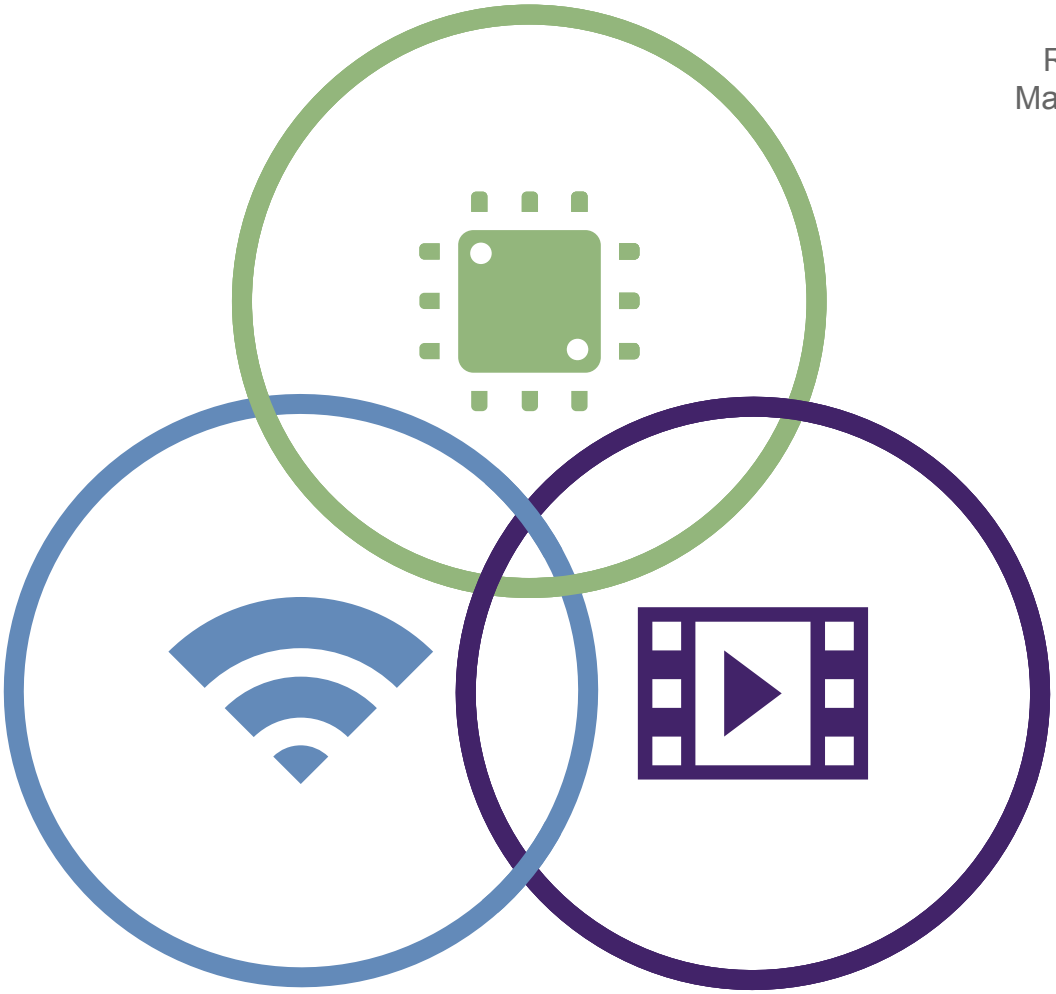
MIPS64® I6500 Multiprocessing System Programmer's Guide

Revision 1.00
March 29, 2017
Public



Document Number: MD01179

Chapter 1

Architecture Overview

This document describes the software-programmable aspects of the 64-bit MIPS I6500 Multiprocessing System (MPS). The device consists of the logic blocks shown in Figure 1.1. The majority of blocks in the diagram have at least one dedicated chapter that describes how to control the hardware using registers and assembly code. The register-programming examples describe a programming sequence of how to set or change a programmable parameter using registers. The assembly code examples show how the MIPS instruction set can be used to perform the same function.

Each chapter provides the relevant background information required by the programmer in order to understand the examples. Common examples such as enabling and initialization are provided for each block, as well as more in depth examples relative to that block.

An overview of the material provided in this document is as follows:

- *Memory Management (MMU)*: This chapter describes the programmable elements of the Translation Lookaside Buffer or TLB of the I6500 Multiprocessing System. The first section gives an overview of the TLB architecture, a description of its functionality and a description of the elements that go into programming the TLB. The sections that follow cover specific information on programming for the Translation Lookaside Buffer (TLB).
- *Caches*: This chapter provides an overview of the cache architecture, a description of its functionality, and a description of the elements that go into programing the caches. A description of the CP0 register interface to each cache is provided, as well as initialization code for all three caches, setting up cache coherency, handling cache exceptions, and testing the cache RAM.
- *Exceptions*: This chapter describes an overview of exception processing and a definition of the interrupt modes. Information on how to program the reset, boot, and general exception vectors in memory is also covered. A list of exception priorities is provided, along with an assembly language example of an exception handler.
- *Coherence Manager* (CM): The I6500 MPS contains a third generation Coherence Manager. This chapter provides an overview of the CM register ring bus and associated table that lists each device ID on the bus. The programmer uses this information to access these devices. An overview of the CM register address space is also provided. In addition, the chapter describes how to program the CM to perform various functions, including setting the base addresses in memory, accessing another VP in the same core, accessing a VP in another core, accessing the Global Interrupt Controller (GIC), Cluster Power Controller (CPC), and/or Debug Unit (DBU) registers via the CM, and setting the clock ratios between the various I6500 system components. For the exact revision number of the Coherence Manager, refer to the Release Notes.

  This chapter also introduces the multi-cluster configuration that allows multiple I6500 Multiprocessing Systems to be connected through a Network-On-Chip (NOC) interface. The section describes the registers used to perform a cluster-to-cluster access.
- *Cluster Power Controller (CPC)*: This chapter provides an overview of how power is managed in the I6500 Multiprocessing System and identifies the various power and clock domains the programmer can use to manage power consumption in the device. In addition, a procedure on how to set the CPC base address in memory is provided. Other programming principles include setting the device to coherent or non-coherent mode, requestor (core or IOCU) access of CPC registers, system power-up policy, programming examples of a clock domain change and clock delay change, powering up the CPC in standalone mode (no cores enabled), reset detection, VP

run/suspend mechanism, local RAM shutdown and wake-up procedure, accessing registers in another power domain, and fine tuning internal and external signal delays to help the programmer easily integrate the device into a system environment.

- *Global Interrupt Controller (GIC)*: This chapter describes how to program the various elements of the GIC using both register examples and code examples. Some of these elements include setting the operating mode, setting up the address map, GIC register layout and distribution, setting the GIC base address, determining the number of external interrupts, and configuring individual interrupt sources.
- *Floating Point Unit (FPU)*: This chapter provides information on how to enable the FPU, how to handle floating point exceptions, how to set the rounding mode, and operation of the Flush-to-Zero (FS) function.
- *MIPS SIMD Architecture (MSA)*: This chapter describes the MIPS Single-Instruction-Multiple-Data (SIMD) architecture. It provides information on how to enable MSA, how to map scalar floating point registers to MSA vector registers, and MSA exception handling.
- *Virtualization (VZ)*: The Virtualization Module defines a set of new instructions, registers, and machine states to the I6500 core to manage the efficient implementation of virtualized systems. The Virtualization Module is designed to enable full virtualization of operating systems and allows for the execution of Guest Operating Systems in a fully virtualized environment.
- *DSPRAM*: The optional Data Scratch Pad RAM (DSPRAM) block provides a general scratch pad RAM used for temporary storage of data. The DSPRAM provides a connection to on-chip memory or memory-mapped registers, which are accessed in parallel with the L1 data cache to minimize access latency.
- *Inter-Thread Communication Unit (ITU)*: The ITU provides an alternative to Linked-Load/Store-Conditional synchronization for fine grained multithreading by utilizing gating storage. The chapter describes the purpose for the ITU and the configuration and programming aspects.
- *Multi-threading*: This chapter provides an overview of the hardware multi-threading mechanism in the I6500 MPS.
- *On-Chip Instrumentation (OCI)*: This chapter provides a brief overview of the interface and external debugging environment required to debug MIPS processors that incorporate the MIPS On-Chip Instrumentation (OCI) debug system for multi-core designs.

Throughout all of the aforementioned chapters, there are assembly language examples that describe how various programming elements are handled in software. These examples can be used by programmer's writing their own code to program a particular block, or for writing a low-level support library, RTOS, or their own tool chain. However, most of the code examples described are part of the MIPS Codescape tool chain. As such, it is not necessary for the programmer to manually execute these code examples when using Codescape as this functionality is already built in to the Codescape software.

This document is meant to be used with two other companion documents:

- *64-bit MIPS I6500 Technical Reference Manual* contained in the document suite. This document contains supplemental information to the *I6500 Programmers Guide*.
- *64-bit MIPS I6500 Multiprocessing System Integrator's Guide* (MD01041), This companion document provides hardware details about the device, including functional verification, system integration, and system implementation.

1.1 Product Overview

The I6500 series is a high performance multi-core microprocessor system that provides a best in class power efficiency for use in system-on-chip (SoC) applications. The I6500 Coherence Manager maintains Level 2 (L2) cache and system level coherency between all cores, main memory, and I/O devices. The I6500 Multiprocessing System (MPS) can be configured with a variable number of cores, I/O coherent interfaces, and L2 cache size.

Each I6500 core implements the Release 6 of the MIPS64 Instruction Set Architecture with full hardware multithreading and hardware virtualization support. In addition, the core can be configured with a SIMD engine supporting integer, single and double precision, and floating and fixed point operations.

The I6500 MPS supports both single-cluster and multi-cluster configurations as described in the following subsections.

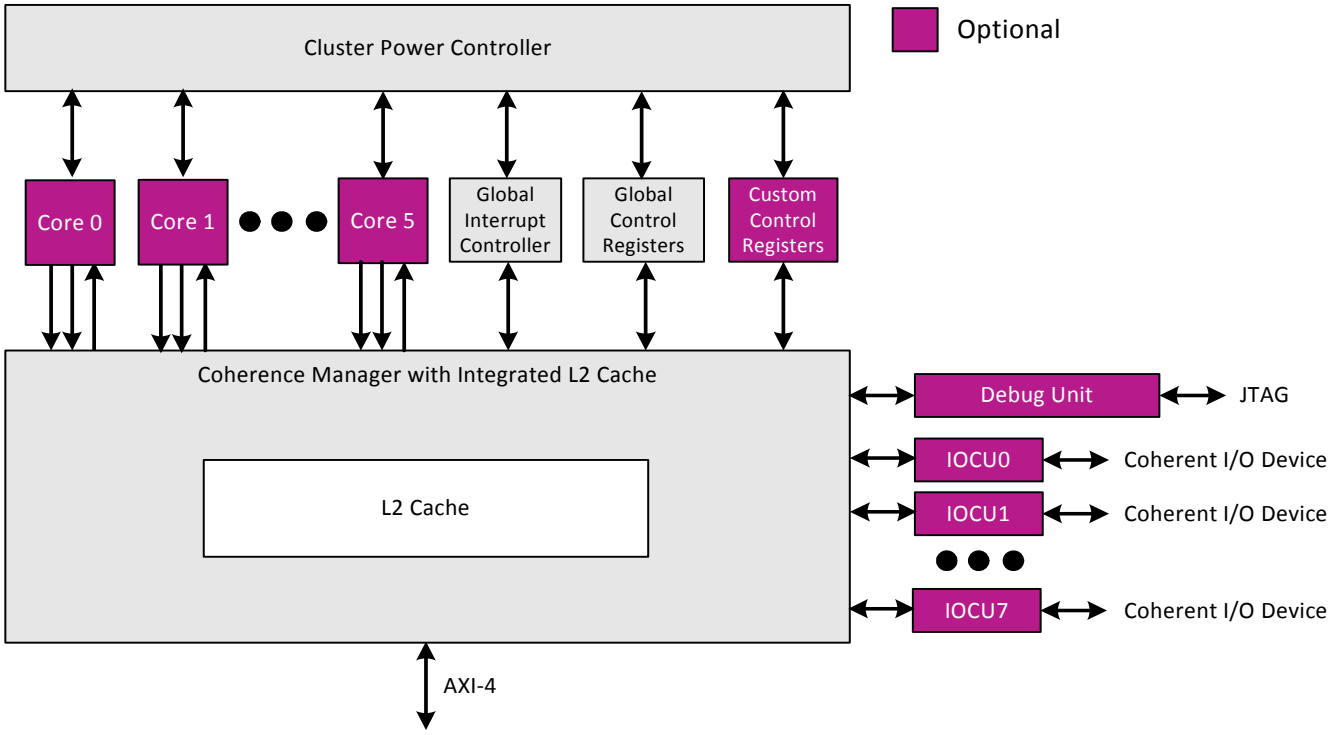
1.1.1 Single-Cluster Configuration

Figure 1.1 shows a block diagram of a single-cluster I6500 Multiprocessing System. The I6500 MPS contains the following logic blocks:

- Up to six cores
- Coherence Manager (CM) with integrated L2-cache
- Up to eight I/O Coherence Units (IOCU)
- Cluster Power Controller (CPC)
- Global Interrupt Controller (GIC)
- Global Configuration Registers (GCR)
- Multiprocessor debug via in-system Debug Unit (DBU)

In the I6500 MPS the total number of cores and IOCUs together must be less than or equal to eight. All cores and IOCU's are optional and can be configured in any combination of up to eight.

Figure 1.1 System-level Block Diagram of Single-Cluster I6500 Multiprocessing System



For more information on the *Cluster Power Controller (CPC)* block, refer to the *Cluster Power Controller* chapter of this manual.

For more information on the *Global Interrupt Controller (GIC)* block, refer to the *Global Interrupt Controller* chapter of this manual.

For more information on the *Coherence Manager (CM)*, refer to the *Coherence Manager* chapter of this manual.

For more information on the *L2 Cache Memory*, refer to the *Caches* chapter of this manual.

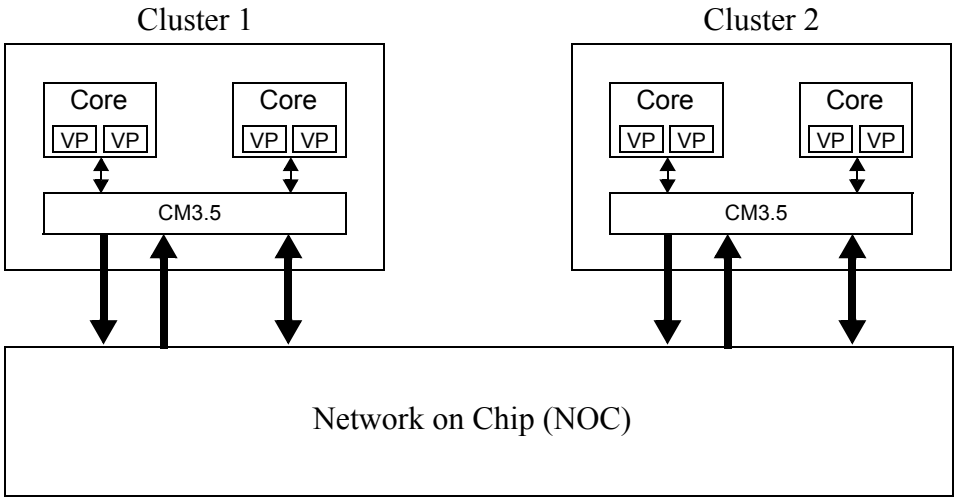
For more information on the *Global Configuration Registers* block, refer to the companion *I6500 Registers* document that is included in the documentation package. Selected registers from this block are used throughout the *Coherence Manager* chapter of this manual to provide various CM register programming examples.

For more information on the programmable blocks within the *Core*, such as *MMU*, *FPU*, *MSA*, etc. refer to the Figure 1.3.

1.1.2 Multi-Cluster Configuration

In addition to the single-cluster configuration shown above, the I6500 also allows for cluster-to-cluster accesses. This allows a core or VP in one cluster to access a core or VP in another cluster through the Network-On-Chip (NOC) interface. This interface is shown in Figure 1.2.

Figure 1.2 Cluster-to-Cluster Accesses Using the NOC



For example, a VP within a core in Cluster 1 can access and update a register in a VP in Cluster 2 as shown. The access is processed by the CM3.5 and driven onto the NOC. The NOC then routes the request to the appropriate cluster where the access is scheduled by the CM3.5 in the destination cluster. The data is fetched and returned to the requesting VP through the NOC.

For more information, refer to Chapter 4, Coherency Manager.

1.2 I6500 Features

The I6500 MPS contains the following architecture, system, and core-level main features as described in the following subsections.

1.2.1 MIPS64® Release 6 Architecture

The MIPS64 Release 6 architecture is based on a fixed-length, regularly encoded instruction set, and it uses a load/store data model. It is streamlined to support optimized execution of high-level languages. Arithmetic and logic operations use a three-operand format, allowing compilers to optimize complex expressions formulation. Availability of 32 general-purpose registers enables compilers to further optimize code generation by keeping frequently accessed data in registers.

1.2.2 MIPS® SIMD Architecture

Single Instruction Multiple Data (SIMD) instructions improve performance by allowing efficient parallel processing of vector operations. The MIPS® SIMD Architecture (MSA) technology incorporates a software-programmable solution to handle those functions not covered by dedicated hardware. This programmable solution allows for increased system flexibility. In addition, the MSA is designed to accelerate many compute-intensive applications by enabling generic compiler support.

The MIPS gcc compiler has been tuned to understand the "vector" type so that your C/C++ code can make use of SIMD vector features.

1.2.3 MIPS® Virtualization

The hardware virtualization support addresses security, privacy and reliability concerns for a wide range of devices. Virtualization can be achieved with software only, or with hardware assistance (fully virtualized). The core element of virtualization is the Hypervisor, a small body of trusted and privileged code that sits above the hardware, managing and orchestrating all of the SoC resources. It manages the resources by defining access policies for each execution environment or "guest." Guests are isolated from each other, but can communicate with the Hypervisor and with each other via secure APIs. This ensures the reliability of the system by allowing the rest of the guests to operate reliably even if one of the guests fails or otherwise becomes corrupted. The hypervisor manages all memory I/O privileges.

Contact MIPS Customer Support through our Partner Portal about recommendations on which Hypervisors are available for use.

1.2.4 System-level Features

- Up to six coherent MIPS64 R6 cores
- Integrated, L2 cache controller supporting 8-way and 16-way set-associativity
    - Inclusive of the L1 data caches
    - 256 KB to 8 MB cache sizes
    - Error correction and detection
- High-speed L2 cache initialization
- CPC to shut down idle cores for power efficiency
- Up to eight IOCUs
- Virtualization module support
- Cache-to-cache data transfers
- Out-of-order data return
- Provides AXI-4 and ACE interfaces for connection to an external Network-On-Chip (NOC), which allows two I6500 clusters to be connected together.
- Hardware L2 cache prefetch controller significantly improves performance of workloads such as **`memcopy`**
- Independent clock ratios on core, memory, and IOCU ports
- SoC system interface supports the AXI-4 bus protocol with 48-bit address and 256-bit data paths
- SoC system interface supports the ACE bus protocol in the multi-cluster configuration
- Supports up to four auxiliary (AUX) AXI-4 ports per cluster.
- High bandwidth 128-bit data paths between each core and the Coherence Manager
- Software controlled core level and cluster level power management
- Debug port supporting multi-core debug (JTAG/APB)

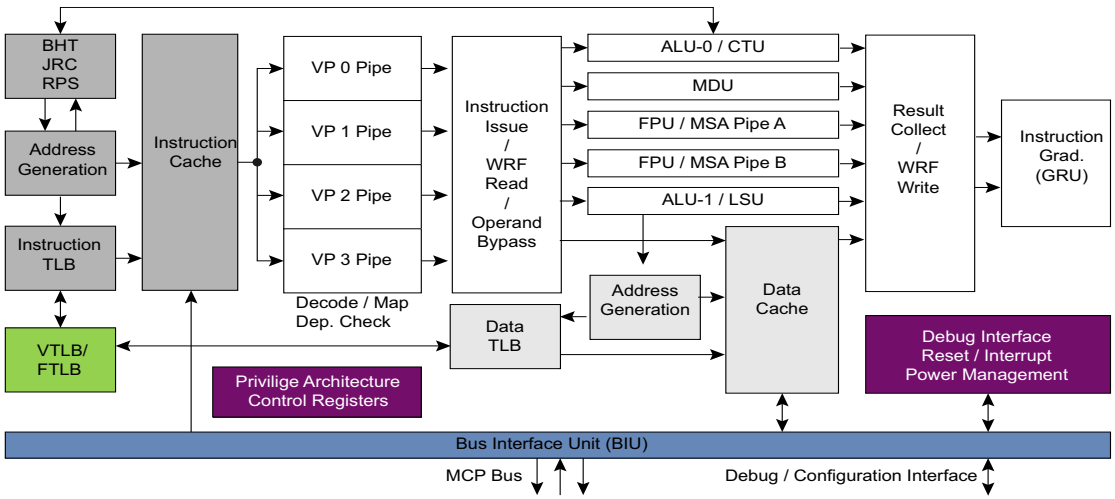
1.2.5 Core-level Features

- Full 64-bit MIPS64 Release 6 Instruction Set Architecture
- 48-bit virtual and physical addresses
- Power efficient design
- Dual issue instruction fetch, decode, issue, and graduate
- Hardware multithreading through seamless Virtual Processor (VP) support
- Virtualization support
- L1 caches with Error Correction Code (ECC) protection
- Memory Management Unit with first-level ITLB/DTLB backed by fast programmable on-core second-level variable page size TLB (VTLB) and fixed page size TLB (FTLB)
- Load and store bonding support
- Unaligned load / store support in hardware
- Uncached Accelerated support
- Optional Inter-thread Communication Unit (ITU)
- Support for uncached and paired Load-Linked and Store Conditional (LL/SC) operations
- Optional Data Scratch Pad (DSP) RAM block for temporary storage

1.3 I6500 Core Block Diagram

Figure 1.2 shows a block diagram of a single I6500 core.

Figure 1.3 I6500 Core-level Block Diagram



For more information on the *Instruction TLB*, *Data TLB*, and *VTLB/FTLB* blocks shown in Figure 1.3, refer to the *Memory Management* chapter of this manual.

For more information on the *L1 Instruction Cache* and *L1 Data Cache* blocks, refer to the *Caches* chapter of this manual.

For more information on the *FPU* block, refer to the *FPU* chapter of this manual.

For more information on the *MSA* block, refer to the *MSA* chapter of this manual.

1.4 CP0 Register to Assembler Mapping

Throughout this document, registers are referred to as C0_<REGISTER>. In order for the assembler to understand these names, they must be mapped to their numerical version that the assembler will understand. The numerical version uses a $x, y reference. Table 1.1 maps the root register names to their respective numerical versions.

Table 1.1 Assembler Mapping of CP0 Registers

Assembler Idiom	Register Name	Assembler Idiom	Register Name	Assembler Idiom	Register Name
$0, 0	C0_INDEX	$12, 6	C0_GUESTCTL0	$23, 0	C0_DEBUG
$0, 4	C0_VPCONTROL	$12, 7	C0_GTOFFSET	$24, 0	C0_DEPC
$2, 0	C0_ENTRYLO0	$13, 0	C0_CAUSE	$25, 0	C0_PERFCTL0
$3, 0	C0_ENTRYLO1	$14, 0	C0_EPC	$25, 1	C0_PERFCNT0

Table 1.1 Assembler Mapping of CP0 Registers *(continued)*

Assembler Idiom	Register Name	Assembler Idiom	Register Name	Assembler Idiom	Registe
$3, 1	C0_GLOBALNUM	$15, 0	C0_PRID	$25, 2	C0_PERFCTL1
$4, 0	C0_CONTEXT	$15, 1	C0_EBASE	$25, 3	C0_PERFCNT1
$4, 2	C0_USERLOCAL	$15, 2	C0_CDMMBASE	$25, 4	C0_PERFCTL2
$4, 4	C0_DBGCONTEXTID	$15, 3	C0_CMGCRBASE	$25, 5	C0_PERFCNT2
$4,5	C0_MMID	$16, 0	C0_CONFIG	$25, 6	C0_PERFCTL3
$5, 0	C0_PAGEMASK	$16, 1	C0_CONFIG1	$25, 7	C0_PERFCNT3
$5, 1	C0_PAGEGRAIN	$16, 2	C0_CONFIG2	$26, 0	C0_ERRCTL
$6, 0	C0_WIRED	$16, 3	C0_CONFIG3	$27, 0	C0_CACHERR
$7, 0	C0_HWRENA	$16, 4	C0_CONFIG4	$28, 0	C0_ITAGLO
$8, 0	C0_BADVADDR	$16, 5	C0_CONFIG5	$28, 1	C0_IDATALO
$8, 1	C0_BADINSTR	$16, 7	C0_CONFIG7	$28, 2	C0_DTAGLO
$8, 2	C0_BADINSTRP	$17, 0	C0_LLADDR	$28, 3	C0_DDATALO
$9, 0	C0_COUNT	$17, 1	C0_MAAR	$29, 1	C0_IDATAHI
$9,6	C0_SAARI	$17, 2	C0_MAARI	$29, 3	C0_DDATAHI
$9,7	C0_SAAR	$18, 0	C0_WATCHLO0	$30, 0	C0_ERROREPC
$10, 0	C0_ENTRYHI	$18, 1	C0_WATCHLO1	$31, 0	C0_DESAVE
$10, 4	C0_GUESTCTL1	$18, 2	C0_WATCHLO2	$31, 2	C0_KSCRATCH1
$10, 5	C0_GUESTCTL2	$18, 3	C0_WATCHLO3	$31, 3	C0_KSCRATCH2
$11, 0	C0_COMPARE	$19, 0	C0_WATCHHI0	$31, 4	C0_KSCRATCH3
$11, 4	C0_GUESTCTL0EXT	$19, 1	C0_WATCHHI1	$31, 5	C0_KSCRATCH4
$12, 0	C0_STATUS	$19, 2	C0_WATCHHI2	$31, 6	C0_KSCRATCH5
$12, 1	C0_INTCTL	$19, 3	C0_WATCHHI3	$31, 7	C0_KSCRATCH6
$12, 2	C0_SRSCTL	$20, 0	C0_XCONTEXT		

Note that the above table indicates those CP0 registers available in Root mode. If the device is in Guest mode, only a subset of these registers are available.

1.5 MIPS Software Tools

MIPS offers a complete portfolio of tools that address all stages of product development, including MIPS Linux, MIPS Android, Codescape SDK, Codescape debugger, compilers, MIPS boot loader, and MIPS RTOS and IoT support. Some of the tools provided are described in the following subsections.

1.5.1 MIPS Linux

MIPS actively supports, develops and improves the Linux kernel for the MIPS® architecture. Linux kernel and distributions that currently support the MIPS architecture include Debian, OpenWRT, Buildroot, Yocto, and GENTOO.

For more information on the MIPS Linux, https://www.mips.com/develop/tools/

1.5.2 MIPS Android

The MIPS emulator can be built from the Android Open Source Project releases by Google. For every Android release a QA is performed and bugs found in the process are fixed. These bug-fixes, along with any MIPS optimizations, go on top of the Android release branch and get released by MIPS. Therefore, it is recommended to download the latest MIPS releases to get the most stable version of Android sources for MIPS. Instructions are on the web site.

For more information on the MIPS Android, https://www.mips.com/develop/tools/

1.5.3 Codescape MIPS SDK

The Codescape MIPS software development toolkit provides a complete suite of compile, debug, and profile tools and libraries for developing and debugging software for MIPS processors. Codescape MIPS SDK include cross compiler (bare-metal and Linux), iaSIM simulator, optimized libraries, and profiling tools.

With Codescape, MIPS brings together a wealth of expertise in the form of Windows and Linux software development tools, hardware support packages and hardware development platforms, linked by a common development environment.

For more information on the MIPS Codescape software, https://www.mips.com/develop/tools/

1.5.4 Codescape Debugger

MIPS debug environment for heterogeneous SoC development. Fully supporting all MIPS architectural features, the Codescape Debugger enables developers to make the most of software running on MIPS cores.

For more information on the MIPS Debugger, https://www.mips.com/develop/tools/

1.5.5 Compilers

MIPS ports and maintains the GNU Compiler Collection (GCC) and provides prebuilt tool chains in the Codescape MIPS SDK. A wide range of other industry leading compilers are also available for MIPS processors.

For more information on the MIPS Compilers, https://www.mips.com/develop/tools/

1.5.6 Boot Loader

MIPS offers a wide range of solutions for initializing MIPS cores and facilitating debugging. These include open-source and proprietary solutions to suit any requirement.

For more information on the MIPS Boot Loader, https://www.mips.com/develop/tools/

1.5.7 MIPS RTOS and IoT Support

MIPS collaborates with open-source and commercial partners to provide MIPS support for many of the popular Real Time Operating Systems (RTOS) and the new generation of IoT specific Operating Systems. In addition, MIPS has developed the MIPS Embedded Operating System (MEOS) with Virtualization extensions that target deeply embedded applications and the IoT space.

For more information on the MIPS RTOS and IoT support,

https://www.mips.com/develop/tools

1.5.8 Developer Resources

MIPS offers a variety of development boards and associated resources to aid in the kernel or user software development process. Resources include development platforms, documentation, and on-line video training courses.

For more information on the MIPS Developer Resources,

https://www.mips.com/develop/tools/

Chapter 2

Memory Management Unit

The MMU translates virtual addresses generated by the core, to physical addresses used to access caches, memory and other devices. Virtual-to-physical address translation is especially useful for operating systems that must manage physical memory to accommodate multiple tasks active in the same virtual address space. The MMU also enforces the protection of memory areas and defines the cache attributes. The I6500 MMU implements a Translation Lookaside Buffer (TLB).

This chapter covers the programmable elements of the TLB in the I6500 Multiprocessing System. The first section gives an overview of the TLB architecture, a description of its functionality and a description of the elements that go into programming the TLB. The sections that follow cover specific information on programming for the TLB.

2.1 Overview

The I6500 TLB translates 48-bit virtual addresses to 48-bit physical addresses and provides access control for different page segments of memory. The core writes to internal coprocessor 0 (CP0) registers with the information used to initialize and modify entries in the TLB, then executes a TLB write instruction (TLBWI or TLBWR) to move the data from the registers to the TLB.

2.1.1 TLB Types

The Memory Management Unit (MMU) in the I6500 core consists of four address-translation lookaside buffers (TLB):

- Instruction TLB (ITLB)[1]. Number of ITLB entries varies based on the number of VPs. The ITLB maps only 4 KB, 16 KB, or 64 KB pages. The ITLB is managed by hardware and is transparent to software.
  - 1 VP = 6 entries
  - 2 VPs = 12 entries
  - 4 VPs = 18 entries
- Data TLB (DTLB)[1]. Number of DTLB entries varies based on the number of VPs. The DTLB maps only 4 KB, 16 KB, or 64 KB pages. The DTLB is managed by hardware and is transparent to software.
  - 1 VP = 8 entries
  - 2 VPs = 14 entries
  - 4 VPs = 20 entries
- 16 dual-entry Variable TLB (VTLB) per VP. The larger VTLB is used as a backup structure for the ITLB. If a fetch address cannot be translated by the ITLB or DTLB, the VTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the ITLB/DTLB for future use.

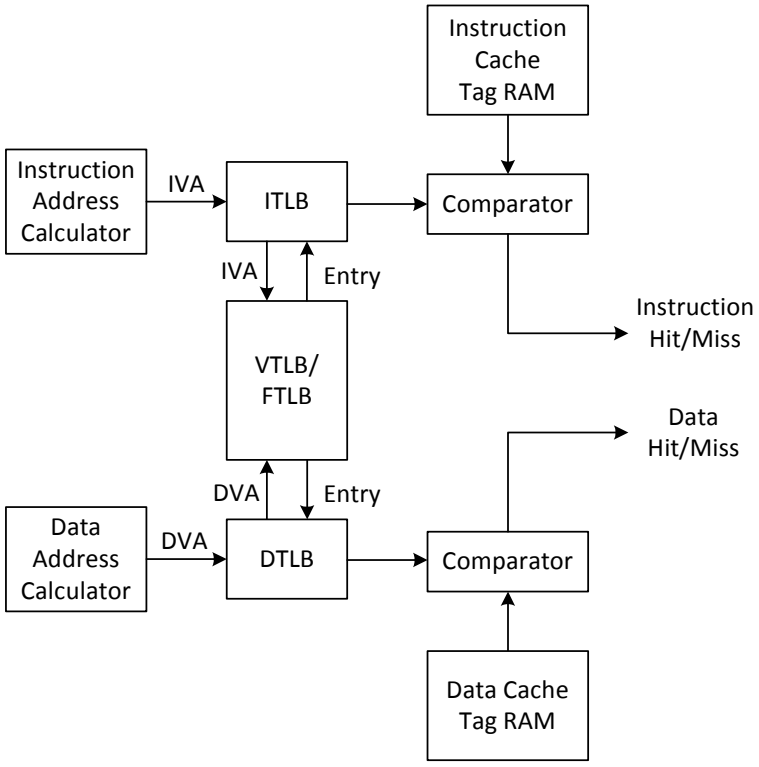
1. The ITLB and DTLB perform address translations for the instruction and data caches respectively. These blocks are not software visible and are shown only for completeness.

Entries are automatically refilled from the VTLB when required, and automatically cleared whenever the associated VTLB is updated.

- 512 dual-entry Fixed TLB (FTLB) that is shared between all VPs. The FTLB extends the size of the VTLB an extra 512 entries and is accessed at the same time as the VTLB when a miss occurs in the ITLB/DTLB.

Figure 2.1 shows an overview of the I6500 MMU architecture.

Figure 2.1 Overview of MMU Architecture in the I6500 Core



When an instruction address is to be translated, the ITLB is accessed first. If the translation is not found, the VTLB/FTLB is accessed. If there is a miss in the VTLB/FTLB, an exception is taken. Similarly, when a data reference is to be translated, the DTLB is accessed first. If the address is not present in the DTLB, the VTLB/FTLB is accessed. If there is a miss in the VTLB/FTLB, an exception is taken. The OS should process the exception by overwriting a TLB entry from the appropriate VTLB or FTLB with the original translation requested.

For more information on the MMU architecture, refer to the *I6500 Technical Reference Manual* included in the documentation package.

2.1.2 TLB Instructions

This section defines the various types of instructions used when accessing the TLB. For information on the Guest TLB instructions used in Virtualization, refer to the Virtualization chapter of this manual.

- **TLBINV** — Invalidates a set of TLB entries based on ASID and Index match. TLB entries which have their G bit set to 1 are not modified.
- **TLBINVF** — Invalidates a set of TLB entries based on Index match.

- **TLBP** — The TLB Probe instruction is used to probe the TLB for a specific virtual address. The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register.
- **TLBR** — The TLB Read instruction causes the *EntryHi, EntryLo0, EntryLo1*, and *PageMask* registers to be loaded with the contents of the TLB entry pointed to by the Index register.
- **TLBWI** — The TLB Write Index instruction causes the TLB entry pointed to by the Index register to be written with the contents of the *EntryHi, EntryLo0, EntryLo1*, and *PageMask* registers.
- **TLBWR** — The TLB Write Random instruction causes a random TLB entry selected by hardware to be written with the contents of the *EntryHi, EntryLo0/1*, and *PageMask* registers.
- **GINVT** — The Global Invalidate TLB instruction provides a way to globally invalidate all TLB entries in multiple ways or the entire TLB. Refer to the Global TLB Invalidate section of this chapter for more information.

2.1.3 Shared FTLB Translations

The I6500 core supports shared FTLB translations across all VPs in a core. On previous generation cores, the FTLB entries were shared across the VPs, but the translations were not. In many applications, there can be multiple threads that are working cooperatively or running the same application on different data. In this situation, some translations are common across VPs and sharing the translations increases the FTLB capacity and reduces contention. Even under Linux, multiple threads can be associated with the same process and use the same translations on different VPs.

To enable sharing of FTLB translations across all VPs in a core and all cores in a cluster, the I6500 uses a feature called MemoryMapID. This feature can be used to replace the traditional Address Space Identifier (ASID) to create a global name space that is common across cores and VPs and will enable the hardware to identify shared translations. However, to maintain backward compatibility, the I6500 core supports both ASID and MMID. Either one can be selected using the CP0 Config5 register.

In the I6500 core, the MemoryMapID (MMID) is a replacement for the ASID. The 16-bit MMID (versus the 10-bit ASID) is sufficient to create a global name space for each Guest ID (GID) that is common across cores and VPs. Aside from reducing the need for a TLB flush when recycling ASIDs, the common name space enables sharing of translations between VPs and globalized TLB invalidates.

When MMID is enabled (Config5.MI = 1), translations are common for all cores/VPs with the same GID + MMID (GID includes GID = 0 as root). This means that the VPID check in the FTLB will be disabled. However, the VPID check is still needed when operating in the legacy mode (Config5.MI = 0).

Software Constraints

When the MMID function is enabled, kernel software must adhere to the following programming constraints:

- All VPs for a given Guest ID (GID) must have the same setting in their Config5.MI field.
- The FTLB must be flushed before a change to Config5.MI.
- All VPs for a given GID must have the same FTLB page size setting.
- Root and Guest need not have the same setting in their Config5.MI fields.

CP0 MMID Register

The MMID is stored in the MemoryMapID register located at CP0 Register 4, Select 5. There is one MemoryMapID register per VP.

Figure 2.2 MemoryMapID Register Format

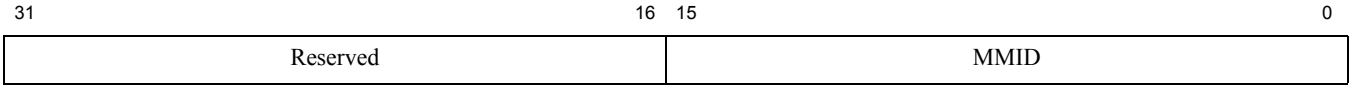


Table 2.1 MemoryMapID Register Descriptions

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	31:16	Reserved.	R	0
MMID	15:0	Stores the memory map ID value used for the translation.	R/W	0

CP0 Config5 Register

The CP0 Config5 register has new fields that indicate if the ASID or MMID mapping is enabled, as well as a 2-bit field that indicates that the core supports global invalidate instructions. Figure 2.3 shows only those bits that are new in the I6500 core. All other Config5 bits remain the same as before.

Figure 2.3 Config5 Register Format

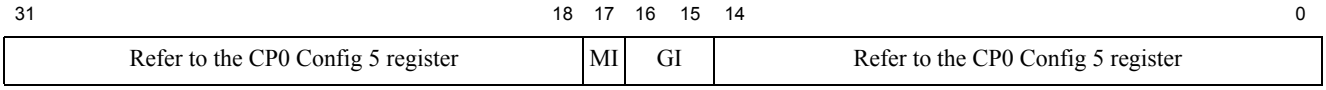


Table 2.2 Config5 Register Field Descriptions

Name	Bit(s)	Description	Read/ Write	Reset State
	31:18	No changes from previous version. Refer to the CP0 Config5 register for more information.		
MI	17	Indicates whether the ASID or MMID is used for FTLB translations. 0: ASID is enabled. 1: MMID is enabled. If MI = 1, MMID writes and reads are allowed, and ASID(X) writes are dropped and read as 0. When MI = 0, ASID(X) writes and reads will access the lower 8+2 bits of the register. MMID writes are dropped and reads return 0.	R/W	0
GI	16:15	Indicates if global invalidate instructions are supported. In the I6500 core, this field is hardwired to 2'b11 to indicate that both global instruction cache and TLB invalidate instructions are supported.	R	2'b11
	14:0	No changes from previous version. Refer to the CP0 Config5 register for more information.		

2.1.4 Global TLB Invalidate

The I6500 core provides kernel software with the ability to globally invalidate the VTLB/FTLB structure using the new GINVT (Global Invalidate TLB) instruction. When this instruction is executed, all entries in the VTLB/FTLB are invalidated in all cores and all clusters. In addition, all Instruction TLB (ITLB) and Data TLB (DTLB) entries that match in the VTLB are also invalidated.

The GINVT instruction provides the option to invalidate the TLB entries in the following ways:

- Invalidate the entire TLB. If the ‘type’ field in the instruction is set to 2’b00, all TLB entries in all cores and all clusters are invalidated, without regard for any virtual address of memory map ID mateh.
- Invalidate by virtual address and Memory Map ID value. If the ‘type’ field is 2’b11, the TLB entries across all cores and clusters are invalidated only for those memory maps that match the MemoryMapID value as well as the virtual address.
- Invalidate by Memory Map ID value only. If the ‘type’ field is 2’b10, the TLB entries across all cores and clusters are invalidated only for those memory maps that match the MemoryMapID value.
- Invalidate by virtual address only. If the ‘type’ field is 2’b01, the TLB entries across all cores and clusters are invalidated only for those addresses that match the virtual address.

2.2 MMU Programming

The following subsections describe some of the programming options for the I6500 MMU. Each section provides CP0 register information listing the register and field(s) used to determine the required information, as well as an assembly code example.

This section is intended to provide examples of how to program the various functions required to manage the MMU. It is a good reference for a programmer writing their own support library, RTOS, or tool chain. Note that most of the functionality of the programming examples provided in this chapter are also provided in the standard tools libraries incorporated into the MIPS Codescape SDK.

2.2.1 Assembly Language Conventions

Throughout the code examples in this document, the CP0 registers are referred to by their register name. For example, the Config4 register is referred to as *C0_Config4*, with C0_ indicating the register is part of the CP0 register set. A separate #define statement indicates that the Config4 register is located at CP0 register 16, select 4 ($16,4). The compiler interprets only the numerical value.

Both variables (register number and select number) are used when the select value is non-zero, such as $16,4. If the select value of a register is 0 (for example, 6,0 for the *Wired* register), the select number is not shown in the #define statement and will be interpreted as zero by the compiler.

2.2.2 Determining the VTLB Size

Register Interface

The 6-bit *MMUSize* field in the *Config1* register (CP0 Register 16, Select 1) indicates the size of the VTLB. This value is loaded by hardware based on the system configuration. In the I6500 core the VTLB size is fixed at 16 entries, so the size of this field is 0x0F (15 decimal).

Determining the VTLB Size Code Example

The following example shows the assembly language instructions used to determine the VTLB size.

```
#define   C0_CONFIG1   $16,1

mfc0      t0, C0_CONFIG1      //read Config1 register and place into t0
ext       t1, t0, 25, 6       //extract value in bits 30:25 and place into t1
addiu     t1, 1               //add 1 to the value in t1
```

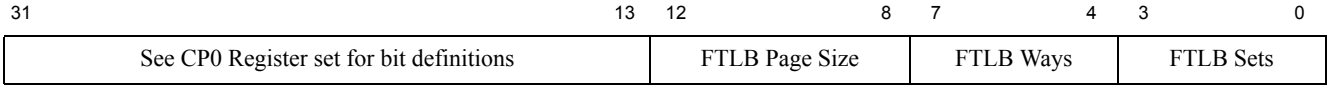
2.2.3 FTLB Page Size Configuration

The FTLB page size and configuration are described in the following subsections.

Register Interface

The I6500 core uses the following CP0 register fields to determine the size and organization of the MMU. The number of FTLB sets and FTLB ways are fixed in the I6500 core. Only the FTLB page size is configurable using the CP0 Config4 register as follows.

Figure 2.4 CP0 Config4 FTLB Register Fields



- Bits 3:0 of the *Config4* register (*Config4*$_{FTLB\ Sets}$) indicates the number of FTLB sets per way. In the I6500 core, this read-only value is always fixed at 64 sets per way.
- Bits 7:4 of the *Config4* register (*Config4*$_{FTLB\ Ways}$) indicates the number of ways in the FTLB. In the I6500 core, this read-only value is always fixed at 4 ways.
- Bits 12:8 of the *Config4* register (*Config4*$_{FTLB\ Page\ Size}$) determines the FTLB page size. This R/W field can be programmed to select pages sizes of 4 KB, 16 KB or 64 KB. The traditional page size has been 4 KB but most OS implementations support 4 KB or 16 KB. MIPS recommends using a 16 KB page size to improve performance by greatly reducing the number of TLB misses. One exception to the 16 KB recommendation is when using the Android OS which only supports a 4 KB page size. This field is encoded as follows:
  - 0x1: 4 KB
  - 0x2: 16 KB
  - 0x3: 64 KB

Setting the FTLB Page Size Code Example

The following example shows the assembly language instructions used to select a 16 KByte page size.

```
#define   C0_CONFIG4    $16,4

mfc0   a0, C0_CONFIG4          // read the Config4 register and place into a0
li     a3, 2                   // set value for a 16k Page size into a3
ins    a0, a3, 8, 5            // insert FTLB Page Size field in bits 12:8 of a0
mtc0   a0, C0_CONFIG4          // write the contents of a0 into the Config4 register
```

2.2.4 VTLB and FTLB Initialization

This section describes the procedure for VTLB/FTLB initialization. When the core is first powered up the TLB is not ready for use. Before virtual addressing can be used the VTLB must be initialized. The FTLB is initialized automatically in hardware and does not require any kernel software involvement.

Register Interface

The following steps and associated CP0 registers are used to initialize the VTLB.

1. Write all zeros to the CP0 *Index*$_{INDEX}$ field.

2. Execute a TLBINVF instruction to initialize all entries in the VTLB.

Initializing the VTLB Code Example

All entries in the VTLB are initialized at the same time using the TLBINVF instruction.

```
#define   C0_INDEX  $0,0

mtc0         zero, C0_INDEX   // move a zero into the Index register
tlbinvf                       // TLB Invalidate Flush flushes internal TLB caches
                              // and invalidates all VTLB entries
```

The instruction sequence above must be executed on each VP before the TLB can be used.

Initializing the FTLB Code Example

In the I6500 the FTLB is initialized in hardware. No kernel software intervention is required to initialize the FTLB.

2.2.5 Indexing the VTLB and FTLB

A 10-bit index value is used to index up to a maximum of 528 dual entries of the VTLB and FTLB. This value is stored in bits 9:0 of the *Index* register (CP0 register 0, Select 0).

The *Index* register determines which TLB entry is accessed by a **TLBWI** instruction. This register is also used for the result of a **TLBP** instruction (used to determine whether a particular address was successfully translated by the core). Note that a **TLBP** instruction which fails to find a match for the specified virtual address sets bit 31 of the *Index* register, indicating a probe failure.

Register Interface

1. Set the $EntryHi_{EHINV}$ bit to indicate that **TLBWI** invalidate is enabled. When this bit is set, the **TLBWI** instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. This bit is ignored on a **TLBWR** instruction.

2. Write the appropriate TLB index to the $Index_{INDEX}$ field. Each VP contains a 16-entry VTLB, which is added to the 512 entry shared FTLB to determine the total number of TLB entries.

Indexing the VTLB Code Example

The following example shows the assembly language instructions used to index VTLB entry 10.

```
#define   C0_INDEX  $0,0

li     t0, 0x0000000A   //set Index field = 10
mtc0   t0, C0_INDEX     //write value into Index register
```

Indexing the FTLB Code Example

The following example shows the assembly language instructions used to index FTLB entry 255.

```
#define   C0_INDEX  $0,0

li     t0, 0x0000010F   //set Index field = 255 + 16 (# of VTLB entries)
mtc0   t0, C0_INDEX     //write value into Index register
```

2.2.6 Programming a TLB Entry

This section describes how to setup the TLB to create a virtual to physical mapping. The TLB entries are filled using the registers listed below. After the registers are loaded with the appropriate information, the *Index* register is used to determine which entry will be written.

Register Interface

Each TLB entry in the VTLB/FTLB consists of a tag portion and dual-data portion as shown in Figure 2.5. In this figure, the following registers are used to manage the TLB entries.

- *Index* — Used when a **`TLBWI`** command is executed. During reset or power-up, this register is set to 0, and this value is written to all VTLB/FTLB entries.
- *EntryHi* — Stores the virtual page number in the TLB tag during normal operation.
- *EntryLo0* — Stores even numbered physical frame number in the TLB data during normal operation.
- *EntryLo1* — Stores odd numbered physical frame number in the TLB data during normal operation.
- *PageMask* — Indicates the TLB page size. This register must be set to the page size of the FTLB when the FTLB is accessed using a **`TLBWI`** command. When a **`TLBWR`** command is executed and the value in the PageMask register does not equal the FTLB page size, the translation is stored in the VTLB. If the value in the PageMask register equals the FTLB page size, the translation will most likely be stored in the FTLB. However, there is a slight chance the translation will be stored to the VTLB.

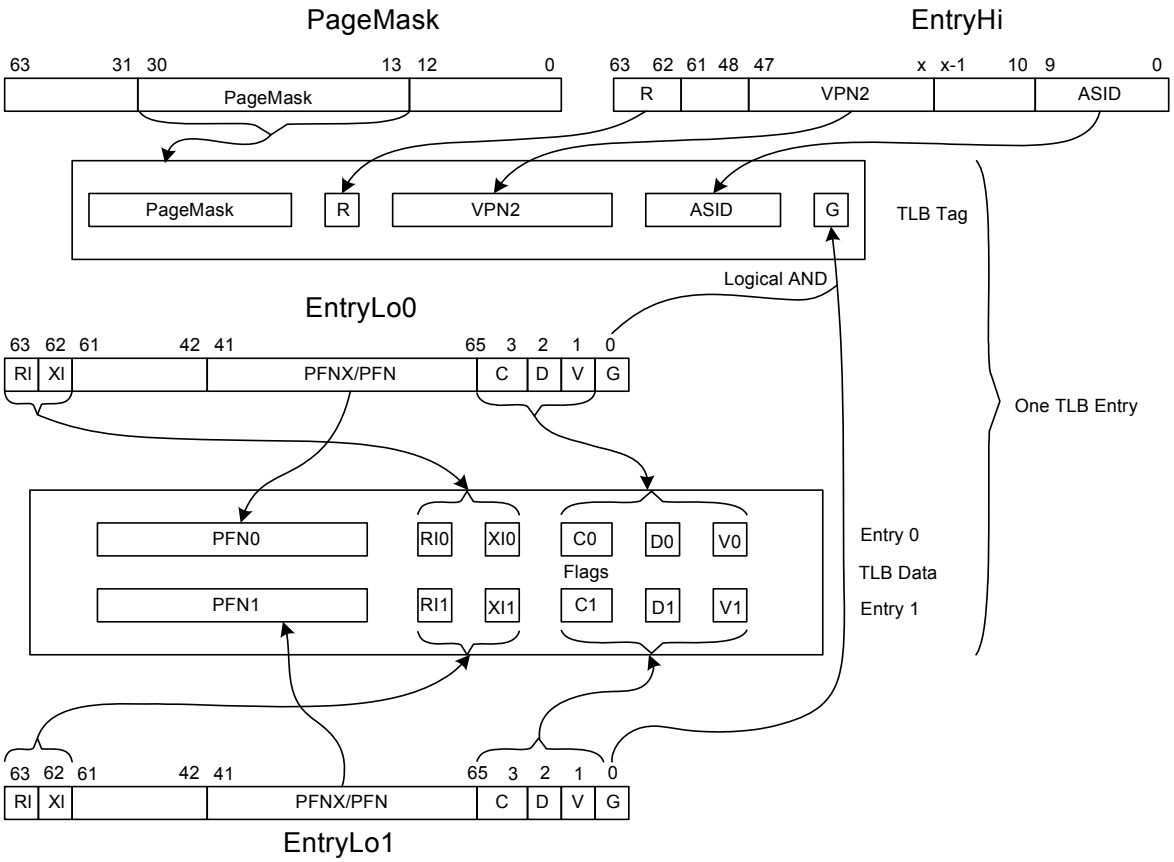
To fill an entry in the VTLB/FTLB, kernel software updates the registers above with the appropriate data, then executes a **`TLBWI`** or **`TLBWR`** instruction. Note that the size of the VPN field in the *EntryHi* register depends on the page size as noted by the 'x' and 'x-1' nomenclature. The registers and corresponding fields to be programmed when accessing the TLB are listed as follows.

- PageMask is set in the CP0 *PageMask* register. This register determines the page size of the TLB entry. For the VTLB, page sizes of 4 KB to 1 GB in powers of four are supported. For the FTLB, page sizes of 4 KB, 16 KB, or 64 KB are supported.
- Virtual address (VPN2) and address space identifier (ASID) are set in the MMU *EntryHi* register. The 'VPN2' designation indicates that this address is for a double-page-size virtual region which maps to a pair of physical pages. The ASID field helps to reduce the frequency of TLB flushing on a context switch. The ASID field extends the virtual address with an 8-bit memory space identifier assigned by the operating system. The ASID allows translations for multiple different applications to co-exist in the TLB (in Linux, for example, each application has different code and data lying in the same virtual address region).

  In the I6500 core, the MemoryMapID (MMID) can be used as a replacement for the ASID. For more information, refer to the section entitled Shared FTLB Translations.
- PFN0/1, C0/1, D0/1, V0/1, G0/1, RI0/1, and XI0/1 bits are set in the MMU *EntryLo0* and *EntryLo1* registers. The '0' indicates the even numbered TLB entry, and the '1' nomenclature indicates the odd numbered TLB entry.
  - The PFN stores the corresponding physical frame number (which along with the offset becomes the physical address).
  - The C field indicates how to cache data for this page. Each data entry can have one of three cache coherency attributes: *Uncached*, *Uncached Accelerated*, or *Cached Coherent Read-Share*.
  - The D bit is the dirty flag and indicates that the page has been written, and/or is writable. If this bit is a one, stores to the page are permitted. If this bit is a cleared, stores to the page cause a *TLB Modified* exception. The D bit should be set in the exception handler on the first write that causes an exception. Software can use this bit to track pages that have been written to by clearing this bit when the page is first mapped.

- The V bit is the valid flag and indicates that the TLB entry, and thus the virtual page mapping, is valid. If this bit is set, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a *TLB Invalid* exception.
- The G bit is the "Global" bit. On a TLB write, the logical AND of the G bits in both the Entry 0 and Entry 1 registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both Entry 0 and Entry 1 reflect the state of the TLB G bit.
- The RI is the read-inhibit flag. If this bit is set in a TLB entry, any attempt to read data on the virtual page causes a a *TLBRI* exception ($PageGrain_{IEC}$ = 1), even if the *V* (Valid) bit is set. The *RI* bit is always writable in the I6500 core since the *RIE* bit of the *PageGrain* register is always set.
- The XI is the execute-inhibit flag. If this bit is set in a TLB entry, any attempt to fetch an instruction from the virtual page causes a TLB Invalid or a TLBXI exception, even if the V (Valid) bit is set. The XI bit is writable only if the XIE bit of the PageGrain register is set, which is always the case in the I6500 core.
- The R field in bits are the region bits that are stored in bits 63:62 of the virtual address and divide the virtual memory map into one of four regions.
- If address sizes larger than 32 bits are used, support for large physical addresses must be enabled by setting the ELPA bit of the CP0 *PageGrain* register. Setting this bit allows for 48-bit physical addresses. When this bit is set, the PFNX field of the *EntryLo0* and *EntryLo1* registers is writable and concatenated with the PFN field to form the full page frame number.

Figure 2.5 Relationship Between CP0 Registers and TLB Entries



Programming a TLB Entry Code Example

The following assemble language example shows how to create a single mapping from virtual address 0x12340000 to physical address 0x23450000 with the C, D, V, and G bits of the entry set to a specific value.

```
#define   C0_ENTRYHI    $10,0
#define   C0_ENTRYLO0   $2,0
#define   C0_ENTRYLO1   $3,0
#define   C0_PAGEMASK   $5,0
#define   C0_INDEX      $0,0

li     t0, 0x12340000          //set VA = 0x12340000, ASID = 0
mtc0   t0, $10                 //write EntryHi with VA and ASID
li     t1, 0x23450000 >> 6     //set PA = 0x23450000 and shift the PFN value right
                               //by 6 bits
li     t2, 0x002F              //load CDVG = 5,1,1,1
dins   t1, t2, 0, 6            //insert CDVG field into EntryLo bits 5:0
mtc0   t1, $2                  //write EntryLo even page
mtc0   zero, $3                //write EntryLo odd page, invalidate
li     t2, 0x00007FFF          //set page size to 16 KB
mtc0   t2, $5                  //write PageMask register
li     t3, 4                   //select VTLB entry 4
mtc0   t3, $0                  //write Index register
tlbwi                          //execute TLBWI instruction to load TLB
```

2.2.7 Hardwiring VTLB Entries

CP0 Programming Interface

The I6500 core allows up to 15 entries of the VTLB to be hardwired such that they cannot be replaced by a TLBWR instruction. This is accomplished using the *Wired* register (CP0 register 6, Select 0). The *Wired* register specifies the boundary between the wired and random entries in the VTLB. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a **TLBWR** instruction. However, wired entries can be overwritten by a **TLBWI** instruction.

The wired entries in the VTLB must be contiguous and start from 0. For example, if the *Wired* field of this register contains a value of 0x5, this indicates that entries 4, 3, 2, 1, and 0 of the VTLB are wired. If the value in the *Wired* register is greater than *Config1.MMUSize*, then the write to the *Wired* register is dropped. The *Wired* register is reset to zero by a Reset exception.

Note that in the I6500 core, the *Wired.Limit* field is set to 0 by default, indicating that all but one of the VTLB entries can be wired. The *Limit* field can be set to the value in the *Config1.MMUSize* field, which leaves at least one entry open for Guest random replacement. Root should configure the *Guest.Config1.MMUSize* field to avoid guest TLB randomization for replacing Root wired entries.

Hardwiring a TLB Entry Code Example

The following example shows the assembly language instructions used to hard-wire a TLB entry. In this example the first 5 entries of the VTLB are wired.

```
#define   C0_WIRED      $6,0

li     t0, 0x000F0005   //set Limit field = 15 and Wired field = 5
mtc0   t0, C0_WIRED     //write value into Wired register
```

2.2.8 FTLB Hashing Scheme and the TLBWI Instruction

The I6500 core uses a hashing scheme based on VPN and page size to index a particular FTLB set in order to maintain consistency.

When a TLBWI instruction is executed targeting the FTLB, the *Index* register must be consistent with the FTLB set calculated from the *EntryHi* and *PageMask* registers. If not, a machine check exception is taken. This scheme is used only when the $EntryHi_{EHINV}$ bit is 0. When the $EntryHi_{EHINV}$ bit is 1, hashing is ignored and the indexing entries are invalidated.

When a TLBWR instruction is executed and the *Pagemask* register matches the page size the FTLB currently supports, hardware uses the hashing scheme to calculate the FTLB set and choose a random way to index the entry. The $EntryHi_{EHINV}$ bit is ignored for a TLBWR instruction.

2.3 TLB Exception Handler

In the event that a TLB miss occurs in either the VTLB or the FTLB, the I6500 core allows for the following types of TLB exceptions.

- Address error (AdEL or AdES)
- TLB Refill (TLBL, TLBS)
- TLB Invalidate (TLBL, TLBS)
- TLB Read Inhibit (TLBRI)
- TLB Execute Inhibit (TLBXI)
- TLB Modified (TLBM)
- FTLB Parity
- Machine Check

The *Address Error* exceptions (AdEL and AdES) are used in kernel, user, and supervisor modes.

- On a load in user mode, an *AdEL* exception is taken when user mode does not have permission for the address being accessed.
- On a store in user mode, an *AdES* exception is taken when user mode does not have permission for the address being accessed.
- On a load in supervisor mode, an *AdEL* exception is taken when supervisor mode does not have permission for the address being accessed.
- On a store in supervisor mode, an *AdES* exception is taken when supervisor mode does not have permission for the address being accessed.

The *TLB Refill* exception (TLBL, TLBS) is taken on any TLB miss regardless of the operating mode and uses the following TLBL and TLBS opcodes.

- TLBL exception: On a non-store in any mode, there is a TLB miss.
- TLBS exception: On a store in any mode, there is a TLB miss.

The *TLB Invalidate* exceptions (TLBL and TLBS) are taken under the following conditions.

- TLBL exception: On a non-store, there is a TLB hit, but the valid bit for that TLB entry is not set.
- TLBS exception: On a store in any mode, there is a TLB hit, but the valid bit for that TLB entry is not set.

The *TLB Read Inhibit* exception (TLBRI) is taken when there is a TLB hit during a load operation, the RI bit of the entry is set, and the $PageGrain_{IEC}$ and $PageGrain_{RIE}$ bits are set, which is always the case in the I6500 core.

The *TLB Execute Inhibit* exception (TLBXI) is taken when there is a TLB hit during an instruction fetch, the XI bit of the entry is set, and the $PageGrain_{IEC}$ and $PageGrain_{XIE}$ bits are set, which is always the case in the I6500 core.

A *TLB Modified* exception is taken whenever there is a TLB hit on a store and the Dirty bit associated with that entry is not set.

An *FTLB Parity* exception is taken whenever a parity error occurs on an FTLB read. The FTLB parity exception is taken only when bit 31 of the CP0 *Error Control* register ($ErrCtl_{.PE}$) is set. If this bit is cleared, FTLB parity errors are ignored.

The *Machine Check* exception occurs when the processor detects an internal inconsistency. The machine check exception can be either precise or imprecise depending on the type of error. The following conditions cause a machine check exception:

- A TLBWI instruction to the FTLB and the index and VPN2 are not consistent when the $EntryHi_{EHINV}$ bit is not set.
- A TLBWI instruction to the FTLB and the PageMask register does not correspond to the FTLB page size setting in bits 12:8 of the Config4 register ($Config4_{FTLB\ Page\ Size}$).
- A TLBP instruction and a duplicate/overlap is detected across the FTLB/VTLB.
- Any TLB lookup and a duplicate/overlap is detected across the FTLB/VTLB.

Register Interface

The I6500 core uses the following CP0 registers to manage TLB exceptions.

- *Context* (CP0 register 4, Select 0): Contains the pointer to an entry in the page table entry (PTE) array.
- *XContext* (CP0 register 20, Select 0): The *XContext* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations.The *XContext* register is primarily intended for use with the XTLB Refill handler, but is also loaded by hardware on a TLB Refill. The *XContext* register duplicates some of the information provided in the BadVAddr register.
- *BadVAddr* (CP0 register 8, Select 0): The 64-bit *BadVAddr* register is a read-only register that captures the most recent virtual address that caused the exception. The *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.
- *BadInstr* (CP0 register 8, Select 1): The 64-bit *BadInstr* register is a read-only register that captures the most recent instruction which caused the exception to occur. The *BadInstr* register is provided to allow acceleration of instruction emulation. The *BadInstr* register is only set by exceptions which are synchronous to an instruction.
- *BadInstrP* (CP0 register 8, Select 2): The *BadInstrP* register is used in conjunction with the *BadInstr* register. The *BadInstrP* register contains the prior branch instruction, when the faulting instruction is in a branch delay slot.

For more information on these registers, refer to the *CP0 Registers* companion document provided in the documentation package.

TLB Exception Handler Code Example

The exception handler can directly use the value in the CP0 *Context* register as the memory address to read the EntryLo0/1 settings. The processor also writes the Virtual Page Number (VPN) that missed to the *EntryHi* register so it is ready to write the TLB entry. The following example shows the assembly language implementation of a TLB exception handler for 32-bit addressing mode.

```
    .set noreorder
    #define   C0_ENTRYLO0   $2,0
    #define   C0_ENTRYLO1   $3,0
    #define   C0_CONTEXT    $4,0
    #define   C0_XCONTEXT   $20,0

TLBmiss32:

    dmfc0  k1, C0_CONTEXT   // Get Context register (CP0 register 4)
    ld     k0, 0(k1)        // Load EntryLo0 into K0
    ld     k1, 8(k1)        // Load EntryLo1 into k1
    dmtc0  k0, C0_ENTRYLO0  // Move k0 to CP0 EntryLo0 (CP0 register 2)
    dmtc0  k0, C0_ENTRYLO1  // Move k0 to CP0 EntryLo1 (CP0 register 3)
    ehb                     // Clear hazard barrier to insure CP0 write takes effect
    tlbwr                   // Write to random TLB entry
    eret                    // Return from TLB exception
```

For 64-bit addressing mode the first instruction in the above example needs to read the XContext register instead of the Context register:

```
    dmfc0  k1, C0_XCONTEXT     // Get XContext register (CP0 register 20)
```

Note that some operating systems like Linux use a 3-level Page Table and do not use the Context or XContext registers for page table lookup. Instead they use the CP0 *BadVaddr* register and their own scheme to access the correct page table entry. Refer to the Linux OS documentation for details on the page table handling.

2.4 Additional Information

The MMU chapter of the *I6500 Technical Reference Manual* serves as a supplement to this chapter and provides additional information about the MMU, including an overview of virtual to physical address translation with 4 KByte, 16 KByte, and 64 KByte page size examples, address translation flow to illustrate the circumstances under which TLB exceptions are taken, FTLB parity errors, address error detection, multi-threading considerations, guest and root operating systems, and an in depth discussion of how to select between 32- and 64-bit addressing modes, and the associated address mapping in the kernel, supervisor, user, and debug operating modes.

Chapter 3

Caches

The I6500 Multiprocessing System contains the following caches: L1 instruction, L1 data, and shared L2. These caches provide on-chip temporary storage of information that can be retrieved much faster than accessing main memory. The dedicated L1 instruction and data caches have the fastest access times and are accessed first. If the data is not present in the L1 cache, the shared L2 cache is accessed. The L2 cache contains both data and instructions, hence the name ‘shared’. If the requested data is not in the L2 cache, the main memory is accessed.

This chapter provides an overview of the cache architecture and a description of the elements that go into programming the caches. A description of the CP0 register interface to each cache is provided, as well as cache initialization code. Other programmable elements include setting up cache coherency and handling cache exceptions.

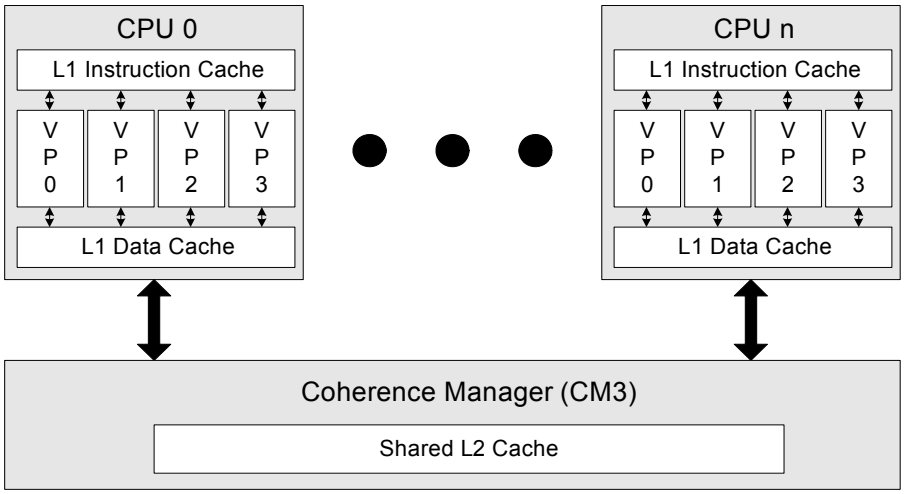
3.1 Cache Subsystem Overview

In the I6500 MPS, the size of each cache can be configured as follows:

- L1 Instruction Cache: 32 KB or 64 KB
- L1 Data Cache: 32 KB or 64 KB
- L2 Cache: 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, or 8 MB

Figure 3.1 shows the relative location of the caches within the I6500 Multiprocessing System. The L1 instruction and L1 data caches are shared by all VP’s in the same core. The L2 cache is shared by all cores.

Figure 3.1 I6500 Multiprocessing System Caches



3.1.1 L1 Instruction Cache

The L1 instruction cache contains two arrays: tag and data. The L1 instruction cache is virtually indexed and physically tagged. An instruction cache data entry contains eight 64-bit doublewords in the line, for a total of 64 bytes.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. An instruction cache tag entry consists of 36-bit physical address bits and 7 Error Correction Code (ECC) bits.

3.1.2 L1 Data Cache

The L1 data cache contains two arrays: tag and data. The L1 Data cache is physically indexed and physically tagged, thus eliminating the chance of a virtual aliasing.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. A tag entry consists of the upper 34 or 35 bits of the physical address (depending on cache size), two coherent state bits, and some ECC bits. A data entry contains 64 bytes of data and associated ECC bits. All 64 bytes in the line are present in the data array together, hence the coherent state bits (2) stored with the tag.

After a valid line is resident in the cache, a store operation can update all or a portion of the words in that line depending on the type of store.

The data cache uses ECC so that single-bit errors can be corrected. ECC code is generated across a 32-bit word. Sub-word stores are handled by doing a read-modify-write sequence. The error checking and correction process is handled entirely by hardware and is transparent to kernel software.

3.1.3 L2 Cache

The L2 cache processes transactions that miss in the L1 caches. The L2 cache is larger than the L1 caches. In the I6500 Multiprocessing System, the L2 cache is integrated into the Coherence Manager (*CMrev*). The L2 communicates with external memory via an AXI-4 interface. The L2 communicates with the cores through the proprietary MIPS Coherence Protocol (MCP) bus.

The associativity of the L2 cache can be either 8 or 16 ways. The 8-way option is used when the cache size is 256 KB. The 16-way option is used for all other cache sizes. The line size is fixed at 64 bytes. The number of sets and ways is selected during the build process and cannot be changed by the kernel software. Software can check the set size by reading the GCR_L2_CONFIG register, which is part of the *CMrev* register address space. Refer to the *Coherence Manager* chapter for more information.

Table 3.1 shows the list of possible L2 cache configurations.

