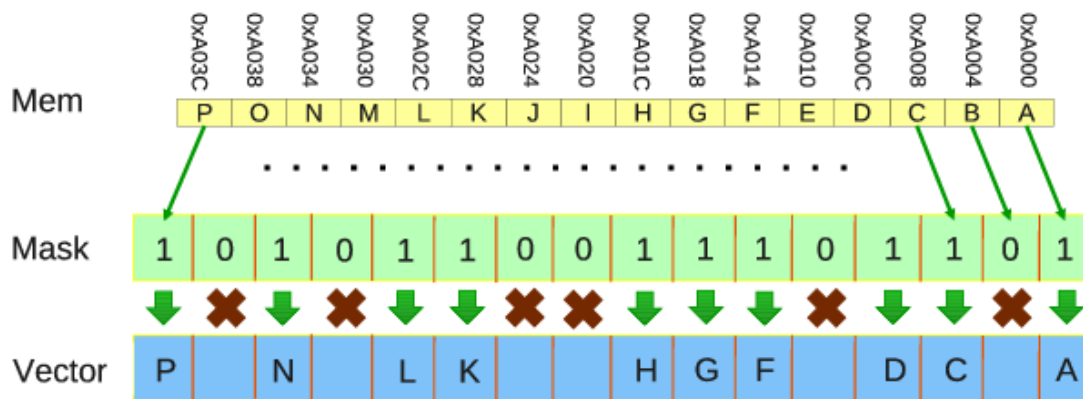


Figure 7-21. The Behavior of the Vector Load Instruction With a Write Mask

When a write mask is used in conjunction with a vector store, only those elements that have their mask bit set to one are stored. This is shown in Figure 7-22.

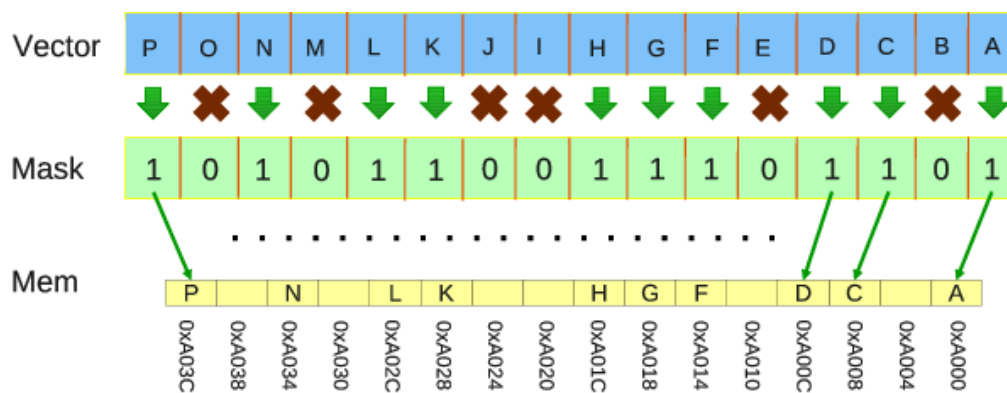


Figure 7-22. The Behavior of the Vector Store Instruction With a Write Mask

Where the mask bit of an element is set to zero, *the corresponding vector element remains unchanged*. For example, if a vector load/store is performed with a mask `k1` set to `0xFF00`, then only the upper eight elements will be loaded from memory, starting at an offset of eight elements from the given load/store address.

All load/store operations must conform to the memory alignment requirements. Failure to use conforming addresses will trigger a general-protection fault.

The following broadcast load instructions are supported:

- **VBROADCASTF32X4**
Broadcast 4xfloat32 vector Uf32(mt) into vector zmm1, under writemask. The 4, 8 or 16 bytes (depending on the conversion and broadcast in effect) at memory address mt are broadcasted and/or converted to a float32 vector. The result is written into float32 vector zmm1.
- **VBROADCASTF64X4**
Broadcast 4xfloat64 vector Uf64(mt) into vector zmm1, under writemask. The 32 bytes at memory address mt are broadcast to a float64 vector. The result is written into float64 vector zmm1.
- **VBROADCASTSS**
Broadcast float32 vector Uf32(mt) into vector zmm1, under writemask. The 1, 2, or 4 bytes (depending on the conversion and broadcasted in effect) at memory address mt are broadcast and/or converted to a float32 vector. The result is written into float32 vector zmm1.
- **VBROADCASTSD**
Broadcast float64 vector Uf64(mt) into vector zmm1, under writemask. The 8 bytes at memory address mt are broadcasted to a float64 vector. The result is written into float64 vector zmm1.

7.13.2 Alignment

Regardless of which vector load or store form is used, all memory-based operations for vectors must be on properly aligned addresses. Intel® Xeon Phi™ coprocessor’s vector architecture supports memory conversions as well as broadcast and subset of the data element, this can make address alignment confusing at first. A simple means of evaluating the alignment requirement is to use an equation:

$$alignment = number_{elements} \times size_{element} \quad (1)$$

The $number_{elements}$ for vector load and store operations is set by the broadcast or subset modifier to the instruction. The number of elements is in the set of [1,4,16]. The $size_{element}$ is based on the *memory storage form*, whether loading or storing. Since the Intel® Xeon Phi™ coprocessor treats all data internally as 32-bit values, only the representation in memory determines what alignment (and how much cache bandwidth) is required. For example, with float16 data, each element is two bytes in size in memory storage form. Table 7-6 summarizes the address alignment requirements.

Table 7-6: Memory Alignment Requirements for Load and Store Operations

<i>Address Alignments</i>			
Memory Storage Form	Number of Elements		Alignment in Bytes
	Load Form	Store Form	
4 bytes	16to16	<i>all</i>	64 bytes
	4to16	<i>dcba</i>	16 bytes
	1to16	<i>a</i>	4 bytes
2 bytes	16to16	<i>all</i>	32 bytes
	4to16	<i>dcba</i>	8 bytes
	1to16	<i>a</i>	2 bytes
1 byte	16to16	<i>all</i>	16 bytes
	4to16	<i>dcba</i>	4 bytes
	1to16	<i>a</i>	1 byte

The vector write mask is *not* consulted during memory alignment decisions – only the broadcast/subset modifier affects the total alignment constraints in addresses. A representative example for forcing address alignments in C or C++ source programs is shown in Figure 7-23

```

__declspec(align(64)) // force 64-byte
alignment
struct Matrices                                     2
{
float x[3 * DATASIZE]; // Vector input
float A[9 * DATASIZE]; // 3x3 Matrix                                     5
float r[3 * DATASIZE]; // vector result
} Matrices[64];

```

Figure 7-23. Compiler Extension to Force Memory Alignment Boundaries in C or C++ Tools

There is a corner case in the address alignment calculation where *number_{elements}* as a term is always considered equal to one. This case reduces the alignment constraint to the size of an element in memory storage form. This corner case applies to the vector pack and vector unpack instructions and to the vector gather and vector scatter instructions.

7.13.3 Packing and Unpacking Operations

There is another form of vector load and store operation that the Intel® Xeon Phi™ coprocessor presents: the *pack* (store) and *unpack* (load) instructions (alternatively known as *compress* and *expand*). The premise behind these constructs is that memory is always read contiguously, but that the contents read or written with respect to the vector register are *not* contiguous. In order to achieve this, the vector mask register is used in a slightly different manner than the write mask would normally be used. The vector unpack instruction behavior is shown in Figure 7-24.

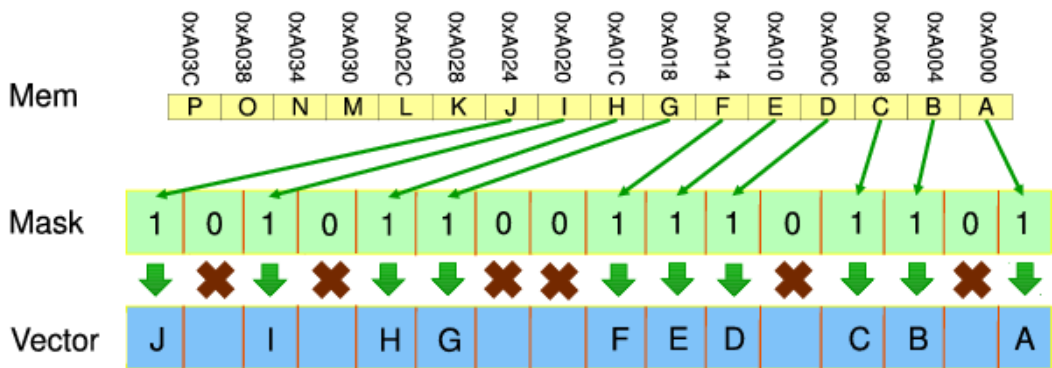


Figure 7-24. The Behavior of the Vector Unpack Instruction

The normal vector load operation reads all 16 elements from memory and overwrites the destination based on the mask, thereby skipping source elements in the serial sense. The unpack instruction keeps the serial ordering of elements and writes them sparsely into the destination.

The first element (A) is read from memory, and the mask register is scanned to find the first bit that is set to the value 1, in a right-to-left ordering. The corresponding element of the mask register with the first bit set is over-written with the value loaded from memory (A).

The second element (B) is read from memory, and the scan of the mask register bits resumes from where it left off. The second mask bit that is set to the value 1 has a corresponding destination element value that is over-written with the value from memory (B).

This process is continued until all of the elements that are masked “on” have their values replaced with a contiguous read from memory. The alternate naming of this instruction *expand* reflects the nature of the operation – expanding contiguous memory elements into noncontiguous vector elements.

The equivalent instruction in vector store form is the *pack* instruction, shown in Figure 7-25.

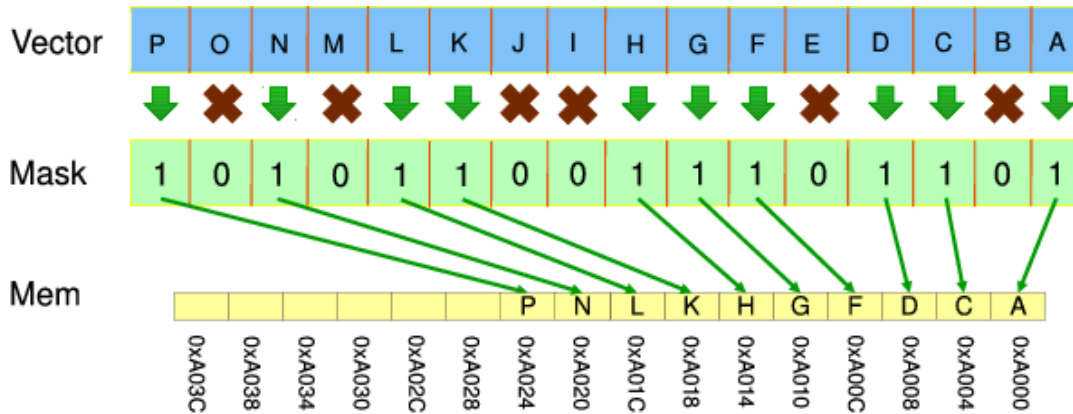


Figure 7-25: The Behavior of the Vector Pack Instruction

Conceptually, these instructions are fairly straightforward in that memory is always read or written contiguously while the vector elements are used only if their corresponding mask bit is set. In actual implementation, however, these abstract instructions complicate the real vector pack and unpack instruction specification. Since these instructions are working with memory on an element-size basis (based on the memory storage form), the address alignment requirement does not include a factor for the number of elements being read or written. A challenge in implementation arises if the write mask is set to a pattern that would cause a starting address to walk across a cache line boundary.

The Intel® Xeon Phi™ coprocessor uses *two* instructions to create the pack or unpack instructions. One instruction operates on the *lower address cache line* while the second instruction operates on the *higher address cache line*. They generate a dependency chain with long latencies (7 cycles per instruction) because they typically use the same destination register. Thus, *vector unpack* is actually comprised of two instructions:

```

vloadunpackld    zmm1:[:{k1}]:,ptr:[:{conversion}]:]
vloadunpackhd    zmm1:[:{k1}]:,ptr+64:[:{conversion}]:]

```

The order of specifying these two instructions is irrelevant. While the write mask is optional, the `vloadunpackhd` version of the instruction *requires you to specify +64 to the same memory address*¹⁰. This requirement stems from another microarchitectural complexity in preserving power- and area-efficiency while automatically calculating the address.

The unpack instruction can take a data conversion modifier, supporting the same choices that `vload` does. However, the unpack instructions cannot take a broadcast modifier; they are restricted to data conversion modifiers.

¹⁰ The +64 requirement does not change the expected behavior, as it is only applied to the “high” cache line operation. This is a requirement that stems from reducing the microarchitectural complexity. Programmers should overlook the presence of the +64 accordingly, though failure to include it will produce incorrect results.

