MIPS32® I7200 Multiprocessing System Programmer's Guide

Revision: 01.20
20/04/2018

Contents

List of Figures

List of Tables

1 MIPS32 I7200 Multiprocessing System Programmer's Guide

This document describes the software-programmable aspects of the 32-bit MIPS I7200 Multiprocessing System (MPS). The I7200 architecture combines a multi-threading pipeline with a highly intelligent coherence manager to deliver best-in-class computational throughput and power efficiency. This document describes how to control the hardware using registers and assembly code. The register-programming examples describe a programming sequence to set or change a programmable parameter using registers. The assembly code examples show how you use the MIPS instruction set to perform the same function.

Each chapter provides the relevant background information programmers require to understand the examples. Each block has common examples such as enabling and initialization, as well as in depth examples specific for that block.

This document describes the following blocks:

- **Memory Management (MMU):** This chapter describes the programmable elements of the Translation Lookaside Buffer or TLB of the I7200 MPS. 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).
- **Enhanced Virtual Address (EVA):** This chapter describes how to implement Enhanced Virtual Address or EVA, which allows for more efficient use of the 32b address space.
- **Memory Protection Unit (MPU):** This chapter describes an alternate to the TLB-based MMU. It provides an overview of the MPU architecture and describes how to program the MPU.
- **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 register interface 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 Coherence Manager with integrated L2 cache (CM) is responsible for establishing the global ordering of requests and for collecting the intervention responses and sending the correct data back to the requester. This chapter describes the CM and provides programming examples.
- **Cluster Power Controller (CPC):** This chapter provides an overview of how power is managed in the I7200 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.
- **Policy Manager:** The Policy Manager provides longer-term hints to the Dispatch Scheduler to achieve the desired system performance allocation. This chapter describes the Thread Scheduling Unit and Policy Manager modes.

- **Inter-Thread Communication Unit (ITU):** The ITU provides an alternative to Load-Linked/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.
- **SPRAM:** The optional Scratch Pad RAM (SPRAM) blocks provide a general scratch pad RAM used for temporary storage of data. The SPRAM 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.
- **Multithreading:** This chapter provides an overview of the hardware multi-threading mechanism in the I7200 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.

Note: Refer to the I7200 Datasheet for a complete feature list.

These chapters include assembly language examples that describe how various programming elements are handled in software. These examples can be used by programmers 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 toolchain. As such, it is not necessary for the programmer to execute these code examples manually when using Codescape because this functionality is already built into the software.

This document is meant to be used with the following documents:

- MIPS32 I7200 Multiprocessor Core Family Datasheet
- 32-bit MIPS I7200 Multiprocessing System Integrator's Guide. This companion document provides hardware details about the device, including functional verification, system integration, and system implementation.

1.1 Overview

The MIPS32® I7200 multiprocessing system (MPS) is a high performance multi-core cluster licensable IP solution. It is designed to deliver both high performance and low-latency responsiveness for system-on-chip (SoC) applications requiring rapid processing of real-time events.

Each core within the multi-core cluster is based on a 9-stage, dual-issue in-order pipeline with support for hardware multi-threading, designed to deliver high throughput and performance, and best-in-class efficiency per unit power and area. To complement the efficient pipeline design, the I7200 MPS utilizes the nanoMIPS instruction set architecture (ISA) to deliver this performance in smallest code size, providing for optimal use of the local memory resources to the CPU.

The cores of the I7200 MPS are coherently connected together via a Coherence Manager (CM, version 2.6) functional block, which includes a number of system level features and functional elements, including:

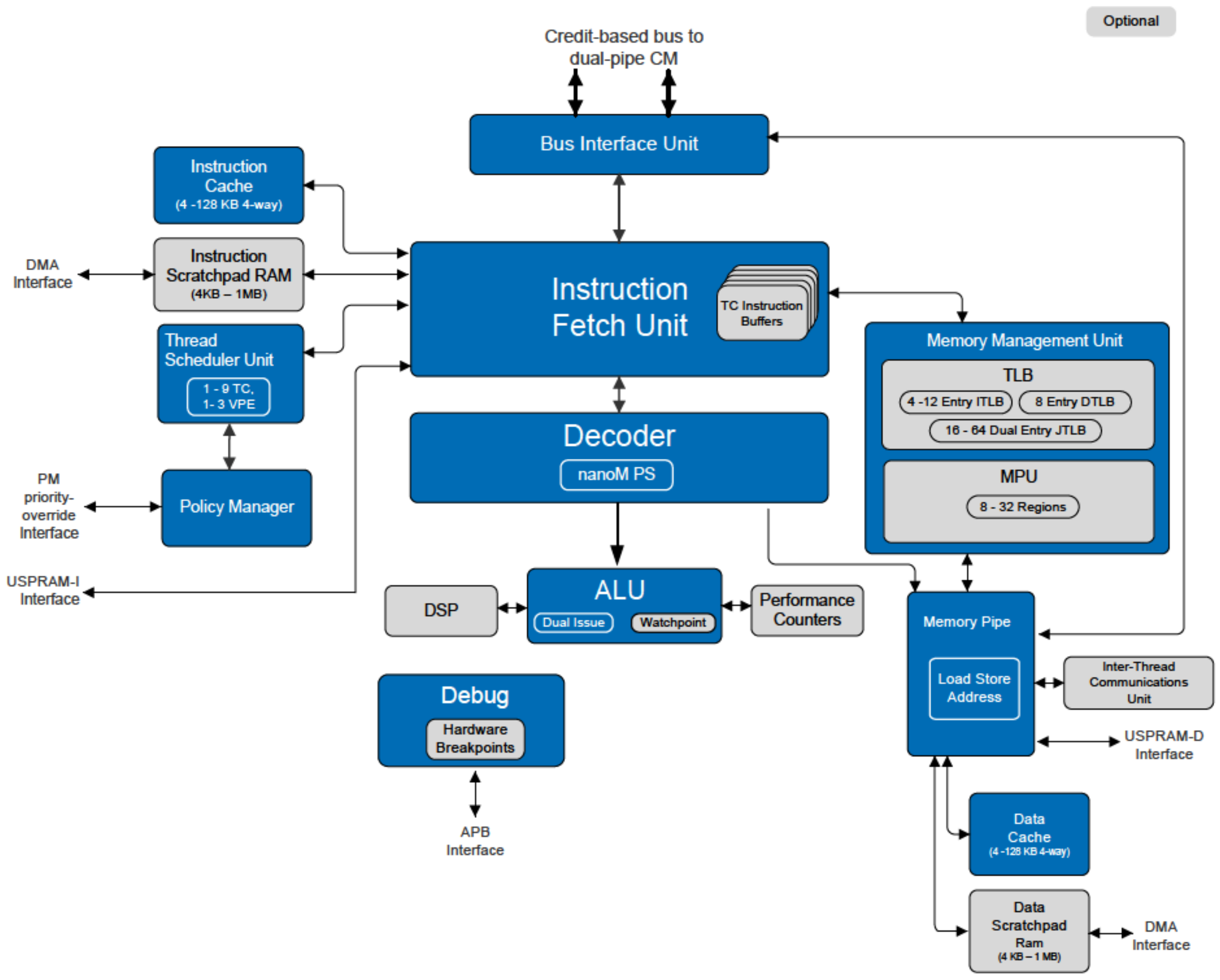
- Shared L2 cache
- Optional Global interrupt controller (GIC)
- Optional Cluster power controller (CPC)
- Accelerator/ IO coherency ports
- Global configuration registers (GCR)

The entire system offers many configurable options at the core and cluster level, and is available as fully synthesizable RTL for implementation in any semiconductor process technology.

1.2 I7200 Core Block Diagram

The following figure shows a block diagram of a single I7200 core.

Figure 1: I7200 Core-Level Block Diagram



Related Concepts

Memory Management Unit on page 17
Memory Protection Unit on page 47
Caches on page 63
Policy Manager on page 163
Inter-Thread Communication Unit on page 167

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 I7200 MMU implements a Translation Lookaside Buffer (TLB). An alternative to the TLB is the Memory Protection Unit. See *Memory Protection Unit* on page 47 for details.

This chapter covers the programmable elements of the TLB in the I7200 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.

The I7200 TLB 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 Memory Management Unit Architecture

The Memory Management Unit (MMU) in the I7200 core consists of three address-translation lookaside buffers (TLB).

- 4 - 12 entry Instruction TLB (ITLB)
- 8-entry Data TLB (DTLB)
- 16, 32, or 64 dual-entry Joint Translation Lookaside Buffer (JTLB) per VPE

When an instruction address is to be translated, the ITLB is accessed first. If the translation is not found, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken. Similarly, when a data reference is to be translated, the DTLB is accessed directly. If the address is not present in the DTLB, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken.

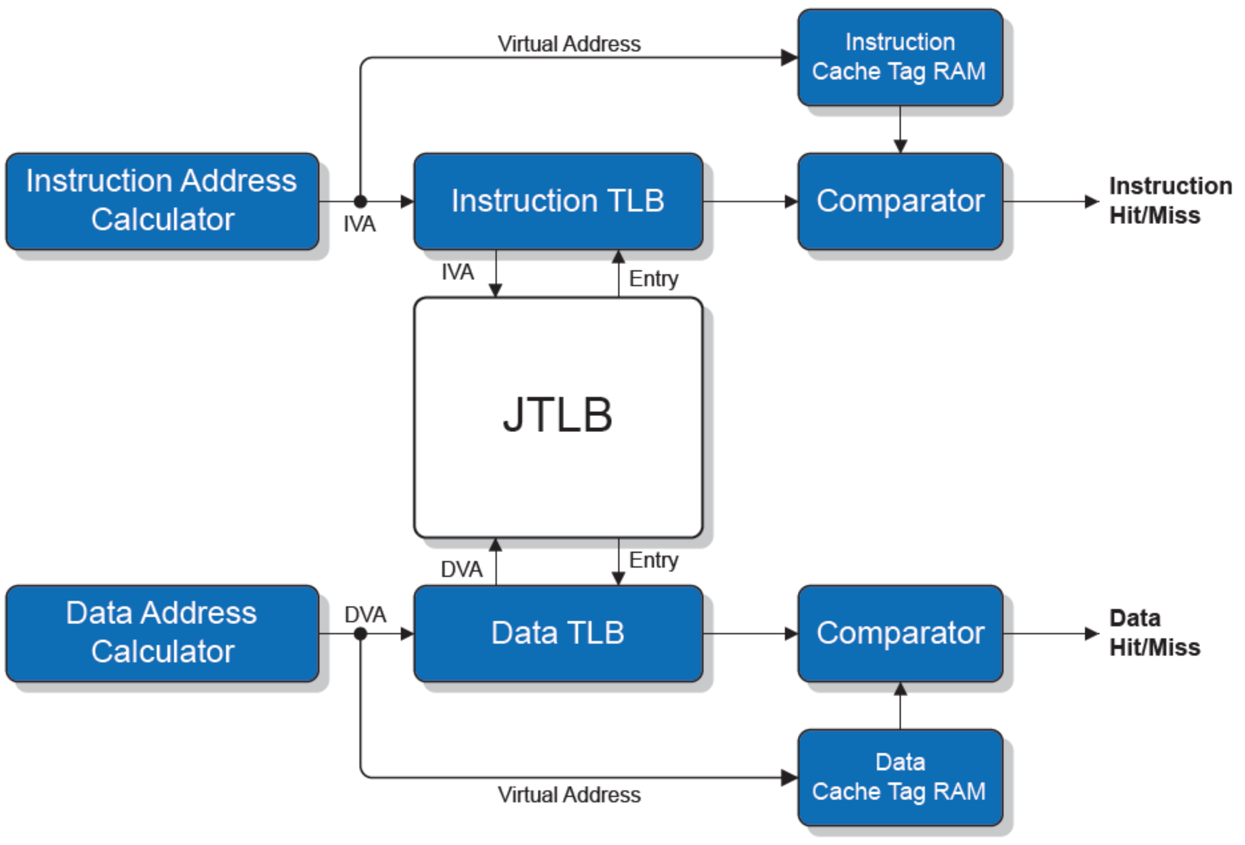
2.1.1 Translation Lookaside Buffer (TLB)

The basic TLB functionality is specified by the MIPS32 Privileged Resource Architecture. A TLB provides mapping and protection capability with per-page granularity. The I7200 implementation allows a wide range of page sizes to be simultaneously present.

The TLB contains a fully associative Joint TLB (JTLB). To enable higher clock speeds, two smaller micro-TLBs are also implemented: the Instruction Micro TLB (ITLB) and the Data Micro TLB (DTLB). When an instruction or data address is calculated, the virtual address is compared to the contents of the appropriate micro TLB (uTLB). If the address is not found in the uTLB, the JTLB is accessed. If the entry is found in the JTLB, that entry is then written into the uTLB. If the address is not found in the JTLB, a TLB exception is taken.

Figure 4 shows how the ITLB, DTLB, and JTLB are implemented in the I7200 CPU.

Figure 2: I7200 Core Address Translation



2.1.2 Joint TLB (JTLB)

The JTLB is a fully associative TLB cache containing 16, 32, or 64-dual-entries per VPE mapping up to 128 virtual pages to their corresponding physical addresses. The address translation is performed by comparing the upper bits of the virtual address (along with the ASID) against each of the entries in the *tag* portion of the joint TLB structure.

The JTLB is organized as pairs of even and odd entries containing pages that range in size from 4 KB to 256 MB, in factors of four, into the 4 GB physical address space. The JTLB is organized in page pairs to minimize the overall size. Each *tag* entry corresponds to two data entries: an even page entry and an odd page entry. The highest order virtual address bit not participating in the tag comparison is used to determine which of the data entries is used. Because page sizes can vary on a page-pair basis, the determination of which address bits participate in the comparison and which bit is used to make the even-odd determination is decided dynamically during the TLB look-up.

2.1.3 Instruction TLB (ITLB)

The ITLB is managed by hardware and is transparent to software. The larger JTLB is used as a backing structure for the ITLB. If a fetch address cannot be translated by the ITLB, the JTLB is used to translate it.

The ITLB contains between 4 and 12 entries and is dedicated to performing translations for the instruction stream. The ITLB is a hybrid structure having 3 entries that are shared by all TCs plus an additional entry dedicated to each TC. Therefore, a core with one VPE and one TC would have a 4-entry TLB with all entries dedicated to one TC. Conversely, a core with 1 VPE and 9 TC's would have a three shared entries, plus one entry per TC, for a total of 12 entries.

The ITLB maps 4 KB or 1 MB pages/subpages. For 4 KB or 1 MB pages, the entire page is mapped in the ITLB. If the main TLB page size is between 4 KB and 1 MB, only the current 4 KB subpage is mapped. Similarly, for page sizes larger than 1 MB, the current 1 MB subpage is mapped.

2.1.4 Data TLB (DTLB)

The DTLB is managed by hardware and is transparent to software. The larger JTLB is used as a backing structure for the DTLB. If a load/store address cannot be translated by the DTLB, a lookup is done in the JTLB. The JTLB translation information is copied into the DTLB for future use.

The DTLB is an 8-entry, fully associative TLB dedicated to performing translations for loads and stores. All entries are shared by all TCs. Similar to the ITLB, the DTLB maps either 4 KB or 1 MB pages/subpages.

2.2 TLB Instructions

The following table describes the TLB-related instructions in the I7200 core.

Table 1: TLB Instructions

Mnemonic	Instruction	Description
TLBP	Translation Lookaside Buffer Probe	Used to determine whether a particular address was successfully translated. When a TLBP instruction is executed and fails to find a match for the specified virtual address, hardware sets bit 31 of the Index register.
TLBR	Translation Lookaside Buffer Read	
TLBWI	Translation Lookaside Buffer Write Index	TLB write extended to support invalidation of individual TLB entries.
TLBWR	Translation Lookaside Buffer Write Random	
TLBINV	Translation Lookaside Buffer Invalidate	Added to support set level invalidation of TLB entries.
TLBINVF	Translation Lookaside Buffer Invalidate Flush	Added to support TLB flush based invalidation of TLB entries.

2.3 Relationship of TLB Entries and CP0 Registers

Each TLB entry in the JTLB consists of a tag portion and dual-data portion as shown in *Figure 3: Relationship Between CP0 Registers and TLB Entries* on page 20. In this figure, the following registers are used to manage the TLB entries.