Table 3.1 L2 Cache Configurations

Line Size	Sets per Way	Number of Ways	Total L2 Cache Size
64 bytes	512	8	256 KBytes
64 bytes	512	16	512 KBytes
64 bytes	1024	16	1 MByte
64 bytes	2048	16	2 MBytes
64 bytes	4096	16	4 MBytes
64 bytes	8192	16	8 MBytes

3.1.4 Cache Instructions

Operations are performed on the L1I, L1D, and L2 caches using the following instructions:

- **CACHE** — This instruction is used to perform various operations on the L1 instruction and data caches and the L2 cache. These operations are described in Table 3.2.
- **PREF** — This instruction causes data to be moved to or from the cache, to improve program performance. PREF does not cause addressing-related exceptions, including TLB exceptions.
- **SYNCI** — This instruction synchronizes a data cache line with an instruction cache line. This instruction should be used when writing to the program image in memory to make the newly stored instruction opcodes visible to the instruction fetch logic via the I-Cache. The SYNCI instruction operates on all instruction caches in a cluster. In a multi-cluster system, this means all L1 instruction caches in all clusters.
- **GINVI** — This instruction is new to the I6500 and can be used to invalidate all L1 instruction caches in the system. In a multi-cluster system, this means all L1 instruction caches in all clusters.

The *SYNCI* and *CACHE I Hit Invalidate* instructions are "globalized", which means that they will invalidate the targeted cache line from all L1 instruction caches in the system. In multi-cluster systems, the CACHE L2 Hit Invalidate, L2 Hit Writeback, and L2 Hit Writeback Invalidate operations are globalized and will perform the specified operation on all L2 caches in the system (including any L1 D-Cache operations required to maintain inclusivity). Note that the I6500 MPS does not globalize the CACHE D Hit Invalidate, D Hit Writeback, or D Hit Writeback Invalidate instructions; these instructions only affect the L1 D-Cache of the core that executed the instruction.

For more information on how these instructions are used, refer to the example in the section entitled Cache Initialization Routines.

Table 3.2 shows the various types of operations that can be performed using the CACHE instruction. In this table, bits 20:18 of the instruction encode the type of operation as shown in the *Code* column.

Bits 17:16 of the instruction indicate the type of cache being accessed as shown in the *Cache* column:

- I indicates L1 instruction cache — Bits [17:16] = 2’b00
- D indicates L1 data cache — Bits [17:16] = 2’b01

- S indicates L2 or secondary cache — Bits [17:16] = 2’b11

Table 3.2 Encoding of Bits [20:18] of the CACHE Instruction

Code	Cache	Name	Operation
3’b000	I	Index Invalidate	Set the state of the cache line at the specified index to invalid. This encoding may be used by kernel software to invalidate the entire instruction cache by stepping through all valid indices.
	D, S	Index Writeback Invalidate	If the state of the cache line at the specified index is valid and dirty, write the line back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache line to invalid. If the line is valid but not dirty, set the state of the line to invalid. This encoding may be used by kernel software to invalidate the entire data cache by stepping through all valid indices, except during cache initialization. Note that Index Store Tag should be used to initialize the cache at power-up. For the L2 cache, this operation will modify the L1 data caches as needed to maintain inclusivity.
3’b001	I	Index Load Tag	Read the tag for the cache line at the specified index into the *ITagLo* register. Read the data corresponding to the dword index into the *IDataLo* and *IDataHi* registers. The data ECC bits are stored to the CP0 IDataHi register, and the tag ECC bits are stored to the CP0 ITagLo register.
	D	Index Load Tag	Read the tag for the cache line at the specified index into the CP0 DTagLo register and read the data corresponding to the word index into the DDataLo register. The data ECC bits are read into the CP0 DDataHi register and the tag ECC bits are read into the CP0 DTagLo register.
	S	Index Load Tag	Read the tag for the cache line at the specified index into the CM *CGR_L2_TAG_ADDR* register at offset 0x0600. Read the data corresponding to the dword index into the CM GCR_L2_DATA register at offset 0x0610.

Table 3.2 Encoding of Bits [20:18] of the CACHE Instruction *(continued)*

Code	Cache	Name	Operation
3’b010	I	Index Store Tag	Write the tag (and data) for the cache line at the specified index from the *ITagLo* register. The data comes from the *IDataLo* register. The hardware automatically generates the ECC bits to write into the cache. For test purposes, the ECC bits from the ITagLo and IDataHi registers are used instead of the automatically generated values when the ErrCtl.PO bit is set. This operation may be used by kernel software to initialize the entire instruction cache by stepping through all valid indices. Doing so requires that the ITagLo register be initialized first.
	D	Index Store Tag	Write the tag and data for the cache line at the specified index from the CP0 *DTagLo and DDataLow* registers. The hardware automatically generates the ECC bits to write into the cache. For test purposes, the ECC bits from the DTagLo and DDataHi registers are used instead of the automatically generated values when the ErrCtl.PO bit is set. This operation may be used by kernel software to initialize the entire data cache by stepping through all valid indices. Doing so requires that the DTagLo register be initialized first.
	S	Index Store Tag	Write the tag for the L2 cache line at the specified index from the CM *CGR_L2_TAG_ADDR* register at offset 0x0600. By default, the tag ECC value is automatically calculated. For test purposes, the ECC bits from the CM *GCR_L2_ECC* register are used if thecorresponding bits are set inthe L2_CONFIG GCR register located in CM address space.
3’b011	I, D	Reserved	Executed as a no-op.
	S	Index Store Data	Write the CM *GCR_L2_DATA* register contents at the way and dword index specified. The ECC bits are always generated by the hardware.
3’b100	I, S	Hit Invalidate	If the cache line contains the specified address, set the state of the cache line to invalid. This operation may be used by kernel software to invalidate a range of addresses from the caches by stepping through the address range by the line size of the cache. This instruction is globalized for the I caches, meaning that when executed, the instruction will invalidate the targeted cache line from all L1 instruction caches in the system. For the L2 cache, the instruction would invalidate all targeted cache lines within all L2 caches in all clusters. For the L2 cache, this operation will modify the L1 data caches as needed to maintain inclusivity.
	D	Hit Invalidate	If the cache line contains the specified address, set the state of the cache line to invalid. This operation may be used by kernel software to invalidate a range of addresses from the caches by stepping through the address range by the line size of the cache. Note that the I6500 MPS does not globalize the CACHE D Hit Invalidate instruction. This instruction only affects the L1 D-Cache of the core that executed the instruction.

Table 3.2 Encoding of Bits [20:18] of the CACHE Instruction *(continued)*

Code	Cache	Name	Operation
3’b101	I	Fill	Fill the cache from the specified address. The cache line is refetched even if it is already in the cache. In that case, the existing copy in the cache is invalidated
	D, S	Hit WriteBack Invalidate	If the cache line contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache line to invalid. If the line is valid but not dirty, set the state of the line to invalid. This operation may be used by kernel software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. Note that the I6500 MPS does not globalize the CACHE D Hit Writeback Invalidate instruction. This instruction only affects the L1 D-Cache of the core that executed the instruction. For the L2 cache, this operation will modify the L1 data caches as needed to maintain inclusivity.
3’b110	D, S	Hit WriteBack	If the cache line contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. Note that the I6500 MPS does not globalize the CACHE D Hit Writeback instruction. This instruction only affects the L1 D-Cache of the core that executed the instruction. For the L2 cache, this operation will modify the L1 data caches as needed to maintain inclusivity.
3’b111	I, D	Fetch and Lock	The Fetch and Lock encoding is not supported in the I6500 L1 instruction and data caches. For the L1 instruction and data caches this operation executes as a no-op.
	L2	Fetch and Lock	If the L2 cache does not contain the specified address, fill it from memory and writeback the data from the line being replaced. Set the state to valid and locked. If the cache already contains the specified address, set the state to locked. The way selected on fill from memory is the least recently used. The lock state is cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation with the lock bit reset in the associated *STATE* field of the GCR *L2 Tag RAM Cache Op Address* register. It is illegal to lock all ways at a given cache index.

3.2 Cache Coherency

The I6500 core defines a set of Cache Coherency Attributes (CCA). The cache coherency can be set in one of three ways:

- In KSEG0 space, coherency is set using the CP0 *Config.K0* field.
- Using the TLB entry for mapped address regions.
- Using the XKPHYS memory segments.

The I6500 core supports the following cacheability attributes:

- *Uncached (code #2)*: Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- *Cacheable, coherent, write-back, write-allocate, read misses request shared. (code #5)*: Use coherent data. Load misses request data in the shared state (will get exclusive if the data is not being shared by another CPU). Multiple caches can contain data in the shared state. Stores bring data into the cache in an exclusive state - no other caches can contain that same line. If a store hits on a shared line in the cache, the line is updated to the exclusive state and any shared copies of the line in other L1 data caches are invalidated.
- *Uncached Accelerated (code #7):* Uncached stores are gathered together for more efficient bus utilization.

The mapping of cacheability attributes is shown in Table 3.3.

Table 3.3 Mapping of Cacheability Attributes for the K0 Field in the CP0 Config Register

K[2:0][1]	Attribute
3'b000	Mapped to '3b101 (code #5).
3'b001	Mapped to '3b101 (code #5).
3'b010	Uncached.
3'b011	Cacheable. Mapped to '3b101 (code #5).
3'b100	Mapped to '3b101 (code #5).
3'b101	Cacheable, coherent, write-back, write-allocate, read misses request shared.
3'b110	Mapped to '3b101 (code #5).
3'b111	Uncached Accelerated.

1. This field is also mapped to the C field in bits 5:3 of the CP0 EntryLo0 and EntryLo1 registers.

3.3 Self-modified Code

When the processor writes memory with new instructions at run-time, software must accommodate the Harvard architecture and write-back policy of the I6500 L1 caches. When using cacheable memory accesses (CCA = 3 or CCA = 5), the following steps must be taken to prevent execution of the previous (stale) contents of the modified memory addresses:

1. Any stale instructions must be invalidated from the L1 I-Cache. The SYNCI, CACHE I Hit Invalidate, CACHE I Index Invalidate or GINVI instructions can be used for this purpose. The SYNCI and CACHE I Hit Invalidate instructions are globalized, which means they invalidate the targeted line from all I-Caches in the system. The GINVI instruction can be more efficient when writing a large block of instructions as it invalidates the entire I-Cache with a single instruction.

2. The processor must wait until all of the new instructions have been written to the L1 D-Cache before attempting to fetch and execute the new instructions. This can be accomplished by the SYNC and JALR.HB or ERET instructions.

Note that unlike some other MIPS cores, on an I-Cache miss the I6500 core fetches the latest coherent data from the L1 D- and L2-Caches (including from other cores and clusters) so there is no need to force a writeback from the L1 D-Cache.

The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. The SYNCI instruction could be replaced with an appropriate CACHE or GINVI instruction (when access to Coprocessor 0 is available), and that the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate. A SYNC instruction is required between the final SYNCI instruction in the loop and the instruction that clears instruction hazards.

```
/*
* This routine makes changes to the instruction stream effective to the
* hardware. It should be called after the instruction stream is written.
* On return, the new instructions are effective.
*

* Inputs;
* a0 = Start address of new instruction stream
* a1 = Size, in bytes, of new instruction stream
*/
        beq a1, zero, 20f                   /* If size==0, */
        nop                                 /*   branch around */
        daddu a1, a0, a1                    /* Calculate end address + 1 */
        rdhwr v0, HW_SYNCI_Step             /* Get step size for SYNCI */
        beq v0, zero, 20f                   /* If no caches require synchronization, */
        nop                                 /* branch around */
10: synci 0(a0)                             /* Synchronize all caches around address */
        daddu a0, a0, v0                    /* Add step size in delay slot */
        sltu v1, a0, a1                     /* Compare current with end address */
        bne v1, zero, 10b                   /* Branch if more to do */
        nop                                 /* branch around */
        sync                                /* Clear memory hazards */
20: jr.hb ra                                /* Return, clearing instruction hazards */
        nop
```

3.4 Register Interface

This section provides information on the CP0 registers used to manage the L1 instruction and data caches, and the L2 cache.

3.4.1 L1 Instruction Cache Control Registers

The I6500 core uses the following CP0 registers for instruction cache operations.

Table 3.4 Instruction Cache Register Interface

CP0 Registers	CP0 Register Number
Config	*Register 16, Select 0*
Config1	*Register 16, Select1*
CacheErr	*Register 27, Select 0*
ITagLo	*Register 28, Select 0*
IDataLo	*Register 28, Select 1*
IDataHi	*Register 29, Select 1*

3.4.1.1 Config Register (CP0 register 16, Select 0)

In the Config register - the K0 field contains the cache attributes for the unmapped KSEG0 memory region.

3.4.1.2 Config1 Register (CP0 register 16, Select 1)

This register allows kernel software to obtain the L1 cache parameters such as number of sets, line size, and cache associativity.

The *Config1.IS* field (bits 24:22) indicates the number of sets per way in the instruction cache. The I6500 L1 instruction cache supports 128 sets per way, which is used to configure as a 32 KB cache, or 256 sets per way, which is used to configure a 64 KB cache.

The *Config1.IL* field (bits 21:19) indicates the line size for the instruction cache. The I6500 L1 instruction cache supports a fixed line size of 64 bytes as indicated by a value of 5 for this field.

The *Config1.IA* field (bits 18:16) indicates the set associativity for the instruction cache. The I6500 L1 instruction cache is fixed at 4-way set associative as indicated by a value of 3 for this field.

These fields are all read-only; their values are determined during IP configuration and are built into the core.

3.4.1.3 CacheErr Register (CP0 register 27, Select 0)

The *CacheErr* register contains information regarding the type of cache error that occurred. This register provides information such as:

- Correctable or uncorrectable error
- Array where the error occurred; L1I tag/data, L1D tag/data. FTLB tag/data, or L2 tag/data
- The cache way where the error was detected

3.4.1.4 L1 Instruction Cache TagLo Register (CP0 register 28, Select 0)

This register stores the cache address tag information of a cache line being read/written by the CACHE load tag/store tag operations (bits 47:14 for a 64-KByte cache, and bits 47:13 for a 32-KByte cache). The register also stores the ECC bits associated with the tag entry. Separate Valid and Error bits indicate that the tag entry is valid, or if an ECC error has occurred.

3.4.1.5 L1 Instruction Cache DataLo Register (CP0 register 28, Select 1)

This register holds the data (instruction opcodes) being read/written by the CACHE load tag/store tag operations. Two registers (*IDataHi*, *IDataLo*) are needed to store both the data and instruction precode information (calculated by hardware unless overridden). This register stores the 64 bits of the load data.

3.4.1.6 L1 Instruction Cache DataHi Register (CP0 register 29, Select 1)

This register works in conjunction with the *IDataLo* register described above to store the associated instruction precode bits, error information, and ECC status.

3.4.2 L1 Data Cache Control Registers

The I6500 core uses the following CP0 registers for data cache operations.

Table 3.5 Data Cache Register Interface

CP0 Registers	CP0 number
Config	*Register 16, Select 0*
Config1	*Register 16, Select 1*
CacheErr	*Register 27, Select 0*
DTagLo	*Register 28, Select 3*
DDataLo	*Register 28, Select 3*
DDataHi	*Register 29, Select 3*

3.4.2.1 Config Register (CP0 register 16, Select 0)

In the Config register - the K0 field contains the cache attributes for the unmapped KSEG0 memory region.

3.4.2.2 Config1 Register (CP0 register 16, Select 1)

The Config1 CP0 register allows kernel software to obtain the following information about the L1 data cache.

The *Config1.*$_{DS}$ field (bits 15:13) indicates the number of sets per way in the data cache. The I6500 L1 data cache supports 128 or 256 sets per way, which corresponds to a 32 KB or 64 KB cache, respectively.

The *Config1.*$_{DL}$ field (bits 12:10) indicates the line size for the data cache. The I6500 L1 data cache supports a fixed line size of 64 bytes as indicated by a value of 5 for this field.

The *Config1.*$_{DA}$ field (bits 9:7) indicates the set associativity for the data cache. The I6500 L1 data cache is fixed at 4-way set associative as indicated by a value of 3 for this field.

These fields are all read-only; their values are determined during IP configuration and are built into the core.

3.4.2.3 CacheErr Register (CP0 register 27, Select 0)

The *CacheErr* register contains information regarding the type of cache error that occurred. This register provides information such as:

- Correctable or uncorrectable error
- Array where the error occurred; L1I tag/data, L1D tag/data. FTLB tag/data, or L2 tag/data
- Fatal or non-fatal error
- The cache way where the error was detected
- Which one of four words in the L1 data cache data line caused the error

3.4.2.4 L1 Data Cache TagLo Register (CP0 register 28, Select 2)

This register stores the data cache address tag information of a cache line being read/written by the CACHE load tag/store tag operations (bits 47:14 for a 64-KByte cache, and bits 47:13 for a 32-KByte cache). The register also stores

the coherence state (MESI) and ECC bits associated with the tag entry. An error flag bit indicates that an ECC error was detected by the index load tag operation.

3.4.2.5 L1 Data Cache DataLo Register (CP0 register 28, Select 3)

This is a staging register for a special **CACHE** instruction, which loads or stores data from or to the cache line. This register stores the lower 32 bits of the load data.

3.4.2.6 L1 Data Cache DataHi Register (CP0 register 29, Select 3)

This is a staging register for a special **CACHE** instruction, which loads or stores data from or to the cache line. This register stores the ECC information from the load data, as well as a bit to indicate if the hardware detected an ECC error during the IndexLoadTag operation.

3.4.3 L2 Cache CM GCR Control Registers

The I6500 Coherency Manager (CM) uses the following GCR registers for L2 cache operations. Note that these registers are located in CM address space at the offsets shown. They are not located in CP0 space like the L1 instruction and data cache control registers. This is unlike most previous MIPS cores which do store L2 configuration information in the CP0 registers. The CP0 Config5.L2C field indicates that the L2 cache information is stored in a memory-mapped register instead of CP0.

Table 3.6 L2 Cache GCR Register Interface

GCR Registers	Offset Address	Address Space
GCR_ERR_CONTROL	0x0038	GCR Global
L2_CONFIG	0x0130	GCR Global
L2_RAM_CONFIG	0x0240	GCR Global
L2_PFT_CONTROL	0x0300	GCR Global
L2_PFT_CONTROL_B	0x0308	GCR Global
GCR_L2_TAG_ADDR	0x0600	GCR Global
GCR_L2_TAG_STATE	0x0608	GCR Global
GCR_L2_DATA	0x0610	GCR Global
GCR_L2_ECC	0x0618	GCR Global
GCR_L2SM_COP	0x0620	GCR Global
GCR_L2SM_TAG_ADDR_COP	0x0628	GCR Global
CPC_STAT_CONFIG	0x0008	CPC Local

Refer to the *CM Coherence Manager* chapter for more information on accessing these registers.

3.4.3.1 GCR_ERR_CONTROL (Offset 0x0038)

In this register, the L2_ECC_SUPPORTED field indicates that the L2 cache has ECC logic. The L2_ECC_EN bit enables ECC.

3.4.3.2 L2_Config Register (Offset 0x0130)

The L2_Config register provides information on the L2 cache configuration. This register contains the following information:

- Read-only fields that provide the organization of the cache (set size, line size, and associativity).
- L2 bypass mode.
- Tag and data ECC write protocol. When these bits are set, the contents of the respective TAG_ECC and DATA_ECC registers are written into the ECC portion of the L2 RAM when an L2 store cacheop is executed.

3.4.3.3 L2_RAM_Config Register (Offset 0x0240)

This register contains three 2-bit fields that indicate the number of wait states for the Tag RAM's, Data RAM's and Way Select RAM's. Another read-only bit is set by hardware when the hardware cache initialization is complete. This register also contains support for HCI supported/done, L2 dynamic sleep mode, and L2 dynamic sleep wake-up delay.

3.4.3.4 L2_PFT_Control Register (Offset 0x0300)

This register contains information on the L2 hardware prefetcher. This includes a prefetcher enable bit, the number of L2 prefetchers in the system, and a mask field that indicates the minimum operating system page size.

3.4.3.5 L2_PFT_Control_B Register (Offset 0x0308)

This register contains additional information on the L2 hardware prefetcher, including how the prefetch unit handles coherent write invalidate requests, L2 prefetching enable per port ID, and global code prefetch enable.

3.4.3.6 L2_TAG_ADDR Register (Offset 0x0600)

This register is loaded with the tag address from the L2 Tag RAMs when the L2 Load Tag CACHE instruction is executed. The value of this register is written to the address portion of the L2 Tag RAM when an L2 Store Tag CACHE instruction is executed.

3.4.3.7 L2_TAG_STATE Register (Offset 0x0608)

This register is loaded with state information from the L2 Tag RAMs and LRU information when the L2 Load Tag CACHE instruction is executed. The value of this register is written to the tag state information portion of the L2 Tag RAM and the LRU data of the LRU and WS RAMs when an L2 Store Tag CACHE instruction is executed. This register contains a tag state field, and a LRU state field.

3.4.3.8 L2_DATA Register (Offset 0x0610)

This register is loaded with data information from the L2 Data RAMs when the L2 Load Data CACHE instruction is executed. The value of this register is written to the L2 Data RAM when an L2 Store Data CACHE instruction is executed. This register contains a 64-bit data field.

3.4.3.9 L2_DATA_ECC Register (Offset 0x0618)

This register is loaded with the ECC information from the L2 Tag and Data RAMs when the L2 Load Tag CACHE instruction is executed. If the *GCR_L2_CONFIG.COP_DATA_ECC_WE* bit is set then value of the DATA_ECC register is written to the ECC portion of the L2 Data RAM when a L2 Store Tag CACHE instruction is executed. If the *GCR_L2_CONFIG.COP_TAG_ECC_WE* bit is set then value of the TAG_ECC register is written to the ECC portion of the L2 Tag RAM when a L2 Store Tag CACHE instruction is executed.

3.4.3.10 L2SM_COP Register (Offset 0x0620)

This register controls the L2 cache state machine during initialization, flush, and burst operations. The state machine can be started and stopped using the L2SM_COP_CMD field in bits 1:0. The L2SM_COP_TYPE field indicates the type of operation to be performed. The L2SM_COP_MODE bit indicates whether the L2 state machine is idle or run-

ning. The L2SM_COP_RESULT field in bits 8:6 is a read-only field that indicates when the operation is complete and if errors were encountered.

3.4.3.11 L2SM_TAG_ADDR_COP Register (Offset 0x0628)

For L2 cache burst operations, this register is used to set the starting address in the cache using the L2SM_COP_START_TAG_ADDR field in bits 47:6. This field indicates the address at where the operation begins. The L2SM_NUM_LINES field in bits 63:48 indicates the number of lines to operate on relative to the starting address. The actual operation to be performed is programmed into the L2SM_COP register as described in the previous subsection.

3.4.3.12 CPC_CL_STAT_CONF Register (Offset 0x0008)

The hardware initialization operations described in the section entitled L2 Cache Initialization Options require that the L2_HW_INIT_EN bit (24) of this register is set before hardware initialization can proceed.

3.5 L2 Cache Initialization Options

The I6500 Multiprocessing System provides three ways to initialize the L2 cache:

- Automatically selected hardware cache initialization
- Manually selected hardware cache initialization
- Software cache initialization

For hardware initialization, there are two types:

- L2 Tag array only (fast)
- L2 Tag and data arrays (slow)

Automatically selected hardware cache initialization (fast mode) initializes only the L2 tag array. Manually selected hardware cache initialization can initialize either the L2 tag array only (fast mode), or both the tag and data arrays (slow mode). For software initialization by the kernel, one or both arrays can be initialized depending on the design of the software.

Each of these options are described in the following subsections.

3.5.1 Automatic Hardware Cache Initialization

The I6500 MPS allows for the L2 cache to be automatically initialized by hardware when the following conditions are met at reset:

- The external input pin (si_cpc_l2_hw_init_inhibit) is driven low, indicating that automatic hardware initialization can proceed.
- Automatic hardware cache initialization is enabled by setting the L2_HW_INIT_EN bit in the *CPC Local Status and Configuration* register (CPC_CL_STAT_CONF_REG) located at offset 0x0008 in CPC CM-local address space.
- The L2 initialization delay has expired. Once this delay has expired, automatic hardware cache initialization can begin.

- MBIST is not enabled. If it is enabled, the cache initialization does not begin until the MBIST operation is complete. Even if the delay has expired, the cache initialization does not begin until the MBIST has completed.

Once all of these conditions are met, the L2 cache Tag RAM is automatically initialized by hardware. No initialization code is required. Once the initialization is complete, hardware sets the HCI_DONE bit in the *L2 RAM Configuration* register (GCR_L2_RAM_CONFIG) at offset address 0x0240 in GCR address space. Software can poll this bit to determine when the initialization is complete.

3.5.2 Manual Hardware Cache Initialization

The I6500 MPS allows for the L2 cache to be manually initialized by hardware. The user can choose to initialize only the Tag RAM, or both the Tag RAM and Data RAM, when the following conditions are met at reset:

- The external input pin (si_cpc_l2_hw_init_inhibit) is driven high, indicating that automatic hardware initialization described in the previous subsection is not selected and cannot proceed.

For manual cache initialization, kernel software indicates the type of cache initialization to be performed using the following procedure.

1. Read the L2SM_COP_REG_PRESENT bit in the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP) at offset address 0x0620 in GCR address space to determine if this register is present. A '1' in this bit indicates that the flush cache operation is supported.

2. Read the L2SM_COP_MODE bit in the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP) at offset address 0x0620 in GCR address space to determine the state of the L2 state machine. This bit must be 0, indicating the state machine is idle, in order for cache initialization to proceed.

3. Set the type of operation to be performed by programming the L2SM_COP_TYPE field in bits 4:2 of the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP). A value of 0x1 in this field indicates that only the Tag RAM is initialized. A value of 0x2 in this field indicates that both the Tag RAM and Data RAM is initialized. Note that this operation is slower than initializing the Tag RAM only.

4. Start the L2 state machine by setting the L2SM_COP_CMD field in bits 1:0 of the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP) to a value of 0x1. This starts the L2 cache initialization process.

5. To determine the result of the initialization, poll the L2SM_COP_RESULT field in bits 8:6 of the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP). A value of 0x0 indicates the process is still running. A value of 0x1 indicates that the process completed with no errors.

3.5.3 Software Cache Initialization

The I6500 MPS allows for the L2 cache to be manually initialized by software. Note that this type of initialization is much slower than either of the hardware initialization options described above. The code used to perform software cache initialization is shown in the section entitled Initializing the Level 2 Cache.

3.6 L2 Cache Flush, Burst, and Abort

This section describes the L2 cache flush, burst, and abort operations.

3.6.1 L2 Cache Flush

An L2 flush operation can only be initiated by software. To flush the entire L2 cache in one operation, perform the following steps:

1. Read the L2SM_COP_REG_PRESENT bit in the *L2 Cache Op State Machine Config/Control* register (GCR_L2SM_COP) at offset address 0x0620 in GCR address space to determine if this register is present. A '1' in this bit indicates that the flush cache operation is supported.

2. Read the L2SM_COP_MODE bit in the *L2 Cache Op State Machine Config/Control* register to determine the state of the L2 state machine. This bit must be 0, indicating the state machine is idle, in order for flush operation to proceed.

3. Program the L2SM_COP_TYPE field in bits 4:2 of the *L2 Cache Op State Machine Config/Control* register to a value of 0x0. This selects the full cache flush operation.

4. Program the L2SM_COP_CMD field in bits 1:0 of the *L2 Cache Op State Machine Config/Control* register to a value of 0x1. This starts the cache flush operation.

5. To determine the result of the flush operation, poll the L2SM_COP_RESULT field in bit 8:6 of the *L2 Cache Op State Machine Config/Control* register. A value of 0x0 indicates the process is still running. A value of 0x1 indicates that the process completed with no errors.

3.6.2 L2 Cache Burst Operations

The L2 Cache supports the following burst operations (CacheOps):

- Hit_Inv
- Hit_WB_Inv
- Hit_WB

These operations can be requested only by software and can be performed on a range of addresses in the cache. Burst operations can be executed using the following procedure. Note that the number of cache lines requested must be less than or equal to the available cache lines in the cache and also less than 65,536.

1. Program the starting address where the flush operation begins into the L2SM_COP_START_TAG_ADDR field in bits 47:6 of the *GCR L2 Cache Op State Machine Tag Address* register (GCR_L2SM_TAG_ADDR_COP) at offset address 0x0628 in GCR address space.

2. Program the L2SM_COP_NUM_LINES field in bits 63:48 of the *GCR L2 Cache Op State Machine Tag Address* register to indicate the number of lines to be flushed from the starting address defined in step 1.

3. Program the type of operation to be performed on each line using the L2SM_COP_TYPE field in bits 4:2 of the *L2 Cache Op State Machine Config/Control* register. A value of 0x4 in this field indicates Hit Invalidate. A value of 0x5 indicates Hit Writeback Invalidate, and a value of 0x6 indicates Hit Writeback.

4. Read the L2SM_COP_MODE bit in the *L2 Cache Op State Machine Config/Control* register to determine the state of the L2 state machine. This bit must be 0, indicating the state machine is idle, in order for the CacheOp to proceed.

5. If the state machine is idle as determined in step 4, program the L2SM_COP_CMD field in bits 1:0 of the *L2 Cache Op State Machine Config/Control* register to a value of 0x1. This initiates the CacheOp starting from the address defined in step 1 and continuing for the number of lines defined in step 2. The operation to be performed in each of the selected cache lines is defined in step 3.

6. To determine the result of the flush operation, poll the L2SM_COP_RESULT field in bit 8:6 of the *L2 Cache Op State Machine Config/Control* register. A value of 0x0 indicates the process is still running. A value of 0x1 indicates that the process completed with no errors.

3.6.3 Abort Operations

During the automatic hardware initialization process described in the section entitled Automatic Hardware Cache Initialization, no coherent requests are permitted. Even if a coherent request is generated during the initialization procedure, it is not allowed to enter the pipeline until the procedure is complete.

For the manual hardware initialization procedure described in the section entitled Manual Hardware Cache Initialization, coherent requests can be generated during this time but are not allowed. It is up to software to mange the flow of these requests during the initialization process. This is also true for the Flush operations described in the section entitled L2 Cache Flush, and cache burst operations described in the section entitled L2 Cache Burst Operations.

3.7 Cache Initialization Routines

The cache must be initialized during power-up or reset to place the lines of the cache in a known state. This is accomplished via the boot code (or, for the L2, by hardware as described in the previous section). This section provides individual routines for initializing the L1 instruction, L1 data, and L2 caches.

A sample boot code is shown in the following subsections. This code is designed to be portable to microprocessors that implement the MIPS ISA, and provides subroutines such as decoding the cache sizes and configuration parameters from the CP0 registers are included.

3.7.1 Initializing the Instruction Cache

The Instruction cache can be initialized using either the a software invalidation routine, or the GINVI instruction. The GINVI instruction fully invalidates all remote primary instruction caches, or a specified single cache, whether local or remote. The local primary instruction cache is also fully invalidated in the case where all remote caches are to be invalidated. Which cache is invalidated depends on the rs field of the instruction.

3.7.1.1 L1 Instruction Cache Invalidation Using the GINVI Instruction

The GINVI instruction is new in the R6 architecture and is used to manage L1 instruction cache invalidation across all cores in the system, including single-cluster and multi-cluster systems. The GINVI instruction can be used in one of two ways:

- Invalidate all L1 instruction cache entries in all cores and all clusters simultaneously
- Invalidate all entries in a specific L1 instruction cache of any core in any cluster

The rs field in bits 20:16 of the GINVI instruction is a pointer to one of 32 general purpose registers (GPR) in the core. If the *rs* field is 0, then all L1 instruction caches of all cores in all clusters are to be invalidated. In this case, based on the value in the rs field, the core executing the GINVI instruction invalidates all entries of its own L1

instruction cache. In addition, the core sends a request to the CM, which in turn broadcasts the request to all other cores and clusters, instructing them to invalidate their own L1 instruction caches.

Individual L1 instruction caches can also be invalidated using the GINVI instruction. If the *rs* field is a non-zero value, the core reads the value of the GPR identified by the*rs* field and compares that register value to the value in the *CP0 Global Number* register, which contains the corresponding cluster number and core number. For example, assume that core 0 executes the GINVI instruction, and the *rs* field of the instruction contains a value of 5. Hardware would then read the GPR 5 register and compare the contents to its own *CP0 Global Number* register. If there is a match, the L1 instruction cache is invalidated for that core and the operation is complete. Note that if the rs field in the instruction is 0 as described above, the caches are invalidated automatically. No GPR register compare is performed and the *CP0 Global Number* register is not used.

If there is not a match, the core sends the request through the CM to all other cores in the cluster. In a multi-cluster system, the request is also routed by the CM3.5 through the coherent interconnect block used to communicate with other clusters in the system. Each core compares the value in the request to their own *Global Number* register to determine if it matches their unique cluster and core numbers. If there is a match, the corresponding L1 instruction cache is invalidated. Note that there is one *Global Number* register per core. Therefore, in a 4-core single-cluster system, there are four L1 instruction caches, so the compare of the CM request would be done four times, once per core. All compares are done simultaneously and independently of one another.

The *Global Number* register assigns a number to each VP in a core, each core in a cluster, and each cluster. This allows each processing element throughout the entire system to have a unique ID number down to the VP level. So when the request is sent out by the CM, both the *ClusterNum* and *CoreNum* fields of the *Global Number* register are compared to determine the exact L1 instruction cache to be invalidated. Note that it is not required for the GINVI instruction to operate at the VP level because the L1 instruction cache is shared between all VP's in a core. As such, the *VPID* field of the *Global Number* register is not used during the compare.

3.7.1.2 L1 Cache Initialization Routine

This section provides the instruction cache initialization routine.

```
LEAF(init_icache)

    // Can be skipped if Config7.HCI is set (Hardware Cache Initialization)
    mfc0   TEMP1, C0_CONFIG, 7        // Read CP0 Config7
    ext    TEMP1, TEMP1, HCI, 1       // extract HCI
    bne    TEMP1, zero, done_icache
    nop

    // Determine how big the I$ is:
    mfc0   CONFIG1_a2, C0_CONFIG1 // read C0_Config1

    // Set line size
    addiu  LINE_SIZE_v1, zero, ILINE_SIZE

    // Since the line size and associativity are fixed values
    // the number of sets in the cache is what determines the size of the cache
    // Here the set size is determined from the value in the C0_CONFIG register
    ext    SET_SIZE_a0, CONFIG1_a2, CFG1_ISSHIFT, 3    // extract IS
    li     TEMP1, 64
    sllv   SET_SIZE_a0, TEMP1, SET_SIZE_a0             // I$ Sets per way

    // Set associativity (number of cache ways)
    addiu  ASSOC_a1, zero, IASSOC

    li  TEMP1, (LINES_PER_ITER)
```

```
    dmul   SET_SIZE_a0, SET_SIZE_a0, ASSOC_a1      // Total number of sets
    dmul   TOTAL_BYTES, SET_SIZE_a0, LINE_SIZE_v1  // Total number of bytes
    dmul   BYTES_PER_LOOP_v0, LINE_SIZE_v1, TEMP1  // Total bytes per loop

    // Set the starting address at the beginning of kgeg0 (0x80000000) which
    // corresponds to way 0 index 0 of the cache
    dli    CURRENT_ADDR, 0x0000000080000000
    dsrl   TEMP1, BYTES_PER_LOOP_v0, 1
    daddu  CURRENT_ADDR, TEMP1, CURRENT_ADDR
    daddu  END_ADDR_a3, CURRENT_ADDR, TOTAL_BYTES         // make ending address
    dsubu  END_ADDR_a3, END_ADDR_a3, BYTES_PER_LOOP_v0    // -1

    // Clear TagLo/TagHi registers
    mtc0   zero, C0_ITAGLO                  // write C0_ITagLo

next_icache_tag:
    // To be more efficient this loop does 8 cache lines at a time
    // Index Store Tag Cache Op invalidates the tag entry, clears the
    // lock bit, and clears the LRF bit
     cache 0x8, (ILINE_SIZE*-2)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*-1)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*0)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*1)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*-4)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*-3)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*2)(CURRENT_ADDR)
     cache 0x8, (ILINE_SIZE*3)(CURRENT_ADDR)
    daddu     CURRENT_ADDR, BYTES_PER_LOOP_v0// Get next starting line address
    bgeuc     END_ADDR_a3,  CURRENT_ADDR, next_icache_tag// Done yet?
    nop    // needed for MIPS64 R6 forbidden slot (following instruction is jalr)

done_icache:

     jalr  zero, ra
     nop
END(init_icache)
```

3.7.2 Initializing the Data Cache

This section provides the data cache initialization routine.

```
#include <mips/regdef.h>// #defines for GPRs
#include <mips/m32c0.h>// #defines for CP0 registers
#include <core_config.h>// #defines for ILINE_SIZE, DLINE_SIZE and HCI

#define LINE_SIZE_v1       v1
#define BYTES_PER_LOOP_v0  v0
#define SET_SIZE_a0        a0
#define ASSOC_a1           a1
#define CONFIG1_a2         a2
#define END_ADDR_a3        a3
#define TOTAL_BYTES        t0
#define CURRENT_ADDR       t1
#define TEMP1              t2
#define TEMP2              t3
```

```
#define LINES_PER_ITER 8 // number of cache instructions per loop

    .set   noreorder    // Don't allow the assembler to reorder instructions.
    .set   noat      // Don't allow the assembler to use r1(at) for synthetic instr.


LEAF(init_dcache)

    // Can be skipped if Config7[HCI] set (Hardware Cache Initialization)
    mfc0   TEMP1, C0_CONFIG, 7           // Read CP0 Config7
    ext    TEMP1, TEMP1, HCI, 1          // extract HCI
    bne    TEMP1, zero, done_dcache
    nop

    mfc0   CONFIG1_a2, C0_CONFIG1        // read C0_Config1

    // Set line size
    addiu  LINE_SIZE_v1, zero, DLINE_SIZE

    // Since the line size and associativity are fixed values the number of sets in
    // the cache is what determines the size of the cache. Here the set size is
    // determined from the value in the C0_CONFIG register

    ext    SET_SIZE_a0, CONFIG1_a2, CFG1_DSSHIFT, 3       // extract DS
    li     TEMP1, 64
    sllv   SET_SIZE_a0, TEMP1, SET_SIZE_a0                // D$ Sets per way

    // Set associativity (number of cache ways)
    addiu  ASSOC_a1, zero, DASSOC

    li  TEMP1, (LINES_PER_ITER)

    dmul   SET_SIZE_a0, SET_SIZE_a0, ASSOC_a1      // Total number of sets
    dmul   TOTAL_BYTES, SET_SIZE_a0, LINE_SIZE_v1  // Total number of bytes
    dmul   BYTES_PER_LOOP_v0, LINE_SIZE_v1, TEMP1  // Total bytes per loop

    // Set the starting address at the beginning of kgeg0 (0x80000000) which
    // corresponds to way 0 index 0 of the cache
    lui    CURRENT_ADDR, 0x8000
    srl    TEMP1, BYTES_PER_LOOP_v0, 1
    addu   CURRENT_ADDR, TEMP1, CURRENT_ADDR

    addu   END_ADDR_a3, CURRENT_ADDR, TOTAL_BYTES         // make ending address
    subu   END_ADDR_a3, END_ADDR_a3, BYTES_PER_LOOP_v0    // -1

    // Clear TagLo/TagHi registers
    mtc0   zero, C0_TAGLO, 2      // write C0_DTagLo

    // due to offset field restrictions, the code assumes the line size is not
    // more than 64 bytes

next_dcache_tag:
    // Index Store Tag Cache Op
    // Invalidates the tag entry, clears the lock bit, and clears the LRF bit
    cache 0x9, (DLINE_SIZE*-2)(CURRENT_ADDR)
     cache 0x9, (DLINE_SIZE*-1)(CURRENT_ADDR)
     cache 0x9, (DLINE_SIZE*0)(CURRENT_ADDR)
     cache 0x9, (DLINE_SIZE*1)(CURRENT_ADDR)
```

```
    cache 0x9, (DLINE_SIZE*-4)(CURRENT_ADDR)
    cache 0x9, (DLINE_SIZE*-3)(CURRENT_ADDR)
    cache 0x9, (DLINE_SIZE*2)(CURRENT_ADDR)
    cache 0x9, (DLINE_SIZE*3)(CURRENT_ADDR)
   daddu    CURRENT_ADDR, BYTES_PER_LOOP_v0// Get next starting line address
   bgeuc    END_ADDR_a3,  CURRENT_ADDR, next_dcache_tag // Done yet?
   nop       // needed for MIPS64 R6 forbidden slot (following instruction is jalr)

done_dcache:

    jalr  zero,      ra
    nop
END(init_dcache)
```

3.7.3 Initializing the Level 2 Cache

This section provides the L2 cache initialization routine. This routine is only used during the software initialization procedure. If either automatic or manual hardware initialization is invoked as described in the section entitled L2 Cache Initialization Options, then this routine is not used.

```
#include <mips/regdef.h>// #defines for GPRs
#include <mips/m32c0.h>// #defines for CP0 registers
#include <core_config.h>// #defines for ILINE_SIZE, DLINE_SIZE and HCI

#define LINE_SIZE_v1       v1
#define BYTES_PER_LOOP_v0  v0
#define SET_SIZE_a0        a0
#define ASSOC_a1           a1
#define CONFIG1_a2         a2
#define END_ADDR_a3        a3
#define TOTAL_BYTES        t0
#define CURRENT_ADDR       t1
#define TEMP1              t2
#define TEMP2              t3

#define LINES_PER_ITER 8 // number of cache instructions per loop

   .set   noreorder    // Don't allow the assembler to reorder instructions.
   .set   noat      // Don't allow the assembler to use r1(at) for synthetic instr.


LEAF(init_L2)

   bnez   r8_core_num, done_L2_cach_init// Only done from core 0.

   // Read L2 Configuration register
   ld  CONFIG_L2_a2, GCR_L2_CONFIG(r22_gcr_addr)
   dli    TEMP_s1, 0x1     // set to uncached (bypass)
   dins   CONFIG_L2_a2, TEMP_s1, 20, 1         // Insert bits
   sd  CONFIG_L2_a2, GCR_L2_CONFIG(r22_gcr_addr)// Write L2 Configuration register
   // Read back the L2 Configuration register
   ld  CONFIG_L2_a2, GCR_L2_CONFIG(r22_gcr_addr)

   // Isolate L2$ Line Size
   dext   LINE_SIZE_v1, CONFIG_L2_a2, 8, 4        // extract LINE_SIZE

   // Skip ahead if No L2$
```

```
    beq    LINE_SIZE_v1, zero, done_l2
    nop

    dli    TEMP_s1, 2
    dsllv  LINE_SIZE_v1, TEMP_s1, LINE_SIZE_v1
    // decode for true L2$ line size in bytes

    // Isolate L2$ Sets per Way
    dext   SET_SIZE_a0, CONFIG_L2_a2, 12, 4// extract SET_SIZE_a0
    dli    TEMP_s1, 64
    dsllv  SET_SIZE_a0, TEMP_s1, SET_SIZE_a0// decode for sets per way

    // Isolate L2$ Associativity
    dext   ASSOC_a1, CONFIG_L2_a2, 0, 4// extract ASSOC_a1
    daddiu ASSOC_a1,ASSOC_a1, 1// decode for # of ways

    dli    TEMP_s1, (LINES_PER_ITER)

    dmul   SET_SIZE_a0, SET_SIZE_a0, ASSOC_a1      // Get total number of sets in L2
    dmul   TOTAL_BYTES, SET_SIZE_a0, LINE_SIZE_v1  // Total number of bytes
    dmul   BYTES_PER_LOOP_v0, LINE_SIZE_v1, TEMP_s1 // Total bytes per loop

    dli    CURRENT_ADDR, 0x80000000// load a KSeg0 address for cacheops
    daddu  END_ADDR_a3, CURRENT_ADDR, TOTAL_BYTES  // make ending address
    dsubu  END_ADDR_a3, END_ADDR_a3, BYTES_PER_LOOP_v0 // -1 bytes per loop

    // Clear L2 Tag registers
    sd  zero, 0x600(r22_gcr_addr)     // GCR_L2_TAG_ADDR
    sd  zero, 0x608(r22_gcr_addr)     // GCR_L2_TAG_STATE
    sd  zero, 0x610(r22_gcr_addr)     // GCR_L2_DATA

    // L2$ Index Store Tag Cache Op. Invalidates the tag entry.

next_L2_cache_tag:

    // Index Store Tag Cache Op
    // Invalidate the tag entry, clear the lock bit, and clear the LRF bit
    cache 0xB, (L2LINE_SIZE*-2)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*-1)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*0)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*1)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*-4)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*-3)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*2)(CURRENT_ADDR)
    cache 0xB, (L2LINE_SIZE*3)(CURRENT_ADDR)
    bne CURRENT_ADDR, END_ADDR_a3, next_L2_cache_tag // Done yet?
    daddu CURRENT_ADDR, BYTES_PER_LOOP_v0      // Get next starting line address

done_L2_cach_init:

// Clear L2 ByPass (enable L2)
ld     a0, GCR_L2_CONFIG(r22_gcr_addr)      // Read L2 Configuration register
dins   a0, zero, 20, 1                      // Insert bits
sd     a0, GCR_L2_CONFIG(r22_gcr_addr)      // Write L2 Configuration register

done_l2:

jalr   zero,      ra
nop
```

```
END(init_L2)
```

3.8 Flushing the L1 Data Cache

The I6500 L1 D-Cache uses a write-back policy, which means that the D-Cache may contain the only copy of data stored by the core. In some situations, software may need to force modified data to be written back from the L1 D-Cache to the L2-Cache (e.g. before powering down the core or when interacting with non-coherent DMA). This section describes the routine for writing back and invalidating all data from the L1 D-Cache. Note that this routine should not be executed until after the data cache has been initialized.

```
LEAF(flush_dcache)

    mfc0   CONFIG1_a2, C0_CONFIG1  // read C0_Config1

    // Isolate D$ Line Size
    ext    LINE_SIZE_v1, CONFIG1_a2, CFG1_DLSHIFT, 3         // extract DL

    // Skip ahead if No D$
    beq    LINE_SIZE_v1, zero, done_flush_dcache
    nop

    li     TEMP1, 2
    sllv   LINE_SIZE_v1, TEMP1, LINE_SIZE_v1// Now have true D$ line size in bytes

    ext    SET_SIZE_a0, CONFIG1_a2, CFG1_DSSHIFT, 3    // extract DS
    li     TEMP1, 64
    sllv   SET_SIZE_a0, TEMP1, SET_SIZE_a0             // D$ Sets per way

    // Config1DA == D$ Assoc - 1
    ext    ASSOC_a1, CONFIG1_a2, CFG1_DASHIFT, 3       // extract DA
    addiu  ASSOC_a1, 1

    li     TEMP1, (LINES_PER_ITER)

    dmul   SET_SIZE_a0, SET_SIZE_a0, ASSOC_a1             // Total number of sets
    dmul   TOTAL_BYTES, SET_SIZE_a0, LINE_SIZE_v1         // Total number of bytes
    dmul   BYTES_PER_LOOP_v0, LINE_SIZE_v1, TEMP1         // Total bytes per loop
    lui    CURRENT_ADDR, 0x8000                 // Get a KSeg0 address for cacheops
    srl    TEMP1, BYTES_PER_LOOP_v0, 1
    addu   CURRENT_ADDR, TEMP1, CURRENT_ADDR

    addu   END_ADDR_a3, CURRENT_ADDR, TOTAL_BYTES         // make ending address
    subu   END_ADDR_a3, END_ADDR_a3, LINE_SIZE_v1         // -1

fnext_dcache_tag:

    // Index writeback invalidate Cache Op
    // Writes any modified data back to memory, invalidates the tag entry, clears
    // the lock bit, and clears the LRU bit
    cache 0x1, (DLINE_SIZE*-2)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*-1)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*0)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*1)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*-4)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*-3)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*2)(CURRENT_ADDR)
    cache 0x1, (DLINE_SIZE*3)(CURRENT_ADDR)
```

```
    daddu    CURRENT_ADDR, BYTES_PER_LOOP_v0       // Get next starting line address
    bgeuc    END_ADDR_a3,  CURRENT_ADDR, fnext_dcache_tag // Done yet?
    nop       // needed for MIPS64 R6 forbidden slot (following instruction is jalr)

done_flush_dcache:
    sync
    jalr   zero,      ra
    nop
    END(flush_dcache)
```

3.9 Setting the KSEG0 Memory Space Cache Coherency

The Cache Coherency attribute for a mapped address is set by the TLB entry for that address. If the address resides in the KSGE0 memory range, the CCA is set in the Config.K0 field. The following code shows how this is done.

Note that the code that does the modification of the CCA for KSEG0 cannot be executed from a KSGE0 address. Rather, it must be done in KSEG1 or an uncached address (not KSGE0 uncached). For the I6500 the CCA is set to coherent because all cached access for the I6500 are coherent.

```
#define   C0_CONFIG    $16,0
LEAF(change_k0_cca)

// NOTE! This code must be executed in KSEG1 (not KSGE0 uncached). Set CCA for
kseg0 to cacheable

mfc0      t1, C0_CONFIG       // read C0_Config0
li        t2, 5               // CCA for coherent
ins       t1, t2, 0, 3        // instert K0
mtc0      t1, C0_CONFIG       // write C0_Config
jalr.hb   zero, ra
nop

END(change_k0_cca)
```

Chapter 4

Exceptions

An exception is defined as any event that causes the core to halt normal execution and branch to a dedicated kernel software routine called an exception handler. The exception handler is responsible for determining and then resolving the exception.

Exception events can occur within the core, which are known as internal events, or external to the core, which are known as external events. Internal events include arithmetic overflows, traps, watch address match, reserved instructions, misses in the translation lookaside buffer (TLB), etc. A complete list of exceptions is shown in Table 4.5.

An external event is known as an interrupt. These are generated by asserting dedicated hardware interrupt pins. When a pin is asserted, an exception is taken. The kernel software then halts execution of the current instruction stream and branches to the interrupt handler to determine and resolve the interrupt. The MIPS architecture provides three types of hardware interrupt modes as described in the section entitled Overview of Exception Processing.

This chapter provides an overview of exception processing and a definition of the interrupts modes. Information on how to program the reset, boot, and general exception vectors in memory is also covered. A list of exception priorities is provided, along with an assembly language example of an exception handler.

4.1 Overview of Exception Processing

The I6500 core includes support for three interrupt modes:

- Interrupt Compatibility mode, in which the behavior of the I6500 core is identical to the behavior of an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt.
- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. Note that the Global Interrupt Controller (GIC) serves as the external interrupt controller when the system is in EIC mode. Refer to the GIC chapter in this manual for more information.

Following reset, the I6500 core defaults to Interrupt Compatibility mode.

Table 4.1 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 4.1 Interrupt Modes

StatusBEV	*CauseIV*	*IntCtlVS*	*Config3VINT*	*Config3VEIC*	**Interrupt Mode**
1	x	x	x	x	Compatibility
x	0	x	x	x	Compatibility

Table 4.1 Interrupt Modes *(continued)*

StatusBEV	*CauseIV*	*IntCtlVS*	*Config3VINT*	*Config3VEIC*	**Interrupt Mode**
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller (EIC)
0	1	≠0	0	0	Cannot occur because *IntCtl* $_{VS}$ cannot be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
"x" denotes don't care					

4.1.1 Exception Types

Exceptions may be precise or imprecise. Precise exceptions are those for which the *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in the delay slot or forbidden slot of a branch (as indicated by the *BD* bit in the *Cause* register), the address of the branch instruction immediately preceding the slot.

Conversely, imprecise exceptions are those for which no return address can be identified. A bus error is an example of an imprecise exception.

4.1.2 Detecting an Exception

When an exception is detected, the core takes the following actions:

- Suspends the normal sequence of instruction execution
- Loads the *Exception Program Counter* (*EPC*) register with the location where execution can restart after the exception has been serviced
- Enters kernel mode
- Forces execution of the software exception handler located at a specific address

Once invoked, the exception handler should save the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so that it can be restored when the exception has been serviced.

4.1.3 Exception Conditions

When a precise exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited. The value in the *EPC* (or *ErrorEPC* for errors or *DEPC* for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in program order.

Imprecise exceptions are taken after the instruction that caused them has completed and potentially after following instructions have completed.

4.2 Defining the Exception Vector Locations

When the processor is powered up or reset, the first instruction is fetched from the boot exception vector (BEV) in memory. Registers in the determine whether the exception vector is mapped to the lower 512 MBytes of physical memory, as in the legacy mode, or anywhere within the 4 GByte physical memory space. Both of these options are described in the following subsections.

The I6500 contains two registers located in CM register space that kernel software can program to set the base address for the reset and boot exception vector locations in memory:

- BEV Base Register (GCR_BEV_BASE)
- VP Local Reset Exception Base Register (GCR_CL_RESET_BASE)

The boot exception vector is stored in a global register and pertains to all VP's in the core. This means that all VP's will access the same BEV during boot-up. Conversely, the reset exception vector is stored in a local register, meaning that each VP can have its own reset vector code in memory.

Control bits in these registers also indicate whether the device maps the exception vector to the lower 512 MBytes of physical memory, or within the lower 4 GByte address range as described in the following subsections.

Both of these registers are described in detail in the CM chapter of the *I6500 Technical Reference* manual.

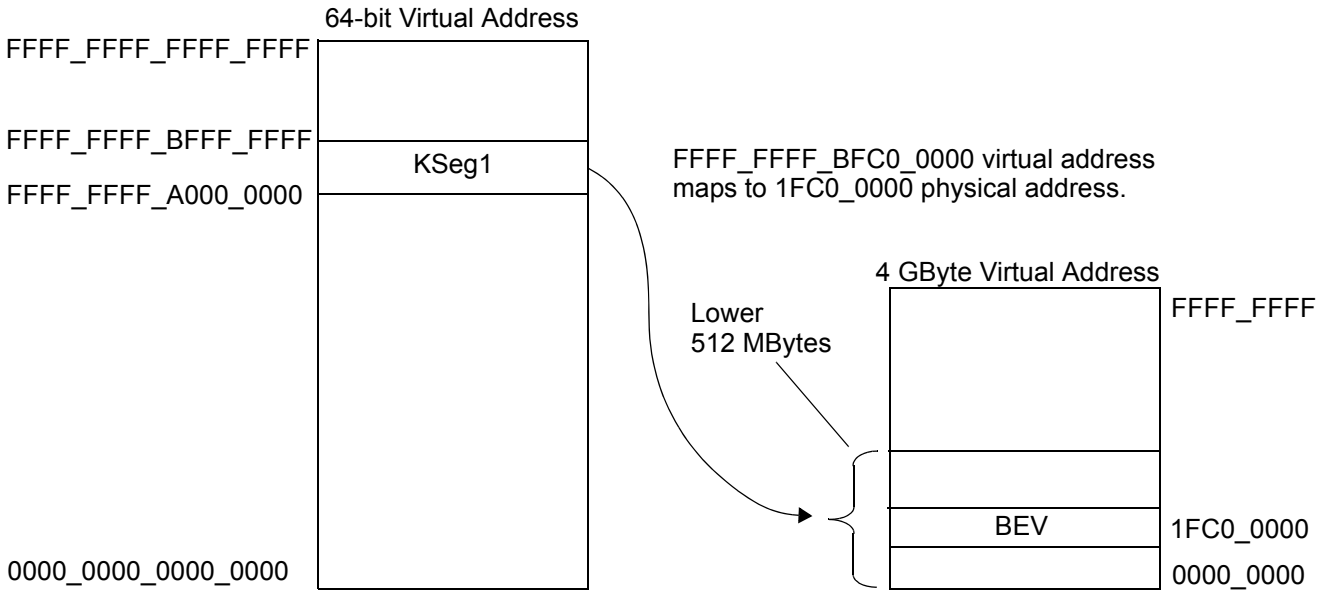
4.2.1 Mapping the BEV to the Lower 512 MBytes of the Physical Address

The boot exception vector is placed in the lower 512 MBytes of physical memory by clearing the BEV_BASE_MODE bit in the GCR_BEV_BASE register located in the CM global register space. When this bit is cleared, bits 31:29 of the BEV_BASE field are forced by hardware to a binary value of 3'b101, causing the BEV to reside in the KSEG 1 address space (always uncached).

The remaining bits 28:12 of the BEV_BASE field are used to place the vector somewhere within the lower 512 MByte space. The hardware configuration default for the field is 0xBFC0_0. This default setting sets the BEV to a virtual address of FFFF_FFFF_BFC0_0000, which directly maps to physical address of 0x0000_0000_1FC0_0000.

This concept is shown in Figure 4.1.

Figure 4.1 Mapping the BEV in the Lower 512 MBytes Using the MIPS Default Address



4.2.2 Mapping the BEV to the Lower 4 GBytes of the Physical Address

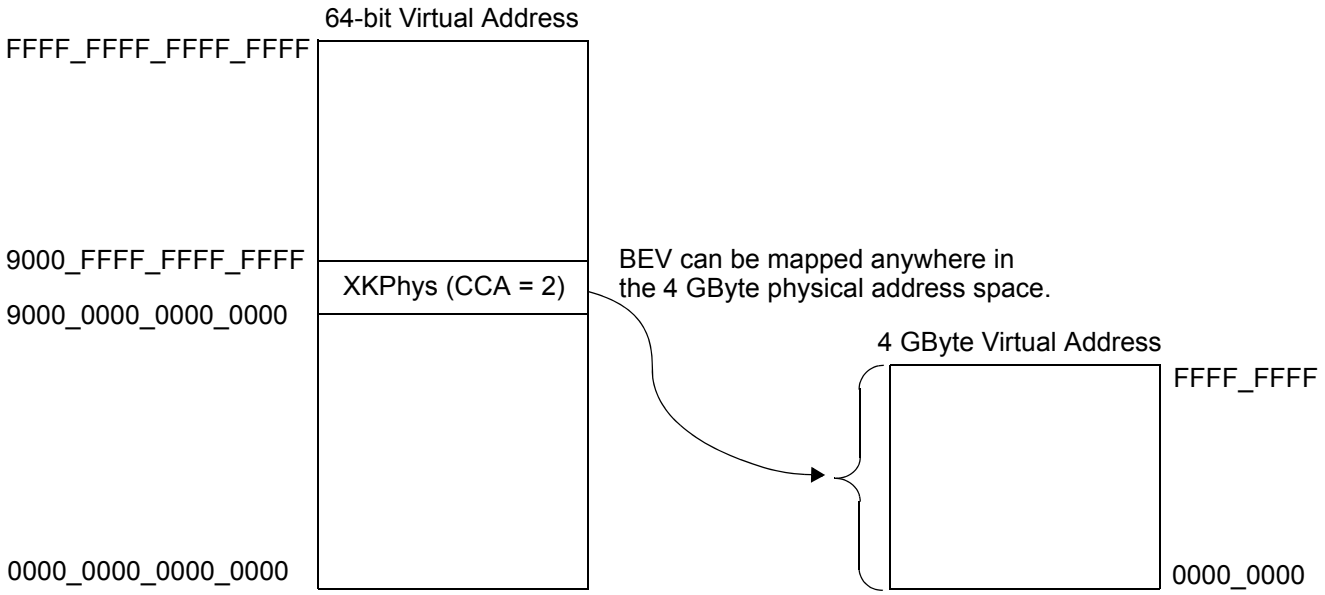
In the case where it is necessary to be able to map the boot exception vector in physical memory to a location outside of the lower 512 MByte range described in the previous subsection, a 4 GByte mapping can be used. In this case the KX bit in the CP0 Status register must be set to enable 64-bit kernel segments.

The boot exception vector can be placed anywhere in the 4 GByte physical address space by setting the BEV_BASE_MODE bit in the GCR_BEV_BASE register located in the CM global register space. When this bit is one, bits 31:12 of the BEV_BASE field are used to map the boot exception vector anywhere within the 4 GByte 32-bit physical address space.

Setting the BEV_BASE_MODE bit maps the 64-bit virtual address to the XKPhys uncached space which starts at virtual address 0x9000_0000_0000_0000. This maps to a physical memory space of (0x0000_0000_0000_0000 - 0x0000_0000_FFFF_FFFF).

This concept is shown in Figure 4.2.

Figure 4.2 Mapping the Boot Exception Vector in 64-bit Mode



4.2.3 Mapping the Reset Vector to the Lower 512 MBytes of the Physical Address

The reset exception vector is mapped in the same manner as the boot exception vector described above. The main difference is that the BEV is global to all VP's in the core, whereas the reset exception vector is local to each VP in the core, meaning that each VP can have its own reset exception vector.

The reset exception vector is placed in the lower 512 MBytes of physical memory by clearing the RESET_BASE_MODE bit in the GCR_CL_RESET_BASE register located in CM Local address space. A logic '0' in this field indicates legacy mode. When this bit is zero, bits 31:29 of the RESET_BASE field are forced by hardware to a binary value of 3'b101, causing the reset exception vector to reside in the KSEG 1 address space (always uncached).

The remaining bits 28:12 of the RESET_BASE field are used to place the reset vector somewhere within the 512 MByte space. The hardware configuration default for the field is BFC0_0. This setting sets the BEV to a virtual address of FFFF_FFFF_BFC0_0000, which directly maps to physical address of 0x0000_0000_1FC0_0000.

This concept is the same as shown in Figure 4.1.

4.2.4 Mapping the Reset Vector to the Lower 4 GBytes of the Physical Address

In the case where it is necessary to be able to map the reset exception vector in physical memory to a location outside of the lower 512 MByte range described in the previous subsection, a 4 GByte mapping can be used. In this case the KX bit in the CP0 Status register must be set to enable 64-bit kernel segments.

The reset exception vector can be placed anywhere in the 4 GByte physical address space by setting the RESET_BASE_MODE bit in the GCR_CL_RESET_BASE register. When this bit is set to one, bits 31:12 of the RESET_BASE field are used to map the boot exception vector anywhere within the 4 GByte 32-bit physical address space. This is different from when RESET_BASE_MODE is 0 as described in the previous subsection, where

RESET_BASE[31:29] are set to a fixed value, forcing the location of the reset vector into the lower 512 MBytes of the 4 GByte physical address space.

Setting the RESET_BASE_MODE bit maps the 64-bit virtual address to the XKPhys uncached space which starts at virtual address 0x9000_0000_0000_0000. This maps to a physical memory space of (0x0000_0000_0000_0000 - 0x0000_0000_FFFF_FFFF).

This concept is the same as shown in Figure 4.2.

4.2.5 Selecting Between the BEV and Reset Exception Vectors

The I6500 core provides a way for the programmer to select which exception vector is used on an exception, the boot exception vector or the reset exception vector. This is accomplished by programming the SELECT_BEV bit (0) in the GCR_CL_RESET_BASE register located at offset 0x0020 in CM GCR local address space.

If the SELECT_BEV bit is 0, then the VP uses the address stored in the GCR_CL_RESET_BASE register to jump to the corresponding Reset exception vector.

If the SELECT_BEV bit is 1, then the VP uses the address stored in the GCR_BEV_BASE register to jump to the global boot exception vector.

4.2.6 Exception Vector Base Address per Exception Type

Table 4.2 Exception Vector Base Addresses

Exception	$Status_{BEV}$	
	0	1
Reset	RESET_BASE + (SELECT_BEV = 0)	
NMI	0xFFFF_FFFF_BFC0.0000	
Debug with *DmxSegEn* = 0 and *DCRDVec* = 0 in the VP_Control1 register.	0xFFFF_FFFF_BFC0.0480	
Debug with *DmxSegEn* = 0 and *RDVec* = 1 in the CP0 VP_Control1 register.	*DebugVectorAddr*[31:7] || 7'b0000000	
Debug with *DmxSegEn* = 1 and *Probetrap* = 1 in the VP_Control1 register.	0xFFFF_FFFF_FF20.0200	
Cache Error	$EBase_{63..30}$ || 1 || $EBase_{28..12}$ || 0x100	BEV_BASE + 0300
Other	$EBase_{63..12}$ || 0x000 Note that $EBase_{31..30}$ has the fixed value of 2'b10.	BEV_BASE + 0200
'||' denotes bit string concatenation		

Table 4.3 shows the offsets from the vector base address as a function of the exception. Note that the IV bit in the *Cause* register causes interrupts to use a dedicated exception vector offset, rather than the general exception vector.

Table 4.4 combines these three tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, it is assumed that $IntCtl_{VS} = 0$.

Table 4.3 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	0x000
General Exception	0x180
Interrupt, $Cause_{IV} = 1$	0x200
Reset, NMI	None (uses either RESET_BASE or BEV_BASE)

Table 4.4 Exception Vectors

Exception	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	DmxSegEn	ProbeTrap	Vector ($IntCtl_{VS} = 0$)
Reset						RESET_BASE[1]
NMI	x	x	x	x	x	0xFFFF_FFFF_BFC0_0000
Debug	x	x	x	0	x	0xFFFF_FFFF_BFC0_0480 (if *VP_Control1.RDVec* = 0) *DebugVectorAddr*[31:7] || 7'b0000000 (if *VP_Control1.RDVec* = 1)
Debug	x	x	x	1	1	0xFFFF_FFFF_FF20.0200
TLB Refill	0	0	x	x	x	*EBase*[63:12] || 0x000
TLB Refill	0	1	x	x	x	*EBase*[63:12] || 0x180
TLB Refill	1	0	x	x	x	BEV_BASE[2] + 0x200
TLB Refill	1	1	x	x	x	BEV_BASE + 0x380
Cache Error	0	x	x	x	x	*EBase*[63:30] || 0b1 || *EBase*[28:12] || 0x100
Cache Error	1	x	x	x	x	BEV_BASE + 0300
Interrupt	0	0	0	x	x	*EBase*[63:12] || 0x180
Interrupt	0	0	1	x	x	*EBase*[63:12] || 0x200
Interrupt	1	0	0	x	x	BEV_BASE + 0380
Interrupt	1	0	1	x	x	BEV_BASE + 0400
All others	0	x	x	x	x	*EBase*[63:12] || 0x180
All others	1	x	x	x	x	BEV_BASE + 0380
'x' denotes don't care, '||' denotes bit string concatenation						

1. Derived from *VP Local Reset Exception Base* register in CM3 GCR address space at offset 0x0020. This register is instantiated per-VP, which allows each VP to have its own reset vector.
2. Derived from the global *Boot Exception Vector Base Address* register located in GCR address space at offset 0x0680. This register is instantiated per-core.

4.3 Core-Level Exception Priorities

Table 4.5 contains a list and a brief description of all core level exception conditions. The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is taken. The number of the exception taken is recorded in the *ExcCode* field of the CP0 *Cause* register.