As with the conceptual unpack instruction, the *vector pack* instruction is actually comprised of two instructions: `ld` and `hd`. This pair of instructions takes the same conversion modifiers as the vector store instruction:

<i>Vpackstoreld</i>	<code>ptr:[{k1}]:,zmm1:[{conversion}]:]</code>
<i>vpackstorehd</i>	<code>ptr+64:[{k1}]:,zmm1:[{conversion}]:]</code>

By careful use of 64-byte aligned buffers in memory, it is possible for an optimized code sequence to execute just one of these instructions for pack or unpack. But it has been found that a common program bug is introduced when a previously aligned buffer becomes unaligned. Therefore, you should always use both the `hd` and `ld` instructions, even when you believe it may not be required.

A problem with using two instructions to achieve one conceptual result (whether pack or unpack), is that there is no guarantee of the code using these instructions. A coherence problem may appear (data race) with incorrect results if the data changes between the execution of the `ld` and `hd` instructions. This would be a programmer error for failing to wrap the pack or unpack instructions in a critical section.

The vector pack and unpack instructions are highly versatile and useful, and they also provide a way to relax the memory alignment constraints. Where `vload` and `vstore` require addresses to be aligned, the pack and unpack instructions only require alignment to the memory storage form. As long as the address to load from is aligned on a boundary for the memory storage form, then executing the pair of `ld` and `hd` instructions loads the full 16 elements (with optional conversions) given a write-mask of `0xFFFF`.

This technique bypasses the more strict alignment but it is limited in that it only provides identical results to the vector load and store operations when the write mask is set to `0xFFFF`. If the mask does not have every bit set, then the result of the two instruction sequence will have either data elements in the vector or data elements in memory re-ordered relative to the standard load and store instructions.

7.13.4 Non-Temporal Data

The *non-temporal hint* is one modifier that can be applied to every Intel® Xeon Phi™ coprocessor memory reference for vector load or store operations, as well as to software prefetching instructions. The NT hint is specified in the same convention as `swizzle` modifiers, denoted by `{NT}`. A typical example of the NT hint in conjunction with a regular `vmovaps` instruction is:

```
Vmovaps:::[rsi]:{k5},:zmm29:{float16}:{dcba}:{NT}
```

Cache hierarchies are designed to optimize efficiency by using the locality of data (and instructions) in a temporal stream. When any given datum is operated on, it is highly likely that the same datum will be used again in the near future. When operating on *streaming data*, however, the expectation of temporal locality is broken. With many streaming applications, data is referenced once, then discarded.

When mixing data that has temporal locality with data that does not have temporal locality, caches become *polluted* in the sense that their efficiency is hampered. The cache replacement policy is what guides where old data is removed from the cache and new data is brought in. Policy management uses state machines and counters so that the cache implementation will track which data has been used most recently (MRU) and which data has been least recently used (LRU), with an ordered series of steps for each entry between these two points (LRU, MRU).

When a cache is fully populated and a new datum needs to be inserted, the replacement policies center on the idea of LRU management. In this case, the LRU datum is evicted from the cache, and the new datum is placed where the LRU used to be. The tracking state for the MRU-LRU state is updated, so that the newest datum is now MRU and every other datum decreases in temporal locality by one step. This, in turn, creates a new LRU entry.

The pollution comes from streaming data making this MRU-LRU state tracking less effective. In essence, streaming data is non-temporal in that there is no expectation for rapid re-use. When streaming data is loaded into a typical cache system, it automatically becomes MRU and displaces the last LRU data. When many streaming references are accessed at once, this causes significant eviction of data that may be temporally useful, causing subsequent loads and stores to stall due to cache misses.

To deal with this, the Intel® Xeon Phi™ coprocessor uses a non-temporal (NT) hint to override the default MRU-LRU state machine. When any load or store operation is tagged with the NT hint, it automatically forces the datum in the cache to become LRU; that is, next in line for eviction by subsequent operations. In the case of a software prefetch instruction, however, the datum is tagged MRU so that it is more likely to be present in the cache on the subsequent load or store operation for that datum.

7.13.5 Prefetching

Most modern microprocessors incorporate at least one hardware prefetching unit in the cache hierarchy to minimize the likelihood of misses in the L1 or L2 caches. The presence of hardware prefetchers typically increases the performance of a given workload by at least ten to fifteen percent for well-optimized programs, with some outliers demonstrating higher performance improvements. For programs that have not been extensively optimized and hand-tuned, a hardware prefetch unit is commonly capable of improving performance by fifty percent or more. Hardware prefetchers can be quite complex in their implementation, with more advanced implementations using adaptive tuning and predictive guesses for patterns observed in a series of cache accesses. This complexity requires a greater hardware design validation effort and post-silicon verification of behavior.

The Intel® Xeon Phi™ coprocessor has a modified autonomous prefetching unit design that originated with the Intel® Pentium® 4. The Intel® Xeon Phi™ coprocessor also introduces a variety of software prefetching instructions, enabling you to tune your program’s behavior without interference from autonomous hardware prefetchers.

7.13.5.1 L1 Prefetching

It is not possible to use the prefetch instructions directly on the L1 instruction cache. Software controlled prefetching for L1 can only directly access the data cache in Intel® Xeon Phi™ coprocessor.

Table 7-7. L1 Prefetch Instructions

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCHO	L1	No	No
VPREFETCHNTA	L1	Yes	No
VPREFETCHEO	L1	No	Yes
VPREFETCHENTA	L1	Yes	Yes

The prefetching instructions for L1 that prefetches at memory line in m8 are:

- VPREFETCHO m8
- VPREFETCHNTA m8
- VPREFETCHEO m8
- VPREFETCHENTA m8

The *exclusive* control bit tells the cache subsystem that the program is executing a data store to the target cache line. This causes the coherence engine to ensure that the cache line is set to either the Exclusive or the Modified state in the MESI protocol, fetching the data if it is not already there. The *non-temporal* (NT) control bit is processed; however, the

NT control with a prefetch *always* sets the LRU state machine to indicate MRU status so that the prefetched data is not lost. The *misshint* (MH) control indicates that the target cache line is *not* expected to be resident in the cache, and that the hardware scheduler should not reschedule this thread until the data has been loaded. This uses the same internal mechanism as the `delay` instruction. Any valid bit-wise OR combination of these controls bits (or none) may be set in the *hints* field of the instruction.

This instruction is considered to be a microarchitecture hint, and the hardware may defer or drop execution. If the requested cache line containing the specified address is already in the L1 data cache, the prefetch is dropped. Similarly, any attempts to prefetch uncacheable or WC memory are ignored.

7.13.5.2 L2 Prefetching

The Intel® Xeon Phi™ coprocessor supports the software-controlled prefetching of code or data directly to the L2 cache. The prefetching instructions for L2 are listed in Table 7-8.

Table 7-8. L2 Prefetch Instructions

Instruction	Cache Level	Non-temporal	Bring as exclusive
VPREFETCH1	L2	No	No
VPREFETCH2	L2	Yes	No
VPREFETCHE1	L2	No	Yes
VPREFETCHE2	L2	Yes	Yes

The exclusive control bit tells the cache subsystem that the program is executing a data store to the target cache line. This causes the coherence engine to ensure that the cache line is set to either the Exclusive or the Modified state in the MESI protocol, fetching the data if it is not already there. The non-temporal (NT) control bit is processed; however, the NT control with a prefetch always sets the LRU state machine to indicate MRU status so that the prefetched data is not lost. Any valid bit-wise OR combination of these controls bits (or none) may be set in the hints field of the instruction.

This instruction is considered to be a microarchitecture hint, and the hardware may defer or drop execution. If the requested cache line containing the specified address is already in the L2 data cache, the prefetch is dropped. Similarly, any attempts to prefetch uncacheable or WC memory are ignored.

7.14 New Instructions

Please refer to the (Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364)) for a complete description of the instruction set supported. The following instructions are those added most recently.

7.14.1 Mask Manipulation Instructions

Mask-manipulation instructions that are supported are listed in the following table.

Table 7-9. Mask Manipulation Instructions

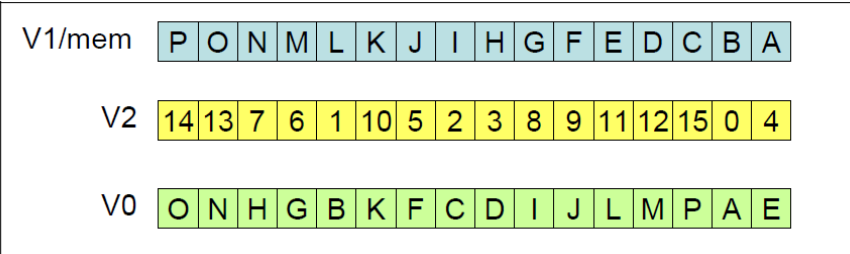
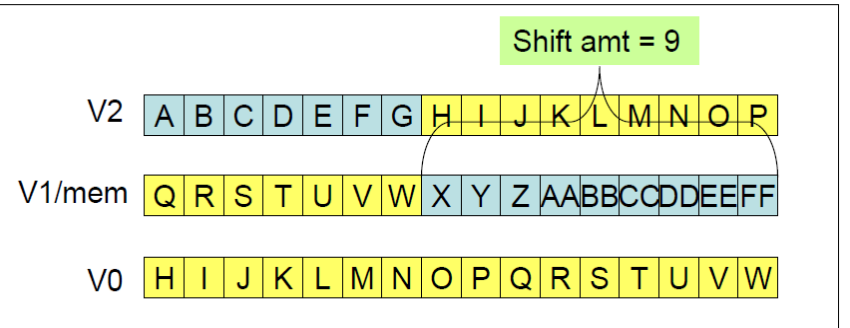
Instruction	Description
<code>vkmov k2, [mem]</code>	Loads 16 bits from a 16b aligned memory address into a mask register.
<code>vkinsert k2, rax, imm8</code>	Loads 16 bits from any quadrant of GPR rax specified by imm8.
<code>vkextract rax, k2, imm8</code>	Stores 16 bits from any quadrant of GPR rax specified by imm8, and the rest of the rax is set to zero.

Instruction	Description
vkmovhd rax, k2, k3	Stores 32 bits from mask registers into the higher half of the GPR rax and zeros the lower half of the GPR rax.
vkmovld rax, k4, k5	Stores 32 bits from mask registers into lower half of the GPR rax and zeros the higher half of the GPR rax.

7.14.2 Packed Typeless Instructions

Table 7-10 lists the supported packed typeless instructions.

Table 7-10. Packed Typeless Instructions

Instruction	Description
vstoredhi [mem] {k1}, v1	Stores the higher eight 32-bit elements from vector v1 to memory location [mem], under writemask k1.
vstoredlo [mem] {k1}, v1	Stores the lower eight 32-bit elements from vector v1 to memory location [mem], under writemask k1.
vpermd v0 {k1}, v2, v1/mem	<p>v1/mem is the register or memory source data to be permuted, v2 is the 16 4-bit index or desired permuted location, and v0 is the destination register. v2 can contain same indexes or same permuted location. VPERMD is implemented using the generic swizzle mux.</p> 
valignd v0 {k1}, v2, v1/mem, imm8	<p>v1/mem is the register or memory source data, v2 is the second register source, imm8 specifies the shift amount (bottom 2 bits are zeroed out for finer valign instructions in the future; only bits 5:2 are used), and v0 is the destination register.</p> 
vscatterq [rax+v2] {k1}, v1	Quadword scatter of vector v1 to vector memory location [rax+v2], under writemask k1. It stores eight 64-bit elements from v1 to memory based on the lower eight 32-bit indexes in v2 given base address [rax].
vgatherq v1{k1}, [rax+v2]	Quadword gather from memory location [rax+v2] into quadword vector v1, under writemask k1. It loads eight 64-bit elements from memory to v1 based on the lower 256-bit (eight 32-bit) indexes in v2 given base address [rax].

Instruction	Description
vgetexppq v1 {k1}, Sf(v2/mt), imm8	Extracts int64 vector of exponents from float64 vector Sf(v2/mt) and stores the result in v1, under writemask. Float64 vector inputs are normalized according to imm8[1:0] and the sign of the exponents is returned as int64. If the source is SNaN, QNaN will be returned. If the source is +INF, source will be returned.

7.14.3 New Packed SP FP Instructions

The new packed SP and FP instructions are:

- vrcp23ps v1 {k1}, v0 // $y = \text{minimax_quad_approx_recip}(x)$
- vsqrt23ps v1 {k1}, v0 // $y = \text{minimax_quad_approx_rsqrt}(x)$
- vlog2ps v1 {k1}, v0 // $y = \text{minimax_quad_approx_log2}(x)$
- vexp2ps v1 {k1}, v2 // $y = \text{minimax_quad_approx_exp2}(f) * 2^i$
- vcvtps2pi v2 {k1}, v0, 0, 8 // $t = \text{convert_to_fixed_point}(X = I+f)$
- vabsdiffps
The absolute difference between two single-precision floating-point numbers by zeroing the sign bit of the result on a floating point subtraction.
- vgetmantps v1 {k1}, Sf(v2/mt), imm8
Extracts float32 vector of mantissas from vector Sf(v2/mt) and stores the result in v1, under writemask. Vector float32 inputs are normalized according to imm8[1:0] and the sign of the mantissa results is set according to imm8[3:2]
- VFIXUPNANPS
Adds NaN source propagation.