- EntryLo0 (CP0 Register 2, Select 0)
- EntryLo1 (CP0 Register 3, Select 0)
- EntryHi (CP0 Register 10, Select 0)
- PageMask (CP0 Register 5, Select 0)

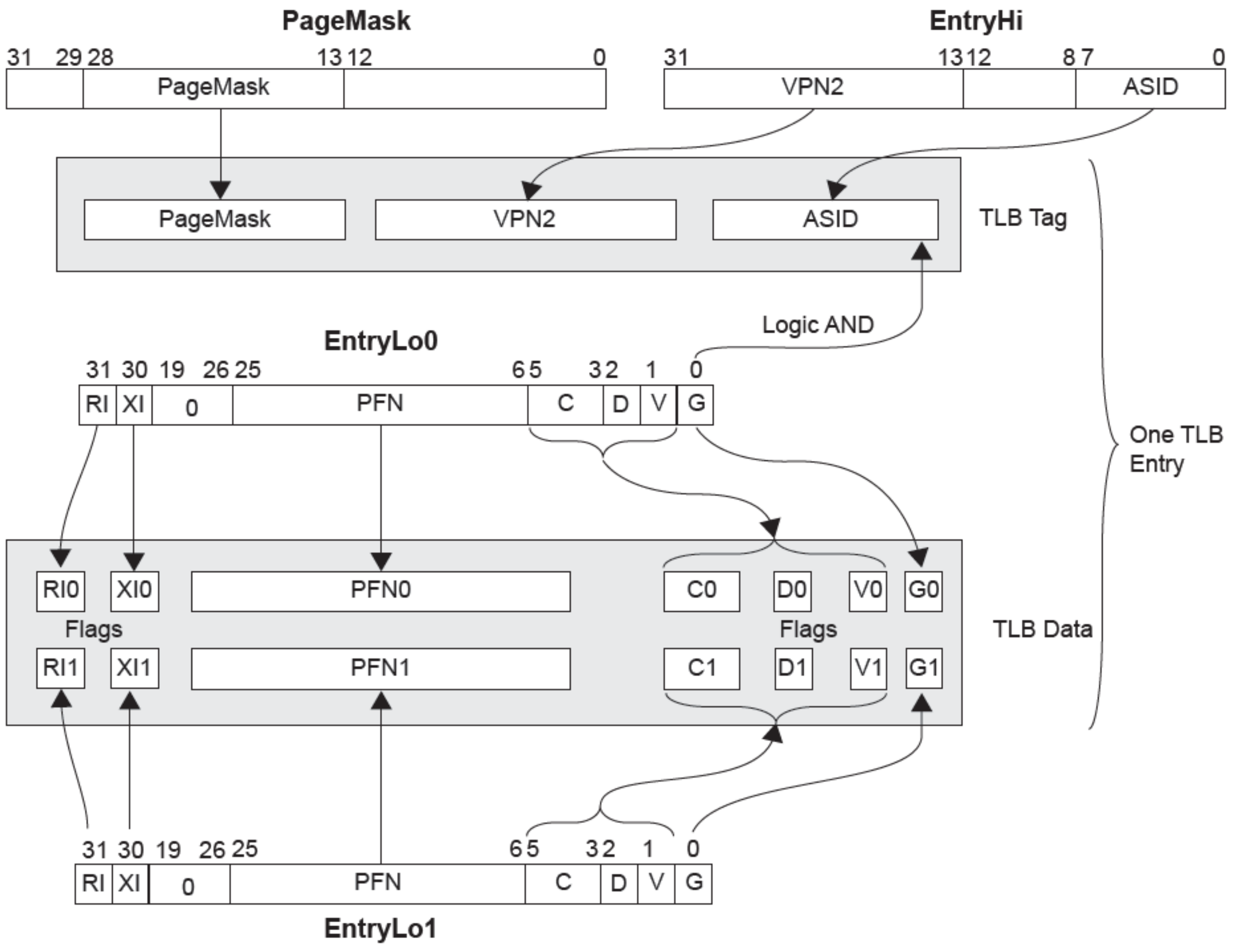
To fill an entry in the JTLB, software executes a TLBWI or TLBWR instruction. Prior to invoking one of these instructions, the CP0 registers listed above must be updated with the information to be written to the TLB entry:

- PageMask is set in the CP0 PageMask register.
- VPN2, and ASID are set in the CP0 EntryHi register.

- PFN0, RI0, XI0, C0, D0, V0, and G bits are set in the CP0 EntryLo0 register.
- PFN1, RI1, XI1, C1, D1, V1, and G bits are set in the CP0 EntryLo1 register.

These register fields and their relationship to a TLB entry is described in the following subsections.

Figure 3: Relationship Between CP0 Registers and TLB Entries



2.3.1 TLB Tag Entry

The tag portion of the TLB entry contains the fields necessary to match an incoming address against that entry. This section describes each field of the TLB tag entry shown in *Figure 3: Relationship Between CP0 Registers and TLB Entries* on page 20.

VPN2 Field

The virtual page number (VPN) contains the high bits of the program (virtual) address. The 'VPN2' designation indicates that this address is for a double-page-size virtual region which will map to a pair of physical pages. The VPN2 field is generated using the EntryHi register.

Note: On a TLB-related exception, the VPN2 field in the EntryHi register is automatically set to the virtual address that was being translated when the exception occurred. If the outcome of the exception handler is to find and install the translation to that address, the VPN2 field will already contain the correct value.

ASID Field

The address space identifier (ASID) 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). The ASID field is generated using the EntryHi register.

PageMask Field

The size of the tag can be configured using the ‘PageMask’ field. This field determines how many incoming address bits to match. For the TLB, the I7200 core allows page sizes of 4 Kbytes up to 256 Mbytes in multiples of four. The PageMask field is generated using the PageMask register.

In the PageMask field, a ‘1’ on a given bit means "don’t compare this address bit when matching this address". However, only a restricted range of PageMask values are legal. The values must start with "1"s filling the PageMask field from the low-order bits upward, two at a time. A list of valid 32-bit PageMask register values, the corresponding binary value of the PageMask[28:13] field, and the corresponding page size is shown in *Table 2: PageMask Value and Corresponding Page Size* on page 21. For the Page-Mask[28:13] field, note that the bits are set two at a time from the least significant bit (LSB) to the most significant bit (MSB).

Table 2: PageMask Value and Corresponding Page Size

32-bit PageMask Register Value	PageMask[28:13]	Page Size	Even/Odd Bank Select Bit
0x0000_0000	0x00_0000_0000_0000_00	4 KBytes	VAddr[12]
0x0000_6000	0x00_0000_0000_0000_11	16 KBytes	VAddr[14]
0x0001_E000	0x00_0000_0000_0011_11	64 KBytes	VAddr[16]
0x0007_E000	0x00_0000_0000_1111_11	256 KBytes	VAddr[18]
0x001F_E000	0x00_0000_1111_1111_11	1 MByte	VAddr[20]
0x007F_E000	0x00_0011_1111_1111_11	4 MBytes	VAddr[22]
0x01FF_E000	0x00_0011_1111_1111_11	16 MBytes	VAddr[24]
0x07FF_E000	0x00_1111_1111_1111_11	64 MBytes	VAddr[26]
0x1FFF_E000	0x11_1111_1111_1111_11	256 MBytes	VAddr[28]

Global (G) Bit

The ‘G’ (global) bit in the tag entry is a logical AND between the G bits of the EntryLo0and EntryLo1registers. When set, it causes addresses to match regardless of their ASID value, thus defining a part of the address space which will be shared by all applications. For example, Linux applications share some ‘kseg2’ space used for kernel extensions.

Note: Because the G bit in the TLB tag entry is a logical AND between two G bits, software must be sure to set EntryLo0$_G$ and EntryLo1$_G$ to the same value.

2.3.2 TLB Data Entry

The data portion describes each field of the TLB data entry shown in *Figure 3: Relationship Between CP0 Registers and TLB Entries* on page 20.

Read Inhibit (RI)

If this bit is set in a TLB entry, an attempt to read data on the virtual page causes a TLBRI exception, even if the V (Valid) bit is set.

Execute Inhibit (XI)

If this bit is set in a TLB entry, an attempt to fetch an instruction from the virtual page causes a TLBXI exception, even if the V (Valid) bit is set.

Page Frame Number (PFN)

The Page Frame Number (PFN) contains the high-order bits of the physical address. For a 4 KByte page size, the 20- bit PFN, together with the lower 12 bits of address that are not translated, make up the 32-bit physical address.

Flag Fields (C, D, V,)

These flag bits contain information about the translated address. All of these bits are generated by the EntryLo0 and EntryLo1 registers.

- **C Field:** This field contains the cacheability attributes for the corresponding TLB entry. It indicates how to cache data for this page. Pages can be marked cacheable, uncacheable, coherent, non-coherent, uncached accelerated, write- back, write-allocate, etc
- **D bit:** The "dirty" flag. Setting this bit 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. Software can use this bit to track pages that have been written to. When a page is first mapped, this bit should be cleared. It is set on the first write that causes an exception.
- **V bit:** The "valid" flag. Indicates that the TLB entry, and thus the virtual page mapping, are 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.
- **G bit:** Global bit. On a TLB write, the logical AND of the G bits from both EntryLo0 and EntryLo1 becomes the G bit in the TLB entry. If the TLB entry G bit is a one, ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both EntryLo0 and EntryLo1 reflect the state of the TLB G bit.

2.3.3 Address Translation Examples

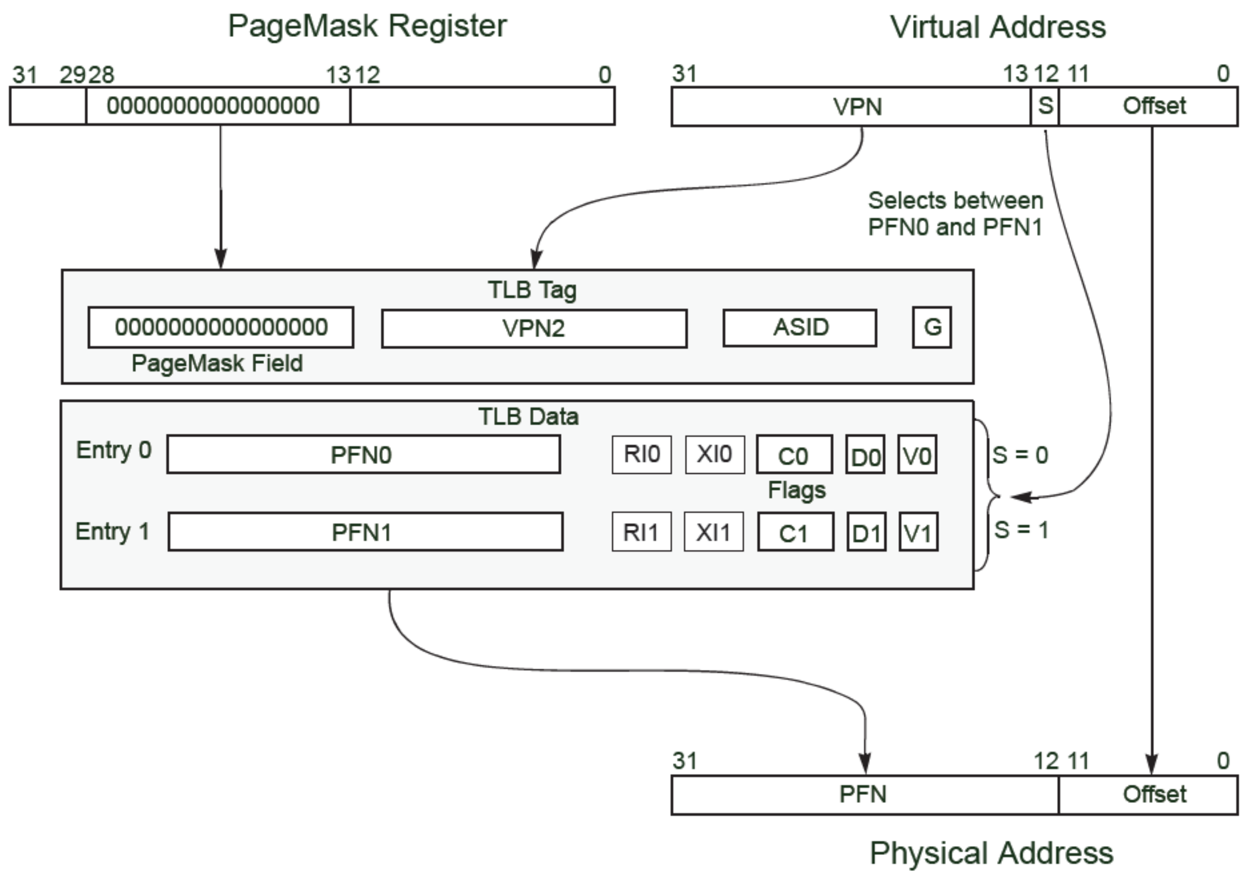
As shown in *Figure 3: Relationship Between CP0 Registers and TLB Entries* on page 20, there are two PFN values for each tag match. Which of them is used is determined by the lowest-order bit of the VPN field of the address. So in standard form (using 4 KByte pages) each entry translates an 8 KByte region of virtual address, but each 4 Kbyte page can be mapped onto any physical address (with any permission flag bits). This concept is described in the following subsections.

4 KByte Page Size Example

In a 4 KByte page size, 12 address bits are required to select an entry within the page. Therefore, 12 bits of the virtual address are used for the offset into the page. The upper 20 bits of the virtual address are used as a pointer to the page table. With a 4 KByte page size, this allows support for up to 1M page table entries.

The upper 20 bits of virtual address pass through the TLB to generate the corresponding physical address. The I7200 core implements a dual-entry JTLB scheme, where each TLB tag corresponds to two data entries. To select between these two entries, hardware reads the low-order bit of the VPN (first bit after the offset, shown as the S bit in the following figure). In a 4 KByte page example, this equates to bit 12.

Figure 4: Selecting Between PFN0 and PFN1 — 4 KByte Page Size



The PageMask field is derived from the PageMask register and is used to determine the page size for the application. Because the I7200 core supports JTLB page sizes in multiples of four (4 KByte, 16 KByte, 64 KByte, etc. up to 256 MByte), page masking is done in pairs. During translation, hardware checks the VPN against the contents of the PageMask field to determine the page size, and therefore, how many VPN bits to compare. Refer to *Table 2: PageMask Value and Corresponding Page Size* on page 21 for a list of valid PageMask values.

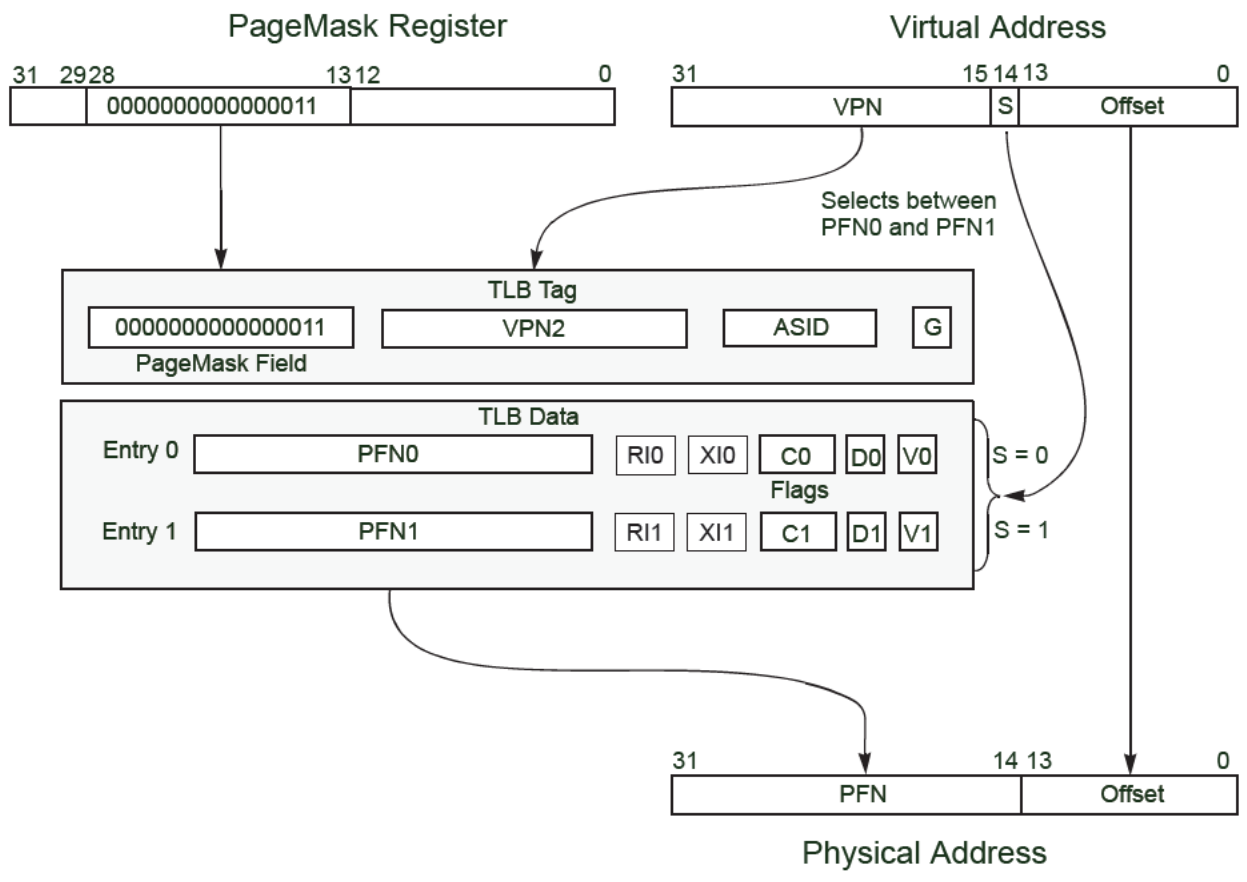
In this example, all of the PageMask field bits are 0, indicating a 4 KByte page size. For a 16 KByte page size, bits 12 and 13 of the PageMask field would be set. This concept is described in the next section.

16 KByte Page Size Example

In a 16 KByte page size, 14 address bits are required to select an entry within the page. Therefore, 14 bits of the virtual address are used for the offset into the page. The upper 18 bits of the virtual address are used as a pointer to the page table. With a 16 KByte page size, this allows support for up to 256K page table entries.

The upper 18 bits of virtual address pass through the TLB to generate the corresponding physical address. The I7200 core implements a dual-entry JTLB scheme, where each TLB tag corresponds to two data entries. To select between these two entries, hardware reads the low-order bit of the VPN (first bit after the offset, shown as the S bit in the following figure). In a 16 KByte page example, this equates to bit 14.