Table 4.5 Priority of Exceptions

Cause.ExcCode Field Encoding		Exception	Description	Mode
Decimal	Hex			
n/a	n/a	Reset	Assertion of SI_Reset signal. In this case the device is reset. No specific register is written when a Reset exception occurs.	Root
n/a	n/a	DSS	Debug Single Step. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers. When a DSS exception occurs, hardware sets the CP0 *Debug.DSS* bit.	Debug
n/a	n/a	DINT	Debug Interrupt. Caused by the assertion of the external *DINT* input, or by setting the appropriate *DINT* bit in the DBU Break register, which is part of the General Interrupt Controller (GIC) register set. Refer to the GIC chapter of this manual for more information. When a DINT exception occurs, hardware sets the CP0 *Debug.DINT* bit. Note that there is one DINT bit per VP.	Debug
n/a	n/a	DDBLImpr	Debug Data Break Load. Imprecise. When a DDBLImpr exception occurs, hardware sets the CP0 *Debug.DDBLImpr* bit.	Debug
n/a	n/a	NMI	Indicates the assertion of the *SI_NMI* signal. When an NMI interrupt occurs, hardware sets the CP0 *Status.NMI* bit.	Root
24	0x18	Machine Check - Lookup	Root, or Root TLB related. This exception occurs during the TLB lookup process and can only occur as part of a guest (second step) address translation, root address translation, and root TLB operation (write, probe) whether for guest or root TLB. It is recommended that the Machine-Check be synchronous. A Machine check exception can have many causes. When this exception occurs during a lookup, the exact cause is encoded by hardware in the CP0 *Root.PageGrain.MCAUSE* field.	Root
			Guest TLB related. This can only occur as part of a guest address translation (first step), and guest TLB operation (write, probe). It is recommended that the Machine-Check be synchronous. A Machine check exception can have many causes. When this exception occurs during a lookup, the exact cause is encoded by hardware in the CP0 *Guest.PageGrain.MCAUSE* field.	Guest
0	0x00	Interrupt	A root-enabled interrupt occurred.	Root

Table 4.5 Priority of Exceptions *(continued)*

Cause.ExcCode Field Encoding		Exception	Description	Mode
Decimal	Hex			
n/a	n/a	Deferred Watch - Root	A Root deferred watch exception, deferred because EXL was a logic '1' when the exception was detected, was asserted after EXL went to '0'. When a deferred WATCH exception occurs in Root mode, hardware sets the WP bit in CP0 *Root.Cause* register. In addition, hardware sets the I, R, or W bits in the CP0 *Root.WatchHi* register depending on whether the exception occurred during a fetch (I), a load (R), or a store (W).	Root
0	0x00	Interrupt	A guest-enabled interrupt occurred.	Guest
n/a	n/a	Deferred Watch - Guest	A Guest deferred watch exception, deferred because EXL was a logic '1' when the exception was detected, was asserted after EXL went to '0'. When a deferred WATCH exception occurs in Guest mode, hardware sets the WP bit in CP0 Guest.Cause register. In addition, hardware sets the I, R, or W bits in the CP0 *Root.WatchHi* register depending in whether the exception occurred during a fetch, a store, or a load.	Guest
n/a	n/a	DIB	An EJTAG Debug Instruction Breakpoint (DIB) condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses. When a DIB exception occurs, hardware writes the DBP bit of the CP0 *Debug* register.	Root
23	0x17	WATCH - Instruction Fetch	A root context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.	Root
			A guest context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.	Guest
4	0x04	AdEL	Instruction fetch address alignment error. A non-word-aligned address was loaded into the PC in the current mode.	Root or Guest
2	0x02	TLBL/XTLBL Refill - instruction fetch or load	Root TLB/XTLB refill - Instruction fetch or load. A Root TLB miss occurred on an instruction fetch or a data load. This can occur due to a Root or Guest translation.	Root
			Guest TLB/XTLB refill - Instruction fetch or data load. A Guest TLB miss occurred on either an instruction fetch or a data load.	Guest
		TLB Invalid - instruction fetch or load	The valid bit was zero in the Root TLB entry mapping the address referenced by an instruction fetch. This can occur due to a Root or Guest translation.	Root
			The valid bit was zero in the guest context TLB entry mapping the address referenced by an instruction fetch.	Guest

Table 4.5 Priority of Exceptions *(continued)*

Cause.ExcCode Field Encoding		Exception	Description	Mode
Decimal	Hex			
3	0x3	TLB Invalid - store	The valid bit was zero in the Root TLB entry mapping the address referenced during a store. This can occur due to a Root or Guest translation.	Root
			The valid bit was zero in the guest context TLB entry mapping the address referenced during a store.	Guest
20	0x14	TLBXI	TLB Execute Inhibit. An instruction fetch matched a valid Root TLB entry which had the XI bit set. This can occur due to a Root or Guest translation.	Root
			An instruction fetch matched a valid Guest TLB entry which had the XI bit set.	Guest
30	0x1E	I-cache Error - instruction fetch	A Cache error occurred on an instruction fetch.	Root
6	0x06	IBE - instruction fetch	A Bus error occurred on an instruction fetch.	Root
n/a	n/a	SDBBP	An EJTAG SDDBP instruction was executed. When this occurs, hardware programs a value of 0x9 into the *DExcCode* field of the CP0 *Debug* register.	Root
8	0x08	Sys (Validity exception)[1]	Execution of SYSCALL instruction.	Root or Guest
9	0x09	Bp (Validity exception)[1]	Execution of BREAK instruction.	Root or Guest
10	0x0A	RI (Validity exception)[1]	Execution of a Reserved Instruction.	Root or Guest
11	0x0B	CpU (Validity exception)[1]	Execution of a coprocessor instruction for a coprocessor that is not enabled. The I6500 core supports the CP0 and CP1 coprocessors.	Root or Guest
			Coprocessor unusable - guest. Access to a coprocessor was permitted by the *Guest.Status.CU1-2* bits, but denied by the setting of the *Root.Status.CU1-2* bits.	Root
21	0x14	MSADis (Validity exception)[1]	MSA Disabled exception - Root.	Root or Guest
			MSA Disabled - guest. Access to the MSA unit was permitted by *Guest.Config5.MSAEn*, but denied by *Root.Config5.MSAEn*.	Root
24	0x18	Machine Check - TLB Operation	Root TLB related. This exception is similar to the higher-priority Machine Check exception listed above, but occurs during the TLB operation rather than during the TLB lookup. This can only occur as part of a Guest or Root address translation, or when a TLBP/TLBWI/TLBGP/TLBGWI is executed in root-mode.	Root
			Guest TLB related. This can only occur as part of a Guest address translation, or when a TLBP/TLBWI instruction is executed in guest-mode.	Guest
15	0x0F	FPE (Execution exception)[2]	Floating Point exception.	Root or Guest

Table 4.5 Priority of Exceptions *(continued)*

Cause.ExcCode Field Encoding		Exception	Description	Mode
Decimal	Hex			
12	0x0C	Ov (Execution exception)[2]	Execution of an arithmetic instruction that overflowed.	Root or Guest
13	0x0D	Tr (Execution exception)[2]	Execution of a trap (when trap condition is true).	Root or Guest
27	0x1B	VzGuest (Execution exception)[2]	Virtualized Guest exception. Note that all of the execution exceptions have the same priority. This encoding encompasses all types of virtualization guest exceptions. The exact type of exception is written to the CP0 *GuestCtl0.GExcCode* field. Refer to Table 4.6 for more information and a listing of VzGuest exception priorities.	Root
n/a	n/a	DDBL / DDBS	Precise Debug Data Address Break. A precise EJTAG data break on load/store (address match only) or a data break on store (address + data match) condition was asserted. Prioritized above data fetch exceptions to allow break on illegal data addresses. When this exception occurs, hardware sets the CP0 *Debug.DDBL* bit if the error occurred during a load, or the *Debug.DDBS* bit if the error occurred during a store.	Root
23	0x17	WATCH - data access	A root context watch address match was detected on the address referenced by a load or store.	Root
			A guest context watch address match was detected on the address referenced by a load or store.	Root
4	0x04	AdEL - Data Access	Load address alignment error. An unaligned address, or an address that was inaccessible in the current processor mode was referenced by a load instruction.	Root or Guest
5	0x05	AdES - Data Access	Store address alignment error. An unaligned address, or an address that was inaccessible in the current processor mode was referenced by a store instruction.	Root or Guest
2	0x02	TLBL/XTLBL refill - data access	Load TLB miss. A root TLB miss occurred on a data access. This can occur due to a Root or Guest translation.	Root
			A guest TLB miss occurred on a data access.	Root or Guest
3	0x03	TLBS	Store TLB miss in Root or Guest mode.	Root or Guest
2	0x2	TLB Invalid - data load	On a data load, a matching root TLB entry was found, but the valid (V) bit was zero. This can occur due to a Root or Guest translation.	Root
			On a data load, a matching guest TLB entry was found, but the valid (V) bit was zero.	Guest
3	0x3	TLB Invalid - data store	On a data store, a matching root TLB entry was found, but the valid (V) bit was zero. This can occur due to a Root or Guest translation.	Root
			On a data store, a matching guest TLB entry was found, but the valid (V) bit was zero.	Guest

Table 4.5 Priority of Exceptions *(continued)*

Cause.ExcCode Field Encoding		Exception	Description	Mode
Decimal	Hex			
19	0x13	TLBRI	TLB Read Inhibit. On a data read access, a matching root TLB entry was found, and the RI bit was set. This can occur due to a Root or Guest translation.	Root
			On a data read access, a matching guest TLB entry was found, and the RI bit was set.	Guest
1	0x01	TLB Modified	The dirty bit was zero in the root TLB entry mapping the address referenced by a store instruction.	Root
			The dirty bit was zero in the guest TLB entry mapping the address referenced by a store instruction.	Guest
30	0x1E	Dcache Error - data access	A cache error occurred on a load or store data reference.	Root
7	0x07	DBE - Data bus error	Load or store bus error. Imprecise.	Root
n/a	n/a	DDBL	Precise Debug Data Address Break. A precise EJTAG data break on load (address + data match only) condition was asserted. Prioritized last because all aspects of the data access must complete in order to do a data value match.	Root

1. All of the Instruction Validity exceptions have the same priority level.
2. All of the Execution exceptions have the same priority level.

4.4 Hypervisor Exception Priorities

In Table 4.5, the *VZGuest* exceptions entry, which appears as encoding 0x1B (27 decimal) in the CP0 *Cause.ExcCode* field, corresponds to all of the Guest related exceptions described in Table 4.6. When one of the guest-related exceptions in the table is taken, the actual exception type is encoded into the *GuestCtl0.GExcCode* field as shown. In addition, hardware writes a value of 0x1B to the CP0 *Cause.ExcCode* field, indicating a virtualization related exception.

During guest mode execution, control can be returned to root mode at any time. When an exception condition is detected during guest mode execution and the condition requires a switch to root mode, the switch is made before any exception state is saved. As a result, exception state in the guest CP0 context is not affected.

The switch to root mode is achieved by setting $Root.Status_{EXL}$ = 1 or $Root.Status_{ERL}$ = 1 (as appropriate) before any other state is saved. This ensures that all exception states are stored into root CP0 context, regardless of whether the processor was executing in root or guest mode when the exception was detected.

Table 4.6 GuestCtl0 GExcCode Values

Exception code value		Mnemonic	Description
Decimal	Hexadecimal		
0	0x00	GPSI	Guest Privileged Sensitive instruction. This exception is taken when execution of a Guest Privileged Sensitive Instruction was attempted from guest-kernel mode, but the instruction was not enabled for guest-kernel mode. Refer to the CP0 *GuestCtl0* register for more information on enabling access to core functions.
1	0x01	GSFC	Guest Software Field Change event. Note that the MC bit (29) of the CP0 GuestCtl0 register must be set in order for the an interrupt to occur on a software initiated change. If this bit is cleared, software initiated changes are not recognized.
2	0x02	HC	Hypercall
3	0x03	GRR	Guest Reserved Instruction Redirect. A Reserved Instruction or MDMX Unusable exception would be taken in guest mode. When $GuestCtl0_{RI}$ = 1, this root-mode exception is raised before the guest-mode exception can be taken.
4 - 7	0x4 - 0x7	RSV	Reserved.
8	0x08	GVA	Guest mode initiated Root TLB exception has Guest Virtual Address available. Set when a Guest mode initiated TLB translation results in a Root TLB related exception occurring in Root mode and the Guest Physical Address is not available.
9	0x09	GHFC	Guest Hardware Field Change event. Note that the MC bit (29) of the CP0 GuestCtl0 register must be set in order for the an interrupt to occur on a hardware initiated change. If this bit is cleared, hardware initiated changes are not recognized.
10	0x0A	GPA	Guest mode initiated Root TLB exception has Guest Physical Address available. Set when a Guest mode initiated TLB translation results in a Root TLB related exception occurring in Root mode and the Guest Physical Address is available.
11 - 31	0x0B - 0x1F	-	Reserved

4.5 General Exception Processing

With the exception of Reset, NMI, cache error, and Debug exceptions, exceptions have the same basic processing flow:

- If the *EXL* bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution is restarted. The *BD* bit is set appropriately in the *Cause* register. The value loaded into the *EPC* register is dependent on whether the instruction is in a forbidden slot, or the delay slot of a branch, or a jump which has delay slots. Table 4.7 shows the value stored in each of the CP0 PC registers, including *EPC*.

  If the *EXL* bit in the *Status* register is set, the *EPC* register is not loaded and the *BD* bit is not changed in the *Cause* register.

Table 4.7 Value Stored in EPC, ErrorEPC, or DEPC on Exception

In Branch/Jump Delay/Forbidden Slot?	Value stored in EPC/ErrorEPC/DEPC
No	Address of the instruction
Yes	Address of the branch or jump instruction (PC-4)

- The *CE* and *ExcCode* fields of the *Cause* registers are loaded with the values appropriate to the exception. The *CE* field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The *EXL* bit is set in the *Status* register.
- The processor begins executing at the exception vector.

The value loaded into the *EPC* register represents the restart address for the exception and need not be modified by exception handler in the normal case. Kernel software need not look at the *BD* bit in the *Cause* register unless it wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type.

Operation:

```
/* If Status_EXL is 1, all exceptions go through the general exception vector */
/* and neither the EPC nor Cause_BD are modified */
if (expn)
if Status_EXL = 1 then
   vectorOffset ← 0x180
else
   if (DS)
      EPC ← PC - 4
      BD = 1
   else
      EPC ← PC
      BD = 0
   endif
      Cause_BD ← 0
   endif

   /* Compute vector offsets as a function of the type of exception */
   if ExceptionType = TLBRefill then
      vectorOffset ← 0x000
```

```
    elseif (ExceptionType = Interrupt) then
        if (Cause_IV = 0) then
            vectorOffset ← 0x180
        else
            if (Status_BEV = 1) or (IntCtl_VS = 0) then
                vectorOffset ← 0x200
            else
                if Config3_VEIC = 1 then
                    VecNum ← Cause_RIPL
                else
                    VecNum ← VIntPriorityEncoder()
                endif
                vectorOffset ← 0x200 + (VecNum × (IntCtl_VS || 0b00000))
            endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
        endif /* if (Cause_IV = 0) then */
    endif /* elseif (ExceptionType = Interrupt) then */
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
Status_EXL ← 1

/* Calculate the vector base address */
if Status_BEV = 1 then
    vectorBase ← 0xFFFF_FFFF_BFC0.0200
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase_63..30 || (vectorBase_29..0 + vectorOffset_29..0)
                          /* No carry between bits 29 and 30 */
```

4.6 Exception Handling and Servicing Flowcharts

Figure 4.3 and Figure 4.4 contain flowcharts for the following exceptions and guidelines for their handlers:

- General exceptions
- TLB miss exceptions

Exceptions are handled by hardware and then serviced by kernel software. Note that unexpected debug exceptions to the debug exception vector at 0xFFFF_FFFF_BFC0_0200 may be viewed as a reserved instruction since uncontrolled execution of an SDBBP instruction caused the exception. The DERET instruction must be used at return from the debug exception handler, in order to leave debug mode and return to non-debug mode. The DERET instruction returns to the address in the *DEPC* register.

Figure 4.3 General Exception Servicing Guidelines (SW)

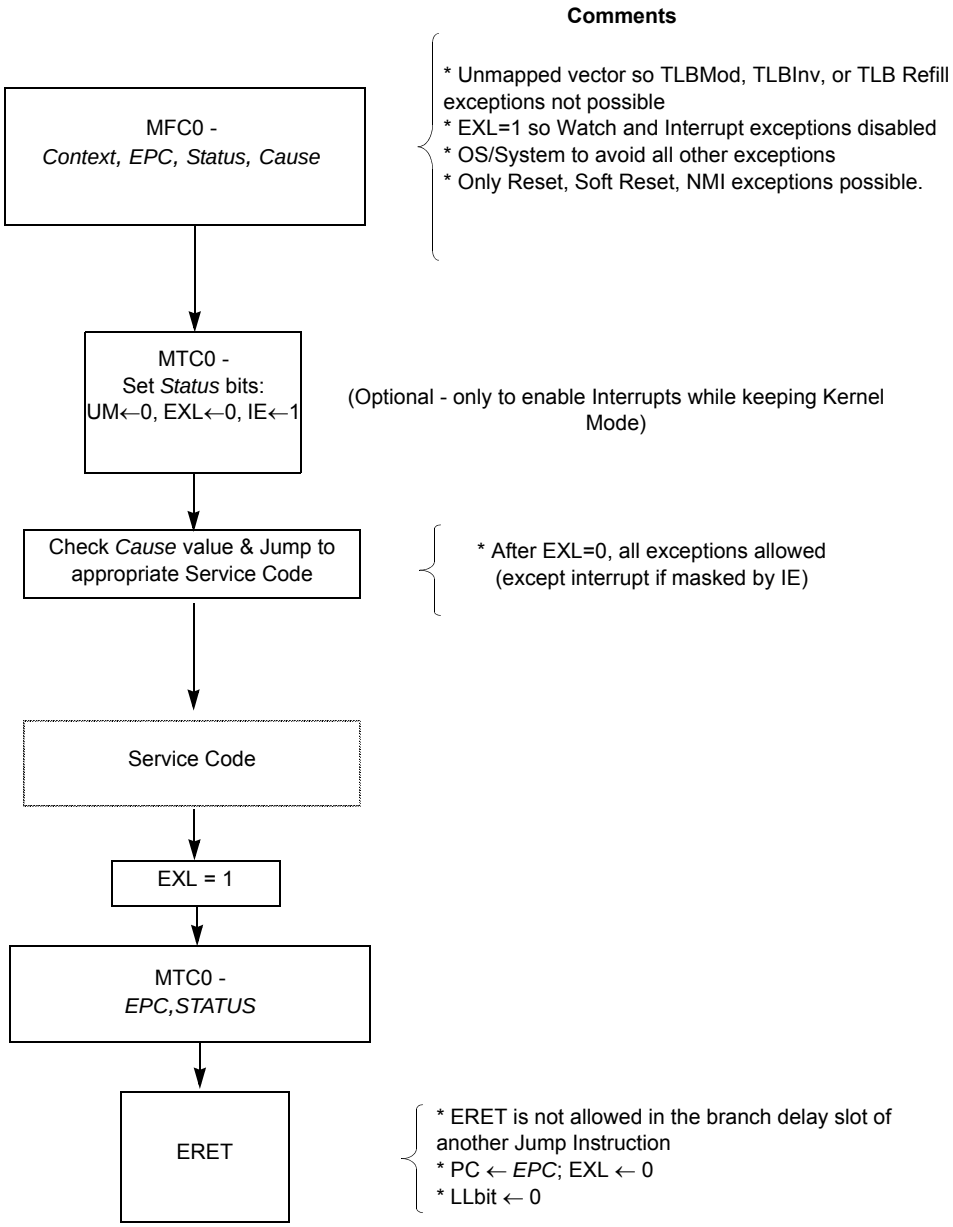
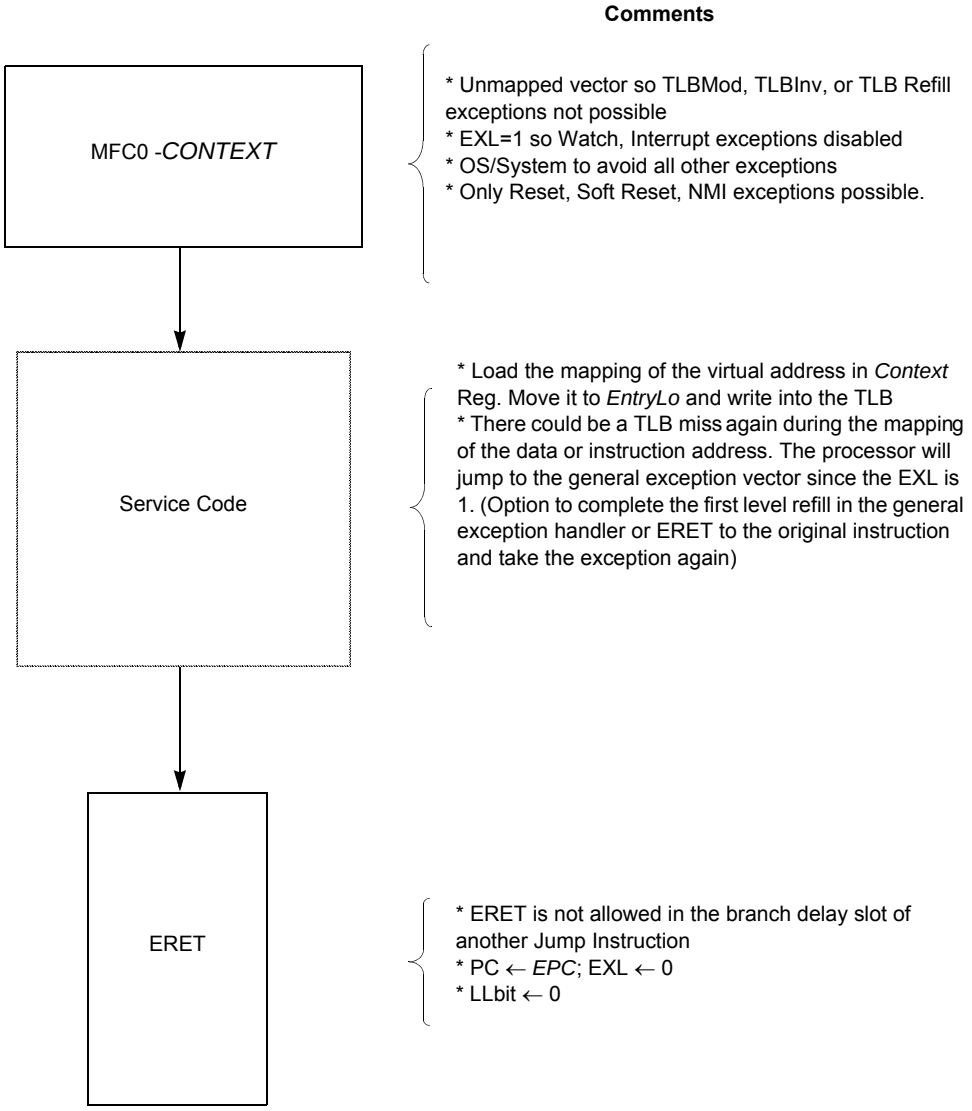


Figure 4.4 TLB Exception Servicing Guidelines (SW)



4.7 Interrupt Mode Code Examples

As described in the section entitled Overview of Exception Processing, the I6500 supports three interrupts modes. The following subsections show how an interrupt handler might look for in each of these modes.

4.7.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 0x180 (if $Cause_{IV} = 0$) or vector offset 0x200 (if $Cause_{IV} = 1$). This mode is in effect when any of the following conditions are true:

- $Cause_{IV} = 0$, or
- $Status_{BEV} = 1$, or
- $IntCtl_{VS} = 0$, which is the case if vectored interrupts are not implemented or have been disabled.

Here is a typical exception handler for compatibility mode:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                 be isolated from the general exception vector before arriving
 *                 here)
 *  - GPRs k0 and k1 are available
 *  - The software priority is IP7..IP0 (HW5..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
   mfc0   k0, C0_CAUSE        /* Read Cause register for IP bits */
   mfc0   k1, C0_STATUS       /* and Status register for IM bits */
   andi   k0, k0, M_CauseIM   /* Keep only IP bits from Cause */
   and    k0, k0, k1          /* and mask with IM bits */
   beq    k0, zero, Dismiss   /* no bits set - spurious interrupt */
   clz    k0, k0              /* Find first bit set, IP7..IP0; k0 = 16..23 */
   xori   k0, k0, 0x17        /* 16..23 => 7..0 */
   sll    k0, k0, VS          /* Shift to emulate software IntCtlVS */
   la     k1, VectorBase      /* Get base of 8 interrupt vectors */
   addu   k0, k0, k1          /* Compute target from base and offset */
   jr     k0                  /* Jump to specific exception routine */
   nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simple UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
```

```
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that "lower" priority interrupts are
 * also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simply return to the interrupted code.
 */
   eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * saving any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below cannot cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Save GPRs here, and setup software context */
   mfc0   k0, C0_EPC             /* Get restart address */
   sw     k0, EPCSave            /* Save in memory */
   mfc0   k0, C0_STATUS          /* Get Status value */
   sw     k0, StatusSave         /* Save in memory */
   li     k1, ~IMbitsToClear     /* Get IM bits to clear for this interrupt */
                                 /*   this must include at least the IM bit */
                                 /*   for the current interrupt, and may include */
                                 /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_STATUS          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

   /*
    * Process interrupt here, including clearing device interrupt.
    * In some environments this may be done with the core running in
    * kernel or user mode. Such an environment is well beyond the scope of
    * this example.
    */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

   di                         /* Disable interrupts - may not be required */
   lw     k0, StatusSave      /* Get saved Status (including EXL set) */
   lw     k1, EPCSave         /*   and EPC */
   mtc0   k0, C0_STATUS       /* Restore the original value */
   mtc0   k1, C0_EPC          /*  and EPC */
```

```
    /* Restore GPRs and software state */
    eret                        /* Dismiss the interrupt */
```

4.7.2 Vectored Interrupt Mode

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. VI mode is in effect when all the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

A typical software handler for Vectored Interrupt mode bypasses the entire sequence of code following the `IVexception` label shown for the compatibility mode handler code example described in the previous subsection. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine.

A nested interrupt is similar to that shown for compatibility mode. Such a routine might look as follows:

```
NestedException:
/*
* Nested exceptions typically require saving the EPC and Status registers,
* disabling the appropriate IM bits in Status to prevent an interrupt loop,
* putting the processor in kernel mode, and re-enabling interrupts. The sample
* code below cannot cover all nuances of this processing and is intended only
* to demonstrate the concepts.
*/
    mfc0   k0, C0_EPC          /* Get restart address */
    sw     k0, EPCSave         /* Save in memory */
    mfc0   k0, C0_STATUS       /* Get Status value */
    sw     k0, StatusSave      /* Save in memory */
    li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                               /*   this must include at least the IM bit */
                               /*   for the current interrupt, and may include */
                               /*   others */
    and    k0, k0, k1             /* Clear bits in copy of Status */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
    mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                  /*   re-enable interrupts */

    /* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

    di                         /* Disable interrupts - may not be required */
    lw     k0, StatusSave      /* Get saved Status (including EXL set) */
    lw     k1, EPCSave         /*   and EPC */
```

```
    mtc0   k0, C0_STATUS       /* Restore the original value */
    mtc0   k1, C0_EPC          /*  and EPC */
    ehb                        /* Clear hazard */
    eret                       /* Dismiss the interrupt */
```

4.7.3 External Interrupt Controller Mode

External Interrupt Controller (EIC) mode redefines the way that the processor interrupt logic is configured in order to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, fast debug channel, and performance counter interrupts, and directly supplying to the processor the vector number of the highest priority interrupt.

EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

The $Config3_{VEIC} = 1$ bit register indicates support for EIC mode. The state of this bit is reflected in the EIC_MODE read-write bit of the GIC VL Control (GIC_VL_CTL) register. This bit can be written by kernel software to enable or disable EIC mode. This is useful for systems that may want to power up in legacy mode, then switch to EIC mode.

In EIC mode, the processor sends the state of the interrupt requests ($Cause_{IP1..IP0}$) and the timer, performance counter, and fast debug channel interrupt requests ($Cause_{TI/PCI/FDCI}$) to the GIC, which prioritizes these interrupts with other hardware interrupts.

A typical exception handler for EIC mode bypasses the entire sequence of code following the `IV exception` label shown for the Compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine.

A nested interrupt is similar to that shown for compatibility mode. It must also copy $Cause_{RIPL}$ to $Status_{IPL}$ to prevent lower priority interrupts from interrupting the handler. Here is an example of such a routine:

```
NestedException:
/*
* Nested exceptions typically require saving the EPC and Status registers,
* disabling the appropriate IM bits in Status to prevent an interrupt loop,
* putting the processor in kernel mode, and re-enabling interrupts.
* The sample code below can not cover all nuances of this processing and is
* intended only to demonstrate the concepts.
*/

    mfc0   k1, C0_CAUSE        /* Read Cause to get RIPL value */
    mfc0   k0, C0_EPC          /* Get restart address */
    srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
    sw     k0, EPCSave         /* Save in memory */
    mfc0   k0, C0_STATUS       /* Get Status value */
    sw     k0, StatusSave      /* Save in memory */
    ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
```

```
   mtc0   k0, C0_STATUS          /* Modify IPL, switch to kernel mode, */
                                 /*   re-enable interrupts */

   /* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

Chapter 5

Coherence Manager

The Coherence Manager (CM) communicates with all cores and other devices in the I6500 Multiprocessing System (MPS), as well as coherent devices external to the I6500 MPS, to achieve system-wide coherence. In a multi-cluster system, the CM also interfaces to an external Network-on-Chip (NOC) controller, which facilitates communication between clusters.

The CM includes an integrated low-latency shared L2 cache. A directory-based coherence protocol is used to efficiently maintain coherence among the L1 data caches of each I6500 core, with up to eight I/O coherence units (IOCUs), providing the I/O subsystem coherent access to the L1 Data and L2 caches.

This chapter provides an overview of the CM register ring bus and associated table that lists each device ID on the bus. The programmer uses this information to access these devices. An overview of the CM register address space is also provided. In addition, the chapter describes how to program the CM to perform various functions, including setting the base addresses in memory, accessing another VP in the same core, accessing a VP in another core, accessing the General Interrupt Controller (GIC), Cluster Power Controller (CPC), and/or Debug Unit (DBU) registers via the CM, and setting the clock ratios between the various I6500 system components. For the exact revision number of the Coherence Manager, refer to the Release Notes.

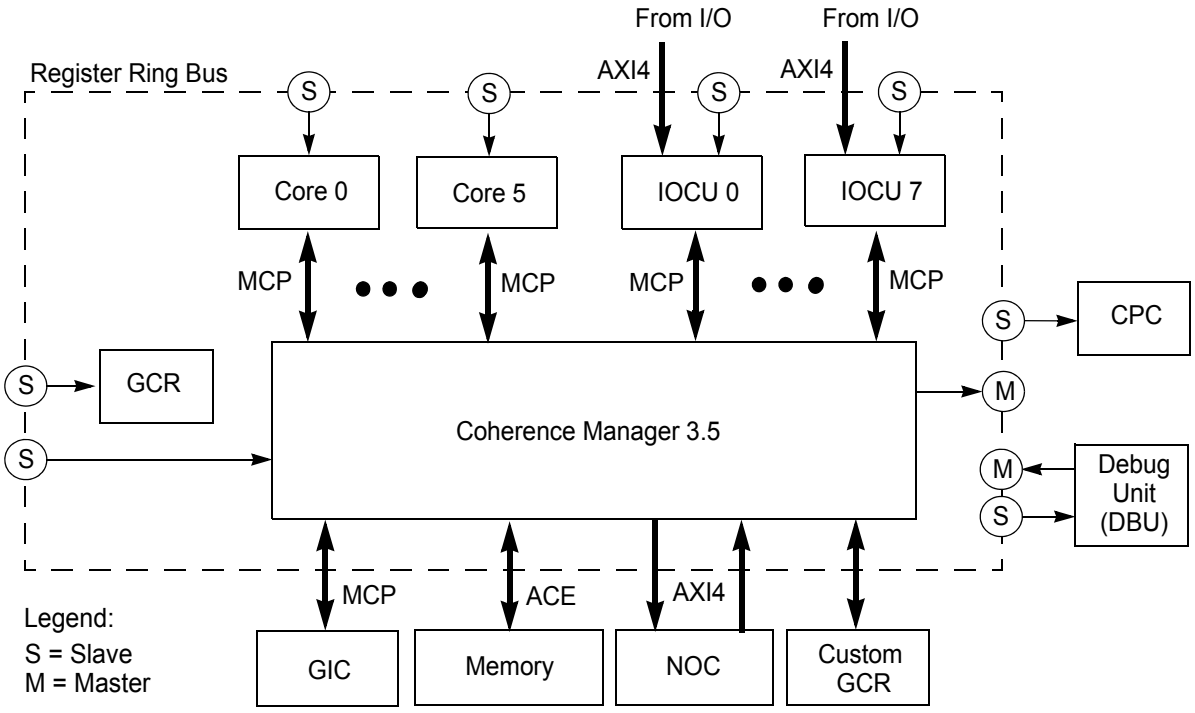
5.1 CM Overview

This section provides an overview of the CM and describes information necessary for programming, including the register ring bus and device ID information, and the CM register map.

5.1.1 CM Interface — Register Ring Bus and Device ID's

The CM communicates with the various system devices via a register ring bus. The devices connected to the CM are shown in Figure 5.1. The I6500 Multiprocessing System can have up to 6 cores per cluster.

Figure 5.1 Interface Ports and Register Ring Bus Interface to the CM



Certain devices such as the cores and IOCU's connect to the CM via an internal proprietary bus called the MIPS Coherence Protocol (MCP) bus. This bus consists of three unidirectional channels used to maximize throughput. The bus implements a credit-based protocol to allow multiple simultaneous in-flight operations. In the above figure, note that the I6500 MPS supports up to a total of eight cores and IOCUs together. For example, if there are four cores, there can only be up to four IOCUs.

The CM accesses the registers of the various devices shown in Figure 5.1 using a register ring bus, indicated by the dotted line. As shown above, the CM and DBU can function as both Master (M) and Slave (S). All other devices, including the cores, are slave devices. Each device on the ring bus is assigned a 6-bit ID value stored in the destination ID (dest_id) or source ID (src_id) fields of the packet being sent. When a device initiates an access to the registers of another device, the corresponding ID is attached to the packet. Only the device whose ID number matches that in the packet accepts the transaction. Table 5.1 lists the ID values for each logic block shown inFigure 5.1. These values are used to write to registers in these blocks as described in the following subsections. All values not shown are reserved.

Table 5.1 Register Ring Bus Device ID Values

dest_id / src_id (Decimal value)	dest_id / src_id (Hexadecimal value)	Device Accessed
0	0x00	Core 0
1	0x01	Core 1
2	0x02	Core 2
3	0x03	Core 3
4	0x04	Core 4
5	0x05	Core 5
16	0x10	IOCU0

Table 5.1 Register Ring Bus Device ID Values *(continued)*

dest_id / src_id (Decimal value)	dest_id / src_id (Hexadecimal value)	Device Accessed
17	0x11	IOCU1
18	0x12	IOCU2
19	0x13	IOCU3
20	0x14	IOCU4
21	0x15	IOCU5
22	0x16	IOCU6
23	0x17	IOCU7
24	0x18	GIC
25	0x19	User Defined GCR's
26	0x1A	Memory
32	0x20	CM
33	0x21	CPC
34	0x22	GCR
35	0x23	DBU Master
36	0x24	DBU dmxseg_normal
37	0x25	DBU dmxseg_debug
40	0x28	AUX 0
41	0x29	AUX 1
42	0x2A	AUX 2
43	0x2B	AUX 3
62	0x3E	No Destination Error
63	0x3F	No Destination OK

The following example shows the path taken in order for core 0 to read a register from the GIC. The data path for this access is shown in Figure 5.2. This figure is similar to Figure 5.1, except only those devices involved in the example transaction are shown. The red color indicates the access request path, and the blue color indicates the data return path. The following sequence is enumerated in Figure 5.2. In this example the following actions would occur.

1. Core 0 sends a request to the CM over the MCP 'Request' bus. Note that Core 0 cannot access the GIC registers directly because it is only a Slave on the ring bus as indicated.

2. The CM processes this request, assigns the appropriate ID number as defined in Table 5.1, and drives this request onto the register ring bus through its Master port.

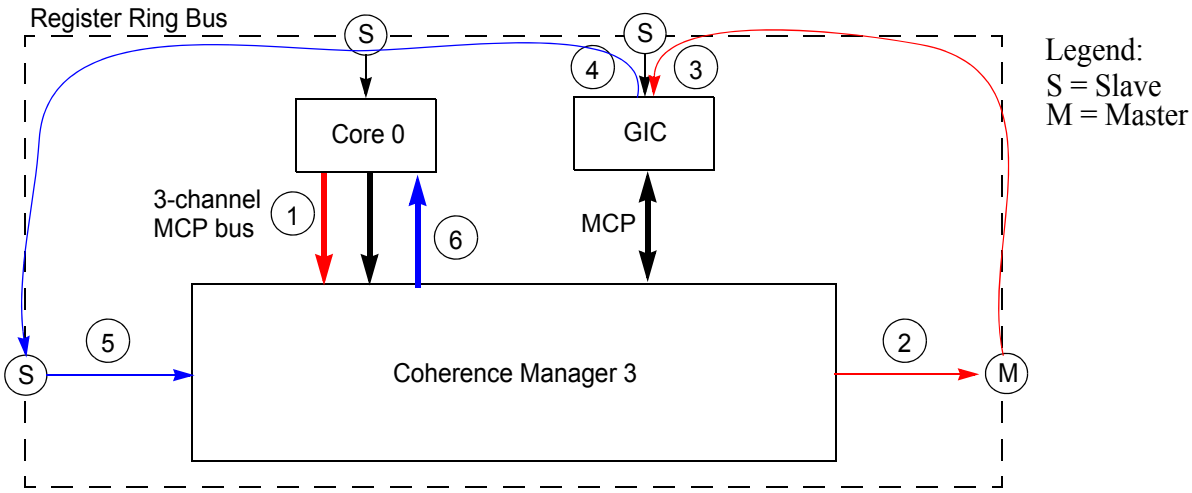
3. The GIC decodes the ID on the bus and gets a match.

4. The GIC then fetches the requested data and drives the data onto the ring bus.

5. Data is returned to the CM through its dedicated register ring bus Slave port.

6. The CM sends the requested data back to Core 0 over the dedicated MCP ‘Response’ bus.

Figure 5.2 Data Path of Core 0 Access of IOCU0 Registers



Refer to the section entitled Core-Local and Core-Other Register Usage for more information on how these ID values are assigned and the programming sequence used when accessing these various devices.

5.1.2 CM GCR Register Map

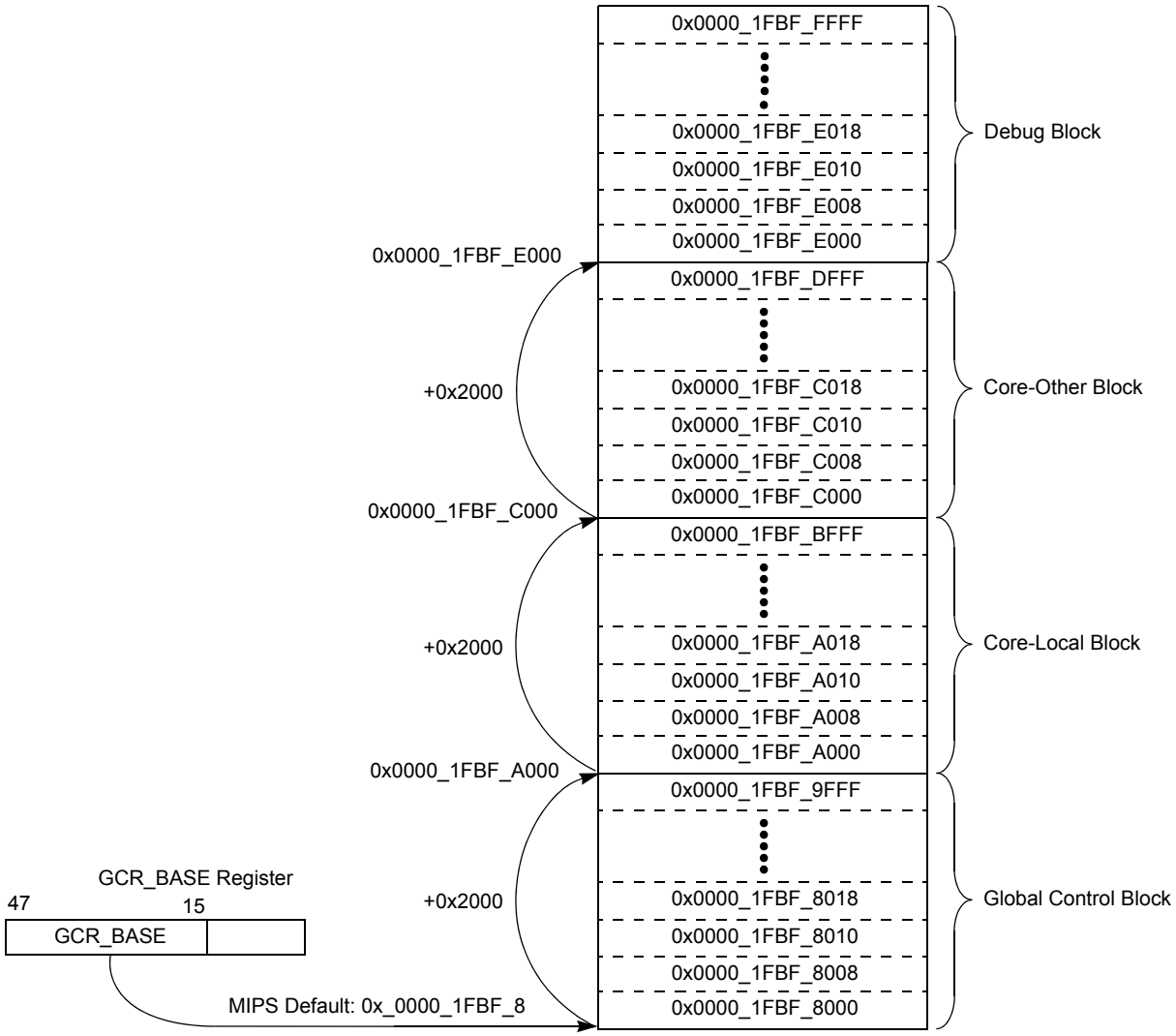
The 32 KB CM GCR register block is divided into four 8 KB subblocks which perform different functions. Table 5.2 shows the address map of the four, 8 KB GCR sub-blocks relative to the *GCR_BASE* as defined in the *GCR Base Register*. This 32 KByte register block can be mapped anywhere in memory on a 32 KByte boundary. The Address Range column shows bits 47:15. Bits 14:0 are always zero so as to align on a 32 KB boundary.

Table 5.2 I6500 Control Space Address Map (Relative to GCR_BASE[47:15])

Address Range GCR_BASE[47:15]	Size (bytes)	Description
0x0000_0000 - 0x0000_1FFF	8 KB	**Global Control Block**. Contains registers pertaining to the global system functionality. All cores can access this block of registers.
0x0000_2000 - 0x0000_3FFF	8 KB	**Core-Local Control Block** (aliased for each I6500 core). Contains registers pertaining to the I6500 core issuing the request. Each core has its own copy of registers within this block.
0x0000_4000 - 0x0000_5FFF	8 KB	**Core-Other Control Block** (aliased for each I6500 core). This block of addresses gives each Core a window into another core’s Core-Local Control Block. Before accessing this space, the *Core-Redirect Register* in the Local Control sub-block must be set with the CORE-NUM of the target Core.
0x0000_6000 - 0x0000_7FFF	8 KB	**Global Debug Block.** Contains global registers useful in debugging the I6500 MPS.

This concept is described in Figure 5.3. For simplicity, the MIPS default value of 0x0000_1FBF_8 is used for the GCR base address. Each register block is assigned to a contiguous 8 KB space as shown in the figure.

Figure 5.3 CM Register Addressing Scheme Using the MIPS Default in GCR_BASE



5.1.3 Core-Local GCRs

The Core-Local GCR block contains the configuration and status registers for a given core and/or Virtual Processor (VP). Some of the Core-Local registers are per-core, and some are per-VP. A core can access its own Core-Local block to determine the configurable parameters for that core. Parameters include base address assignments, reset exception base, etc.

5.1.4 Core-Other GCRs

The Core-Other GCR block is a single block that all of the cores have access to, and provides a way for one core to access the Core-Local registers of another core. Before a core can access the Core-Other space, the *Core-Redirect* register in that core's own Core-Local Control Block must be set with the core number (CORENUM) of the target core. In this case, a particular core would program the *Core-Redirect* register in its own Core-Local block with the

core number to be accessed. The core would then write the contents of the register to be accessed into the Core-Other address space.

5.1.5 Core-Local and Core-Other Register Usage

As listed in Table 5.2, the CM provides two blocks of registers.

- Core-Local (offset range 0x2000 - 0x3FFF)
- Core-Other (offset range 0x4000 - 0x5FFF)

The CM maintains a copy of selected registers in these blocks. For registers that are instantiated per-core, the CM keeps one copy of the register per core. For registers that are instantiated per-VP, the CM keeps one copy of the register for each VP in a given core. The Core-Local address space contains the GCR registers for that core. The Core-Other address space allows a core to access the GCR registers for another core's Core-Local GCR block.

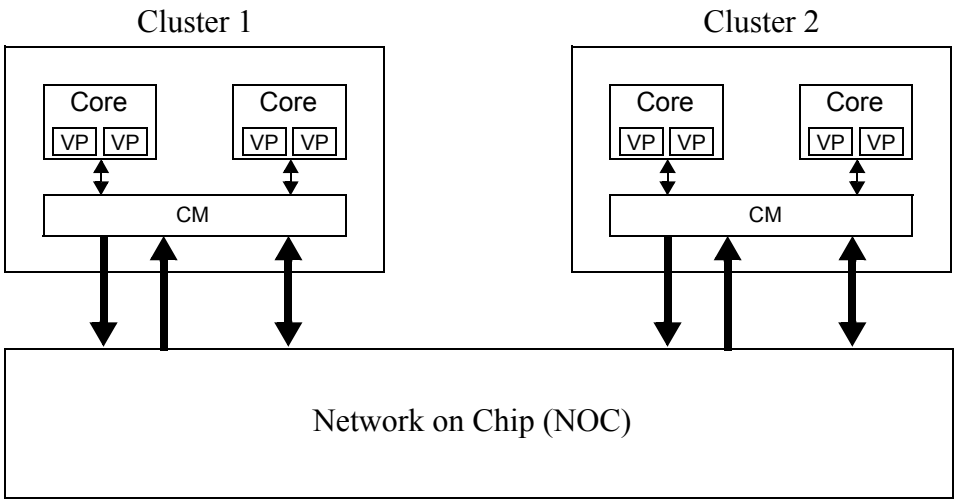
These registers can be located anywhere in physical memory if this option is selected during IP configuration. If this option is not selected, the location of these registers are located at the MIPS default address of 0x0000_1FBF_8000.

The Core-Local block represents registers corresponding to the core that is accessing them. If a core wishes to modify the contents of its own set of CM GCR registers, it writes to the Core-Local block located at the address range shown in Table 5.2. If a core wishes to program the GCR registers of another core, it selects the core number and writes this value into the *Core-Redirect* register in its own Core-Local block at offset address 0x0018. The actual register in the other core to be written would use the corresponding offset in the Core-Other block shown in Table 5.2.

5.1.6 Cluster to Cluster Accesses

In addition to facilitating core-to-core and VP-to-VP accesses within the same cluster, the I6500 also allows for cluster-to-cluster accesses. This allows a core or VP in one cluster to access theregisters in a core or VP of another cluster through the Network-On-Chip (NOC) interface. This interface is shown in Figure 5.4.

Figure 5.4 Cluster-to-Cluster Register Accesses Using the NOC



For example, a VP within a core in Cluster 1 can access and update a register in a VP in Cluster 2 as shown. The access is processed by the CM and driven onto the NOC. The NOC then routes the request to the appropriate cluster where the access is scheduled by the CM in the destination cluster.

If a register access is within a given cluster as shown above, the NOC is not used and the access is placed onto the Register Ring Bus (RRB) described in the section entitled CM Interface — Register Ring Bus and Device ID's. If the register access is to another cluster, the NOC is used to transfer the access request where it is placed onto the RRB of the destination cluster. There are dedicated unidirectional AXI bus interfaces that move the access from the cluster to the NOC, and from the NOC to the cluster. A separate bidirectional bus is used to manage coherence as shown above.

For a programming example of a cluster to cluster access, refer to the section entitled Cluster to Cluster Access.

5.2 Verifying Overall System Configuration

At IP configuration time, the customer selects the number of cores in the system, the number of I/O coherency units (IOCU's), and the number of address regions. When the device is built, these values are hard wired into the *Global Configuration* register at offset address 0x0000. All of these fields are read-only and allow kernel software to quickly determine the system configuration.

CM GCR Register Interface

Reading the *Global Configuration* register provides the following information:

- Bits 7:0 — Number of cores in the system (up to 6)
- Bits 11:8 — Number of IOCU's (up to 8)
- Bits 19:16 — Number of MMIO address regions
- Bits 22:20 — Number of auxiliary memory ports
- Bits 29:23 — Number of clusters in the system
- Bit 31 — Indicates if an Inter-Thread Communication Unit is present
- Bits 39:32 — Indicates the ID number for the current cluster. Each cluster has a unique ID number.
- Bit 40 — Indicates if a Debug Unit is present
- Bits 43:41 — Indicates the type of hardware interface to the Network-On-Chip (NOC) coherent interconnect.

5.3 Programming the Base Addresses in Memory

This section describes how to set the base address of the various CM logic blocks.

CM GCR Register Interface

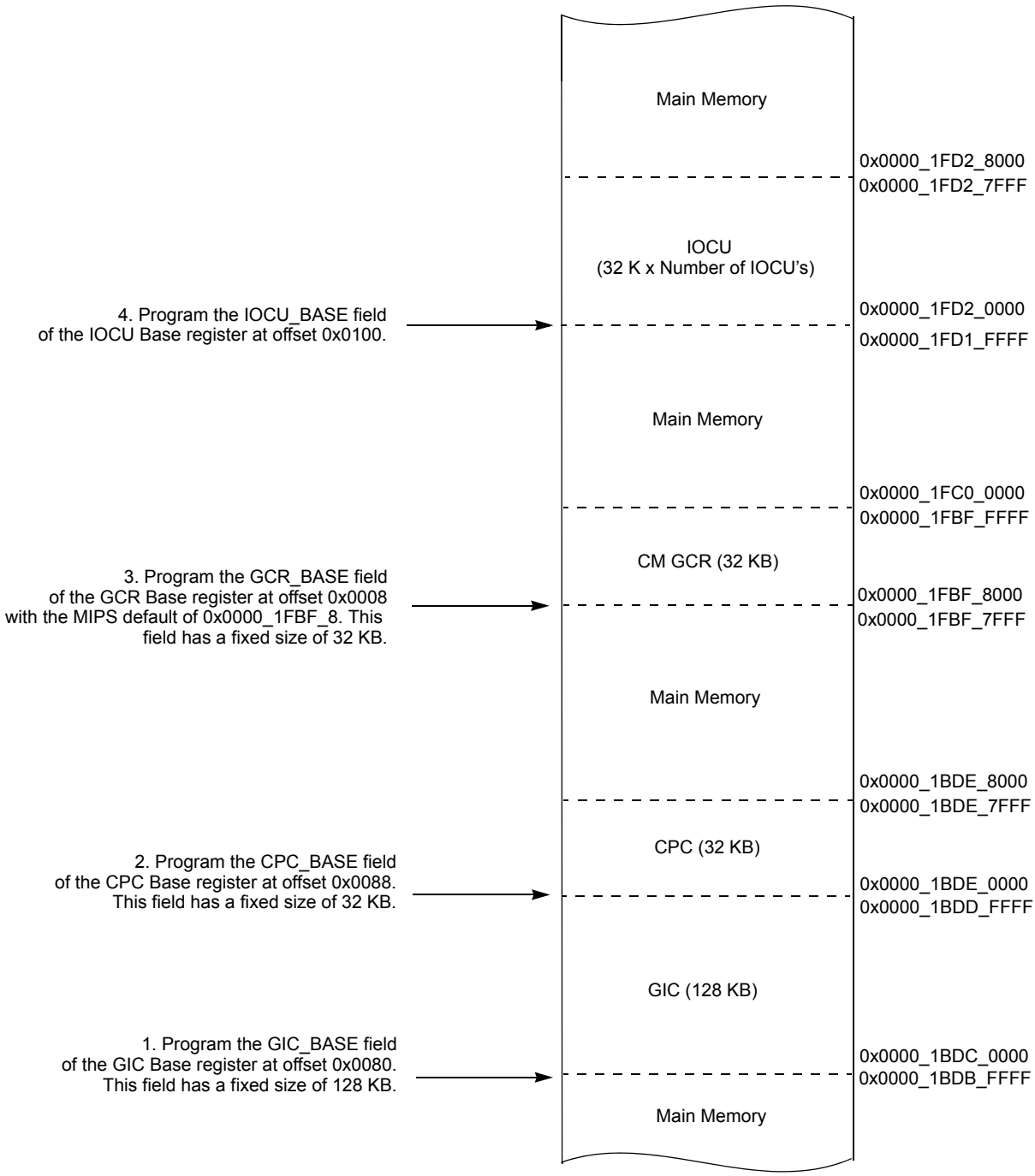
The address map is programmable through a set of registers located in the GCR as summarized in Table 5.3. Up to 5 fixed-size regions can be mapped anywhere in physical memory using the associated Base Address register. Each register indicates the starting address of that block in memory.

Table 5.3 Setting the Base Address for the CM Peripheral Devices

Block	Register Name	Offset Address	Field Name	Bits	Description
GCR	GCR_BASE	0x0008	GCR_BASE_ADDR	47:15	GCR Base Address register. Sets the base address of the GCR registers. Note that this region must reside on a 32 KB boundary.
Custom GCR	GCR_CUSTOM_BASE	0x0060	CUSTOM_ BASE	47:16	Custom Base Address register. Sets the base address of the Customer GCR registers. This region may be disabled via the GGU_EN bit in the *GCR Custom Base* Register. Note that this region must reside on a 64 KB boundary.
GIC	GCR_GIC_BASE	0x0080	GIC_BASE_ADDR	47:17	GIC Base Address register. Sets the base address of the GIC. This GIC region may be disabled via the GIC_EN bit in the *GCR_GIC_BASE* register. Note that this region must reside on a 128 KB boundary.
CPC	GCR_CPC_BASE	0x0088	CPC_BASE_ADDR	47:15	CPC Base Address register. Sets the base address of the CPC. This CPC region may be disabled via the CPC_EN bit in the *GCR_CPC_BASE* register. Note that this region must reside on a 32KB boundary.
IOCU	GCR_IOC_BASE	0x0100	IOC_BASE_ADDR	47:15	Sets the base address of the IOCU. This block contains the IOMMU and associated registers. The IOCU region may be disabled via the *IOCU_REG_EN* bit in the *GCR_IOCU_BASE* register. Note that this region must reside on a 32 KB boundary.

Figure 5.5 provides an example of memory mapping for all of the aforementioned regions at different locations using the MIPS default base address.

Figure 5.5 Address Map Programming Example



The following register programming sequence is used to configure the memory map as shown in Figure 5.5 above. For more information on the corresponding Base Address register, refer to the *I6500 Registers* companion document.

1. To set the base address of the GIC registers to the MIPS default, program the *GIC_BASE* field of the *GIC Base* register located at offset 0x0080 with a value of 0x0000_1BDC. This sets the base address of the GIC registers.

2. To set the base address of the CPC registers to the MIPS default, program the *CPC_BASE* field of the *CPC Base* register located at offset 0x0088 with a value of 0x0000_1BDE_0. This sets the base address of the CPC registers.

3. To set the base address of the CM GCR registers to the MIPS default, program the *GCR_BASE* field of the *GCR Base* register located at offset 0x0008 with a value of 0x0000_1FBF_8. This sets the base address of the 32 KB block of GCR registers. This block is divided into four 8 KB subblocks that contain the Global, Core-Local, Core-Other, and Debug register blocks.

4. To set the base address of the IOCU registers to the MIPS default, program the *IOCU_BASE_ADDR* field of the *IOCU Base Address* register located at offset 0x0100 with a value of 0x0000_1FD2.

5.4 CM Register Access Permissions

A requestor can request access to selected CM registers. A requestor can be either a core or an IOCU. The CM allows up to eight requestors in a system in any combination of cores and IOCU's, from 8 cores and no IOCU's, to 8 IOCU's and no cores, or anywhere in between. Note that all requestor's have read permission to all CM GCR registers, but write access to these registers must be granted.

CM GCR Register Interface

During boot time, the programmer can decide which requestor's are provided access to the CM registers by programming the *ACCESS_EN* field of the *Global CSR Access Privilege* register located at offset 0x0120. Bits 5:0 and 23:16 of this field each correspond to a specific requestor. In bits 5:0, each bit corresponds to a core, with bit 0 mapping to core 0 and bit 5 mapping to core 5. For bits 23:16, bit 16 maps to IOCU0, and bit 23 maps to IOCU7. The MIPS default for this field is 0x0000_0000_00FF_00FF, meaning that all requestor's in the system (all cores and all IOCU's) have access to the CM register set.

To disable access to the registers for a particular requestor, the programmer need only clear the corresponding bit of this field to zero and all write requests to the CM registers by that requestor are ignored.

Note that by setting one of these bits described above, write access is granted to the requestor for both the CM GCR register block, as well as the Cluster Power Controller (CPC) register block. Refer to the *CPC Programming* chapter in this manual for more information.

Register Access Permissions Code Example

The base address for the location of the CM GCR registers is programmed into the CP0 CMGCRBase register. In this example, the base address could be any value. As a reference, a value of 0x0000_1FBF_8 is used (MIPS default) to indicate the base location of the CM global control registers. In this case, the base value is read and an offset is added to it to derive the exact register address.

By default all IOCU's and cores are enabled. This example reprograms the CM *Global Access Privilege* register to enable only IOCU0 and core 0.

```
#define c0_CMGCRBASE          $15,3

mfc0   t1, c0_CMGCRBASE       // move contents of CP0 CMGCRBase register into t1
dsll   t1, t1, 4              // shift value in t1 left by 4 bits
li     t2, 0xA000_0000        // assign KSeg1 base
```

```
or      t1, t2, t1              // create VA from CGRBase
li      t0, 0x0001_0001         // set value to enable IOCU0 and core0 only
sd      t0, 0x120 (t1)          // write value in t0 to the base address in t1 plus
                                // an offset of 0x120.
```

5.5 CM Programming Examples

This section describes how to program the CM to accomplish the following tasks:

- Section 5.5.1 “Programming Another Virtual Processor (VP) in the Same Core”
- Section 5.5.2 “Programming Local GCR’s Corresponding to Another Core”
- Section 5.5.3 “Accessing the CPC Local Registers via the CM”
- Section 5.5.4 “Powering Up the Debug Unit (DBU) via the CM”
- Section 5.5.5 “Setting the Clock Ratios Between the I6500 System Components”
- Section 5.5.7 “Accessing the Core-Local and Core-Other Registers in the Global Interrupt Controller”

5.5.1 Programming Another Virtual Processor (VP) in the Same Core

The I6500 MPS provides the ability for a given core to access registers within the same core, but corresponding to a different Virtual Processor (VP). This is done using the *VP-Local GCR Redirect* register located at offset address 0x0018 in the Core-Local register block. There is one instantiation of this register per VP. As such, there can be up to four of these registers in a given core (in a 4-VP configuration). Bits 13:8 (CORE_REDIRECT) of this register indicate the core to be accessed, and bits 2:0 (VP_REDIRECT) indicate the VP to be accessed. In this case, a different VP is being accessed inside the same core.

For example, assume that core 1, VP 0 wants to modify the reset exception base address in core 1, VP 2. To facilitate this transaction, kernel software would program a value of 1 into the CORE_REDIRECT field in bits 13:8 of the *VP-Local GCR Redirect* register located at offset address 0x0018, indicating the transaction is intended for core 1. Software would also program a value of 2 into the VP_REDIRECT field of this same register, indicating that the transaction is intended for VP 2. Software would then write the modified value to the *Core-Local Reset Exception Base Address* register located at offset address 0x0020. Both of these registers are instantiated per-VP in the I6500 core.

When accessing another core or VP's Core-Local registers, set the CLUSTER_REDIRECT_EN bit 31 of the VP Local GCR Redirect register at offset 0x0018 to 0 to indicate that the access stays in this cluster. Also set the BLOCK_REDIRECT field in bits 25:24 to 0 to indicate that the access should be redirected to the Core-Local block of registers.

In addition to allowing one VP to access another VP within a given core, the I6500 allows different register blocks within the same VP to be accessed. This is accomplished by setting the CLUSTER_REDIRECT_EN bit 31 of the VP Local GCR Redirect register at offset 0x0018 to 0 to indicate that the access stays in this cluster, and by setting the BLOCK_REDIRECT field in bits 25:24 of the *VP-Local GCR Redirect* register located at offset address 0x0018. In this example, the *Core-Local Reset Exception Base Address* register being modified is in the Core-Local register block of VP2, so the value in the BLOCK_REDIRECT field would be 0 to indicate the Core-Local block of registers. However, if a Global register block or Debug register block was being accessed by VP1, the value in this field would reflect the appropriate register block. The BLOCK_REDIRECT field supports other values to allow core-other accesses to be redirected to other blocks such as the Global block or Debug block when accessing registers in other clusters.

CM GCR Register Interface

The following steps show the register programming sequence for this example. Refer to the *I6500 Registers* companion document for more information on this register.

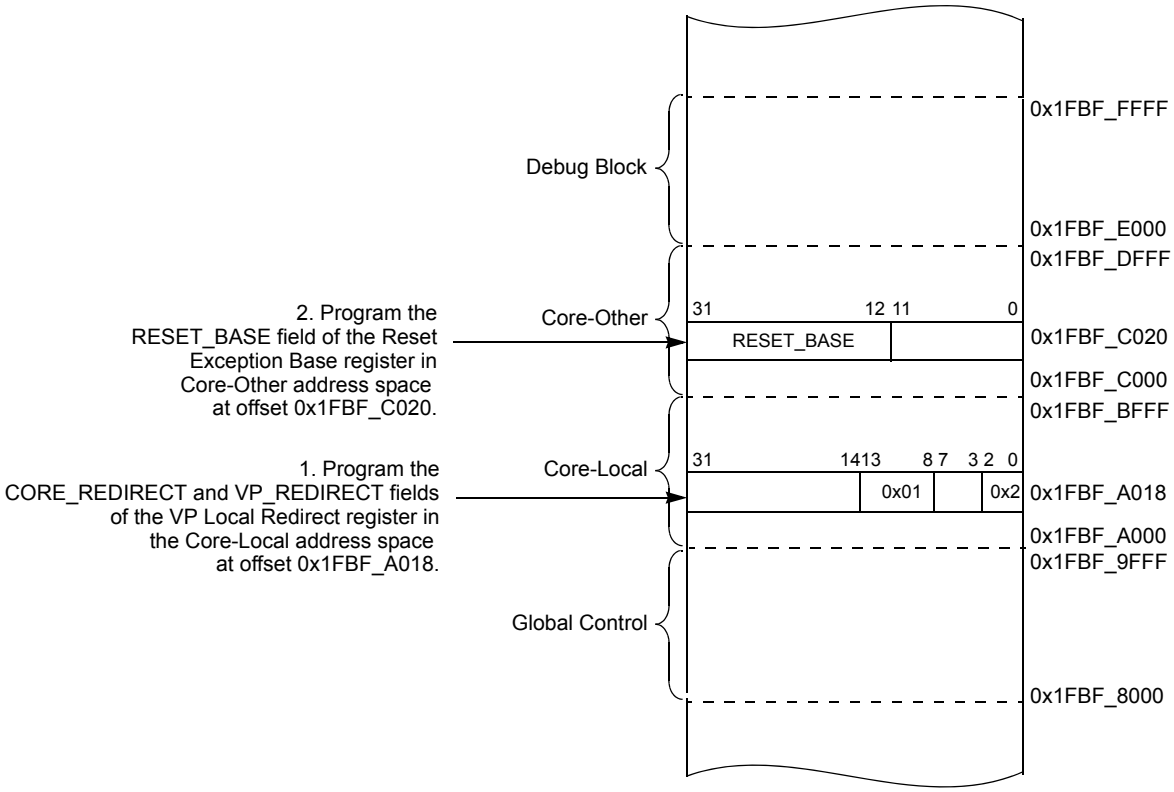
1. Core 1 writes a value of 0x01 to the *CORE_REDIRECT* field (bits 13:8) of the *VP-Local GCR Redirect* register located at offset 0x0018 (physical address of 0x1FBF_A018). This indicates that the register to be programmed corresponds to core 1.

2. Core 1 also writes a value of 0x2 to the *VP_REDIRECT* field (bits 2:0) of the *VP Local Redirect* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018). This indicates that the register to be programmed corresponds to VP 2 of core 1.

3. Since the Reset Exception Base register is instantiated on a per-VP basis, Core 1 writes the appropriate value into the *EXCBase* field (bits 31:12) of the *VP-Local Reset Exception Base* register located in the Core-Other block at offset 0x0020 (physical address of 0x1FBF_C020).

This concept is shown in Figure 5.6.

Figure 5.6 Core 1, VP 0 Accessing the BEV_BASE GCR of Core 1, VP 2



5.5.2 Programming Local GCR's Corresponding to Another Core

In a multiprocessor system, it is common for one core to boot up first, then have that core boot the other cores in the system. In the following example, assume core 0 is booted up first. Then core 0 is used to program the GCR registers in core 1. This example examines how core 0 would program the boot exception vector location for core 1, VP 0 by

setting its Reset Exception Base register. Note that this example uses the MIPS default addressing scheme and assumes that core 1 has already been powered up through the CPC. If the core has not been powered up, refer to the section entitled Accessing the CPC Local Registers via the CM.

CM GCR Register Interface

The following steps show the register programming sequence for this example. Refer to the *I6500 Registers* companion document for more information on this register.

1. Core 0 writes a value of 0x01 to the *CORE_REDIRECT* field (bits 13:8) of the *VP-Local Redirect* register located located at offset 0x0018 (physical address of 0x1FBF_A018). This indicates that the register to be programmed corresponds to core 1.

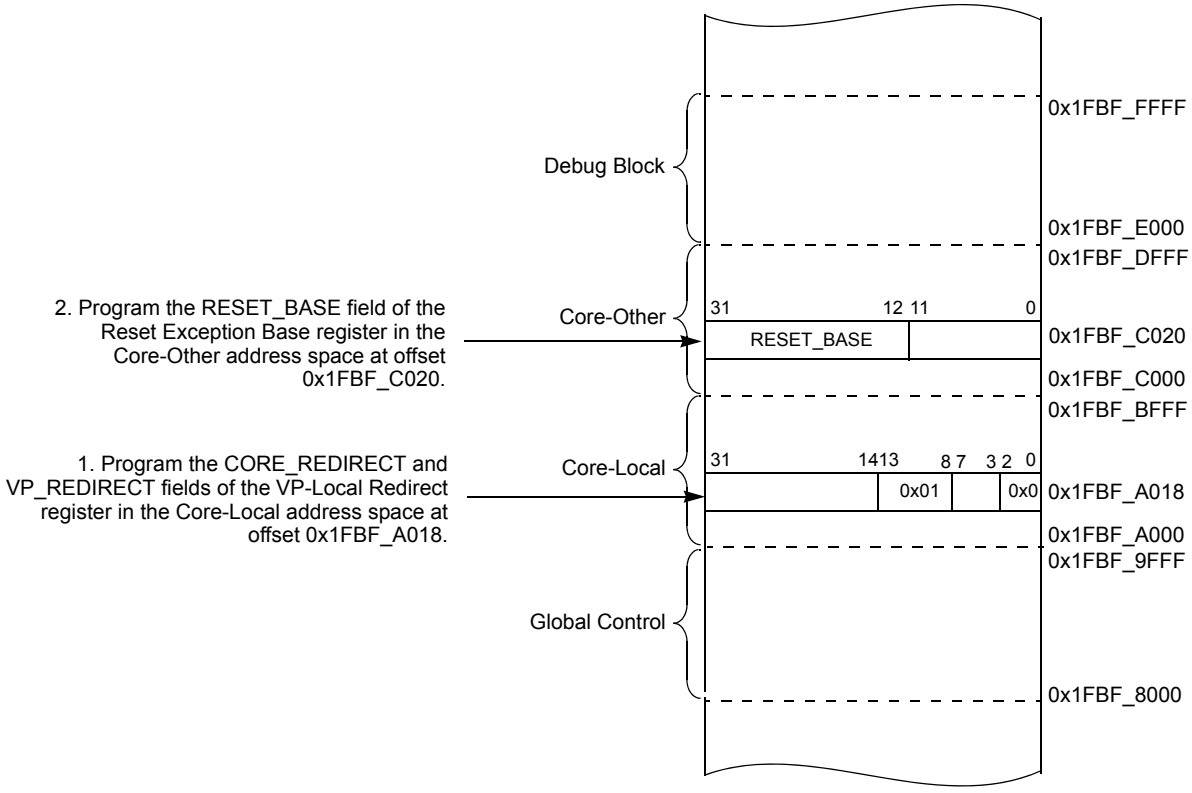
2. In addition, Core 0 also writes a value of 0x0 to the *VP_REDIRECT* field (bits 2:0) of the *VP-Local Redirect* register. This indicates that the register to be programmed corresponds to VP 0 of core 1.

3. Core 0 writes the appropriate value into the *Reset_Base* field (bits 31:12) of the *Reset Exception Base* register located in the Core-Other block at offset 0x0020 (physical address of 0x1FBF_C020). Because core 0 is setting the Reset base value for core 1, as opposed to its own core, the write is done to the Core-Other address block.

Whenever one core reads or writes to the registers associated with another core, the number of the core to be written is programmed into that cores local *CORE_REDIRECT* field as described in step 1 above. Similarly, the number of the VP to be written is programmed into that cores local *VP_REDIRECT* field as described in step 2 above. The actual register to be programmed is accessed via the Core-Other block as described in step 3 above.

Since there is only one Core-Other block in Table 5.2, this means that when one core wants to access any of the other cores in the system, the register to be accessed always resides in the Core-Other block, regardless of the number of cores in the system. The state of the *CORE_REDIRECT* field in the *VP_Local Redirect* register in that core's own Core-Local space determines which core the data is written to. This concept is shown in Figure 5.7.

Figure 5.7 Core 0 Accessing the Reset Exception Base Register of Core 1



The reset vector can either be placed in the lower 512 MBytes of address space, or anywhere within the 4 GByte address space, depending on the programming of the *RESET_BASE_MODE* bit of the *Reset Exception Base* register located in the Core-Other block at offset 0x0020. This bit can be set during device configuration and is normally not changed once it is set.

Note that in addition to the *CORE_REDIRECT* field used to indicate the number of the destination core as described in #1 above, a core can determine its own core number by reading the *CORENUM* field in its own *Core-Local Identification* register located at offset 0x0028 in the GCR Core-Local address space.

5.5.3 Accessing the CPC Local Registers via the CM

This example shows how Core 0 uses the Core-Local and Core-Other registers to power up core 1. This sequence is different that the one described in the section entitled Programming Local GCR's Corresponding to Another Core above, which assumes that core 1 has already been powered up. Note that this example uses the default addressing scheme. Refer to the *I6500 Registers* companion document for more information on this register.