7.14.4 New Packed Double-Precision Floating-Point Instructions

Table 7-11 describes the most recently introduced double-precision floating-point instructions.

Table 7-11. New Packed DP FP Instructions

Instruction	Description
vroundpd v1 {k1}, v2/mem, RC, expadj	This instruction rounds float64 vector Sf (v2/mt) and stores the result in v1, using RC as rounding control, and expadj to optionally adjust the exponent before conversion to nearest integer or fixed point value representation in float64, under writemask. VROUNDPD performs an element-by-element rounding of the result of the swizzle/broadcast/conversion from memory or float64 vector v2. The rounding result for each element is a float64 containing an integer or fixed-point value, depending on the value of expadj. The direction of rounding depends on the value of RC. The result is written into float64 vector v1. This instruction doesn't actually convert the result to an int64; the results are float64s, just like the input, but are float64s containing the integer or fixed-point values that result from the specified rounding and scaling.

Instruction	Description
vclampzpd v1 {k1}, v2, Sf (v3/mt)	This instruction clamps float64 vector v2 between zero and float64 vector Sf (v3/mt) and stores the result in v1, under writemask. VCLAMPZPD performs an element-by-element clamp of float64 vector v2 to the range between zero and the float64 vector result of the swizzle/broadcast/conversion process on memory or float64 vector v3. The result is written into float64 vector v1. Note that this instruction behaves differently from VCLAMPZPD when the third argument is negative. The order of Max followed by Min is required for positive clamping values.
VFIXUPPD	One with NaN propagation FIXUPNANPD; one without NaN propagation FIXUPPD.
vgetmantpd v1 {k1}, Sf(v2/mt), immd8	Extracts float64 vector of mantissas from vector Sf(v2/mt) and stores the result in v1, under writemask. Vector float64 inputs are normalized according to immd8[1:0] and the sign of the mantissa results is set according to immd8[3:2].

7.14.5 New Packed Int32 Instructions

Table 7-12 describes the recently introduced int32 instructions.

Table 7-12. Packed Int32 Instructions

Instruction	Description
VABSDIFPI	Determines the absolute difference between two 32-bit integer numbers either by leaving the result as is when subtract result is positive, or by inverting and adding 1 to the result when subtract result is negative.
vgetexppi v1 {k1}, Sf(v2/mt), immd8	Extracts int32 vector of exponents from float32 vector Sf(v2/mt) and stores the result in v1, under writemask. Float32 vector inputs are normalized according to immd8[1:0] and the sign of the exponents is returned as int32. If the source is SNaN, then QNaN will be returned. If the source is +INF, then the source will be returned.

8 Glossary and Abbreviations

Term	Description
ABI	Application Binary Interface
I	Autonomous Compute Node
AGI	Address Generation Interlock
AP	Application Program
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
BA	Base Address
BLCR*	Berkeley Lab Checkpoint Restore
BMC	Baseboard Management Controller
BSP	Bootstrap Processor
CL*	open Computing Language
CLD	Cache Line Disable
CMC	Channel Memory Controller
COI	Coprocessing Offload Infrastructure
CPI	Carry-Propagate Instructions
CPU	Central Processing Unit
CPUID	Central Processing Unit Identification
C/R	Check and Restore
CRI	Core Ring Interface
C-state	Core idle state
CSR	Configuration Status Register
DAPL	Direct Access Programming Library
DBS	Demand-Based Scaling
DMA	Direct Memory Access
DRAR	Descriptor Ring Attributes Register
DTD	Distributed Tag Directory
DP	Dual Precision
ECC	Error Correction Code
EMU	Extended Math Unit
EMON	Event Monitoring
ETC	Elapsed Time Counter
FBox	Part of the GBox, the FBox is the interface to the ring interconnect.
FIFO	First In, First Out
FMA	Fused Multiply and Add
FMS	Fused Multiply Subtract
FPU	Floating Point Unit
GBox	memory controller
GDDR	Graphics Double Data Rate
GDDR5	Graphics Double Data Rate, version 5
GDT	Global Descriptor Table
GOLS	Globally Owned, Locally Shared protocol
GP	General Protection
HCA	Host Channel Adaptor
HPC	High Performance Computing

Term	Description
HPI	Head Pointer Index
I ² C	("i-squared cee" or "i-two cee") Inter-Integrated Circuit
IA	Intel Architecture
IB	InfiniBand*
IBHCA	InfiniBand* Host Communication Adapter
ID	Identification
INVLPG	Invalidate TBL Entry
IpoIB	Internet Protocol over InfiniBand*
IPMI	Intelligent Platform Management Interface
iWARP	Internet Wide Area RDMA Protocol
Intel® MPSS	Intel® Manycore Platform Software Stack
I/O	Input/Output
IOAPIC	Input/Output Advanced Programmable Interrupt Controller
ISA	Instruction Set Architecture
LAPIC	Local Advanced Programmable Interrupt Controller
LKM	Loadable Kernel Modules
LRU	Least Recently Used
LSB	Linux* Standard Base
MBox	The request scheduler of the GBox.
MCA	Machine Check Architecture
MCE	Machine Check Exception
MESI	Modified, Exclusive, Shared, Invalid states
MOESI	Modified, Owner, Exclusive, Shared, Invalid states
MKL	Intel® Math Kernel Library
MMIO	Memory-Mapped Input/Output
MMX	
MPI	Message Passing Interface
MPSS	Intel® Many Integrated Core Architecture Platform Software Stack
MRU	Most Recently Used
MSI/x	
MSR	Model-Specific Register or Machine-Specific Register
NT	Non-Temporal
MTRR	Memory Type Range Register
mux	multiplexor
MYO	Intel® Mine Yours Ours Shared Virtual Memory
NFS	Network File System
OpenCL*	Open Computing Language
OFA*	Open Fabrics Alliance
OFED*	Open Fabrics Enterprise Distribution
PBox	
PC	Power Control
PCH	Platform Controller Hub
PCI Express*	Peripheral Component Interconnect Express
PCU	Power Control Unit
PEG port	

Term	Description
PDE	Psge Directory Entry
PF	Picker Function
PHP scripts	
P54C	Intel® Pentium® Processor
PM	Power Management or Process Manager
PMON	Performance Monitoring
PMU	Performance Monitoring Unit
PnP	Plug and Play
POST	Power-On Self-Test
P-state	Performance level states
RAS	Reliability Accessibility Serviceability
RDMA	Remote Direct Memory Access
RFO	Read For Ownership
RMA	Remote Memory Access
RS	Ring Stack
SAE	Suppress All Exceptions
SBox	System Box (Gen2 PCI Express* client logic)
SCIF	Symmetric Communication Interface
SC (SCM) protocol	Socket Connection Management
SDP	Software Development Platform
SDV	Software Development Vehicle
SEP	SEP is a utility that provides the sampling functionality used by VTune analyzer
SHM	Shared Memory
SI	Intel® Xeon Phi™ Coprocessor System Interface
SIMD	Single Instructions, Multiple Data
SM	Server Management
SMC	System Management Controller
SMP	Symmetric Multiprocessor
SMPY	System Memory Page Table
SP	Single Precision
SSE	Streaming SIMD Extensions
SSH	Secure Shell
SVID	System V Interface Definition
Sysfs	a virtual file system provided by Linux*
TCU	Transaction Control Unit
TPI	Tail Pointer Index
TSC	Timestamp Counter
TD	Tag Directory
TLB	Translation Lookaside Buffer
TMU	Thermal Monitoring Unit
TSC	Timestamp Counter
UC	Uncacheable
μDAPL	User DAPL
coprocessor OS	Micro Operating System
verbs	A programming interface
VIA	Virtual Interface Architecture
VMM	Virtual Machine Manager

Term	Description
VPU	Vector Processing Unit
VT-d	Intel® Virtualization Technology for Directed I/O
WB	Write Back
WC	Write Combining
WP	Write Protect
WT	Write Through

9 References

1. *Institute of Electrical and Electronics Engineers Standard for Floating Point Arithmetic.* (2008). www.ieee.org.
2. *Intel® Virtualization Technology for Directed I/O.* (2011). Intel.
3. *Knights Corner System Integration Tools User's Guide.* (2011). Intel.
4. *Intel® MPI Library for Intel® MIC Architecture.* (2011-2012).
5. *Intel® Xeon Phi™ Coprocessor Performance Monitoring Units, Document Number: 327357-001.* (2012). Intel.
6. *Aubrey Isle New Instruction Set Reference Manual Version 1.0.* (n.d.).
7. *Aubrey Isle Software Developer's Guide Version 1.1.* (n.d.).
8. Bratin Saha, X. Z. (2009). *Programming model for a heterogeneous x86 platform* (Vol. 44). Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.
9. <http://crd.lbl.gov/~jcduell/papers/blcr.pdf>. (n.d.).
10. https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html. (n.d.).
11. https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html. (n.d.).
12. <https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html#batch>. (n.d.).
13. Intel. (n.d.). Intel® 64 Architecture Processor Topology Enumeration.
14. *Intel® 64 and IA-32 Architectures Software Developer Manuals.* (n.d.). Intel Corporation.
15. *Intel® COI API Reference Manual Version 1.0.* (n.d.).
16. *Intel® Composer XE for Linux* Alpha including Intel(R) MIC Architecture Installation Guide and Release Notes for MIC (Release_Notes_mic_nnn.pdf).* (n.d.).
17. *Intel® Many Integrated Core (Intel® MIC) Profiling Guide White Paper (MIC_Profiling_Guide_nnnnn_rev1.pdf).* (n.d.).
18. *Intel® Many Integrated Core Architecture (Intel® MIC Architecture) Optimization Guide White Paper (Intel_MIC_Architecture_Optimization_Guide_xxx.pdf).* (n.d.).
19. *Intel® MIC Quick Start Developers Guide - Alpha 9.pdf.* (n.d.).
20. *Intel® SCIF API Reference Manual Version 1.0.* (n.d.).
21. *Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual (Reference Number: 327364).* (n.d.). Intel.
22. *MPI overview and specification - <http://www.mpi-forum.org/>.* (n.d.).
23. MPI, I. (n.d.). [TBD] Intel® MPI with the Intel® Xeon Phi™ coprocessors.
24. OFED --- <http://www.openfabrics.org/>. (n.d.).
25. OpenCL™ -- <http://www.khronos.org/opencv/>. (n.d.).
26. OpenMPI--- <http://www.open-mpi.org/>. (n.d.).
27. OpenMP™ -- <http://openmp.org/>. (n.d.).

Appendix: SBOX Control Register List

CR Name	MIMO Byte Addr Offset	MIMO Byte Addr End	Dec offset	Dword Offset	CR Size	No. Reg	Protection Level	Protection Method	Host Visibility	coprocessor OS/Ring Visibility	Initialized By	Reset Domain	Register Access	Description
OC_I2C_ICR	1000	1000	4096	0400	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM,I2C	I2C Control Register for coprocessor Over-clocking Unit
OC_I2C_ISR	1004	1004	4100	0401	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM,I2C	I2C Status Register for coprocessor Over-clocking Unit
OC_I2C_ISAR	1008	1008	4104	0402	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM,I2C	I2C Slave Address Register for coprocessor Over-clocking Unit
OC_I2C_IDBR	100C	100C	4108	0403	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM,I2C	I2C Data Buffer Register for coprocessor Over-clocking Unit
OC_I2C_IDMR	1010	1010	4112	0404	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM,I2C	I2C Bus Monitor Register for coprocessor Over-clocking Unit
THERMAL_STATUS	1018	1018	4120	0406	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Status and Log info for all the thermal interrupts
THERMAL_INTERRUPT_ENABLE	101C	101C	4124	0407	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_GRP	TRM	Register that controls the interrupt response to thermal events
MICROCONTROLLER_FAN_STATUS	1020	1020	4128	0408	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Upto data Status information from the Fan microcontroller
STATUS_FAN1	1024	1024	4132	0409	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	32 bit Status of Fan #1
STATUS_FAN2	1028	1028	4136	040A	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	32 bit Status of Fan #2
SPEED_OVERRIDE_FAN	102C	102C	4140	040B	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_GRP	TRM	32 bit Status of Fan #2
BOARD_TEMP1	1030	1030	4144	040C	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Temperature from Sensors 1 and 2 on coprocessor card
BOARD_TEMP2	1034	1034	4148	040D	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Temperature from Sensors 3 and 4 on coprocessor card
BOARD_VOLTAGE_SENSE	1038	1038	4152	040E	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Digitized value of Voltage sense input to coprocessor
CURRENT_DIE_TEMP0	103C	103C	4156	040F	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Consists of Current Die Temperatures of sensors 0 thru 2
CURRENT_DIE_TEMP1	1040	1040	4160	0410	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Consists of Current Die Temperatures of sensors 3 thru 5
CURRENT_DIE_TEMP2	1044	1044	4164	0411	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Consists of Current Die Temperatures of sensors 6 thru 8
MAX_DIE_TEMP0	1048	1048	4168	0412	32	1	Ring 0	Paging	No	Yes	RTL	CSR_GRP	TRM	Consists of Maximum Die Temperatures of sensors 0 thru 2