Figure 5: Selecting Between PFN0 and PFN1 — 16 KByte Page Size



The PageMask field is used to determine the page size for the application. During translation, hardware checks the VPN against the contents of the PageMask field to determine the page size, and therefore how many VPN bits to compare. In this example, the lower 2 bits of the PageMask field bits are 11, indicating a 16 KByte page size. Refer to *Table 2: PageMask Value and Corresponding Page Size* on page 21 for a list of valid PageMask values.

2.4 MMU Programming

The following subsections describe some of the programming options for the I7200 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 programmers 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.

2.4.1 Indexing the JTLB

In the I7200 core, the JTLB is 64 dual entries. This value is stored in the Index register (CP0 register 0, Select 0).

Table 3: Index Register Format Depending on TLB Size

31	30 … 6	5 … 0
P	0	Index

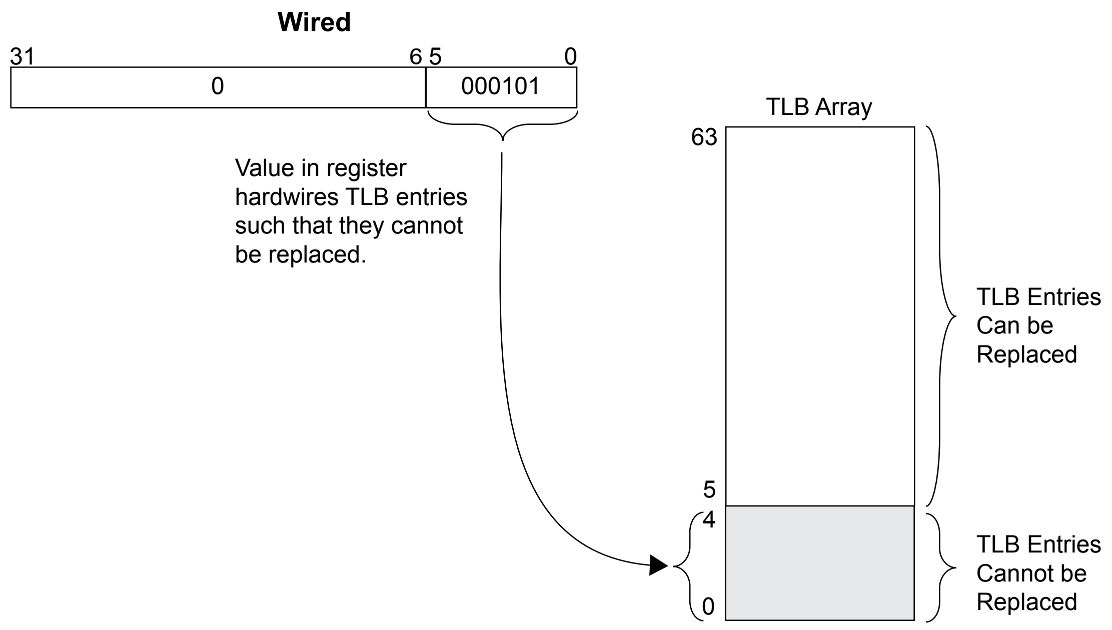
The Index register determines which TLB entry is accessed by a `TLBWI` instruction. This register is also used for the result of a `TLBP` instruction (used to determine whether a particular address was successfully translated by the CPU). Note that a `TLBP` instruction that fails to find a match for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

2.4.2 Hardwiring JTLB Entries

The I7200 core allows up to 63 entries of the JTLB to be hardwired such that they cannot be replaced. 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 JTLB. 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.

Wired entries in the JTLB must be contiguous and start from 0. For example, if the Wired field of this register contains a value of 5, this indicates that entries 4, 3, 2, 1, and 0 of the TLB are wired. The Wired register is reset to zero by a Reset exception. Writing to the Wired register may cause the Random register to change state. The following figure shows an example of hardwiring the lower 5 entries of the TLB. A value of 0x0 in the Wired register indicates that no entries are hardwired and that all entries are available for replacement.

Figure 6: Hardwiring Entries in the TLB


Wired
31
6 5
0
0
000101
Value in register hardwires TLB entries such that they cannot be replaced.
TLB Array
63
5
4
0
TLB Entries Can be Replaced
TLB Entries Cannot be Replaced

2.4.3 JTLB Random Replacement

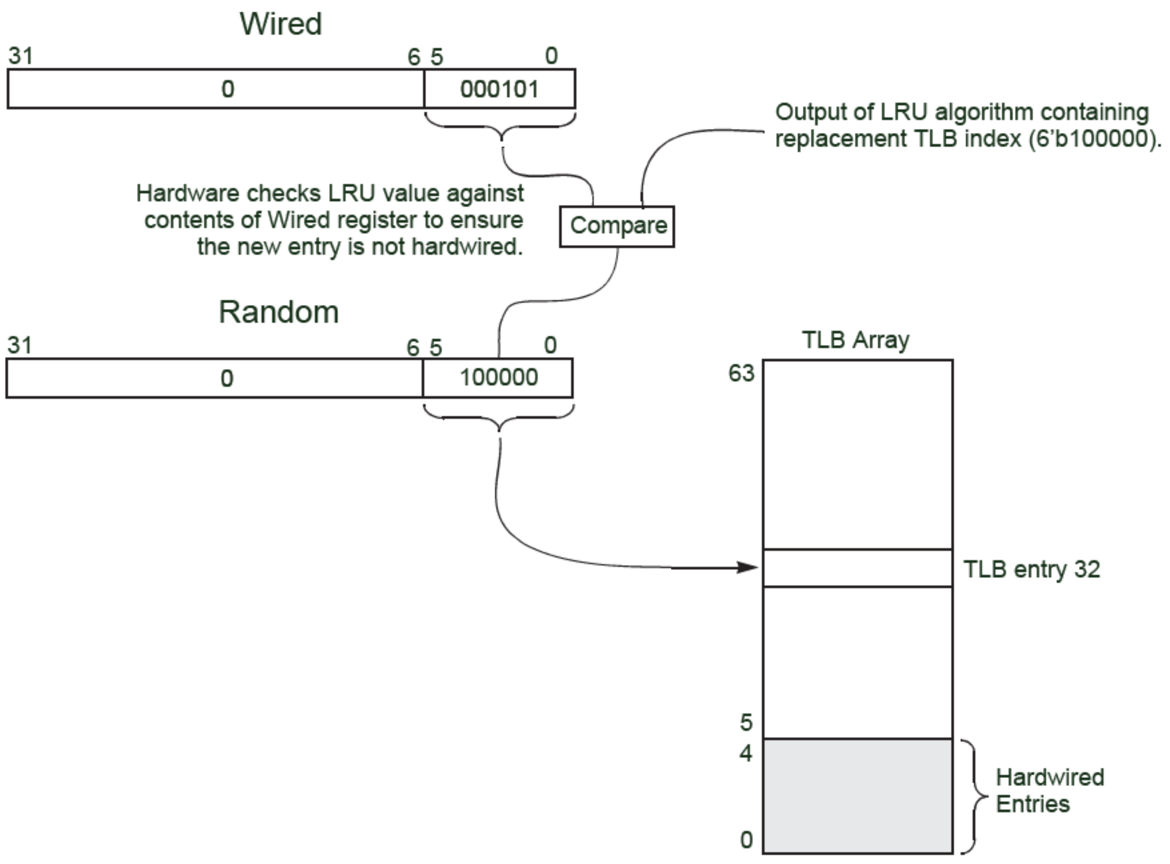
The I7200 core performs random replacement within the 64 dual-entry JTLB using the CP0 Random register (CP0 register 1, Select 0). This read-only register is used to index the TLB during a `TLBWR` instruction. It provides a quick way of replacing a JTLB entry at random.

The Random register employs a pseudo-random least-recently-used (LRU) algorithm, which ensures that no wired entries are selected. Only those LRU entries that are not in the Wired register are targeted for replacement. The contents of the Random register are modified after a JTLB write, or on a write to the Wired register.

The processor initializes the Random register to the reflect the maximum number of entries (63) on a Reset exception. The Random register is used only for JTLB accesses.

The following figure shows an example of a random replacement to entry 32 of the JTLB with the lower five entries of the JTLB hardwired.

Figure 7: Random Replacement of a JTLB Entry


Wired
31
6 5
0
0
000101
Output of LRU algorithm containing replacement TLB index (6'b100000).
Hardware checks LRU value against contents of Wired register to ensure the new entry is not hardwired.
Compare
Random
31
6 5
0
0
100000
TLB Array
63
TLB entry 32
5
4
0
Hardwired Entries

2.5 TLB Exception Handler

In the event that a TLB miss occurs in the JTLB, the I7200 core allows for the following types of TLB exceptions.

- Address error (AdEL or AdES)
- TLB Refill
- TLB (TLBL, TLBS)
- TLB Modified (TLBM)

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 is taken on any TLB miss regardless of the operating mode.

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.

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

Register Interface

The I7200 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.
- *ContextConfig* (CP0 register ??, Select 0): Defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written (*BadVPN2*), and how many bits of that virtual address will be extracted. In the *Context* register, bits above the selected *BadVPN2* field are read/write to software and serve as the *PTEBase* field. Bits below the selected *BadVPN2* field serve as the *PTEBaseLow* field.
- *BadVAddr* (CP0 register 8, Select 0): 32-bit 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 because they are not addressing errors.

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:

mfc0 k1, C0_CONTEXT    // Get Context register (CP0 register 4)
lw k0, 0(k1)           // Load EntryLo0 into k0
lw k1, 8(k1)           // Load EntryLo1 into k1
mtc0 k0, C0_ENTRYLO0   // Move k0 to CP0 EntryLo0 (CP0 register 2)
mtc0 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
```

Note: Some operating systems like Linux use a 3-level Page Table and do not use the *Context* 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.

2.6 TLB Duplicate Entries

The JTLB entries come up in a random state on power-up and must be initialized by hardware before use. Typically, bootstrap software initializes each entry in the TLB. Because the JTLB is a fully-associative array and entries are written by index, it is possible to load duplicate entries, where two or more entries match the same virtual address/ASID.

If duplicate entries are detected on a TLB write, no machine check is generated and the older entries are simply invalidated. The new entry gets written. When writing to the TLB, all entries of the JTLB are searched for duplicates.

2.7 Modes of Operation

The MMU's virtual-to-physical address translation is determined by the mode in which the processor is operating. The I7200 core operates in one of four modes:

- User mode
- Supervisor mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. Supervisor mode is an intermediate privilege level with access to an additional region of memory and is only supported with the TLB-based MMU. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and usually occurs within a software development tool.

Table 4: Selecting the Addresing Mode

Mode	Status			Debug	Description
	EXL	ERL	KSU	DM	
User	0	0	2'b2	0	User addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
Supervisor	0	0	2'b1	0	Supervisor addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
Kernel	x	x	2'b0	0	Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	x	1	x	0	Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	1	x	x	0	Kernel addressing mode. In this mode, a TLB miss goes to the general exception handler as opposed to the TLB Refill handler.
Debug	x	x	x	1	Debug mode.

3 Enhanced Virtual Address

Traditional MIPS virtual memory support divides up the virtual address space into fixed size segments, each with fixed attributes and access privileges. Such a scheme limits unmapped kernel access to 512 MBytes, the size of kseg0/kseg1. Furthermore, application sizes are growing beyond the 2 GB limit imposed by the useg user segment.

Programmable Memory Segmentation relaxes these limitations. The size of virtual address space segments can be programmed, as can their attributes and privilege access. With this ability to overlap access modes, kseg0 can now be extended up to 3.0 GB[1], leaving at least one 1.0 GB segment for mapped kernel accesses. This extended kseg0 (xkseg0) overlaps with useg, because segments in xkseg0 are programmed to support mapped user accesses and unmapped kernel accesses. Consequently, user space is equal to the size of xkseg0, which can be up to 3.0 GB.

To allow for efficient kernel access to user space, load and store instructions allow kernel-mapped access to useg. These instructions, along with Programmable Memory Segmentation, permit implementation of Enhanced Virtual Address or EVA, which allows for more efficient use of 32b address space.

Note: EVA is only supported when TLB is configured. When the core is configured with an MPU, EVA, segment-control registers, and physical-address relocation overlays are not supported. Refer to *MPU and Segment Control* on page 61 for the behaviors of these bits and registers when MPU is configured.

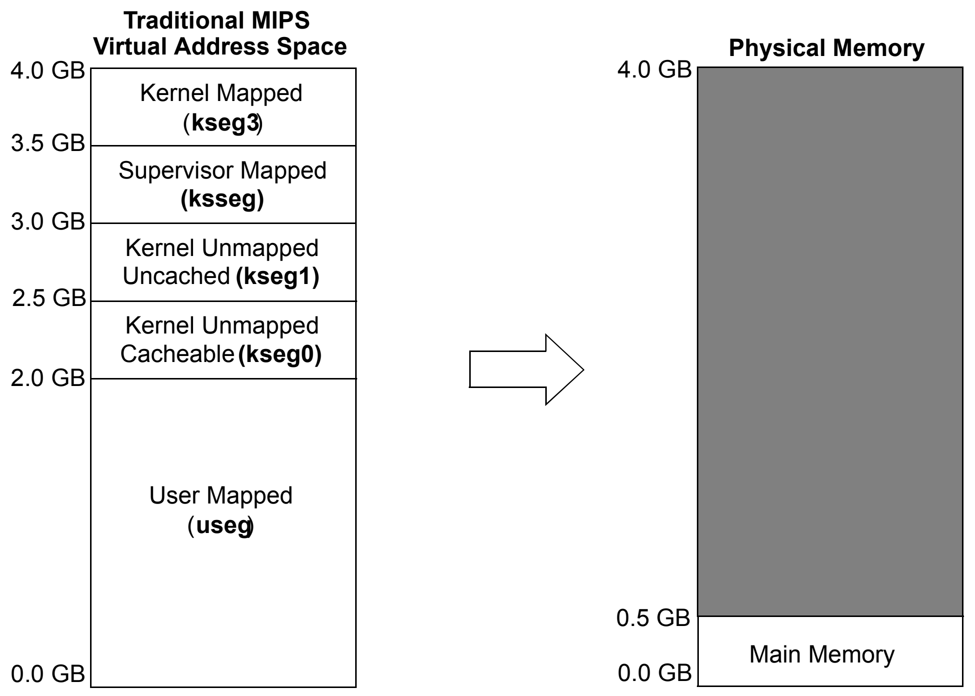
3.1 Virtual and Physical Address Maps

In previous generation MIPS32 processors, the address map was fixed. In this architecture, physical memory is limited by kseg0 to 0.5GB, the amount of kernel unmapped cached address space. This

[1] If necessary, xkseg0 can be extended to 3.5 GB, allowing 0.5 GB for Kernel mapped virtual address space (now kseg2).

memory must also be shared by the I/O and kernel, thus in reality less than 0.5GB is available to any user process.

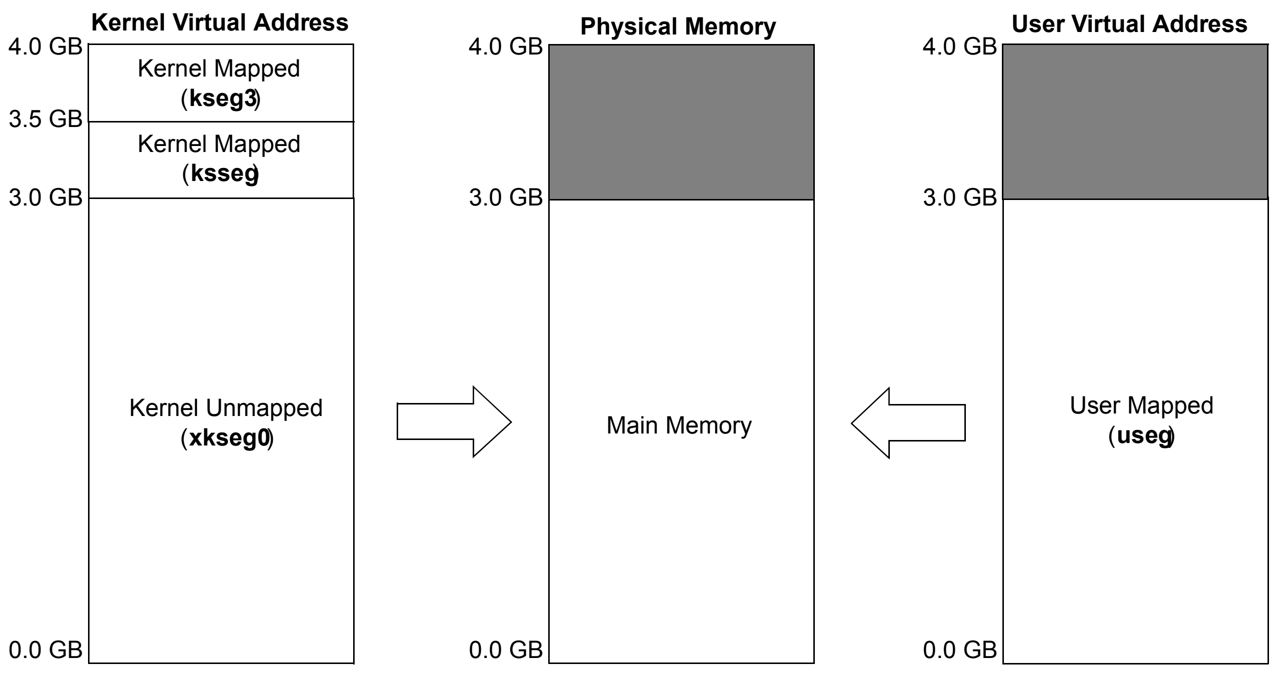
Figure 8: Traditional Virtual Address Mapping in Previous Generation MIPS32 Processors


Traditional MIPS Virtual Address Space
4.0 GB
Kernel Mapped (kseg3)
3.5 GB
Supervisor Mapped (ksseg)
3.0 GB
Kernel Unmapped Uncached (kseg1)
2.5 GB
Kernel Unmapped Cacheable (kseg0)
2.0 GB
User Mapped (useg)
0.0 GB
Physical Memory
4.0 GB
0.5 GB
Main Memory
0.0 GB

The following figure shows an example of how the traditional MIPS kernel virtual address space can be remapped using programmable memory segmentation to facilitate the EVA scheme. As a result of defining the larger kernel segment as xkseg0, the kernel has unmapped access to the lower 3GB of the virtual address space. The larger user segment could be defined because the address space is not statically partitioned. This allows for a total of 3.5GB of DRAM to be supported in the system.

Note: xkseg0 is equivalent to the previous kseg0 space in that it is a kernel unmapped, cacheable region.

Figure 9: Example of Remapping Kernel and User Virtual Address Space Using EVA



3.2 Initial EVA Configuration Parameters

During build time, you select the EVA parameters through the GUI. These selections are registered into the CM, which drives the core EVA pins with the appropriate value. For a listing of EVA related pins and their function, refer to *Boot Exception Vector Relocation in Kernel Mode* on page 38.

3.3 Programmable Segmentation Control

Programmable segmentation allows for the virtual address space segments to be programmed with different access modes and attributes. Control of the 4GB of virtual address space is divided into six segments that are controlled using three CP0 registers; SegCtl0 through SegCtl2. Each register has two 16-bit fields. Each field controls one of the six address segments as shown in in the following table.

Table 5: Programmable Segmentation Register Interface

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range
SegCtl2	Register 5 Select 4	CFG5	31:16	0.0 - 1.0 GB	0x0000_0000 - 0x3FFF_FFFF
		CFG4	15:0	1.0 - 2.0 GB	0x4000_0000 - 0x7FFF_FFFF
SegCtl1	Register 5 Select 3	CFG3	31:16	2.0 - 2.5 GB	0x8000_0000 - 0x9FFF_FFFF
		CFG2	15:0	2.5 - 3.0 GB	0xA000_0000 - 0xBFFF_FFFF

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range
SegCtl0	Register 5 Select 2	CFG1	31:16	3.0 - 3.5 GB	0xC000_0000 - 0xDFFF_FFFF
		CFG0	15:0	3.5 - 4.0 GB	0xE000_0000 - 0xFFFF_FFFF

Each 16-bit field listed in the above table contains information on the corresponding memory segment such as address range (for kernel unmapped segments), access mode, and cache coherency attributes. The following table describes the 16-bit configuration fields (CFG0 - CFG5) defined in the SegCtl0 - SegCtl2 registers.

Table 6: CFG (Segment Configuration) Field Descriptions

CFGn Fields		Description
Name	Bits	
PA	15:9, 31:25	Physical address bits 31:29 for segment, for use when unmapped. These bits are used when the virtual address space is configured as kernel unmapped to select the segment in memory to be accessed. For segments 0, 2, and 4, CFG[11:9] correspond to physical address bits 31:29. CFG[15:12] correspond to physical address bits 35:32 in a 36-bit addressing scheme and are reserved for future use. The state of CFG[15:12] are read/write and can be programmed, but these bits are not driven onto the address bus. For segments 1, 3, and 5, CFG[27:25] correspond to physical address bits 31:29. CFG[31:28] correspond to physical address bits 35:32 in a 36-bit addressing scheme and are reserved for future use. These bits are not used by the CFG4 and CFG5 spaces when these segments are programmed to be kernel mapped and the physical address is determined by the TLB. They are also not used for any of the user mapped (useg) region for the same reason.
Reserved	8:7, 24:23	Reserved.
AM	6:4, 22:20	Access control mode. For programmable segmentation, these bits are set as shown in Table 3.5. Bits 6:4 correspond to segments 0, 2, and 4. Bits 22:20 correspond to segments 1, 3, and 5.
EU	3, 19	Error condition behavior. Segment becomes unmapped and uncached when Status$_{ERL}$ = 1. Bit 3 corresponds to segments 0, 2, and 4. Bit 19 corresponds to segments 1, 3, and 5.
C	2:0, 18:16	Cache coherency attribute, for use when unmapped. For programmable segmentation, these bits are set as shown in Table 3.5. Bits 2:0 correspond to segments 0, 2, and 4. Bits 18:16 correspond to segments 1, 3, and 5.

Cache Coherency Attribute Control and the Segmentation Control Registers

The CP0 memory segmentation control registers (SegCtl0 - SegCtl2) are used to control the size and function of the various memory map segments in the I7200 core. Each segmentation control register contains its own cache coherency attribute field to allow for maximum flexibility when assigning cacheability attributes to the memory. However, because existing code may not be aware of the existence of the SegCtl0 - SegCtl2 registers, the I7200 core allows a mechanism for the cache coherency attributes (CCA) of kseg0 to be set either by the Config.K0 field or by the CFG3_C field (bits 18:16) of the SegCtl1 register. This allows existing code to configure virtual memory for a legacy setting.

To control where the cache coherency attributes for the memory are taken from, the CP0 Config5 register uses the Config5.K bit. If the Config5.K bit is cleared, the cache coherency attributes for kseg0 are derived from the 3-bit Config.K0 field of the CP0 Config register. This can be done when booting the I7200 core using existing code. If the Config5.K bit is set, the cache coherency attributes are derived from the 3-bit SegCtlx.CFGy_C field of the segmentation control registers (where x indicates the segmentation control register number 0 - 2, and y indicates memory segments 0 - 5). When configured for EVA, each of the six memory segments can be indivudually defined with its own cache coherency attributes.

The initial programming of Config5.K bit is determined by the state of EVAReset bit [31] in the Core-Local Reset Exception Extended Base Register at reset.

Functions of the Config5.K Bit

The Config5.K bit effects the cache coherency attributes, the boot exception vector overlay mechanism, and the location of the exception vector. When the Config5.K bit is cleared, the following events occur:

1. The 3-bit Config.K0 field is used to set the cache coherency attributes for the kseg0 region (0x8000_0000 - 0x9FFF_FFFF).
2. Hardware creates two boot overlay segments, one for kseg0 and one for kseg1 (corresponding to SEGCTL1.CFG2 and SEGCTL1.CFG3).
3. Hardware ignores the state of bits 31:30 of the EBase register as well as the Core Local Reset Exception Base Register [31:30] bits. Instead, hardware forces these bits to a value of 2’b10, causing the vectors to reside in kseg0/kseg1 space.

When the Config5.K bit is set, the following events occur:

1. The 3-bit Config.K0 field is ignored and the cache coherency attributes are derived from the CFGn_C fields of the various segmentation control registers (SegCtl0 - SegCtl2).
2. Hardware creates one boot overlay segment that can reside anywhere in virtual address space.
3. The exception vectors are not forced to reside in kseg0/kseg1. Rather, bits 31:30 of the EBase register, as well as the Core Local Reset Exception Base Register bits [31:30] are used to place the exception vectors anywhere within virtual address space.

Setting the Memory Addressing Scheme — EVAReset and CONFIG5.K

The EVAReset bit determines the addressing scheme and whether the device boots up in the legacy setting or the EVA setting. The legacy setting is defined as having the traditional MIPS virtual memory map used in previous generation processors. The EVA setting places the device in the enhanced virtual address configuration, where the initial size and function of each segment in the virtual memory map is determined from the segmentation control registers (SegCtl0 - SegCtl2).

If the EVAReset bit is set at reset, the I7200 core comes up in the legacy configuration and hardware takes the following actions:

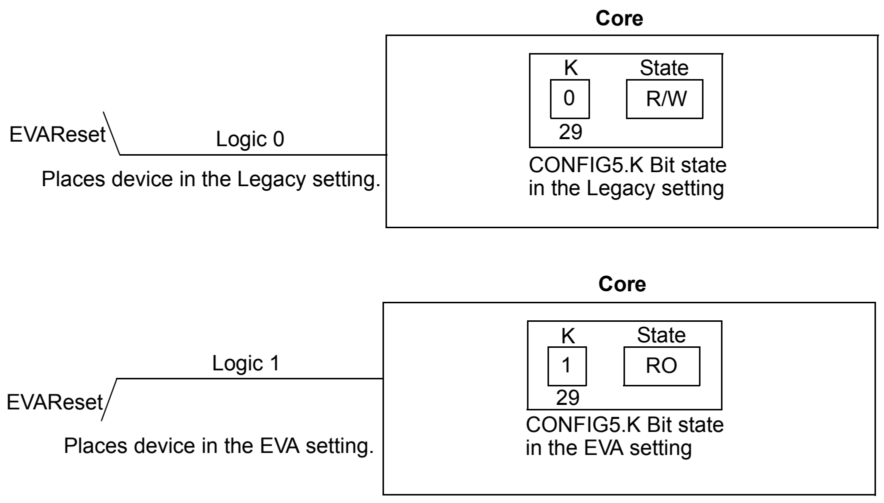
- The CONFIG5.K bit becomes read-write and is programmed by hardware to a value of 0 to indicate the legacy configuration. In this case, the cache coherency attributes for the kseg0 segment are derived from the Config.K0 field as described in the previous subsection. In addition to selecting the location of the cache coherency attributes, the CONFIG5.K bit also causes hardware to generate two boot exception overlay segments, one for kseg0 and one for kseg1.
- Hardware programs the CP0 memory segmentation registers (SegCtl0 - SegCtl2) for the legacy setting. An example of this programming is shown in Table 3.11. Note that these registers are new in the I7200 core and are not used by legacy software. However, they are used by hardware during normal operation, so their default values should not be changed.

If the EVAReset bit is set at reset, the I7200 core comes up in the EVA configuration (default is xkseg0 space = 3 GB) and hardware takes the following actions:

- The CONFIG5.K bit becomes read-only and is forced to a value of 1 to indicate the EVA configuration. In this case, the CONFIG.K0 field is ignored and is no longer used to determine the kseg0 cache coherency attributes (CCA). Rather, the values in bits 2:0 (segments 0, 2, and 4) and bits 18:16 (segments 1, 3, and 5) of the SegCtl0 - SegCtl2 registers are used to define the CCA for each memory segment as shown in Table 3.3. In this case, hardware generates only one BEV overlay segment.
- Hardware sets the CP0 memory segmentation registers (SegCtl0 - SegCtl2) for the EVA configuration.

These two options are illustrated in the following figure.

Figure 10: Relationship Between EVAReset and CONFIG5.K at Reset


Core
K
State
0
R/W
29
EVAReset
Logic 0
Places device in the Legacy setting.
CONFIG5.K Bit state in the Legacy setting
Core
K
State
1
RO
29
Logic 1
EVAReset
Places device in the EVA setting.
CONFIG5.K Bit state in the EVA setting

Enhanced Virtual Address Detection and Support

As described above, the SegCtl0 - SegCtl2 registers are used to control the various memory segments. In addition to these registers, two other configuration registers are also used in EVA.

The EVA bit in the Config5 register (Config5.EVA) is used to detect support for the enhanced virtual address scheme. This read-only bit is always 1 to indicate support for EVA.

In addition to the EVA bit, the SC bit in the Config3 register (Config3SC) is used by hardware to detect the presence of the SegCtl0 - SegCtl2 registers. This read-only bit is always 1 in the I7200 core to indicate the presence of these registers. Note that both of these features must be present to configure the virtual address space for EVA.

Setting the Access Control Mode

In addition to setting the Config5.EVA and Config3.SC bits, each memory segment must be set to the programmable segmentation mode. Bits 6:4 (segments 0, 2, and 4) and bits 22:20 (segments 1, 3, and 5) of the SegCtl0 through SegCtl2 registers define the access control mode.

To set the programmable segmentation registers to mimic the traditional MIPS32 virtual address mapping, the AM and C subfields of each 16-bit CFG field of the SegCtl0 - SegCtl2 registers should be programmed as shown in the following table.

Table 7: CFG (Segment Configuration) Field Descriptions

SegCtl Register	CFGn	CFGn Subfields		Segment Size (GB)	Location in Virtual Memory (GB)	Description
		AM	C			
0	0 (bits 15:0)	MK (bits 6:4 = 0x1)	0x3 (bits 2:0)	0.5	3.5 - 4.0	Mapped kernel region.
0	1 (bits 31:16)	MSK (bits 22:20 = 0x2)	0x3 (bits 18:16)	0.5	3.0 - 3.5	Mapped kernel, supervisor region.
1	2 (bits 15:0)	UK (bits 6:4 = 0x0)	0x2 (bits 2:0)	0.5	2.5 - 3.0	Kernel unmapped, uncached region.
1	3 (bits 31:16)	UK (bits 22:20 = 0x0)	0x3 (bits 18:16)	0.5	2.0 - 2.5	Kernel unmapped, cached region.
2	4 (bits 15:0)	MUSK (bits 6:4 = 0x3)	0x3 (bits 2:0)	1.0	1.0 - 2.0	User, supervisor, and kernel mapped region.
2	5 (bits 31:16)	MUSK (bits 22:20 = 0x3)	0x3 (bits 18:16)	1.0	0.0 - 1.0	User, supervisor, and kernel mapped region.

To set the programmable segmentation registers to implement EVA with a 3.0 GB xkseg0 space, the AM and C subfields of each CFG field of the SegCtl0 - SegCtl2 registers should be programmed as shown in the following table.

Table 8: Setting the Access Control Mode for the EVA Configuration

SegCtl Register	CFGn	CFGn Subfields		Segment Size (GB)	Location in Virtual Memory (GB)	Description
		AM	C			
0	0 (bits 15:0)	MK (bits 6:4 = 0x1)	0x3 (bits 2:0)	0.5	3.5 - 4.0	Mapped kernel region.
0	1 (bits 31:16)	MK[2] (bits 22:20 = 0x1)	0x3 (bits 18:16)	0.5	3.0 - 3.5	Mapped kernel region.
1	2 (bits 15:0)	MUSUK (bits 6:4 = 0x4)	0x2 (bits 2:0)	0.5	2.5 - 3.0	Mapped user/ supervisor, unmapped kernel region.
1	3 (bits 31:16)	MUSUK (bits 22:20 = 0x4)	0x3 (bits 18:16)	0.5	2.0 - 2.5	Mapped user/ supervisor, unmapped kernel region.

[2] This segment can also be mapped to MSK (bits 22:20 = 0x2) if supervisor mode is supported.

SegCtl Register	CFGn	CFGn Subfields		Segment Size (GB)	Location in Virtual Memory (GB)	Description
		AM	C			
2	4 (bits 15:0)	MUSUK (bits 6:4 = 0x4)	0x3 (bits 2:0)	1.0	1.0 - 2.0	Mapped user/ supervisor, unmapped kernel region.
2	5 (bits 31:16)	MUSUK (bits 22:20 = 0x4)	0x3 (bits 18:16)	1.0	0.0 - 1.0	Mapped user/ supervisor, unmapped kernel region.

MUSUK is an acronym for Mapped User/Supervisor, Unmapped Kernel. This mode sets the kernel unmapped virtual address space to xkseg0.

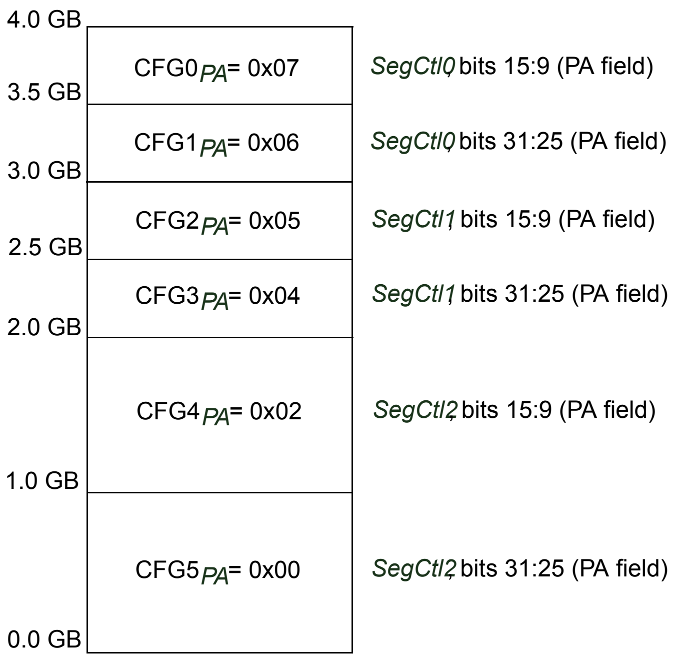
Defining the Physical Address Range for Each Memory Segment

As shown in *Programmable Segmentation Control* on page 31, each of the six 16-bit CFGn fields of the SegCtl0 through SegCtl2 fields controls a specific portion of the physical address range. Bits 11:9 (segments 0, 2, and 4) and bits 27:25 (segments 1, 3, and 5) of the SegCtl0 through SegCtl2 registers represent the state of physical address bits 31:29 and defines the starting address of each segment. These bits control the six segments of the physical address.

Note: Bits 31:28 and bits 15:12 are also part of the physical address field, but they are not used in the I7200 core and are reserved for future use by devices that implement a 36-bit address.

The following figure shows an example of how each segment of the physical address can be mapped to the SegCtl0 through SegCtl2 registers.

Figure 11: Mapping of SegCtl 0 - 2 Registers to Physical Address Space


4.0 GB
3.5 GB
3.0 GB
2.5 GB
2.0 GB
1.0 GB
0.0 GB
CFG0 PA = 0x07
CFG1 PA = 0x06
CFG2 PA = 0x05
CFG3 PA = 0x04
CFG4 PA = 0x02
CFG5 PA = 0x00
SegCtl0 bits 15:9 (PA field)
SegCtl0 bits 31:25 (PA field)
SegCtl1 bits 15:9 (PA field)
SegCtl1 bits 31:25 (PA field)
SegCtl2 bits 15:9 (PA field)
SegCtl2 bits 31:25 (PA field)

For example, to program the xkseg0 region to a size of 3.0 GB, the PA field of each register would be programmed as follows:

Table 9: Programmable Segmentation Register Interface

Register	CFGn Field	Bits	PA Field	Memory Segment	Virtual Address Range
SegCtl0[3]	CFG0	15:9	0x07	kseg2	0xE000_0000 - 0xFFFF_FFFF
	CFG1	31:25	0x06		0xC000_0000 - 0xDFFF_FFFF
SegCtl1	CFG2	15:9	0x05		0xA000_0000 - 0xBFFF_FFFF
	CFG3	31:25	0x04		0x8000_0000 - 0x9FFF_FFFF
SegCtl2	CFG4	15:9	0x02	xkseg0	0x4000_0000 - 0x7FFF_FFFF
	CFG5	31:25	0x00		0x0000_0000 - 0x3FFF_FFFF

Enhanced Virtual Address (EVA) Instructions

By default, an implementation that supports EVA requires a number of new load/store instructions that are used when the enhanced virtual address scheme is enabled. These kernel-mode user load/store instructions allow the kernel mapped access to user address space as if it were in user mode.

For example, the kernel can copy data from user address space to kernel physical address space by using such instructions with user virtual addresses. Kernel system-calls from user space can be conveniently changed by replacing normal load/store instructions with these instructions. Switching modes (kernel to user) is an alternative but this is an issue if the same virtual address is being simultaneously used by the kernel. Further, there is a performance penalty in context-switching.

The opcode for these instructions is embedded into bits 2:0 of the instruction, known as the Type field. Note that some fields can have the same encoding depending whether the operation is a load or a store. The load/store designation is determined by the AIU L/S field, or bits 5:3 of the instruction.

Table 10: Load/Store Instructions in Programmable Memory Segmentation Mode

Instruction Mnemonic	Instruction Name	Description
LBE	Load Byte Kernel	Load byte (as if user from) kernel extended virtual addressing load from user virtual memory while operating in kernel mode.
LBUE	Load Byte Unsigned Kernel	Load byte unsigned (as if user from) kernel.
LHE	Load Halfword Kernel	Load halfword (as if user from) kernel.
LHUE	Load Halfword Unsigned Kernel	Load halfword unsigned (as if user from) kernel.
LWE	Load Word Kernel	Load word (as if user from) kernel.
SBE	Store Byte Kernel	Store byte (as if user from) kernel extended virtual addressing load from user virtual memory while operating in kernel mode.
SHE	Store Halfword Kernel	Store halfword (as if user from) kernel.

[3] In the 3GB xkseg0 example, the PA portion of the CFG0 and CFG1 fields are not used because they are associated with kernel mapped address spaces. In this case the PA fields are not required since the physical address is determined by the TLB. In the maximum configuration, xkseg0 can be extended to 3.5 GB. In this case, the CFG1 field of the SegCtl0 register would become part of the xkseg0 segment and the PA subfield would be used.

Instruction Mnemonic	Instruction Name	Description
SWE	Store Word Kernel	Store word (as if user from) kernel.

3.4 Boot Exception Vector Relocation in Kernel Mode

Historically in MIPS processors, the boot exception vector (BEV) has always been at the same location in both virtual and physical memory, being mapped from a virtual address of 0xBFC0_0000 to a physical address of 0x1FC0_0000.