CM GCR Register Interface

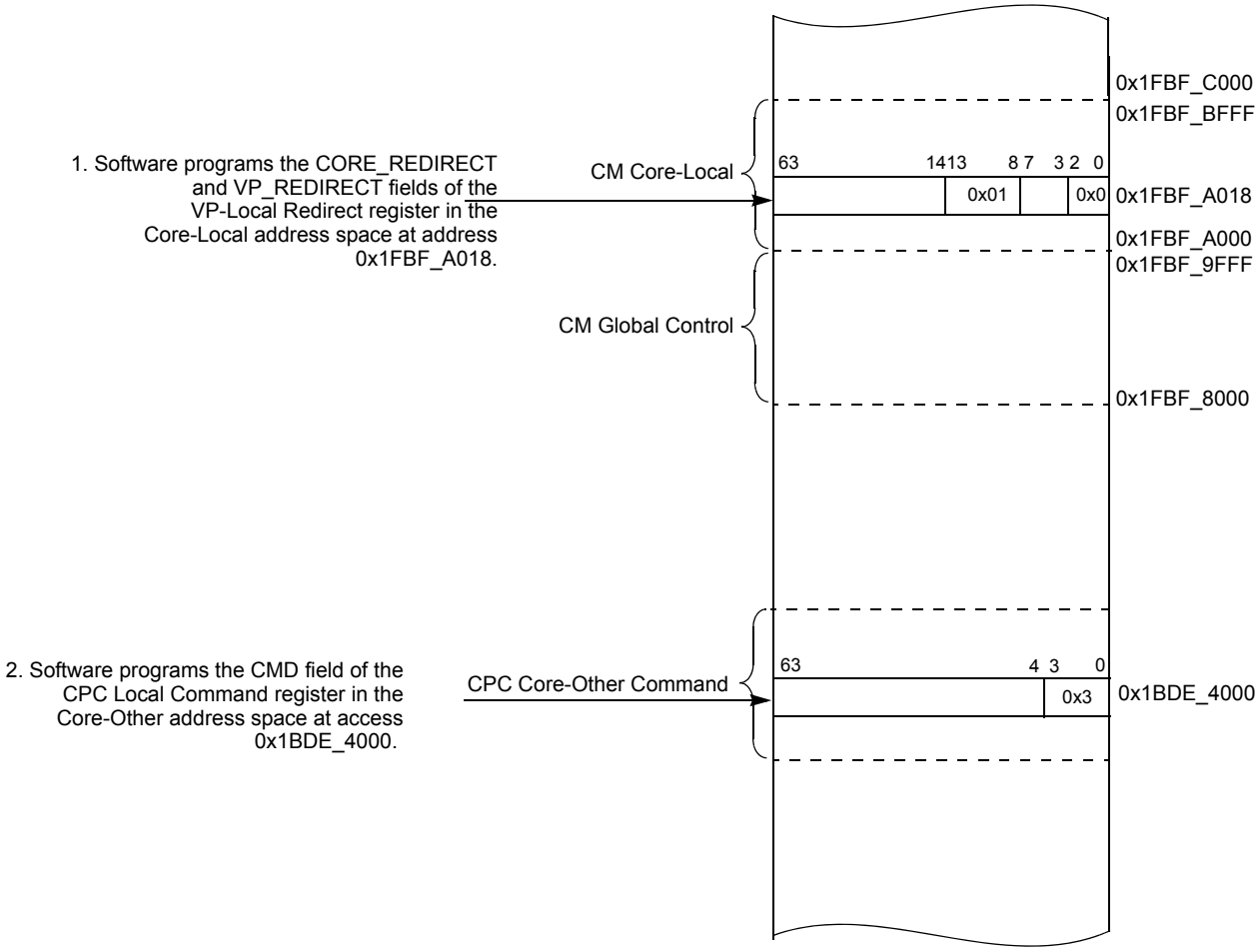
The register programming sequence for this example would be as follows:

1. Core 0 writes a value of 0x01 to the *CORE_REDIRECT* field (bits 13:8) of the *VP_Local Redirect* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018 in Figure 5.5). This indicates that the register to be programmed corresponds to core 1.

2. Core 0 then writes a value of 0x3 into the *CMD* field (bits 3:0) of the *CPC Local Command* register located in the CPC Core-Other block at offset 0x0000 (physical address of 0x1BDE_2000 in Figure 5.5). A value of 0x3 in this field indicates to the CPC to power up the core indicated in the *CORE_REDIRECT* field (bits 13:8) of the *VP_Local Redirect* register.

This concept is shown in Figure 5.8.

Figure 5.8 Core 0 Using the CPC Core Local Register to Power Up Core 1



5.5.4 Powering Up the Debug Unit (DBU) via the CM

The I6500 MPS contains a dedicated Debug Unit (DBU) that is used to perform debug and analysis on the various components in the system. This section describes how to power up the Debug Unit by accessing the DBU copy of the CPC Core-Local Command register. The DBU has a ring ID value of 35, or 0x23 as described in Table 5.1. The ring ID value of 35 is used in the CORE_REDIRECT field of the VP-Local Redirect register to indicate that Core-Other accesses should target the DBU copy of the register. Note that this example uses the MIPS default addressing scheme.

CM GCR Register Interface

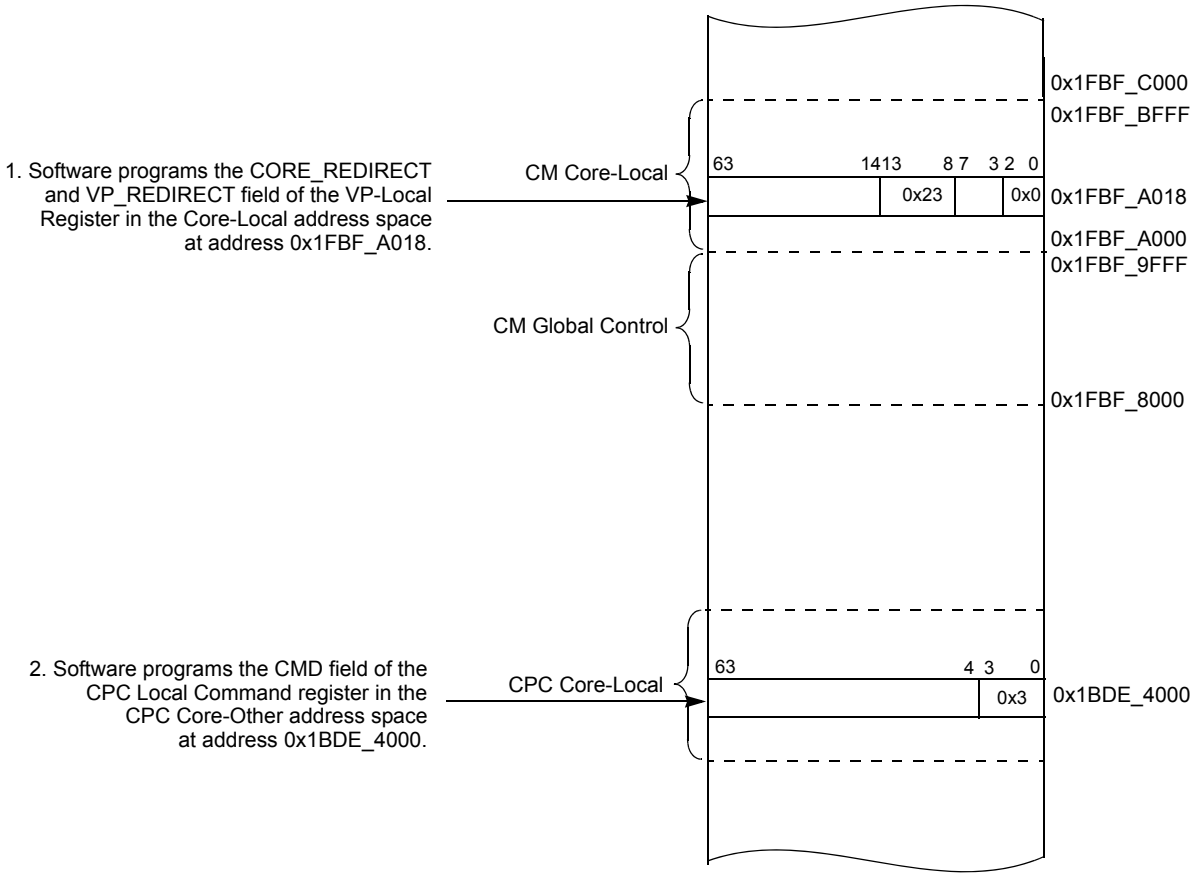
The register programming sequence for this example would be as follows. Refer to the I6500 Registers companion document for more information on this register.

1. Core 0 writes a value of 0x23 to the *CORE_REDIRECT* field (bits 13:8) of the *VP-Local Redirect* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018 in Figure 5.5). This indicates that the register to be programmed corresponds to the Debug Unit (DBU).

2. Core 0 then writes a value of 0x3 into the *CMD* field (bits 3:0) of the *CPC Local Command* register located in the CPC Core-Other block at offset 0x0000 (physical address of 0x1BDE_4000 in Figure 5.5). A value of 0x3 in this field indicates to the CPC to power up the component indicated in the *CORE_REDIRECT* field (bits 13:8) of the *GCR Redirect* register, which in this case is the DBU (0x23).

This concept is shown in Figure 5.9.

Figure 5.9 Core 0 Using the CPC Core Other Register to Power Up the Debug Unit



5.5.5 Setting the Clock Ratios Between the I6500 System Components

In addition to powering up elements such as cores and the DBU as described in the previous subsections, the I6500 Multiprocessing System also allows these different elements to run at various clock frequencies relative to each other

This section describes how to set a 4:1 clock ratio between the core clock and IOCU 0 by writing to an IOCU 0 copy of the *CPC Local Clock Change Control* register in the CPC address space. The IOCU 0 has a ring ID value of 16, or 0x10 as described in Table 5.1. The ring ID value of 0x10 is used in the CORE_REDIRECT field of the VP-Local Redirect register to indicate that Core-Other accesses should target the IOCU copy of the CPC register. Note that this example uses the MIPS default addressing.

CM GCR Register Interface

The following steps show the register programming sequence for this example. Refer to the *I6500 Registers* companion document for more information on these registers.

1. Core 0 writes a value of 0x10 to the *CORE_REDIRECT* field (bits 13:8) of the *VP-Local Redirect* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018 in Figure 5.5). This indicates that the clock ratio to be programmed corresponds to IOCU 0.

2. Core 0 then writes a value of 0x3 into the *CLK_RATIO* field (bits 3:0) of the *CPC Local Clock Change Control* register located in the CPC Core-Other block at offset 0x0018 (physical address of 0x1BDE_4018 in Figure 5.5). A value of 0x3 in this field indicates a clock ratio of 4:1 between the prescaled clock and IOCU 0. Hardware reads the *VP-Local Redirect* register in step 1 to determine that IOCU 0 is the device to be programmed with the associated clock ratio.

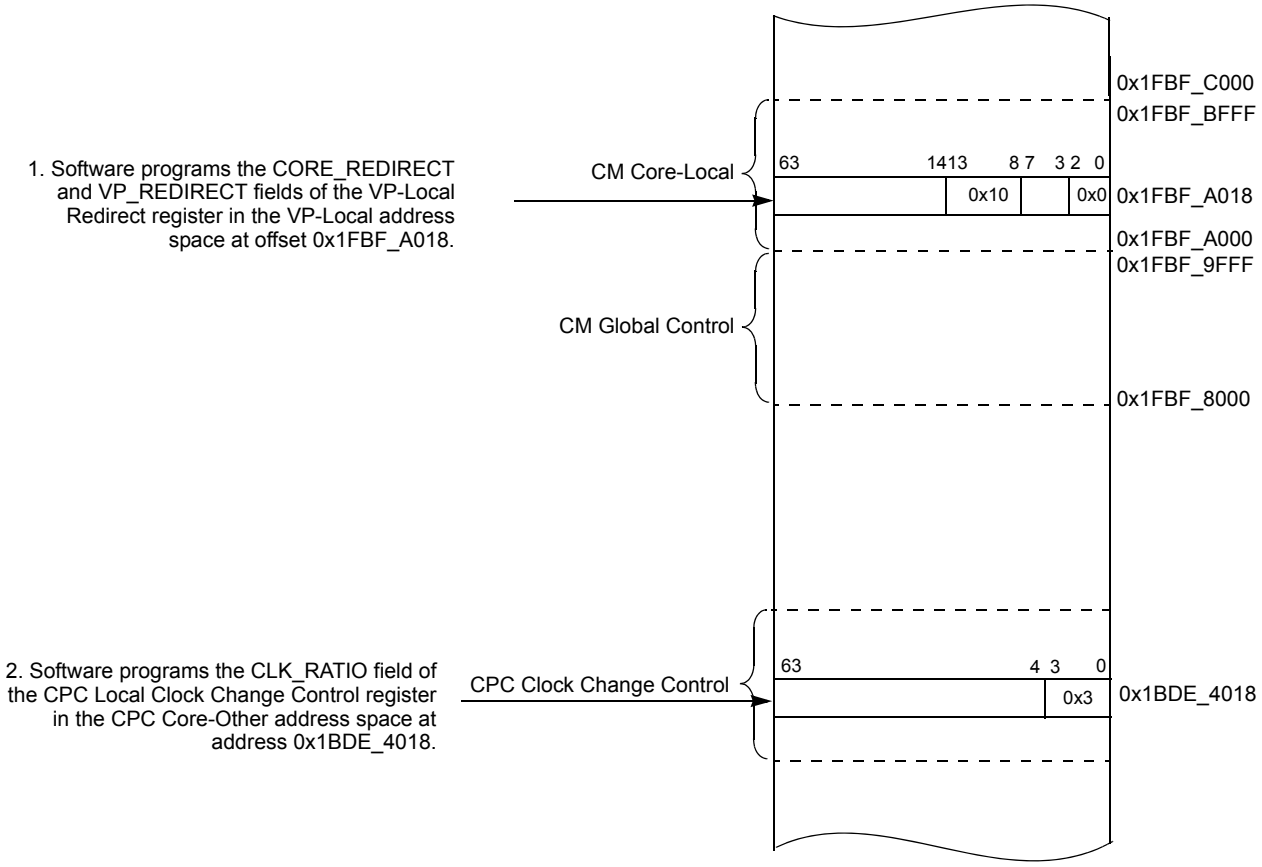
3. Core 0 writes a value of 0x1 into the CLK_RATIO_CHANGE_EN field (bit 8) of the CPC Local Clock Change Control register located in the CPC Core-Other block at offset 0x0018 (physical address of 0x1BDE_4018 in Figure 5.5). A value of 0x1 in this field enables the clock domain to change rates when the clock change sequence is started.

4. Initiate the clock change sequence by writing a value of 0x1 into the SET_CLK_RATIO field (bit 0x8) of the CPC Global Clock Control register in the CPC Global block at offset 0x0028 (physical address of 0x1BDE_0028).

5. Poll for clock changes complete by reading the CPC Global Clock Control register in the CPC Global block at offset 0x0028 (physical address of 0x1BDE_0028). If the field SET_CLK_RATIO (bit 8) is set, then the clock change sequence is still pending. If the field CLK_CHANGE_ACTIVE (bit10) is set, then the clock change sequence is in process. If both fields are zero, then the clock change has completed.

This concept is shown in Figure 5.10.

Figure 5.10 Core 0 Using the CPC Core Local Register to Set the IOCU 0 Clock Ratio



The above procedure can be used to set the clock ratio for any of the programmable clock domains. For example, to program the clock ratio for IOCU 1, simply substitute a value of 0x11 for 0x10 in the above example since IOCU 1 is located at ring bus ID 17. Similarly, to set the main memory clock ratio, simply substitute a value of 0x1A in the above example since main memory is located at ring bus ID 26. Once the device has been selected, the CLK_RATIO field can be used to set the ratio between the prescaled clock and all selected devices to a value between 1:1 and 1:8, except the CM, which is limited to a clock ratio of 1:1 or 1:2.

5.5.6 Cluster to Cluster Access

As described in Section 5.1.6, the I6500 CM allows a core or IOCU in one cluster to access a software visible registers in a different cluster.

To access a register on a remote cluster, the kernel software must use the GCR_CL_REDIRECT register. There is one GCR_CL_REDIRECT register per VP and IOCU in the cluster. The kernel software writes the target cluster, core, VP, and target register block into this register, along with setting the CLUSTER_REDIRECT_EN bit. Then an access to the target device's Core-Other block causes the CM to drive the request to the NOC for transfer to the target cluster.

In the following example, core 1 in Cluster 1 reads the Global CM Error Cause register in Cluster 2 to determine the cause of an error.

The following steps show the register programming sequence for this example. Refer to the *I6500 Registers* companion document for more information on the registers discussed in this example.

1. Core 1 writes a value of 0x0000_0000_8102_0000 to the *VP-Local Redirect* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018. This indicates an access to Cluster 2. The value written to this register is broken down as follows:
   - The CLUSTER_REDIRECT_EN bit 31 is set to enable a request to another cluster. Setting this bit enables use of the CLUSTER_REDIRECT field.
   - The BLOCK_REDIRECT field in bits 25:24 is set to 0x1 to indicate that a register in the CM Global Register Block of the destination cluster is to be accessed.
   - The CLUSTER_REDIRECT field in bits 21:16 is set to 0x02 to indicate that Cluster 2 is being accessed.
   - The CORE_REDIRECT field in bits 13:8 VP_REDIRECT field in bits 2:0 are not programmed in this example as the access is not to a specific core of VP of cluster 1, but rather a GCR register inside the CM.
2. Core 1 then reads the value in the GCR Error Cause (GCR_ERR_CAUSE) register in the Core-Other block at offset address 0x0048. Hardware reads the *VP-Local Redirect* register in step 1 to determine that read should be routed to cluster 2. Based on this information, the CM sends the access through the NOC to the other cluster. The CM in the destination cluster decodes the information and reads its local GCR_ERR_CAUSE register. The result is then returned through the NOC to the CM in cluster 1.

5.5.7 Accessing the Core-Local and Core-Other Registers in the Global Interrupt Controller

In the previous subsections, the VP-Local, Core-Local, and Core-Other registers of the CM have been used to modify parameters in other cores, other VP's within the same core, and other non-core devices on the register ring bus. This programming mechanism is applicable for all devices on the register ring bus except the Global Interrupt Controller (GIC). The GIC has it's own Core-Local and Core-Other register set that is instantiated on a per-VP basis. The *VP-Local Redirect* register is used to select the target VP copy of the register to access when a*GIC Core-Other* register is read or written. For more information, refer to the Global Interrupt Controller (GIC) chapter of this manual.

5.6 Coherency Enable

The I6500 Multiprocessing System allows each power domain to be placed in either a coherent or non-coherent mode. Because the I6500 implements a directory-based coherence protocol, MIPS recommends that each domain be placed in coherent mode during normal operation. The non-coherent mode should only be used during boot-up and power-down. Software should not execute any cacheable memory accesses (instruction fetch or load/store) while coherence is disabled.

In the CM, coherency is either enabled or disabled using the *Core-Local Coherence Enable* register at offset 0x0008 in the Core-Local register block.

CM GCR Register Interface

Coherency is enabled when hardware asserts the external *Coherence Enable* pin. The state of this pin is reflected in bit 11 (COH_EN) of the Core-Local Status and Configuration register. This register resides in the CM local register block at offset address 0x0008. There is one of these registers per power domain.

For more information on this register, refer to the I6500 Registers companion document included in the release.

Note that if a power domain is in coherent mode and a change to the power state is initiated, the caches must be flushed prior to disabling coherence mode.

Coherency Enable Code Example

```
li t1, CPC_BASE_ADDR    // move CPCBase  value into t1
li t0, 0x0000_0001      // Enable coherence
sd t0, 0x2008 (t1)      // write value in t0 to the base address in t1 plus
                        // an offset of 0x2008 to access the Coherence Enable
                        // register.
```

5.7 L2 Cache Prefetch

The coherence manager in the I6500 MPS contains an L2 prefetcher used to enhance L2 performance. The L2 prefetcher is managed using two CM GCR registers.

- L2 Prefetch Control register (GCR_L2_PFT_CONTROL) at offset 0x0300
- L2 Prefetch 2nd Control register (GCR_L2_PFT_CONTROL_B) at offset 0x0308

These registers control the following L2 capabilities:

- Minimum operating system page size (supports 4K - 64K pages in multiples of two)
- Prefetch enable
- Coherent invalidate requests
- Code prefetch enable
- L2 prefetching port ID. Each bit corresponds to a CM port ID. If the bit is set, the corresponding CM port is monitored for prefetching.

5.7.1 Prefetch Enable

The number of prefetch units implemented in the I6500 Multiprocessing System is determined by the user during IP configuration. This value is programmed by hardware into the NPFT field (bits 7:0) of the L2 Prefetch Control register (GCR_L2_PFT_CONTROL) located at offset address 0x0300 in the GCR Global register space. This read-only field allows kernel software a convenient way to determine the number of prefetch units implemented.

CM GCR Register Interface

Prefetching is enabled by setting the PFTEN bit in the GCR_L2_PFT_CONTROL register. Note that the number of prefetch units implemented as described above must be greater than 0 in order for this bit to have meaning.

5.7.2 Select Ports for L2 Prefetching

The CM allows up to 8 ports to be selected for L2 prefetching. These ports correspond to the (up to) six cores and (up to) eight IOCU's as shown in Figure 5.1. L2 prefetching can be selected for some of all of these ports using the 8-bit PORT_ID field in the GCR_L2_PFT_CONTROL_B register. Each bit of this field corresponds to a single port. There can be any number of cores and IOCU's up to the maximum or eight. For example, if there are 8 cores, then there must be 0 IOCU's to make a total or 8, or 4 cores and 4 IOCU's, etc. If a given bit is set, L2 prefetching is monitored for that port. If the bit is cleared, L2 prefetching does not occur.

The field is organized as cores followed by IOCU's starting from bit 0. So in a 4-core and 2-IOCU system, bits 0 - 3 of the field would represent cores 0 - 3 respectively. Bits 4 - 5 of the field would represent IOCU 0 - 1 respectively. Bits 6 - 7 would not be used in this example.

5.7.3 Enabling Code Prefetch

In addition to data prefetching, the CM allows prefetching of the code stream. Code prefetching is enabled by setting the CEN bit in the GCR_L2_PFT_CONTROL_B register.

5.8 CM Uncached Semaphore Management

The I6500 CM provides a mechanism for managing uncached semaphores. This mechanism is managed by the *Global CM Semaphore* (GCR_SEM) register located at offset address 0x0640.

A write to this register with write data bit 31 = 1 is inhibited if the SEM_LOCK bit is already 1. A write to this register proceeds normally if the write data has bit 31 = 0 or if the SEM_LOCK bit is currently 0.

CM GCR Register Interface

To acquire the semaphore:

1. Write this register with bit 31 = 1 and the lower bits with the threads VPID.
2. Read the register.
3. If the value read in step #2 is the same as the value as written in step #1, then a semaphore has been acquired, else go to step #1.

To release the semaphore:

1. Write the register with bit 31 = 0.

For more information, refer to the CM GCR Semaphore Lock register (GCR_SEM) at offset 0x0640 in the I6500 Registers companion document.

5.9 Custom GCR Implementation

The CM provides the ability for the system designer to implement a 64 KB block of custom registers that can be used to control system level functions. These registers are defined by the system designer and then instantiated into the design.

The existence of a custom GCR implementation in the system is selected during IP Configuration. If this option is selected, the GGU_EX bit is set in the *Global Custom Status* register at offset address 0x0068 in GCR Global address space. This bit indicates that a custom GCR block is connected to the CM.

CM GCR Register Interface

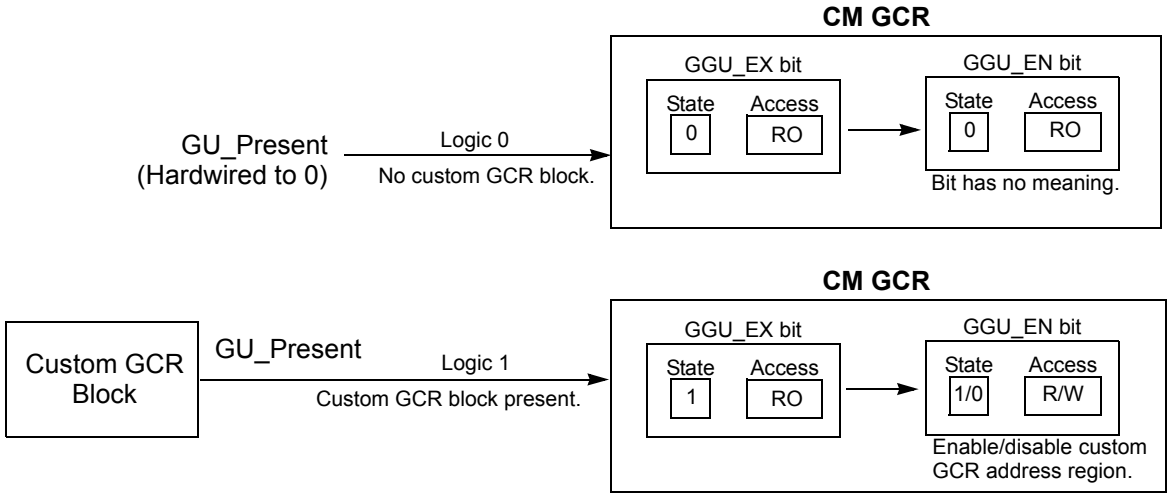
The CM provides two global registers to handle the implementation of custom registers: the *Global Custom Base* register at offset 0x0060, and the *Global Custom Status* register located at offset 0x0068. If a custom block is imple-

mented, the starting address in memory of the 64 KB block is determined using the 16-bit CUSTOM_BASE field in the *Global Custom Base* register. Note that the CUSTOM_BASE field does not have a default base address and this field is undefined at reset. Therefore, it is programmer's responsibility to program the base address into this field during boot time if a custom GCR block is implemented.

In addition, the selected address region where the registers will reside must be enabled by setting the GGU_EN bit in the *Global Custom Base* register. Note that the accessibility of this bit depends on the state of the *GGU_EX* bit. If *GGU_EX* is cleared (zero), indicating that no custom GCR is connected to the CM, then the *GGU_EN* bit becomes RO and is not accessible by the kernel. If this bit is set, indicating that a custom GCR is connected to the CM, then the *GGU_EN* bit becomes R/W and is accessible by kernel software.

This concept is described in Figure 5.11.

Figure 5.11 Relationship Between the CM_Present Signal and the GGU_EX and GGU_EN Bits at Reset



5.10 Error Processing

The CM detects, reports, and handles several types of hardware and software errors. When an error is detected, information that may be useful in debugging the error is captured in the *Global CM Error Cause Register* and *Global CM Error Address Register*. The encoding of these registers is determined by the type of error. For more information, refer to the registers in the *I6500 Technical Reference* manual.

CM GCR Register Interface

When an error occurs, hardware updates the read-only ERR_TYPE field (bits 63:58) of the *Global CM Error Cause* register with one of the values listed in Table 5.4. When this field is written, hardware also updates the 58-bit ERROR_INFO field that provides additional information about the error. The organization of this field varies depending on the value in the ERR_TYPE field. When an error occurs, kernel software can read this register to determine the type of error and take the appropriate actions.

If a second error is detected, it is captured in bits 63:58 of the *CM Error Multiple Register*. The only exception is if the first error was an L2 RAM correctable error (MP_CORRECTABLE_ECC_ERR). In this case, the second error overwrites the first error stored in the *Global CM Error Cause* register. Note that for the second error, only the error type is captured, not the associated error address.

The *GCR_ERROR_CAUSE.ERR_TYPE* field and the *GCR_ERROR_MULT.ERR_TYPE* fields can be cleared by either a reset or by writing the current value of *GCR_ERROR_CAUSE.ERR_TYPE* to the *GCR_ERROR_CAUSE.ERR_TYPE* register.

When the *Global CM Error Cause Register* is loaded, an interrupt may be generated if the corresponding bit for that type of error is set in the *Global CM Error Mask Register* located at offset address 0x0040 (physical address 0x1FBF_8040).

Note that in the CM, the error response is independent of the mask setting, which is different from the previous generation CM2. If the normal response should be an ERROR, then an ERROR response is returned regardless of the *Error Mask Register* setting. The mask setting controls whether an interrupt is generated in addition to the normal error response.

Table 5.4 lists the errors detected by the CM. The following subsections describe each type of error in more detail and provides the encoding of the ERR_INFO field for each error type. For a detailed description of each error type and the encoding of each error code field, refer to the *I6500 Technical Reference Manual*.

Table 5.4 CM Error Types

ERROR TYPE	Error Name	Description	Action
0	-	Reserved	-
1	*MP_CORRECTABLE_ECC_ERR*	A correctable ECC error occurred during an L2 cache access.	The error is corrected Signal an interrupt if CM_ERROR_MASK[1] = 1
2	*MP_REQUEST_DECODE_ERR*	A decoding error was detected in the request.	Respond with an error to the original requestor. Signal an interrupt if CM_ERROR_MASK[2] = 1
3	*MP_UNCORRECTABLE_ECC_ERR*	An uncorrectable ECC error occurred during an L2 cache access.	Signal an interrupt if *CM_ERROR_MASK[3]* = 1
4	*MP_PARITY_ERR*	A parity error was detected in the L2 data coming from either the core of the memory.	Signal an interrupt if *CM_ERROR_MASK[4]* = 1
5	*MP_FNL_ERR*	If an L2 fetch and lock (FNL) cacheop is processed when only one or zero ways of the cache are unlocked, including pseudo-locks, then the FNL fails.	Signal an interrupt if *CM_ERROR_MASK[5]* = 1
6	*CMBIU_REQUEST_DECODE_ERR*	A decoding error was detected during a request on the BIU.	Signal an interrupt if *CM_ERROR_MASK[6]* = 1
7	*CMBIU_PARITY_ERR*	The BIU detected a parity error.	Signal an interrupt if *CM_ERROR_MASK[7]* = 1
8	*CMBIU_AXI_RESP_ERR*	The BIU detected a response error was detected on the AXI bus.	Signal an interrupt if *CM_ERROR_MASK[8]* = 1
9	*CMBIU_WID_ERR*		Signal an interrupt if *CM_ERROR_MASK[9]* = 1
10	*RBI_BUS_ERR*	An error occurred on the Register Ring Bus during a register access.	Signal Interrupt if *CM_ERROR_MASK[10]* = 1
11	*IOC_REQUEST_ERR*	An error occurred during an AXI request.	Signal Interrupt if *CM_ERROR_MASK[11]* = 1

Table 5.4 CM Error Types *(continued)*

ERROR TYPE	Error Name	Description	Action
12	*IOC_PARITY_ERR*	The IOCU detected a parity error.	Signal Interrupt if *CM_ERROR_MASK[12]* = 1
13	*IOC_RESP_ERR*	The IOCU detected a response error.	Signal Interrupt if *CM_ERROR_MASK[13]* = 1
14	*HALF_PIPE_ERR*	The main pipeline received an error from the half-pipe.	Signal Interrupt if *CM_ERROR_MASK[14]* = 1
15	*RBI_REGTC_REQ_ERR*	An illegal request was received by the REGTC.	Signal Interrupt if *CM_ERROR_MASK[15]* = 1

5.11 IOCU Interface

The I6500 CM contains up to eight I/O Coherency Units (IOCU) for managing cache coherency between the CM and external devices. The IOCU is a hardware block and is not directly programmable. However, the IOCU can be indirectly controlled using the following register fields:

- The read-only NUMIOCU field in bits 11:8 of the *Global Config* register (GCR_CONFIG) located at offset 0x0000 of CM GCR address space and indicates the number of IOCUs instantiated in the design. This field is filled by hardware during IP configuration.
- IOCU requests are prevented from being issued to MMIO regions by setting the bit 13 of the *Global CM Control* register (GCR_CONTROL) at offset 0x0010 in CM GCR address space.
- IOCU requests to external devices are counted toward the outstanding request limit when bit 12 of the *Global CM Control* register (GCR_CONTROL) at offset 0x0010 in CM GCR address space. If this bit is set, IOCU accesses to MMIO regions are blocked once the MMIO outstanding limit is reached. Note that bit 13 of this register must be 0 for this bit to have meaning as described above.
- Software can select which IOCUs are allowed to access the CM GCR registers by programming bits 23:16 of the *Global CSR Access Privilege* register (GCR_ACCESS) at offset 0x0120 in CM GCR address space. Each bit corresponds to one of eight IOCUs. If the corresponding bit is set, accesses from that IOCU are allowed to write the GCR and Cluster Power Controller (CPC) registers.

5.12 MMIO Address Regions

As described in the section entitled Verifying Overall System Configuration, the number of MMIO address regions is determined at IP configuration time. The I6500 supports up to four MMIO regions. Each region is assigned an upper and lower address bound.

The MMIO regions are intended to be used with communicating with external PCIe devices. The MMIO registers allow for counting of number of non-speculative code fetches of uncached requests in order to avoid potential deadlock condition by having too many requests outstanding. This is accomplished by programming the MMIO_REQ_LIMIT field.

5.12.1 CM GPR Register Interface

Software can set the number of MMIO requests that can be in-flight at any given time by programming the MMIO_REQ_LIMIT field of the MMIO Request Limit register (GCR_MMIO_REQ_LIMIT) at offset 0x6F8.

In addition, the address range of each MMIO region is defined using the Upper and Lower Bound MMIO region registers. A pair of registers are used for each MMIO region, with each register containing a 32-bit address bound value. These registers are located at:

- Lower bound of MMIO region 0 (GCR_MMIO0_BOTTOM) at offset 0x0700
- Upper bound of MMIO region 0 (GCR_MMIO0_TOP) at offset 0x0708
- Lower bound of MMIO region 1 (GCR_MMIO1_BOTTOM) at offset 0x0710
- Upper bound of MMIO region 1 (GCR_MMIO1_TOP) at offset 0x0718
- Lower bound of MMIO region 2 (GCR_MMIO2_BOTTOM) at offset 0x0720
- Upper bound of MMIO region 2 (GCR_MMIO2_TOP) at offset 0x0728
- Lower bound of MMIO region 3 (GCR_MMIO3_BOTTOM) at offset 0x0730
- Upper bound of MMIO region 3 (GCR_MMIO3_TOP) at offset 0x0738

5.12.2 MMIO Region Control

Each of the four MMIO regions listed above can be enabled or disabled by programming the MMIO_EN bit that resides in the Lower Bound register for each MMIO region (GCR_MMIO[0-3]_BOTTOM). If the MMIO region is enabled, then the request address and CCA are used to determine if the request falls into an MMIO Region. The decoded address is used to determine if the access is to a MMIO region as shown in the following equation:

MMIO_BOTTOM_ADDR[47:16] <= phys_address[47:16] <= MMIO_TOP_ADDR[47:16]

If bits 47:16 of the physical address fall between the value in MMIO_BOTTOM_ADDR[47:16] and MMIO_TOP_ADDR[47:16], then the access is to the corresponding MMIO region.

If MMIO_CCA is set to 0x0, just the request address is used to determine whether the request is to an MMIO region as shown above. If MMIO_CCA is set to 0x01, then the address comparison above is further qualified by whether the request has CCA = UC. In other words, only UC requests will be considered eligible to hit the MMIO region. If MMIO_CCA is set to 0x2, then the request is qualified by CCA = UCA. If MMIO_CCA = 0x3, then the request is qualified by CCA = UC or CC = UCA. In other words, either UC or UCA requests can match the MMIO region.

If an address hits in multiple MMIO register address regions, then the lowest-numbered enabled MMIO region hit takes precedence for determining which MMIO region the request matches. Once a request is determined to reside in an MMIO region, that region MMIO_PORT field in the Lower Bound register determines where the request will be routed. Options are the main memory port or an Auxiliary interface. See section 5.13.

The user can limit the total number of MMIO requests issued by the CM, which can be useful to avoid deadlock when accessing PCIe bridges that also service incoming coherent requests. The limit is defined by the MMIO_REQ_LIMIT field in bits 7:0 of the MMIO Request Limit (GCR_MMIO_REQ_LIMIT) register at offset 0x06F8 in GCR address space. Once the limit is reached, the CM stops serializing uncached and code fetches until a response to an MMIO request has been received. For example, a value of 0x01 in this field indicates one outstanding MMIO request is permitted. Setting this value to 0x00 disables the MMIO limiting feature, allowing any amount of outstanding requests to occur. The MMIO_DISABLE_REQ_LIMIT bit in the region's Lower Bound Register can be set to indicate that requests to the particular MMIO region should not be limited.

By default, IOCU uncached requests are never considered part of the MMIO limit (to allow for forward progress). However, this is controllable via the GCR_CONTROL.CM_MMIO_IOCU_ENABLE_REQ_LIMIT. When this bit is

set, IOCU uncached requests are counted as outstanding MMIO requests. In this case, IOCU uncached requests are blocked if the MMIO request limit has been reached.

5.13 Auxiliary Interfaces

The CM supports up to four non-coherent Auxiliary AXI4 buses, called AUX0 - AUX3. The AUX master ports are intended to be used for lower latency access to peripherals or instruction SRAM. Each cluster supports up to four AUX ports. Each AUX interface has a configurable data width. Values of 32, 64, 128, 256 and 512 are supported. The data width is determined during IP configuration. Each AUX address width is 48 bits. The number of AUX ports is stored in the 3-bit NUMAUX field of the *Global Configuration register* (GCR_CONFIG) at offset 0x0000 in GCR address space.

The clock for each AUX interface can be provided internally by the cluster or provided externally from outside the cluster. Each internally provided AUX clock can have an independent clock ratio. An externally provided clock can be provided on the external AUX clock pin. An externally provided clock is assumed to be asynchronous to the cluster. Selection between an internal versus external clock is done during IP configuration.

The AUX ports are memory mapped by the MMIO GCR control registers. There are up to 4 MMIO regions. Each GCR_MMIO<x>_BOTTOM register listed above contains an MMIO_PORT field in bits 5:2 that indicates which auxiliary port the request should be routed to. This field is encoded as shown in Table 5.5.

Table 5.5 Encoding of MMIO_PORT Field

Field Name	Register Bits	Encoding	Port Accessed
MMIO_PORT	5:2	0x0	Main memory
		0x8	AUX port 0
		0x9	AUX port 1
		0xA	AUX port 2
		0xB	AUX port 3

Chapter 6

Power Management

Power management in the I6500 Multiprocessing System is handled by the Cluster Power Controller (CPC). The I6500 CPC uses the concept of domains to manage both power and clocking throughout the device. Using registers, the programmer can enable or disable these domains in order to reduce overall power consumption.

The CPC implements two types of domains; power and clock. In each case, registers are instantiated on a per-domain basis so that the domain can be individually controlled by kernel software. This is true for each power domain and each clock domain.

- For the power domains, kernel software uses registers in the CPC to control the power to individual elements in the system such as cores, IOCU's, and the Coherence Manager (CM). The various power domains that can be individually controlled are defined in the section entitled Power Domains.
- For the clock domains, kernel software uses registers in the CPC to control the clock frequency to the individual elements in the system such as cores, IOCU's, Coherence Manager (CM), and memory. In addition to clock management for the various devices in the I6500 Multiprocessing System, the CPC also provides the ability to change the clock ratios in memory, and put the caches into a low-power state. The various clock domains that can be individually controlled are defined in the section entitled Clock Domains.

This chapter provides an overview of how power is managed in the I6500 Multiprocessing System and identifies the various power and clock domains the programmer can use to manage power consumption in the device. In addition, a procedure on how to set the CPC base address in memory is provided. Other programming principles include setting the device to coherent or non-coherent mode, requestor access of CPC registers, system power-up policy, programming examples of a clock domain change and clock delay change, powering up the CPC in standalone mode (no cores enabled), reset detection, VP run/suspend mechanism, local RAM shutdown and wakeup procedure, accessing registers in another power domain, and fine tuning internal and external signal delays to help the programmer easily integrate the device into a system environment.

6.1 Overview

This section provides an overview of the power and clock management schemes implemented in the I6500 Multiprocessing System.

6.1.1 Power Domains

Figure 6.1 shows the various power domains in the I6500 Multiprocessing System. Registers are instantiated for each power domain to allow for individual control. Note that in this figure, core 1 through core n are optional blocks depending on the system configuration.

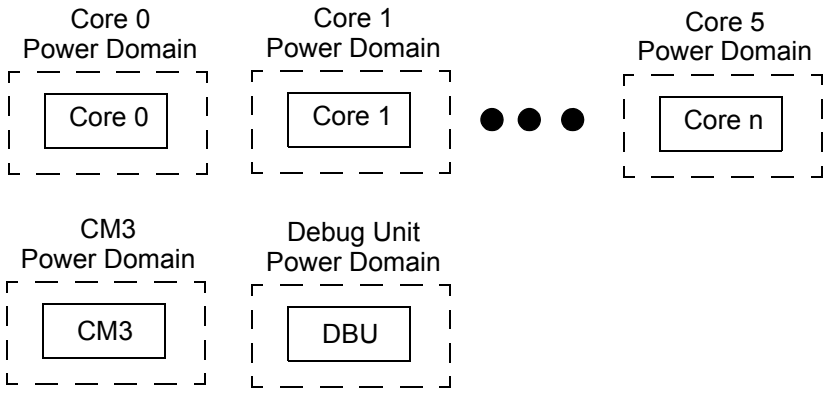


Figure 6.1 Power Domains in the I6500 Multiprocessing System

6.1.2 Clock Domains

Figure 6.2 shows the various clock domains in the I6500 Multiprocessing System. Each clock domain shown can be individually controlled using the CPC register interface.

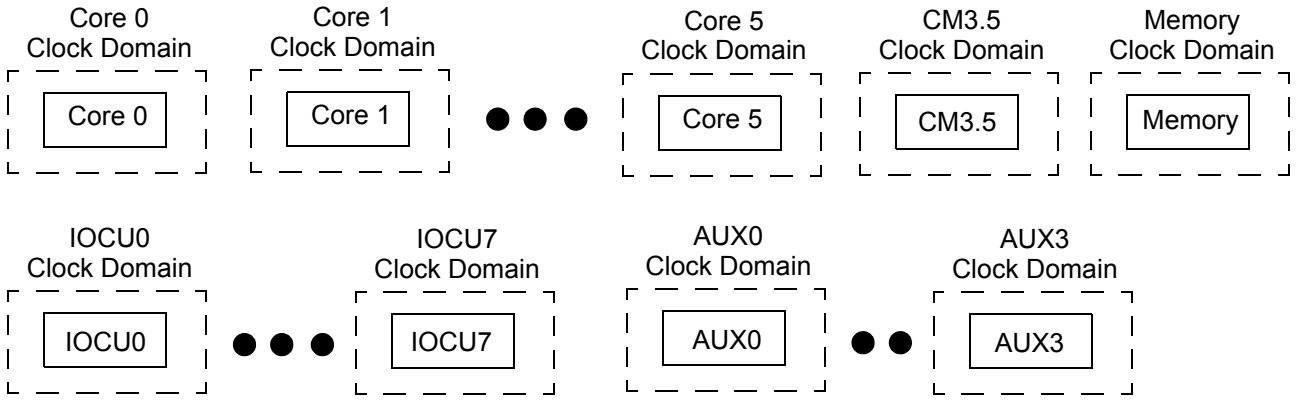


Figure 6.2 Clock Domains in the I6500 Multiprocessing System

6.1.3 Core and IOCU Selection

Figure 6.2 shows the maximum possible number of cores and IOCUs that can be instantiated into the I6500 MPS. However, the total number of cores and IOCUs cannot exceed eight. So for example, if there are two cores, there cannot be more than six IOCUs. If there are four cores, there cannot be more than four IOCUs, etc.

6.1.4 Overview of Power States

Each device in Figure 6.1, except the CM, contains its own set of Core-Local registers that can be used to independently place each device into one of the following four power states by programming the CMD field (bits 3:0) of the *CPC Local Command* Register. For more information on this register, refer to the *I6500 Registers* companion document included in the release.

Note that each command can only be executed in non-coherent mode. If a command is executed in coherent mode, the command is queued, but not processed by the CPC until the device has transitioned from coherent mode to non-coherent mode. For more information, refer to the section entitled Enabling Coherent Mode.

The states are as follows:

- **ClockOff** - a power domain is brought into *ClockOff* state when a value of 0x1 is programmed into the 4-bit CMD field of the *CPC_CL_CMD_REG* register. If the domain was powered down before, the power-on sequence is applied according to *CPC_CL_STAT_CONF_REG* settings. If the domain was active before and was in non-coherent operation, the power domain is brought into the *ClockOff* state. A domain in the *ClockOff* state can be sent into operation using the *PwrUp* command.

  A *ClockOff* command given to a domain in coherent operation remains inactive until the device has left the coherent mode of operation. Sending a *ClkOff* command to the CPC before a previous command has completed causes the CPC domain target to be redirected towards *ClockOff*. However, the previous steady state can be observed temporarily before the newly programmed state is reached. Refer to the section entitled Enabling Coherent Mode for more information on enabling and disabling coherence mode.

- **PwrDown**. A power domain is brought into *PwrDown* state when a value of 0x2 is programmed into the 4-bit CMD field of the *CPC_CL_CMD_REG* register. This command uses setup values in the *CPC_CL_STAT_CONF_REG* register.

  A *PwrDown* command given to a domain in coherent operation will remain inactive until the device has left the coherent mode of operation. Sending a *PwrDown* command to the CPC before a previous command has completed causes the CPC domain target to be redirected towards *PwrDown*.

- **PwrUp** - A power domain is brought into *PwrUp* state when a value of 0x3 is programmed into the 4-bit CMD field of the *CPC_CL_CMD_REG* register. This command uses setup values in the *CPC_CL_STAT_CONF_REG* register. The execution of this command depends on the previous domain power state. If the domain is in the powered-down state, a *PwrUp* command enables power for the domain, applies the clocks and reset, and brings the domain into an operational state.

- **Reset** - A power domain is brought into *Reset* state when a value of 0x4 is programmed into the 4-bit CMD field of the *CPC_CL_CMD_REG* register. This command allows a domain in the non-coherent operation to be reset. It also can be sent to a domain in power-down or clock-off mode. The domain will then become active, and a reset sequence is executed which leads to an operational steady state of the domain.

6.2 CPC Register Programming

This section describes some of the programming functions that can be performed via the CPC registers.

6.2.1 Cluster Power Controller Register Address Map

The CPC uses memory locations within the global, core-local, and core-others address space. The CPC location within the CPU address map is determined by the *GCR_CPC_BASE* register. All address locations in this document are relative to this base address.

In Table 6.1, all registers are accessed using 32-bit aligned uncached load/stores. In addition, the block offsets shown are relative to bits 31:15 of the *GCR_CPC_Base* register located in the CM3. Refer to Figure 6.3 for more information on how to use this register.

Table 6.1 CPC Address Map (Relative to GCR_CPC_BASE[31:15])

Block Offset	Size (bytes)	Description
0x0000 - 0x1FFF	8 KB	**Global Control Block**. Contains registers pertaining to the global system functionality. This address section contains a single set of registers that is visible to all CPUs.
0x2000 - 0x3FFF	8 KB	**Core-Local Control Block**. Aliased for each I6500 core. Contains registers pertaining to the core issuing the request. Each core has its own copy of registers within this block.
0x4000 - 0x5FFF	8 KB	**Core-Other Control Block**. Aliased for each I6500 core. This block of addresses gives each Core a window into another Core.

6.2.2 CPC Base Address

As mentioned above, the base address of the CPC registers is stored in bits 47:15 of the Cluster Power Controller Base Address register (GCR_CPC_BASE) located at offset 0x0088 in CM address space. The remaining bits (14:0) of the address are always zero to indicate that the CPC registers reside on a 32 KB boundary.

The following `init_cpc` code example is used to read the value in the GCR_CPC_BASE register and store it locally for future use.

LEAF(init_cpc)

The code uses the known value of the location of CPC within the system and writes that to the Cluster Power Controller Base Address Register. This is a physical address. Also, bit 0 is set, to enable the address region for the CPC.

```
      li a0, CPC_P_BASE_ADDR                // Locate CPC
      sd a0, GCR_CPC_BASE (r22_gcr_addr)    // GCR_CPC_BASE
```

Then the code stores this address for later use in r30_cpc_addr using the KSEG1 equivalent address, and is now done setting up the CPC.

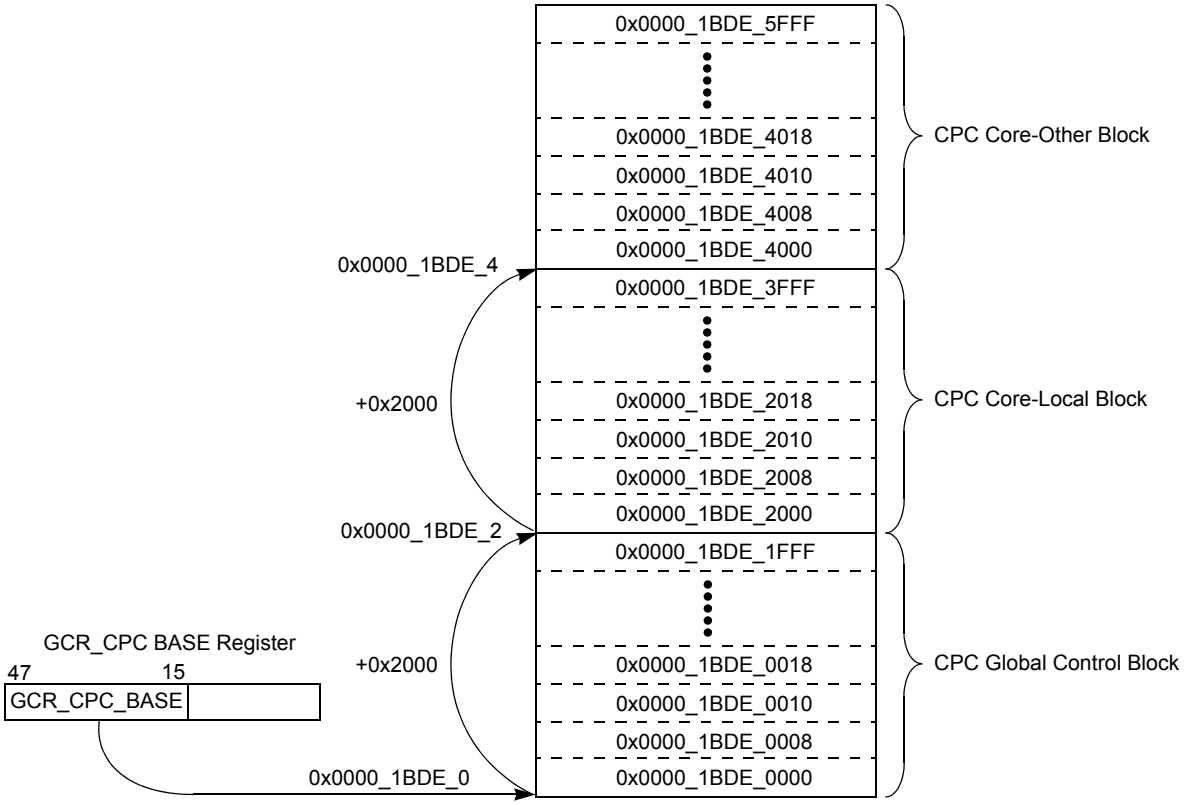
```
      li r30_cpc_addr, CPC_BASE_ADDR     // copy to register
```

This completes the CPS initialization and the code returns to start.

```
done_init_cpc:
      jr      ra
      nop
END(init_cpc)
```

This concept is described in Figure 6.3.

Figure 6.3 CPC Register Addressing Scheme Using an Example Base Address of 0x0000_1BDE_0



6.2.3 Global Control Block Register Map

All registers in the Global Control Block are 64 bits wide and should only be accessed using aligned 64-bit uncached load/stores. Reads from unpopulated registers in the CPC address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

For more information on these registers, refer to the *I6500 Registers* companion document.

6.2.4 Local and Core-Other Control Blocks

All registers in the *CPC Local Control Block* are 64 bits wide and should only be accessed using aligned 64-bit uncached load/stores. Reads from unpopulated registers in the CPC address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

A set of these registers exists for each core in the I6500 MPS. In the case of some CPC registers, a set of registers exists per power domain or per clock domain. These registers can also be accessed from other cores by first writing the GCR *Core-Local Redirect Register (GCR_CL_REDIRECT)* in the Core-Local Control Block of the CM.

For more information on these registers, refer to the CPC chapter in the *I6500 Technical Reference Manual* companion document.

6.2.5 Requestor Access to CPC Registers

Register Interface

The CPC allows up to eight requestor's in a system. A requestor can be either a core or an IOCU. The requestor may not have unrestricted access to the CPC registers. During boot time, the programmer determines which requestor's are provided access to the CPC registers by programming the *Global Access Privilege* register located at offset 0x120 in the CM register map. The 6-bit *ACCESS_EN* field (bits 5:0) of this register selects up to six cores, and bits 23:16 enable access for IOCU7 through IOCU0 respectively.

The MIPS default for *ACCESS_EN* field is 0x3F, meaning that all cores in the system have access to the CPC register set. In addition, bits 23:16 are set to allow IOCU7 through IOCU0 access to the CPC register set. To disable access to the registers for a particular requestor, kernel software need only clear the bit corresponding to that core or IOCU, and all write requests to the CPC registers by that requestor will be ignored.

For more information on this register, refer to the CPC register listed in the *I6500 Register* companion document included in the release.

6.2.6 Enabling Coherent Mode

The I6500 Multiprocessing System allows each power domain to be placed in either a coherent or non-coherent mode. Because the I6500 implements a directory-based coherence protocol, MIPS recommends that each domain be placed in coherent mode during normal operation. The non-coherent mode should only be used during boot-up and power-down. Software should not execute any cacheable memory accesses (instruction fetch or load/store) while coherence is disabled.

Register Interface

Coherency is enabled when hardware asserts the external *Coherence Enable* pin. The state of this pin is reflected in bit 11 (COH_EN) of the *Core-Local Status and Configuration* register. This register resides in the CM local register block at offset address 0x0008. There is one of these registers per power domain.

For more information on this register, refer to the *I6500 Registers* companion document included in the release.

Note that if a power domain is in coherent mode and a change to the power state is initiated, the caches must be flushed prior to disabling coherence mode.

Coherent Mode Enable Code Example

The base address for the location of the CM GCR registers is programmed into the CP0 CMGCRBase register. As a reference, a value of 0x0000_1FBF_8 is used (MIPS default) to indicate the base location of the CM global control registers. In this case, the base value is read from the CP0 register and an offset is added to it to derive the exact register address where the *Core Local Coherence Control* register is located.

By default, coherence is disabled in the I6500 MPS.

```
#define c0_CMGCRBASE          $15,3

mfc0   t1, c0_CMGCRBASE       // move contents of CP0 CMGCRBase register into t1
dsll   t1, t1, 4              // shift value in t1 left by 4 bits
li     t2, 0xA000_0000        // Assign KSeg1 base
```

```
or      t1, t2, t1              // Create VA from CGRBase
li      t0, 0x0000_0001         // Enable coherence
sd      t0, 0x2008 (t1)         // write value in t0 to the base address in t1 plus
                                // an offset of 0x2008 to access the Coherence Enable
                                // register.
```

6.2.7 Master Clock Prescaler

The clock prescaler is used to reduce the frequency of all devices in the system simultaneously.

CP0 Interface

The prescaler can be programmed as follows using the global *CPC Prescale Clock Change Control* register located at offset address 0x0048.

1. Verify that the *PRESCALE_CLK_RATIO_CHANGE_EN* bit of this register (bit 8) is set. This bit must be set before the *CLK_PRESCALE* field can be changed.

2. Optionally, the programmer can read the *PRESCALE_CLK_RATIO* field in bits 26:23 of this register to determine the current clock prescaler ratio.

3. Program the *CLK_PRESCALE* field (bits 7:0) to set the clock ratio. A value of 0x00 indicates a 1:1 clock ratio (no difference between input and output frequency of the prescaler). A value of 0xFF indicates a 1:256 ratio between the master input clock and the output of the prescaler.

The 8-bit CLK_PRESCALE field can be programmed as follows to select the prescaler ratio.

Table 6.2 Encoding of the CLK_PRESCALE Field

Encoding	Description
0x00	No prescaling
0x01	Divide input clock by 2
0x02	Divide input clock by 3
0x03	Divide input clock by 4
0x04	Divide input clock by 5
.....	.....
0xFD	Divide input clock by 254
0xFE	Divide input clock by 255
0xFF	Divide input clock by 256

For an example of how to program these fields, refer to step 1 of the procedure in Section 6.2.8.1, "Clock Domain Change Example — Register Programming Sequence".

For more information on this register, refer to the *CM Registers* companion document included in the release.

The base address for the location of the CPC registers is programmed into the CP0 CMGCRBase register. As a reference, a value of 0x0000_BBDE_0000 is used (MIPS default) to indicate the virtual address base location of the CPC registers.

By default, the clock prescaler is disabled in the I6500 MPS. In this example, the clock prescaler is enabled and the clock divide ratio is set to divide by 4. Note that the PRESCALE_CLK_RATIO field in bits 23:16 of this register is a read-only field that is updated by hardware and allows kernel software to quickly read this register to determine the current clock ratio. In this example this field is ignored.

```
li     t1, 0x0000_BBDE_0000    // move CPCBase register VA value into t1
li     t0, 0x0000_0103         // Enable clock prescaler and set divide ratio to 4
sd     t0, 0x48(t1)            // write value in t0 to the base address in t1 plus
                               // an offset of 0x48 to access the CPC Global Clock
                               // Prescale register.
```

6.2.8 Individual Device Clock Ratio Modification

Based on the input clock frequency to each individual device supplied by the clock prescaler, each device can further reduce the clock by a frequency range of 1:1 to 1:8, except for the CM, which can be programmed with a frequency ratio of either 1:1 or 1:2 relative to its input clock as shown in the figure. This is accomplished by programming the CLK_RATIO field (bits 2:0) of each *CPC Local Clock Change Control* register located at offset address 0x0018. For an example of how to program this field, refer to step 2 of the procedure in the section entitled Clock Domain Change Example — Register Programming Sequence.

6.2.8.1 Clock Domain Change Example — Register Programming Sequence

The following example shows how to run core 0 at full speed, and core 2 at quarter-speed to save power. Assume the following:

- 2-core system
- 1 VP per core
- si_ref_clk input frequency of 1 GHz
- Prescaler output of 1 GHz
- Core 0 input frequency of 1 GHz
- Core 1 input frequency of 250 MHz

In this example, the *si_ref_clk* input to the clock prescaler is 1 GHz. As shown above, the output frequency of the prescaler in this example is also 1 GHz. This ratio is accomplished by programming the global *CPC Prescale Clock Change Control* register located at offset address 0x0048 as follows. Note that this register is global and is seen by all cores and all individual devices (clock domains) in the system.

Register Interface

To program the clock prescaler for this example:

1. Write a value of 0x100 to the global *CPC Prescale Clock Change Control* register located at offset address 0x0048. This value sets the CLK_PRESCALE field to a value of 0x00, indicating a 1:1 relationship between the input clock and the output clock. This value also sets the PRESCALE_CLK_RATIO_CHANGE_EN bit to indicate that the value in the CLK_PRESCALE field is valid. Refer to the *I6500 Registers* companion document for more information on this register.

2. In this example the core 0 is running at full speed. Core 1 is running at 1/4 speed. To set the ratio of the clock generators for core 0 so it operates at 1 GHz, and core 1 so it operates at 250 MHz, program the individual *CPC*

Local Clock Change Control registers. This register is instantiated as one per clock domain, so in this case each core has its own register since each core is in its own domain.

3. Set the SET_CLK_RATIO bit in the *CPC Global Clock Change Control* register located at offset 0x0028 to initiate a clock change for all clock domains participating in the clock change, which is cores 0 - 3 in this example. This bit is cleared by hardware once the clock change has completed.

Table 6.3 shows the programming of the CLK_RATIO field (bits 2:0) of the corresponding *CPC Local Clock Change Control* register located at offset address 0x0018.

Table 6.3 Programming the CLK_RATIO Field of the CPC Local Clock Change Registers

Core	CLK_RATIO Value	Clock Ratio	Core Clock Frequency
0	3'b000	1:1	1 GHz
1	3'b100	4:1	250 MHz

Poll the following registers to determine when the clock change has completed.

- Read the CPC_CC_CTL_REG register to determine when bit 8 (SET_CLK_RATIO) is 0. If SET_CLK_RATIO is 1, the change request is still pending.
- Read the CPC_CC_CTL_REG to determine when bit 10 (CLK_CHANGE_ACTIVE) is 0. If CLK_CHANGE_ACTIVE = 1, the clock change is in progress.
- When both of these bits are zero, the clock change has completed. At this point, another clock change could be requested.

Clock Ratio Change Code Example

```
#define c0_CMGCRBASE    $15,3

li t0, GCR_BASE_ADDR          // move GCRBase value into t0

// Store VA for CPCBase register into t1

li t1, CPC_BASE_ADDR          // move CPCBase value into t1

// Set the clock prescaler divide ratio to 1:1 in the CPC_PRESCALE_CC_CTL register
li     t2, 0x0000_0100        // enable clock prescaler and set divide ratio to 1:1
sd     t2, 0x48(t1)           // write value in t2 to the base address in t1 plus
                              // an offset of 0x48 to access the CPC Global Clock
                              // Prescale register.

// Set the core number to 0 in the GCR_CL_REDIRECT register
li     t2, 0x0000_0000        // set CORE number to 0 and VP number to 0
sd     t2, 0x2018 (t0)        // store contents to GCR_CL_REDIRECT register at
                              // 0x2018 from GCRBase
sync

//Program the CPC_CO_CC_CTL register CLOCK_RATIO field to 0 (1:1 ratio)
li     t2, 0x0000_0100        // enable clock change and set ratio to 1:1
sd     t2, 0x4018 (t1)        // store contents to CPC_CO_CC_CTL register at 0x4018

// Set the core number to 1 in the GCR_CL_REDIRECT register
li     t2, 0x0000_0001        // set CORE number to 1 and VP number to 0
```

```
sd     t2, 0x2018 (t0)         // store conents to GCR_CL_REDIRECT register at
                               // 0x2018
sync

//Program the CPC Local Clock Change register CLOCK_RATIO field to 3 (4:1 ratio)
li     t2, 0x0000_0103         // enable clock change and set ratio to 4:1
sd     t2, 0x4018 (t1)         // store contents CPC_CO_CC_CTL register at 0x4018

// Initate register based clock change - program bit 8 of the CPC_CC_CTL_REG
// register at offset 0x0018 from CPCBase
ld     t2, 0x0028 (t1)         // load CPC_CC_CTL_REG register in to t2
ori    t2, t2, 0x100           // set the SET_CLK_RATIO bit in the CPC_CC_CTL REG
                               // register - logically OR bit 8 with t2 and copy
                               // back into t2. This sets the clock change enable
sd     t2, 0x0028 (t1)         // store new value in t2 back to the CPC_CC_CTL reg

Loop:
// Poll CPC_CC_CTL_REG clock change control register until bits 8 and 10 are low.
ld     t2, 0x0028 (t1)         // read contents of CPC_CC_CTL_REG into t2
andi   t2, t2, 0x0500          // AND t2 and 0x0500, copy result into t2
bne    t2, r0, loop            // loop until bits 8 and 10 are low, indicating a
                               // successful clock change
nop
```

6.2.8.2 Clock Change Delay

The *CPC_CC_CTL_REGCC_DELAY* field in bits 29:20 of the *CPC Global Clock Control* register is used to optimize the amount of delay during a clock change. This can be done if all clock domain ratios are low. For example, if all current clock ratios are less than 1:4 the value of the delay could be reduced. The intent is that clock domain changes do not happen very often, so setting the default of 80 clocks should not be a problem and leaving this value at its default delay is recommended. This register could also be used to extend the state delay period if desired.

6.2.9 CM Standalone Powerup

Normally, the CM is automatically powered-up if any core is powered-up. Conversely, the CM is automatically powered-down if all cores are powered-down. The I6500 allows for the CM to be powered-up even if no core is powered-up. This is useful for system debug/setup via the DBU.

Register Interface

This functionality is controlled by the CPC Global Power Up register (CPC_PWRUP_CTL_REG) located at offset address 0x0030.

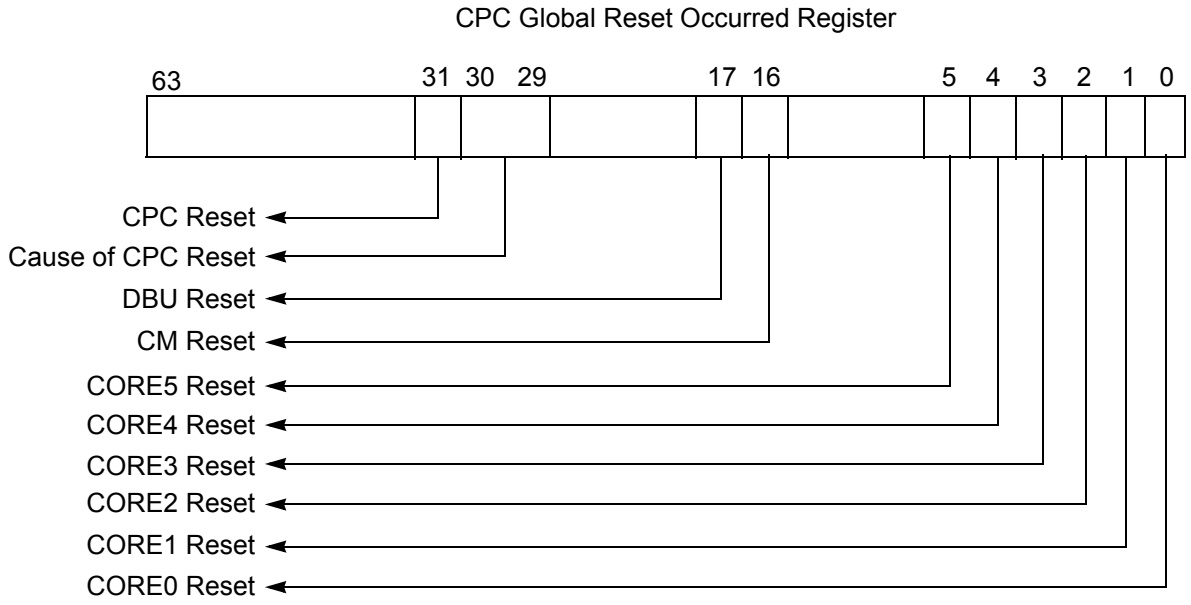
The DBU may execute a one-time power-up of the CM by writing a 1 to this register. If the CM is not operational at the time this bit is set by the DBU, it will transition from its current state to an operational state. If the CM is already operational, setting this bit has no meaning and the register write is ignored.

6.2.10 Reset Detection

The CM provides a series of read-only bits that allow the programmer to determine when a given device connected to the CM has been reset, including the CPC itself. Whenever a device is reset, the corresponding bit of the *CPC Global Reset Occurred* register (CPC_ROCC_CTL_REG) at offset 0x0040 is set. Refer to the *I6500 Registers* companion document included in the release for more information on this register.

In addition to the reset detection, this register also contains a 2-bit field (RESET_CAUSE) that indicates the type of reset for the CPC block. Reset options are cold reset, external warm reset, and watchdog timer reset. The functionality of this register is shown in Figure 6.4.

Figure 6.4 Reset Detection in the I6500 Multiprocessing System



6.2.11 VP Run/Suspend

Three registers are used to control the power state of each VP in the system. The I6500 Multiprocessing system supports up to four VP's per core, and up to six cores per system. Each of these registers is instantiated per core.

Three registers are used to control this functionality:

- *VP Run* register (WO)
- *VP Stop* register (WO)
- *VP Running* register (RO)

Register Interface

The *VP Run* register is a Write-only register used to set each VP to the run state. The *VP Run* register contains a 4-bit field, where each bit is dedicated to a particular VP, up to four. Prior to setting one of these bits, kernel software must ensure that the VP in question is not already running by reading the corresponding bit in the *VP Running* register. If a given bit in the *VP Running* register is cleared, setting the corresponding bit in the *VP Run* register places the VP in the run state. If a given bit in the *VP Running* register is already set, setting the corresponding bit in the *VP Run* register has no meaning. The value in this register is reset whenever the associated core is reset. The *VP Run* register can also be cleared by hardware, as well as the Debug unit.

The *VP Stop* register is a write-only register used to stop a VP. If a given bit in the *VP Running* register is set, setting the corresponding bit in the *VP Stop* register places the VP in the suspend state. Writing a 0 to any of the bits in the *VP Stop* register has no effect.

The *VP Running* register is a read-only register that indicates the run state of each VP in a given core. These bits are set and cleared by hardware based on the programming of the *VP Run* and *VP Stop* registers by kernel software as described above.

Note that for each of these registers, the four VP's correspond to the register bits as follows:

- Bit 0 = VP0
- Bit 1 = VP1
- Bit 2 = VP2
- Bit 3 = VP3

For example, to set VP2 of a given core to the Run state, kernel software would do the following,

1. Read bit 2 of the *VP Running* register. If this bit is already set, VP2 is already running and no action need be taken.

2. If bit 2 of the *VP Running* register is cleared, indicating that VP2 is in the Suspend state, kernel software sets bit 2 of the *VP Run* register to set VP2 to the Run state.

To set VP2 of a given core to the Suspend state, kernel software would do the following,

1. Read bit 2 of the *VP Running* register. If this bit is already cleared, VP2 is already in the Suspend state and no action need be taken.

2. If bit 2 of the *VP Running* register is set, indicating that VP2 is in the Run state, kernel software sets bit 2 of the *VP Stop* register to set VP2 to the Suspend state.

6.2.12 Local RAM Deep Sleep / Shutdown and Wakeup Delay

The CM allows the local RAM's within a given power domain (cores, CM, IOCU, etc) to be placed into either Shutdown mode where the clocks are turned off, or Deep Sleep mode where the clocks are running at a fraction of their normal frequency. This functionality is controlled through the *CPC Local RAM Sleep Control* register (*CPC_CL_RAM_SLEEP*) located at offset 0x0050 (or 0x2050 relative to the CPC base address).

This register is instantiated per power domain, so each domain has the ability to power cycle its own local RAM devices.

6.2.12.1 RAM Deep Sleep Mode

When bit 31 (RAM_DEEP_SLEEP_DISABLE) of the *CPC_CL_RAM_SLEEP* is cleared (logic '0'), the RAM's on the local device enter the Deep Sleep low power state when the CPC power state for the device reaches the ClockOff state. In this state the clocks to the local RAM's within that power domain are running at a fraction of their normal frequency.

The CPC also provides a way to delay the transition from the deep sleep state to the run state using bits 23:16 RAM_DEEP_SLEEP_WAKEUP_DELAY) of the *CPC_CL_RAM_SLEEP* register. Once awoken, the CPC delays the transition to the run state by the value programmed into this field in order to provide sufficient time for the RAMs to wake up from Deep Sleep. The delay can range from 1 to 255 (0xFF) clocks.