MAX_DIE_TEMP1	104C	104C	4172	0413	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Consists of Maximum Die Temperatures of sensors 3 thru 5
MAX_DIE_TEMP2	1050	1050	4176	0414	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Consists of Maximum Die Temperatures of sensors 6 thru 8
MIN_DIE_TEMP0	1054	1054	4180	0415	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Consists of Minimum Die Temperatures of sensors 0 thru 2
MIN_DIE_TEMP1	1058	1058	4184	0416	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Consists of Minimum Die Temperatures of sensors 3 thru 5
MIN_DIE_TEMP2	105C	105C	4188	0417	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Consists of Minimum Die Temperatures of sensors 6 thru 8
THERMAL_CONSTANTS	1060	1060	4192	0418	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Constants that define thermal response
THERMAL_TEST	1064	1064	4196	0419	32	1	Ring 0	Paging	No	Lockable	RTL	CSR_G RPA	TRM	System Interrupt Cause Set Register 1
GPU_HOT_CONFIG	1068	1068	4200	041A	32	1	Ring 0	Paging	Yes	Lockable	RTL	PERST	TRM	Configuration CSR for GPU HOT
NOM_PERFORMANCE	106C	106C	4204	041B	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Nominal Performance Monitors
PMU_PERIOD_SEL	1070	1070	4208	041C	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	PMU period
ELAPSED_TIME_LOW	1074	1074	4212	041D	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	"Elapsed Time Clock" Timer - lower 32 bits
ELAPSED_TIME_HIGH	1078	1078	4216	041E	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	"Elapsed Time Clock" Timer - higher 32 bits
THERMAL_STATUS_INTERRUPT	107C	107C	4220	041F	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Status and Log info for coprocessor new thermal interrupts
THERMAL_STATUS_2	1080	1080	4224	0420	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Thermal Status for coprocessor
THERMAL_TEST_2	1084	1084	4228	0421	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPA	TRM	Thermal Testability for coprocessor
EXT_TEMP_SETTINGS0	1090	1090	4240	0424	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #0
EXT_TEMP_SETTINGS1	1094	1094	4244	0425	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #1
EXT_TEMP_SETTINGS2	1098	1098	4248	0426	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #2
EXT_TEMP_SETTINGS3	109C	109C	4252	0427	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #3
EXT_TEMP_SETTINGS4	10A0	10A0	4256	0428	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #4
EXT_TEMP_SETTINGS5	10A4	10A4	4260	0429	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Setting - Sensor #5
EXT_CONTROLPARAMS0	10A8	10A8	4264	042A	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #0
EXT_CONTROLPARAMS1	10AC	10AC	4268	042B	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #1

EXT_CONTROLP ARAMS2	10B0	10B0	4272	042C	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #2
EXT_CONTROLP ARAMS3	10B4	10B4	4276	042D	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #3
EXT_CONTROLP ARAMS4	10B8	10B8	4280	042E	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #4
EXT_CONTROLP ARAMS5	10BC	10BC	4284	042F	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Parameters - Sensor #5
EXT_TEMP_STA TUS0	10C0	10C0	4288	0430	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Status - Sensor #0 ~ #2
EXT_TEMP_STA TUS1	10C4	10C4	4292	0431	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	External Thermal Sensor Status - Sensor #3 ~ #5
INT_FAN_STAT US	10C8	10C8	4296	0432	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Internal Thermal Sensor Status
INT_FAN_CONT ROLO	10CC	10CC	4300	0433	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Internal Thermal Sensor Setting/Parameters and FCU Configuration - 0
INT_FAN_CONT ROL1	10D0	10D0	4304	0434	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Internal Thermal Sensor Setting/Parameters and FCU Configuration - 1
INT_FAN_CONT ROL2	10D4	10D4	4308	0435	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	Internal Thermal Sensor Setting/Parameters and FCU Configuration - 2
FAIL_SAFE_STA TUS	2000	2000	8192	0800	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Fail Safe Image and Repair Status register
FAIL_SAFE_OFF SET	2004	2004	8196	0801	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Fail Safe Offset register
SW_OVR_CORE _DISABLE	2008	2008	8200	0802	32	1	Ring 0	Paging	No	Yes	OTHE R	PERST	TRM	Software controlled core Disable register that says how many cores are disabled - deprecated
CORE_DISABLE _STATUS	2010	2010	8208	0804	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	core Disable status register
FLASH_COMPO NENT	2018	2018	8216	0806	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Flash Component register
INVALID_INSTR 0	2020	2020	8224	0808	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Invalid Instruction register
INVALID_INSTR 1	2024	2024	8228	0809	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Invalid Instruction register
JEDECID	2030	2030	8240	080C	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	JEDEC ID register. This is a SW only register, SPI Controller reads these bits from the flash descriptor and reports the values in this register.
VENDOR_COM P_CAPP	2034	2034	8244	080D	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Vendor Specific component capabilities register. This is a SW only register, SPI Controller reads these bits from the flash descriptor and reports the values in this register.
POWER_ON_ST ATUS	2038	2038	8248	080E	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Power On status register
VALID_INSTR0	2040	2040	8256	0810	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch
VALID_INSTR1	2044	2044	8260	0811	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch

VALID_INSTR2	2048	2048	8264	0812	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch
VALID_INSTR_T YP0	2050	2050	8272	0814	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch
VALID_INSTR_T YP1	2054	2054	8276	0815	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch
VALID_INSTR_T YP2	2058	2058	8280	0816	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Scratch
HW_SEQ_STAT US	2070	2070	8304	081C	32	1	Ring 0	Paging	No	Yes	RTL	PERST	TRM	HW Sequence Flash Status Register
FLASH_ADDR	2078	2078	8312	081E	32	1	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	This is the starting byte linear address of a SPI read/write/erase command
FLASH_DATA	2090	20AC	8336	0824	32	8	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	Flash Data registers
SW_SEQ_STAT US	20B0	20B0	8368	082C	32	1	Ring 0	Paging	No	Lock able	Flash, RTL	PERST	TRM	SW Sequence Flash Status Register
SW_SEQ_CTRL	20B4	20B4	8372	082D	32	1	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	SW Sequence Flash Control Register
OPCODE_TYP_C ONFIG	20B8	20B8	8376	082E	32	1	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	This register specifies information the type of opcode. Entries in this register correspond to the entries in the Opcode Menu Configuration register.
OPCODE_MEN U0	20C0	20C0	8384	0830	32	1	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	This register lists the allowable opcodes. Software programs an SPI opcode into this field for use when initiating SPI commands through the Control Register. Flash opcodes that are tagged as invalid via the flash descriptor will immediate assert the PERR bit in the Hardware Sequencing Flash Status register.
OPCODE_MEN U1	20C4	20C4	8388	0831	32	1	Ring 0	Paging	No	Lock able	RTL	PERST	TRM	This register lists the allowable opcodes. Software programs an SPI opcode into this field for use when initiating SPI commands through the Control Register. Flash opcodes that are tagged as invalid via the flash descriptor will immediate assert the PERR bit in the Hardware Sequencing Flash Status register.

OPCODE_MEN U2	20C8	20C8	8392	0832	32	1	Ring 0	Paging	No	Lock able	FLASH	PERST	TRM	This register lists the allowable opcodes. Software programs an SPI opcode into this field for use when initiating SPI commands through the Control Register. Flash opcodes that are tagged as invalid via the flash descriptor will immediate assert the PERR bit in the Hardware Sequencing Flash Status register.
FAIL_SAFE_REP AIR_OFFSET	20CC	20CC	8396	0833	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Fail Safe Offset for Repair Sector register
AGENT_DISABL E_FLASH0	20D0	20D0	8400	0834	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
AGENT_DISABL E_FLASH1	20D4	20D4	8404	0835	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
AGENT_DISABL E_FLASH2	20D8	20D8	8408	0836	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
AGENT_DISABL E_FLASH3	20DC	20DC	8412	0837	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
AGENT_DISABL E_FLASH4	20E0	20E0	8416	0838	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
AGENT_DISABL E_FLASH5	20E4	20E4	8420	0839	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	Agent Disable Value from Flash register
SW_OVR_AGEN T_DISABLE0	20E8	20E8	8424	083A	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SW_OVR_AGEN T_DISABLE1	20EC	20EC	8428	083B	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SW_OVR_AGEN T_DISABLE2	20F0	20F0	8432	083C	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SW_OVR_AGEN T_DISABLE3	20F4	20F4	8436	083D	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SW_OVR_AGEN T_DISABLE4	20F8	20F8	8440	083E	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SW_OVR_AGEN T_DISABLE5	20FC	20FC	8444	083F	32	1	Ring 0	Paging	No	Lock able	OTHE R	PERST	TRM	Software controlled (and TAP override) Agent Disable register that says how many agent are disabled
SPI_FSM	2100	2100	8448	0840	32	1	Ring 0	Paging	No	Yes	FLASH	PERST	TRM	SPI FSM Status register
SOFT_STRAP_0	2400	2400	9216	0900	32	1	Ring 0	Paging	No	Lock able	FLASH	CSR_G RPA	TRM	soft strap registers - 0

SOFT_STRAP_1	2404	2404	9220	0901	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 1
SOFT_STRAP_2	2408	2408	9224	0902	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 2
SOFT_STRAP_3	240C	240C	9228	0903	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 3
SOFT_STRAP_4	2410	2410	9232	0904	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 4
SOFT_STRAP_5	2414	2414	9236	0905	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 5
SOFT_STRAP_6	2418	2418	9240	0906	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 6
SOFT_STRAP_7	241C	241C	9244	0907	32	1	Ring 0	Paging	No	Lockable	FLASH	CSR_G RPA	TRM	soft strap registers - 7
STARV_THRSH	3010	3010	12304	0C04	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Starvation thresholds per T2P FIFO.
TX_ARB_PRIOR	3014	3014	12308	0C05	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Transmit Arbiter Priorities
TX_ARB_STRV_PRIOR	3018	3018	12312	0C06	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Transmit Arbiter Starvation Priorities.
TX_BURST_LEN	301C	301C	12316	0C07	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Transmit Burst Length Per FIFO
PCI_EP_DBG_CAPT	3024	3024	12324	0C09	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	PCI endpoint debug capture
TX_ARB_FIX	3028	3028	12328	0C0A	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Fixes Tx arbiter onto one selection
GH_SCRATCH	303C	303C	12348	0C0F	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Scratch register
EP_TX_CONTR_OL	3044	3044	12356	0C11	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Endpoint Transmit Control
DFTUR	304C	304C	12364	0C13	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Chicken Bits for GHost.
SPARE	3050	3050	12368	0C14	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Spares for H/W debug/fixes.
RX_HALT	306C	306C	12396	0C1B	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Halts reception of all PCI packets.
MCX_CTL_LO	3090	3090	12432	0C24	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	MCX CTL LOW
MCX_STATUS_LO	3098	3098	12440	0C26	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	MCX Status
MCX_STATUS_HI	309C	309C	12444	0C27	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	MCX Status HI
MCX_ADDR_LO	30A0	30A0	12448	0C28	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	MCX Addr Low
MCX_ADDR_HI	30A4	30A4	12452	0C29	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	MCX Addr High
MCX_MISC	30A8	30A8	12456	0C2A	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Machine Check Miscellaneous #1
MCX_MISC2	30AC	30AC	12460	0C2B	32	1	Ring 0	Paging	No	Yes	RTL	PERST	CRU,TRM	Machine Check Miscellaneous #2
SMPT00	3100	3100	12544	0C40	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 00. *Not Host accessible on A-step.
SMPT01	3104	3104	12548	0C41	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 01.
SMPT02	3108	3108	12552	0C42	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 02.
SMPT03	310C	310C	12556	0C43	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 03.

SMPT04	3110	3110	12560	0C44	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 04.
SMPT05	3114	3114	12564	0C45	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 05.
SMPT06	3118	3118	12568	0C46	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 06.
SMPT07	311C	311C	12572	0C47	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 07.
SMPT08	3120	3120	12576	0C48	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 08.
SMPT09	3124	3124	12580	0C49	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 09.
SMPT10	3128	3128	12584	0C4A	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 10.
SMPT11	312C	312C	12588	0C4B	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 11.
SMPT12	3130	3130	12592	0C4C	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 12.
SMPT13	3134	3134	12596	0C4D	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 13.
SMPT14	3138	3138	12600	0C4E	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 14.
SMPT15	313C	313C	12604	0C4F	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 15.
SMPT16	3140	3140	12608	0C50	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 16.
SMPT17	3144	3144	12612	0C51	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 17.
SMPT18	3148	3148	12616	0C52	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 18.
SMPT19	314C	314C	12620	0C53	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 19.
SMPT20	3150	3150	12624	0C54	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 20.
SMPT21	3154	3154	12628	0C55	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 21.
SMPT22	3158	3158	12632	0C56	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 22.
SMPT23	315C	315C	12636	0C57	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 23.
SMPT24	3160	3160	12640	0C58	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 24.
SMPT25	3164	3164	12644	0C59	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 25.
SMPT26	3168	3168	12648	0C5A	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 26.
SMPT27	316C	316C	12652	0C5B	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 27.
SMPT28	3170	3170	12656	0C5C	32	1	Ring 0	Paging	Yes/ No*	Yes	RTL	PERST	CRU,T RM	System Memory Page Table, Page 28.