With the advent of memory segmentation, the BEV vector may not always map to a physical address of 0x1FC0_0000. This can cause a scenario where the boot exception vector resides at two different physical addresses depending on the memory mode. To address this issue, the I7200 core implements a boot exception vector overlay scheme that allows the BEV to be mapped to a single location in physical memory, regardless of the memory mode.

This section describes how to define the BEV overlay segment and the BEV relocation process for both the legacy setting and the Enhanced Virtual Address (EVA) setting, which is one element of the I7200 memory segmentation scheme.

Note: Boot exception vector relocation is performed only in Kernel mode.

3.4.1 Boot Configurations

In kernel mode, the core can be powered up in legacy or EVA address setting.

- **Legacy setting:** The legacy setting is the traditional boot mode followed by all MIPS processor prior to interAptiv, where the boot exception vector (BEV) is located at 0xBFC0_0000 in virtual address space, and maps to 0x1FC0_0000 in physical address space.
- **EVA setting:** In the EVA setting, the boot exception vector can be located anywhere in virtual address space and mapped to anywhere in physical address space.

3.4.2 Registers and Fields Used to Support Boot Exception Vector Relocation

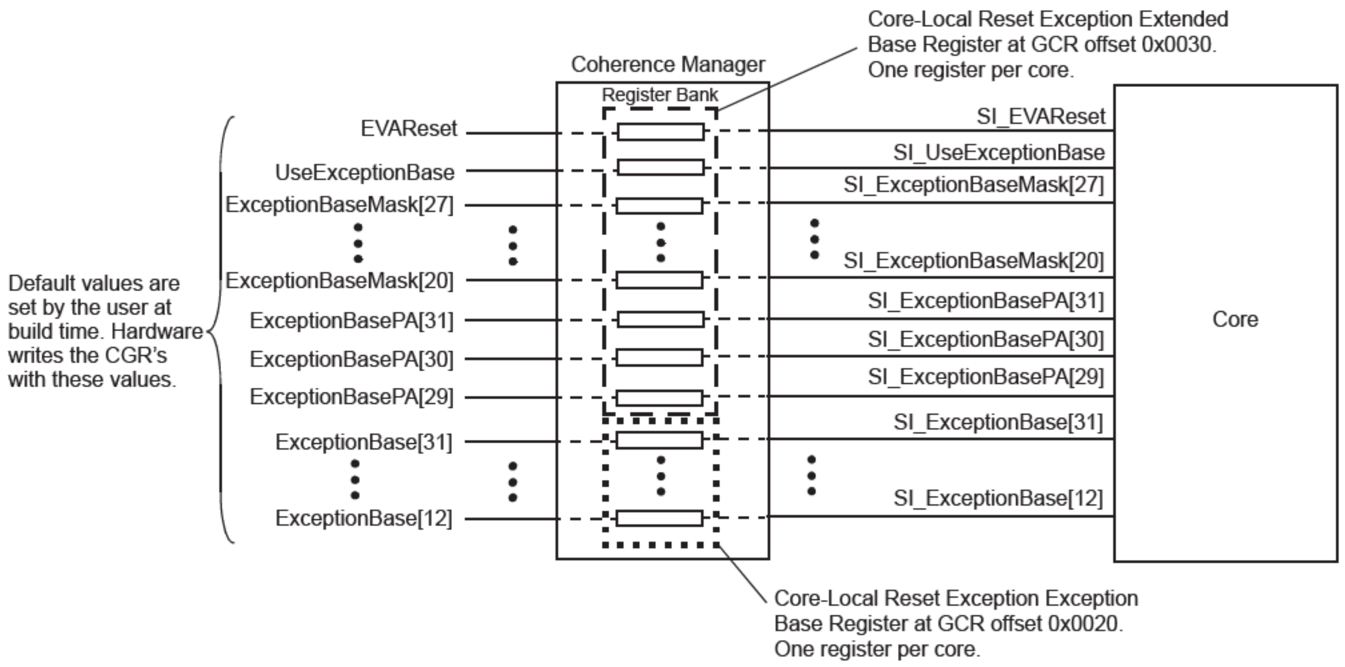
To facilitate the BEV overlay scheme, a number of pins were added to the I7200 core that allow the user to select the boot overlay parameters at build time. The initial state of the default values selected by the user at build time are registered inside the Coherence Manager (CM) block using two Global Configuration Registers (GCR)

There are two GCR registers used per core: the Core Local Reset Exception Base Register and the Core-Local Reset Exception Extended Base Register. Each core has its own pair of GCR registers and its own set of BEV related pins. This allows each core to be programmed in a different manner and independently from one another.

The CM drives these values to the I7200 cores at reset. Note that the two CGR registers are loaded only on a cold boot and are programmed with the values selected by the user at build time. Each of these pins is described in the following subsections.

The following figure shows the boot exception vector pins for a single I7200 core.Each additional core would have an identical set of CM registers and set of BEV related pins shown in the figure.

Figure 12: Registered Boot Exception Vector Relocation Pins — One Core



There is one pair of GCR registers for each core. This allows each I7200 core to be powered up in a different memory mode and independently from one another.

The boot exception vector relocation fields are described in the following table.

Table 11: I7200 Core-Local Reset Exception Extended Base Register bits

Field	Field Size (Bits)	CM GCR Register Mapping	Description
EVAReset	1	Bit 31 of the Core-Local Reset Exception Extended Base Register (offset = 0x0030)	If this bit is set at reset, the I7200 core comes up in the EVA configuration. In this case the CONFIG5.K bit becomes read-only with a fixed value of 1 to indicate EVA as the addressing scheme. In addition, the SegCtl0 - SegCtl2 registers are configured with values that correspond to the EVA mapping. If this bit is not set at reset, the I7200 core comes up in the legacy setting. In this case the CONFIG5.K bit becomes read-write with an initial value of 0 to indicate legacy mode. This bit is modified by software when switching from legacy mode to EVA mode. This bit is used in both the legacy and EVA settings. There is one EVAReset bit per core.

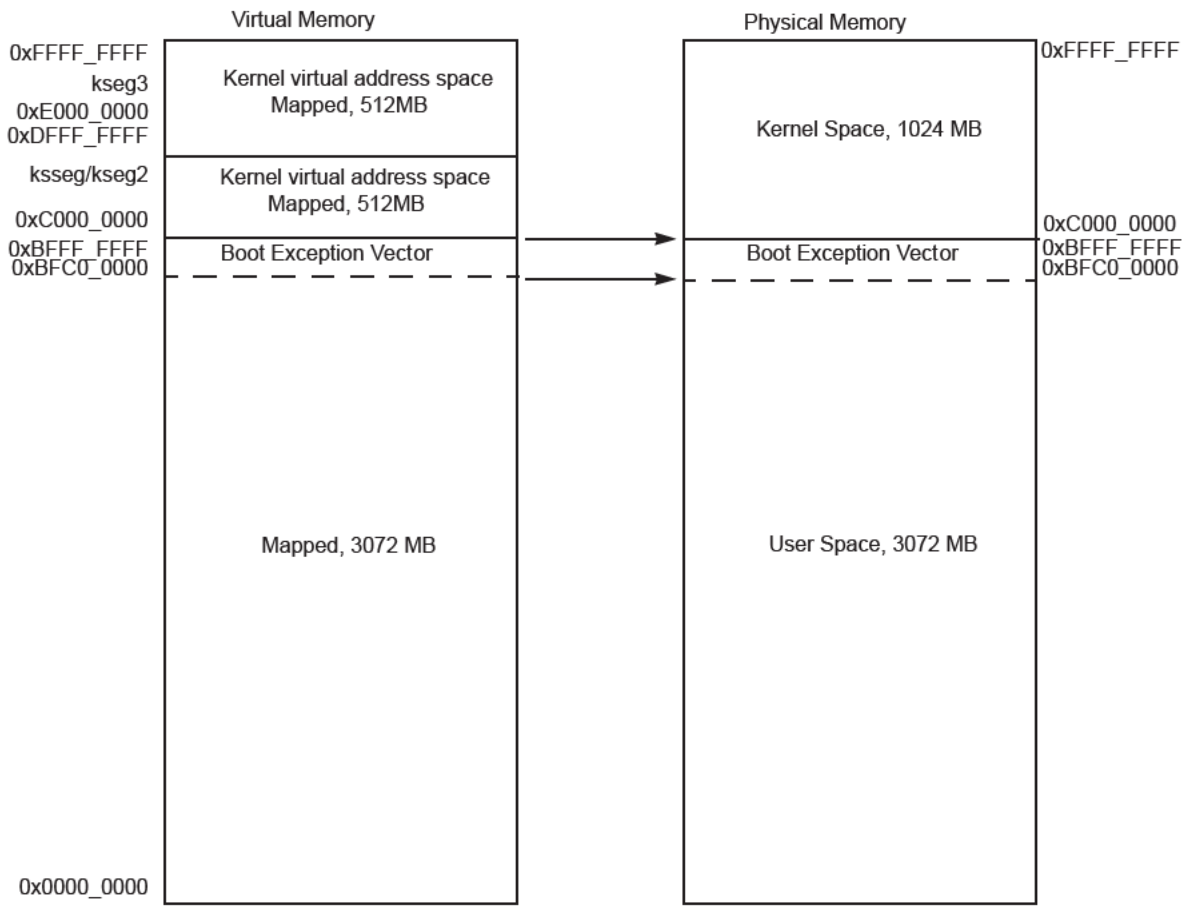
Field	Field Size (Bits)	CM GCR Register Mapping	Description
LegacyUseExceptionBase	1	Bit 30 of the Core-Local Reset Exception Extended Base Register (offset = 0x0030)	In the legacy configuration, if the LegacyUseExceptionBase bit is not set, then the BEV location defaults to 0xBFC0_0000. If the LegacyUseExceptionBase bit is set, address bits Core Local Reset Exception Base Register [31:30] are forced to a value of 2'b10 to force the BEV location into the KSEG0/KSEG1 space. This bit is only used in the legacy configuration. There is one LegacyUseExceptionBase bit per core.
BEVExceptionBaseMask[27:20]	8	Bits 27:20 of the Core-Local Reset Exception Extended Base Register (offset = 0x0030)	Used to determine the size of the boot exception vector overlay region from 1 MB to 256 MB in powers of two. These bits are used in both the legacy and EVA configurations. There is one set of BEVExceptionBaseMask bits per core.
BEVExceptionBasePA[31:29]	3	Bits 3:1 of the Core-Local Reset Exception Extended Base Register (offset = 0x0030)	Upper physical address bits. The size of the overlay region defined by BEVExceptionBasePA[27:20] is remapped to a location in physical address space pointed to by the BEVExceptionBasePA[31:29] bits. This allows the overlay region to be placed into one of the 512 MB segments in physical memory. These pins are used in both the legacy and EVA configurations. There is one set of BEVExceptionBasePA bits per core.
BEVExceptionBase[31:12]	20	Bits 31:12 of the Core-Local Reset Exception Base Register (offset = 0x0020)	The Core Local Reset Exception Base Register [31:12] bits define the boot address in virtual address space which is used to define the overlay region. These pins, along with the Core-Local Reset Exception Extended Base Register [27:20] bits, determine the size and location of the BEV region within virtual address space. Note that the CONFIG5.K CP0 register bit is used to determine which pins of the Core Local Reset Exception Base Register bits [31:12] address are used to calculate the overlay. These pins are used in the EVA setting and can also be used in the legacy setting. There is one set of Core Local Reset Exception Base Register bits per core.

3.4.3 Example Mapping of the Boot Exception Vector in the EVA Configuration

In the I7200 core, physical memory sizes can be up to 3.5 GB. In the legacy configuration, the BEV is remapped from 0xBFC0_0000 in virtual memory to 0x1FC0_0000 in physical memory. However, in the

EVA configuration, if the physical memory size is set to 3.0 GB, no remapping of the BEV is required. In this case, the BEV is remapped from 0xBFC0_0000 in virtual memory to 0xBFC0_0000 in physical memory. This relocation of the BEV between the two memory modes creates a conflict if software switches from the legacy configuration to the EVA configuration because the BEV will be mapped to two different locations in physical memory. An example of mapping of the BEV in the EVA configuration with a 3.0 GB xkseg0 space is shown in in the following figure.

Figure 13: Example of Mapping the Boot Exception Vector in the EVA Configuration



3.4.4 Defining the Boot Exception Vector Overlay Region

To solve the problem of having the boot exception vector residing at different physical address locations based on the memory mode, the interAptiv core defines a boot exception vector overlay region that can be programmed from 1 MB to 256 MB (1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, or 256 MB) in powers of two using an 8-bit virtual address mask as described below.

This space is then mapped to a predetermined location in physical memory, regardless of the memory configuration (legacy or EVA).

The BEV overlay not only allows the location of the boot exception vector to be mapped to an area common to both the Legacy and EVA configuraitons, but also eliminates the non-contiguous chunk of memory that was created by having the boot exception vector at the top of the first 512M of physical memory space as in previous generation processors.

To set the boot exception overlay region, the following steps are taken:

1. Determine whether the core boots up in Legacy mode or EVA mode.
2. Determine which virtual address bits will be used to calculate the boot exception vector base address.

3. Determine the size and location of the overlay region in virtual address space.
4. Determine the location of the overlay region in physical address space.

Setting the Type of Memory Addressing Mode

The EVAReset bits in the Core-Local Reset Exception Extended Base Register, along with the CONFIG5.K bit, determines whether the addressing scheme is set to legacy or EVA at reset.

Using theLegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register and CONFIG5.K to Determine How to Calculate the BEV Base Address

The LegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register and the CONFIG5.K register bit are also used to determine the addressing scheme and how the location of the boot exception vector will be calculated. The relationship between the LegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register and the CONFIG5.K register is shown in Table 3.9. This table shows how to use the various address fields (Core-Local Reset Exception Extended Base Register [27:20] and Core Local Reset Exception Base Register [31:12]).

Table 12: LegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register and CONFIG5.K Encoding

CONFIG5.K Bit	LegacyUse ExceptionBase bits in the Core Local Reset Exception Extended Base Register	Condition	Action
0	0	Legacy Configuration Core Local Reset Exception Base Register [31:12] bits are not used.	Use default BEV location of 0xBFC0_0000.
0	1	Legacy Configuration Use only Core Local Reset Exception Base Register [29:12] for the BEV base location. Bits 31:30 are forced to a value of 2'b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.	The BEV location is determined as follows: Core Local Reset Exception Base Register [31:12] = 2'b10, Core Local Reset Exception Base Register [29:12] bits, 12'b0 Bits 31:30 are forced to a value of 2'b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.
1	Don't Care	EVA Configuration Use Core Local Reset Exception Base Register [31:12] bits.	The Core Local Reset Exception Base Register [31:12] bits are used directly to derive the BEV location. The LegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register are ignored.

Determining the Size and Location of the Overlay Region in Virtual Address Space

The starting location of the overlay region in virtual address space is defined using either the Core Local Reset Exception Base Register [31:12] bits, or the Core Local Reset Exception Base Register [29:12] bits depending on the state of the LegacyUseExceptionBase bits in the Core-Local Reset Exception Extended Base Register and CONFIG5.K bit. The size of the overlay region where the BEV is located is determined using the Core-Local Reset Exception Extended Base Register [27:20] bits as shown in the following table.

Table 13: Encoding of BEVExceptionBaseMask [27:20]

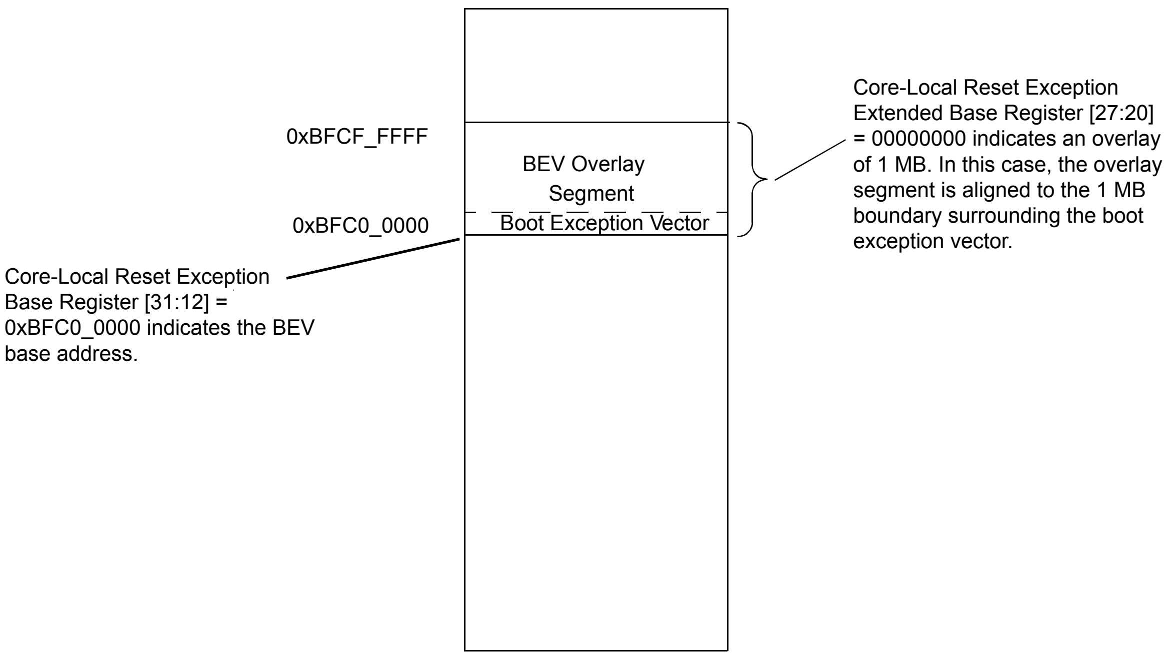
Core-Local Reset Exception Extended Base Register [27:20]	Segment Size (MB)
00000000	1
00000001	2
00000011	4
00000111	8
00001111	16
00011111	32
00111111	64
01111111	128
11111111	256

Consider the following example:

- The location of the BEV is at 0xBFC0_0000
- The overlay size is 1 MB (BEVExceptionBaseMask [27:20] = 00000000)
- The CONFIG5.K CP0 register bit is set

In this case the BEV segment would be located in virtual address space as shown in the following figure.

Figure 14: Size and Location of Overlay Region in Virtual Address Space — 1 MB Example


0xBFCF_FFFF
0xBFC0_0000
BEV Overlay Segment
Boot Exception Vector
Core-Local Reset Exception Extended Base Register [27:20] = 00000000 indicates an overlay of 1 MB. In this case, the overlay segment is aligned to the 1 MB boundary surrounding the boot exception vector.
Core-Local Reset Exception Base Register [31:12] = 0xBFC0_0000 indicates the BEV base address.

The start of the BEV is aligned on a 1 MB boundary and therefore is at the start of the 1MB address space. This may not always be the case depending on the size of the overlay region.

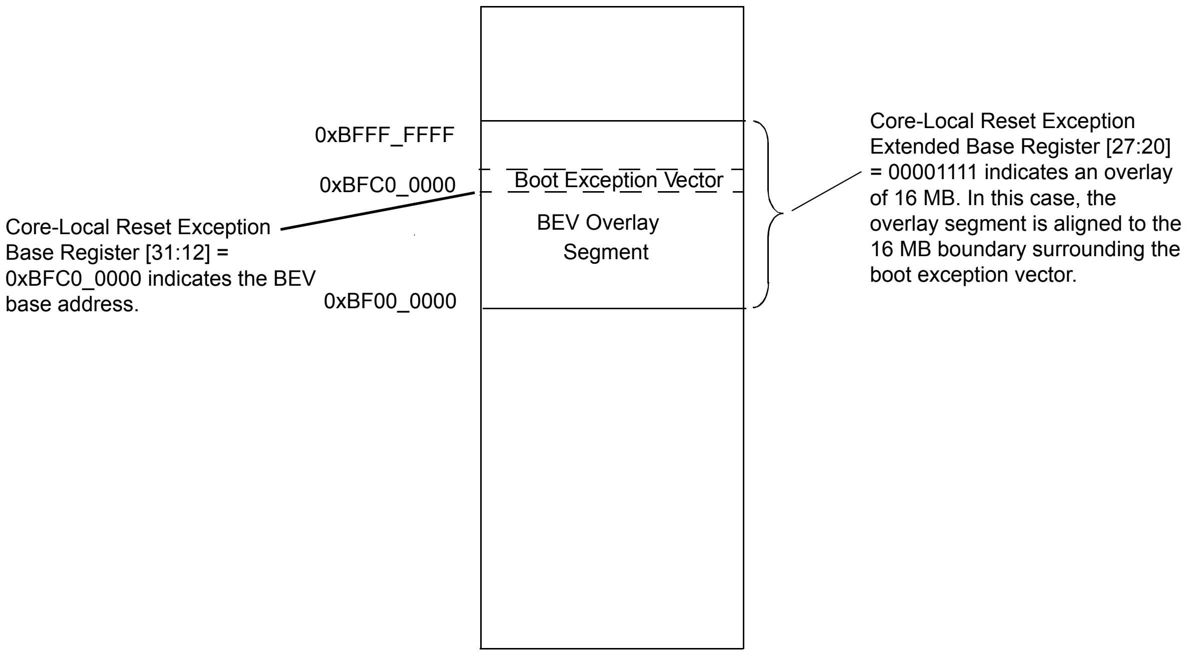
In another example:

- The location of the BEV is at 0xBFC0_0000
- The overlay size is 16 MB (BEVExceptionBaseMask [27:20] = 00001111)

- The CONFIG5.K CP0 register bit is set

In this case the BEV segment would be located in virtual address spac as shown in the following figure.

Figure 15: Size and Location of Overlay Region in Virtual Address Space — 16 MB Example

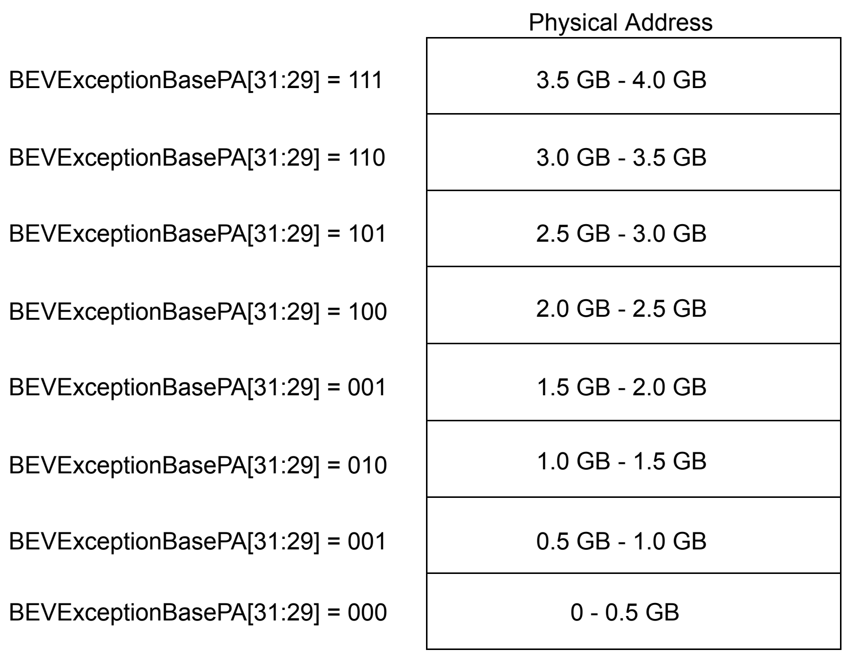

0xBFFF_FFFF
0xBFC0_0000
Core-Local Reset Exception Base Register [31:12] = 0xBFC0_0000 indicates the BEV base address.
0xBF00_0000
Boot Exception Vector
BEV Overlay Segment
Core-Local Reset Exception Extended Base Register [27:20] = 00001111 indicates an overlay of 16 MB. In this case, the overlay segment is aligned to the 16 MB boundary surrounding the boot exception vector.

Determining the Location of the Overlay Region in Physical Memory

As described in the previous subsections, the Core Local Reset Exception Base Register [31:12] and Core-Local Reset Exception Extended Base Register [27:20] fields are used to determine the size and location of the overlay within virtual address space. This segment of virtual memory is then remapped to physical memory at a location determined by theBEVExceptionBasePA Field in the Core-Local Reset Exception Extended Base register. These bits divide the physical address space into a number of 512

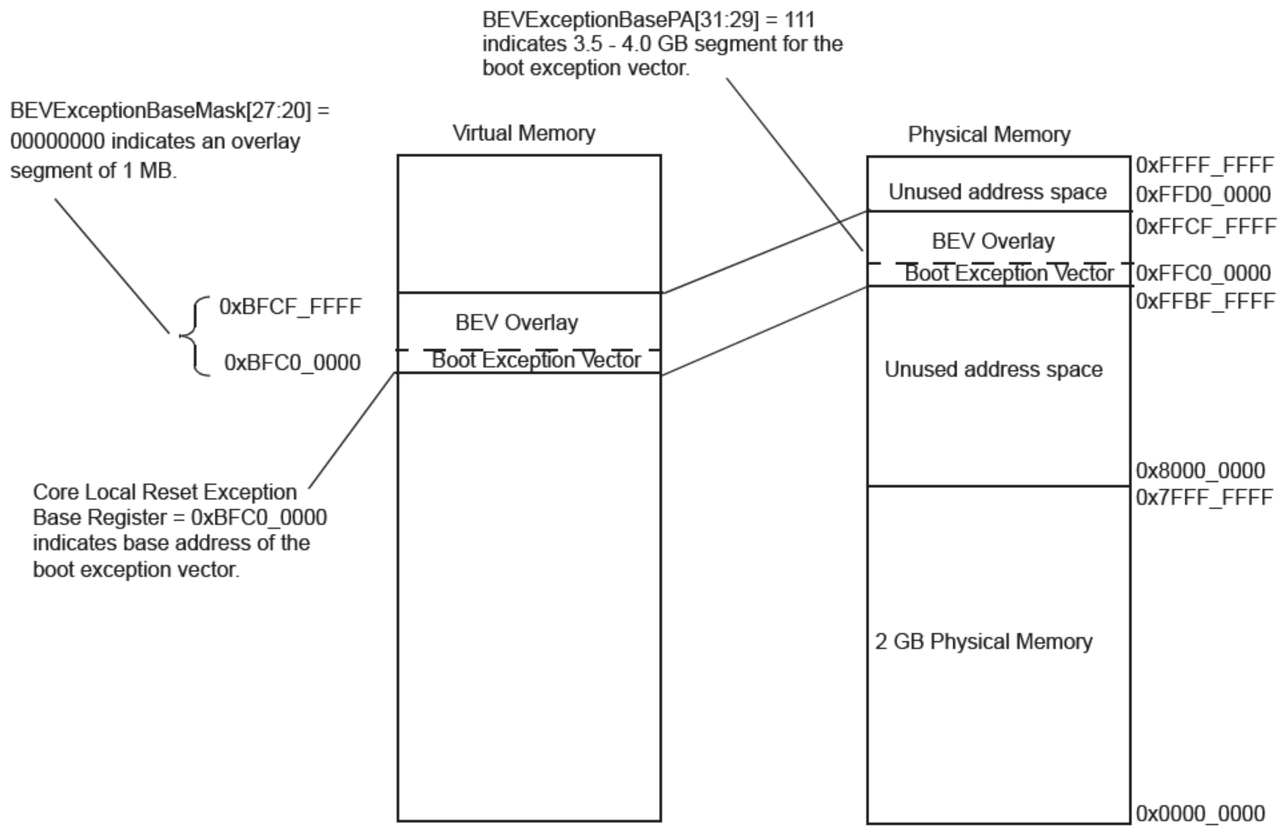
MByte segments. For example, in a 4 GB physical address space, the space can be divided into eight 512 MByte segments. This concept is shown in the following figure.

Figure 16: Physical Address Space Segmentation Using BEVExceptionBasePA [31:29]


Physical Address
BEVExceptionBasePA[31:29] = 111
3.5 GB - 4.0 GB
BEVExceptionBasePA[31:29] = 110
3.0 GB - 3.5 GB
BEVExceptionBasePA[31:29] = 101
2.5 GB - 3.0 GB
BEVExceptionBasePA[31:29] = 100
2.0 GB - 2.5 GB
BEVExceptionBasePA[31:29] = 001
1.5 GB - 2.0 GB
BEVExceptionBasePA[31:29] = 010
1.0 GB - 1.5 GB
BEVExceptionBasePA[31:29] = 001
0.5 GB - 1.0 GB
BEVExceptionBasePA[31:29] = 000
0 - 0.5 GB

For example, assume that the boot exception vector resides at a virtual address of 0xBFC0_0000, and the size of the segment is 1 MB as determined by the BEVExceptionBaseMask[27:20] bits. The physical memory size (amount of DRAM) is 2 GB, and the boot ROM that contains the BEV has been relocated to the top 512 MB of the 4 GB physical address space using the BEVExceptionBasePA[31:29] bits, which selects the segment from 3.5 GB to 4.0 GB. The remapping of the boot exception vector would be as shown in the following figure.

Figure 17: Example of Relocating the Boot Exception Vector


BEVExceptionBasePA[31:29] = 111 indicates 3.5 - 4.0 GB segment for the boot exception vector.
BEVExceptionBaseMask[27:20] = 00000000 indicates an overlay segment of 1 MB.
Virtual Memory
Physical Memory
0xBFCF_FFFF
0xBFC0_0000
BEV Overlay
Boot Exception Vector
Core Local Reset Exception Base Register = 0xBFC0_0000 indicates base address of the boot exception vector.
Unused address space
BEV Overlay
Boot Exception Vector
Unused address space
2 GB Physical Memory
0xFFFF_FFFF
0xFFD0_0000
0xFFCF_FFFF
0xFFC0_0000
0xFFBF_FFFF
0x8000_0000
0x7FFF_FFFF
0x0000_0000

In this example, because the overlay region has been defined, the boot exception vector would be relocated to the same address space, regardless of whether the addressing scheme is legacy or EVA. In addition, the memory space that contains the BEV no longer need be shared with actual physical memory in the first 512 MB of memory space as with previous MIPS processors, thereby allowing for all of the memory to be contiguous and available to the user.

4 Memory Protection Unit

The I7200 core can optionally be built with a Memory Protection Unit (MPU) instead of a TLB. The main difference between the MPU and TLB is that the MPU does not perform virtual to physical address translation. Instead it directly maps program addresses to physical addresses..

The MPU breaks the 4 GB address range into Default Segments. Each Default Segment can be configured for 4 attributes:

- Cache access
- Read
- Write
- Execute protection

For finer control, the Default Segment attributes can be redefined for parts of the Default Segment address space using Regions, which can overlay parts of a segment's address space with different values for the 4 attributes.

The MPU does not place restrictions on access permissions. All modes (kernel, user, and debug) are subject to the same permissions (No separate kernel or user mode address spaces). In addition, all threads are subject to the same permissions.

Note: Supervisor mode is not supported when an MPU is implemented.

Even though there are no restrictions on access permissions, debug memory spaces are still restricted to use only in debug mode. Additionally, if MPU segmentation is disabled, user mode access to legacy kernel segments still causes an address error. While the core is in debug mode, access to DRSEG will not cause MPU exceptions.

The MPU segmentation can be enabled or disabled by selecting the option at build time. Disabling the MPU forces a fixed mapping translation (FMT) address translation mechanism using CCA values from the CP0 Config register. This can be used to leverage legacy software and infrastructure. Refer to the K23 and KU fields of the CP0 Config register (Register 16, Select 0) for more information.

4.1 Default Segment Control Overview

The I7200 core MPU divides the 4 GB memory space into a series of sixteen 256 MB segments. These fixed size segments can efficiently set the default attributes for each memory address.

The default segments and the register fields that control them are shown in *Table 14: Default Memory Segments* on page 48. Each segment contains the following programmable elements:

- **Cache Coherency Attributes (CCA):** Indicates the coherency attributes for the entire 256 MB segment.
- **Read-Inhibit (RI):** Determines if data reads are allowed. If the RI bit of the corresponding segment control register is set and a data read is attempted anywhere within that 256 MB segment, an exception occurs.
- **Write-Inhibit (WI):** Determines if data writes are allowed. If the WI bit of the corresponding segment control register is set and a data write is attempted anywhere within that 256 MB segment, an exception occurs.
- **Execute-Inhibit (XI):** Determines if code fetches are allowed. If the XI bit of the corresponding segment control register is set and a code fetch is attempted anywhere within that 256 MB segment, an exception occurs.

For example, a 256 MB segment could be allocated for I/O devices and should allow data reads and writes but not fetches. In this case, the RI and WI bits for that segment would be programmed with a value of 0, allowing read or write operations to occur. The XI bit would be programmed with a value of 1, indicating that code fetches are not allowed from that memory segment.

Conversely, for a segment configured for only code fetch accesses, the RI and WI bits would be programmed with a value of 1, indicating data reads and writes are not allowed, and the XI bit would be programmed with a value of 0, indicating that code fetches are allowed.

Typically, a combination of default segment mapping and region mapping will be used. The default attributes from the segment mapping will be overridden if the address is programmed as part of a region. Any memory space not specifically defined as a region using the Region Control register uses the default segment mapping.

Each segment is set to the default cache coherency attributes (CCA) and associated permissions. More details will follow later in this chapter.

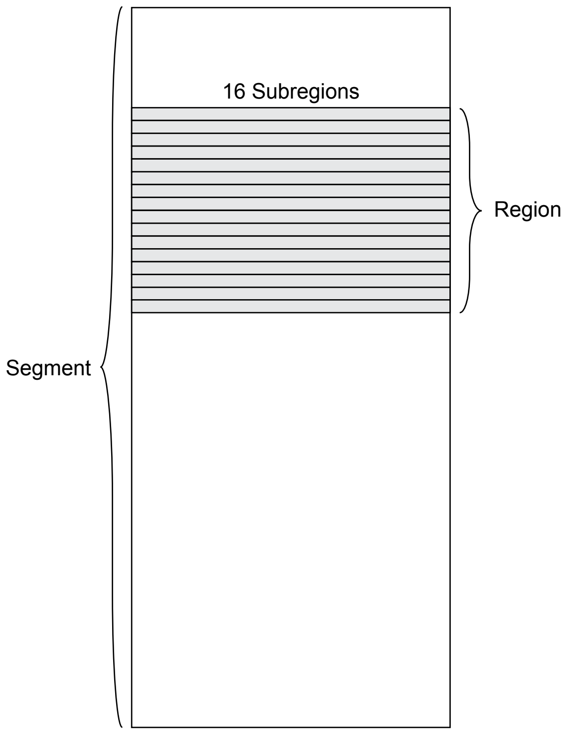
Table 14: Default Memory Segments

Segment #	Starting Address
15	0xF0000000
14	0xE0000000
13	0xD0000000
12	0xC0000000
11	0xB0000000
10	0xA0000000
9	0x90000000
8	0x80000000
7	0x70000000
6	0x60000000
5	0x50000000
4	0x40000000
3	0x30000000
2	0x20000000
1	0x10000000
0	0x00000000

4.2 Regions Overview

The MPU can use regions to control finer grain subregions of the address space by overlaying the memory segment configuration that was discussed in the previously. There can be up to 32 regions. Each region is divided into 16 equal sized subregions that share attributes.

Figure 18: Regions



4.3 CDMM Configuration Registers

MIPS processors that implement an MPU do so with a memory-mapped section called the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping I/O device configuration registers within a MIPS processor. The CDMM helps aggregate various device mappings into one area, preventing fragmentation of the memory address space. It also enables the use of access control and memory address translation mechanisms for these device registers. The CDMM occupies a maximum of 32 KB in the physical address map.

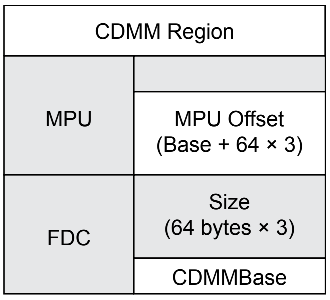
The base address of the CDMM region is set in the CP0 CDMMBase register.

The 32 KB CDMM region is divided into smaller 64-byte naturally aligned Device Register Blocks (DRBs). Each block has access control and status registers (ACSRs), followed by I/O device registers. For implementations that have multiple VPEs, the I/O devices and their ACSRs are instantiated once per VPE, but the CDMMBase register is shared between the VPEs.

The memory mapped registers located within the CDMM region must be accessed only using uncached memory transactions.

The Fast Debug Channel (FDC) also uses the CDMM region. The FDC is a UART-like communication device that uses the JTAG probe pins to move data to the external world.

Figure 19: MPU and FDC



Programming the CP0 CDMMBase Register

The following table shows layout of the CP0 CDMMBase register.

Table 15: CP0 CDMMBase Register

Register Fields		CDMMBase Register	Reset State
Name	Bits	(CP0 Register 15, Select 2)	
CDMM_UPPER_ADDR	27:11	Bits 31:15 of the base physical address of the memory mapped registers.	undefined
EN	10	Enables the CDMM region	0
CDMMSize	8:0	number of 64-byte Device Register Blocks instantiated in the core Note 0 = 1 64-byte Block	Preset

The base address of the CDMM memory region must be set before MPU registers can be programmed. The following example code that shows how to program the of the CDMM base address into the CDMM_UPPER_ADDR field and set the enable bit for the CDMM at the same time.