6.2.12.2 RAM Shut Down Mode

When bit 15 (RAM_SHUT_DOWN_DISABLE) of the *CPC_CL_RAM_SLEEP* is cleared (logic '0'), the RAM's on the local device enter the Shutdown low power state when the CPC power state for the device reaches the PwrDwn state. In this state the clocks to the local RAM's within that power domain are off. The RAM's remain in the Shutdown low power state even if the CPC power state changes to ClkOff without transitioning to the operational state.

The CPC also provides a way to delay the transition from the shutdown state to the run state using bits 7:0 RAM_SHUT_DOWN_WAKEUP_DELAY) of the *CPC_CL_RAM_SLEEP* register. Once awoken, the CPC delays the transition to the run state by the value programmed into this field in order to provide sufficient time for the RAMs to wake up from the Shut Down state. The delay can range from 1 to 255 (0xFF) clocks.

6.2.13 Accessing the CPC Registers in Another Power Domain

Each power domain shown in Figure 6.1 contains its own set of CPC Core-Local and Core-Other registers. This allows master devices such as a core or IOCU to access these registers to modify the power parameters for a given domain. This is accomplished by writing to registers within the CM address space using the Core number and the VP number of the device to be accessed.

For more information on accessing the CPC registers of another core or VP, refer to the section on *Core-Local and Core-Other Register* usage in the *CM Programming* chapter of this manual.

6.2.14 Fine Tuning Internal and External Signal Delays

This section describes those register fields that can be used to delay the assertion of external signals relative to one another, as well as the internal domain sequencer state machine. These registers are used to help accommodate a wide variety of timing constraints in the system. Signals can be lengthened or shortened accordingly in order to meet system timing.

6.2.14.1 Global Sequence Delay Count

The Sequence Delay register (*CPC_SEQDEL_REG*) located at offset 0x0008 in the CPC Global Control Block, contains a 10-bit MICROSTEP field that describes the number of clock cycles each domain sequencer state machine will take to advance to the next state.

The 10-bit MICROSTEP field contains a default value of 0x002, indicating a 2-cycle delay. However, should additional delay be required based on the system implementation, this register provides the programmer with the ability to increase the sequence delay as necessary.

Domain sequencing begins once the RAILDELAY field has counted down to zero. Refer to the section entitled Rail Delay for more information.

The 10-bit MICROSTEP field is encoded as follows:

Table 6.4 Encoding of MICROSTEP Field

Encoding	Description
0x000	1-cycle delay
0x001	2-cycle delay
0x002	3-cycle delay
0x003	4-cycle delay

Table 6.4 Encoding of MICROSTEP Field

Encoding	Description
0x004	5-cycle delay
.....	.....
0x3FD	1022-cycle delay
0x3FE	1023-cycle delay
0x3FF	1024-cycle delay

6.2.14.2 Rail Delay

The Rail Delay register (*CPC_RAIL_REG*) located at offset 0x010 in the CPC Global Register Block contains a 10-bit counter field (*RAILDELAY*) used to schedule delayed start of power domain sequencing after the *RailEnable*[1] signal has been activated by the CPC. This allows the CPC to compensate for slew rates at the gated rail.

The 10-bit counter value (*RAILDELAY*) delays the power-up sequence per domain. The power-up sequence starts after *RAILDELAY* has been loaded into the internal counter and a count-down to zero has concluded. At IP configuration time, the contents of the *CPC_RAIL_REG* register are preset. However, for fine tuning, the register can be written at run time.

The 10-bit RAILDELAY field is encoded as follows:

Table 6.5 Encoding of RAILDELAY Field

Encoding	Description
0x000	1-cycle delay
0x001	2-cycle delay
0x002	3-cycle delay
0x003	4-cycle delay
0x004	5-cycle delay
.....	.....
0x3FD	1022-cycle delay
0x3FE	1023-cycle delay
0x3FF	1024-cycle delay

The default value for this register has been determined by MIPS as the value that should work in the majority of system implementations. As such, this value should not need to be changed. However, should a problem arise where additional delay is required in order to meet system timing, this register provides the programmer with the ability to increase the delay as necessary.

For more information on how this counter is used, refer to the *Global Sequence Delay Count* section in the System Integration chapter of the *I6500 Integrator's Guide* for more information.

1. This signal is shown only for illustration purposes. Refer to the *I6500 Integrator's Guide* for the exact name and usage of this signal.

6.2.14.3 Reset Delay

During the power-up sequence, reset is applied. Typically, reset is active until the domain responds by asserting the internal *Reset_Hold* signal. However, the *Global Reset Width Counter* register (*CPC_RESETLEN_REG)* at offset 0x0018 allows reset to be extended beyond the assertion of *Reset_Hold*. A series of down-counters are used to delay various reset pins used to boot the CM as described in the following subsections.

The default value for this register has been determined by MIPS as the value that should work in the majority of system implementations. As such, this value should not need to be changed. However, should a problem arise where additional delay is required in order to meet system timing, this register provides the programmer with the ability to increase the delay as necessary.

For more information on these counters and the corresponding hardware signals that can be delayed, refer to the *Reset Delay* section in the I6500 Integrator's Guide for more information.

Programming the Global Reset Width Counter Register (RESETLEN)

The RESETLEN down counter is used to extend the various reset signals using bits 9:0 of the *CPC Global Reset Width Counter Register* (*CPC_RESETLEN_REG*) at offset 0x0018. This register field is programmed with a delay value between 1 and 1024 clock cycles as shown in Table 6.6.

Table 6.6 Encoding of the RESETLEN Field

Encoding	Description
0x000	1-cycle delay
0x001	2-cycle delay
0x002	3-cycle delay
0x003	4-cycle delay
0x004	5-cycle delay
.....	.....
0x3FD	1022-cycle delay
0x3FE	1023-cycle delay
0x3FF	1024-cycle delay

Programming the Global Reset Release Register — Core Reset Release (RESREL1)

The output of the RESETLEN counter described above is used to load a secondary internal counter with the value programmed into the RES_REL_LEN field of the CPC Global Reset Release Register (*CPC_RES_REL_REG*) located at offset 0x0038. This register is used to determine the amount of delay between the time the configuration signals are stable at the respective core(s), and the time that the core reset is released.

Bits 9:0 of this register (RES_REL_LEN) are programmed with a delay value between 1 and 1024 clock cycles. The encoding of this field is identical to the RESETLEN field shown in Table 6.6. Once this counter reaches 0, the *Domain_Reset_n*[2] signal is deasserted to the core(s), allowing them to come out of reset.

2. This signal is shown only for illustration purposes. Refer to the *Global Sequence Delay Count* section of the I6500 Integrator's Guide for more information on the usage of this signal.

Programming the Global Reset Release Register — Domain Ready (RESREL2)

The output of the RESREL1 counter is used to load a third internal counter (RESREL2) with the value programmed into the RES_REL_LEN field of the CPC Global Reset Release Register (*CPC_RES_REL_REG*) located at offset 0x0038. This register is used to determine the amount of delay between the time the *Domain_Reset_n* signal is deasserted, and the deassertion of the *Domain_Ready* signal, indicating that the core is ready to begin execution. Note that the same register field (RES_REL_LEN) of the *CPC_RES_REL_REG* register is used to load both the RESREL1 and RESREL2 counters.

The third internal counter (RESREL2) requires that the RESREL1 counter has reached zero before counting can begin. Once the RESREL2 counter reaches 0, the *Domain_Ready* signal is asserted to the core(s), allowing the core to begin execution.

For more information on how these counters are loaded and the signals affected once the counts reach zero, refer to the *Global Sequence Delay Count* section in the *System Integration* chapter of the *I6500 Integrator's Guide*.

Chapter 7

Global Interrupt Controller

The Global Interrupt Controller (GIC) processes internal and external interrupts in the I6500 Multiprocessing System. Each cluster in the system supports up to 256 external interrupts in multiples of 8, as well as numerous internal interrupts. The GIC is responsible for mapping each internal and external interrupt to any VP within the I6500 MPS for servicing.

7.1 Overview

External events are defined as those that originate outside of the I6500 Multiprocessing System and require servicing to determine where they originated from and how they can be resolved. Internal events are those that occur within the I6500 Multiprocessing System. Internal events can include performance counters, watchdog timers, software, and Fast Debug Channel (FDC).

Interrupts are events which interrupt program flow and require servicing to determine the type of and reason for the event. Events are categorized by priority. High priority events are those which require immediate attention. These events are handled before the lower priority events. The servicing of these events is accomplished through an interrupt service routine, which is a piece of software that resides in memory. Each time an interrupt event is detected, the program flow is interrupted and the code branches to the interrupt service routine. This routine is specifically designed to deal with the interrupt event.

This chapter describes how to program the various elements of the GIC using both register examples and code examples. Some of these elements include setting the operating mode, setting up the address map, GIC register layout and distribution, setting the GIC base address, determining the number of external interrupts, and configuring individual interrupt sources.

7.1.1 GIC Virtualization

The I6500 Multiprocessing System incorporates virtualization into the interrupt control system, allowing separate interrupt controllers for guest and root processes. This chapter contains information on virtualization as it relates to interrupts and the programming of the GIC-related Root and Guest registers. Refer to the chapter on Virtualization in this manual for more information.

7.1.2 GIC Operating Modes

The GIC supports two types of operating modes:

- Non-EIC mode
- External Interrupt Controller (EIC) mode

7.1.2.1 Non-EIC Mode

The non-EIC mode includes both Compatibility mode, the most basic type of interrupt mode, and also Vectored Interrupt (VI) mode. The main difference is that in VI mode, each interrupt type is assigned its own location in memory,

whereas in Compatibility mode all interrupts go to the same vector and kernel software determines the source of the error. In addition, for both Compatibility and VI modes, the six interrupt pins on the VP are used as individual interrupts, and are not an encoded value as in the EIC mode.

7.1.2.2 EIC Mode

The EIC mode provides support for up to 63 individual interrupts by encoding the value on the six interrupt pins of each VP. The GIC is responsible for encoding the proper vector number onto the six interrupt pins prior to driving the value to the appropriate VP.

Software can enable EIC mode by setting the *GIC_VL_CTL.EICMODE* read-write register bit. Refer to the *I6500 Registers* companion document for more information on this register. The state of the *GIC_VL_CTL.EICMODE* bit is driven onto the **SI_EICPresent** pin. Hardware then uses the state of this pin to set or clear the CP0 *CONFIG3.VEIC* bit. Therefore, if kernel software changes the state of this bit, the change is reflected in the CP0 *CONFIG3.VEIC* bit. The *GIC_VL_CTL.EICMODE* bit allows kernel software to boot up in non-EIC mode, then switch to EIC mode once the appropriate interrupt connections have been established through the GIC.

7.1.3 GIC Register Types

The GIC address space is accessed with uncached load/store commands. The physical address and the VP number of the requester is supplied for each load/store command. The VP number is used as an index to reference the appropriate subset of the instantiated control registers. By using the VP number information, the hardware writes/reads the correct subset of the control registers pertaining to that VP. Software does not need to explicitly calculate the register index for the core in question. This done entirely by hardware.

Two address “windows” are made available to the programmer:

- A window for the “Local” VP (as specified by the VP number information).
- A second window for an “Other” VP that allows that VP to access the register set belonging to another VP. The “Other” VP is specified by first writing the *VP_REDIRECT* field to the desired VP target and setting the *GIC_REDIRECT_EN* field to 0x1. Both of these fields are in the *VP-Local GCR Redirect* register

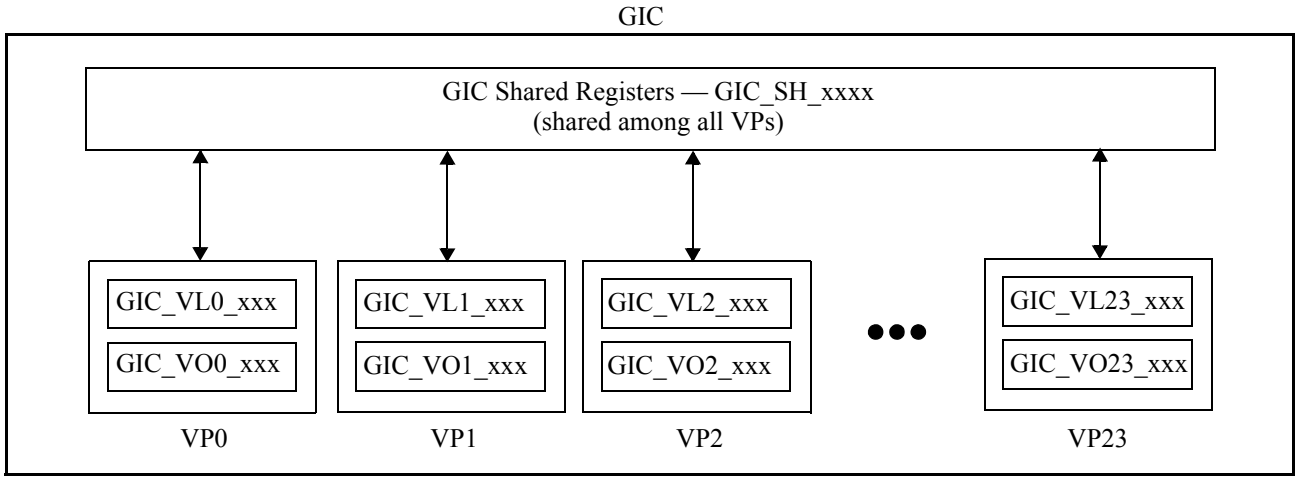
In the I6500 Multiprocessing System, any VP can access the registers of any other VP by using the VP-*Other* address spaces. Software must write the *VP-Local GCR Redirect* register located at offset 0x0018 (physical address of 0x1FBF_A018) before accessing these address spaces. Set the *REDIRECT_VP* field to select the correct the desired VP subset of registers and set the *GIC_REDIRECT_EN* bit to 1 to indicate that a *GIC VP-OTHER* access should be redirected.

The value of this register is used by hardware to index the appropriate subset of the control registers. For more information, refer to the *GCR Redirect* (GCR_CL_REDIRECT) register in the *I6500 Registers* companion document. Also in this manual, an additional section called the *User-Mode Visible Register* is used to give quick user-mode read access to specific GIC registers. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers.

7.1.4 GIC Register Distribution

The GIC contains various register blocks described above that are located at different address spaces as shown in Figure 7.2. Some of the registers are shared between all VP’s in the system, while other registers are local to a particular VP. This relationship is shown in Figure 7.1.

Figure 7.1 GIC Register Distribution



7.1.5 GIC Address Space Configuration

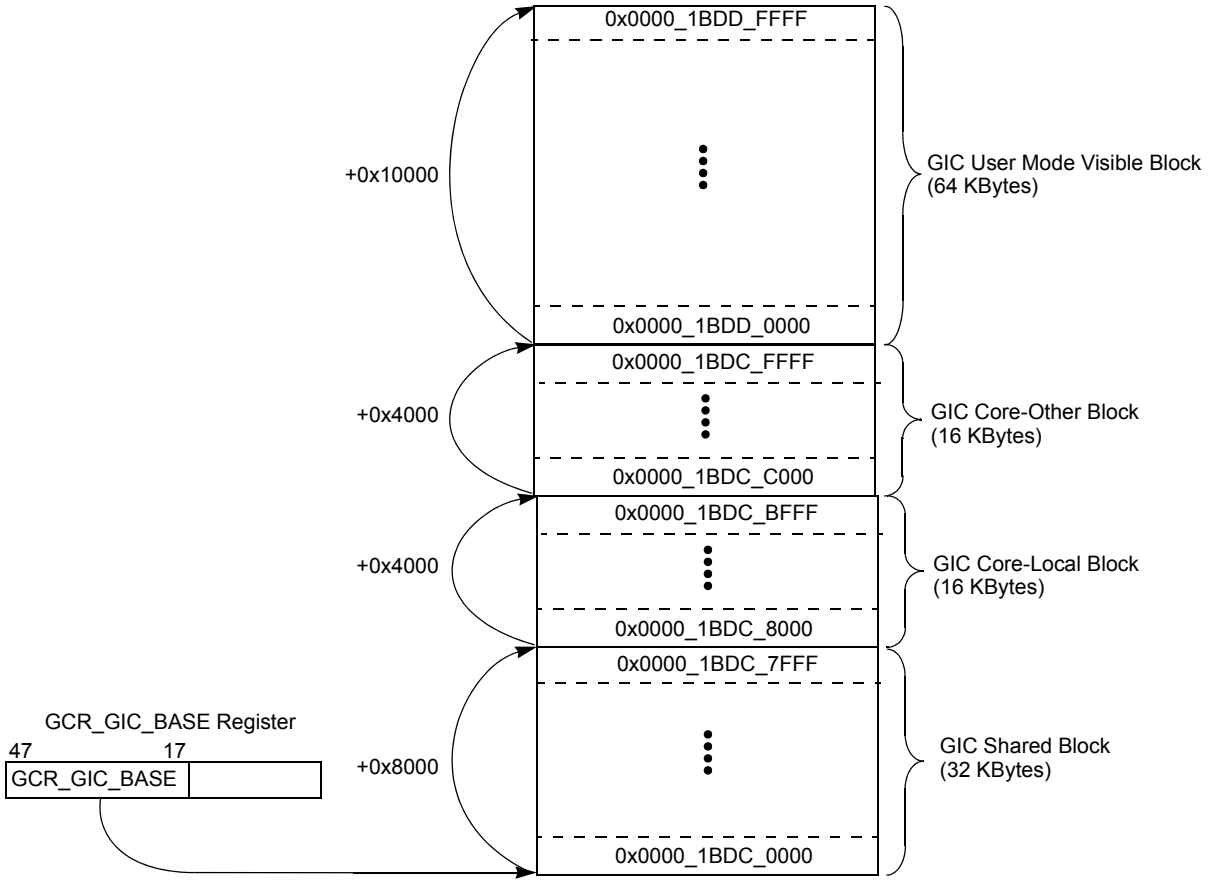
The GIC address space is divided into four blocks:

- A 32 KByte *Shared* section in which the external interrupt sources are registered, masked, and assigned to a particular VP and interrupt pin. This section is shared by all VPs and all cores in the system.
- A 16 KByte VP-*Local* section in which interrupts local to a VP are registered, masked, and assigned to a particular interrupt pin. If External Interrupt Controller Mode (EIC) mode is used for a particular VP, the EIC encoder is instantiated here.
- A 16 KByte *VP-Other* section in which the local VP can access the VP-Local section of another VP. by which the interrupt can be registered, masked, and assigned to a particular interrupt pin of another VP. Using the VP-other segment, the "local" VP can access the registers of another VP by using the VP-Other address space. Software must write the *VP-Local GCR Redirect* register located at offset 0x0018 (physical address of 0x1FBF_A018) Register before accessing these spaces. The value of this register is used by hardware to index the appropriate subset of the control registers for the other core(s). One VP can setup the GIC for all VPs in the system using this section.
- A 64 KByte *User Mode Visible* section that contains aliases for GIC registers that are read so often that it makes sense to make them available to user-mode programs without requiring a system call. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers. Currently, the only register aliased into this space is the *GIC_SH_COUNTER* register. Refer to Figure 7.1.2 for more information.

In the GIC, the *Shared*, VP-*Local*, and VP-*Other* sections are meant to be located in privileged system virtual address space, in which only kernel mode software can initialize and update the interrupt controller.

Figure 7.2 shows the mapping of the GIC registers. In this figure an example base address of 0x1BDE_0 is used. Each register is mapped using the GIC base address, the register block, and the corresponding register offset within that block. Refer to the *I6500 Technical Reference Manual* for more information on the derivation of these addresses.

Figure 7.2 GIC Register Addressing Scheme Using an Example Base Address of 0x1BDC_0



7.2 GIC Programming

This section covers the programming for the following tasks.

- Setting the GIC Base Address and Enabling the GIC
- Configuration of interrupt sources
- External interrupt source configuration
    - Level Sensitivity, active high or active low
    - Edge Sensitivity, dual or single edge (falling or rising)
- Routing of external interrupts to specific processors
- Enabling or disabling interrupts
- Inter-Processor Interrupts (IPI)
- Local device interrupt configuration

7.2.1 Setting the GIC Base Address and Enabling the GIC

Register Interface

The GIC base address is the starting address of the GIC memory-mapped registers. The GIC base address is a 31-bit value that is programmed into bits 47:17 of the *GCR_GIC_BASE* field in the *GCR GIC Base* register. This register is located at offset address 0x0080 in the Global Control Block of the CM registers. Refer to the *GCR_GIC_BASE Register* in the *Coherence Manager* chapter for more information on this register.

The following code example determines the base address of the GCR_GIC_BASE register in CM address space, then loads the physical GIC base address into the register and sets the GIC enable bit.

Setting the GIC Base Address Code Example

The following code example uses the following defines to make the code easier to read:

```
#define GCR_CONFIG_ADDR 0xffffffffbfbf8000  // KSEG1 address of the GCR registers
#define GCR_CONFIG_ADDR_PB 0xffffffffbfbf8000  // Post Boot address of the GCR
registers
#define GIC_P_BASE_ADDR0x000000001bdc0000  // physical address of the GIC
#define GIC_BASE_ADDR0xffffffffbbdc0000  // KSEG1 address of the GIC
#define GIC_BASE_ADDR_PB0xffffffffbbdc0000// Post Boot address of the GIC
#define NUMINTERRUPTS 16
#define NUMINTERRUPTS_S 8
```

The code loads the address of the GIC Base Address Register into `a1` using the `li` instruction.

```
li      a1, GCR_CONFIG_ADDR + GCR_GIC_BASE
```

The code then loads a0 with the physical address of the GIC using the `li` instruction. Then bit 0 is set, which enables the GIC. This value is stored to the GCR_GIC_BASE register using the `sw` instruction.

```
li      a0, (GIC_P_BASE_ADDR | 1) // Physical address + enable
sw      a0, 0(a1)
```

7.2.2 Determining the Number of External Interrupts in the System

Register Interface

The number of external interrupt sources is a fixed value configured at build time. This number of external interrupts in the system is stored in the "GIC Configuration Register", *GIC_SH_CONFIG*. For more information, refer to the *Global Configuration Register* (GIC_SH_CONFIG at offset 0x0000) in the *I6500 Registers* companion document contained in the release.

The following is a code example used to determine the number of external interrupts in the system. This code reads the *GIC_SH_CONFIG* register and isolates the NUMINTERRUPTS field in bits 23:16. Interrupt sources are configured in the core in groups of 8. This field indicates how many groups of 8 the core has.

The define GIC_BASE_ADDR is the address of the Shared section of the GIC which is loaded into `a1`. The Shared Configuration register is located at offset 0. The code loads the value of the register into `a0`. This examples assumes there are 40 external interrupts in the system.

Number of External Interrupts Code Example

// Verify GIC is 5 "slices" of 8 interrupts giving 40 interrupts.

```
li      a1, GIC_BASE_ADDR           // load GIC KSEG0 Address
lw      a0, GIC_SH_CONFIG (a1)      // GIC_SH_CONFIG
```

Then the code extracts the number of interrupt groups.

```
ext a0, NUMINTERRUPTS, NUMINTERRUPTS_S    //extract NUMINTERRUPTS
```

For this example, the code loads the expected value of NUMINTERRUPTS into `a3`. This example is expecting 40 interrupt sources (4 + 1 times 8). If the code does not detect this value, it executes a debug breakpoint to stop at a point where a debug probe can be used to evaluate the problem.

```
li      a3, 4
bqe     a0, a3, configure_slices
nop
sdbbp   // Failed assertion of 40 interrupts
```

7.2.3 EIC Mode Setting

Register Interface

EIC mode is controlled through kernel software by setting the EIC_MODE bit in the Local interrupt Control Register, GIC_ VPi_CTL. Setting this bit enables EIC mode for that VP. This bit defaults to 0, vectored interrupt mode. For more information, refer to the *Local Interrupt Control Register* (GIC_VL_CTL at offset 0x0000) in the *I6500 Registers* companion document included in the release.

Note that the state of the EIC_MODE bit is driven onto the **SI_EICPresent** pin. Hardware uses this pin to update the state of the CP0 *Config3.VEIC* bit to indicate support for and status of the EIC mode.

Note that the interrupt mode is a system wide setting that is determined during IP configuration time. The GIC and cores of the system are programmed by hardware accordingly to enable or disable this mode.

EIC Mode Setting Code Example

```
// Read CTL local

li a1, GIC_BASE_ADDR          // load GIC KSEG0 Address
daddiu a1, a1, 0x08000        // add offset for local addressing space
lw a0, GIC_VL_CTL(a1)         // read GIC_VL_CTL

// Set EIC_MODE (bit 0) in GIC_VL_CTL and read CP0 Config3

li a2, 0x1
ins a0, a2, 0, 1              // set bit 0
sw a0, GIC_VL_CTL(a1)         // write GIC_VL_CTL
mfc0 a2, $16, 3               // read Config3 (reg 16, select 3)
```

7.2.4 Configuring Interrupt Sources

The triggering of interrupts is configured through several registers in the GIC that are shared by all processors. While all processors can access these registers, in practice they are usually programmed at boot time by processor 0, or by

the boot code for the operating system (OS). There are three register groups that control the interrupt triggering configuration.

- Trigger type register group
- Edge type register group
- Polarity register group

Each interrupt source is represented by one bit in each register group. Each register in a group is 64 bits so each register controls 64 interrupt sources. The first register in each group would control interrupts 63:0, the next 127:64, and so on. Since there can be 256 interrupt sources there are 4 registers in each group. Interrupts are allotted in groups of 8, from 16 to 256.

Each of the interrupt sources can have either positive (asserted high) or negative (asserted low) polarity. Similarly, any of these sources can be either level-sensitive, single-edge-sensitive, or dual-edge-sensitive using the polarity control registers (*GIC_SH_POLx_y*), the trigger type control registers (*GIC_SH_TRIGx_y*) and dual edge control registers (*GIC_SH_DUALx_y*). When interrupts are driven from the GIC to the VP, all of the interrupts are normalized to positive, level-sensitive signals as this is the interrupt type supported by the CPU interrupt inputs.

For single-edge signaling, the *Polarity* register denotes which edge is used for setting the interrupt register and which edge is ignored. For double-edged signaling, both the rising and falling edges are used to set the interrupt register. These three registers work in conjunction with one another to define the characteristics of each specific interrupt in the system. Each bit of each register corresponds to an interrupt. So for a given bit, the corresponding interrupt characteristics would be defined as shown in Table 7.1. The ‘n’ in the table entries denotes that it can be any bit of a given register, but must be the same bit of each register.

Table 7.1 Selecting Interrupt Polarity, Edge Sensitivity, and Triggering

Polarity (GIC_SH_POL[n])	Trigger (GIC_SH_TRIG[n])	Single/Dual Edge (GIC_SH_DUAL[n])	Description
0	0	x	Interrupt is level sensitive and active low. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
1	0	x	Interrupt is level sensitive and active high. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
0	1	0	Interrupt is single edge triggered on the falling edge of the signal.
1	1	0	Interrupt is single edge triggered on the rising edge of the signal.
x	1	1	Interrupt is dual edge triggered. In this case the contents of the GIC_SH_POL have no meaning because interrupts occur on both the rising and falling edges of the signal.

7.2.4.1 Trigger Type Register Group

Register Interface

The trigger type register group is made up of four "Global Interrupt Trigger Type registers", GIC_SH_TRIGx_y. The trigger type can be set to level or edge sensitive. Setting each bit in the register configures the corresponding interrupt

to be edge sensitive and clearing it configures it to be level sensitive. For example, to set the interrupt source 64 to edge sensitive bit 0 of the second GIC_SH_TRIG register (GIC_SH_TRIG127_64) should be set. For more information, refer to the *Global Interrupt Trigger Type Registers* (GIC_SH_TRIGx_y) in the *I6500 Registers* companion document.

Trigger Type Code Example

The following code example programs interrupt source 31 to be edge-sensitive.

```
#define GIC_SH_TRIG63_0 0x0180    // offset from the GIC base address for
                                  // trigger bits for interrupt sources 0 - 63
dli      a1, GIC_BASE_ADDR        // load virtual base address of the GIC
                                  // registers. NOTE: must be uncached address
dli      a0, 0x0000000080000000   // interrupt source 31 (bit 31)
sd   a0, GIC_SH_TRIG63_0(a1)      // (edge sensitive)
```

7.2.4.2 Edge Type Register Group

Register Interface

The edge type register group is made up of four "Global Dual Edge Registers", GIC_SH_DUALx_y. This register group is used if the trigger type described in the previous section is set to edge sensitive and has no effect if the trigger type is set to level sensitive. The edge type can be either single or dual edge. Setting each bit in this register group configures the corresponding interrupt source to be dual edge and clearing it configures it to be single edge. For example, to set interrupt source 64 to dual edge sensitive, bit 0 of the second Global Dual Edge register (GIC_SH_DUAL127_64) should be set. For more information, refer to the *Global Interrupt Dual Edge Registers* (GIC_SH_DUALx_y) in the *I6500 Registers* companion document.

Edge Type Code Example

The following code example programs interrupt source 31 to be dual-edge sampled.

```
#define GIC_SH_DUAL63_0 0x0200 // offset from the GIC base address for dual
bits for interrupt sources 0 - 63
dli      a1, GIC_BASE_ADDR // load virtual base address of the GIC registers
                           // NOTE: must be uncached address
dli      a0, 0x0000000080000000 // interrupt source 31 (bit 31)
sd   a0, GIC_SH_DUAL63_0(a1) // Dual
```

7.2.4.3 Polarity Type Register Group

Register Interface

The polarity register group is made up of four "Global Interrupt Polarity Registers", GIC_SH_POLx_y. This register group is used to determine the polarity sensitivity of the source. Setting each bit configures the corresponding interrupt to be active high, and clearing it configures it to be active low.

If the interrupt is single-edge-sensitive, then setting the source bit configures the source to rising edge toggle and setting clearing it configure it to be falling edge toggle. This register group has no effect if the edge type was set to dual edge sensitive. For more information, refer to the *Global Interrupt Polarity Registers* (GIC_SH_POLx_y) in the *I6500 Registers* companion document.

Polarity Type Code Example

The following code example programs interrupt source 31 for active high or rising edge:

```
#define GIC_SH_POL63_0      0x0100     //offset from the GIC base address for
                                       //polarity bits for interrupt sources 0 - 63
dli a1, GIC_BASE_ADDR                  //load virtual base address of the GIC
                                       //registers NOTE: must be uncached address
dli       a0, 0x0000000080000000       // interrupt source 31 (bit 31)
sd    a0, GIC_SH_POL63_0(a1)           // (high/rise for 31)
```

7.2.5 Interrupt Routing

The routing of interrupts to a specific input on a specific VP is controlled by the setting of 2 registers.

- Global Interrupt Map to Processor register, GIC_SH_MAP_VP — maps the interrupt to a specific VP number.
- Global Interrupt Map to Pin Register, GIC_SH_MAP_PIN — maps interrupt to a specific signal on a VP.

There is one of each of these registers for each external interrupt source. The mapping of external interrupt pins and the registers that control them is listed in Table 7.2.

Table 7.2 Register Mapping Based on External Interrupts

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
0	0x2000	GIC_SH_MAP0_VP	248	0x3F00	GIC_SH_MAP248_VP
	0x0500	GIC_SH_MAP0_PIN		0x08E0	GIC_SH_MAP248_PIN
1	0x2020	GIC_SH_MAP1_VP	249	0x3F20	GIC_SH_MAP249_VP
	0x0504	GIC_SH_MAP1_PIN		0x08E4	GIC_SH_MAP249_PIN
2	0x2040	GIC_SH_MAP2_VP	250	0x3F40	GIC_SH_MAP250_VP
	0x0508	GIC_SH_MAP2_PIN		0x08E8	GIC_SH_MAP250_PIN
3	0x2060	GIC_SH_MAP3_VP	251	0x3F60	GIC_SH_MAP251_VP
	0x050C	GIC_SH_MAP3_PIN		0x08EC	GIC_SH_MAP251_PIN
4	0x2080	GIC_SH_MAP4_VP	252	0x3F80	GIC_SH_MAP252_VP
	0x0510	GIC_SH_MAP4_PIN		0x08F0	GIC_SH_MAP252_PIN
5	0x20A0	GIC_SH_MAP5_VP	253	0x3FA0	GIC_SH_MAP253_VP
	0x0514	GIC_SH_MAP5_PIN		0x08F4	GIC_SH_MAP253_PIN
6	0x20C0	GIC_SH_MAP6_VP	254	0x3FC0	GIC_SH_MAP254_VP
	0x0518	GIC_SH_MAP6_PIN		0x08F8	GIC_SH_MAP254_PIN
7	0x20E0	GIC_SH_MAP7_VP	255	0x3FE0	GIC_SH_MAP255_VP
	0x051C	GIC_SH_MAP7_PIN		0x08FC	GIC_SH_MAP255_PIN
8 - 247	0x2100 - 0x0520	GIC_SH_MAP8_VP - GIC_SH_MAP247_VP			
	0x3EC0 - 0x08DC	GIC_SH_MAP8_PIN - GIC_SH_MAP247_PIN			

7.2.5.1 Mapping an Interrupt Source to a VP

Register Interface

There are 256 "Global Interrupt Map to VP Registers", (GIC_SH_MAPi_VP). These registers map each external interrupt source to a specific VP in the system. The ‘i’ indicates the number of the interrupts in the system, between 16 and 256. Each register contains a 24-bit field which allows each external interrupt to be mapped to up to a maximum of 24 VP’s (6 cores x 4 VP’s/core). For more information, refer to the Global Interrupt Map to VP registers (GIC_SH_MAPx) in the *I6500 Registers* companion document.

Table 7.3 shows the physical mapping of the MAP_VP field to the actual core and VP number in the system. Note that the encoding of this field is fixed and does not change based on the number of VP’s per core. For example, if there are two VP’s per core and two cores in the system, the mapping would be as follows:

- MAP_VP bits 0 and 1 would represent the two VP’s in core 0
- MAP_VP bits 2 and 3 would not be used
- MAP_VP bits 4 and 5 would represent the two VP’s in core 1
- All other bits in the MAP_VP field are unused

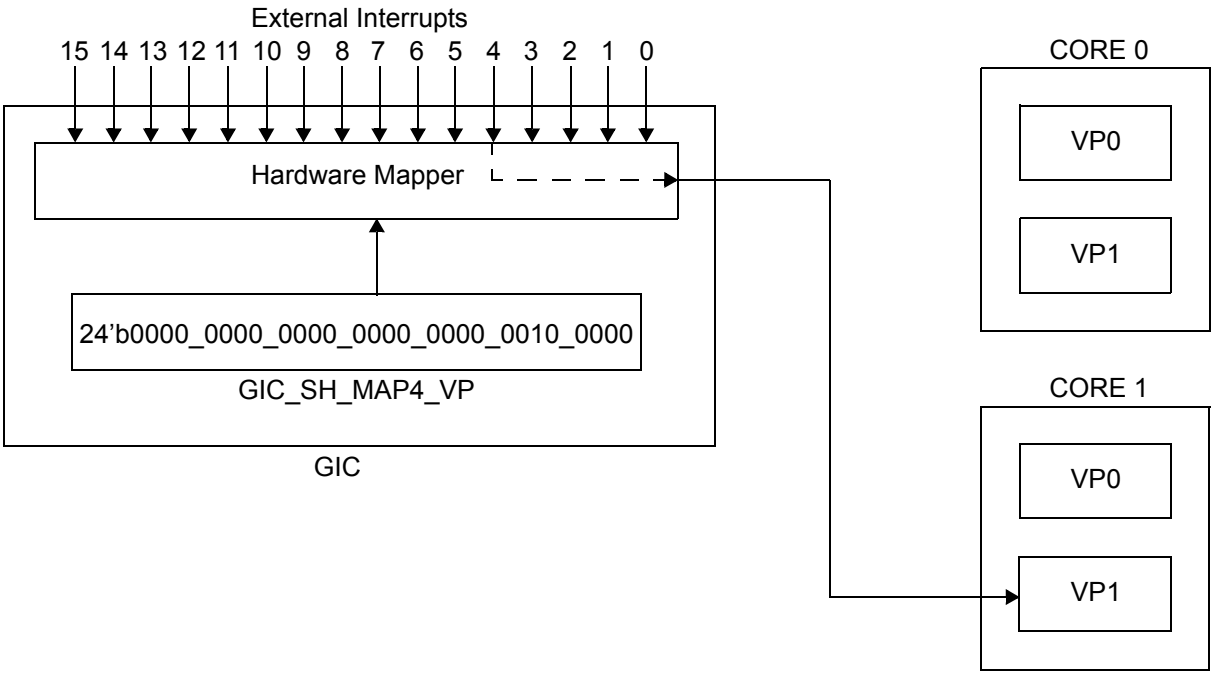
Note that this mapping scheme represents the view in non-virtualized mode, as well as the view of the Root in virtualized mode.

Table 7.3 Physical Mapping of MAP_VP Field to Core and VP Number

MAP_VP Bit	Core and VP Number		MAP_VP Bit	Core and VP Number
0	Core0, VP0		12	Core3, VP0
1	Core0, VP1		13	Core3, VP1
2	Core0, VP2		14	Core3, VP2
3	Core0, VP3		15	Core3, VP3
4	Core1, VP0		16	Core4, VP0
5	Core1, VP1		17	Core4, VP1
6	Core1, VP2		18	Core4, VP2
7	Core1, VP3		19	Core4, VP3
8	Core2, VP0		20	Core5, VP0
9	Core2, VP1		21	Core5, VP1
10	Core2, VP2		22	Core5, VP2
11	Core2, VP3		23	Core5, VP3

The following example shows a system with 2 cores, 2 VP’s/core, and 16 external interrupts, where external interrupt 4 is mapped to core 1, VP1.

Figure 7.3 Example of Mapping External Interrupt 4 to a Core 1, VP 1



In Figure 7.3, the GIC_SH_MAP4_VP register is programmed with a value of 0x00_0020 to select Core 1, VP 1 as the destination VP for external interrupt 4. For more information on how the 24-bit MAP_VP field is organized, refer to Table 7.3.

7.2.5.2 Mapping an Interrupt Source to a Specific Processor Pin

Register Interface

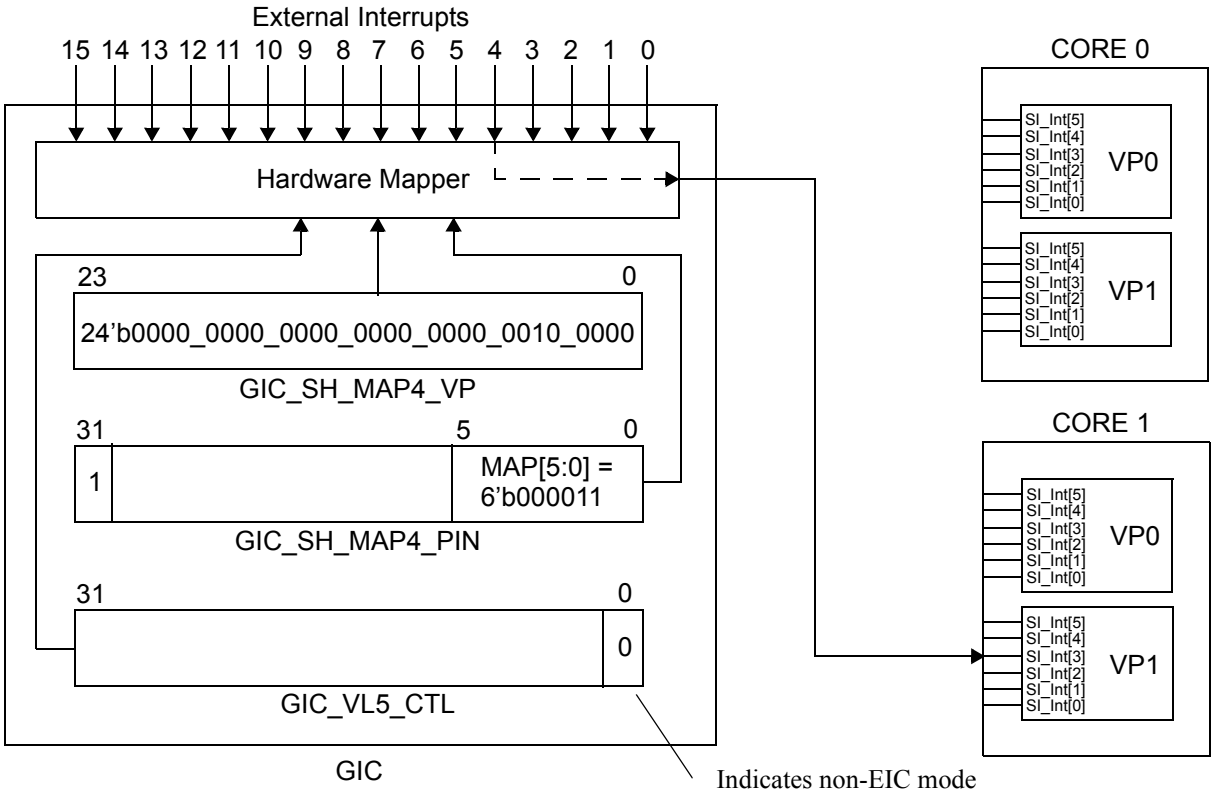
In addition to mapping each external interrupt to a particular VP as described above, each individual interrupt can also be mapped to a specific interrupt pin of a given VP. There are 256 "Global Interrupt Map to Pin Registers" (GIC_SH_MAPi_PIN) used to map each external interrupt to a specific interrupt pin on a core. The 'i' indicates the number of the interrupts in the system, between 8 and 256. Hence there are a maximum of 256 registers, one per interrupt. Each register contains a 6-bit field which allows each external interrupt to be mapped.

The type of interrupt mode determines how the interrupt pins are interpreted. In External Interrupt Controller (EIC) mode, the 6-bit field is an encoded value that can decode up to 64 different interrupt levels. In non-EIC mode, each individual pin is an interrupt, allowing for a total of six interrupts.

Interrupt Configuration Example 1 — Non-EIC Mode

The following examples show a system with 2 VP's/core and 16 external interrupts. In the first example, external interrupt 4 is mapped to interrupt pin 3 of core 1, VP 1.

Figure 7.4 Example of Mapping External Interrupt 4 to a Core 1, VP 1, Int 3 — Non-EIC Mode



In Figure 7.4, the GIC_SH_MAP4_VP register is programmed to select Core 1, VP 1 as the destination VP for external interrupt 4. Note that this is not an encoded value. Each bit in this field represent a specific VP up to 24 in the system. Refer to Table 7.3 for the core/VP mapping of this register.

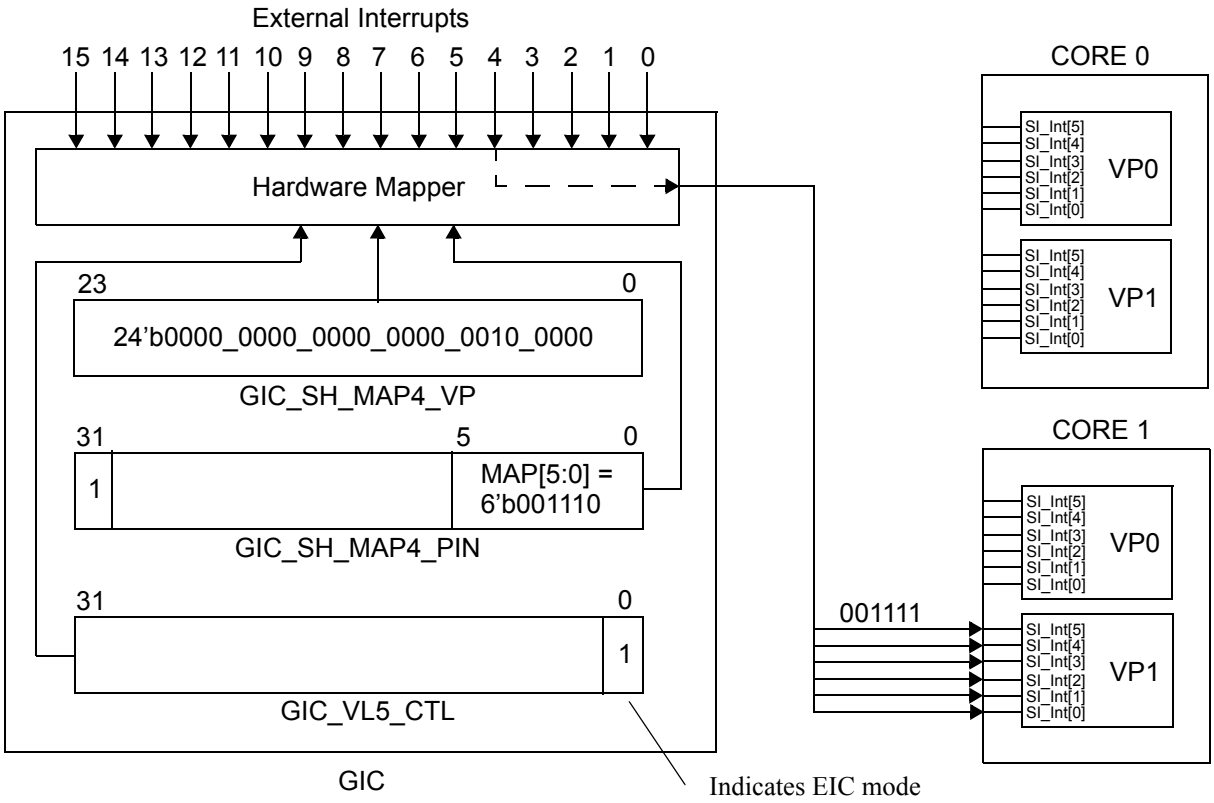
The MAP field of the GIC_SH_MAP4_PIN register is programmed with a value of 0x03, which selects interrupt pin 3 of the VP selected by the GIC_SH_MAP4_VP register. Bit 31 of the GIC_SH_MAP4_PIN register is set to 1 to indicate the external interrupt corresponds to an interrupt and not an NMI. Note that the MAP field is an encoded value and represents a binary value for interrupt pin 3 of the VP.

Bit 0 of the GIC_VL5_CTL register is set to 0 to indicate non-EIC interrupt mode.

Interrupt Configuration Example 2 — EIC Mode

The following examples show a system with 2 VP's/core and 16 external interrupts. In the first example, external interrupt 4 is mapped to the interrupt pins of core 1, VP 1, interrupt level 15.

Figure 7.5 Example of Mapping External Interrupt 4 to a Core 1, VP 1, Int Level 15 — EIC Mode



In Figure 7.5, the GIC_SH_MAP4_VP register is programmed to select Core 1, VP 1 as the destination VP for external interrupt 4. Note that this is not an encoded value. Each bit in this field represents a specific VP up to 24 in the system.

The MAP field of the GIC_SH_MAP4_PIN register is programmed with a value of 0x0E, which routes the encoded interrupt level of 15 to the VP selected by the GIC_SH_MAP4_VP register. It is important to note that when programming the GIC_SH_MAPi_PIN registers in EIC mode, the value in this field represents one less than the actual EIC interrupt level. In this case, a value of 0x0E represents interrupt level 15. Bit 31 of the GIC_SH_MAP_PIN register is set to 1 to indicate the external interrupt corresponds to an interrupt and not an NMI.

Bit 0 of the GIC_VL5_CTL register is set to 1 to indicate EIC interrupt mode. In this mode, the interrupt level sent to the target VP is a 6-bit encoded value between 0 and 63, with 0 meaning no interrupts.

In this example, a value of 6'b001110 indicates the value on the interrupt bus corresponds to external interrupt 15. The value in the register is one less than the actual interrupt level as described above.

7.2.6 Enabling, Disabling, and Polling Interrupts

The enabling, disabling and polling of interrupts is configured through several registers in the GIC that are shared by all VP's.

There are 4 shared registers groups for enabling, disabling and polling of interrupts.

- Enabling an interrupt using the "GIC Set Mask Registers", GIC_SH_SMASK

- Disabling an interrupt using the "GIC Reset Mask Registers", GIC_SH_RMASK
- Determining the Enable/Disable state of an interrupt state using "GIC Mask Register", GIC_SH_MASK
- Polling the interrupt active state using the "GIC Pending Register", GIC_PEND_MASK

Like the trigger registers, each interrupt source is represented by one bit in each register group. Each register in a group is 64 bits so each controls 64 interrupt sources. The first register in each group would control interrupts sources 0 - 63, the next 127 - 64 and so on. Since there can be 256 interrupt sources there are 4 registers in each group.

The number of interrupt sources is a fixed value configured at build time, so the actual number of interrupts may be less than 256. The actual number of system interrupts can be determined by reading the NUMINTERRUPTS field of the "GIC Configuration Register", GIC_SH_CONFIG.

7.2.6.1 Enabling External Interrupts

Register Interface

The GIC Set Mask register group is used to enable external interrupts. It is made up of "GIC Set Mask Registers", GIC_SH_SMASK. For synchronization purposes this is a write-only register. Setting the source bit enables the interrupt.

7.2.6.2 Disabling External Interrupts

Register Interface

The GIC Reset Mask register group is used to disable external interrupts. The GIC supports a maximum of 256 external interrupts. The GIC Reset Mask register group is made up of four write-only "GIC Reset Mask Registers":

- *GIC_SH_RMASK_63_0*
- *GIC_SH_RMASK_127_64*
- *GIC_SH_RMASK_191_128*
- *GIC_SH_RMASK_255_192*

Each bit in these registers corresponds to an external interrupt. Setting a bit to one resets and disables the corresponding interrupt.

7.2.6.3 Determining the Enabled or Disabled Interrupt State

Register Interface

The GIC Mask register group is used to determine if an external interrupt is enabled. It is made up of the following GIC_SH_MASK register. These registers are read-only.

- *GIC_SH_MASK_63_0*
- *GIC_SH_MASK_127_64*
- *GIC_SH_MASK_191_128*
- *GIC_SH_MASK_255_192*

If a bit is set the corresponding interrupt source is enabled. If it is clear the corresponding interrupt is disabled.

7.2.6.4 Polling for an Active Interrupt

Register Interface

The GIC Pending register group is used to determine if a external interrupt is active. These bits are set by hardware when an interrupt event occurs. The group is made up of the following GIC_SH_PEND read-only registers.

- *GIC_SH_PEND_63_0*
- *GIC_SH_PEND_127_64*
- *GIC_SH_PEND_191_128*
- *GIC_SH_PEND_255_192*

If a bit is set the corresponding interrupt source is active. If it is clear the corresponding interrupt is inactive.

7.2.6.5 Programming Example

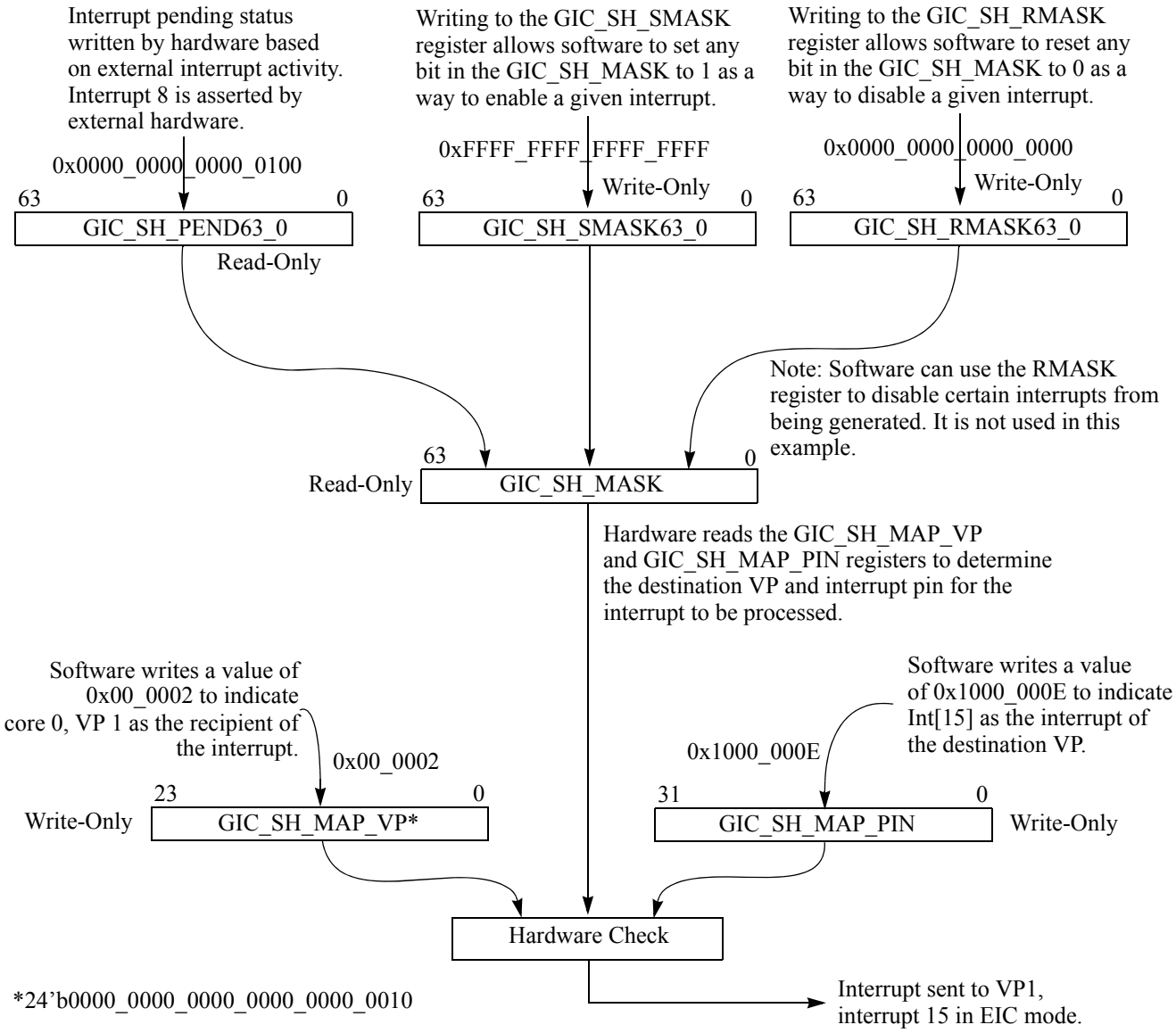
When an interrupt occurs, the corresponding bit in the *GIC_SH_PEND* register is set by hardware. If the corresponding interrupt enable bit in the *GIC_SH_MASK* bit is set, the GIC delivers the interrupt to the appropriate VP. The hardware does this by using the *GIC_SH_MAP_VP* register to send the interrupt to the appropriate VP and the *GIC_SH_MAP_PIN* register to set the interrupt pins for that *VP*.

In the following example:

- External interrupt 8 is asserted
- All bits of the *GIC_SH_SMASK* register are set, enabling all 64 interrupts.
- The receiving VP is core 0, VP 1, and the receiving interrupt level is 15.

This example is shown in Figure 7.6.

Figure 7.6 Masking and Mapping of Interrupts in the GIC



7.2.7 Inter-processor Interrupts

Register Interface

Each processor in the system can interrupt any other processor. Each inter-processor interrupt is configured just like an external interrupt using sources not being used by external devices. The interrupt sources chosen for this purpose must be configured to be edge sensitive by setting the appropriate bits in GIC_SH_TRIG registers.

The "Global Interrupt Write Edge Register", GIC_SH_WEDGE is a shared register used to deliver an interrupt to another processor (only one per system). It is also used to clear an interrupt. There are two fields in the GIC_SH_WEDGE register used to do this.

- The *RW* bit determines if the interrupt is being set (delivered) or cleared. Setting this bit delivers an interrupt and clearing the bit clears the interrupt.
- The *INTERRUPT* field should be set to the interrupt number to be set or cleared.

7.2.7.1 WEDGE Register Programming Example

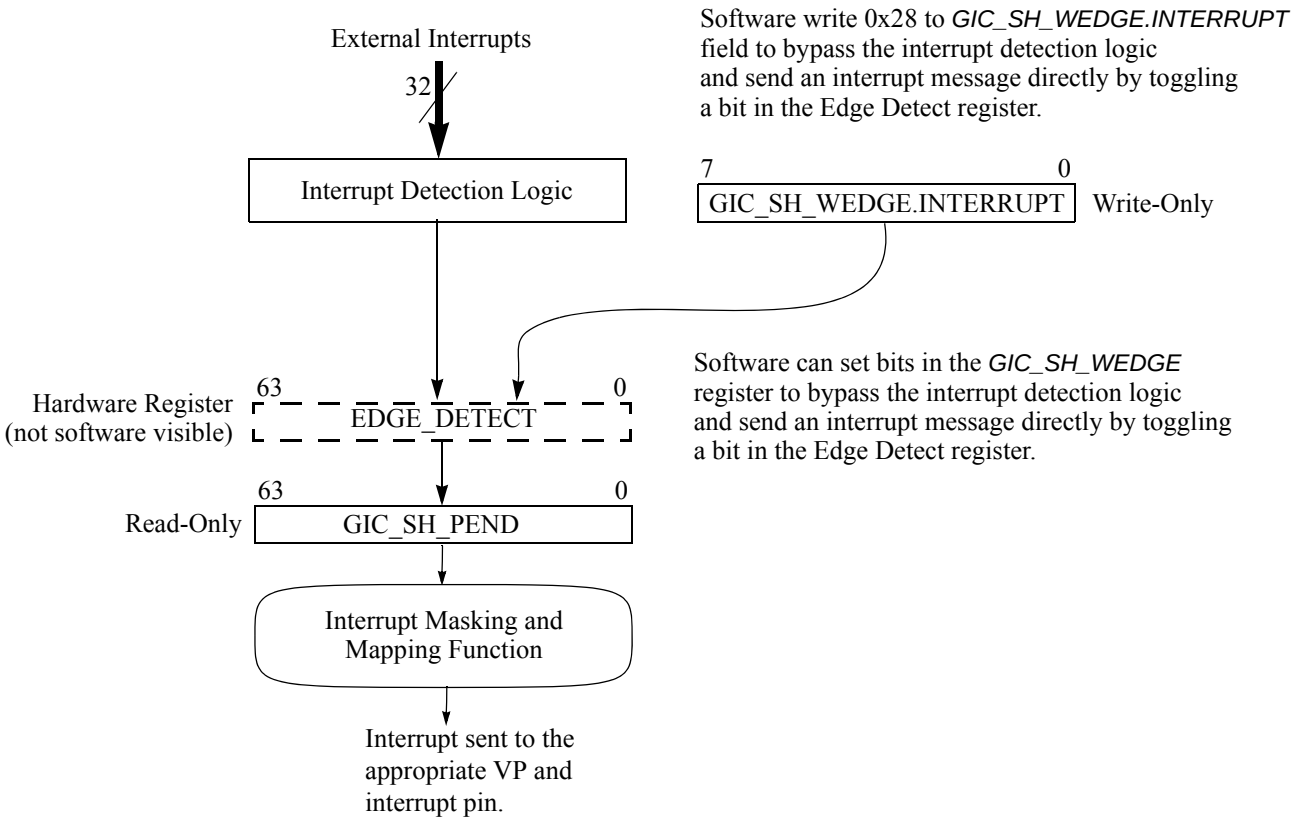
A write that sets the R/W bit of the *Write Edge* (WEDGE) register is treated equivalently to having the edge detection logic see an active edge. Because the programming of the Write Edge register has a direct effect on the state of the internal Edge Detect register, the *Write Edge* register can be used to bypass the edge detection logic. Thus, it does not matter whether the corresponding interrupt is configured to be rising, falling, or dual edge sensitive.

When VP 0 wants to interrupt VP 1, the number of the interrupt to be used is programmed into the GIC_SH_WEDGE register. The selected interrupt must be mapped to the target VP (VP1 in this example) using the GIC_SH_MAPi_VP register).

For example, assume VP 0 wants to toggle interrupt 40. In this case, kernel software writes a value of 0x28 into the GIC_SH_WEDGE register. Hardware then writes the value in the WEDGE register into the Edge Detect hardware register, effectively bypassing the edge detection logic. Hardware determines that interrupt being toggled belongs to VP 1, not VP 0. The GIC routing logic then routes interrupt 40 onto the appropriate VP 1 interrupt pins.

Figure 7.7 shows how the *Write Edge* register can be used to bypass the interrupt detection logic and assert interrupt directly. Setting a bit in the *Write Edge* register in turn sets the corresponding bit in the internal Edge Detect register, forcing an interrupt to be generated and allowing for inter-processor interrupts within the GIC.

Figure 7.7 Sending Inter-Processor Interrupts in the GIC



Example of Sending an Inter-Processor Interrupt — C Code

The following is a C code example of sending an inter-processor interrupt. First the #defines:

#define	Value	Description
GIC_SH_WEDGE	*((volatile unsigned int*) (0xbbdc0280))	Address of the GIC_WEDGE_REGISTER.
FIRST_IPI	32	Source number for the first IPI.

void set_ipi(int cpu_num) {

// Add the enable bit, the first IPI number and the cpu number and write it to the GIC_SH_WEDGE register

```
GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num ;
```

Code Example of Clearing an Inter-Processor Interrupt

Once received, the interrupt routine should do whatever action is intended for the interrupt and clear the interrupt by writing the interrupt number to the GIC_SH_WEDGE register before executing the ERET instruction. NOTE: only the interrupt number is set before the write so the R/W bit is cleared indicating that the interrupt is to be cleared.

```
li      k0, (GIC_SH_WEDGE | GIC_BASE_ADDR)
mfc0    k1, C0_EBASE                        // Get CP0 EBase
ext     k1, k1, 0, 10                       // Extract CPUNum
addiu   k1, 0x20                            // Offset to base of IPI interrupts.
sw      k1, 0(k0)                           // Clear this IPI.
```

7.2.8 Local Timer Configuration

The GIC also controls how devices within the processor and the GIC are configured and mapped locally to the processor.

There are 2 devices that are added as part of the GIC described in this section:

- GIC Interval Timer - a 64 bit timer that compares a local compare register, *GIC_CORE_COMPARE* of a processor with a global counter, *GIC_SH_COUNTER* in the GIC and activates an interrupt when they match.
- GIC Watchdog Timer - a 32 bit decrementing counter, GIC_VO_WD_COUNT.

7.2.8.1 GIC Interval Timer

The interval timer is similar to the CP0 Count/Compare timer within each processor. The difference is that the *GIC_SH_ COUNTER* register is global to the cluster so that all processors in the same cluster have the same time reference.

Counter Registers

The counter register (*GIC_SH_COUNTER)* is in the shared section of the GIC memory map. The counter must be stopped before it is set. This is done by setting the COUNTSTOP bit of the *GIC_SH_CONFIG* register. In practical use the counter is usually set by an OS at boot time by one processor. This counter register is also available (read only) in user mode located at offset 0 of the User Mode Visible Section of the GIC.

The COUNTBITS field of the *GIC_SH_CONFIG* register is used to set up the width of the *GIC_SH_Counter* register. In the GIC design, this field has a default value of 0x8, indicating a total counter size of 64-bits.

Compare Registers

The compare register (*GIC_VLi_COMPARE)* is located in the local section of the GIC memory map making the count specific to each processor. These registers can be written at any time. When the count value equals the compare value an interval timer interrupt is asserted. The interrupt is cleared (de-asserted) by writing to the *GIC_VLi_COMPARE* register.

Determining the Counter Width

To derive the total width of the counter, the following formula is used:

```
32 + COUNTBITS x 4
```

Where:

‘32’ is the minimum width of the counter and ‘COUNTBITS’ is the value in the COUNTBITS field of the *GIC_SH_CONFIG* register.

For example, if the COUNTBITS field contains a value of 0x8, the overall width of the counter would be:

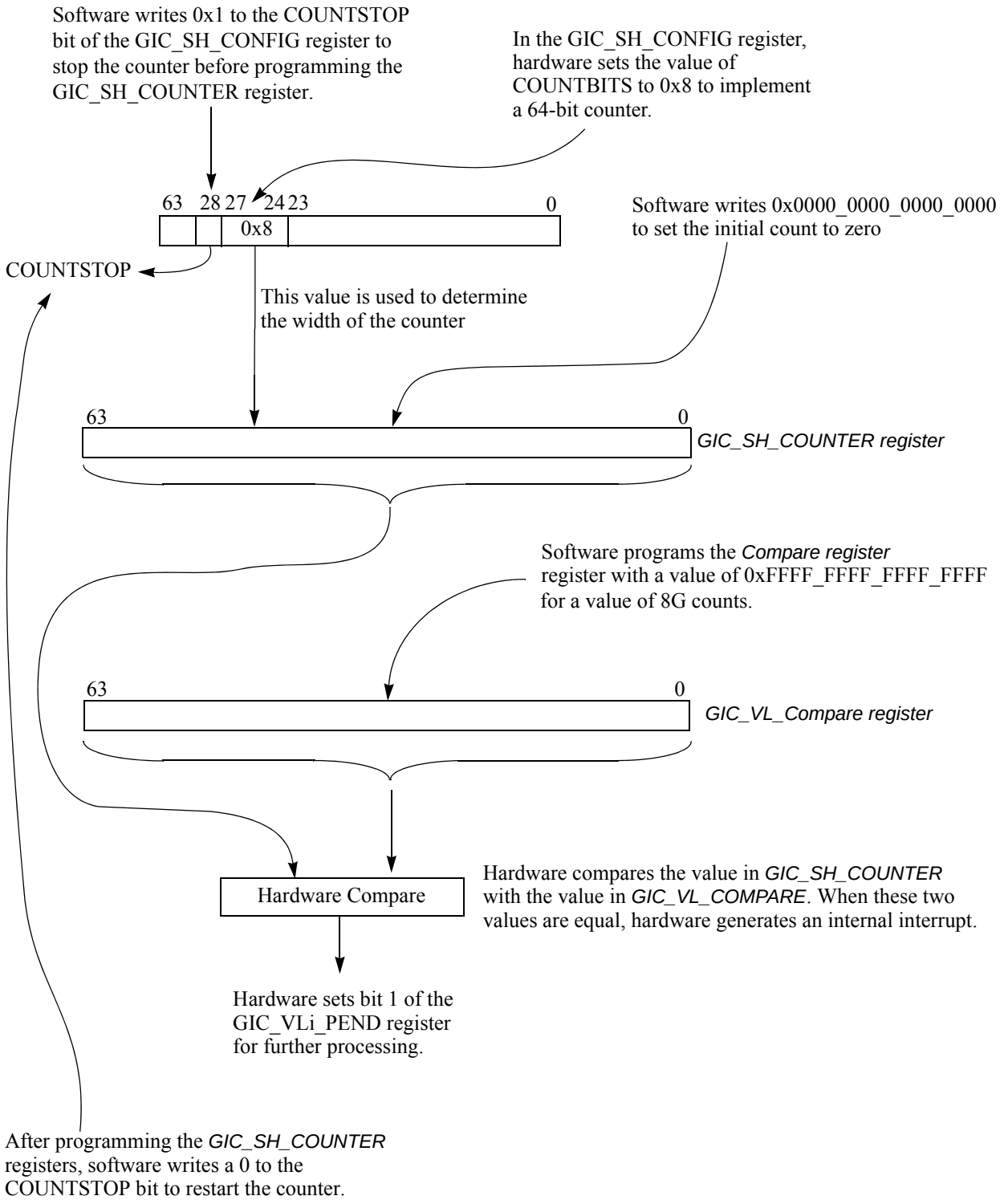
```
32 + 8 x 4 = 64 bits
```

In the GIC design, the counter can be a value between 32 and 64 bits in increments of 4. For example, 32 bits, 36 bits, 40 bits, etc.

Counter Based Interrupt Example

In the example shown in Figure 7.8, the width of the counter is 64-bits, and the Compare value is 0x1_FFFF_FFFF which corresponds to 8G clock cycles. When this count is reached, hardware generates an internal interrupt.

Figure 7.8 Example of GIC Internal Counter-Based Interrupt Generation



7.2.8.2 GIC Watchdog Timer

Register Interface

Each VP supports a Watchdog timer that is controlled by the following three registers.

- The "*GIC Watchdog Timer Configuration Register*", GIC_VLi_WD_CONFIG is local to each processor and reports state information and configures the characteristics of the timer.
- The "*Watchdog Timer Initial Count Register*", GIC_VLi_WD_INITIAL is local to each processor and is used to set the timer interval.
- The "*Watchdog Timer Count Register*", GIC_VLi_WD_COUNT is a read only register local to each processor that contains the current value of the countdown.

GIC Watchdog Timer Configuration Register

The GIC Watchdog Timer Configuration register contains bits that control the function of the timer.

- Clearing the WAIT bit of GIC_VLi_WD_CONFIG register (default value) causes the counter to stop counting when the processor is executing a WAIT instruction or is in a low power state controlled by the Cluster Power Controller (CPC). Setting this bit causes it to continue counting down.
- Clearing the Debug bit (default value) causes the counter to stop counting when the VP enters debug mode. When this bit is set the count continues counting down.
- The TYPE field in bits 3:1 of this register determines what happens when the timer reaches 0.

Table 7.4 GIC Watchdog Timer Modes

Encoding	Mode	Behavior
0x0	One Trip	An interrupt is asserted and the timer stops (typically an NMI).
0x1	Second Countdown	An interrupt is asserted and the timer reloads. If the timer expires for the second time before being reloaded again, all cores are reset. This mode provides a way to distinguish between a Software hang and a Hardware hang. Usually the Watchdog Timer Interrupt is routed to the NMI interrupt. This causes the processor to soft reboot. That is what happens in this mode when the timer expires the first time. If this was a software hang during the reboot the kernel software should reload the Watchdog Timer, thus avoiding the second expiration. If the processor itself does not respond to the interrupt, then it is assumed to be a hardware issue. Therefore, when the count expires the second time a reset signal is sent to all processors in the system.
0x2	Programmable Interval Timer (PIT)	An interrupt is asserted, the initial count is reloaded and the time starts counting down again interrupting each time the counter reaches 0. This mode provides a per-processor interval timer. This is one mode where the interrupt should not be routed to NMI. It should instead be routed to a normal interrupt where for example the interrupt could be used in a time slicing OS.