SMPT29	3174	3174	12660	0C5D	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 29.
SMPT30	3178	3178	12664	0C5E	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 30.
SMPT31	317C	317C	12668	0C5F	32	1	Ring 0	Paging	Yes/No*	Yes	RTL	PERST	CRU,TRM	System Memory Page Table, Page 31.
PDAT	3200	3200	12800	0C80	32	1	Ring 0	Paging	Yes	Lockable	RTL	PERST	CRU,TRM	LDAT Primary DAT
SDAT	3208	3208	12808	0C82	32	1	Ring 0	Paging	Yes	Lockable	RTL	PERST	CRU,TRM	LDAT Secondary DAT
DATOUT	3210	3210	12816	0C84	32	1	Ring 0	Paging	Yes	Lockable	RTL	PERST	CRU,TRM	LDAT data out
DATINO	3218	3218	12824	0C86	32	1	Ring 0	Paging	Yes	Lockable	RTL	PERST	CRU,TRM	LDAT data in
TAP_IDCODE	3A00	3A00	14848	0E80	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	TAP IDCODE
TAP_SUBSTEP	3A04	3A04	14852	0E81	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	TAP substepping
CR_RX_BUF_STATUS	3A08	3A08	14856	0E82	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	VC0 buffer status
CR_RX_LFSR_FTS	3A0C	3A0C	14860	0E83	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	RX LFSR N_FTS
CR_P_CONSUMED	3A10	3A10	14864	0E84	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT CONSUMED COUNTER POSTED
CR_NP_CONSUMED	3A14	3A14	14868	0E85	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT CONSUMED COUNTER NON-POSTED
CR_CPL_CONSUMED	3A18	3A18	14872	0E86	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT CONSUMED COUNTER COMPLETION
CR_P_LIMIT	3A1C	3A1C	14876	0E87	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT LIMIT COUNTER POSTED
CR_NP_LIMIT	3A20	3A20	14880	0E88	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT LIMIT COUNTER NON-POSTED
CR_CPL_LIMIT	3A24	3A24	14884	0E89	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT LIMIT COUNTER COMPLETION
CR_P_ALLOCATED	3A28	3A28	14888	0E8A	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT ALLOCATED COUNTER POSTED
CR_NP_ALLOCATED	3A2C	3A2C	14892	0E8B	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT ALLOCATED COUNTER NON-POSTED
CR_CPL_ALLOCATED	3A30	3A30	14896	0E8C	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	CREDIT ALLOCATED COUNTER COMPLETION
PSMI_ERR_STATUS	3A34	3A34	14900	0E8D	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	PSMI ERROR STATUS
PSMI_CONFIG	3A38	3A38	14904	0E8E	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	PSMI CONFIG
PSMI_SEQ_NUM	3A3C	3A3C	14908	0E8F	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	CRU,TRM	TX/RX SEQ NUM
REAL_TIME_CLOCK	3A44	3A44	14916	0E91	32	1	Ring 0	Paging	Yes	Yes				Sample of real time clock from the endpoint
RGCR	4010	4010	16400	1004	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Reset Global Control
DSTAT	4014	4014	16404	1005	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Device Status Register

PWR_TIMEOUT	4018	4018	16408	1006	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Timeout value used in the reset engine to timeout various reset external events. Slot Power, GrpBPwrGd assertion, Connector status timeout period. The number in this register is used to shift 1 N places. N has to be less than 32
RDBCTL	4020	4020	16416	1008	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPA	TRM	Reset Debug Control Register
RDBSTAT	4024	4024	16420	1009	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPA	TRM	Reset Debug Status Register
CurrentRatio	402C	402C	16428	100B	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	The expected MCLK Ratio that is sent to the corepll
iccOverClock0	4040	4040	16448	1010	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	core Overclocking Only, protected by overclocking disable fuse (OverclockDis)
iccOverClock1	4044	4044	16452	1011	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Mem Overclocking Only, protected by overclocking disable fuse (OverclockDis)
iccOverClock2	4048	4048	16456	1012	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Display Bend1, Always open, no fuse protection
iccOverClock3	404C	404C	16460	1013	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	TRM	Display Bend2, Always open, no fuse protection
COREFREQ	4100	4100	16640	1040	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	Core Frequency
COREVOLT	4104	4104	16644	1041	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	Core Voltage
MEMORYFREQ	4108	4108	16648	1042	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	Memory Frequency
MEMVOLT	410C	410C	16652	1043	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	Memory Voltage
SVIDControl	4110	4110	16656	1044	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	SVID VR12/MVP7 Control Interface Register
PCUControl	4114	4114	16660	1045	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	SNAR F	Power Control Unit Register
HostPMState	4118	4118	16664	1046	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	SNAR F	Host PM scratch registers
μOSPMState	411C	411C	16668	1047	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	SNAR F	coprocessor OS PM Scratch registers
C3WakeUp_Timer	4120	4120	16672	1048	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	C3 Wakeup Timer Control for autoC3
L1_Entry_Timer	4124	4124	16676	1049	32	1	Ring 0	Paging	Yes	Yes	RTL, OTHER	PERST, HOT	SNAR F	L1 Entry Timer
C3_Timers	4128	4128	16680	104A	32	1	Ring 0	Paging	Yes	Yes	RTL, OTHER	PERST	SNAR F	C3 Entry and Exit Timers
coprocessor OS_PCUCONTROL	412C	412C	16684	104B	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	SNAR F	coprocessor OS PCU Control CSR.. i.e. not for host consumption
SVIDSTATUS	4130	4130	16688	104C	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	SVID Status
COMPONENTID	4134	4134	16692	104D	32	1	Ring 0	Paging	Yes	Yes		PERST	SNAR F	COMPONENTID
GBOXPMControl	413C	413C	16700	104F	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	SNAR F	GBOX PM Control

GPIO_Input_Stat atus	4140	4140	16704	1050	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, OTHER	SNAR F	GPIO Input Status
GPIO_Output_C ontrol	4144	4144	16708	1051	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	GPIO Output Control
EMON_Control	4160	4160	16736	1058	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	EMON Control Register
EMON_Counter 0	4164	4164	16740	1059	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST	SNAR F	EMON Counter 0
PCIE_VENDOR_ ID_DEVICE_ID	5800	5800	22528	1600	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_COM MAND_AND_ST ATUS	5804	5804	22532	1601	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_REVIS ION_ID_AND_C _OX8	5808	5808	22536	1602	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_CACH E_LINE_SIZE_L _OXC	580C	580C	22540	1603	32	1	Ring 0	Paging	No	Yes				
PCIE_MEMORY_ BAR_0	5810	5810	22544	1604	32	1	Ring 0	Paging	No	Yes				
PCIE_UPPER_D WORD_OF_ME MOR_OX14	5814	5814	22548	1605	32	1	Ring 0	Paging	No	Yes				
PCIE_IO_BAR_2	5818	5818	22552	1606	32	1	Ring 0	Paging	No	Yes				
PCIE_MEMORY_ BAR_1	5820	5820	22560	1608	32	1	Ring 0	Paging	No	Yes				
PCIE_UPPER_D WORD_OF_ME MOR_OX24	5824	5824	22564	1609	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_SUBS YSTEM	582C	582C	22572	160B	32	1	Ring 0	Paging	No	Yes				
PCIE_EXPANSIO N_ROM_BAR	5830	5830	22576	160C	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_CAPA BILITIES_POINT ER	5834	5834	22580	160D	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_INTER RUPT_LINE_PIN	583C	583C	22588	160F	32	1	Ring 0	Paging	No	Yes				
PCIE_PCI_PM_C APABILITY	5844	5844	22596	1611	32	1	Ring 0	Paging	No	Yes				
PCIE_PM_STAT US_AND_CONT RO_OX48	5848	5848	22600	1612	32	1	Ring 0	Paging	No	Yes				
PCIE_PCIE_CAP ABILITY	584C	584C	22604	1613	32	1	Ring 0	Paging	No	Yes				
PCIE_DEVICE_C APABILITY	5850	5850	22608	1614	32	1	Ring 0	Paging	No	Yes				
PCIE_DEVICE_C ONTROL_AND_ STATUS	5854	5854	22612	1615	32	1	Ring 0	Paging	No	Yes				
PCIE_LINK_CAP ABILITY	5858	5858	22616	1616	32	1	Ring 0	Paging	No	Yes				
PCIE_LINK_CON TROL_AND_STA _OX5C	585C	585C	22620	1617	32	1	Ring 0	Paging	No	Yes				
PCIE_DEVICE_C APABILITY_2	5870	5870	22640	161C	32	1	Ring 0	Paging	No	Yes				
PCIE_DEVICE_C ONTROL_AND_ S_OX74	5874	5874	22644	161D	32	1	Ring 0	Paging	No	Yes				
PCIE_LINK_CON TROL_AND_STA TUS_2	587C	587C	22652	161F	32	1	Ring 0	Paging	No	Yes				

PCIE_MSI_CAP ABILITY	5888	5888	22664	1622	32	1	Ring 0	Paging	No	Yes
PCIE_MESSAGE _ADDRESS	588C	588C	22668	1623	32	1	Ring 0	Paging	No	Yes
PCIE_MESSAGE _UPPER_ADDR ESS	5890	5890	22672	1624	32	1	Ring 0	Paging	No	Yes
PCIE_MESSAGE _DATA	5894	5894	22676	1625	32	1	Ring 0	Paging	No	Yes
PCIE_MSIX_CA PABILITY	5898	5898	22680	1626	32	1	Ring 0	Paging	No	Yes
PCIE_MSIX_TA BLE_OFFSET_BI R	589C	589C	22684	1627	32	1	Ring 0	Paging	No	Yes
PCIE_PBA_OFFS ET_BIR	58A0	58A0	22688	1628	32	1	Ring 0	Paging	No	Yes
PCIE_ADVANCE D_ERROR_CAP ABILITY	5900	5900	22784	1640	32	1	Ring 0	Paging	No	Yes
PCIE_UNCORRE CTABLE_ERROR _0X104	5904	5904	22788	1641	32	1	Ring 0	Paging	No	Yes
PCIE_UNCORRE CTABLE_ERROR _MASK	5908	5908	22792	1642	32	1	Ring 0	Paging	No	Yes
PCIE_UNCORRE CTABLE_ERROR _0X10C	590C	590C	22796	1643	32	1	Ring 0	Paging	No	Yes
PCIE_CORRECT ABLE_ERROR_S TATUS	5910	5910	22800	1644	32	1	Ring 0	Paging	No	Yes
PCIE_CORRECT ABLE_ERROR_ MASK	5914	5914	22804	1645	32	1	Ring 0	Paging	No	Yes
PCIE_ADVANCE D_ERROR_CAP A_0X118	5918	5918	22808	1646	32	1	Ring 0	Paging	No	Yes
PCIE_ERROR_H EADER_LOG_D WORD_0	591C	591C	22812	1647	32	1	Ring 0	Paging	No	Yes
PCIE_ERROR_H EADER_LOG_D WORD_1	5920	5920	22816	1648	32	1	Ring 0	Paging	No	Yes
PCIE_ERROR_H EADER_LOG_D WORD_2	5924	5924	22820	1649	32	1	Ring 0	Paging	No	Yes
PCIE_ERROR_H EADER_LOG_D WORD_3	5928	5928	22824	164A	32	1	Ring 0	Paging	No	Yes
PCIE_LTSSM_ST ATE_CONTROL	5C00	5C00	23552	1700	32	1	Ring 0	Paging	No	Yes
PCIE_LTSSM_ST ATE_STATUS	5C04	5C04	23556	1701	32	1	Ring 0	Paging	No	Yes
PCIE_SKIP_FRE QUENCY_TIME R	5C08	5C08	23560	1702	32	1	Ring 0	Paging	No	Yes
PCIE_LANE_SEL ECT	5C0C	5C0C	23564	1703	32	1	Ring 0	Paging	No	Yes
PCIE_LANE_DES KEW	5C10	5C10	23568	1704	32	1	Ring 0	Paging	No	Yes
PCIE_RECEIVER _ERROR_STATU S	5C14	5C14	23572	1705	32	1	Ring 0	Paging	No	Yes
PCIE_LANE_NU MBER_CONTRO L	5C18	5C18	23576	1706	32	1	Ring 0	Paging	No	Yes