```
#define CDMM_P_BASE_ADDR    0x1fc10000         // physical address of the CDMM Register
#define CDMM_Enable         (1<<10)            // Enable bit 10
li   a0, (CDMM_P_BASE_ADDR>>4)|CDMM_Enable)    // load CDMM register base address + enable bit
mtc0  a0, C0_CDMMBASE
ehb                                            // ehb remove cp0 hazard
```

4.4 MPU Configuration Registers

There are two MPU general access and configuration registers.

MPU Access Control and Status Register

The MPU Access Control and Status Register, located at offset 0 of the MPU section of the CDMM region, provides information on the MPU device type, how many blocks the MPU uses in the CDMM region, and the MPU revision number. It also has settings for user mode access to all MPU registers.

- Setting the Uw bit allows writing the MPU registers in user mode.
- Setting the Ur bit allows reading the MPU registers in user mode.

Table 16: MPU Access Control and Status Register

Register Fields		MPU_ACSR Register (CDMM address + MPU Offset + 0x0000)	Read/Write	Reset State
Name	Bits			
DevType	31:24	MPU Device Type	R	0x01
DevSize	21:16	Number of 64-byte blocks allocated to this device	R	Preset
DevRev	15:12	MPU Revision Number	R	0x00
Uw	3	Enable User mode write access	RW	0
Ur	2	Enable User mode read access	RW	0

MPU Configuration Register

The MPU Configuration Register contains the bit to enable the MPU and information on MPU exceptions when they happen.

Table 17: MPU Configuration Register

Register Fields		MPU_Config Register (CDMM address + MPU Offset + 0x0008)	Read/Write	Reset State
Name	Bits			
En	31	Enable MPU segment control. Use FMT mode if not enabled.	R	Preset
ExcR	19	Last Protection exception caused by a load to a RI address.	R	0
ExcW	18	Last Protection exception caused by a store to a WI address.	R	0
ExcX	17	Last Protection exception caused by a fetch to a XI address.	R	0
Exc_Reg_Match	16	Last Protection exception hit in region.	R	0
Exc_Reg_Num	12:8	Region number for last Protection exception. Undefined if no region match.	R	0
NumRegions	4:0	Number of protected regions implemented. (Actual = NumRegion +1)	R	Preset

4.5 Segments

The attributes for the 16 default segments are programmed into the MPU segment control registers. Each segment control register is divided into control 4 segments so there are effectively 4 segment control registers. The following table shows the offset into the MPU section for each register, the segments in each register, and the bit fields for each segment.

Table 18: MPU_Segment Control Register (CDMM address + MPU Offset + 0x0010 + (SegmentCTL Register Number * 0x4)

SegmentCTL Register Number and Offset		29	28	27	26 24	21	20	19	18 16	13	12	11	10 8	5	4	2	2 0
		RI	WI	XI	CCA	RI	WI	XI	CCA	RI	WI	XI	CCA	RI	WI	XI	CCA
0	0x10	Segment 3				Segment 2				Segment 1				Segment 0			
1	0x14	Segment 7				Segment 6				Segment 5				Segment 4			
2	0x18	Segment 11				Segment 10				Segment 9				Segment 8			
3	0x1C	Segment 15				Segment 14				Segment 13				Segment 12			

The following table shows the attributes configured by the segment control register.

Table 19: Attributes Configured by Segment Control Register

Field	Description	Read/Write	Reset State
RI	Read inhibit. Trigger MPUL exception on data read.	R/W	Preset
WI	Write inhibit. Trigger MPUS exception on data write.	R/W	Preset
XI	Execute inhibit. Trigger MPUL exception on instruction fetch.	R/W	Preset
CCA	Cache Coherency Attributes.	R/W	Preset

The following example code shows how to set CCA for segment 0 to coherent and uncached for segments 1 - 3.

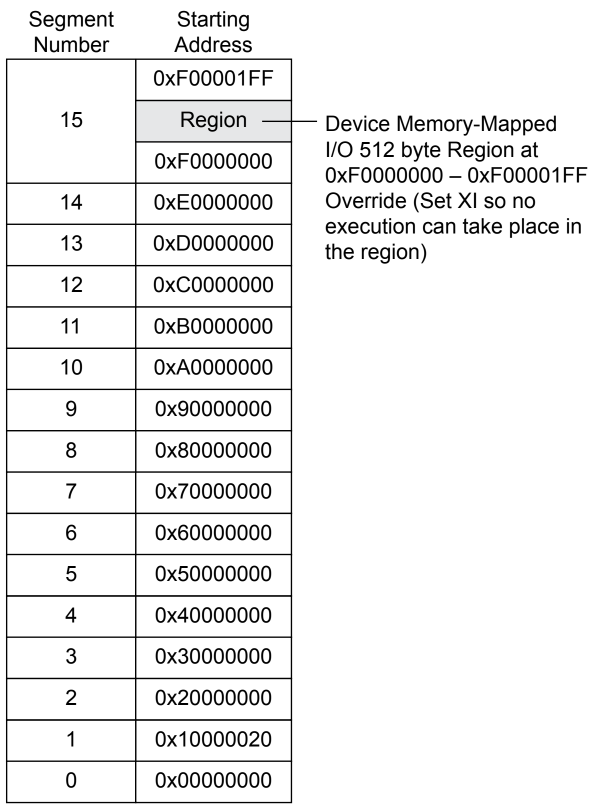
```
#define CDMM_P_BASE_ADDR     0x1fc10000           // physical address of the CDMM Register
#define MPU_CDMM_OFFSET      (64*3)
#define MPU_SegmentControl0  0x10