Clearing the WD_START bit disables the timer and when it is set it enables the timer. Writing WD_START with a 1 triggers a reload of the GIC_VL_WD_COUNT register with the value in the GIC_VLi_WD_INITIAL register. For

more information, refer to the *Watchdog Timer Config Register* (GIC_VL_WD_CONFIG) at offset 0x0090 in the *I6500 Registers* companion document included in the release.

Watchdog Timer Initial Count Register

Register Interface

The "Watchdog Timer Initial Count Register", GIC_VLi_WD_INITIAL is local to each processor and is used to set the timer interval. To start the counter for the first time the counter should be disabled by clearing the WD_START bit in the GIC_VLi_WD_CONFIG register and the countdown value loaded into this register and then the counter enabled by setting the WD_START bit. For more information, refer to the *Watchdog Timer Initial Count Register* (GIC_WD_INITIAL) at offset 0x0098 in the *I6500 Registers* companion document included in the release.

Watchdog Timer Count Register

The "Watchdog Timer Count Register", GIC_VL_WD_COUNT is a read only register that contains the current value of the countdown. This register is reloaded with the value in the GIC_VLi_WD_INITIAL register each time the WD_START bit in the GIC_VLi_WD_CONFIG register is set. For more information, refer to the *Watchdog Timer Count Register* (GIC_WD_COUNT) at offset 0x0094 in the *I6500 Registers* companion document included in the release.

Watchdog Timer Masking and Mapping

Register Interface

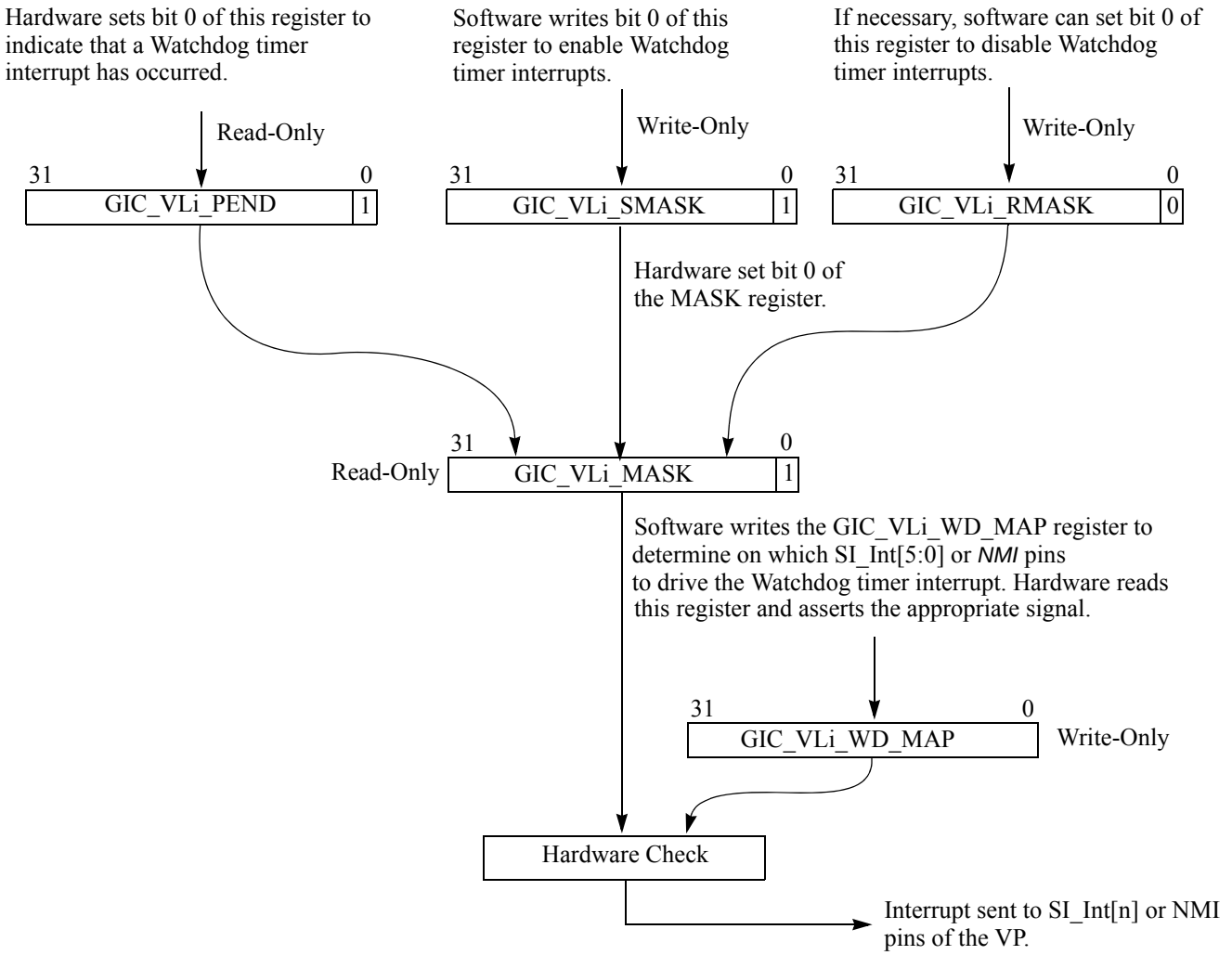
Once a Watchdog timer interrupt is generated, hardware sets bit 0 of the *Local Interrupt Pending* register (*GIC_VLi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 0 in the *Local Interrupt Mask* register (*GIC_VLi_MASK*) at offset address 0x0008 to determine whether the Watchdog timer interrupt has been masked. The *GIC_VLi_MASK* register is read-only.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_VLi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_VLi_RMASK*) at offset address 0x000C. Software sets bit 0 of the *SMASK* register to enable the Watchdog timer interrupt, or it can set bit 0 of the *RMASK* register to disable Watchdog timer interrupts. Note that when the WatchDog timer is programmed to generate a hardware reset, the reset cannot be masked by the *Local Interrupt Mask* register.

Once hardware has determined the masking characteristics of the interrupt, it uses the *Watchdog Timer Map-to-Pin* register at offset address 0x0040 to determine which *SI_Int[5:0]* or *NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VP interrupts. For example, if kernel software programs this field with a value of 0x2, then the Watchdog timer interrupt will be driven onto *SI_Int[2]*. In non-EIC mode, only encodings 0 - 5 are valid.

In EIC mode, the VP encodes this field to support up to 64 interrupts. For example, if kernel software programs this field with a value of 0x20, then the Watchdog timer interrupt corresponds to interrupt 32. This encoded value is then driven onto *SI_Int[5:0]*.

Figure 7.9 Watchdog Timer Interrupt Masking and Mapping in the GIC



Watchdog Timer and Debug Mode

Under certain conditions, kernel software may want to suspend Watchdog timer operation while the I6500 Multiprocessing System is in debug mode. This can be accomplished by clearing the DEBUG bit of the *Watchdog Timer Config* register located at offset address 0x0090. When this bit is cleared, counting is stopped. Note that the DM bit of the CP0 *Debug* register (*DEBUG*$_{DM}$) must be set to place the device in debug mode.

If the DEBUG bit is set, entering debug mode has no effect on the Watchdog timer counting process.

Watchdog Timer and Low Power Mode

Under certain conditions, kernel software may want to suspend Watchdog timer operation while the I6500 Multiprocessing System is in low power mode. This can be accomplished by clearing the WAIT bit of the *Watchdog Timer Config* register located at offset address 0x0090. When this bit is cleared, counting is stopped, including when low power mode is entered via the WAIT instruction.

If this bit is set by the kernel, entering low power mode has no effect on the Watchdog timer counting process.

7.2.9 Local Interrupt Routing and Masking

Local interrupts are internal events that occur within the I6500 Multiprocessing System. The routing and masking of local interrupts is handled in a similar manner to the Watchdog timer interrupts described in the previous section. The local interrupts are defined as follows:

- Count/Compare interrupt
- Local CPU timer interrupt
- Performance counter interrupt
- Two software interrupts
- Fast Debug Channel (FDC) interrupt

Each of these interrupts can be routed and masked as described in the following subsections.

7.2.9.1 Local Interrupt Routing

There is a Local Interrupt Map-to-Pin Register for each local interrupt source that maps the local interrupt to a specific input on the processor. There are two bits, MAP_TO_PIN and MAP_TO_NMI that control the type of input that is assigned to the interrupt source. Only one of these bits can be set at any one time for each interrupt.

- The local MAP_TO_PIN registers map each local interrupt source to Interrupt Pending bits in the CP0 Cause register of the core. If the MAP_TO_PIN bit (31) of the corresponding register is set, indicating that the interrupt source is mapped to an interrupt pin, the actual interrupt is mapped to a specific interrupt pin using the 6-bit MAP field of this register. This field contains the encoded value of the interrupt (0 - 63) in EIC mode. For example, a value of 0x20 in the MAP field indicates interrupt 33 (decimal). For vectored interrupt (non-EIC) mode, each bit of the MAP field corresponds to one of six interrupt pins.
- If the MAP_TO_NMI bit (30) of this register is set, this indicates that the interrupt source will be mapped to the NMI bit in the CP0 Status register. This in essence will cause the core to soft boot using the boot exception vector as the start of the interrupt routine.

7.2.9.2 Local Interrupt Masking

Register Interface

In addition to the routing of interrupts, the I6500 core also provides the ability to mask interrupts using the following registers:

- Local Interrupt Pending register (*GIC_VL_PEND*). This read-only register indicates the status of each of the local interrupts listed at the beginning of the section entitled Local Interrupt Routing and Masking.
- Local Interrupt Mask register (*GIC_VL_MASK*). This read-only register indicates the whether a given local interrupts has been enabled prior to interrupt processing. This register manages the mask status for each of the local interrupts listed at the beginning of the section entitled Local Interrupt Routing and Masking.
- Local Interrupt Reset Mask register (*GIC_VL_RMASK*). This write-only register allows the programmer to disable one or more of the local interrupts by setting the bits of this register. If a given bit is set, interrupt are disabled for the corresponding interrupt type. This register manages the reset mask function for each of the local interrupts listed at the beginning of the section entitled Local Interrupt Routing and Masking.

- Local Interrupt Set Mask register (*GIC_VL_SMASK*). This write-only register allows the programmer to enable one or more of the local interrupts by setting the bits of this register. If a given bit is set, interrupts are enabled for the corresponding interrupt type. This register manages the set mask function for each of the local interrupts listed at the beginning of the section entitled Local Interrupt Routing and Masking.

When any of the local interrupts occurs, hardware sets the corresponding of the *Local Interrupt Pending* register (*GIC_VLi_PEND*) at offset address 0x0004. Hardware then reads the state of the *Local Interrupt Mask* register (*GIC_VLi_MASK*) at offset address 0x0008 to determine whether the interrupt has been masked. If a bit in this register is set, the corresponding interrupt is ignored.

Local interrupts can be enabled by setting the appropriate bits of the *Local Interrupt Set Mask* register (*GIC_VLi_SMASK*) at offset address 0x0010. Conversely, interrupts can be disabled by setting the appropriate bits of the *Local Interrupt Reset Mask* register (*GIC_VLi_RMASK*) at offset address 0x000C.

Each of the registers listed in the above examples can be found in the *I6500 Registers* companion document included in the release.

7.3 Virtualization Support

The I6500 MPS supports virtualization and the concept of guest and root modes. The following list shows some of the changes made to the GIC to support Virtualization. Each external interrupt source is assigned a GuestID for this purpose. The Hypervisor is expected to program these fields prior to initializing interrupts in the system.

7.3.1 Enabling Virtualization Mode

Register Interface

The I6500 GIC provides Virtualization support as indicated by a logic 1 in the *GIC_CONFIG.VZP* bit. The GIC can be programmed by kernel software to operate in either virtualized (*GIC_CONFIG.VZE* = 1) or non-virtualized (*GIC_CONFIG.VZE* = 0) modes.

In the GIC non-virtualized mode, the following rules apply:

- Any registers, or any fields in the Shared and VP-Local sections that have been added for virtualization should be considered reserved and read-only.
- Any Core-Local state is maintained in the fully populated root context.
- The GIC interface to the guest context in the core, known as the Guest Interrupt Bus, is always inactive (always 0) in both EIC and non-EIC modes.
- If the core is enabled for virtualization, all guest accesses must be ignored (loads return 0s, stores are dropped).

7.3.2 Routing of Guest External Source Interrupts

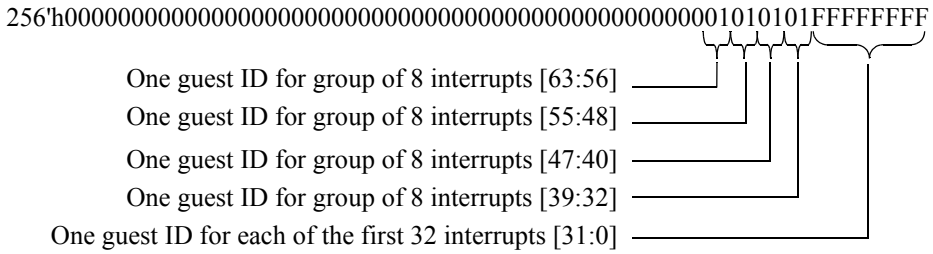
Each external interrupt source, or a logical group of external interrupt sources, is assigned a GuestID. This GuestID may be a maximum of 8-bits. The per-external-interrupt source *GuestID* field has been added to the shared section *Global Interrupt Map to Pin* registers.

The developer may choose to assign one GuestID to each external interrupt source. Alternatively, since the number of interrupt sources may be large (up to 256 interrupts), an implementation may choose to group external interrupt sources by GuestID, or provide an intermediate configuration such that some number of sources are each assigned a

GuestID, while the remaining are grouped, and each group is assigned a GuestID. An example intermediate solution is one where the 1st 32 interrupt sources are individually assigned GuestIDs, while the remaining sources are divided up into groups of 8, each group with a GuestID.

To facilitate the configuration of GuestID grouping, a 256-bit wide vector is provided which needs to be set at build time as per the required GuestID grouping scheme. This vector is 256 bits wide, which is the maximum number of external interrupt sources supported by the I6500 GIC. However, only the relevant lower indexed bits will take effect when the GIC is configured for less than 256 external interrupts.

Each bit in this vector represents whether or not a physical GuestID register exists ('1' in the bit) or not ('0' in the bit) for that bits corresponding external interrupt source. In the case where a physical GuestID register does not exist for an external interrupt source, that external interrupt source uses the GuestID value from whatever the next lower indexed external interrupt source which has a physical GuestID register. For example, in a 64-interrupt system where the 1st 32 interrupt sources are individually assigned GuestIDs and the remaining sources are divided up into groups of 8, the 256-bit GuestID grouping vector would be configured with the value shown below:



Software can determine the GuestID grouping scheme configured at build time by reading this 256-bit GuestID grouping vector registers. For more information, refer to the ID Group Configuration Registers (GIC_SH_GID_CONFIG) at offsets 0x0080 through 0x0098 in the *I6500 Registers* companion document included in the release.

By convention, a GuestID of 0 specifies root, while a non-zero GuestID specifies a guest. In addition, each Core-Local section in the GIC is aware of the GuestID resident in the physical core. These resident GuestIDs will be brought into the GIC via the SI*_GID input ports and this is equal to the core cores $GuestCtl1_{ID}$ CP0 register field.

7.3.3 Qualification of Root or Guest Software Access to GIC registers

In general, only the root software (hypervisor) requires access to the GIC configuration registers. Such configuration registers include, but not limited to, are for the specification of each interrupt's type (e.g., polarity, edge/level etc), Core assignment, interrupt routing etc. However, the guest software may require access to a subset of GIC registers for reading interrupt pending information, masking and clearing interrupts etc. Since a subset of GIC registers are shared by multiple guests and root, any guest-specific reads/writes must be qualified to avoid effecting the interrupts that are not associated with the intended guest.

The shared section registers listed below need to be directly accessed by guest. The individual interrupts are represented using the n_m nomenclature,

where n_m = the range of interrupt. For example, GIC_SH_PEND_63_0, where n = 63 and m = 0. As such, for all registers using the n_m nomenclature, there are four registers of each type to manage all 256 possible interrupts.

- GIC_SH_WEDGE - to cause Inter-Processor-Interrupts (IPI) and clear EDGE registered external interrupts.

- GIC_SH_PENDn_m - to determine which external interrupts are pending.
- GIC_SH_MASKn_m - to determine which external interrupts are masked.
- GIC_SH_SMASKn_m - to set mask bits for external interrupts.
- GIC_SH_RMASKn_m - to clear mask bits for external interrupts.
- GIC_SH_TRIGn_m - to allow guest to set EDGE for causing IPI to other cores.
- GIC_SH_POLn_m - there is currently no identified reason for guest access to this register, but it is safe to do so.
- GIC_SH_DUALn_m - there is currently no identified reason for guest access to this register, but it is safe to do so.

Apart from the WEDGE register, all of the above listed registers contains one bit per external interrupt source. Guest accesses to each of these per external interrupt source bits are qualified with a per-external interrupt source valid vector. On guest writes to the WEDGE register, the encoded interrupt number value gets decoded out to drive the per-external interrupt source logic. Guest writes to the WEDGE register are qualified by gating this driving of per-external-interrupt source logic with the same per-external-interrupt source valid vector.

The following guest context replicated VP-Local section registers may need to be directly accessed by guest software.

- GIC_VLi_PEND - for guest software to determine which local guest interrupts are pending.
- GIC_VLi_MASK - for guest software to determine which local guest interrupts are masked.
- GIC_VLi_SMASK - for guest software to set mask bits for local guest interrupts.
- GIC_VLi_RMASK - for guest software to clear mask bits for local guest interrupts.
- GIC_VLi_Compare - This allows the guest software to directly set its compare value after sampling its offset counter value.

where i = 0 to 31, the max number of configured cores.

Each of these registers is described in the previous sections of this chapter.

7.3.4 Guest Mode Count-Compare Timer Interrupts

For guest context use of the Count-Compare (CC) timer interrupts, the global counter value that is common to root and all guests cannot be used. Therefore, a counter which is offset by an n-bit (set to 8 by default) value is used for each guest context. To specify this guest counter offset value, a GIC_VLi_COFFSET register is added to each Core-Local section and the root is expected to program this offset value register. In addition, the compare value registers are replicated for the guest context and these are added as GIC_VLi_Compare register to each Core-Local section. This allows guest and root contexts in each core to set compare independently.

To facilitate this guest context interrupt routing, the Count-Compare register bits are replicated for guest context registers GIC_VL_[PEND/MASK/SMASK/RMASK]_MAP registers and also the GIC_VL_COMPARE register replicated for guest context.

Note the guest software is not allowed to write to *GIC_SH_COUNTER* register and also cannot disable the counter by writing to the $GIC\_SH\_CONFIG_{COUNTSTOP}$ field.

7.3.5 Watchdog (WD) Timer Guest and Root Interrupts

In the GIC, a single WatchDog timer is present for the root context. The root may allow the guest to utilize this single WatchDog timer by setting the newly added control bit GEN in the *GIC_VLi_WD_CONFIG* register. In virtualized mode (*GIC_SH_CONFIG*$_{VZP}$ = 1 and *GIC_SH_CONFIG*$_{VZE}$ = 1) if the root software sets GEN = 1, then the guest software is allowed to access the WatchDog timer related registers *GIC_VLi_WD_[MAP/CONFIG/COUNT/INITIAL]*. However, in non-virtualized mode (*GIC_SH_CONFIG*$_{VZP}$ = 1 and *GIC_SH_CONFIG*$_{VZE}$ = 0), this GEN control bit is a don't care and is not used to qualify any GIC register accesses.

Even when guest is allowed access to WatchDog timer with GEN = 1, there are further restrictions for guest accesses of certain WatchDog timer related register fields. These further restrictions are as follows,

- Guest has limited access to *GIC_VLi_WD_CONFIG* register:
  - The *WDRESET*, *WAIT* and *DEBUG* fields are read-only 0 for guest.
  - The guest can only set the *TYPE* field with values 0x0 and 0x2 and not the value of 0x1. Thus when guest writes this 3-bit field, the LSB is dropped and for guest reads, the LSB returns 0.
- Guest has limited access to *GIC_VLi_WD_MAP* register.
  - The guest writes to *MAP_TO_NMI* field is further gated by *GIC_SH_CONFIG*$_{GNMI}$ field.

When guest is allowed access to WatchDog timer, the guest may handle the generated WatchDog interrupts without root intervention. To facilitate this, the WatchDog related bits are replicated in *GIC_VLi_[PEND/MASK/RMASK/SMASK]* registers for guest context and guest software is given direct access to them.

7.4 GIC User-Mode Visible Section

The Shared (SH), VP-local (VL), and VP-other (VO) sections of the GIC register map are meant to be located in privileged system virtual address space, in which only kernel mode software can initialize and update the interrupt controller.

A separate 64KB address space is allocated so that it may be mapped to user-mode virtual address space. Within this address space are aliases for GIC registers that are read so often that it makes sense to make them available to user-mode programs without requiring a system call. The aliases for these registers are read-only. Currently, the only register aliased into this space is the shared Counter registers.

The addresses for the registers within the User-Mode Visible Section of the GIC are calculated as follows:

```
SharedSection_Register_Physical_Address = GIC_baseaddress +
UMVisible_Section_baseoffset + Register_Offset
```

Table 7.5 User-Mode Visible Section Register Map

Register Offset	Name	Type	Description
0x0000	GIC Counter (`GIC_SH_COUNTER`)	R	Read-only alias for GIC Shared Counter.
Any Other Offsets	Reserved		Reserved for future extensions.

Note that register is located at an offset of 1_0000 relative to the GIC base address.

Chapter 8

Floating-Point Unit (FPU)

The I6500 core features an optional IEEE 754 compliant 3rd generation Floating Point Unit (FPU3) with SIMD that handles all floating point operations within the I6500 Multiprocessing System. The I6500 core can issue up to two instructions per cycle to the FPU.

This chapter provides information on how to enable the FPU, how to handle floating point exceptions, how to set the rounding mode, operation of the Flush-to-Zero (FS) function, and a programming example.

8.1 Overview

The FPU supports fused multiply-adds as defined by the IEEE 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*. Most FPU instructions have a one-cycle throughput. All floating point denormalized input operands and results are fully supported in hardware.

The FPU contains thirty-two, 128-bit vector registers shared between SIMD and FPU instructions (FPU uses only the lower 64-bits of these registers). Single precision floating point instructions use the lower 32 bits of the 128 bit register. Double precision floating point instructions use the lower 64 bits of the 128 bit register. SIMD instructions use the entire 128 bit register interpreted as multiple vector elements; 16 x 8-bit, 8 x 16-bit, 4 x 32-bit, and 2 x 64-bit vector elements.

8.1.1 IEEE Standard 754

The IEEE Standard 754-2008 defines the following:

- Floating-point data types
- The basic arithmetic, comparison, and conversion operations
- A computational model

The standard does not define specific processing resources nor does it define an instruction set.

8.1.2 Floating Point Registers

The FPU programmable functions described in the following subsections are controlled by the *Floating Point Control and Status Register* (FCSR). These include elements such as enabling selected types of FPU exceptions, rounding mode, and flush-to-zero operation. Normally these fields are updated using a read-modify-write operation. The FSCR is read, the new value is logically OR'd with the existing value and merged into a single value. The result is written back to the FCSR.

To avoid having to use a read-modify-write sequence, the FPU provides two additional registers that allow indirect updates to the FCSR fields to be performed in a single write operation.

- The Floating Point Exceptions register (FEXR) located at CP1 Register 26, is an alternative way to read and write the Cause and Flags fields of the FCSR using a single write operation. In this case the programmer would update the fields of the FEXR in a single write. Hardware would then move these updated fields into the FCSR.
- The Floating Point Enables Register (FENR) located at CP1 register 28, is an alternative way to read and write the exception type enables field, the rounding mode field, and the flush-to-zero field in the FCSR using a single write operation. In this case the programmer would update the fields of the FENR in a single write. Hardware would then move these updated fields into the FCSR.

8.2 Enabling the Floating-Point Unit

The Floating Point Unit is known as Coprocessor 1 (CP1). To enable CP1, set the CU1 bit in the CP0 *Status* register.

When this bit is cleared, Coprocessor 1 is disabled. Any attempt to execute a floating-point instruction causes a *Coprocessor Unusable* exception.

Floating Point Enable Code Example

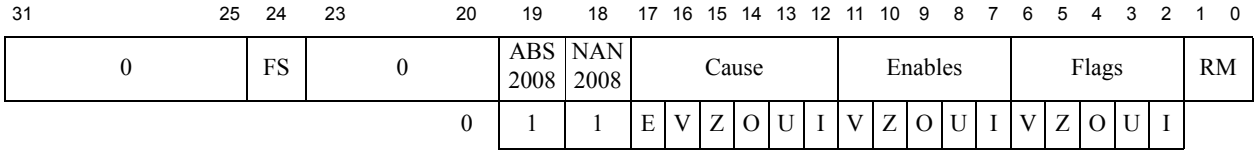
```
#define   C0_STATUS     $t12,0

mfc0   t0, C0_STATUS    //move CP0 Status register contents to register t0
li     t1, 0x20000000   //load value into t1 register with bit 29 set
or     t0, t1, t0       //logically OR contents of t0 and t1 and copy result into t0
mtc0   t0, C0_STATUS    //write t0 into CP0 Status register with bit 29 set
```

8.3 Setting a Floating Point Exception

The *Floating Point Control and Status Register* (FCSR located at CP1 register 31) is used to set and monitor floating point exceptions. The format of this register is shown in Figure 8.1.

Figure 8.1 FCSR Format



This register contains three fields that are used for the following purposes.

- Program the ‘Enables’ field in bits 11:7 to enable up to 5 types of exceptions as described in Table 8.1.
- If the ‘Enables’ field is programmed, use the ‘Cause’ field in bits 17:12 to determine the type of error once the exception occurs.
- Use the ‘Flags’ field in bits 6:2 is used when no exception conditions are enabled. It provides kernel software with the ability to check to see the type of error that occurred even though no exception type was enabled and no exception was taken.

Table 8.1 Cause, Enable, and Flag Field Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation. This bit exists only in the Cause field.

Table 8.1 Cause, Enable, and Flag Field Definitions *(continued)*

Bit Name	Bit Meaning
V	Invalid Operation. The Invalid Operation Exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed. When the Invalid Operation Exception is not enabled, the default floating-point result is a quiet NaN.
Z	Divide by Zero. The Divide by Zero Exception is signaled if and only if an exact infinite result is defined for an operation on finite operands. When the Divide by Zero Exception is not enabled, the default result is an infinity correctly signed according to the operation.
O	Overflow. The Overflow Exception is signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result, were the exponent range unbounded. When the Overflow Exception is not enabled, the overflowed rounded result is delivered to the destination. In addition, the Inexact bit in the Cause field is set.
U	Underflow. If enabled, the Underflow Exception is signaled when a tiny non-zero result is detected after rounding regardless of whether the rounded result is exact or inexact. Under default exception handling, i.e. when the Underflow Exception is not enabled, the rounded result is delivered to the destination and: • If the rounded result is inexact, the Inexact bit in the Cause field is set. • If the rounded result is exact, no bit in the Flags field is set. Such an underflow condition has no observable effect under default handling.
I	Inexact. Unless stated otherwise, if the rounded result of an operation is inexact -- that is, it differs from what would have been computed were both exponent range and precision unbounded -- then the Inexact Exception is signaled. When the Inexact Exception is not enabled, the rounded result is delivered to the destination.

8.4 Setting the Rounding Mode

To set the rounding mode for floating point operations, program the RM field (bits 1:0) in the Floating Point Control and Status Register (FCSR located at CP1 register 31) shown in Figure 8.1. The RM field is encoded as follows.

Table 8.2 Rounding Modes Definitions

RM Field Encoding	Meaning
0	Round to nearest / ties to even. Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even).
1	Round toward zero. Rounds the result to the value closest to, but not greater in magnitude than the result.
2	Round towards positive / plus infinity. Rounds the result to the value closest to, but not less than the result.

Table 8.2 Rounding Modes Definitions

RM Field Encoding	Meaning
3	Round towards negative / minus infinity. Rounds the result to the value closest to, but not greater than the result.

8.5 Operation of the FS Bit

The Flush to Zero (FS) bit in the *Floating Point Control and Status Register* (FCSR located at CP1 register 31) modifies the handling of denormalized operands.

If Flush to Zero is set, every input subnormal value and tiny non-zero result is replaced with zero of the same sign. In addition:

- Tiny non-zero results are detected before rounding. Flushing of tiny non-zero results causes Inexact and Underflow Exceptions to be signaled for all instructions, except the approximate reciprocals.
- Flushing of subnormal input operands in all instructions except comparisons causes an Inexact Exception to be signaled.
- For floating-point comparisons, the Inexact Exception is not signaled when subnormal input operands are flushed.

8.6 Programming the Floating Point FCSR Register

This section contains a programming example for programming the various bits of the FCSR as described in Sections 10.4 (Exceptions), 10.5 (Rounding Mode), and 10.6 (Flush-to-Zero bit).

In this example, the Cause and Flags fields shown in Figure 8.1 are set to 0, all exception types are enabled, the rounding mode is set to ‘round towards zero’, and the FS bit is set.

```
#define       C1_FCSR    $31

mfc1    t0, C1_FCSR      //move CP1 FCSR register contents to register t0
li      t1, 0x01000F81   //load value into t1 register with Cause field = 0,
                         //Enables field = 5'b11111, Flags field = 0, and Rounding
                         //Mode = 1
or      t0, t1, t0       //logically OR contents of t0 and t1 and copy result into t0
mtc1    t0, C1_FCSR      //write t0 into the CP1 FCSR register
```

Chapter 9

MIPS® SIMD Architecture (MSA)

This chapter describes the MIPS Single-Instruction-Multiple-Data (SIMD) architecture, known as MSA (MIPS SIMD Architecture). The SMA module adds more than 150 new instructions to the MIPS architecture that allow efficient parallel processing of vector operations.

The MSA provides increased system flexibility by incorporating a software-programmable solution for handling emerging CODECs or other functions not covered by the dedicated hardware in the device. Rather than focusing on narrowly defined instructions that must have optimized code written manually in assembly language in order to be utilized, the MSA is designed to accelerate compute-intensive applications in conjunction with leveraging generic compiler support. Applications such as data mining, feature extraction in video, image and video processing, human-computer interaction, and others, have some built-in data parallelism that lends itself well to SIMD.

The SIMD instructions are easily supported within high-level languages such as C or OpenCL, enabling fast and simple development of new code, as well as leverage of existing code.

This chapter provides a brief hardware overview of the MSA architecture, including how to map scalar floating point registers to MSA vector registers. Programming concepts include how to enable the MSA, MSA exception handling, a description of each field in the MSA Control register (MSACSR) and assembly language programming example, and GNU compiler support.

9.1 Overview of the SIMD Architecture

The MIPS® SIMD instructions operate on 32 vector registers of 8-, 16-, 32-, and 64-bit integer, 16-and 32-bit fixed-point, or 32- and 64-bit floating-point data elements. In the I6500 core, MSA implements 128-bit wide vector registers shared with the 64-bit wide floating-point unit (FPU) registers.

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™-2008. All standard operations are provided for 32-bit and 64-bit floating-point data. 16-bit floating-point storage format is supported through conversion instructions to/from 32-bit floating-point data.

9.1.1 MSA Instruction Formats

MSA instructions have 2- or 3-register, immediate, or element operands. One of the destination data format abbreviations shown in Table 9.1 is appended to the instruction name. Note that the data format abbreviation is the same regardless of the instruction's assumed data type. For example, all integer, fixed-point, and floating-point instructions operating on 32-bit elements use the same word (.W in Table 9.1) data format.

Table 9.1 Data Format Abbreviations

Data Format	Abbreviation
Byte, 8-bit	.B
Halfword, 16-bit	.H

Table 9.1 Data Format Abbreviations *(continued)*

Data Format	Abbreviation
Word, 32-bit	.W
Doubleword, 64-bit	.D
Vector	.V

The FPU contains thirty-two, 128-bit vector registers shared between SIMD and FPU instructions. SIMD instructions use the entire 128 bit register interpreted as multiple vector elements and relate to Table 9.1 as follows:

- 16 elements x 8-bits/element (.B)
- 8 elements x 16-bits/element (.H)
- 4 elements x 32-bits/element (.W)
- 2 elements x 64 bits/element (.D)
- 1 element x 128-bits (.V)

9.1.2 SIMD Instructions

In addition to Floating point instructions the Floating point Unit (FPU3) contains a full set of over 150 SIMD instructions that are compliant with the MIPS64® SIMD Architecture. For a complete list of all new SIMD instructions, refer to document MD00868, '*MIPS Architecture for Programmers Volume IV-j; The MIPS64® SIMD Architecture Module*'.

SIMD instructions enable:

- Efficient vector parallel arithmetic operations on integer, fixed-point and floating-point data.
- Operations on absolute value operands.
- Rounding and saturation options available.
- Full precision multiply and multiply-add.
- Conversions between integer, floating-point, and fixed-point data.
- Complete set of vector-level compare and branch instructions with no condition flag.
- Vector (1D) and array (2D) shuffle operations.
- Typed load and store instructions for endian-independent operation.

The FPU plus SIMD is fully synthesizable and operates at the same clock speed as the CPU.The I6500 core can issue up to two instructions per cycle to the FPU.

The FPU contains two execution pipelines for SIMD instruction execution. These pipelines operate in parallel with the integer core and do not stall when the integer pipeline stalls. This allows long-running SIMD operations to be partially masked by system stall and/or other integer unit instructions.

The FPU is optimized for SIMD performance. Most SIMD instructions have one cycle throughput.

9.1.3 MSA Vector Registers

The MSA operates on thirty-two 128-bit wide vector registers. If both MSA and the scalar floating-point unit (FPU) are present, the 128-bit MSA vector registers extend and share the 64-bit FPU registers.

MSA vector registers have four data formats: byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit).

9.1.4 Layout of MSA Registers

Figure 9.1 through Figure 9.4 show the vector register layout for elements of all four data formats.

Figure 9.1 MSA Vector Register Byte Elements (16 x 8)

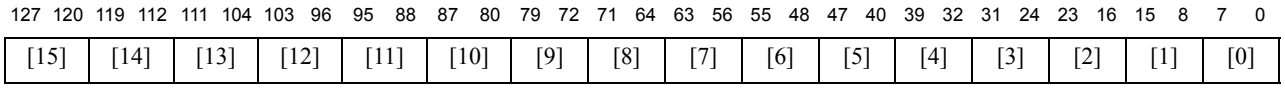


Figure 9.2 MSA Vector Register Halfword Elements (8 x 16)

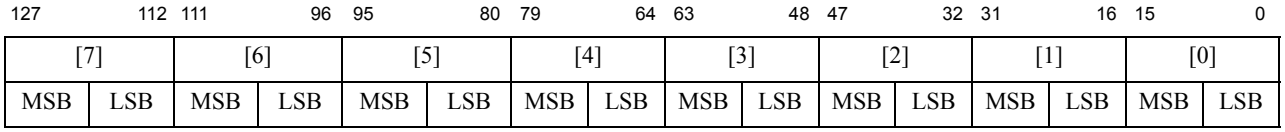


Figure 9.3 MSA Vector Register Word Elements (4 x 32)

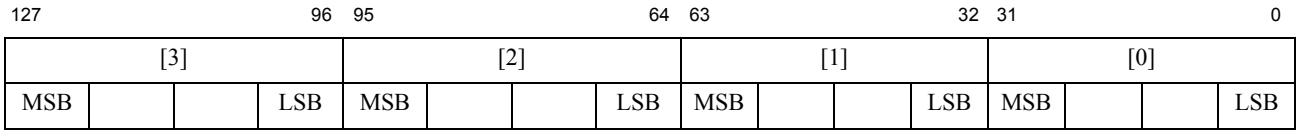
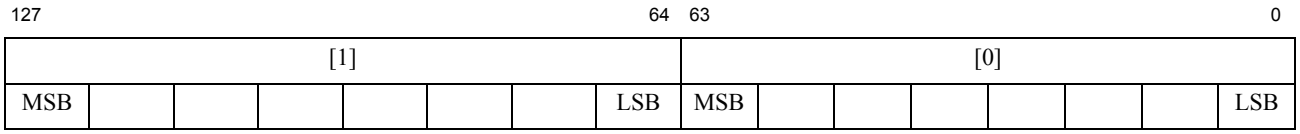


Figure 9.4 MSA Vector Register Doubleword Elements (2 x 64)



9.1.5 Mapping of Scalar Floating-Point Registers to MSA Vector Registers

The scalar floating-point unit (FPU) registers are mapped on the MSA vector registers. To facilitate register data sharing between scalar floating-point instructions and vector instructions, the FPU is required to use 64-bit floating-point registers operating in 64-bit mode.

The read and write operations for the FPU/MSA mapped floating-point registers are defined as follows:

- A read operation from the floating-point register *r,* where *r* = 0, …, 31, returns the value of the element with index 0 in the vector register *r*. The element's format is word for 32-bit (single precision floating-point) read or double for 64-bit (double precision floating-point) read.
- A 32-bit read operation from the high part of the floating-point register *r,* where *r* = 0, …, 31, returns the value of the word element with index 1 in the vector register *r*.
- A write operation of value *V* to the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the element with index 0 in the vector register *r* and all remaining elements are UNPREDICTABLE. Figure 9.5 and Figure 9.6 show the vector register *r* after writing a 32-bit (single precision floating-point) and a 64-bit (double precision floating-point) value *V* to the floating-point register *r*.
- A 32-bit write operation of value *V* to the high part of the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the word element with index 1 in the vector register *r,* **preserves** word element 0, and all remaining elements are UNPREDICTABLE. Figure 9.7 shows the vector register *r* after writing a 32-bit value *V* to the floating-point register *r*.

Figure 9.5 FPU Word Write Effect on the MSA Vector Register

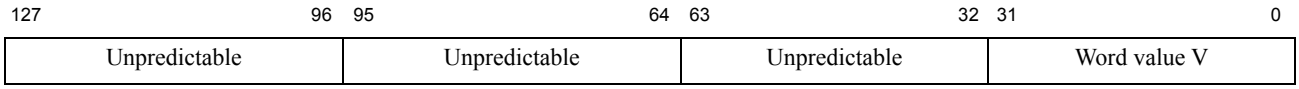


Figure 9.6 FPU Doubleword Write Effect on the MSA Vector Register

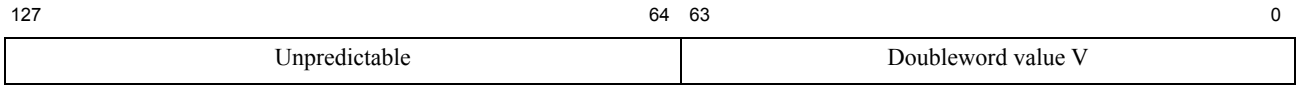
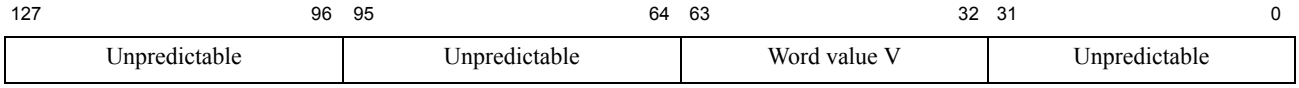


Figure 9.7 FPU High Word Write Effect on the MSA Vector Register



9.2 MSA Programming

The following subsections describes some programming elements of the MSA block.

9.2.1 Enabling MSA

Register Interface

The presence of the MIPS SIMD architecture (MSA) implementation is indicated by the state of the Config3.MSAP bit (CP0 Register 16, Select 3, bit 28) at reset. The MSAP bit is fixed by the hardware implementation and is read-only for the kernel software. Software can determine if MSA is implemented by checking if the MSAP bit is set, which is always the case in the I6500 core.

To enable the MSA block, the following CP0 register bits must be programmed:

- The Config5.MSAEn bit (CP0 Register 16, Select 5, bit 27) is used to enable access to the MSA instructions and the MSA vector registers. Executing a MSA instruction when MSAEn bit is not set causes a *MSA Disabled Exception*.
- The Status.CU1 bit must be set.

Enabling MSA Code Example

The following code example describes how to enable the MSA block using the register bits described above.

```
#include <mips/m32c0.h>
#include <mips/regdef.h>
#define   C0_STATUS    $12,0
#define   C0_CONFIG5   $16,5

//this code ensures that the CU1 and FR bits of the CP0 Status register are set.
mfc0   v1, C0_STATUS          //move contents of CP0 Status Register into v1
li     v0, SR_FR | SR_CU1     //OR the FR and CU1 bits and place result into v0
or     v1, v1, v0             //OR v1 and v0 and place result back into v1
mtc0   v1, C0_STATUS          //write result out to CP0 Status register with FR
                              //and CU1 bits set
ehb

//Set MSA enable bit in Config5.
mfc0   v1, C0_CONFIG5         //move CP0 Config5 register into v1
li     v0, CFG5_MSAEN         //load value into v0 that sets the MSAEN bit
or     v1, v1, v0             //OR v1 and v0 and place result back into v1
mtc0   v1, C0_CONFIG5         //write out result to CP0 Config5 with MSAEN bit set
ehb
```

This example is for a programmer writing their own code to enable the MSA block while writing a low-level support library, RTOS, or their own tool chain. However, this code is part of the MIPS CodeScape. As such, it is not necessary for the programmer to manually enable MSA when using CodeScape as this functionality is already built in to the CodeScape software.

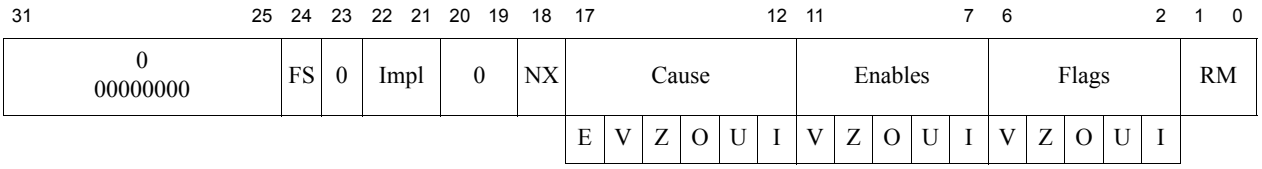
9.2.2 Setting a MSA Exception

The MSA Control and Status Register (*MSACSR*) is a 32-bit read/write register that controls the operation of the MSA unit. Figure 9.8 shows the format of the *MSACSR*.

The kernel software can read and write the *MSACSR* using the CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions.

The Floating Point Control and Status Register (*FCSR*, CP1 Control Register 31) and MSA Control and Status Register (*MSACSR*) are closely related in their purpose and register layout. However, each serves a different functional unit and can exist independently of the other.

Figure 9.8 MSACSR Register Format



This register contains three fields that are used for the following purposes.

- Program the ‘Enables’ field in bits 11:7 to enable up to 5 types of exceptions as described in Table 9.2.
- If the ‘Enables’ field is programmed, use the ‘Cause’ field in bits 17:12 to determine the type of error once the exception occurs.
- Use the ‘Flags’ field in bits 6:2 is used when no exception conditions are enabled. It provides kernel software with the ability to check to see the type of error that occurred even though no exception type was enabled and no exception was taken.

Table 9.2 Cause, Enable, and Flag Field Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation. This bit exists only in the Cause field.
V	Invalid Operation. The Invalid Operation Exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed. Under default exception handling, i.e. when the Invalid Operation Exception is not enabled, the default floating-point result is a quiet NaN.
Z	Divide by Zero. The Divide by Zero Exception is signaled if and only if an exact infinite result is defined for an operation on finite operands. Under default exception handling, i.e. when the Divide by Zero Exception is not enabled, the default result is an infinity correctly signed according to the operation.
O	Overflow. The Overflow Exception is signaled if and only if the destination format’s largest finite number is exceeded in magnitude by what would have been the rounded floating-point result, were the exponent range unbounded. Under default exception handling, i.e. when the Overflow Exception is not enabled, the overflowed rounded result is delivered to the destination. In addition, the Inexact bit in the Cause field is set.

Table 9.2 Cause, Enable, and Flag Field Definitions *(continued)*

Bit Name	Bit Meaning
U	Underflow. If enabled, the Underflow Exception is signaled when a tiny non-zero result is detected after rounding regardless of whether the rounded result is exact or inexact. Under default exception handling, i.e. when the Underflow Exception is not enabled, the rounded result is delivered to the destination and: • If the rounded result is inexact, the Inexact bit in the Cause field is set. • If the rounded result is exact, no bit in the Flags field is set. Such an underflow condition has no observable effect under default handling.
I	Inexact. Unless stated otherwise, if the rounded result of an operation is inexact -- that is, it differs from what would have been computed were both exponent range and precision unbounded -- then the Inexact Exception is signaled. Under default exception handling, i.e. when the Inexact Exception is not enabled, the rounded result is delivered to the destination.

Refer to Section 9.2.6, "Programming the MSA CSR Register" for more information on programming the exception types.

Refer to Section 9.3, "MSA Exceptions" for more information on MSA exception types.

9.2.3 Setting the Rounding Mode

To set the rounding mode for floating point operations, program the RM field (bits 1:0) in the *MSA Control and Status Register* (*MSACSR*) shown in Table 9.3. The RM field is encoded as follows.

Table 9.3 Rounding Modes Definitions

RM Field Encoding	Meaning
0	Round to nearest / ties to even. Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even).
1	Round toward zero. Rounds the result to the value closest to, but not greater in magnitude than the result.
2	Round towards positive / plus infinity. Rounds the result to the value closest to, but not less than the result.
3	Round towards negative / minus infinity. Rounds the result to the value closest to, but not greater than the result.

Refer to Section 9.2.6, "Programming the MSA CSR Register" for more information on setting the rounding mode.

9.2.4 Operation of the FS Bit

The Flush to Zero (FS) bit in the *MSA Control and Status Register* (*MSACSR*) modifies the handling of denormalized operands.

If Flush to Zero is set, every input subnormal value and tiny non-zero result is replaced with zero of the same sign. In addition:

- Tiny non-zero results are detected before rounding. Flushing of tiny non-zero results causes Inexact and Underflow Exceptions to be signaled for all instructions, except the approximate reciprocals.
- Flushing of subnormal input operands in all instructions except comparisons causes an Inexact Exception to be signaled.
- For floating-point comparisons, the Inexact Exception is not signaled when subnormal input operands are flushed.

Refer to Section 9.2.6, "Programming the MSA CSR Register" for more information on setting the FS bit.

9.2.5 Operation of the NX Bit

Setting the NX bit in the MSA CSR sets the MSA block in non-trapping floating point exception mode.

In normal exception mode, the destination register is not written and the floating point exceptions set the Cause bits and trap.

In non-trapping exception mode (NX bit set), the operations that would normally signal floating point exceptions do not write the Cause bits and do not trap. All the destination register's elements are set either to the calculated results or, if the operation would normally signal an exception, to signaling NaN values with the least significant 6 bits recording the specific exception type detected for that element in the same format as the Cause field. The Flags bits are updated for all floating-point operation with an IEEE exception condition that does not result in a MSA floating point exception (i.e., the Enable bit is off). This field is encoded as follows:

0: Normal exception mode.
1: Non-trapping exception mode.

Refer to Section 9.2.6, "Programming the MSA CSR Register" for more information on setting the NX bit.

For more information on the NX bit, refer to Section 9.3.2 "MSA Non-Trapping Exceptions".

9.2.6 Programming the MSA CSR Register

This section contains a programming example for programming the various bits of the MSACSR as described in Sections 11.4 (Exceptions), 11.5 (Rounding Mode), and 11.6 (Flush-to-Zero bit).

In this example, the Cause and Flags fields shown in Figure 9.8 are set to 0, all exception types are enabled, the rounding mode is set to 'round towards zero', and the FS bit is set.

```
cfcmsa    t0, $1           //move MSA CSR register contents to register t0
li        t1, 0x01040F81   //load value into t1 register with FS bit set, Cause
                           //field = 0, Enables field = 5'b11111, Flags field = 0,
                           //Rounding Mode = 1, and NX bit set
or        t0, t1, t0       //logically OR contents of t0 and t1 and copy result
                           //into t0
```

```
ctcmsa    t0, $1          //write t0 into the MSA CSR register
```

9.3 MSA Exceptions

FPU exceptions are implemented in the MIPS FPU/MSA architecture with the Cause, Enables, and Flags fields of the *FCSR/MSACSR*. The flag bits implement IEEE exception status flags, and the cause and enable bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions. The Cause field has an additional exception bit, Unimplemented Operation, used to trap for kernel software emulation assistance. If an exception type is enabled through the Enables field of the *FCSR/MSACSR*, then the FPU is operating in precise exception mode for this type of exception.

9.3.1 MSA Exception Types

MSA instructions can generate the following exceptions:

- Reserved Instruction, if bit Config3.MSAP (CP0 Register 16, Select 3, bit 28) is not set, or if the usable FPU operates in 32-bit mode; Status.CU1 (CP Register 12, Select 0, bit 29) is set. This exception uses the common exception vector with ExcCode field in Cause CP0 register set to 0x0A.
- Coprocessor Unusable, if CFCMSA or CTCMSA instructions attempt to read or write privileged MSA control registers without Coprocessor 0 access enabled. This exception uses the common exception vector with ExcCode field in Cause CP0 register set to 0x0B and CE field set to 0 to indicate Coprocessor 0.
- MSA Disabled, if bit Config5.MSAEn (CP0 Register 16, Select 5, bit 27) is not set or, when vector registers partitioning is enabled (i.e. MSAIR.WRP set), if any MSA vector register accessed by the instruction is either not available or needs to be saved/restored due to a context switch. This exception uses the common exception vector with ExcCode field in Cause CP0 register set to 0x15.
- MSA Floating Point, a data dependent exception signaled by the MSA floating point instruction. This exception uses the common exception vector with ExcCode field in Cause CP0 register set to 0x0E. The exact reason for taking this exception is in the Cause bits of the MSA Control and Status Register (MSACSR).

Table 9.4 Exception Codes

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
10	0x0a	RI	Reserved Instruction exception
11	0x0b	CPU	Coprocessor unusable exception
14	0x0e	MSAFPE	MSA Floating Point exception
21	0x15	MSADis	MSA Disabled exception

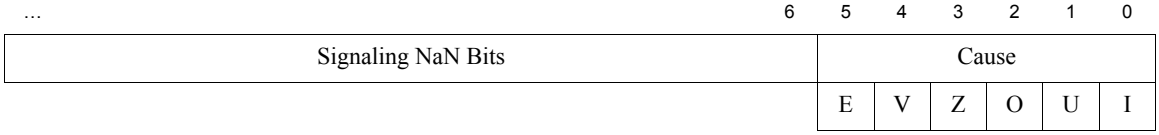
9.3.2 MSA Non-Trapping Exceptions

MSA provides a non-trapping exception mode (bit NX) that enables determining which element in the MSA vector caused the floating point exception.

In normal operation mode, floating point exceptions are signaled if at least one vector element causes an exception enabled by the Enable bit. There is no precise indication in this case on which elements are at fault and the corresponding exception causes. The exception handling routine should set the non-trapping exception mode bit NX and re-execute the MSA floating point instruction. All elements which would normally signal an exception according to the Enable bit-field are set to signaling NaN values, where the least significant 6 bits have the same format as the

Cause field (see Figure 9.9) to record the specific exception or exceptions detected for that element. The other elements will be set to the calculated results based on their operands.

Figure 9.9 Output Format for Faulting Elements when NX is Set



When the non-trapping exception mode bit NX is set, no floating point exception will be taken, not even the always enabled Unimplemented Operation Exception. Note that by setting the NX bit, the *MSACSR* Enable bit is not changed and is still used to generate the appropriate default results. Regardless of the NX value, if a floating point exception is not enabled, i.e. the corresponding *MSACSR* Enable bit is 0, the floating point result is a default value.

9.3.3 MSACSR Cause Register Field Update Pseudocode

The following pseudocode shows the process of updating the *MSACSR* Cause bits and setting the destination's value. This process is invoked element-by-element for all elements the instruction operates on. It is assumed *MSACSR* Cause bits are all cleared before executing the instruction. The *MSACSR* Flags bits are updated after all the elements have been processed and *MSACSR* Cause contains no enabled exceptions. If there are enabled exceptions in *MSACSR* Cause, a MSA floating-point exception will be signaled and the *MSACSR* flags are not updated.

$MSACSR_{Cause}$ Update Pseudocode

```
Input
   c: current element exception(s) E, V, Z, O, U, I bitfield
      (bit E is 0x20, O is 0x04, U is 0x02, and I is 0x01)
   d: default value to be used in case of a disabled exception
   e: signaling NaN value to be used in case of NX set, i.e. a non-trapping
      exception
   r: result value if the operation completed without an exception

Output
   v: value to be written to destination element
   Updated MSACSR_Cause

enable ← MSACSR_Enable | E /* Unimplemented (E) is always enabled */

/* Set Inexact (I) when Overflow (O) is not enabled */
if (c & O) ... 0 and (enable & O) = 0 then
   c ← c | I
endif

/* Clear Exact Underflow when Underflow (U) is not enabled */
if (c & U) ... 0 and (enable & U) = 0 and (c & I) = 0 then
   c ← c ^ U
endif

cause ← c & enable

if cause = 0 then
   /* No enabled exceptions, update the MSACSR Cause with all current exceptions */
   MSACSR_Cause ← MSACSR_Cause | c

   if c = 0 then
```

```
        /* Operation completed successfully, destination gets the result */
        v ← r
    else
        /* Current exceptions are not enabled, destination
           gets the default value for disabled exceptions case */
        v ← d
    endif
else
    /* Current exceptions are enabled */
    if MSACSR_NX = 0 then
        /* Exceptions will trap, update MSACSR Cause with all current exceptions,
           destination is not written */
        MSACSR_Cause ← MSACSR_Cause | c
    else
        /* No trap on exceptions, element not recorded in MSACSR Cause,
           destination gets the signaling NaN value for non-trapping exception */
        v ← ((e >> 6) << 6) | c
    endif
endif
```

MSACSR$_{Flags}$ Update and Exception Signaling Pseudocode

```
if (MSACSR_Cause & (MSACSR_Enable | E)) = 0 then /* Unimplemented (bit E 0x20)
                                                    is always enabled */
    /* No enabled exceptions, update the MSACSR Flags with all exceptions */
    MSACSR_Flags ← MSACSR_Flags | MSACSR_Cause
else
    /* Trap on the exceptions recorded in MSACSR Cause,
       MSACSR Flags are not updated */
    SignalException(MSAFPE, MSACSR_Cause)
```

9.4 MSA GNU Compiler Support

The GNU C Compiler (GCC) support for SIMD operations is based on a number of standard pattern names used for code generation. Ideally, the instruction set should implement as many of these operations as possible. In the process of MSA instruction selection and definition, supporting the standard GCC SIMD patterns was one of the most important objectives. Most of these patterns translate directly in single MSA instructions.

Another aspect related to efficient vector code compilation for SIMD architectures is the interoperability between the C language arrays (of scalar data types) and the native vector data types. To support seamless mixing of scalar and vector data types operations, the MSA provides a rich set of typed data transfer instructions.

9.4.1 MSA ABI

The O32 ABIs have been extended to allow efficient use of the vector registers and instructions defined by MSA. The MSA ABI extensions are compatible with the base ABIs in the sense that existing binaries run unchanged on systems supporting MSA. In other words, there are no incompatibilities between the base O32 ABI and the corresponding MSA extended ABI.

In particular, MSA ABI extensions;

- Do not change the base ABI data types layout / alignment

- Do not introduce new callee-saved (aka saved) registers
- Preserve the call-clobbered (aka temporary) or callee-saved (aka saved) status of the aliased floating-point registers.

However, vector data types are considered part of the MSA ABI by default and passed / returned by value without any MSA flags results in a compiler warning.

9.4.1.1 ABI Requirements

To be compatible with the MSA hardware, an ABI extension for MSA must support 32 64-bit floating point registers and a stack frame aligned to the size of the vector registers. The O32 FR1 ABI permits use of 64-bit floating point registers.

It is possible to adjust the stack alignment at run time using an existing compiler mechanism called dynamic stack realignment. Any ABI that does not meet the MSA stack alignment will therefore use dynamic stack re-alignment. For example, the 16-byte stack alignment of N32 and N64 ABIs is enough for MSA's 128-bit vector registers. However, the O32 ABI must perform dynamic stack re-alignment in this case.

9.4.1.2 Command Line Options and Function Attributes

Compiling for MSA (using the MSA defined instructions and vector registers) is enabled by the -mmsa command line option. A function compiled for MSA is referred to as a MSA function.

By default, the -mmsa option enables a faster calling convention for those functions passing vectors by value. This is achieved by using the vector registers for passing MSA vectors by value and returning MSA vector values.

A second MSA-related command line argument, -msimd-abi=none, can be used to disable the parameter passing/returning values in the vector registers. With -msimd-abi=none, all vector data types follow the calling conventions of the base ABI.

The use of vector types passed by value without the -mmsa option results in an ABI warning stating that a non-default ABI will be emitted. This warning can be disabled by explicitly passing the -msimd-abi=none option. It is illegal to use the -msimd-abi=msa option without -mmsa.

The functionality enabled by the command line option -mmsa can be disabled using -mno-msa. The SIMD ABI can be controlled by varying the value given to the -msimd-abi option. In particular, two SIMD ABIs are defined:

- none - Use the base calling convention
- msa - Use the MSA calling convention (default)

Equivalently, the same functionality could be enabled/disabled at the function level using __attribute__() as shown below.

- -mmsa __attribute__((msa))
- -mno-msa __attribute__((no_msa))
- -msimd-abi=none __attribute__((simd_abi_none))
- -msimd-abi=msa __attribute__((simd_abi_msa))

For convenience, pre-processor symbols are defined for each option as follows:

- -mmsa __MSA__

- -mno-msa __NO_MSA__
- -msimd-abi=none __SIMD_ABI_NONE___
- -msimd-abi=msa __SIMD_ABI_MSA__

9.4.1.3 Vector and Floating-Point Register Usage for -mmsa and -msimd-abi=msa

The MSA vector registers are temporary, and all live vector registers must be saved before calling a function. This ensures MSA functions can call any other function and compatibility with future MSA extensions.

The first 8 vector parameters are passed via vector registers w4 to w11 and vector results are returned via vector register w0. Floating-point registers are passed and returned as specified by the particular ABI.

For functions with variable arguments, no vector registers are used to pass vector parameters. This falls back to the original variable argument passing scheme from the particular ABI.

Note that compilers need to preserve the aliased callee-saved floating-point registers as specified by the O32 FR1, N32, and N64 ABIs: even f20, f22, ..., f30 for O32 FR1 and N32, and f24, f25, ..., f30, f31 for N64. For example, if the vector register w30 is used, the aliased floating point register f30 has to be preserved under all ABIs.

9.4.1.4 Inter-calling Between MSA and non-MSA Functions

A function that takes a MSA vector by value as a parameter or returns a MSA vector by value and is compiled with -mmsa can be called only by functions compiled with -mmsa.

Any function compiled with -msimd-abi=none can be called by non-MSA functions, i.e. a functions compiled under the base ABI with MSA disabled.

9.4.1.5 MSA GNU Options and Directives

The MSA is supported by the GNU tool chain starting with GAS (GNU Assembler) 2.22.51 and GCC 4.7.3. The command line options and assembly directives to enable/disable MSA are shown in Table 9.5.

The GCC options `-mfp64` and `-mhard-float` enforce the compatibility of the calling conventions of MSA and FPU, based on the fact that in the current release, MSA vector registers are shared with the 64-bit wide floating-point unit (FPU) registers.

Table 9.5 MSA GNU Options and Directives

	GAS		GCC	
	Enable	**Disable**	**Enable**	**Disable**
Command Line Options	`-mmsa`	`-mno-msa`	`-mmsa` -mfp64 -mhard-float	`-mno-msa`
Assembly Directives	`.set msa`	`.set nomsa`		

The GCC integer and floating-point vector data types with generic MSA operation support are listed in Table 9.6 and Table 9.7.

Table 9.6 GCC Integer Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of signed bytes	typedef signed char **wi8_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));
Vector of unsigned bytes	typedef unsigned char **wu8_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));
Vector of signed halfwords	typedef short **wi16_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));
Vector of unsigned halfwords	`typedef unsigned short` **`wu16_t`** `__attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of signed words	typedef int **wi32_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`
Vector of unsigned words	typedef unsigned int **wu32_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`
Vector of signed doublewords	typedef long long **wi64_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`
Vector of unsigned double-words	typedef unsigned long long **wu64_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`

Table 9.7 GCC Floating-Point Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of single precision floating-point values	typedef float **wf32_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`
Vector of double precision floating-point values	typedef double **wf64_t** __attribute__ ((vector_size(16))) `__attribute__ ((aligned(16)));`

MSA instructions are available to the C/C++ programmer either by the inline assembly `__asm__` directive, by `msa_`*`mnemonic`*`()` intrinsics, or when using most of the C/C++ operators on vector data types. The list of supported vector C/C++ operators include: `+`, `-`, `*`, `/`, `%`, `^`, `|`, `&`, `<<`, `>>`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `~`.

For example, adding or comparing two single-precision floating-point vectors, as in:

```
wi32_t t;
wf32_t a, b, c;

a = b + c;
t = b < c;
```

compiles directly in MSA word floating-point add and compare instructions:

```
fadd.w $w3,$w0,$w1 # a is in $w3, b in $w0, c in $w1
```

```
fclt.w $w4,$w0,$w1 # t is in $w4
```

Regarding the vector parameter passing conventions, MSA registers are all caller-saved, i.e. temporary registers are not preserved between function calls. The first eight vector parameters are passed in vector registers W4 to W11. When compiled for the MSA, the stack pointer is always aligned to 16 bytes.

9.4.2 MSA Vector Element Selection

MSA instructions select the n^{th} element in the vector register *ws* (*ws[n]* in assembly language) based on the data format *df*. Valid element index values for various data formats and vector register sizes are shown in Table 9.8.

Table 9.8 Valid Element Index Values

Data Format	Element Index
Byte	$n = 0, \ldots, 15$
Halfword	$n = 0, \ldots, 7$
Word	$n = 0, \ldots, 3$
Doubleword	$n = 0, 1$

9.4.3 Examples

Assume that vector registers W1 and W2 are initialized to the word values shown in Figure 9.10, Figure 9.11, and that general-purpose register R2 is initialized as shown in Figure 9.12.

Figure 9.10 Source Vector W1 Values

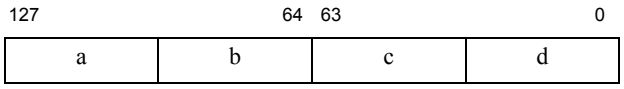


Figure 9.11 Source Vector W2 Values

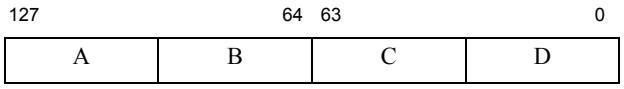
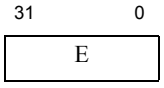


Figure 9.12 Source GPR 2 Value



Regular MSA instructions operate element-by-element with identical source, target, and destination data types. Figure 9.13 through Figure 9.16 have the resulting values of destination vectors W5, W6, W7, and W8 after executing the following sequence of word additions and move instructions with different types of operands.

```
addv.w $w5,$w1,$w2      //add two vector operands in $w1 and $w2 and move
                        //into $w5. The .w indicates at the 128-bit MSA registers
                        //are divided into four 32-bit words.
fill.w $w6,$2           //replicate contents of GPR register $2 into w6
addvi.w $w7,$w1,17      //vector immediate operand. Add immediate 17 into w1
                        //and move the result into w7
splati.w $w8,$w2[2]     //replicate word 2 of w2 into all elements of w8
```

Figure 9.13 Destination Vector W5 Value for ADDV.W Instruction

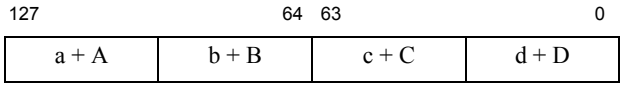


Figure 9.14 Destination Vector W6 Value for FILL.W Instruction

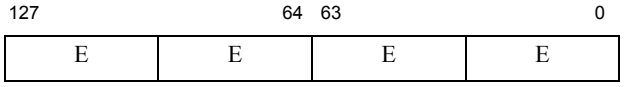


Figure 9.15 Destination Vector W7 Value for ADDVI.W Instruction

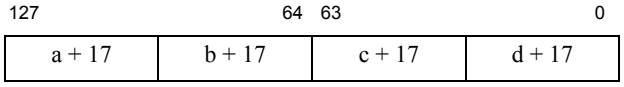
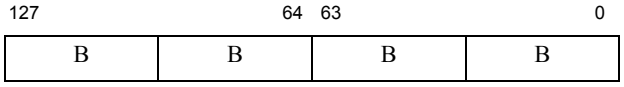


Figure 9.16 Destination Vector W8 Value for SPLAT.W Instruction

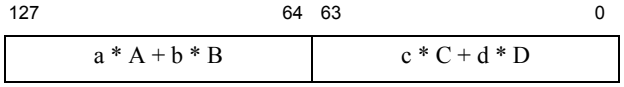


Other MSA instructions operate on adjacent odd/even source elements, generating results on data formats twice as wide. The signed doubleword dot product DOTP_S is such an instruction (see Figure 9.17):

```
dotp_s.d $w9,$w1,$w2
```

Note that the actual instruction specifies .D (doubleword) as the destination's data format. The data format of the source operands is inferred as being also signed and half the width, i.e. word, in this case.

Figure 9.17 Destination Vector W9 Value for DOTP_S Instruction



Chapter 10

Virtualization

The Virtualization (VZ) Module defines a set of new instructions, registers, and machine states to the I6500 core to mange the efficient implementation of virtualized systems. The Virtualization Module is designed to enable full virtualization of operating systems. The Virtualization Module allows for the execution of guest Operating Systems in a fully virtualized environment.

This chapter provides an overview of the VZ module, introduction to Root and Guest operating systems, and modes of operation, register structure in Guest mode, software detection of Virtualization, Guest address translation, exception handling in Root and Guest mode and interrupt handling, and an overview of Guest debug mode.

10.1 Overview

The Virtualization Module defines the following elements:

- Guest Operating Mode
- Partial CP0 register set (or context) for Guest Mode use
- Registers for Guest Mode control
- Guest interrupt system
- Detection of Virtualization features

The Virtualization Module provides separate Coprocessor 0 register sets (or contexts) for guest mode operation, which is physically separate from, and a subset of, the Root Coprocessor 0 context.

10.1.1 Root and Guest Operating Modes

The virtualization module contains a operating modes for one **Root** and multiple **Guests**. The non-guest operating mode is known as **root mode**. The pre-existing kernel, user and supervisor operating modes can be referred to as **root-kernel**, **root-user** and **root-supervisor** respectively, to distinguish them from their guest-mode equivalents.

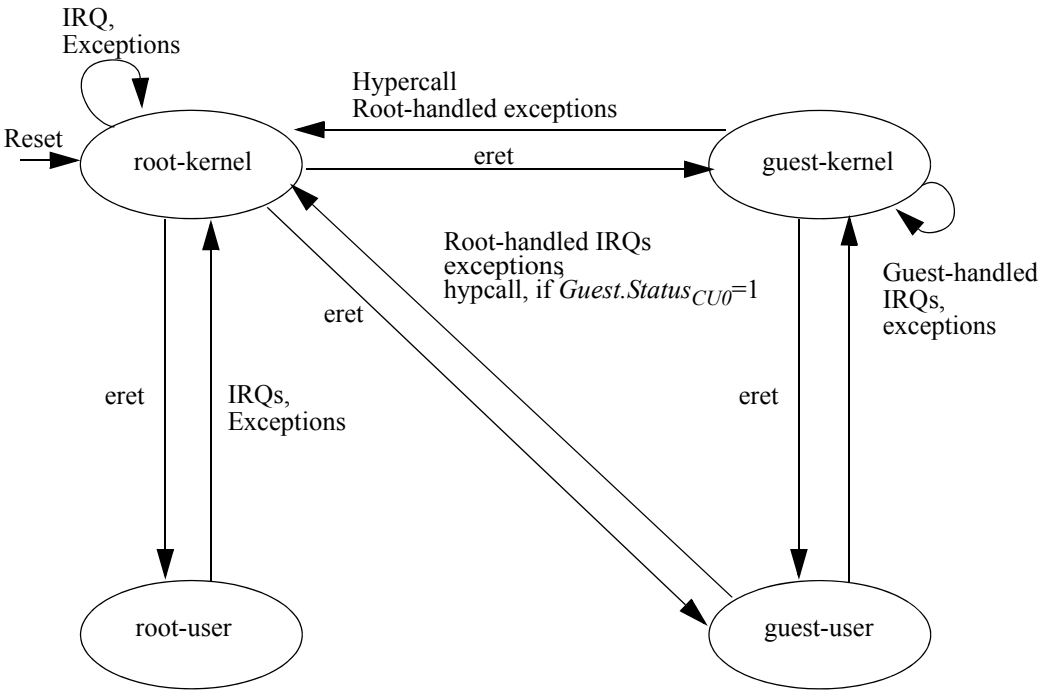
Guest mode consists of new operating modes guest-kernel, guest-user and guest-supervisor modes. The guest mode allows the separation between kernel, user and supervisor modes to be retained for a guest operating system running within a virtual machine. The guest-kernel mode can handle interrupts and exceptions, and manage virtual memory for guest-user mode processes.