PCIE_N_FTS_CO NTROL	5C1C	5C1C	23580	1707	32	1	Ring 0	Paging	No	Yes
PCIE_LINK_STA TUS	5C20	5C20	23584	1708	32	1	Ring 0	Paging	No	Yes
PCIE_SYNC_BYP ASS	5C2C	5C2C	23596	170B	32	1	Ring 0	Paging	No	Yes
PCIE_ACK_REPL AY_TIMEOUT	5C38	5C38	23608	170E	32	1	Ring 0	Paging	No	Yes
PCIE_SEQUENC E_NUMBER_ST ATUS	5C3C	5C3C	23612	170F	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_PM _TIMER	5C50	5C50	23632	1714	32	1	Ring 0	Paging	No	Yes
PCIE_PME_TIM EOUT	5C54	5C54	23636	1715	32	1	Ring 0	Paging	No	Yes
PCIE_ASPM_L1 _TIMER	5C58	5C58	23640	1716	32	1	Ring 0	Paging	No	Yes
PCIE_ASPM_RE QUEST_TIMER	5C5C	5C5C	23644	1717	32	1	Ring 0	Paging	No	Yes
PCIE_ASPM_L1 _DISABLE	5C60	5C60	23648	1718	32	1	Ring 0	Paging	No	Yes
PCIE_ADVISOR Y_ERROR_CON TROL	5C68	5C68	23656	171A	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_ID	5C70	5C70	23664	171C	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_CLA SSCODE	5C74	5C74	23668	171D	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_SU BSYSTEM_ID	5C78	5C78	23672	171E	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_DE VICE_CAPABI LITY	5C7C	5C7C	23676	171F	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_LIN K_CAPABILITY	5C80	5C80	23680	1720	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_PM _CAPABILITY	5C88	5C88	23688	1722	32	1	Ring 0	Paging	No	Yes
PCIE_GPEX_LIN K_CONTROL_ST ATUS	5C9C	5C9C	23708	1727	32	1	Ring 0	Paging	No	Yes
PCIE_ERROR_C OUNTER	5CAC	5CAC	23724	172B	32	1	Ring 0	Paging	No	Yes
PCIE_CONFIGU RATION_READY	5CB0	5CB0	23728	172C	32	1	Ring 0	Paging	No	Yes
PCIE_FC_UPDA TE_TIMEOUT	5CB8	5CB8	23736	172E	32	1	Ring 0	Paging	No	Yes
PCIE_FC_UPDA TE_TIMER	5CBC	5CBC	23740	172F	32	1	Ring 0	Paging	No	Yes
PCIE_LOAD_VC _BUFFER_SIZE	5CC8	5CC8	23752	1732	32	1	Ring 0	Paging	No	Yes
PCIE_VC_BUFFE R_SIZE_THRES HOLD	5CCC	5CCC	23756	1733	32	1	Ring 0	Paging	No	Yes
PCIE_VC_BUFFE R_SELECT	5CD0	5CD0	23760	1734	32	1	Ring 0	Paging	No	Yes
PCIE_BAR_ENA BLE	5CD4	5CD4	23764	1735	32	1	Ring 0	Paging	No	Yes
PCIE_BAR_SIZE _LOWER_DWO RD	5CD8	5CD8	23768	1736	32	1	Ring 0	Paging	No	Yes
PCIE_BAR_SIZE _UPPER_DWOR D	5CDC	5CDC	23772	1737	32	1	Ring 0	Paging	No	Yes

PCIE_BAR_SELECT	5CE0	5CE0	23776	1738	32	1	Ring 0	Paging	No	Yes	
PCIE_CREDIT_COUNTER_SELECT	5CE4	5CE4	23780	1739	32	1	Ring 0	Paging	No	Yes	
PCIE_CREDIT_COUNTER_STATUSES	5CE8	5CE8	23784	173A	32	1	Ring 0	Paging	No	Yes	
PCIE_TLP_HEADER_SELECT	5CEC	5CEC	23788	173B	32	1	Ring 0	Paging	No	Yes	
PCIE_TLP_HEADER_LOG_DWORD_0	5CF0	5CF0	23792	173C	32	1	Ring 0	Paging	No	Yes	
PCIE_TLP_HEADER_LOG_DWORD_1	5CF4	5CF4	23796	173D	32	1	Ring 0	Paging	No	Yes	
PCIE_TLP_HEADER_LOG_DWORD_2	5CF8	5CF8	23800	173E	32	1	Ring 0	Paging	No	Yes	
PCIE_TLP_HEADER_LOG_DWORD_3	5CFC	5CFC	23804	173F	32	1	Ring 0	Paging	No	Yes	
PCIE_RELAXED_ORDERING_CONTROL	5D00	5D00	23808	1740	32	1	Ring 0	Paging	No	Yes	
PCIE_BAR_PREFETCH	5D04	5D04	23812	1741	32	1	Ring 0	Paging	No	Yes	
PCIE_FC_CHECK_CONTROL	5D08	5D08	23816	1742	32	1	Ring 0	Paging	No	Yes	
PCIE_FC_UPDATE_TIMER_TRAFFIC	5D18	5D18	23832	1746	32	1	Ring 0	Paging	No	Yes	
PCIE_UNCORRECTABLE_ERROR_0X5F0	5DF0	5DF0	24048	177C	32	1	Ring 0	Paging	No	Yes	
PCIE_CLOCK_GATING_CONTROL	5E1C	5E1C	24092	1787	32	1	Ring 0	Paging	No	Yes	
PCIE_GEN2_CONTROL_CSR	5E40	5E40	24128	1790	32	1	Ring 0	Paging	No	Yes	
PCIE_GPEX_IP_RELEASE_VERSION	5EFC	5EFC	24316	17BF	32	1	Ring 0	Paging	No	Yes	
AFEBND0_CFG0	6000	6000	24576	1800	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 0, 0x6000
AFEBND1_CFG0	6004	6004	24580	1801	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 1, 0x6004
AFEBND2_CFG0	6008	6008	24584	1802	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 2, 0x6008
AFEBND3_CFG0	600C	600C	24588	1803	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 3, 0x600C
AFEBND4_CFG0	6010	6010	24592	1804	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 4, 0x6010
AFEBND5_CFG0	6014	6014	24596	1805	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 5, 0x6014
AFEBND6_CFG0	6018	6018	24600	1806	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 6, 0x6018
AFEBND7_CFG0	601C	601C	24604	1807	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 0, Bundle 7, 0x601C

AFEBND0_CFG1	6020	6020	24608	1808	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 0, 0x6020
AFEBND1_CFG1	6024	6024	24612	1809	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 1, 0x6024
AFEBND2_CFG1	6028	6028	24616	180A	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 2, 0x6028
AFEBND3_CFG1	602C	602C	24620	180B	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 3, 0x602C
AFEBND4_CFG1	6030	6030	24624	180C	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 4, 0x6030
AFEBND5_CFG1	6034	6034	24628	180D	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 5, 0x6034
AFEBND6_CFG1	6038	6038	24632	180E	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 6, 0x6038
AFEBND7_CFG1	603C	603C	24636	180F	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 1, Bundle 7, 0x603C
AFEBND0_CFG2	6040	6040	24640	1810	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 0, 0x6040
AFEBND1_CFG2	6044	6044	24644	1811	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 1, 0x6044
AFEBND2_CFG2	6048	6048	24648	1812	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 2, 0x6048
AFEBND3_CFG2	604C	604C	24652	1813	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 3, 0x604C
AFEBND4_CFG2	6050	6050	24656	1814	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 4, 0x6050
AFEBND5_CFG2	6054	6054	24660	1815	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 5, 0x6054
AFEBND6_CFG2	6058	6058	24664	1816	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 6, 0x6058
AFEBND7_CFG2	605C	605C	24668	1817	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 2, Bundle 7, 0x605C
AFEBND0_CFG3	6060	6060	24672	1818	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 0, 0x6060
AFEBND1_CFG3	6064	6064	24676	1819	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 1, 0x6064
AFEBND2_CFG3	6068	6068	24680	181A	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 2, 0x6068
AFEBND3_CFG3	606C	606C	24684	181B	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 3, 0x606C
AFEBND4_CFG3	6070	6070	24688	181C	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 4, 0x6070
AFEBND5_CFG3	6074	6074	24692	181D	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 5, 0x6074
AFEBND6_CFG3	6078	6078	24696	181E	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 6, 0x6078

AFEBND7_CFG3	607C	607C	24700	181F	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 3, Bundle 7, 0x607C
AFEBND0_CFG4	6080	6080	24704	1820	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 0, 0x6080
AFEBND1_CFG4	6084	6084	24708	1821	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 1, 0x6084
AFEBND2_CFG4	6088	6088	24712	1822	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 2, 0x6088
AFEBND3_CFG4	608C	608C	24716	1823	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 3, 0x608C
AFEBND4_CFG4	6090	6090	24720	1824	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 4, 0x6090
AFEBND5_CFG4	6094	6094	24724	1825	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 5, 0x6094
AFEBND6_CFG4	6098	6098	24728	1826	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 6, 0x6098
AFEBND7_CFG4	609C	609C	24732	1827	32	1	Ring 0	Paging	No	Yes	AFE Bundle Config Register 4, Bundle 7, 0x609C
AFEBND0_LBC5	60A0	60A0	24736	1828	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 0, 0x60A0
AFEBND1_LBC5	60A4	60A4	24740	1829	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 1, 0x60A4
AFEBND2_LBC5	60A8	60A8	24744	182A	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 2, 0x60A8
AFEBND3_LBC5	60AC	60AC	24748	182B	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 3, 0x60AC
AFEBND4_LBC5	60B0	60B0	24752	182C	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 4, 0x60B0
AFEBND5_LBC5	60B4	60B4	24756	182D	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 5, 0x60B4
AFEBND6_LBC5	60B8	60B8	24760	182E	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 6, 0x60B8
AFEBND7_LBC5	60BC	60BC	24764	182F	32	1	Ring 0	Paging	No	Yes	AFE Bundle Load Bus Control Register 5, Bundle 7, 0x60BC
AFELN0_CFG0	60C0	60C0	24768	1830	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 0, 0x60C0
AFELN1_CFG0	60C4	60C4	24772	1831	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 1, 0x60C4
AFELN2_CFG0	60C8	60C8	24776	1832	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 2, 0x60C8
AFELN3_CFG0	60CC	60CC	24780	1833	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 3, 0x60CC
AFELN4_CFG0	60D0	60D0	24784	1834	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 4, 0x60D0
AFELN5_CFG0	60D4	60D4	24788	1835	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 5, 0x60D4

AFELN6_CFG0	60D8	60D8	24792	1836	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 6, 0x60D8
AFELN7_CFG0	60DC	60DC	24796	1837	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 7, 0x60DC
AFELN8_CFG0	60E0	60E0	24800	1838	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 8, 0x60E0
AFELN9_CFG0	60E4	60E4	24804	1839	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 9, 0x60E4
AFELN10_CFG0	60E8	60E8	24808	183A	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 10, 0x60E8
AFELN11_CFG0	60EC	60EC	24812	183B	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 11, 0x60EC
AFELN12_CFG0	60F0	60F0	24816	183C	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 12, 0x60F0
AFELN13_CFG0	60F4	60F4	24820	183D	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 13, 0x60F4
AFELN14_CFG0	60F8	60F8	24824	183E	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 14, 0x60F8
AFELN15_CFG0	60FC	60FC	24828	183F	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 0, Lane 15, 0x60FC
AFELN0_CFG1	6100	6100	24832	1840	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 0, 0x6100
AFELN1_CFG1	6104	6104	24836	1841	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 1, 0x6104
AFELN2_CFG1	6108	6108	24840	1842	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 2, 0x6108
AFELN3_CFG1	610C	610C	24844	1843	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 3, 0x610C
AFELN4_CFG1	6110	6110	24848	1844	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 4, 0x6110
AFELN5_CFG1	6114	6114	24852	1845	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 5, 0x6114
AFELN6_CFG1	6118	6118	24856	1846	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 6, 0x6118
AFELN7_CFG1	611C	611C	24860	1847	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 7, 0x611C
AFELN8_CFG1	6120	6120	24864	1848	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 8, 0x6120
AFELN9_CFG1	6124	6124	24868	1849	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 9, 0x6124
AFELN10_CFG1	6128	6128	24872	184A	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 10, 0x6128
AFELN11_CFG1	612C	612C	24876	184B	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 11, 0x612C
AFELN12_CFG1	6130	6130	24880	184C	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 12, 0x6130
AFELN13_CFG1	6134	6134	24884	184D	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 13, 0x6134
AFELN14_CFG1	6138	6138	24888	184E	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 14, 0x6138

AFELN15_CFG1	613C	613C	24892	184F	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 1, Lane 15, 0x613C
AFELN0_CFG2	6140	6140	24896	1850	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 0, 0x6140
AFELN1_CFG2	6144	6144	24900	1851	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 1, 0x6144
AFELN2_CFG2	6148	6148	24904	1852	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 2, 0x6148
AFELN3_CFG2	614C	614C	24908	1853	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 3, 0x614C
AFELN4_CFG2	6150	6150	24912	1854	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 4, 0x6150
AFELN5_CFG2	6154	6154	24916	1855	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 5, 0x6154
AFELN6_CFG2	6158	6158	24920	1856	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 6, 0x6158
AFELN7_CFG2	615C	615C	24924	1857	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 7, 0x615C
AFELN8_CFG2	6160	6160	24928	1858	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 8, 0x6160
AFELN9_CFG2	6164	6164	24932	1859	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 9, 0x6164
AFELN10_CFG2	6168	6168	24936	185A	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 10, 0x6168
AFELN11_CFG2	616C	616C	24940	185B	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 11, 0x616C
AFELN12_CFG2	6170	6170	24944	185C	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 12, 0x6170
AFELN13_CFG2	6174	6174	24948	185D	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 13, 0x6174
AFELN14_CFG2	6178	6178	24952	185E	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 14, 0x6178
AFELN15_CFG2	617C	617C	24956	185F	32	1	Ring 0	Paging	No	Yes	AFE Lane Config Register 2, Lane 15, 0x617C
AFECMN_CFG0	6180	6180	24960	1860	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 0, 0x6180
AFECMN_CFG1	6184	6184	24964	1861	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 1, 0x6184
AFECMN_CFG2	6188	6188	24968	1862	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 2, 0x6188
AFECMN_CFG3	618C	618C	24972	1863	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 3, 0x618C
AFECMN_CFG4	6190	6190	24976	1864	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 4, 0x6190
AFECMN_CFG5	6194	6194	24980	1865	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 5, 0x6194
AFECMN_CFG6	6198	6198	24984	1866	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 6, 0x6198
AFECMN_CFG7	619C	619C	24988	1867	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 7, 0x619C

AFECMN_CFG8	61A0	61A0	24992	1868	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 8, 0x61A0
AFECMN_CFG9	61A4	61A4	24996	1869	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 9, 0x61A4
AFECMN_CFG10	61A8	61A8	25000	186A	32	1	Ring 0	Paging	No	Yes	AFE Common Config Register 10, 0x61A8
AFERD_STAT0	61AC	61AC	25004	186B	32	1	Ring 0	Paging	No	Yes	AFE Read Status Register 0, 0x61AC
AFERD_STAT1	61B0	61B0	25008	186C	32	1	Ring 0	Paging	No	Yes	AFE Read Status Register 1, 0x61B0
AFERD_STAT2	61B4	61B4	25012	186D	32	1	Ring 0	Paging	No	Yes	AFE Read Status Register 2, 0x61B4
LPIVBPHY_CTRL0	61B8	61B8	25016	186E	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 0, 0x61B8
LPIVBPHY_CTRL1	61BC	61BC	25020	186F	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 1, 0x61BC
LPIVBPHY_CTRL2	61C0	61C0	25024	1870	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 2, 0x61C0
LPIVBPHY_CTRL3	61C4	61C4	25028	1871	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 3, 0x61C4
LPIVBPHY_CTRL4	61C8	61C8	25032	1872	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 4, 0x61C8
LPIVBPHY_CTRL5	61CC	61CC	25036	1873	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 5, 0x61CC
LPIVBPHY_CTRL6	61D0	61D0	25040	1874	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 6, 0x61D0
LPIVBPHY_CTRL7	61D4	61D4	25044	1875	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 7, 0x61D4
LPIVBPHY_CTRL8	61D8	61D8	25048	1876	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 8, 0x61D8
LPIVBPHY_CTRL9	61DC	61DC	25052	1877	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 9, 0x61DC
LPIVBPHY_CTRL10	61E0	61E0	25056	1878	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 10, 0x61E0
LPIVBPHY_CTRL11	61E4	61E4	25060	1879	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 11, 0x61E4
LPIVBPHY_CTRL12	61E8	61E8	25064	187A	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Control Register 12, 0x61E8
AFEPLL_CFG0	61EC	61EC	25068	187B	32	1	Ring 0	Paging	No	Yes	AFE PLL Config Register 0, 0x61EC
AFEPLL_CFG1	61F0	61F0	25072	187C	32	1	Ring 0	Paging	No	Yes	AFE PLL Config Register 1, 0x61F0
LPIVBPHY_RXD0	61F4	61F4	25076	187D	32	1	Ring 0	Paging	No	Yes	LPIVBPHY RX Detect Control Register 0, 0x61F4
LPIVBPHY_RSV1	61F8	61F8	25080	187E	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Reserved Register 1, 0x61F8
LPIVBPHY_TBC	61FC	61FC	25084	187F	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Trace Bus Control Register, 0x61FC

PCS_CFG0	6200	6200	25088	1880	32	1	Ring 0	Paging	No	Yes	PCS Config Register 0, PIPE Interface Override, 0x6200
PCS_CFG1	6204	6204	25092	1881	32	1	Ring 0	Paging	No	Yes	PCS DFX Control, 0x6204
PCS_STAT0	6208	6208	25096	1882	32	1	Ring 0	Paging	No	Yes	PCS PIPE Status Register, 0x6208
PCS_STAT1	620C	620C	25100	1883	32	1	Ring 0	Paging	No	Yes	PCS RX Status Register, 0x620c
PCS_HLBCC	6210	6210	25104	1884	32	1	Ring 0	Paging	No	Yes	PCS Hardware link speed control, RW
PCS_HLSCS	6214	6214	25108	1885	32	1	Ring 0	Paging	No	Yes	PCS link speed status, 0x6214
PCS_HLWCS	6218	6218	25112	1886	32	1	Ring 0	Paging	No	Yes	Link Width Change Status Register, 0x6218
PCS_CFG2	621C	621C	25116	1887	32	1	Ring 0	Paging	No	Yes	PCS PSMI Configuration Register, 0x621c
PCS_STAT2	6220	6220	25120	1888	32	1	Ring 0	Paging	No	Yes	PCS PSMI Status register, 0x6220
PCS_SVPORT	6224	6224	25124	1889	32	1	Ring 0	Paging	No	Yes	PCS SV Port Status register, 0x6224
PCS_DEBUG	6228	6228	25128	188A	32	1	Ring 0	Paging	No	Yes	PCS DEBUG Configuration Register, 0x6228
PCS_CFG3	622C	622C	25132	188B	32	1	Ring 0	Paging	No	Yes	PCS Elastic Buffer Configuration Register, 0x622c
PCS_CFG4	6230	6230	25136	188C	32	1	Ring 0	Paging	No	Yes	PCS RX Align Configuration Register, 0x6230
PCS_CFG5	6234	6234	25140	188D	32	1	Ring 0	Paging	No	Yes	PCS RX Squelch Configuration Register, 0x6234
PCS_CFG6	6238	6238	25144	188E	32	1	Ring 0	Paging	No	Yes	PCS Gen3 RX EQ Configuration Register, 0x6238
PCS_CFG7	623C	623C	25148	188F	32	1	Ring 0	Paging	No	Yes	PCS TX/RX BERT Configuration Register, 0x623c
PCS_CFG8	6240	6240	25152	1890	32	1	Ring 0	Paging	No	Yes	PCS TX/RX BERT Pattern Register 1, 0x6240
PCS_CFG9	6244	6244	25156	1891	32	1	Ring 0	Paging	No	Yes	PCS TX/RX BERT Pattern Register 2, 0x6244
PCS_CFG10	6248	6248	25160	1892	32	1	Ring 0	Paging	No	Yes	PCS TX/RX BERT Pattern Register 3, 0x6248
PCS_CFG11	624C	624C	25164	1893	32	1	Ring 0	Paging	No	Yes	PCS TX/RX BERT Pattern Register 4, 0x624c
PCS_STAT3	6250	6250	25168	1894	32	1	Ring 0	Paging	No	Yes	PCS RX BERT Status register, 0x6250
PCS_CFG12	6254	6254	25172	1895	32	1	Ring 0	Paging	No	Yes	PCS RX-Detect Configuration Register, 0x6254
PCS_STAT4	6258	6258	25176	1896	32	1	Ring 0	Paging	No	Yes	PCS BERT Status Register, Address = 0x6258
LPIVBPHY_TBD AT0	6300	6300	25344	18C0	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Trace Bus Data Register 0, Address = 0x6300
LPIVBPHY_TBD AT1	6304	6304	25348	18C1	32	1	Ring 0	Paging	No	Yes	LPIVBPHY Trace Bus Data Register 1, Address = 0x6304

LPIVBPHY_TBD AT2	6308	6308	25352	18C2	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 2, Address = 0x6308
LPIVBPHY_TBD AT3	630C	630C	25356	18C3	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 3, Address = 0x630C
LPIVBPHY_TBD AT4	6310	6310	25360	18C4	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 4, Address = 0x6310
LPIVBPHY_TBD AT5	6314	6314	25364	18C5	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 5, Address = 0x6314
LPIVBPHY_TBD AT6	6318	6318	25368	18C6	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 6, Address = 0x6318
LPIVBPHY_TBD AT7	631C	631C	25372	18C7	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 7, Address = 0x631C
LPIVBPHY_TBD AT8	6320	6320	25376	18C8	32	1	Ring 0	Paging	No	Yes				LPIVBPHY Trace Bus Data Register 8, Address = 0x6320
COSPARE	6324	6324	25380	18C9	32	1	Ring 0	Paging	No	Yes				IVB-C0 spare confer bits, 0x6324
MSIXRAM	7000	7000	28672	1C00	32	1	Ring 0	Paging	Yes	Yes	RTL	OTHER	TRM	MSI-X RAM
sysint_debug	9000	9000	36864	2400	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Cause Read Register 0
SICRO	9004	9004	36868	2401	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Cause Read Register 0
SICSO	9008	9008	36872	2402	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Cause Set Register 0
SICE0	900C	900C	36876	2403	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Cause Enable Register 0
SICCO	9010	9010	36880	2404	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Cause Clear Register 0
SIACO	9014	9014	36884	2405	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Auto Clear Register 0
SIAEO	9018	9018	36888	2406	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Interrupt Auto Enable Register 0
SDBIC	9030	9040	36912	240C	32	5	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	System Doorbell Interrupt Command Register 0 to 4
MXAR	9044	9080	36932	2411	32	16	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	MSI-X Allocation Register 0 to 15
MSIXPBACR	9084	9084	36996	2421	32	1	Ring 0	Paging	Yes	Yes	RTL	PERST, HOT	TRM	MSI-X PBA Clear Register
DCAR_0	A000	A000	40960	2800	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_0	A004	A004	40964	2801	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_0	A008	A008	40968	2802	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_0	A00C	A00C	40972	2803	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_0	A010	A010	40976	2804	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_0	A014	A014	40980	2805	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register

DRAR_HI_0	A018	A018	40984	2806	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_0	A01C	A01C	40988	2807	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_0	A020	A020	40992	2808	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_0	A024	A024	40996	2809	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_0	A028	A028	41000	280A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_0	A02C	A02C	41004	280B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCHERRMSK_0	A030	A030	41008	280C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCAR_1	A040	A040	41024	2810	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_1	A044	A044	41028	2811	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_1	A048	A048	41032	2812	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_1	A04C	A04C	41036	2813	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_1	A050	A050	41040	2814	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_1	A054	A054	41044	2815	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_1	A058	A058	41048	2816	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_1	A05C	A05C	41052	2817	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_1	A060	A060	41056	2818	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_1	A064	A064	41060	2819	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_1	A068	A068	41064	281A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_1	A06C	A06C	41068	281B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCHERRMSK_1	A070	A070	41072	281C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCAR_2	A080	A080	41088	2820	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_2	A084	A084	41092	2821	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_2	A088	A088	41096	2822	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_2	A08C	A08C	41100	2823	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_2	A090	A090	41104	2824	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_2	A094	A094	41108	2825	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_2	A098	A098	41112	2826	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_2	A09C	A09C	41116	2827	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_2	A0A0	A0A0	41120	2828	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register

DTPWBR_LO_2	A0A4	A0A4	41124	2829	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_2	A0A8	A0A8	41128	282A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_2	A0AC	A0AC	41132	282B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCHERRMSK_2	A0B0	A0B0	41136	282C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCAR_3	A0C0	A0C0	41152	2830	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_3	A0C4	A0C4	41156	2831	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_3	A0C8	A0C8	41160	2832	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_3	A0CC	A0CC	41164	2833	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0 RPB
DAUX_HI_3	A0D0	A0D0	41168	2834	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0 RPB
DRAR_LO_3	A0D4	A0D4	41172	2835	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_3	A0D8	A0D8	41176	2836	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_3	A0DC	A0DC	41180	2837	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_3	A0E0	A0E0	41184	2838	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_3	A0E4	A0E4	41188	2839	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_3	A0E8	A0E8	41192	283A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_3	A0EC	A0EC	41196	283B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCHERRMSK_3	A0F0	A0F0	41200	283C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCAR_4	A100	A100	41216	2840	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_4	A104	A104	41220	2841	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_4	A108	A108	41224	2842	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_4	A10C	A10C	41228	2843	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0 RPB
DAUX_HI_4	A110	A110	41232	2844	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0 RPB
DRAR_LO_4	A114	A114	41236	2845	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_4	A118	A118	41240	2846	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_4	A11C	A11C	41244	2847	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_4	A120	A120	41248	2848	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_4	A124	A124	41252	2849	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_4	A128	A128	41256	284A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_4	A12C	A12C	41260	284B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register