The separation between root mode and the limited-privilege guest mode allows root mode software to be in full control of the machine at all times even when a guest is running. Backward compatibility is retained for existing kernel software running in root mode.

The *GuestCtl0* register contains the GM (Guest Mode) bit. This bit is used along with root-mode exception and error status bits ($Status_{EXL}$, $Status_{ERL}$) and the Debug Mode bit ($Debug_{DM}$) to determine whether the processor is operating in guest mode or root mode.

Figure 10.1 shows the state transitions between operating modes.

Figure 10.1 State Transitions Between Operating Modes



10.1.2 Introduction to the Hypervisor

Virtualization is enabled by kernel software. The key element is a control program known as a Virtual Machine Monitor (VMM) or 'Hypervisor'. The Hypervisor is in full control of machine resources at all times. When an operating system (OS) kernel is run within a virtual machine (VM), it becomes a 'guest' of the Hypervisor. All operations performed by a guest must be explicitly permitted by the Hypervisor. To ensure that it remains in control, the Hypervisor always runs at a higher level of privilege than a guest operating system kernel. The hypervisor is responsible for managing access to sensitive resources, maintaining the expected behavior for each VM, and sharing resources between multiple VMs.

In a traditional operating system, the kernel (or 'supervisor') typically runs at a higher level of privilege than user applications. The kernel provides a protected virtual-memory environment for each user application, inter-process communications, and I/O device sharing. The hypervisor performs the same basic functions in a virtualized system - except that the Hypervisor's clients are full operating systems rather than user applications.

The virtual machine execution environment created and managed by the Hypervisor consists of the full Instruction Set Architecture, including all Privileged Resource Architecture facilities, plus any device-specific or board-specific peripherals and associated registers. It appears to each guest operating system as if it is running on a real machine with full and exclusive control.

The Virtualization Module enables full virtualization, and is intended to allow VM scheduling to take place while meeting real-time requirements, and to minimize costs of context switching between VMs.

In virtualization, the guest operating system operates in unprivileged mode. All privileged operations attempted by the guest traps back to the Hypervisor, which executes in the privileged mode. The Hypervisor emulates all guest privileged operations, keeps track of the guest view of privileged state, and ensures that the system behaves as

expected by the guest. Full address translation allows an unmodified guest kernel to execute from its original location in memory, and allows the hypervisor to manage address translation to match the expectations of the guest kernel.

10.1.3 Enabling Guest Mode Translations

The Virtualization Module in the I6500 core provides a separate CP0 register set and MMU for guest-mode execution. In guest mode two levels of address translation are performed as described above.

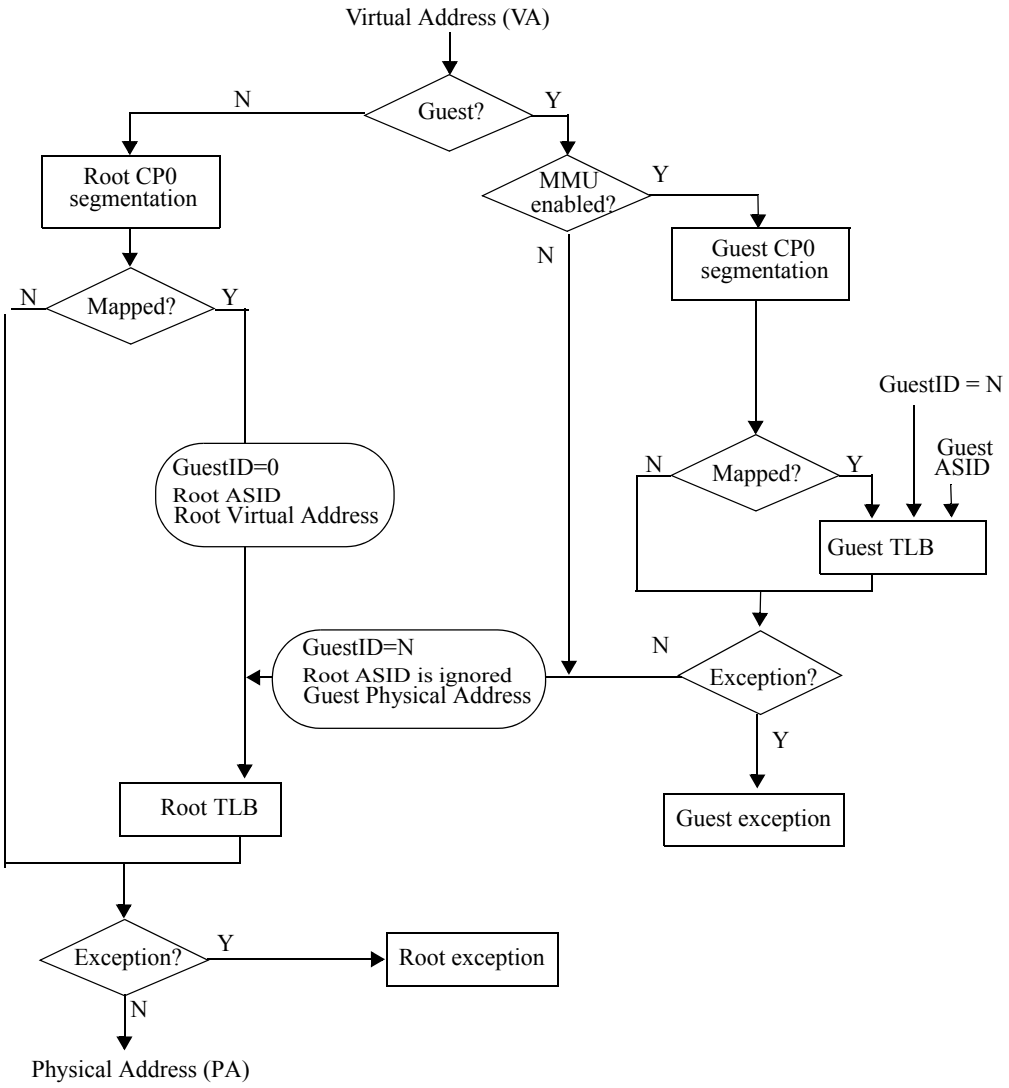
10.1.4 MMU Considerations

For the TLB-based guest MMU, MIPS recommends that the number of entries be equal to the number of entries in the root-context TLB used for Guest mappings. The page sizes used in the root-mode TLB must be carefully considered to allow sufficient control for root-mode software, while maximizing the number of guest-mode TLB entries which are mapped through each root-mode TLB entry. Larger root TLB pages will likely result in better performance.

Both the guest and root MMU's can be active at the same time. MIPS recommends that the Root TLB maintain an adequate amount of reserved TLB entries for its own use to avoid cascading TLB evictions (thrashing).

Figure 10.2 shows the outline of address translation in the Virtualization Module.

Figure 10.2 Outline of Address Translation



Guest mode segmentation controls and the guest mode MMU have no effect on the root mode address space.

10.1.5 Guest ID

The ‘GuestID’ field (*GuestCtl1*$_{ID}$ or *GuestCtl1*$_{RID}$) represents a unique identifier for Root and all Guest Virtual Address spaces. Each Guest’s address space is identified by a unique non-zero GuestID. The GuestID value zero is reserved for Root address space. The *GuestCtl1* CP0 register is unique in the Root register space and inaccessible in guest mode. GuestID is an optimization, designed to minimize TLB invalidation overhead on a virtual machine context switch and simplify Root access to Guest TLB entries.

10.1.6 CP0 Structure in Root and Guest Mode

In the I6500 core, Coprocessor 0 (CP0) contains system control registers and can be accessed only by privileged instructions. The presence of virtualization in the I6500 core means that a subset of the Coprocessor 0 register set are physically replicated for use by the Guest Operating System.

During guest mode execution, both the guest Coprocessor 0 and the root Coprocessor 0 are active. The presence of two simultaneously active Coprocessor 0 contexts is fundamental to the operation of the Virtualization Module. The presence of these two sets of Coprocessor 0 (CP0) registers allows for an immediate switch between guest and root modes without requiring a context switch to/from memory. Simultaneously accesses to the guest and root Coprocessor 0 registers allows guest-kernel privileged code accesses to execute with the minimum hypervisor intervention, and ensures that key root-mode machine systems such as timekeeping, address translation and external interrupt handling continue to operate without major changes during guest execution.

Table 10.1 describes the how the various CP0 register fields are used to enter or exit an operating mode.

Table 10.1 Guest, Root and Debug Modes

<table>
<tr><th colspan="5">Root</th><th colspan="3">Guest</th><th rowspan="2">Mode</th></tr>
<tr><th>Debug$_{DM}$</th><th>Status$_{ERL}$</th><th>Status$_{EXL}$</th><th>Status$_{KSU}$</th><th>GuestCtl0
GM</th><th>Status$_{ERL}$</th><th>Status$_{EXL}$</th><th>Status$_{KSU}$</th></tr>
<tr><td>1</td><td colspan="7">Don’t care</td><td>Debug</td></tr>
<tr><td rowspan="8">0</td><td>1</td><td colspan="6">Don’t care</td><td>Root-Kernel</td></tr>
<tr><td rowspan="7">0</td><td>1</td><td colspan="5">Don’t care</td><td></td></tr>
<tr><td rowspan="3">0</td><td>00</td><td rowspan="3">0</td><td colspan="3" rowspan="3">Don’t care</td><td></td></tr>
<tr><td>01</td><td>Root-Supervisor</td></tr>
<tr><td>10</td><td>Root-User</td></tr>
<tr><td rowspan="4">Don’t care</td><td rowspan="4">1</td><td>1</td><td colspan="2">Don’t care</td><td rowspan="2">Guest-Kernel</td></tr>
<tr><td>0</td><td>1</td><td>Don’t care</td></tr>
<tr><td rowspan="2">0</td><td>00</td><td></td></tr>
<tr><td>01</td><td>Guest-Supervisor</td></tr>
<tr><td colspan="2">Don’t care</td><td>10</td><td>Guest-User</td></tr>
<tr><td colspan="2">Don’t care</td><td>11</td><td>UNPREDICTABLE</td></tr>
<tr><td colspan="2">Don’t care</td><td>11</td><td colspan="3">Don’t care</td><td>UNDEFINED</td></tr>
</table>

10.1.7 New CP0 Registers

Coprocessor 0 registers are added by the Virtualization Module to control the guest context. Table 10.2 describes CP0 registers introduced by the Virtualization Module. Refer to Chapter 2 of this manual for more information.

Table 10.2 CP0 Registers Introduced by the Virtualization Module

Register Number	Sel	Register Name	Description
12	6	GuestCtl0	Controls guest mode behavior.
10	4	GuestCtl1	Guest ID
10	5	GuestCtl2	Virtual Interrupts
11	4	GuestCtl0Ext	Extension to GuestCtl0
12	7	GTOffset	Offset for guest timer value

10.1.8 New CP0 Instructions

The Virtualization Module introduces new instructions for root mode access to the guest CP0 context, and for a guest to make a call into root mode - a ‘hypervisor call’.

Table 10.3 describes CP0 instructions introduced by the Virtualization Module.

Table 10.3 CP0 Instructions Introduced by the Virtualization Module

Instruction	Description
HYPCALL	Hypercall - call to root mode.
MFGC0	Move from Guest CP0
MTGC0	Move to Guest CP0
DMFGC0	Doubleword Move from Guest CP0
DMTGC0	Doubleword Move to Guest CP0
GINVGT	Global Invalidate Guest TLB
TLBGINV	Guest TLB Invalidate
TLBGINVF	Guest TLB Invalidate Flush
TLBGP	Probe Guest TLB
TLBGR	Read Guest TLB
TLBGWI	Write Guest TLB
TLBGWR	Write Random to Guest TLB

10.2 Software Detection of Virtualization

Software can determine if the Virtualization Module is implemented by checking the state of the VZ bit in the *Config3* CP0 register. If Virtualization is supported ($Config3_{VZ} = 1$), and GuestID is supported, then explicit invalid TLB entry support (EHINV) is required in order for a Guest to be able to detect invalid entries in the Guest TLB.

Figure 10.3 Config3 Register Format

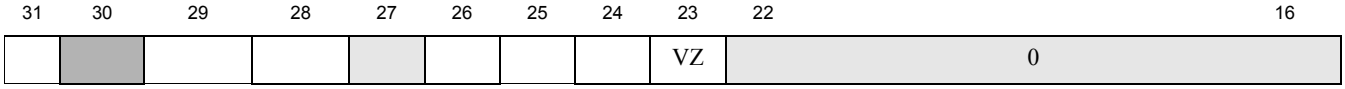


Table 10.4 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
VZ	23	Virtualization Module implemented. This bit indicates whether the Virtualization Module is implemented. This bit is always 1 for the I6500 core. 0: Virtualization module not implemented 1: Virtualization module is implemented	R	1

10.3 Modes Of Operation

10.3.1 Root Mode Operation

Root mode operation uses one set of Coprocessor 0 registers and Guest mode operation the other. The software visible state is the contents of these registers and any state which is accessed via these registers, such as TLB entries and Segmentation Control configurations.

For a Hypervisor to save, restore or switch context from one guest to another, it is the entire visible state which must be saved and restored, not solely the replicated registers themselves, but also the physical resources which are shared between Root and Guest, such as the GPRs, FPRs and Hi/Lo registers.

The following subroutine can be used to test whether processor is in root-mode.

```
subroutine IsRootMode() :
    if (
        (GuestCtl0GM=0) or
        ((GuestCtl0GM=1) and not ((Root.DebugDM=0) and
        (Root.StatusERL=0) and (Root.StatusEXL=0))
        ) then
        return(true)
    else
        return(false)
    endif
endsub
```

10.3.2 Guest Mode Operation

In guest mode, all guest operations are first tested against the guest CP0 context, and then against the root CP0 context. An ‘operation’ is any process which can trigger an exception. This includes address translation, instruction fetches, memory accesses for data, instruction validity checks, coprocessor accesses and breakpoints.

Guest mode software has no access to the root Coprocessor 0. Root mode software can access the guest Coprocessor 0, and if required can emulate guest-mode accesses to disabled or unimplemented features within guest Coprocessor 0. The guest Coprocessor 0 is partially populated - only a subset of the complete root Coprocessor 0 is implemented.

The recommended method of entering Guest mode is by executing an ERET instruction when *Root.GuestCtl0*$_{GM}$=1, *Root.Status*$_{EXL}$=1, *Root.Status*$_{ERL}$=0 and *Root.Debug*$_{DM}$=0.

Guest mode operation is determined as follows. This subroutine can be used to test whether processor is in guest-mode.

```
subroutine IsGuestMode() :
    if (GuestCtl0GM=1) and (Root.DebugDM=0) and
        (Root.StatusERL=0) and (Root.StatusEXL=0) then
        return(true)
    else
        return(false)
    endif
endsub
```

10.3.3 Debug Mode

For processors that implement OCI debug, the processor is operating in debug privileged execution mode (Debug Mode) when *Root.Debug*$_{DM}$=1. If the processor is running in Debug Mode, it has full access to all resources that are available to Root Kernel Mode operation.

Debug Mode, Root Mode and Guest Mode are mutually exclusive. At any given time, the processor can only be in one of the three modes. Note that Debug mode operates in the Root context, while Guest mode operates in its own unique context.

10.4 Address Translation Pseudocode

The following pseudocode describes the complete address translation process for the I6500 Virtualization Module. Segmentation, TLB lookups, hardware TLB refill and second-level address translation are invoked. The process is described in top-down order - subsequent sections describe the subroutines called.

```
/* Inputs
 * vAddr  - Virtual Address
 * IorD   - Access type - INSTRUCTION or DATA
 * LorS   - Access type - LOAD or STORE
 * pLevel - Privilege level - USER, SUPER, KERNEL
 *
 * Outputs
 * pAddr  - physical address
 * CCA    - cache attribute (valid when mapped)
 *
 * Exceptions: See called functions
 * Called from guest or root context.
 */
subroutine AddressTranslation(vAddr, IorD, LorS, pLevel)

   // Initialization.
   // GuestID is only applicable if GuestCtl0_RAD=0. Otherwise GuestID
   // is ignored (not applicable) in process of address translation.
   GuestID ← ignored

   if (IsGuestMode()) then
      // This is a Guest Address translation
      // step 1: Guest Virtual -> Guest Physical Address translation
      if (GuestCtl0_RAD=0)
            GuestID ← GuestCtl1_ID
      endif
      (mapped, addr, CCA) ← AddressDecode(vAddr, pLevel)
      if (Config_MT=1 or Config_MT=4) then // TLB type MMU
         if (mapped) then
            asid ← Guest.EntryHi_ASID
            (addr, CCA) ← Guest.TLBLookup(asid, GuestID, addr, IorD, LorS)
         endif
      else
         if (Config_MT=0) then
            # MMU=None case is undefined
            UNDEFINED
         else
            # Other MMU type, FMT or BAT. BAT will use LorS.
            (addr, CCA) ← Guest.OtherMMULookup(addr, CCA, LorS, pLevel)
```

```
            endif
        endif
        if (exception)
            Guest Exception
            // TLB exceptions may include Refill, Invalid, Execute-Inhibit for
            // Instruction, Refill, Invalid, Modified, Read-Inhibit for Data.
            // Guest segment map related exceptions may include Address Error
        endif

        // step 2: Guest Physical -> Root Physical Address translation
        // if GuestCtl0_RAD=0, then guest entry ASID is global in Root TLB.
        // H/W must set G=1 for guest entry for TLBWI and TLBWR.
        asid ← Root.EntryHi_ASID
        pAddr ← Root.TLBLookup(asid, GuestID, addr, IorD, LorS)
        if (exception)
            Root Exception
            // This is a Root exception initiated in guest context
            // This includes all TLB exceptions.
            // Segment map Address Error exception not included, as guest does not
            // lookup root segment map.
        endif

    else
        // This is a Root Address translation
        // Root Virtual -> Root Physical Address translation
        // If GuestCtl0_DRG=1,GuestCtl1_RID is non-zero,Root.Status_EXL,ERL=0,
        // and Debug_DM=0, then all root kernel data accesses are mapped and root
        // SegCtl is ignored.H/W must set G=1 as if the access were for guest.
        drg_valid ← (GuestCtl0_DRG=1 and Root.Status_KSU=00 and Root.Status_EXL=0 and
        Root.Status_ERL=0 and Debug_DM=0 and GuestCtl1_RID!=0 and !Instruction)
        if (drg_valid) then
            mapped ← 1
            addr ← vAddr
        else
            (mapped, addr, CCA) ← AddressDecode(vAddr, pLevel)
        endif
        if (!mapped) then
            pAddr ← addr
        else   if (GuestCtl0_RAD=0)
                    if (Instruction or (!drg_valid))
                            GuestID ← 0
                    else
                            GuestID ← GuestCtl1_RID
                    endif
                endif
            asid ← Root.EntryHi_ASID
            (pAddr, CCA) ← Root.TLBLookup(asid, GuestID, addr, IorD, LorS)
        endif
    endif
    if (exception)
        Root Exception
        // Includes all TLB and Segment related exceptions in Root context.
        // If drg_valid, and access is not by root-kernel,then an Address Error
        // exception is caused.
    endif

    return (pAddr,CCA)
end
```

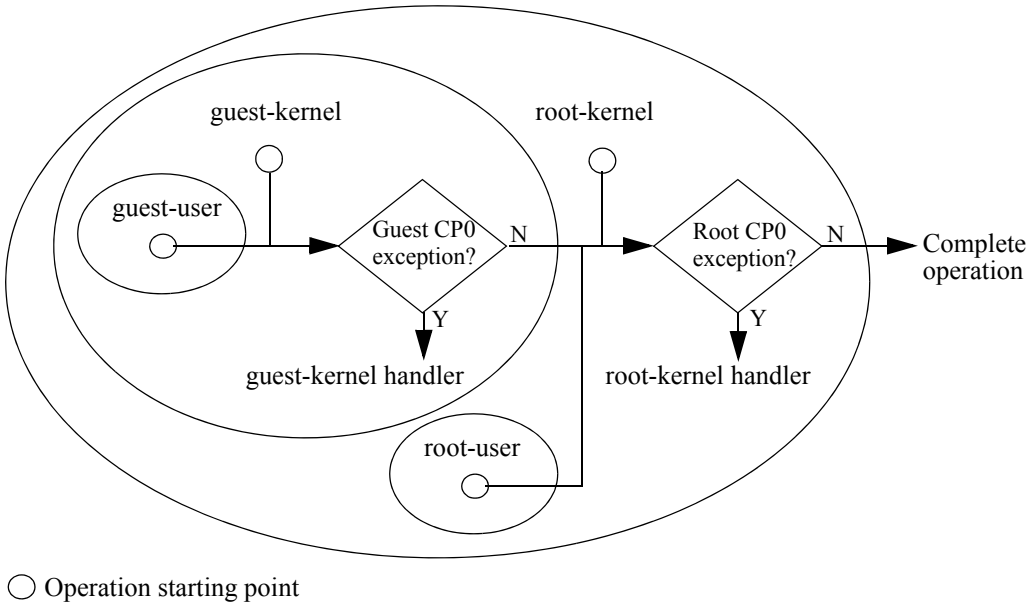
```
subroutine AddressDecode(vAddr, pLevel) :
    # Determine whether address is mapped
    # - if unmapped, obtain physical address and cache attribute
    if (Config3SC) then
        // optional Segmentation Control based address decode
        (mapped, addr, CCA) ← SegmentLookup(vAddr, pLevel)
    else
        (mapped, addr, CCA) ← LegacyDecode(, pLevel)
    endif
    return (mapped, addr, CCA)
endsub
```

10.5 Exception Handling in Root and Guest Mode

Exceptions are handled in the mode whose context triggered the exception. An exception triggered by the guest CP0 context is handled in guest mode. An exception triggered by the root CP0 context is handled in root mode.

Figure 10.4 shows the how exceptions are handled in each of the operating modes (supervisor modes are omitted for clarity).

Figure 10.4 Exception Handling in Root and Guest Mode



In Figure 10.4, an operation executed in guest-user mode must travel through the root kernel to complete the operation.

The first layer to be crossed is the guest CP0 context (controlled by guest-kernel mode software). All exception and translation rules defined by the guest CP0 context are applied, and resulting exceptions are taken in guest mode by the guest kernel handler.

If the operation does not trigger a guest-context exception, the next layer to be crossed is the root CP0 context (controlled by root-kernel mode software). All exception and translation rules defined by the root CP0 context are applied, and resulting exceptions taken in root mode by the root kernel handler as shown.

For example, an access to Coprocessor 1 (the Floating Point Unit) must first be permitted by the guest context $Status_{CU1}$ bit, and then by the root context $Status_{CU1}$ bit. However, access of guest to Coprocessor 0 is not qualified by root context $Status_{CU0}$ as Coprocessor 0 state is not shared with root.

Table 10.5 specifies the association of GuestID with TLB instructions. For supporting information, refer to Section 10.1.8.

Table 10.5 GuestID Use by TLB Instructions

TLB Operation	GuestID ($GuestCtl1_{ID}$/$GuestCtl1_{RID}$)
GINVGT	$GuestCtl1_{RID}$
TLBGINV	$GuestCtl1_{RID}$
TLBGINVF	$GuestCtl1_{RID}$
TLBGP	$GuestCtl1_{RID}$
TLBGR	$GuestCtl1_{RID}$
TLBGWI	$GuestCtl1_{RID}$
TLBGWR	$GuestCtl1_{RID}$
TLBINV	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBINVF	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBP	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBR	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBWI	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBWR	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
GINVT	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$

10.5.1 Root and Guest Shared TLB Operation

The I6500 core shares a common physical TLB amongst root and guest. The I6500 core contains a TLB structure that incorporates a VTLB (Variable page size TLB) and FTLB (Fixed page size TLB). As such, the VTLB must accommodate wired entries for both root and guest in a shared structure.

10.5.1.1 Root and Guest Access to the Shared TLB

In a shared TLB implementation, the root index increases from the bottom of the physical TLB while the guest index increases from the top of the physical TLB. This is to avoid overlap of root and guest wired entries. On the other

hand, the root and guest indices to the FTLB grow from the bottom of the FTLB. Both guest and root TLB operations must interpret the TLB index accordingly.

10.5.1.2 Wired Register Management

The Root allocates the appropriate number of wired entries to itself, and then writes the guest *Config1* and *Config4* related fields to set the available VTLB entries for guest. The Root then reads the $Guest.Config4_{MMUExtDef}$ field to determine which of the guest *Config4* MMU size extension fields need to be written. Since the entries allocated for guest use also includes non wired entries shared by both root and guest, root software must be careful not to allocate all remaining non root-wired entries to the guest. This prevents the guest from populating all remaining non root-wired entries with its own guest-wired entries, leaving no entries for non root-wired entries.

Root software should not change guest MMU configuration while the guest is in operation, as is the case for any guest configuration that is read-only to guest but writeable by root.

10.5.1.3 CP0 Register Allocation

The Virtualization Module provides a partial set of CP0 registers for use by the guest. This is known as the *guest context.* When in guest mode, the behavior of the machine is controlled by the combination of the guest CP0 context and the root CP0 context. When in root mode, the behavior of the machine is controlled entirely by the root CP0 context.

The guest CP0 context consists of a base set plus optional features. Access to features within the guest CP0 context is controlled from root mode. The *Guest.Config0* through *Guest.Config7* registers determine which features are active during guest mode execution. The *GuestCtl0* register controls whether a guest access to a privileged feature triggers an exception.

10.5.1.4 CP0 Register Access

Guest CP0 registers can be accessed from root mode by using the root-only *MFGC0* and *MTGC0* instructions. Guest TLB contents can be accessed by using the root-only *TLBGP, TLBGR, TLBGWI* and *TLBGWR* instructions.

10.5.1.5 CP0 Register Initialization and Control

Root context software (hypervisor) is required to manage the initial state of writable Guest context registers. On power-up, the initial state defaults to the hardware reset state. On a Guest context save and restore, the hypervisor is required to preserve and re-initialize the Guest state. For virtual boot of a Guest, the hypervisor is required to initialize the Guest state equivalent to the hardware reset state. The Root may deconfigure one or more guest CP0 registers by writing to the guest configuration registers.

The Virtualization Module requires that scratch registers *KScratch1* and *KScratch2* are present in the root context. This ensures that hypervisor exception handlers have an adequate number of scratch registers to save and restore all general purpose registers in use by the guest.

10.6 Exceptions

Normal execution of instructions can be interrupted when an exception occurs. Such events can be generated as a by-product of instruction execution (e.g., an integer overflow caused by an add instruction or a TLB miss caused by a load instruction), by an illegal attempt to use a privileged instruction (e.g. MTC0 from user mode), or by an event not directly related to instruction execution (e.g., an external interrupt).

When an exception occurs, the processor stops processing instructions, saves sufficient state to resume the interrupted instruction stream, enters Exception or Error mode, and starts a software exception handler. The saved state and the

address of the software exception handler are a function of both the type of exception, and the current state of the processor.

10.6.1 Exceptions in Guest Mode

The Virtualization Module adds new rules for the processing of exception conditions detected during guest-mode execution.

The ‘onion model’ requires that every guest-mode operation be checked first against the guest CP0 context, and then against the root CP0 context. Exceptions resulting from the guest CP0 context can be handled entirely within guest mode without root-mode intervention. Exceptions resulting from the root-mode CP0 context (including *GuestCtl0* permissions) require a root mode (hypervisor) handler.

During guest mode execution, the mode in which an exception is taken is determined by the following:

- Guest-mode operations must first be permitted by guest-mode CP0 context and then by root mode CP0 context
    - This includes all operations for which exceptions can be generated - memory accesses, coprocessor accesses, breakpoints and so forth.
- Exceptions are always taken in the mode whose CP0 state triggered the exception
    - When architecture features in the guest context are present and enabled by the *Guest.Config* registers, exceptions triggered by those features are taken in guest mode.
    - Exceptions resulting from control bits set in the *Root.GuestCtl0* register, and exceptions resulting from address translation of guest memory accesses through the root-mode TLB are taken in root mode.

Asynchronous exceptions such as Reset, NMI, Memory Error, Cache Error are taken in root mode. External interrupts are received by the root CP0 context, and if enabled are taken in root mode. If an interrupt is not enabled in root mode and is bypassed to the guest CP0 context, and is enabled in the guest CP0 context, the interrupt is taken in guest mode.

When an exception is detected during guest mode execution, any required mode switch is performed after the exception is detected and before any machine state is saved. This allows machine state to be saved to either the root or guest contexts, and allows the exception to be handled in the proper mode. See also Section 10.6.2.

```
# Booleans, indicating source of exception:
#   root_async       - Asynchronous root context exception
#   root_sync        - Synchronous exception triggered by root context
#   guest_async      - Asynchronous exception triggered by guest context
#   guest_sync       - Synchronous exception triggered by guest context
#
# Exceptions directed to root context set Root.Status.ERL or Root.Status.EXL,
# meaning that the processor executes the handler in root mode.

# Ordering of exception conditions
if (root_async) then
    ctx ← Root
elsif (guest_async) then
    ctx ← Guest
elsif (guest_sync) then
    ctx ← Guest
elsif (root_sync) then
    ctx ← Root
```

```
else
    ctx ← null
endif
```

10.6.2 Faulting Address for Exceptions from Guest Mode

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions.

- Address error
- TLB Refill
- TLB Invalid
- TLB Modified
- TLB Execute Inhibit
- TLB Read Inhibit

10.6.3 Guest Initiated Root TLB Exception

When an exception is triggered as a result of a root TLB access during guest-mode execution, the handler executes in root mode, and exception state is stored into root CP0 registers. The registers affected are *GuestCtl0, Root.EPC*, *Root.BadVAddr*, *Root.EntryHi*, *Root.Cause* and $Root.Context_{BadVPN2}$.

The faulting address value stored into *Root.BadVAddr* and $Root.Context_{BadVPN2}$ is ideally the Guest Physical Address (GPA) presented to the root TLB by the guest context. A Guest Virtual Address (GVA) unmapped by the Guest MMU is considered a GPA from the root's perspective.

If a GVA is mapped by the Guest MMU, yet the GPA is not available for write to root context, then $GuestCtl0_{GExcCode}$ must indicate this.

The GPA presented to the root TLB is the result of translation through the guest TLB if it is in a mapped region of memory. The value stored in *Root.BadVAddr* and $Root.Context_{BadVPN2}$ is the Guest Physical Address being accessed by the guest.

This process ensures that after an exception, both *Root.BadVAddr* and $Root.Context_{BadVPN2}$ refer to a virtual address which is immediately usable by a root-mode handler, irrespective of whether the exception was triggered by root-mode or guest-mode execution.

10.6.4 Exception Priority

Table 10.6 lists all possible exceptions, and the relative priority of each, highest to lowest. The table also lists new exception conditions introduced by the Virtualization Module, and defines whether a switch to root mode is required before handling each exception.

Table 10.6 Priority of Exceptions

Exception	Description	Type	Taken in mode
Reset	The Cold Reset signal was asserted to the processor	Asynchronous Reset	Root
Soft Reset	The Reset signal was asserted to the processor		
Debug Single Step	A Debug Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.	Synchronous Debug	Root
Debug Interrupt	A debug interrupt (DbgBrk or DINT) was asserted.	Asynchronous Debug	Root
Imprecise Debug Data Break	An imprecise debug data break condition was asserted.		
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor.	Asynchronous	Root
Machine Check	Root, or Root TLB related. This can only occur as part of a guest (second step) address translation, root address translation, and root TLB operation (write, probe) whether for guest or root TLB. It is recommended that the Machine-Check be synchronous. A TLB instruction must cause a synchronous Machine Check.	Asynchronous or Synchronous	Root
	An internal inconsistency was detected by the processor.		Root
	Guest TLB related. This can only occur as part of a guest address translation (first step), and guest TLB operation (write, probe). It is recommended that the Machine-Check be synchronous. A TLB instruction must cause a synchronous Machine Check.		Guest
Interrupt	A root enabled interrupt occurred.	Asynchronous	Root
Deferred Watch	A Root watch exception, deferred because EXL was one when the exception was detected, was asserted after EXL went to zero. A deferred root watch exception may occur in guest mode in which case it is prioritized higher than a simultaneous occurring guest interrupt.	Asynchronous	Root
Interrupt	A guest enabled interrupt occurred.	Asynchronous	Guest
Deferred Watch	A Guest watch exception, deferred because Guest EXL was one when the exception was detected, was asserted after EXL went to zero.	Asynchronous	Guest
Debug Instruction Break	A debug instruction break condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses.	Synchronous Debug	Root

Table 10.6 Priority of Exceptions

Exception	Description	Type	Taken in mode
Watch - Instruction fetch	A root context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses. Refer to 'Watch Registers' - Section 10.8.	Synchronous	Root
	A guest-context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses. Refer to 'Watch Registers' - Section 10.8.		Guest
Address Error - Instruction fetch	A non-word-aligned address was loaded into PC.	Synchronous	Current
TLB Refill - Instruction fetch	A Guest TLB miss occurred on an instruction fetch	Synchronous	Guest
	A Root TLB miss occurred on an instruction fetch. This can occur due to a Root or Guest translation.		Root
TLB Invalid - Instruction fetch	The valid bit was zero in the guest context TLB entry mapping the address referenced by an instruction fetch.	Synchronous	Guest
	The valid bit was zero in the Root TLB entry mapping the address referenced by an instruction fetch. This can occur due to a Root or Guest translation.		Root
TLB Execute-inhibit	An instruction fetch matched a valid Guest TLB entry which had the XI bit set.	Synchronous	Guest
	An instruction fetch matched a valid Root TLB entry which had the XI bit set. This can occur due to a Root or Guest translation.		Root
Cache Error - Instruction fetch	A cache error occurred on an instruction fetch.	Synchronous or Asynchronous	Root
Bus Error - Instruction fetch	A bus error occurred on an instruction fetch.		
SDBBP	A debug SDBBP instruction was executed.	Synchronous Debug	Root
Guest Reserved Instruction Redirect	A guest-mode instruction triggers a Reserved Instruction Exception. When $GuestCtl0_{RI}$=1, this root-mode exception is raised before the guest-mode exception can be taken. Reserved Instruction Exception processing otherwise follow standard rules of prioritization within a given context - Reserved Instruction Redirect is taken as a side-effect of this processing.	Synchronous Hypervisor	Root

Table 10.6 Priority of Exceptions

Exception	Description	Type	Taken in mode
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources, or was illegal: Coprocessor Unusable, Reserved Instruction, MSA disabled. If exceptions occur on the same instruction, the Coprocessor Unusable, MSA disabled Exception take priority over the Reserved Instruction Exception.	Synchronous	Current
	Coprocessor unusable - guest. Access to a coprocessor was permitted by the $Guest.Status_{CU1-2}$ bits, but denied by $Root.Status_{CU1-2}$ bits. MSA disabled - guest. Access to the MSA unit was permitted by Guest.$Config5_{MSAEn}$, but denied by Root.$Config5_{MSAEn}$.		Root
Machine Check	Root TLB related. This can only occur as part of a Guest or Root address translation, or a TLBP/TLBWI/TLBGP/TLBGWI executed in root-mode.	Synchronous	Root
	Guest TLB related. This can only occur as part of a Guest address translation, or a TLBP/TLBWI executed in guest-mode		Guest
	An internal inconsistency was detected by the processor.		Root
Guest Privileged Sensitive Instruction Exception	An instruction executing in guest-mode could not be completed because it was denied access to the required resources by the *Root.GuestCtl0* register.	Synchronous Hypervisor	Root
Hypercall	A HYPCALL hypercall instruction was executed.	Synchronous Hypervisor	Root
Guest Software Field-Change	During guest execution, a software initiated change to certain CP0 register fields occurred.	Synchronous Hypervisor	Root
Guest Hardware Field-Change	During guest execution, a hardware initiated set of $Status_{EXL/TS}$ *occurred*.	Synchronous Hypervisor	Root
Execution Exception	An instruction-based exception occurred: Integer overflow, trap, system call, breakpoint, floating point, coprocessor 2 exception.	Synchronous	Current
Precise Debug Data Break	A precise debug data break on load/store (address match only) or a data break on store (address+data match) condition was asserted. Prioritized above data fetch exceptions to allow break on illegal data addresses.	Synchronous Debug	Root
Watch - Data access	A root context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses. Refer to 'Watch Registers' - Section 10.8.	Synchronous	Root
	A guest context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses. Refer to 'Watch Registers' - Section 10.8.		Guest

Table 10.6 Priority of Exceptions

Exception	Description	Type	Taken in mode
Address error - Data access	An unaligned address, or an address that was inaccessible in the current processor mode was referenced, by a load or store instruction.	Synchronous	Current
TLB Refill - Data access	A guest TLB miss occurred on a data access.	Synchronous	Guest
	A root TLB miss occurred on a data access. This can occur due to a Root or Guest translation.		Root
TLB Invalid - Data access	On a data access, a matching guest TLB entry was found, but the valid (V) bit was zero.	Synchronous	Guest
	On a data access, a matching root TLB entry was found, but the valid (V) bit was zero. This can occur due to a Root or Guest translation.		Root
TLB Read-Inhibit	On a data read access, a matching guest TLB entry was found, and the RI bit was set.	Synchronous	Guest
	On a data read access, a matching root TLB entry was found, and the RI bit was set. This can occur due to a Root or Guest translation.		Root
TLB Modified - Data access	The dirty bit was zero in the guest TLB entry mapping the address referenced by a store instruction	Synchronous	Guest
	The dirty bit was zero in the root TLB entry mapping the address referenced by a store instruction. This can occur due to a Root or Guest translation.		Root
Cache Error - Data access	A cache error occurred on a load or store data reference	Synchronous or Asynchronous	Root
Bus Error - Data access	A bus error occurred on a load or store data reference		
Precise Debug Data Break	A precise debug data break on load (address+data match only) condition was asserted. Prioritized last because all aspects of the data fetch must complete in order to do data match.	Synchronous Debug	Root

The “Type” column of Table 10.6 describes the type of exception. Table 10.7 explains the characteristics of each exception type.

Table 10.7 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous Reset	Denotes a reset-type exception that occurs asynchronously to instruction execution. These exceptions always have the highest priority to guarantee that the processor can always be placed in a running state. These exceptions always require a switch to root mode.
Asynchronous Debug	Denotes a debug exception that occurs asynchronously to instruction execution. These exceptions have very high priority with respect to other exceptions because of the desire to enter Debug Mode, even in the presence of other exceptions, both asynchronous and synchronous. These exceptions always require a switch to root mode.

Table 10.7 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous	Denotes any other type of exception that occurs asynchronously to instruction execution. These exceptions are shown with higher priority than synchronous exceptions mainly for notational convenience. If one thinks of asynchronous exceptions as occurring between instructions, they are either the lowest priority relative to the previous instruction, or the highest priority relative to the next instruction. The ordering of the table above considers them in the second way. These exceptions always require a switch to root mode.
Synchronous Debug	Denotes an debug debug exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions are prioritized above other synchronous exceptions to allow entry to Debug Mode, even in the presence of other exceptions. These exceptions always require a switch to root mode.
Synchronous Hypervisor	Denotes an exception that occurs as a result of guest-mode instruction execution which requires hypervisor intervention. It is reported precisely with respect to the instruction that caused the exception. These exceptions always require a switch to root mode.
Synchronous	Denotes any other exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions tend to be prioritized below other types of exceptions, but there is a relative priority of synchronous exceptions with each other. In some cases, these exceptions can be handled without switching modes.

10.6.5 Exception Vector Locations

Exception vector locations are as defined in the base architecture.

The vector location is determined from the values of *EBase*, $Status_{EXL}$, $Status_{BEV}$, $IntCtl_{VS}$ and $Config3_{VEIC}$ obtained from the context in which the exception is handled.

The General Exception entry point is used for new hypervisor exceptions Guest Privileged Sensitive Instruction, Guest Reserved Instruction Redirect, Guest Software Field Change, Guest Hardware Field Change and Hypercall.

10.6.6 Synchronous and Synchronous Hypervisor Exceptions

During guest mode execution, control can be returned to root mode at any time. When an exception condition is detected during guest mode execution and the condition requires a switch to root mode, the switch is made before any exception state is saved. As a result, exception state in the guest CP0 context is not affected.

The switch to root mode is achieved by setting $Root.Status_{EXL}$=1 or $Root.Status_{ERL}$=1 (as appropriate) before any other state is saved. This ensures that all exception state is stored into root CP0 context, regardless of whether the processor was executing in root or guest mode at the point where the exception was detected.

Refer to the Exceptions chapter for more information on these exceptions.

10.6.7 Guest Exception Code in Root Context

In the case of a guest exception which causes a guest exit to root, hardware must supply the appropriate value for $Root.Cause_{ExcCode}$ and $GuestCtl0_{GExcCode}$, as described in the following pseudo-code.

if guest exception is (GPSI or GSFC or GHFC or HC or GRR or IMP) then
    $Root.Cause_{ExcCode}$ ← “GE”|

```
        Root.GuestCtl0GExcCode ← “GPSI” or “GSFC” or “GHFC” or “HC” or “GRR” or “IMP”
elseif  guest exception is (Root TLB-Refill or TLB-Invalid)
            Root.CauseExcCode ← “TLBS” or “TLBL”
            # loading of GPA for both TLB-Refill and TLB-Invalid is recommended.
            Root.GuestCtl0GExcCode ← “GPA”
elseif guest exception is (Root TLB-Execute_Inhibit or TLB-Read_Inhibit)
        if (Root.PageGrainIEC = 0) then
            Root.CauseExcCode ← “TLBL”
            Root.GuestCtl0GExcCode ← “GPA” or GVA”
        elseif  (TLB Execute-Inhibit)
            Root.CauseExcCode ← “TLBXI”
            Root.GuestCtl0GExcCode ← “GVA” or “GPA”
        else
            Root.CauseExcCode ← “TLBRI”
            Root.GuestCtl0GExcCode ← “GVA” or “GPA”
        endif
elseif  guest exception is (TLB Modified)
            Root.CauseExcCode ← “MOD”
            Root.GuestCtl0GExcCode ← “GVA” or “GPA”
else
        Root.CauseExcCode ← baseline “ExcCode”
        Root.GuestCtl0GExcCode ← “UNDEFINED”
endif
```

10.7 Interrupts

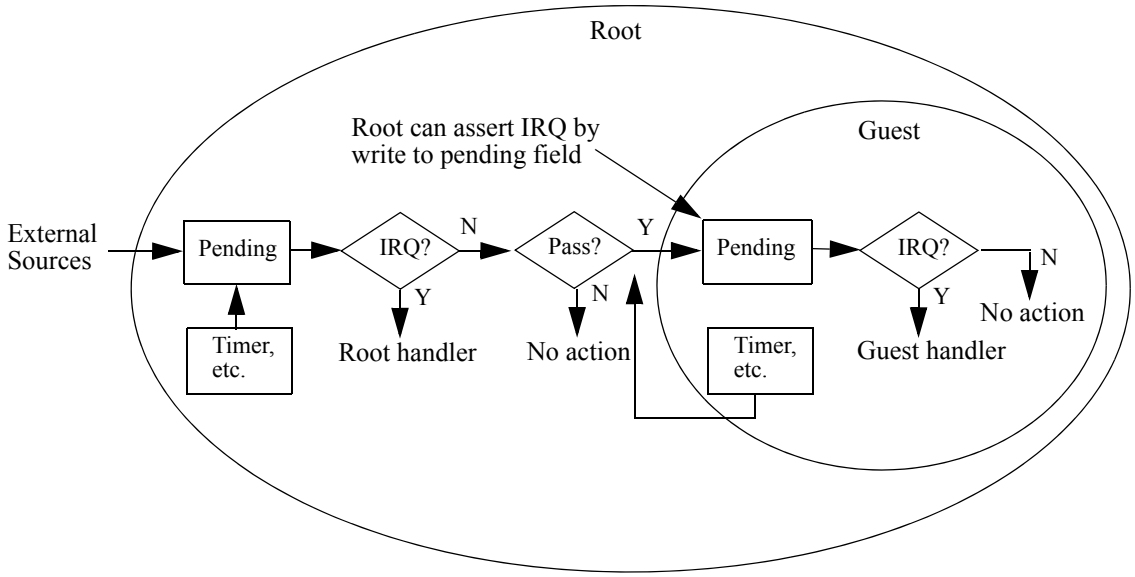
The Virtualization Module provides a virtualized interrupt system for the guest.

The root context interrupt system is always active, even during guest mode execution. An interrupt source enabled in the root context always results in a root-mode interrupt. Guests cannot disable root mode interrupts.

Standard interrupt rules are used by both root and guest contexts to determine when an interrupt should be taken. An interrupt enabled in the root context is taken in root mode. An interrupt masked by root and enabled in the guest context is taken in guest mode. Root interrupts take priority over guest interrupts.

Figure 10.5 shows the how virtualized interrupts are managed in the I6500 core.

Figure 10.5 Interrupt Handling in the Virtualization Module I



The *Guest.Cause*$_{RIPL/IP}$ field is the source of guest interrupts. The behavior of this field is controlled from the root context. Two methods can be used to trigger guest interrupts - a root-mode write to the *Guest.Cause* register, or direct assignment of real interrupt signal to the guest interrupt system. Interrupt sources are combined such that both methods can be used.

Timers and related interrupts are available in both guest and root contexts.

The set of pending interrupts seen by the guest context is the combination (logical OR) of:

- External interrupts passed through from the root context, enabled by *GuestCtl0*$_{PIP}$ if implemented.
- Interrupts generated within the guest context (e.g., Timer interrupts, Software interrupts)
- Root asserted interrupts, set by software write to *GuestCtl2*$_{VIP}$ field in non-EIC mode, or hardware capture of a guest interrupt in *GuestCtl2*$_{GRIPL}$ in EIC mode.

Software should enable direct interrupt assignment only when root and guest agree on the interpretation of interrupt pending/enable fields in the *Status* and *Cause* registers. Direct assignment is appropriate if both Root and Guest use EIC mode, or if both use non-EIC mode. Root can track changes to the guest interrupt system status using the field-change exceptions which result from guest initiated changes to fields *Status*$_{BEV}$, *Cause*$_{IV}$ or *IntCtl*$_{VS}$.

Root must assign interrupts to Guest with caution. For example, in non-EIC mode, if an interrupt pin (HW[5:0]) is shared by multiple interrupt sources, then enabling direct guest visibility (in Guest *Cause*$_{IP[n]}$ via *GuestCtl0*$_{PIP[n]}$=1) causes all the interrupt sources on that pin to be visible to the Guest, possibly removing Root intervention capability. If Root Software needs to guarantee Root intervention capability on an interrupt then that interrupt should not be directly visible to Guest.

In non-EIC mode, the guest timer interrupt is always applied to the interrupt source indicated by the *Guest.IntCtl*$_{IPTI}$ field and is not affected by the *GuestCtl0*$_{PIP}$ field. Similarly, Guest software interrupts are not affected by the *GuestCtl0*$_{PIP}$ field, and are always applied to the interrupt source indicated by *Guest.IntCtl*$_{IPPCI}$

A virtualization-based external interrupt delivery system, whether EIC or non-EIC provides the following capabilities:

1. Root assignment of External Interrupt.

   Hardware delivers interrupt to root context, with root-mode servicing of external interrupt.

2. Guest assignment of External Interrupt with Root Intervention.

   Hardware delivers interrupt to root context, with root-mode hand-off to guest by writing to $GuestCtl2_{vIP}$, followed by guest servicing of external interrupt.

   If root requires visibility into guest interrupts, then root should use this method to deliver interrupts to guest.

3. Guest assignment of External Interrupt without Root Intervention.

   Hardware delivers interrupt to guest context without root intervention, followed by guest servicing of external interrupt. The interrupt is not visible to root as root has made the choice to assign to guest.

A MIPS enabled virtualized external interrupt delivery system also provides support for Virtual Interrupts. Root can simulate a guest interrupt by writing 1 to $GuestCtl2_{vIP}$. It can subsequently clear the interrupt by writing 0 to $GuestCtl2_{vIP}$.

Virtual Interrupt capability can be used to support guest virtual drivers. The Root injects an interrupt into the Guest context. The Guest fields the interrupt, and in so doing causes a trap to Root, either by device activity or protected memory access. Root may then clear the interrupt by writing to guest $Cause_{IP}$ set earlier.

10.7.1 External Interrupts

10.7.1.1 Non-EIC Interrupt Handling

This section provides a detailed description of non-EIC handling in a recommended implementation. The term HW is used to represent an external interrupt source. HW is alternatively referred to as IRQ in other sections of the Module. HW is a set of interrupt pins common to both root and guest context.

Whether an external interrupt is visible to guest context or root context is dependent on $GuestCtl0_{PIP}$ (Pending Interrupt Pass-through). If $GuestCtl0_{PIP[n]}$ =1, then HW[n] is visible to guest context through $Guest.Cause_{IP[n+2]}$, otherwise it is visible to root context through $Root.Cause_{IP[n+2]}$.

If $GuestCtl0_{PIP[n]}$=0, but Root needs to transfer the external interrupt to Guest, then it must write to a software visible register, $GuestCtl2_{vIP[n]}$ (Interrupt Pending, Virtual). This method is also used by Root to inject a virtual interrupt into guest context. It is also a convenient way for Root to save and restore interrupt state of a Guest, if an interrupt had been injected by Root, but needs to be preserved across context switches. In the absence of $GuestCtl2_{vIP}$, Root would need to derive the equivalent of vIP by reading $Guest.Cause_{IP}$ which may be problematic since other interrupts could also be present.

$GuestCtl2_{vIP}$, $Guest.Cause_{IP}$ and $Root.Cause_{IP}$ handling is described in relation to $GuestCtl2_{vIP}$ and $GuestCtl0_{PIP}$. The application of $GuestCtl2_{HC}$ is discussed below.

$GuestCtl2_{vIP}$ Handling:

```
if (MTC0[GuestCtl2vIP[n]]=1)
        GuestCtl2vIP[n] ← 1
else if ((Deassertion of HW[n] and GuestCtl2HC[n]) or (MTC0[GuestCtl2vIP[n]]=0))
        GuestCtl2vIP[n] ← 0
endif
```

Guest.Cause$_{IP}$ Handling:

$$Guest.Cause_{IP[n+2]} = ((HW[n] \text{ and } GuestCtl0_{PIP[n]}) \text{ or } GuestCtl2_{vIP[n]})$$

Root.Cause$_{IP}$ Handling:

$$Root.Cause_{IP[n+2]} = (HW[n] \text{ and } !(GuestCtl0_{PIP[n]} \text{ or } (GuestCtl2_{vIP[n]} \text{ and } GuestCtl2_{HC[n]})))$$

GuestCtl2$_{HC}$ is provided to control how *GuestCtl2$_{vIP}$* is reset. If a bit of *GuestCtl2$_{HC}$* is 1, then the deassertion of related external interrupts always causes the associated *GuestCtl2$_{vIP}$* to be cleared. If a bit of *GuestCtl2$_{HC}$* is 0 then the deassertion of HW[n] does not cause *GuestCtl2$_{vIP}$* to be cleared. In this case, it is the responsibility of root software to clear by writing 0 to *GuestCtl2$_{vIP\ [n]}$* .

In summary, interrupt injection in guest context serves two purposes - root assignment of external interrupts and injection of virtual interrupts to Guest. *GuestCtl2$_{HC}$* provides the means to root software to distinguish between the two. Root software can use this facility to transfer an external interrupt HW[n] for guest servicing. In this scenario, *GuestCtl2$_{HC[n]}$*=1 and the assertion of *GuestCtl2$_{vIP[n]}$* causes corresponding *Root.Cause$_{IP[n+2]}$* to be cleared, thus transparently affecting the transfer. Otherwise, Root would have to disable interrupts for that specific source by clearing *Root.Status$_{IM[n]}$*. On the other hand, Root can use this capability to inject interrupts into Guest context for guest virtual device drivers, as an e.g. In this case, *GuestCtl2$_{HC[n]}$*=0, the assumption is that there is no external interrupt tied to the injected interrupt, and thus assertion of *GuestCtl2$_{vIP\ [n]}$* should not cause *Root.Cause$_{IP[n+2]}$* to be cleared. *Guest.Cause$_{IP[n+2]}$* is asserted in both cases described.

Virtual interrupt handling is an option that can be detected by the presence of *GuestCtl2*. Hardware clear capability is also an option, even if virtual interrupts are supported. This capability exists if the field is writeable or preset to 1.

10.7.1.2 EIC Interrupt Handling

In EIC mode, the external interrupt controller (EIC) is responsible for combining internal and external sources into a single interrupt-priority level, which appears in the *Cause$_{RIPL}$* field.

When an implementation makes EIC mode available (as indicated by *Guest.Config3$_{VEIC}$*=1), two interrupt priority-level signals must be generated within the EIC - one for the root context (affecting *Root.Cause$_{RIPL}$*), and one for the guest context (affecting *Guest.Cause$_{RIPL}$*). The root and guest timer interrupt signals are combined in an implementation-dependent way with external inputs to produce the root and guest interrupt priority levels.

In addition to RIPL, the interrupt Vector (offset or number), and EICSS is also sent on each of the Root and Guest interrupt buses. The Vector from the EIC is either utilized by hardware as is, or derived from the EIC input. A GuestID accompanies only the root bus, providing GuestID is supported in the implementation. This is because the EIC can also send an interrupt for guest on the root interrupt bus. Thus the GuestID for the root interrupt bus may be non-zero. The GuestID for a guest interrupt taken in root mode must be registered in *GuestCtl1$_{EID}$*. The guest associated with the guest bus is by default equal to *GuestCtl1$_{ID}$*.

In the architecture as defined, the type of vector a virtualized core can accept from the EIC is fixed - it is either a vector number or offset but never both. This is because currently there is no capability to distinguish between the two types, intentionally so. It is recommended that a typical virtualized EIC source a vector number to the core.

The EIC should assign interrupts to root and guest interrupt buses as per the following rules:

- Root interrupts must always be taken in root context and thus be presented on root interrupt bus by the EIC.
- If a guest interrupt requires root intervention, then it must be presented on the root interrupt bus by the EIC. And interrupt for a non-resident guest must always be sent on the root interrupt bus. An interrupt for the resident guest may also be sent on the root interrupt bus.

  A guest interrupt while the processor is in root mode can cause an interrupt immediately unless masked by $Root.Status_{IPL}$. Hardware should not stall the interrupt until the processor enters guest mode.
- Only an interrupt for a resident guest can be sent on the guest interrupt bus. If software programs the EIC to send an interrupt for a non-resident guest on the guest interrupt bus, then an implementation of the core is not required to respond to this interrupt.

To allow the EIC to distinguish between resident and non-resident guests, the core must send $GuestCtl1_{ID}$ to the EIC. An implementation must account for the delay between when the $GuestCtl1_{ID}$ changes and when it is visible to the EIC to avoid a spurious interrupt for a non-resident guest from being sent on the guest interrupt bus.

The processor and EIC are required to implement a protocol to avoid the above mentioned race. On a guest context switch, root software must first write 0 to $GuestCtl1_{ID}$. This is equivalent to a STOP command for the EIC. The EIC recognizes this as a stall and does not send interrupts to guest context by setting the requested interrupt priority level to 0 on the guest interrupt bus to the core. Root software can then save and restore guest context, followed by a write of new GuestID to $GuestCtl1_{ID}$. Once the write is complete, root software can enable guest mode operation. If an EIC implementation and root software follow this recommendation, then this prevents loss of an interrupt posted to the guest interrupt bus while root is switching guest context. An interrupt for the formerly active guest is posted on the Root interrupt bus.

An EIC mode interrupt is generated in either guest or root context whenever hardware detects a change in RIPL on the respective interrupt buses from the EIC. It is possible for an EIC implementation to have active interrupts on both bus. In this case the root interrupt is always higher priority then the guest interrupt.

For the case of an interrupt in root context, two different interrupt vectors are used, one for root, the other for guest. Hardware is able to distinguish between the two by checking the GuestID on the root interrupt bus. The following pseudo-code describes how hardware generates the interrupt vector, depending on whether the EIC provides a vector offset (vectorOffset) or vector number (vectorNumber).

```
EIC_mode ← Config3.VEIC=1 && IntCtl.VS!=0 && Cause.IV=1 && Status.BEV=0

if EIC_mode

        if (EIC provides vectorNumber)
                if (GuestID=0)
                        vectorOffset ← 0x200 + (EIC_vectorNumber x (IntCtl.VS || 0b00000))
                else //GuestID is non-zero
                        vectorOffset ←0x200
                endif
        else // EIC provides vectorOffset
                if (GuestID=0) // EIC provides an offset relative to 0x200
                        vectorOffset ←EIC_vectorOffset
                else //GuestID is non-zero
                        vectorOffset ←0x200
                endif
        endif
endif
```

If the interrupt is for guest, then the handler must compare GuestCtl1EID to GuestCtl1ID. If they are not equal, then interrupt is for non-resident guest, and interrupt servicing may either continue in root or guest context. If interrupt servicing is to continue in guest context, then the handler must first save the resident guest state (CP0, GPRs etc) following by a restore of the new guest's context. The root ERET instruction causes a transfer to guest mode (when GuestCtl0GM=1), followed by a guest interrupt providing GuestCtl2GRIPL is non-zero.

If GuestCtl1EID and GuestCtl1ID are equal, then save and restore is not needed. Interrupt servicing may either continue in root or guest context. If the interrupt is to be serviced in guest context, then the root ERET instruction causes a change to guest mode (when GuestCtl0GM=1), following by a guest interrupt providing GuestCtl2GRIPL is non-zero.

As described above, for any change in $GuestCtl1_{ID}$, root software must first insert a STOP command on interface to EIC by writing 0 to $GuestCtl1_{ID}$. Once quiescent, root software may execute whatever software sequence it needs to. This is followed by a write of new GuestID to $GuestCtl1_{ID}$, then the root ERET instruction. There may be some arbitrary delay between write of GuestID and ERET instruction where EIC can respond with an interrupt on guest bus, but hardware does not trigger an interrupt because processor is in root mode.

A root interrupt must use $Root.SRSCtl_{EICSS}$. Otherwise, hardware forces use of $Root.SRSCtl_{ESS}$ if the interrupt on the root interrupt bus is for any guest.

The guest interrupt in the scenario where the interrupt is transferred from root context after having been received on the root interrupt bus is caused when the processor enters guest mode and hardware detects that $GuestCtl2_{GRIPL}$ *is non-zero.*

Once in guest mode, the guest interrupt handler completes with an ERET instruction. The guest continues execution from its *EPC*, and not transfer back to root mode even if there was a change in guest context. If a return to root mode is required, then the HYPERCALL instruction must be used.

The root CP0 register, *GuestCtl2,* where the root interrupt bus Vector, EICSS and RIPL storage in root CP0 state is required because in a typical EIC-based implementation, an acknowledgement is returned to the EIC when the interrupt is triggered. If an interrupt for the guest is initially triggered in root context, then the use of these fields does not occur until the root ERET instruction is executed to effect a change to guest mode. In the meanwhile, another root interrupt can occur which can overwrite the fields on the bus. Saving the fields as root CP0 register allows for nesting of these fields, and thus supports nesting of interrupts.

Hardware optimizes the transfer of $GuestCtl2_{GRIPL}$ *and* $GuestCtl2_{EICSS}$ *into guest CP0 context on guest entry. Hardware writes* $GuestCtl2_{GRIPL}$ to $Guest.Cause_{RIPL}$, and $GuestCtl2_{EICSS}$ to $Guest.SRSCtl_{EICSS}$ providing $GuestCtl2_{GRIPL}$ *is non-zero. Root software thus has the option of preventing hardware transfer by clearing* $GuestCtl2_{GRIPL}$ *before guest entry.*

In the case where root injects an interrupt into guest context after the interrupt was received on the root interrupt bus, hardware must ensure that two acknowledgements are not returned to the EIC as this may cause a loss of an interrupt. In the case where an interrupt is received on the root interrupt bus, hardware must always send an acknowledgement on the root interrupt bus. But in the case where the interrupt was injected into guest context by root, hardware should not send an acknowledgement on the guest interrupt bus as the interrupt was not received on this bus. Hardware can determine this because $GuestCtl2_{GRIPL}$ would be a non-zero value for the case of root injection.

Access to COP1 FPR and COP2 may be protected setting $Root.Status_{CU[2:1]}$ appropriately. If access is disabled in root context, then it is also disabled in guest and causes the appropriate exception (Coprocessor Unusable in root context). Hi/Lo registers are not protected by any means, and must be saved/restored if necessary.

10.7.2 Derivation of Guest.Cause$_{IP/RIPL}$

The interrupt pending value seen by the guest is calculated as shown in the following pseudocode example. The result value can be read by the guest (and the root) from the *Guest.Cause$_{RIPL\ /\ IP}$* field and is the value used to determine whether a guest interrupt is taken. Note that the value returned from *Guest.Cause$_{RIPL\ /\ IP}$* on a read is generated from the value originally written by the root and from the status of directly assigned external interrupts. Hence the value written by the root may not be equal to the value read back.

```
# Returns:
# Non-EIC     IP7..0.
# EIC -       (RIPL << 2) + IP1..0

subroutine GuestInterruptPending() :

if ((Guest.Config3VEIC = 1) and
    (Guest.IntCtlVS != 0) and
    (Guest.CauseIV = 1) and
    (Guest.StatusBEV = 0)) then
    # Guest in EIC mode
    # - GuestCtl0PIP does not apply in EIC mode.
    # - EIC must include guest interrupt sources in the EICGuestLevel signal
    #   - This includes Guest's TI, IP1, IP0 and PCI if implemented.
        - FDCI is only visible in root context.
    # - GuestCtl2 required in EIC mode.
    if (EICGuestLevel > GuestCtl2GRIPL)
        irq ← EICGuestLevel
    else
        irq ← GuestCtl2GRIPL
        # h/w must clear if GuestCtl2GRIPL is source of interrupt.
        GuestCtl2GRIPL ← 0
    endif
    # Guest.CauseIP[1:0] is incorporated in EIC.
    # State of Guest.CauseIP[1:0] is however preserved.
    r  ←(irq << 2) OR Guest.CauseIP[1:0]

else
    # Guest in non-EIC mode
    # - External interrupts factored in if guest passthrough enabled.
    # - Internal interrupts applied here, if implemented
    # - Includes support for guest interrupt injection by root.
    irq[7:2]  ← HW[5:0]
    if (GuestCtl0PT=0)
        # All interrupts processed first by root.
        if (GuestCtl0G2=1)
            # root software injects interrupts.
            r  ← GuestCtl2vIP[5:0]
        else
            # if GuestCtl2vIP is not supported, then root writes Guest.Cause.IP
            # to inject interrupt in guest context. H/W captures the write in a
            # shadow register called Root_HW_VIP.
            r  ← Root_HW_VIP[5:0]
        endif
    else
        # Guest interrupt passthrough supported.
        if (GuestCtl0G2=1)
            r  ← Root.GuestCtl2vIP[5:0] OR (irq[7:2] AND Root.GuestCtl0PIP[5:0])
```

```
        else
            r   ← Root_HW_VIP[5:0] OR (irq[7:2] AND Root.GuestCtl0PIP[5:0])
        endif
    endif
    r           ← r << 2
    r           ← r OR (GuestTimerInterrupt << Guest.IntCtlIPTI)
    r           ← r OR (PCIEvent << Guest.IntCtlIPPCI)
    r           ← r OR Guest.CauseIP[1:0]

endif

return(r)
endsub
```

The value returned by GuestInterruptPending() is subsequently qualified by Guest $Status_{IM}$ in non-EIC mode or Guest $Status_{IPL}$ in EIC mode, as per the base architecture.

Fields in Guest *Config* registers indicate which interrupt options are available to the guest.

10.7.3 Timer Interrupts

Root may inject a timer interrupt in guest context by setting Guest $Cause_{TI}$ and indirectly Guest $Cause_{IP[IPTI]}$. This may happen under the scenario where a guest has been switched out, but its virtual timer, maintained by root, is triggered. Root would set Guest $Cause_{TI}$ before entering guest mode for the guest. Guest would take a timer interrupt, clear Guest *Compare*, which would then clear Guest $Cause_{TI}$. As per baseline MIPS architecture, a write to *Compare* clears $Cause_{TI}$.

Root maintaining a virtual timer for a guest is recommended if there are multiple guests in operation. Otherwise, if there is only one guest, but the processor is in root mode, then a match on Guest *Count* and Guest *Compare* is allowed in an implementation to set Guest $Cause_{TI}$ and Guest $Cause_{IP[IPTI]}$. Once Root transitions to guest mode, then guest timer interrupt can be signaled in guest mode.

Root Injection of Guest TI:

```
if (MTGC0[Guest.CauseTI]=1)
        Root.Guest.CauseTI ← 1
else if ((MTC0[Guest.Compare]))
        Root.Guest.CauseTI ← 0
endif
```

where Root.Guest.$Cause_{TI}$ is a hardware shadow copy of Guest.$Cause_{TI}$ that is set when Guest.$Cause_{TI}$ is written by Root.

$Guest.Cause_{IP[IPTI]}$ = `Root.Guest.CauseTI or "Other External and Internal interrupts".`

where “Other External and Internal interrupts” is defined in Section 10.7.2.

10.7.4 Performance Counter Interrupts

Root can configure the definition of performance counters in the Guest context via Guest $Config1_{PC}$ as follows:

- Guest $Config1_{PC}$=0, then performance counters are unimplemented in the guest context, access is **UNPREDICTABLE**.
- Guest $Config1_{PC}$=1, the performance counters are virtually shared by root and guest contexts.

The *PerfCnt* register(s) are never implemented in the Guest context. A Guest may have direct access to virtual performance counter registers under root software management when $Config1_{PC}$=1. If virtually shared, the encodings of $PerfCnt_{EC}$ as 0 or 1 cause a GPSI Exception to be raised on Guest access to a performance counter register. Root software may choose to configure performance counters for legal Guest access by encoding $PerfCnt_{EC}$ as 2 or 3.

Software may choose to assign all performance counters to Guest or Root, but not both. This is the recommended policy for sharing between Root and Guest. Root typically configures the Guest access when it initializes guest context. If assigned to Guest then Guest access does not cause a GPSI Exception.

Alternatively, an implementation may optionally choose to assign a subset of the total *PerfCnt* registers in Root CP0 context to Guest. Read of guest $PerfCnt(N)_M$ should return root $PerfCnt(N+1)_{EC[1]}$ to indicate *PerfCnt(N+1)* is owned by guest. If all *PerfCnt* pairs are allocated to guest, then guest read of the last M bit must return 0. Guest *PerfCnt* pairs assigned to Guest in this manner must be a contiguous range, starting from the least significant pair. It is further assumed that the allotment of performance counters to a guest is not dynamic - once established after initial guest access (which caused GPSI), then the allotment must remain as such for duration of guest.

Once assigned to Guest or Root (default) context, that context independently manages the performance counters, including interrupts. E.g., if the performance counters are enabled for Root, then Root $Cause_{PCI}$ and Root $Cause_{IP[IPPCI]}$ are set by hardware on counter overflow. Otherwise, counter overflow sets Guest.$Cause_{PCI}$ and Guest.$Cause_{IP[IPPCI]}$.

If Root software needs to inject a performance counter interrupt into Guest context, it must do so by setting the most-significant bit of the *PerfCnt* counter. Similarly Root may clear a guest performance counter interrupt by clearing the most-significant bit of the counter. Thus, Root does not require the ability to read/write $Guest.Cause_{PCI}$.

The $PerfCnt_{EC}$ field is Root only virtualization control and is not visible to the Guest.

PerfCnt use of *Status* register *K, S, U,* and *EXL* fields is taken from the current Root or Guest context.

PerfCnt interrupt behavior is solely governed by $PerfCnt_{IE}$, enabled context *Status* register interrupt masks and enable.

10.8 Watchpoint Debug Support

Root and Guest Watchpoint debug support is provided by Coprocessor 0 *WatchHi* and *WatchLo* register pair(s). These registers are present in Root if Root $Config1_{WR}$ = 1 and in Guest if Guest $Config1_{WR}$ = 1.

A virtualized implementation may choose to provide no Watch register support, Root-only Watch register support, or Root and Guest Watch register support. Virtualized handling applies to both *WatchHi* and *WatchLo* registers but is generically referred to as “Watch” registers.

In Table 10.8, the state of Guest *Config1*$_{WR}$. conveys what support is available to Guest.

Table 10.8 Guest Watchpoint Support

Guest *Config1*$_{WR}$ Value	R/W State	Function
0	R	No Guest Watch registers.
1	R	Guest Watch registers present.
0/1	R (Guest) R/W (Root)	Virtual Guest Watch support provided.

Root-only Watch registers (Root *Config1*$_{WR}$ = 1 and Guest *Config1*$_{WR}$ = 0) allows for Root Watch of Root Virtual Addresses (RVA). If both Root and Guest Watch registers are present (Guest *Config1*$_{WR}$ = 1), then Root and Guest Watch operates independently.

The Virtualization Debug definition also allows for virtual Guest Watch via Root Watch registers (Guest *Config1*$_{WR}$ = 0/1). This feature is optional. Root Software can test R/W state of Guest *Config1*$_{WR}$ to determine whether virtual Guest Watch registers are supported.

Table 10.9 Watch Control

Guest *Config1*$_{WR}$ Value (in R/W State)	Root *WatchHi*$_{WM[1:0]}$	Function	Guest Exception on Access	Guest Exception on Match	Root Exception
0	X0	Root Watch RVA	UNPREDICTABLE	None	Watch
1	00	Root Watch RVA	GPSI	None	Watch
1	10	Guest Watch GVA	None	Watch	None
1	11	Reserved	-	-	-

There is no support for Root emulation of Guest watch registers. Root emulation of Guest watch registers would require that every guest read and write trap to Root. In sharing mode, once a watch register pair is assigned to Guest, Guest can setup registers without Root intervention.

Referring to Table 10.9, if Guest *Config1*$_{WR}$ = 0, then no watch register pairs are enabled for Guest watch. A Guest access is treated as UNPREDICTABLE. The I6500 will no-op an MTC0 and return 0s on MFC0. If Guest *Config1*$_{WR}$ = 1, then a Guest access is treated normally except a MTC0 cannot modify *WatchHi*$_{WM}$, and an MFC0 will return 0s for *WatchHi*$_{WM.}$

If Guest *Config1*$_{WR}$ = 1, then selected Root Watch register pairs are enabled for Root or Guest watch. Referring to Table 10.9, this is determined by Root *WatchHi*$_{WM}$[1]. Root *WatchHi*$_{WM}$[0] determines whether Root is watching RVA or GPA. Root Watch of GPA is optional. A write of 1 to Root *WatchHi*$_{WM}$[1:0], will write 0, defaulting to RVA watch.

If under Guest control, Guest can only watch GVA. A write of 3 to Root *WatchHi*$_{WM}$[1:0] writes 2 in this configuration, defaulting to GVA watch. Root can take away privilege from Guest at any time by writing to Root Watch registers. The Root access thus does not take an exception on access of a shared pair of registers under Guest control. If under Root control with Root *WatchHi*$_{WM}$[1]=0 then a Guest access results in a GPSI exception. Root may choose to assign this register pair to Guest at this point, or return to the guest instruction following the move.

Guest watch is enabled strictly in guest mode as defined by the equation:

($Root.GuestCtl0_{GM}$ = 1 and $Root.Status_{EXL}$ = 0 and $Root.Status_{ERL}$ = 0 and $Root.Debug_{DM}$ = 0)

There is no facility for Guest to watch addresses related to Root intervention events. That is, events occurring when the following equation is true:

($Root.GuestCtl0_{GM}$ = 1 and ($Root.Status_{EXL}$ = 1 or $Root.Status_{ERL}$ = 1 or $Root.Debug_{DM}$ = 1))

The I6500 supports virtual sharing between Root and Guest. As such, Root software may choose to assign all *WatchHi* and *WatchLo* to Guest or Root, but not both. This is the recommended policy for sharing between Root and Guest. If assigned to Guest then Guest access does not cause a GPSI exception.

10.9 Guest Mode and Debug Features

The Virtualization Module provides full access to debug facilities. When the processor is running in debug privileged execution mode, it has full access to all resources that are available in the Root context.

As per Table 10.1, The debug privileged execution mode exists in the root context. A processor supporting virtualization operates in two contexts, Root and Guest. Within Guest, there are three privileged execution modes; kernel, supervisor and user, and in Root context, there are four; kernel, supervisor, user and debug.

Table 10.10 lists debug features and their application to the Virtualization Module.

Table 10.10 Debug Features and Application to Virtualization Module

Feature	Description	Reference
Debug mode	Guest mode is mutually exclusive with Debug mode. When in Debug mode ($Debug_{DM}$=1), the processor is not in guest mode.	
	When the processor is running in Debug mode, it has full access to all resources that are available to Root-Kernel mode operation.	MIPS On-Chip Instrumentation Debug Technical Reference Manual
Debug Segment (dseg)	When the processor is running in Debug mode, the memory map is determined by the root context.	MIPS On-Chip Instrumentation Debug Technical Reference Manual
Access to guest CP0 context	Debug tools access general purpose registers (GPRs) and coprocessor registers by executing instructions in the processor pipeline. Access to the guest CP0 context must use the Virtualization Module instructions provided to transfer data between the root and guest contexts - MTGC0 and MFGC0. Accesses to the guest TLB must use the instructions provided to initiate guest TLB operations from the root context - TLBGP, TLBGR, TLBGWI, TLBGWR. These operations are used to transfer data between the guest TLB and the guest CP0 context. When accessing the guest TLB in debug mode, a two-step process is required - to transfer data to/from the guest CP0 context and guest TLB, and to transfer data to/from the root CP0 context and guest CP0 context.	Section 10.1.8

Table 10.10 Debug Features and Application to Virtualization Module

Feature	Description	Reference
Hardware Breakpoints	When implemented, hardware breakpoints are part of the root context. The root context remains active during guest mode execution, allowing hardware breakpoints to be used to debug guest software. Exceptions resulting from hardware breakpoints are of type Synchronous Debug or Asynchronous Debug. In both cases, the exceptions are handled in Debug mode.	Section 10.6.4
Watch registers	Support for use of watch-point from the Guest is optionally provided.	Refer to Section 10.8

Chapter 11

Data Scratch Pad RAM

The optional Data Scratch Pad RAM (DSPRAM) block provides a general scratch pad RAM used for temporary storage of data. The DSPRAM provides a connection to on-chip memory or memory-mapped registers, which are accessed in parallel with the L1 data cache to minimize access latency.

The default RAM size is 64 KB, but can be set to any power of 2 size (128 KB, 256 KB, etc.) The base address of the DSPRAM in memory is set using two new CP0 registers.

11.1 Overview

The DSPRAM module has the following features:

- 16 Byte wide data path for both read and write operations
- Data can be protected (parity/ECC/none on 32 bit granularity)
- One or multi-cycle latency for read/write in byte invariant format
- Multi-threaded design, so the blocking of one thread may not block other thread
- Root physical address (RPA) is checked against base and range to validate access. No other tag array.

11.1.1 New CP0 Registers

Two new Coprocessor 0 registers have been added to facilitate access to the DSPRAM as shown in Table 11.1 .

Table 11.1 CP0 Registers Used for Accessing the DSPRAM Module

Register Number	Sel	Register Name	Description
9	6	SAARI	Special Address Access Register Index. Provides an index into the SAAR register to indicate whether the DSPRAM or other module is being accessed. There is one SAARI register per VP.
9	7	SAAR	Special Address Access Register. Stores the base address where the DSP will be located, as well as the block size. There is one SAAR register per core.

The bit assignments for each of these register is shown below.

11.1.1.1 Special Address Access Register Index — SAARI (CP0 Register 9, Select 6)

The SAARI register is instantiated per-VP and provides an index value that determines whether the DSPRAM is accessed, or another block is accessed such as the Inter-Thread Communication Unit (ITC). There is one SAARI register per VP. This means that multiple SAARI registers use the same SAAR register to access the associated block.

Each SAARI register contains a 6-bit TARGET field that selects between the Inter-Thread Communication unit (ITC) and the DSPRAM. If the value is set to 0x01, the DSPRAM block is accessed. If the value is set to 0x00, the ITC block is accessed.

Refer to the section entitled Register Programming Sequence for more information on how this register is used.

Figure 11.1 SAARI Register Format

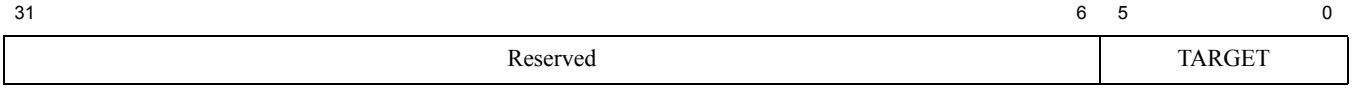


Table 11.2 Field Descriptions for SAARI Register

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	31:6	Reserved. This bits should be written as zero. Reads are undefined.	R	0
TARGET	5:0	Selects between the logic block to be accessed by the SAAR register. This field is encoded as follows: 0x00: ITC 0x01: DSPRAM 0x02 - 0x03F: Reserved Writes to reserved values will be dropped.	R/W	0

11.1.1.2 Special Address Access Register — SAAR (CP0 Register 9, Select 7)

The 64-bit SAAR register is instantiated per-core and stores the base address and size of the DSPRAM block.

The 32-bit ADDR[47:16] field indicates the base address of the DSPRAM. This field is stored in bits 43:12 of the SAAR register.

The 5-bit SIZE field of this register encodes the size of the DSPRAM. This field is encoded as 2^SIZE to indicate the size of the DSPRAM. The default for this field is 64 KBytes. For example, the default value is 0x10, corresponding to a decimal value of 16. This means that the default DSPRAM size is 2^16, or 64 KBytes. Similarly, a value of 0x11 corresponds to a decimal value of 17, meaning that the size of the DSPRAM is 2^17, or 128 KBytes.

An ENABLE bit located in bit 0 of this register enables DSPRAM / ITC accesses. Once this information is programmed into the CP0 SAAR, it is moved by hardware into either the SAAR0_ITC or SAAR1_DSPRAM hardware registers depending on the programming of the SAARI TARGET field described above. Refer to the section entitled Accessing the DSPRAM for more information on these two hardware registers.

Refer to the section entitled Register Programming Sequence for more information on how the SAAR register is used.

Figure 11.2 SAAR Register Format

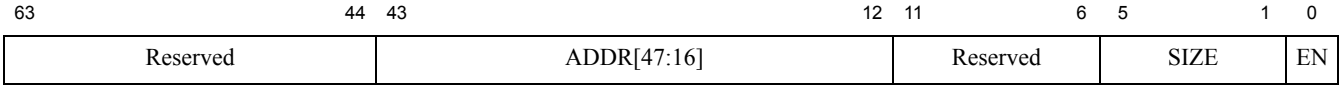


Table 11.3 Field Descriptions for SAAR Register

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	63:44	Reserved. This bits should be written as zero. Reads are undefined.	R	0

Table 11.3 Field Descriptions for SAAR Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
ADDR[47:16]	43:12	Base Address. This field specifies the base physical address for the location of the DSPRAM in memory. The address must be at least 64 KB-aligned (ADDR[47:16] = PA[47:16]).	R/W	0
Reserved	11:6	Reserved. This bits should be written as zero. Reads are undefined.	R	0
SIZE	5:1	Size of the device. Encoded as 2^SIZE bytes. This is preset at build time. For a 64 KB memory the SIZE field should be 0x10. The actual size of the device size can be set to a smaller than 64 KB, but the minimum size of the address window must be 64 KB. For example, if the memory occupies only 16 KB, the upper 48 KB of address would not be used.	R/W	0x10
EN	0	This enable bit must be set to allow DSPRAM accesses. A read gives the current value of the bit.	R/W	0

11.1.2 Changes to Existing CP0 Registers — Error Reporting

To accommodate error reporting by the DSPRAM, two existing CP0 registers have been modified as shown below.

11.1.2.1 Error Control — ErrCtl (CP0 Register 26, Select 0)

In the I6500 CP0 *ErrCtl* register, bit 31 is used to enable the DSPRAM to trigger on a cache error exception. When this bit is set and an error occurs, the CP0 *CacheErr* register provides details about the error. The remaining bits of this register behave the same as in previous cores. Refer to the I6500 CP0 Register document for more information.

11.1.2.2 Cache Error — CacheErr (CP0 Register 27, Select 0)

In the I6500 CP0 *CacheErr* register, the following fields are described. Note that only those bits that have changed are defined here. If not defined, the field(s) behave the same as in previous generation cores. Refer to the I6500 CP0 Registers document for more information.

- Bits 29:26 are used to indicate the array where the error was detected. Encoding 0x8 of this field was added to indicate a DSPRAM error.
- Bits 21:20 are used to indicate the way where the error occurred. However, since the DSPRAM is a 1-way set associative memory, this field is not used.

11.2 DSPRAM Software Interface

The DSPRAM is accessed by Load and Store instructions. Read requests for load instructions can be issued to the DSPRAM module speculatively. Write requests for store instructions are non-speculative. The read/write access to the DSPRAM is 16 Bytes (128 bits) wide for data.

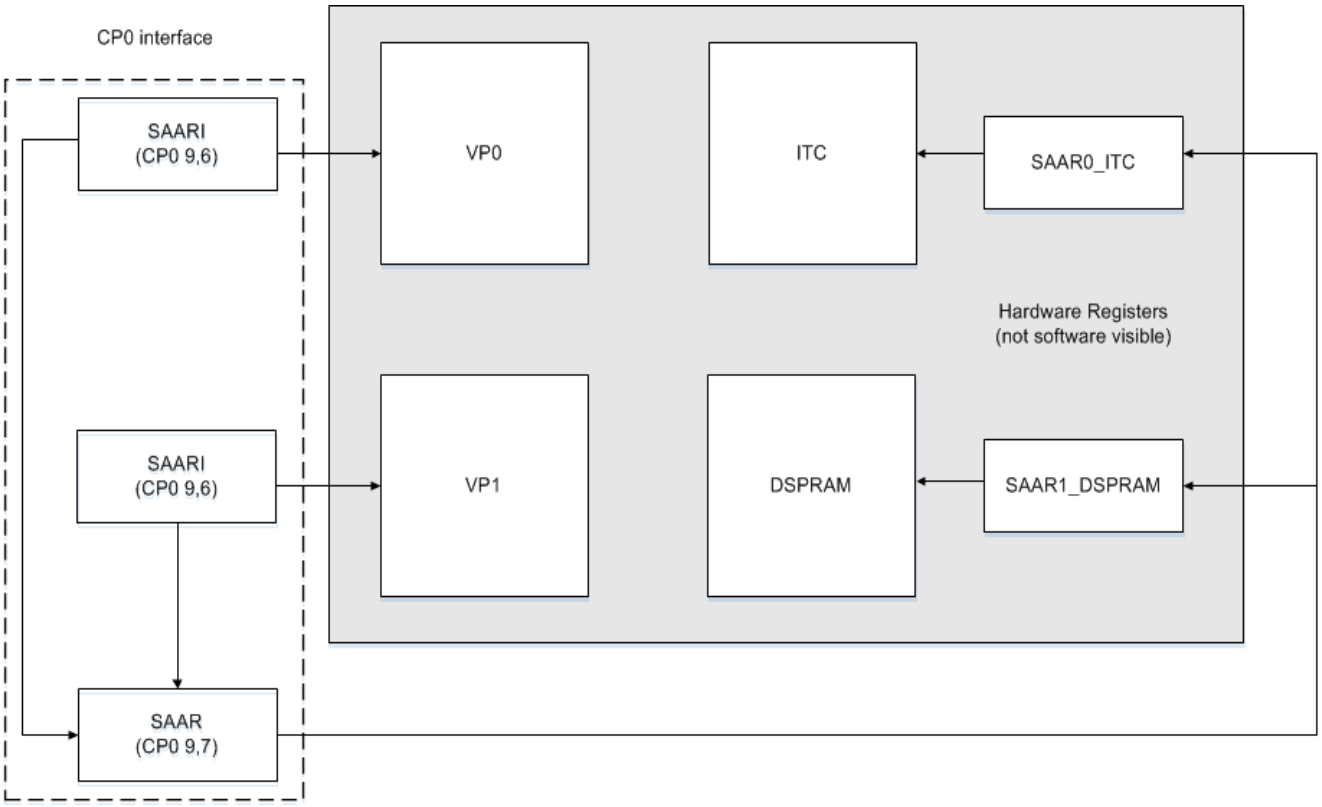
The CACHE, LL/SC variants, GINV*, and PREF instructions are not supported on the address space of the DSPRAM. The address of the load/store instruction to the DSPRAM must be aligned to the size of access (i.e. 4 bytes, 8 bytes, or 16 bytes). Any violation of the address alignment can cause an address error exception (i.e. unaligned loads/stores to DSPRAM are not supported). The SYNC instruction will enforce ordering of DSPRAM loads and stores.

11.3 Accessing the DSPRAM

As mentioned above, the DSPRAM is accessed using two CP0 registers; the *Special Address Access Register (SAAR)* located at CP0 (Reg 9, Sel 7), and the *Special Address Access Register Index (SAARI*) located at CP0 (Reg 9, Sel 6). From a kernel software perspective, there is one SAAR register per core, and one SAARI register per VP. This means that there can be multiple SAARI registers per core depending on the number of VP's instantiated.

From a hardware perspective, there are two SAAR registers per core. The SAAR0_ITC register is dedicated to the Inter-Thread Communication (ITC) unit, and the SAAR1_DSPRAM register is dedicated to the DSPRAM. These two registers are NOT software visible. Which register is accessed by hardware depends on the programming of the SAARI register as described in Figure 11.3. This figure shows a 2-VP implementation.

Figure 11.3 Accessing the DSRPAM



11.3.1 Register Programming Sequence

When the TARGET field of the SAARI register is set to 0x01, kernel software programs the SAAR register with the base address and size of the DSPRAM block. Hardware then uses this information to program the internal SAAR1_DSPRAM register that is dedicated to the DSPRAM, thereby setting the base address and the size of the DSPRAM block.

Conversely, when the TARGET field of the CP0 SAARI register is set to 0x00, kernel software programs the SAAR register with the base address and size of the ITC block. Hardware uses this information to program the internal SAAR0_ITC register that is dedicated to the ITC block.

For example, to select the DSPRAM block and set the address, size, and enable fields of the SAAR, the programming sequence is as follows:

1. Program the TARGET field of the CP0 SAARI register with a value of 0x01 to indicate a DSPRAM access.

2. Program the base address location of the DSPRAM in the ADDR[47:16] field of the CP0 SAAR register. This sets the base address for the DSPRAM. Note that this field resides in bits 43:12 of the register.

3. Program the SIZE field to indicate the size of the DSPRAM as described above if it is different from the default value of 64 KBytes. If the size is 64 KBytes, this field need not be programmed.

4. Set the ENABLE bit in the CP0 SAAR register to enable DSPRAM accesses.

5. This is done by the privileged software (i.e. by operating system software if Virtualization is not implemented or by hypervisor if Virtualization is implemented).

These steps can be represented by the following assembly language code, along with an example transfer of data:

```
#define c0_SAARI $9,6
#define c0_SAAR $9,7

li t0, 000000001 \\Set bit 0 to select DSPRAM as the target
mtc0 t0, c0_SAARI \\Write SAARI register with a value of 1 to select DSPRAM
dli t1, 0000000080000021 \\Set base address to 80000; set size to 64K; enable
                          \\DSPRAM
dmtc0 t1, c0_SAAR \\Write SAAR register with contents of register t1
dli t0, 5555AAAA5555AAAA \\load data value to be transferred to DSPRAM
dli t1, 0000000000082000 \\load address of 82000 into t1; this is offset 2000 from
the base address of 80000
sd t0, 0(t1) \\Store the data in t0 to the address in t1; address offset = 0; Data
             \\is stored to DSPRAM address 82000
```

11.3.2 Programming Constraints

The DSPRAM is shared across all VP's in the core. As such, accesses to the DSPRAM must adhere to the following constraints:

1. The SAARI register must be programmed before the SAAR register, so that hardware knows to move the contents of the CP0 SAAR register to either the SAAR0_ITC or SAAR1_DSPRAM hardware registers.

2. If multiple VP's are present, each VP can access the DSPRAM independent of the other. Therefore, if one VP stores data to a location in the DSPRAM, that data can be overwritten by another VP at any time.

3. Since there is only one SAAR register per core, each VP can write to the SAAR register. Therefore, if one VP sets the base address and size for the DSPRAM, that information can be overwritten by another VP at any time. For example, in the code example above, VP0 places the DSPRAM at a base of 0x80000 with a size of 64K, so the DSPRAM resides from 0x80000 - 0x8FFFF in memory. However, if VP1 sets the base address at a different value, such as 0xA0000, then the location of the DSPRAM will be moved.

It is incumbent upon software to ensure that these conditions do not occur.

Chapter 12

Inter-Thread Communication Unit

The Inter-thread Communication Unit (ITU) is a configuration option that provides an alternative to Linked-Load/Store-Conditional synchronization for fine grained multithreading by utilizing gating storage. Gating storage is used to synchronize execution streams, thereby allowing Inter-thread communication between threads. It is achieved through Load and Store requests which may be blocked until the state of the storage location changes, allowing a response to be given. A blocked load or store request can be precisely aborted if necessary and restarted by the controlling operating system.

The ITU is memory mapped and is accessed through load and store instructions. The ITU memory region is a fixed size of 128 KBytes.

The ITU is made up of cells. Each cell contains tag state information and associated data. Cells can be configured to be single element data storage cells, or multi-element FIFO data storage cells. Both cell types appear as a single addressable memory location, and the ITU can contain a mix of single element cells and multi-element FIFO cells. The cells are configured during IP configuration based on customer requirements.

12.1 Overview

The ITU module provides the following features:

- 64-bit data path
- 4 or 8 Byte read and write operations
- Eliminates spin locking encountered with the LL/SC instructions
- Provides support for semaphore types which reduces the number of extra instructions required to perform them
- Root physical address (RPA) is checked against base and range to validate access.

12.1.1 New CP0 Registers

Two new Coprocessor 0 registers have been added to facilitate access to the ITU as shown in Table 12.1 .

Table 12.1 CP0 Registers Used for Accessing the ITU Module

Register Number	Sel	Register Name	Description
9	6	SAARI	Special Address Access Register Index. Provides an index into the SAAR register to indicate whether the ITU or other block is being accessed. There is one SAARI register per VP.
9	7	SAAR	Special Address Access Register. Stores the base address where the ITU will be located, as well as the block size. There is one SAAR register per core.

The bit assignments for each of these registers is as follows.

12.1.1.1 Special Access Address Register Index — SAARI (CP0 Register 9, Select 6)

The 32-bit SAARI register is instantiated per-VP and provides an index value that determines whether the ITU is accessed, or another block is accessed such as the DSPRAM. Since there is one SAARI register per VP, this means that multiple SAARI registers use the same SAAR register to access the associated block.

Each SAARI register contains a 6-bit TARGET field that selects between the Inter-Thread Communication Unit (ITU) and the DSPRAM. If the value is set to 0x00, the ITU block is accessed. If the value is set to 0x01, the DSPRAM block is accessed. The default value for this field is 0x00, which is the ITU module.

Refer to the section entitled Register Programming Sequence for more information on how this register is used.

Figure 12.1 SAARI Register Format

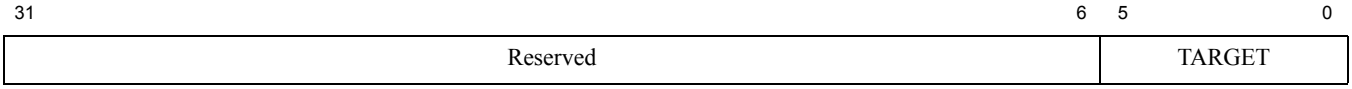


Table 12.2 Field Descriptions for SAARI Register

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	31:6	Reserved. This bits should be written as zero. Reads are undefined.	R	0
TARGET	5:0	Selects between the logic block to be accessed by the SAAR register. This field is encoded as follows: 0x00: ITU (default) 0x01: DSPRAM 0x02 - 0x03F: Reserved Writes to reserved values will be dropped.	R/W	0

12.1.1.2 Special Access Address Register — SAAR (CP0 Register 9, Select 7)

The 64-bit SAAR register is instantiated per-core and stores the base address and size of the ITU block.

The 32-bit ADDR[47:16] field indicates the base address of the ITU. This field is stored in bits 43:12 of the SAAR register.

The 5-bit SIZE field of this register encodes the size of the ITU. This field is encoded as 2^SIZE to indicate the size of the ITU. The default for this field is 128 KBytes, which corresponds to a decimal value of 17. This means that the default ITU size is 2^17, or 128 KBytes.

Note that default value for the SIZE field depends on which module is being accessed. If the SARRI.TARGET field is 0, indicating the ITU, a read of the SAAR.SIZE field yields a value of 0x11, which corresponds to a default size of 128 KBytes. However, if the SARRI.TARGET field is 1, indicating the DSPRAM, a read of the SAAR.SIZE field yields a value of 0x10, which corresponds to a default size of 64 KBytes.

An ENABLE bit located in bit 0 of this register enables DSPRAM / ITU accesses. Once this information is programmed into the CP0 SAAR, it is moved by hardware into either the SAAR0_ITU or SAAR1_DSPRAM hardware register depending on the programming of the SAARI.TARGET field described above. These registers are indirectly accessible through the CP0 SAAR register. Refer to the section entitled Accessing the ITU Module for more information.

Refer to the section entitled Register Programming Sequence for more information on how the SARRI register is used.

Figure 12.2 SAAR Register Format

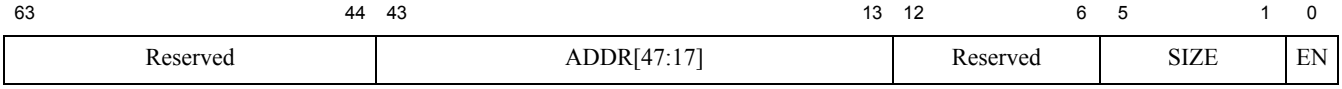


Table 12.3 Field Descriptions for SAAR Register

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	63:44	Reserved. This bits should be written as zero. Reads are undefined.	R	0
ADDR[47:16]	43:13	Base Address. This field specifies the base physical address for the location of the ITU block in memory. The address must be at least 128 KB-aligned (ADDR[47:17] = PA[47:17]).	R/W	0
Reserved	12:6	Reserved. This bits should be written as zero. Reads are undefined.	R	0
SIZE	5:1	Size of the device. Encoded as 2^SIZE bytes. For 128 KB, the SIZE field should be 0x11.	R/W	0x11
EN	0	This enable bit must be set to allow ITU accesses. A read gives the current value of the bit.	R/W	0

12.1.2 ITU Control Register

This register can be modified when accessing cell 0 of the ITU using cell view 0xF as described in Table 12.6. Note that the total number of cells is implementation dependent.

Table 12.4 CM3 Registers Used for Accessing the ITU Module

Register Name	Description
ICR0	ITU Control register 0. Contains information about the ITU configuration and controls for the global operation of the ITU. This register is memory mapped. It is not physically part of cell 0, but can be accessed when PA[6:3] contains a value of 0xF during an access to cell 0.

Figure 12.3 ICR0 Register Format

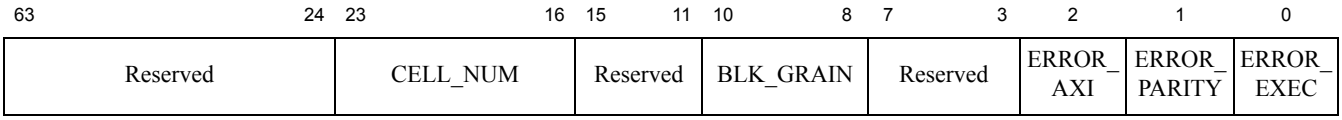


Table 12.5 Field Descriptions for ICR0 Register

Name	Bit(s)	Description	Read/ Write	Reset State
Reserved	63:24	Reserved. This bits should be written as zero. Reads are undefined.	R	0
CELL_NUM	23:16	Cell number. This field contains the number of cells configured in the ITU. Maximum number of cells is 32.	R	IP Config
Reserved	15:11	Reserved. This bits should be written as zero. Reads are undefined.	R	0
BLK_GRAIN	10:8	Block granularity. This field spreads out the cell addressing by virtually replicating the cells, giving multiple addresses to access the same cell. This is to allow cell access control by mapping individual or groups of cells across different pages. Refer to the examples in Section 12.2.5.	R/W	0
Reserved	7:3	Reserved. This bits should be written as zero. Reads are undefined.	R	0

Table 12.5 Field Descriptions for ICR0 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
ERROR_AXI	2	This bit is set by hardware to indicate an error on the AXI bus. Software must set this bit to clear the error.	R/W1	0
ERROR_PARITY	1	This bit is set by hardware to indicate a parity error occurred when accessing the ITU module. Software must set this bit to clear the error.	R/W1	0
ERROR_EXC	0	This bit is set by hardware to indicate an execution error during an access to the ITU module. Software must set this bit to clear the error.	R/W1	0

12.2 ITU Cell Structure

This section describes the ITU cell structure, including cell types, view, state, and indexing.

12.2.1 ITU Cell Types

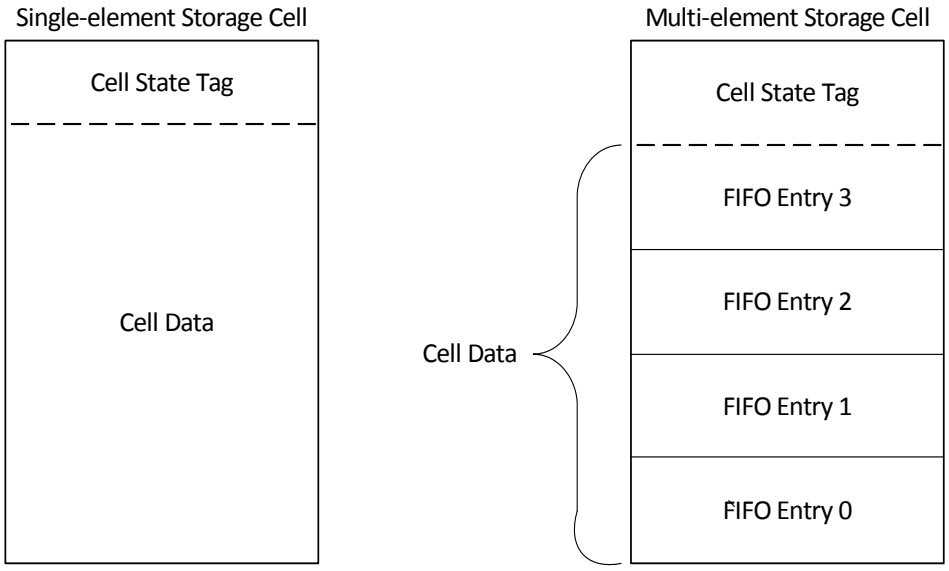
The ITU consists of a series of memory cells. Two cell types are supported;

- Single-element data storage cells
- Multi-element FIFO data storage cells

Single element storage cells contain a 64-bit data value and associated state information. Multi-element FIFO data storage cells contains multiple 64-bit data values in a FIFO format. The state information includes the FIFO depth, full/empty indication, and read pointer to indicate which entry in the FIFO is being read.

Note that a multi-element FIFO cell is accessed in the same manner as a single-element cell. In a 4-entry FIFO as shown in Figure 12.4, a write operation writes to entry 0 of the FIFO. Subsequent writes cause the data to be shifted through the entries in the FIFO. On a read operation, the first entry written to the FIFO is read out first.

Figure 12.4 Single- and Multi-Element Storage Cells



The ITU supports numerous kinds of cell operations, including cell data update, cell state update, and load/store blocking/nonblocking semphore requests based on the empty/full state of the target cell. The exact operation is embedded into bits 6:3 of the physical address when the ITU is accessed. Each type of cell can be accessed only in certain cell views as shown in Table 12.6. Refer to the section entitled Cell Views for more information.

Table 12.6 Cell Views Supported by Cell Type

Cell Type	Cell View	Description
Single Entry	Bypass Control E/F synchronized[1] E/F try P/V synchronized[2] P/V try	In the single entry cell type a single memory mapped word contains a single entry. This can be used to pass a single value between threads using E/F view or as an event counter using P/V view. All views are legal for single entry cells.
Multi-entry FIFO	Bypass Control E/F synchronized E/F try	In the multi-entry FIFO cell type a single memory mapped word contains multiple entries in a FIFO, allowing for hardware messaging queue support. The FIFO can be empty, full or neither depending on the number of entries already pushed onto the queue. Note, P/V type views are not allowed on multi-entry FIFO cells. In the Bypass view, a read operation will read the head of the FIFO. Bypass stores will overwrite the tail.

1. E/F = Empty/Full
2. P/V = Probeer (try)/Verhoog (increment)

12.2.2 Cell Views

The cell can operate in any of the views listed Table 12.6 above. Each of these views is described in the following table. The cell view is indicated in bits 6:3 of the physical address. Refer to the section entitled ITU Cell Addressing for more information. Refer to Table 12.6 for a definition of E/F and P/V.

Table 12.7 Cell Views

Addr[6:3]	Cell View	Blocking	Description
0000	Bypass	No	Load or Store of cell data, no effect on cell state/control information. Operation of the SC instruction is undefined when using this view.
0001	Control	No	Load or Store of cell state/control information.
0010	E/F synchronized	Yes	A Load request to an empty cell blocks the requesting thread. A Load request will set the cell empty flag, on completion, if it is returning the last available stored value. A Store request to a full cell will block the requesting thread. A Store request will set the cell full flag, on completion, if the cell is full after storing the last possible value. Operation of the SC instruction is undefined when using this view.

Table 12.7 Cell Views

Addr[6:3]	Cell View	Blocking	Description
0011	E/F/try	No	A Load request to an empty cell will fail, returning a value of 0. A Load request, which fails, will not affect cell state. A Store request to a full cell will not block, it will fail silently. A Store request, which fails, will not affect cell state. SC (Store Conditional) instructions referencing the E/F Try view will indicate success or failure based on whether the ITU store succeeds or fails.
0100	P/V synchronized	Yes	A Load request will block if the cell value is 0. A Load request will return the 16 least significant bits of cell value; zero extended, and causes an atomic post decrement by 1. A Store request will never block. A Store request's write data is ignored and an increment by 1 occurs. A Store request's increment saturates at 16 bits (0xFFFF) Load/Store requests will not affect cell state. The empty and full state bits should be cleared as part of the initialization for P/V semaphore use. Operation of the SC instruction is undefined when using this view.
0101	P/V try	No	A Load request will never block. A Load request will return the 16 least significant bits of cell value; zero extended, and causes an atomic post decrement by 1. (Decrement only occurs if cell value is not 0) A Store request will never block. A Store request's write data is ignored and an increment by 1 occurs. A Store request's increment saturates at 16 bits (0xFFFF) Load/Store requests will not affect cell state. The empty and full state bits should be cleared as part of the initialization for P/V semaphore use. Operation of the SC instruction is undefined when using this view.
0110 - 1110	Reserved	No	Behaves the same as the Bypass view.
1111	ICR Register	No	ITU control register access (ICR0) A Load request reads from the ICR0 register. A Store request writes to the ICR0 register.

12.2.3 Cell State

The cell state can be accessed in the Control view when PA[6:3]) = 0001 as described in Table 12.7. The cell state provides the following information.

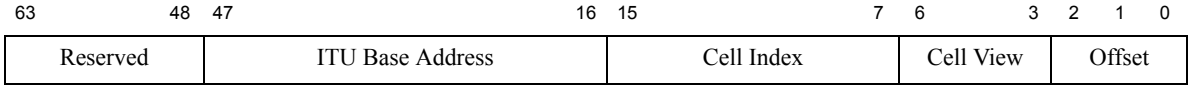
Table 12.8 Cell State

Bits	Field	Description	Type	Reset
63:30	R	Reserved.	R	0
29:28	FIFO_DEPTH	Indicates that the cell FIFO depth is 2^n, where n is the value in this field. This means that he FIFO can be encoded as follows: 00: 2^0 = 1 entry 01: 2^1 = 2 entries 10: 2^2 = 4 entries 11: 2^3 = 8 entries Note that single-element cells this field is always 0.	R	IP Config Value 0 (Single-element) 1 - 3 (Multi-element)
27:19 27:20 27:21	R	Reserved. The size of this field depends on the size of the FIFO_RD_PTR field. See below	R	0
18 19:18 20:18	FIFO_RD_PTR	Indicates the pointer value for the oldest entry whose value will be returned on the next read. The size of this pointer depends on the state of the FIFO_DEPTH field in bits 29:28. This field is encoded as follows: Bit 18: Used when the cell is configured as single-element, or as a multi-element cell with a 2-entry FIFO. A logic 0 on bit 18 selects the odd FIFO entry, and a logic 1 on bit 18 selects the even FIFO entry. In this configuration bits 20:19 are reserved. Bits 19:18: Used when the cell is configured as multi-element with a 4-entry FIFO. In this configuration bit 20 is reserved. Bits 20:18. Used when the cell is configured as multi-element with an 8-entry FIFO.	R	IP Config Value
17	FIFO_TYPE	0 = single entry cell 1 = multi-entry FIFO	R	IP Config Value
16:2	R	Reserved.	R	0
1	Full	Cell contains valid data (single-entry cell type) Cell contains the maximum number of entries in the FIFO (multi-entry FIFO cell type)	R/W	0
0	Empty	Cell contains no valid data.	R/W	1

12.2.4 ITU Cell Addressing

The ITU is accessed using a 48-bit physical address that is organized as follows.

Figure 12.5 ITU Cell Addressing Format



The address is divided into the following fields:

- ITU Base Address: This field is taken from bits 43:12 of the CP0 SAAR register described in Section 12.1.1.2.
- Cell Index: The size of the cell index varies depending on the number of cells. The position of the cell index within bits 15:7 is indicated by the state of the BLK_GRAIN field as described in the following subsections.
- Cell View: This field encodes one of the cell views as described in Table 12.7.
- Offset: Each cell is 64-bits wide. In this field bits 1:0 are always 0. Bit 2 is used to select between the upper and lower 32-bit words.

12.2.5 Cell Indexing Examples

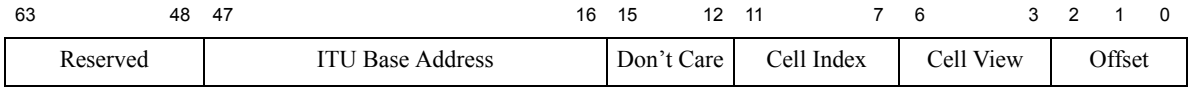
The Cell Index is derived from information stored in the CELL_NUM and BLK_GRAIN fields of the ICR0 register as described in Section 12.1.2. As shown in Figure 12.5, the cell index falls somewhere within bits 15:7 of the physical address. The exact location depends on the programming of these fields.

12.2.5.1 Example 1: 32 Cells with No Index Shift and No Invalid Cells

In this example the 32 cells are divided into 16 single-element cells and 16 FIFO cells. The value configured in the CELL_NUM field is 32 (0x20). The value programmed into the BLK_GRAIN field is zero.

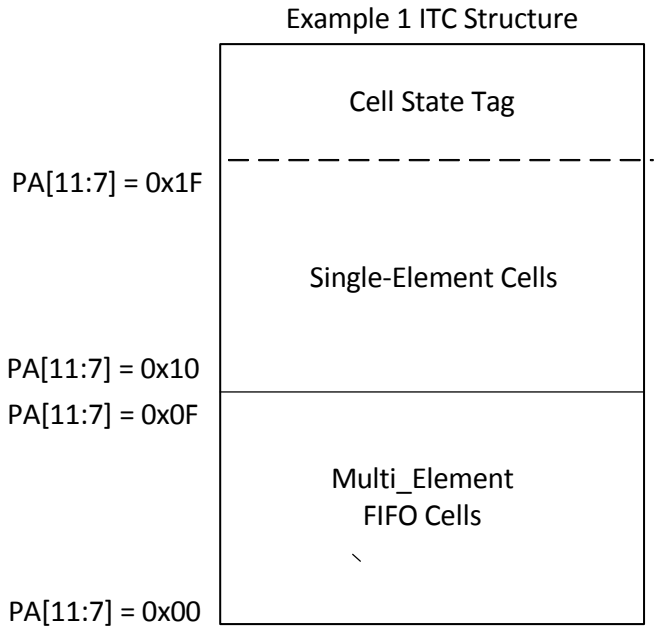
As shown in Figure 12.5, the cell index starts at PA[7]. Since 5 address bits are required to access these 32 cells, the Cell Index would reside at PA[11:7]. In this case, bits 15:12 are don't care. Since all 32 cells are used, there are no invalid cells. The physical address PA[63:0] would be organized as shown in Figure 12.6.

Figure 12.6 Example 1: 32 Cells with No Index Shift



In Figure 12.7, the cell state is accessed when PA[6:3] = 0001 as described in Table 12.7.

Figure 12.7 32 Cells with No Index Shift and No Invalid Cells

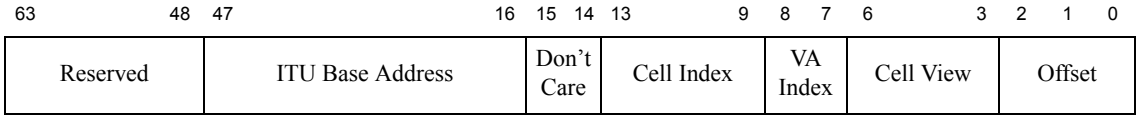


12.2.5.2 Example 2: 32 Cells with 2-Bit Index Shift and No Invalid Cells

In this example the 32 cells are divided into 16 single-element cells and 16 FIFO cells. The value configured in the CELL_NUM field is 32 (0x20). The value programmed into the BLK_GRAIN field is two.

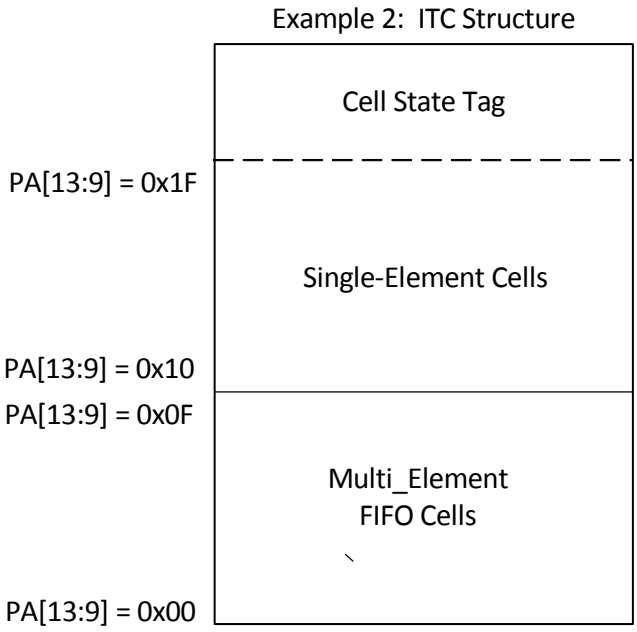
As shown in Figure 12.5, the cell index starts at PA[7]. In this example the BLK_GRAIN field contains a value of 0x2, causing the cell index to be shifted to the left 2 bits. Since 5 address bits are required to access these 32 cells, the Cell Index would reside at PA[13:9]. In this case, bits 15:14 are don't care. Bits 8:7 (VA Index) can be used to map the ITU to multiple virtual addresses, in this case up to four. In this example there are no invalid cells. The physical address PA[63:0] would be organized as shown in Figure 12.8.

Figure 12.8 Example 1: 32 Cells with 2-bit Index Shift



In Figure 12.9, the cell state is accessed when PA[6:3] = 0001 as described in Table 12.7.

Figure 12.9 32 Cells with 2-bit Index Shift and No Invalid Cells



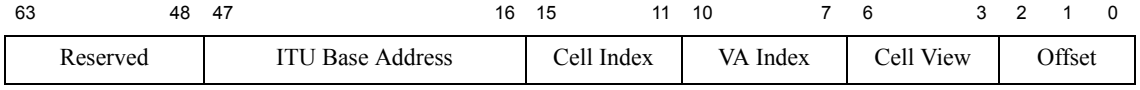
12.2.5.3 Example 2: 20 Cells with 4-Bit Index Shift and Invalid Cells

In this example there are 20 cells that are divided into 10 single-element cells and 10 FIFO cells. The value programmed into the CELL_NUM field is 20 (0x14). The value programmed into the BLK_GRAIN field is four.

As shown in Figure 12.5, the cell index starts at PA[7]. In this example the BLK_GRAIN field contains a value of 0x4, causing the cell index to be shifted to the left 4 bits.

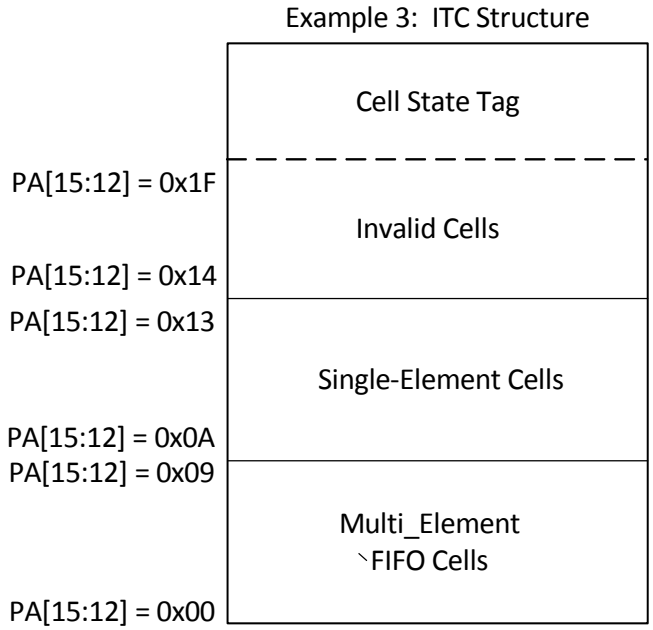
Even though only 20 cells are required, the number of memory locations allocated must be on a power of 2, which means that 5 address bits are still required to access this memory space. Since 5 address bits are required to access these 20 cells, the Cell Index would reside at PA[15:11]. In this case, bits 10:7 can be used to map the ITU to multiple virtual address, in this case up to 16. In this example there are 20 valid cells and 12 invalid cells. The physical address PA[63:0] would be organized as shown in Figure 12.10.

Figure 12.10 Example 1: 20 Cells with 4-bit Index Shift



In the diagram below, the cell state is accessed when PA[6:3] = 0001 as described in Table 12.7.

Figure 12.11 20 Cells with 4-bit Index Shift and Invalid Cells



12.3 ITU Software Interface

The ITU is accessed by Load (LD) and Store (ST) instructions. Write requests for store instructions are non-speculative. The read/write access to the ITU module is via a 64-bit wide data path.

From the core, ITU accesses are always uncached. When the physical address (PA) matches that in the SAAR0_ITU.ADDR[47:16] field, then the access is uncached. The SAAR0_ITU address match overwrites and Cache Coherency Attributes (CCA) that may be associated with the translated address.

The CACHE, GINV*, PREF, and SYNCI instructions are not supported on the address space of the ITU. The address of the load/store instruction to the ITU must be aligned to the size of access (i.e. 4 bytes or 8 bytes). Any violation of the address alignment can cause an address error exception (i.e. unaligned loads/stores to the ITU are not supported).

Store Conditional (SC) instructions referencing the ITU space are executed independent of the core's CP0 LLAddr.LLbit. If *E/F try* view is used, the SC instruction indicates success or failure depending on whether the ITU store succeeds or fails due to the Full state. In other views, SC pass/fail behavior is undefined.

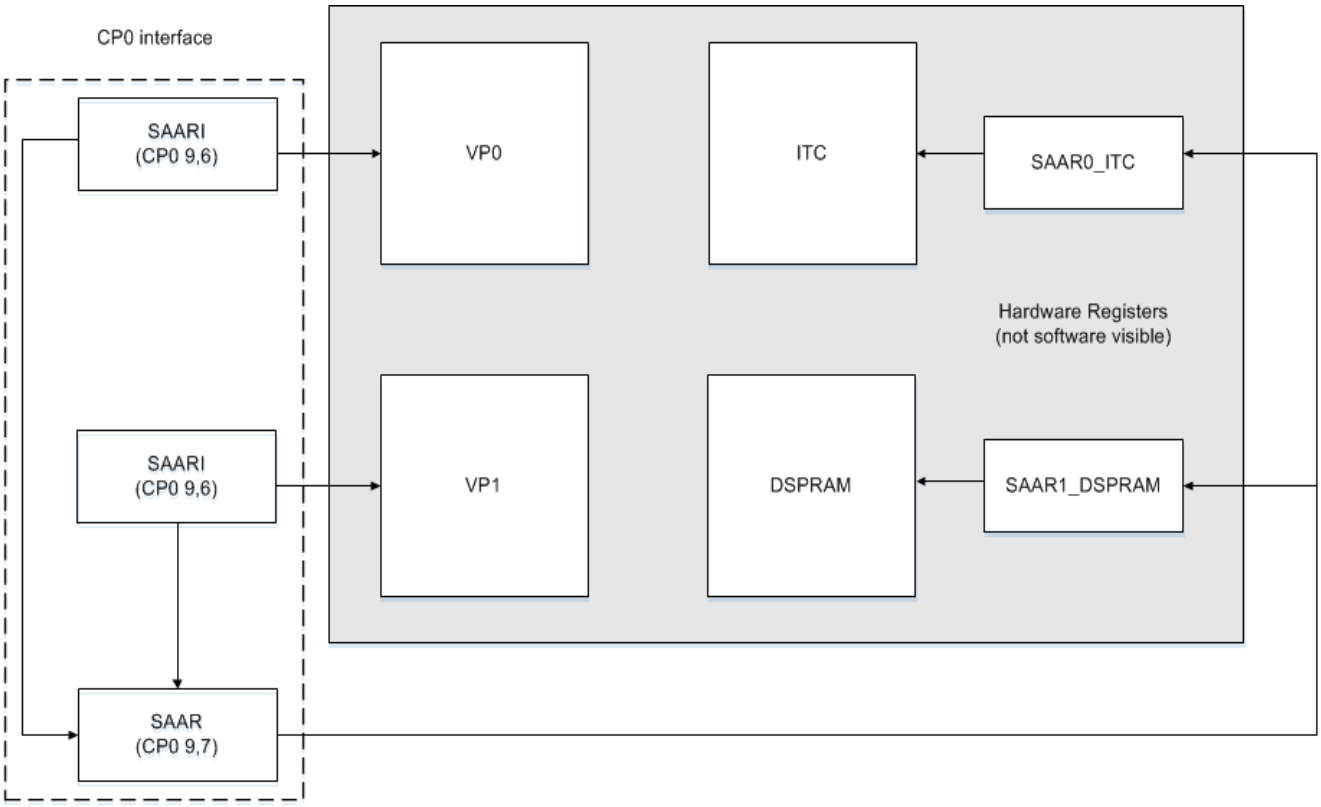
12.4 Accessing the ITU Module

As mentioned above, the ITU is accessed using two CP0 registers; the *Special Address Access Register (SAAR)* located at CP0 (Reg 9, Sel 7), and the *Special Address Access Register Index (SAARI*) located at CP0 (Reg 9, Sel 6). From a software perspective, there is one SAAR register per core, and one SAARI register per VP. This means that there can be multiple SAARI registers per core depending on the number of VP's instantiated. Refer to the section entitled ITU Software Interface for more information.

From a hardware perspective, there are two SAAR registers per core. The SAAR0_ITU register is dedicated to the Inter-Thread Communication (ITU) unit, and the SAAR1_DSPRAM register is dedicated to the DSPRAM. These two registers are not software visible. Which register is accessed by hardware depends on the programming of the

SAARI.TARGET field as described Figure 12.12. Both of these registers are indirectly accessed using the SAAR register described above. This concept is shown for a 2-VP core.

Figure 12.12 Accessing the ITU



12.4.1 Register Programming Sequence

When the SAARI.TARGET field is set to 0x00, software programs the SAAR register with the base address and size of the ITU block. Hardware then uses this information to program the internal SAAR0_ITU register that is dedicated to the ITU, thereby setting the base address and the size of the ITU block.

Conversely, when the TARGET field of the CP0 SAARI register is set to 0x01, software programs the SAAR register with the base address and size of the DSPRAM block. Hardware uses this information to program the internal SAAR1_DSPRAM register that is used to access the memory.

For example, to select the ITU block and set the address, size, and enable fields of the SAAR, the programming sequence is as follows:

1. Program the TARGET field of the CP0 SAARI register with a value of 0x00 to indicate an ITU access.

2. Program the base address location of the ITU in the ADDR[47:16] field of the CP0 SAAR register. This sets the base address for the ITU module. Note that this field resides in bits 43:12 of the register.

3. Set the ENABLE bit in the CP0 SAAR register to enable ITU accesses.

   Note that the ITU module is a fixed size of 128 KB. As a result, the SIZE field always contains a value of 0x11 and need not be programmed.

4. Read back the contents of the SAAR register. If the EN bit is not set, the ITU is not present in the system.

5. Programming is done by the privileged software (i.e. by operating system software if Virtualization is not implemented or by hypervisor if Virtualization is implemented).

These steps can be represented by the following assembly language code, along with an example transfer of data:

```
#define c0_SAARI $9,6
#define c0_SAAR $9,7

li t0, 000000000       \\Clear bit 0 to select ITU as the target
mtc0 t0, c0_SAARI      \\Write SAARI register with a value of 0 to select ITU
dli t1, 0000000040000031 \\Set base address to 40000; set size to 128K; enable ITU
dmtc0 t1, c0_SAAR          \\Write SAAR register with contents of register t1
dli t0, 5555AAAA5555AAAA   \\load data value to be transferred to ITU
dli t1, 0000000000042000   \\load address of 42000 into t1; this is offset 2000
                           \\ from the base address of 40000
sd t0, 0(t1) \\Store the data in t0 to the address in t1; address offset = 0; Data
             \\is stored to ITU address 42000
```

12.4.2 Programming Constraints

The ITU is shared across all VP's in the core. As such, accesses to the ITU must adhere to the following constraints:

1. The SAARI register must be programmed before the SAAR register.

2. Since there is only one SAAR register per core, each VP can write to the SAAR register. Therefore, if one VP sets the base address for the ITU, that information can beoverwritten by another VP at any time. For example, in the code example above, VP0 places the ITU at a base of 0x40000 with a size of 128K, so the ITU resides from 0x40000 - 0x5FFFF in memory. However, if VP1 sets the base address at a different value, such as 0xA0000, then the location of the ITU will be moved.

It is incumbent upon software to ensure that these conditions do not occur.

12.5 ITU Error Reporting

As shown in the section entitled ITU Control Register, the ICR0 register contains three error bits that are used by hardware to report the following error types.

- AXI bus error
- Parity error
- Execution error

12.5.1 AXI Bus Error

The ERROR_AXI bit in the ICR0 register is set by hardware when an error occurs on the AXI bus. The ITU only supports 32- and 64-bit transactions. It does not support burst transactions. As such, the SIZE field of the transaction must be equal to either 3'b010 [32bit] or 3'b011 [64bit]. If the value of this field is any other value, hardware sets the ICR0.ERROR_AXI bit.

12.5.2 Parity Error

The ERROR_PARITY bit in the ICR0 register is set by hardware when a parity error occurs during an access to the ITU module. Write transactions are even-parity checked when their data is received.

12.5.3 Execution Error

The ERROR_EXEC bit is set under any of the following conditions:

- On a E/F Load or Store to a single-element cell, when the E and F flags either both set or both cleared.
- On a P/V Load or Store to a single-element cell when cell does not have both the E and F cleared.
- On E/F Load or Store to a multi-element FIFO cell when cell has its E and F flags both set.
- On any P/V Load or Store to a multi-element FIFO cell

In addition, the ERROR_EXEC bit is set by hardware when either of the following invalid accesses occurs:

- Invalid address
- Invalid ICR0 register access

Invalid Address

The range of address bits used to decode the cell address varies based on the ICR0.BLK_GRAIN field and the number of cells configured based on the ICR0.CELL_NUM field. The cell address range is always a power of 2 regardless of how many cells are configured.

The remaining address space within the cell address range is part of the invalid cell address range. This range is only accessed when the total number of cells is not a power of 2. If a normal request is made to an invalid cell address range, an error response is returned and hardware sets the ICR0.ERROR_EXEC bit. For more information, refer to Example 2: 20 Cells with 4-Bit Index Shift and Invalid Cells

Invalid ICR0 Register Access

In the ITU, the ICR0 register can be accessed only in cell 0. As such, any ICR view request (PA[6:3] = 0xF) to a cell address which has no corresponding ICR register (any cell except cell 0) will generate an execution error, causing hardware to set the ICR0.ERROR_EXEC bit. Note that software will see this condition as a Bus Error.

Chapter 13

Multithreading

The I6500 Multiprocessing System (MPS) incorporates hardware multithreading that executes multiple threads in such a way that the threads appear to be run in parallel. This functionality is performed entirely in hardware and does not require any software control. Hence this chapter is only intended to provide an overview of multithreading and how it is implemented in the I6500 MPS.

In the I6500, each thread is referred to as a Virtual Processor (VP). Each VP contains a complete system state (General, CP0, FP, and MSA registers, TLB mappings, interrupt and exception model). In addition, each thread has its own system debug, reset and various boot and exception vectors, and memory coherency.

There are multiple types of multithreading implementations. The I6500 MPS implements Simultaneous Multithreading, where the core can execute multiple threads in parallel every cycle. In addition, instructions from different threads can execute at the same time in the same pipeline stage. This allows for maximum throughput and minimization of idle hardware during execution. The I6500 is a dual-issue machine, allowing up to two threads to execute in a single pipeline stage. In the I6500, all threads (up to 4) can be fetched, decoded, issued, executed, and graduated in parallel.

13.1 Instruction Flow

The I6500 Instruction Fetch Unit (IFU) fetches instructions from a shared Instruction Cache (IC) for all four threads. It fetches two instructions (for a single thread) in a cycle, using a program counter (PC) for that thread. This pair of instructions are sent to the Execution Unit (EXU). The IFU fetches instructions in a round-robin manner.

The IFU also manages a shared Instruction TLB (ITLB) structure. The ITLB performs instruction address translation, allowing complete independence amongst threads. This ITLBs are backed up by the larger Variable TLB (VTLB) and Fixed TLB (FTLB). The number of shared ITLB entries depends on the number of VPs implemented.

- 1 VP = 6 entries
- 2 VPs = 12 entries
- 4 VPs = 18 entries

For example, if there is only one VP, all entries of the ITLB are used by the VP. Conversely, if there are four VPs, there are 18 ITLB entries that are shared between all of the VPs.

In order to support virtualization, the thread's Instruction Virtual Address (IVA) is translated to a Guest Physical Address (GPA) and then the GPA is translated to Root Physical address (RPA). The ITLBs are used to store the double translation to minimize the number of entries and more importantly improve performance by doing the double translation in a single cycle.

The translated instructions are passed to the Execution Unit (EXU), which is responsible for decoding, issuing, executing and graduating the instructions. In addition, the EXU resolves all data and resource conflicts and manages precise exceptions. In the I6500, the instructions are issued and graduated in order.

Every cycle, the EXU decodes the top two instructions from each of the (up to) four threads and determines which two (of the possible eight) instructions are ready to issue based on resource availability and data dependencies. It is capable of issuing instructions from any of the four VPs, hence the term Simultaneous Multithreading (SMT). Note that the I6500 always issues instructions in order. If multiple instructions (>2) are available to issue, the EXU uses a fair issue policy to make sure all threads get equal representation.

Once the instructions are issued, they are executed in one of the functional units. During its execution, each instruction is appropriately tagged for thread identification and instruction order. This allows the proper instruction order to be maintained at graduation (completion) time. If an instruction completes, but an earlier instruction from the same thread has not graduated, the completed instruction remains in the graduation queue to maintain in-order completion.

13.2 Data Flow

Like the IFU mentioned above, the Load-Store Unit (LSU) manages a shared data cache to perform loads and stores for all threads. The LSU also performs a load or a store for a single thread in a cycle, but multiple loads and stores from differing threads can be queued up to access the data cache.

The LSU processes loads and stores in the order received to maintain cache coherency between threads. The data cache is organized as 4-way set associative cache, which eliminates most of the cache conflicts.

The LSU also manages a shared Data TLB (DTLB) structure. The DTLB performs data address translation, allowing complete independence amongst threads. The shared DTLB is backed up by the larger Variable TLB (VTLB) and Fixed TLB (FTLB). The number of shared ITLB entries depends on the number of VPs implemented.

- 1 VP = 8 entries
- 2 VPs = 14 entries
- 4 VPs = 20 entries

For example, if there is only one VP, all eight entries of the DTLB are used by the VP. Conversely, if there are four VPs, there are 20 DTLB entries that are shared between all of the four VPs.

In addition, the 16 dual-entry Variable TLB (VTLB) is instantiated on a per-VP basis. The 512 dual-entry Fixed TLB (FTLB) is shared between all VPs.

In order to support virtualization, the thread's Data Virtual Address (DVA) is translated to a Guest Physical Address (GPA) and then the GPA is translated to a Root Physical address (RPA). The DTLBs operate much like the ITLBs to perform a double translation in a single cycle.

Data stored by one thread does not become visible to other threads until the store instruction has graduated and the core has obtained ownership of the associated cache line (for cacheable accesses). In other words, data stored by one thread becomes visible to other threads in the same core at exactly the same point that it becomes visible to other cores in the system.

The I6500 manages allocation of shared resources (such as data buffers) between threads to prevent starvation and ensure that all threads can make forward progress.

13.3 Thread Management

Each of the threads operate independently, except to share some common resources. However, there are times when the processor needs to make sure the system is being accessed in a very controlled manner. The MIPS R6 Instruction

Set Architecture (ISA) includes specialized instructions to manage threads. These instructions allow a thread, operating in privileged state, to suspend or resume the execution of other threads to achieve that goal.

13.3.1 Disable Virtual Processor (DVP) Instruction

The DVP instruction suspends execution in all other threads. For the suspended threads, no state information is lost and the thread can be restarted exactly where they left off. DVP is a privileged instruction and is only available to Root (Hypervisor) Kernel operating mode. When appropriate, the suspended threads can be re-enabled via the EVP instruction.

13.3.2 Enable Virtual Processor (EVP) Instruction

The EVP instruction re-enables execution in all other threads. There is no loss of information for the resumed thread. The only visible effect might be the system timer continues to count while a thread is suspended, hence the system timing could be changed. This is a privileged instruction and is only available to Root (Hypervisor) Kernel operating mode.

13.4 Independent Exception Model

Since each thread has a completely independent exception model, one thread cannot block another thread. This independent exception model includes: Synchronous Exceptions (Overflow, TLB Miss, etc.), Asynchronous Interrupts (Int, NMI, etc.), Debug Exceptions (DIint), and Reset. A thread can be reset to reboot, while the other threads are completely unaffected.

Chapter 14

MIPS On-Chip Instrumentation

This chapter provides a brief overview of the interface and external debugging environment required to debug MIPS processors that incorporate the MIPS On-Chip Instrumentation (OCI) debug system for multi-core designs. Please refer to the following community pages link for information on MIPS probes, Codescape debug tools, SDKs and documentation: https://www.mips.com/develop/tools/

The MIPS OCI debug system has been developed to provide comprehensive debugging and performance-monitoring capabilities for multi-core processor designs where there can be two or four Virtual Processors (VP) per core or multiple cores per cluster.

14.1 OCI Debug System Overview

The MIPS OCI Debug System comprises a dedicated on-chip module called the Debug Unit and various on-chip components that have dedicated debug resources from which debug data is gathered. These are connected by a Register Bus (RRB).

14.1.1 Debug Unit (DBU)

There is one DBU per cluster of cores or Virtual Processors (VPs) in a system. The DBU provides several functions to assist debugging. Figure 14.1 shows the OCI system as implemented in a typical single-cluster core.

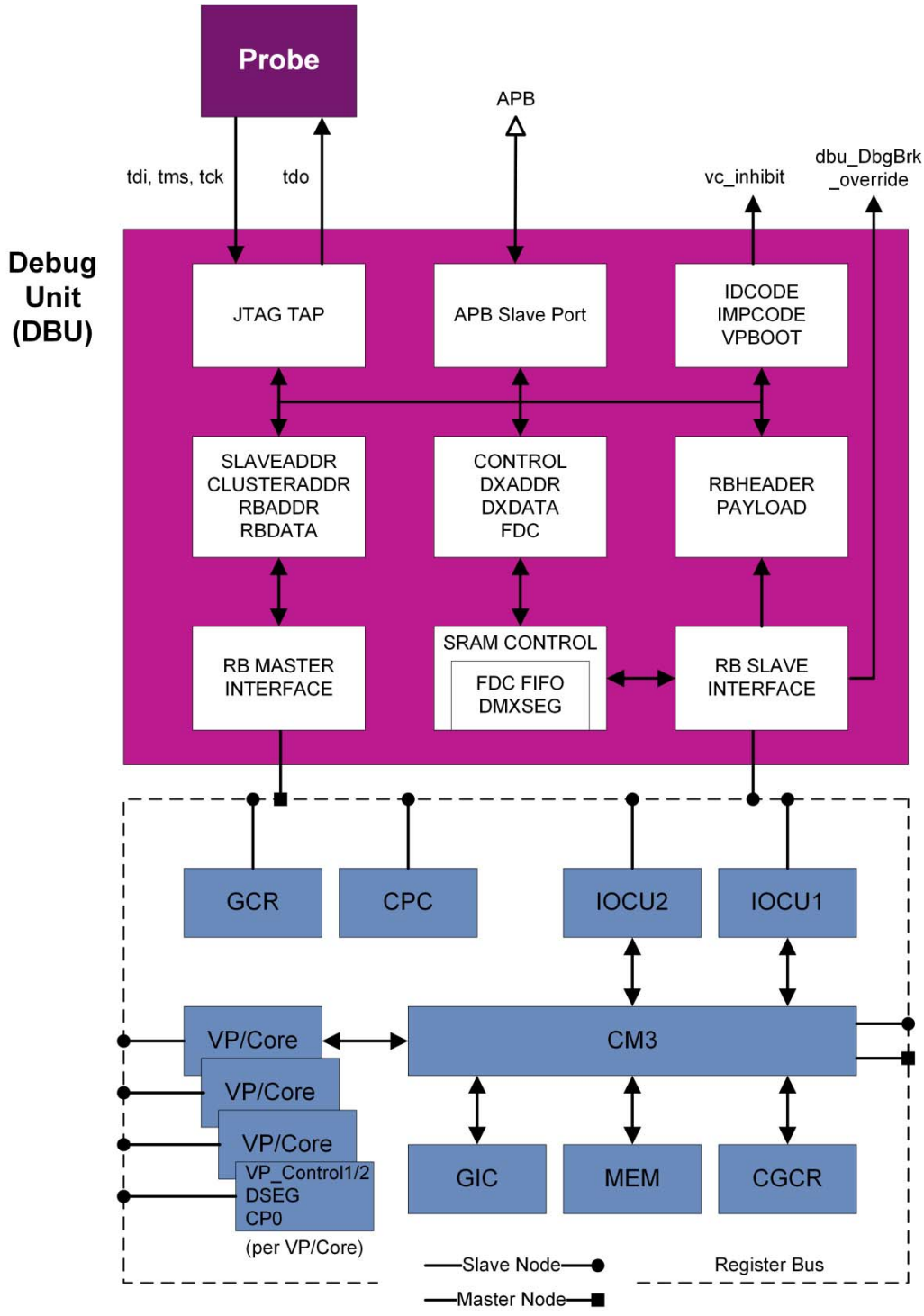


Figure 14.1 OCI System Block Diagram

14.1.1.1 APB Slave Port

An APB Slave Port in the DBU provides connection to an APB enabled on-chip debug controller or emulator transactor interface.

14.1.1.2 JTAG TAP

A serial JTAG TAP allows connection to a JTAG debug probe. The JTAG TAP data registers reside in the DBU and allow read/write requests to the VPs being debugged via debug monitor code.

14.1.1.3 Debug Monitor

Debug monitor code is loaded into RAM in the DBU and schedules debug commands to the VPs via the Register Bus.

14.1.1.4 RAM

A dedicated block of RAM in the DBU that hosts the debug monitor code and contains the memory mapped area, dmxseg. Dmxseg is a mapped to the VPs debug memory segment, dmseg, and is accessed by the VP when running in debug mode and when a debug probe is attached. This RAM also contains the FIFOs for Fast Debug Channels.

14.1.2 Register Bus

The DBU connects to VPs and other cluster-level coherent devices on the Register Bus (RRB) using a packet-based protocol.

14.1.3 Number of Breakpoints

The I6500 MPS implements 8 instruction triggers, the lower 4 of which have range triggering and the upper 4 have equality/mask, and 4 data triggers. Breakpoints are shared between all VPs.

14.1.4 Per Core/VP Resources

14.1.4.1 Breakpoint Controller

Each VP has its own independent breakpoint control logic and configuration registers.

14.1.4.2 Dseg

A memory mapped area of main memory, accessible from the processor in debug mode only. It contains the combined dmseg and drseg areas.

14.1.4.3 Dmseg

The debug memory segment of dseg that is accessed by the core when running in debug mode when a debug probe is attached. This area is mirrored by dmxseg in DBU RAM.

14.1.4.4 Drseg

A region of dseg that includes registers that are mapped to debug resources such as breakpoint configuration registers and sampling registers. The VP and the DBU can read and write these registers indirectly via the coherence manager (CM). Drseg registers can be accessed from the DBU during normal and debug mode execution.

14.1.4.5 CP0 Registers

CP0 contains specific registers that facilitate and configure various aspects of a VP's debug features.

14.1.5 Coherence Devices

The I6500 Multiprocessing System contains the following coherent devices.

14.1.5.1 GIC (Global Interrupt Controller)

This handles the distribution of interrupts between the VPs in a cluster or core and provides signals to put VPs into Debug Mode. The GIC also provides debug mode status monitoring and controls debug team assignments for synchronous stop/go of multiple VPs.

14.1.5.2 CPC (Cluster Power Controller)

Provides stop/run signals for VPs; reset occurred signals for the DBU, CM and VPs; registers for determining the state of each VPs power and clock rate; and power up and clock gating of the CM.

14.1.5.3 GCR (Global Configuration Registers)

A set of memory mapped registers that are used to configure and control various aspects of the CM, the coherence scheme and CM performance counters.

14.1.5.4 CGCR - (Custom Global Configuration Registers)

An optional block of custom registers that can be used to control system level functions.

14.1.5.5 CM - (Coherence Manager) (v3)

Controls the global ordering of requests and responses across core devices.

14.1.5.6 IOCU (I/O Coherence Unit)

Connects coherent devices to the Coherence Manager.

14.2 More Information

For more information on the MIPS OCI debug system, refer to the document entitled;

MIPS On-Chip Instrumentation; Debug Technical Reference Manual

The I6500 MPS also contains PDTrace and CMTrace functionality that can be used to aid in the debug process. For more information, refer to the document entitled;

MIPS On-Chip Instrumentation PDtrace Specification

Appendix A

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
01.00	March 29, 2017	Initial version of I6500 Multi-Cluster Programmers Guide