DCHERRMSK_4	A130	A130	41264	284C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY_B	CRU	DMA Channel Error Register
DCAR_5	A140	A140	41280	2850	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_5	A144	A144	41284	2851	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_5	A148	A148	41288	2852	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_5	A14C	A14C	41292	2853	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_5	A150	A150	41296	2854	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_5	A154	A154	41300	2855	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_5	A158	A158	41304	2856	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_5	A15C	A15C	41308	2857	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_5	A160	A160	41312	2858	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_5	A164	A164	41316	2859	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_5	A168	A168	41320	285A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_5	A16C	A16C	41324	285B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY_B	CRU	DMA Channel Error Register
DCHERRMSK_5	A170	A170	41328	285C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY_B	CRU	DMA Channel Error Register
DCAR_6	A180	A180	41344	2860	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_6	A184	A184	41348	2861	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register
DTPR_6	A188	A188	41352	2862	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_6	A18C	A18C	41356	2863	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_6	A190	A190	41360	2864	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_6	A194	A194	41364	2865	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_6	A198	A198	41368	2866	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_6	A19C	A19C	41372	2867	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_6	A1A0	A1A0	41376	2868	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_6	A1A4	A1A4	41380	2869	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_6	A1A8	A1A8	41384	286A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_6	A1AC	A1AC	41388	286B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY_B	CRU	DMA Channel Error Register
DCHERRMSK_6	A1B0	A1B0	41392	286C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY_B	CRU	DMA Channel Error Register
DCAR_7	A1C0	A1C0	41408	2870	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Attributes Register
DHPR_7	A1C4	A1C4	41412	2871	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Head Pointer Register

DTPR_7	A1C8	A1C8	41416	2872	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Tail Pointer Register
DAUX_LO_7	A1CC	A1CC	41420	2873	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DAUX_HI_7	A1D0	A1D0	41424	2874	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Auxiliary Register 0
DRAR_LO_7	A1D4	A1D4	41428	2875	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DRAR_HI_7	A1D8	A1D8	41432	2876	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Descriptor Ring Attributes Register
DITR_7	A1DC	A1DC	41436	2877	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Interrupt Timer Register
DMA_DSTAT_7	A1E0	A1E0	41440	2878	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Channel Status Register
DTPWBR_LO_7	A1E4	A1E4	41444	2879	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register Lo
DTPWBR_HI_7	A1E8	A1E8	41448	287A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	DMA Tail Pointer Write Back Register
DCHERR_7	A1EC	A1EC	41452	287B	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DCHERRMSK_7	A1F0	A1F0	41456	287C	32	1	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	DMA Channel Error Register
DMA_REQUEST_SIZE	A200	A200	41472	2880	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	DMA Request control
DCR	A280	A280	41600	28A0	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	DMA Configuration Register
DQAR	A284	A284	41604	28A1	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Descriptor Queue Access Register
DQDR_TL	A288	A288	41608	28A2	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Descriptor Queue Data Register Top Left
DQDR_TR	A28C	A28C	41612	28A3	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Descriptor Queue Data Register Top Right
DQDR_BL	A290	A290	41616	28A4	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Descriptor Queue Data Register Bottom Left
DQDR_BR	A294	A294	41620	28A5	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Descriptor Queue Data Register Bottom Right
DMA_SPARE0	A298	A298	41624	28A6	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Spare DMA register full 32-bits available
TRANSLIMIT	A29C	A29C	41628	28A7	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	TPE Attributes Register
DMA_SPARE2	A2A0	A2A0	41632	28A8	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Spare DMA register full 32-bits available
DMA_MISC	A2A4	A2A4	41636	28A9	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Misc bits such as chicken bits -- etc...
CP_MEMREGIO N_BASE	A2B0	A2B0	41648	28AC	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Base Address: Address at 4K granularity
CP_MEMREGIO N_TOP	A2B4	A2B4	41652	28AD	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Memory Region Length (4K)
PSMIA_0	A3A0	A3A0	41888	28E8	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_1	A3A4	A3A4	41892	28E9	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_2	A3A8	A3A8	41896	28EA	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_3	A3AC	A3AC	41900	28EB	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_4	A3B0	A3B0	41904	28EC	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register

PSMIA_5	A3B4	A3B4	41908	28ED	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_6	A3B8	A3B8	41912	28EE	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIA_7	A3BC	A3BC	41916	28EF	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_0	A3C0	A3C0	41920	28F0	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_1	A3C4	A3C4	41924	28F1	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_2	A3C8	A3C8	41928	28F2	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_3	A3CC	A3CC	41932	28F3	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_4	A3D0	A3D0	41936	28F4	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_5	A3D4	A3D4	41940	28F5	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_6	A3D8	A3D8	41944	28F6	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIB_7	A3DC	A3DC	41948	28F7	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_0	A3E0	A3E0	41952	28F8	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_1	A3E4	A3E4	41956	28F9	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_2	A3E8	A3E8	41960	28FA	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_3	A3EC	A3EC	41964	28FB	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_4	A3F0	A3F0	41968	28FC	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_5	A3F4	A3F4	41972	28FD	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_6	A3F8	A3F8	41976	28FE	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
PSMIC_7	A3FC	A3FC	41980	28FF	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI register
DMA_LOCK	A400	A400	41984	2900	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Master Lock register
APICIDR	A800	A800	43008	2A00	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	APIC Identification Register
APICVER	A804	A804	43012	2A01	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	APIC Version Register
APICAPR	A808	A808	43016	2A02	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	APIC Priority Register
APICRT	A840	A908	43072	2A10	64	26	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	APIC Redirection Table
APICICR	A9D0	AA08	43472	2A74	64	8	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	APIC Interrupt Command Register 0 to 7
MCA_INT_STAT	AB00	AB00	43776	2AC0	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	MCA Interrupt Status Register
MCA_INT_EN	AB04	AB04	43780	2AC1	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	MCA Interrupt Enable Register
SCRATCH	AB20	AB5C	43808	2AC8	32	16	Ring 0	Paging	Yes	Yes	RTL	STICKY _B	CRU	Scratch Registers for Software
PSMI_MEMSHA DOW_CNTRL	AC00	AC00	44032	2B00	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI shadow memory size bits
PSMI_EN	AC04	AC04	44036	2B01	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI Enable bits
PSMI_TIM_CNT RL	AC08	AC08	44040	2B02	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	PSMI Time Control
CONCAT_CORE _HALTED	AC0C	AC0C	44044	2B03	64	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Concatenated core halted status

FORCE_BPM	AC48	AC48	44104	2B12	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Forces the BPM output pins for instrumentation purposes
CORE_HALTED	AC4C	AD40	44108	2B13	32	62	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Core of same number writes a 1 just before halting
AGF_CONTROL	AFEC	AFEC	45036	2BFB	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	This register contains the AGF control fields
AGF_PERIOD_N	AFF0	AFF0	45040	2BFC	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	This register contains the 8 possible AGF period settings used for dynamic frequency changes. See also AGF_MASTER_MCLK_RANGE, AGF_MASTER_DELAY_OUT, AGF_MASTER_DELAY_IN and AGF_MCLK_RANGE
AGF_MASTER_DELAY_IN	AFF4	AFF4	45044	2BFD	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	This register contains the 8 possible AGF master input delay settings used for dynamic frequency changes. See also AGF_MASTER_MCLK_RANGE, AGF_MASTER_DELAY_OUT, AGF_MASTER_DELAY_IN and AGF_MCLK_RANGE
AGF_MASTER_DELAY_OUT	AFF8	AFF8	45048	2BFE	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	This register contains the 8 possible AGF master output delay settings used for dynamic frequency changes. See also AGF_MASTER_MCLK_RANGE, AGF_MASTER_DELAY_OUT, AGF_MASTER_DELAY_IN and AGF_MCLK_RANGE
AGF_MASTER_MCLK_RANGE	AFFC	AFFC	45052	2BFF	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	This register contains the 8 range selects which are used to determine the parameters used by the AGF. The MCLK frequency is communicated on 6 bits. The ranges are cumulative. For example the range for group 2 is RANGE_0+RANGE_1 to RANGE_0+RANGE_1+RANGE_2. See also AGF_MASTER_MCLK_RANGE, AGF_MASTER_DELAY_OUT, AGF_MASTER_DELAY_IN and AGF_MCLK_RANGE
RDMSR0	B180	B180	45440	2C60	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMSR1	B184	B184	45444	2C61	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMSR2	B188	B188	45448	2C62	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMSR3	B18C	B18C	45452	2C63	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register

RDMASR4	B190	B190	45456	2C64	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMASR5	B194	B194	45460	2C65	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMASR6	B198	B198	45464	2C66	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
RDMASR7	B19C	B19C	45468	2C67	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Remote DMA register
C6_SCRATCH	C000	C054	49152	3000	32	22	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Scratch Pad registers for package-C6
APR_PHY_BASE	C11C	C11C	49436	3047	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	
RS_CR_ADAK_CTL	CC00	CC00	52224	3300	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Ring Stop ADAK Control Register
RS_CR_BL_CTL	CC04	CC04	52228	3301	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Ring Stop BL Control Register
SBQ_LOCK	CC0C	CC0C	52236	3303	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Master Lock register
BSPID	CC10	CC10	52240	3304	32	1	Ring 0	Paging	No	Yes	OTHE R	CSR_G RPB	CRU	BSP ID Register
UNSTALL_DELAY	CC14	CC14	52244	3305	32	1	Ring 0	Paging	No	Yes	RTL	CSR_G RPB	CRU	Number of clocks of delay to be inserted after all fuse packets sent
SBOX_RS_EMON_Selectors	CC20	CC20	52256	3308	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	SBOX RS EMON selectors
SBOX_EMON_CNT_OVFL	CC24	CC24	52260	3309	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	This indicates if there's any overflow in any EMON counter
EMON_CNT0	CC28	CC28	52264	330A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	EMON counter 0
EMON_CNT1	CC2C	CC2C	52268	330B	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	EMON counter 1
EMON_CNT2	CC30	CC30	52272	330C	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	EMON counter 2
EMON_CNT3	CC34	CC34	52276	330D	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	EMON counter 3
SBQ_MISC	CC38	CC38	52280	330E	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Misc register with sbq chicken bits, etc.
SPARE1	CC3C	CC3C	52284	330F	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Spare register full 32-bits available
SPARE2	CC40	CC40	52288	3310	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Spare register full 32-bits available
SPARE3	CC44	CC44	52292	3311	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Spare register full 32-bits available
DBOX_BW_RESERVATION	CC50	CC50	52304	3314	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	8-bits DBOX reservation slot value from SW
RSC0	CC54	CC54	52308	3315	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 0
RSC1	CC58	CC58	52312	3316	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 1
RSC2	CC5C	CC5C	52316	3317	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 2
RSC3	CC60	CC60	52320	3318	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 3
RSC4	CC64	CC64	52324	3319	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 4
RSC5	CC68	CC68	52328	331A	32	1	Ring 0	Paging	No	Yes	FUSE, FLASH	CSR_G RPB	CRU	Ring Stop Agent Configuration Register 5

RESET_FSM	CC78	CC78	52344	331E	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	Reset FSM's status Register
EMON_TCU_CONTROL	CC84	CC84	52356	3321	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	TCU EMON Control Register
Doorbell_INT	CC90	CC9C	52368	3324	32	4	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	System Doorbell Interrupt Command Register 0 to 3
MarkerMessage_Disable	CCA0	CCA0	52384	3328	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	32-bits to disable interrupts
MarkerMessage_Assert	CCA4	CCA4	52388	3329	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	32-bits to assert interrupts
MarkerMessage_Send	CCA8	CCA8	52392	332A	32	1	Ring 0	Paging	Yes	Yes	RTL	CSR_G RPB	CRU	32-bits to log INTSCR field of Marker Message
PDAT	CCE0	CCE0	52448	3338	32	1	Ring 0	Paging	No	Lockable	RTL	CSR_G RPB	CRU	Primary DAT register for LDAT logic
SDAT	CCE8	CCE8	52456	333A	32	1	Ring 0	Paging	No	Lockable	RTL	CSR_G RPB	CRU	Secondary DAT register
DATOUT	CCF0	CCF0	52464	333C	32	1	Ring 0	Paging	No	Lockable	RTL	CSR_G RPB	CRU	Spare register full 32-bits available
DATINO	CCF8	CCF8	52472	333E	32	1	Ring 0	Paging	No	Lockable	RTL	CSR_G RPB	CRU	DATIN register