//  change the MPU CCA setting to cacheable
li  a0, CDMM_P_BASE_ADDR                          // a0 address for CDMM
// Segment 0 set to Cacheable
// Segments 1, 2 and 3 uncached
li  a1, (0x2)<<24 | (0x2)<<16 | (0x2)<<8 | (0x5)
sw  a1, MPU_CDMM_OFFSET+MPU_SegmentControl0(a0)
sync
```

4.6 Regions

Regions are used to override the segment defaults. For example, an I/O device may need a small memory area for configuration and status registers. This area would only need to be accessible using loads or stores, therefore, for security reasons it would be desirable to prevent execution of instructions in those smaller areas instead of preventing execution for a whole 256 MB segment. This functionality is one of the purposes of subregions: to overlay smaller sections of memory with different attributes. Additionally, they are flexible enough to cover larger areas of the memory map.

The following figure shows a region for a device's memory mapped registers that overrides the default attributes by setting the XI bit to inhibit execution for the region's memory range (assuming the whole region is enabled).

Figure 20: Region within the Address Space

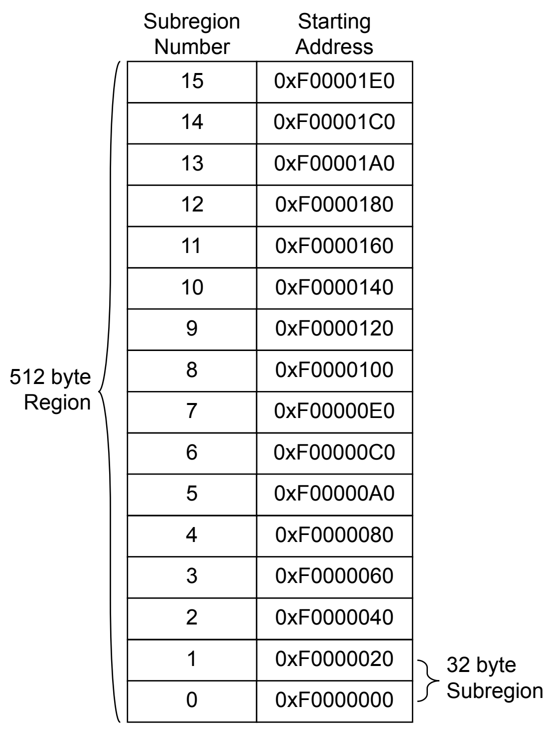


Segment Number	Starting Address
15	0xF00001FF
	Region
	0xF0000000
14	0xE0000000
13	0xD0000000
12	0xC0000000
11	0xB0000000
10	0xA0000000
9	0x90000000
8	0x80000000
7	0x70000000
6	0x60000000
5	0x50000000
4	0x40000000
3	0x30000000
2	0x20000000
1	0x10000020
0	0x00000000

Device Memory-Mapped I/O 512 byte Region at 0xF0000000 – 0xF00001FF Override (Set XI so no execution can take place in the region)

Compared to the previous example of a region within the entire address space, this figure shows just the region itself. The region is 512 bytes (0x200) made up of 16 enabled subregions of 32 bytes (0x20) all sharing the same attributes.

Figure 21: 512 byte Region

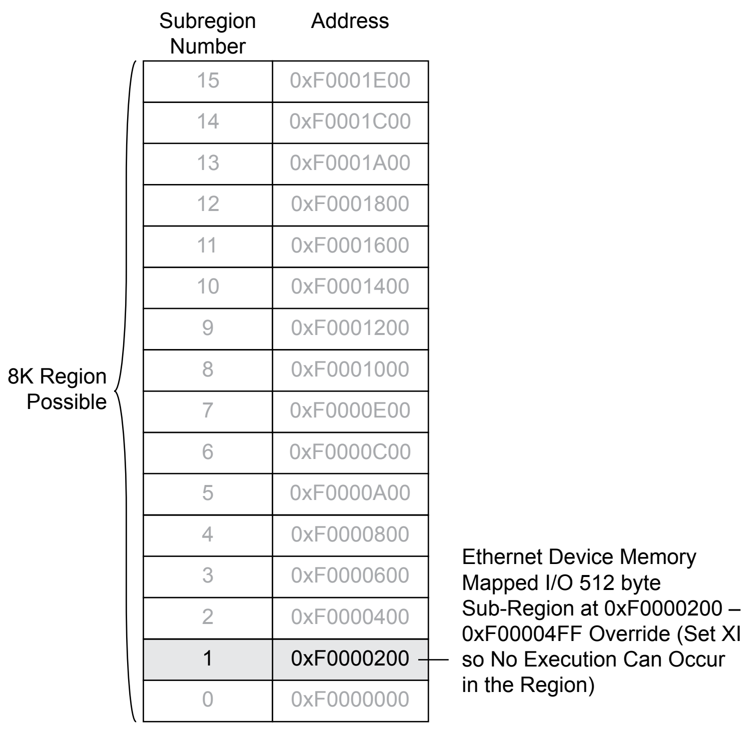


Subregion Number	Starting Address
15	0xF00001E0
14	0xF00001C0
13	0xF00001A0
12	0xF0000180
11	0xF0000160
10	0xF0000140
9	0xF0000120
8	0xF0000100
7	0xF00000E0
6	0xF00000C0
5	0xF00000A0
4	0xF0000080
3	0xF0000060
2	0xF0000040
1	0xF0000020
0	0xF0000000

512 byte Region
32 byte Subregion

Not all of the 16 subregions in a region set need to be enabled. If a subregion is not enabled, the attributes revert to the default segment programming. Subregion enabling is selectable by using the subregion's address. It is also possible to overlap subregions for even finer attribute control (the end of this chapter provides several code examples). The following figure shows a 512 byte subregion size and a starting address of 0xF000 0200. The enabled subregions start (and end) with subregion 1. The rest of the sub-regions use the default attributes because they are not enabled.

Figure 22: Enable Only One Region



4.6.1 MPU Region Address and Control Registers

To configure a region, use the region registers starting at CDMM+MPU offset plus an offset of hex 20. Each region has an address register that configures the address of the starting subregion. Each region also has a control register that configures the subregion size, number of subregions and their attributes, and an enable bit to enable the region. The following table shows how the first 2 and the last possible region fall in the MPU configuration registers.

Table 20: Region Address and Control Registers

Region Number	Offset	Register
0	0x20	Address
0	0x24	Control
1	0x28	Address
1	0x2C	Control
…	…	…
31	0x118	Address
31	0x11C	Control

MPU Region Address Register

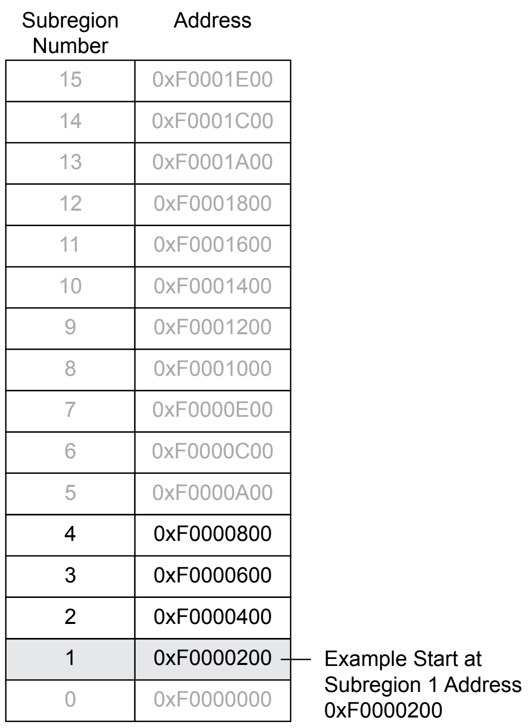
The first register in the region configuration set is the base address register.

Table 21: Base Address Register Layout

Register Fields		MPU Region Base Address Register (CDMM address + MPU Offset + 0x0020 + (Region number * 8))	Read/Write	Reset State
Name	**Bits**			
BaseAddress	31:5	Bits 5 – 31 of the address of the starting region must be aligned to sub-region size	R/W	Undefined
-	5:0	-	R	0

BaseAddress is the address of the first subregion of the region to be enabled. The following figure shows the BaseAddress starting at 0xF000 0200 which is the start of subregion 1.

Figure 23: Region Base Address



Subregion Number	Address
15	0xF0001E00
14	0xF0001C00
13	0xF0001A00
12	0xF0001800
11	0xF0001600
10	0xF0001400
9	0xF0001200
8	0xF0001000
7	0xF0000E00
6	0xF0000C00
5	0xF0000A00
4	0xF0000800
3	0xF0000600
2	0xF0000400
1	0xF0000200
0	0xF0000000

Region Control Register

The second register in the set is the region control register. It contains the enable bit for the region. This bit enables the subregions starting with the subregion at the base address. The count is the number of consecutive subregions that are enabled from the starting subregion. Last is the attributes that override the segment attributes.

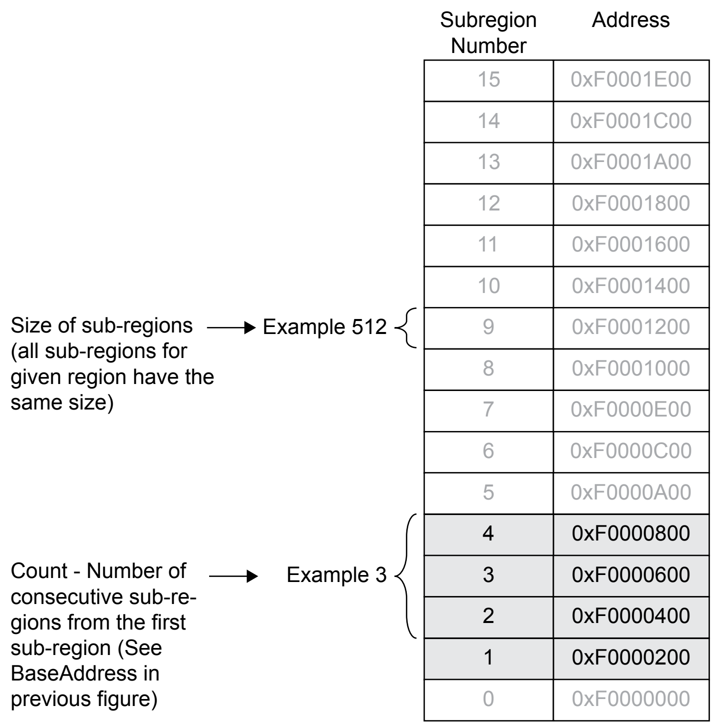
Table 22: MPU Region Control Register

Register Fields		MPU Region Control Register (CDMM address + MPU Offset + 0x0024 + (Region number * 8))	Read/Write	Reset State
Name	**Bits**			
-	31:16	-	R	Undefined

Register Fields		MPU Region Control Register (CDMM address + MPU Offset + 0x0024 + (Region number * 8))	Read/Write	Reset State
Name	**Bits**			
EN	15	Enable bit for this region. This bit must be set in order to enable a given region.	R/W	0
Size	14:10	Size of a subregion in powers of 2 coded 5 – 28 with 5 corresponding to 32 Bytes and 28 corresponding to 256 MB.	R/W	Undefined
Count	9:6	Number of additional consecutive subregions. Cannot extend beyond the region size (which is possible when BaseAddress is not region aligned).	R/W	Undefined
RI	5	Read inhibit. Trigger MPUL exception on data read.	R/W	Undefined
WI	4	Write inhibit. Trigger MPUS exception on data write.	R/W	Undefined
XI	3	Execute inhibit. Trigger MPUL exception on instruction fetch.	R/W	Undefined
CCA	2:0	Cache Coherency Attributes.	R/W	Undefined

In the following figure, subregion 1 starts at 0xF000 0200 instead of 0xF000 0000. Each subregion shares the same 512 byte size and the count of 3 adds 3 subregions that are enabled in addition to the starting one. The remaining subregions are not enabled so they retain their segment default attributes.

Figure 24: Region Control Register

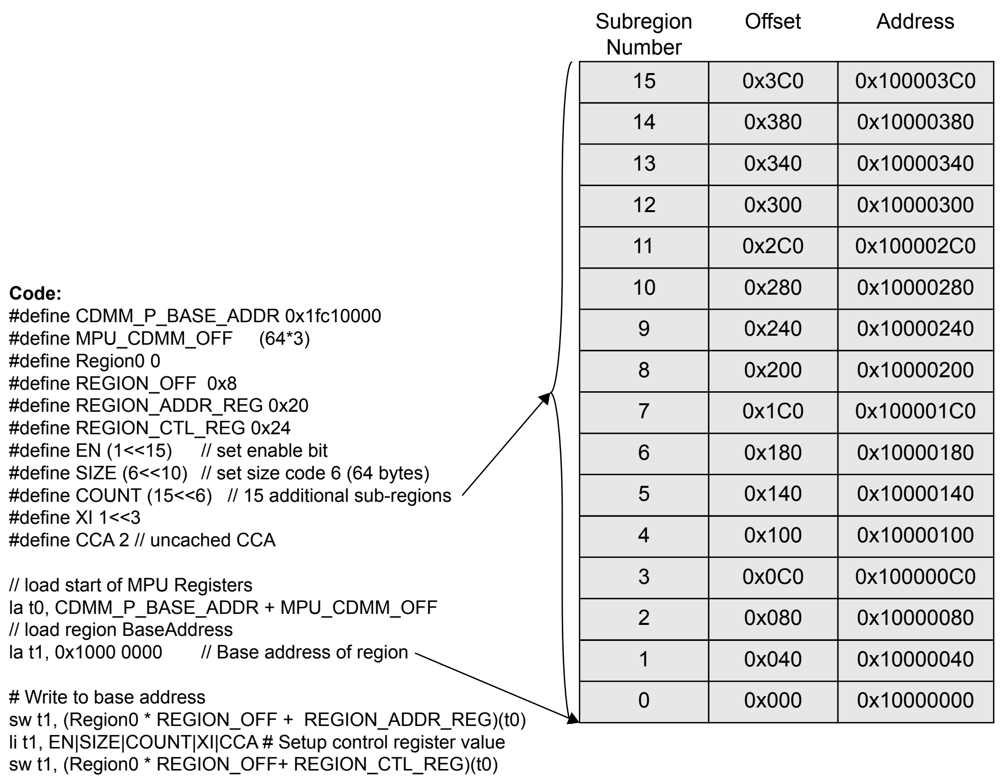


Region Example Code 1

This example sets a region where the first subregion and region address are the same and all 16 subregions are enabled. Given:

- Region: 0
- Subregion size: 64 (0x40) bytes, encoding 6
- Total Region size: 64x16=1024 (0x400) enabled (1024)
- Region addres: 0x1000 0000
- Subregion BaseAddress: 0x1000 0000
- Number of additional subregions: 15
- CCA: 2 (uncached)
- RI: 0 (readable)
- WI: 0 (writable)
- XI: 1 (execute inhibit)

The code is:



Code:

```
#define CDMM_P_BASE_ADDR 0x1fc10000
#define MPU_CDMM_OFF     (64*3)
#define Region0 0
#define REGION_OFF  0x8
#define REGION_ADDR_REG 0x20
#define REGION_CTL_REG 0x24
#define EN (1<<15)      // set enable bit
#define SIZE (6<<10)   // set size code 6 (64 bytes)
#define COUNT (15<<6)   // 15 additional sub-regions
#define XI 1<<3
#define CCA 2 // uncached CCA

// load start of MPU Registers
la t0, CDMM_P_BASE_ADDR + MPU_CDMM_OFF
// load region BaseAddress
la t1, 0x1000 0000       // Base address of region

# Write to base address
sw t1, (Region0 * REGION_OFF +  REGION_ADDR_REG)(t0)
li t1, EN|SIZE|COUNT|XI|CCA # Setup control register value
sw t1, (Region0 * REGION_OFF+ REGION_CTL_REG)(t0)
```

Subregion Number	Offset	Address
15	0x3C0	0x100003C0
14	0x380	0x10000380
13	0x340	0x10000340
12	0x300	0x10000300
11	0x2C0	0x100002C0
10	0x280	0x10000280
9	0x240	0x10000240
8	0x200	0x10000200
7	0x1C0	0x100001C0
6	0x180	0x10000180
5	0x140	0x10000140
4	0x100	0x10000100
3	0x0C0	0x100000C0
2	0x080	0x10000080
1	0x040	0x10000040
0	0x000	0x10000000

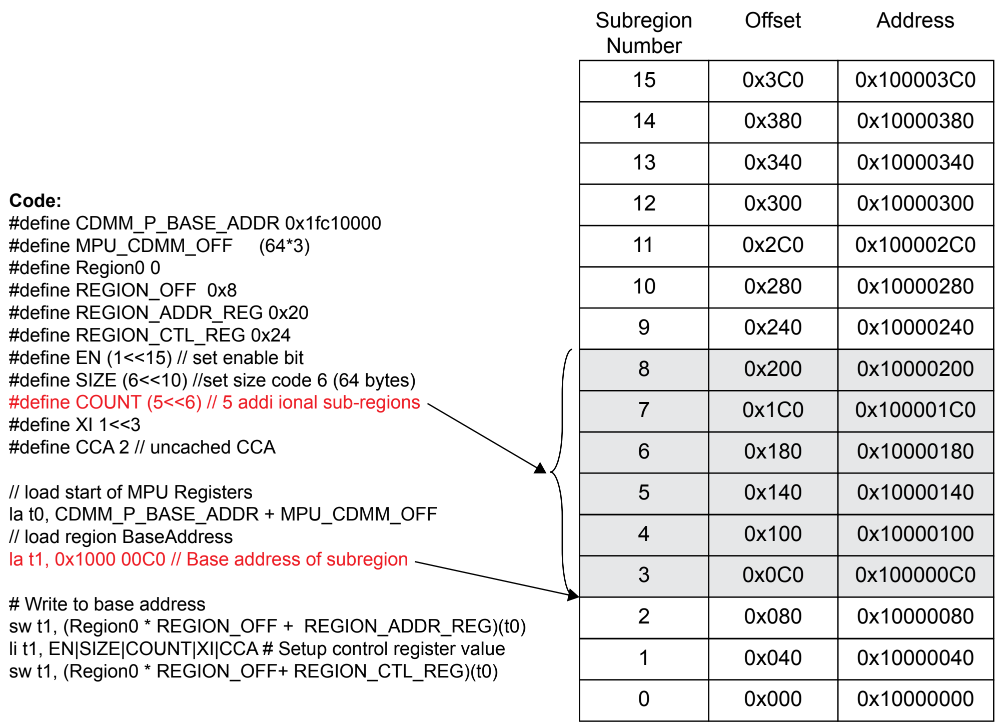
Region Example Code 2

This example sets a region where the first subregion is not the same as the region address and only 6 subregions are enabled. Changes from the previous example are highlighted. The remaining (disabled) subregions retain their segment default attributes. Given:

- Region: 0
- Subregion size: 64 (0x40) bytes, encoding 6
- Total Region size: 64x16=1024 (0x400) enabled (1024)
- Region addres: 0x1000 0000

- **Subregion BaseAddress: 0x1000 00C0**
- **Number of additional subregions: 5**
- CCA: 2 (uncached)
- RI: 0 (readable)
- WI: 0 (writable)
- XI: 1 (execute inhibit)

The code is:



Code:

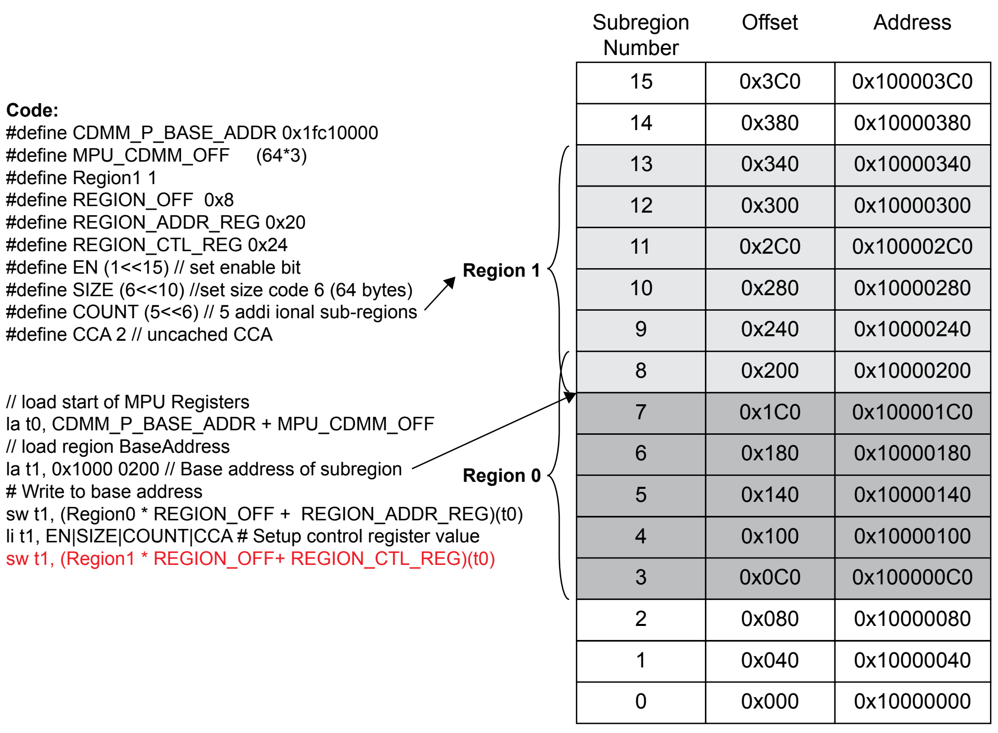
```
#define CDMM_P_BASE_ADDR 0x1fc10000
#define MPU_CDMM_OFF     (64*3)
#define Region0 0
#define REGION_OFF  0x8
#define REGION_ADDR_REG 0x20
#define REGION_CTL_REG 0x24
#define EN (1<<15) // set enable bit
#define SIZE (6<<10) //set size code 6 (64 bytes)
#define COUNT (5<<6) // 5 addi ional sub-regions
#define XI 1<<3
#define CCA 2 // uncached CCA

// load start of MPU Registers
la t0, CDMM_P_BASE_ADDR + MPU_CDMM_OFF
// load region BaseAddress
la t1, 0x1000 00C0 // Base address of subregion

# Write to base address
sw t1, (Region0 * REGION_OFF +  REGION_ADDR_REG)(t0)
li t1, EN|SIZE|COUNT|XI|CCA # Setup control register value
sw t1, (Region0 * REGION_OFF+ REGION_CTL_REG)(t0)
```

Subregion Number	Offset	Address
15	0x3C0	0x100003C0
14	0x380	0x10000380
13	0x340	0x10000340
12	0x300	0x10000300
11	0x2C0	0x100002C0
10	0x280	0x10000280
9	0x240	0x10000240
8	0x200	0x10000200
7	0x1C0	0x100001C0
6	0x180	0x10000180
5	0x140	0x10000140
4	0x100	0x10000100
3	0x0C0	0x100000C0
2	0x080	0x10000080
1	0x040	0x10000040
0	0x000	0x10000000

Region Example Code 3

This example shows an additional region with different attributes and overlaps an enabled subregion of a lower numbered region. region 0 is the same as example 2 and region 1 is at a different starting subregion address (highlighted). The starting subregion in region 1 overlaps subregion 8 of region 0.

The attributes of a higher numbered region take precedence over any overlapping subregions. Therfore, subregion 8 uses the attribute settings of region 1. The remaining (disabled) subregions retain their segment default attributes.



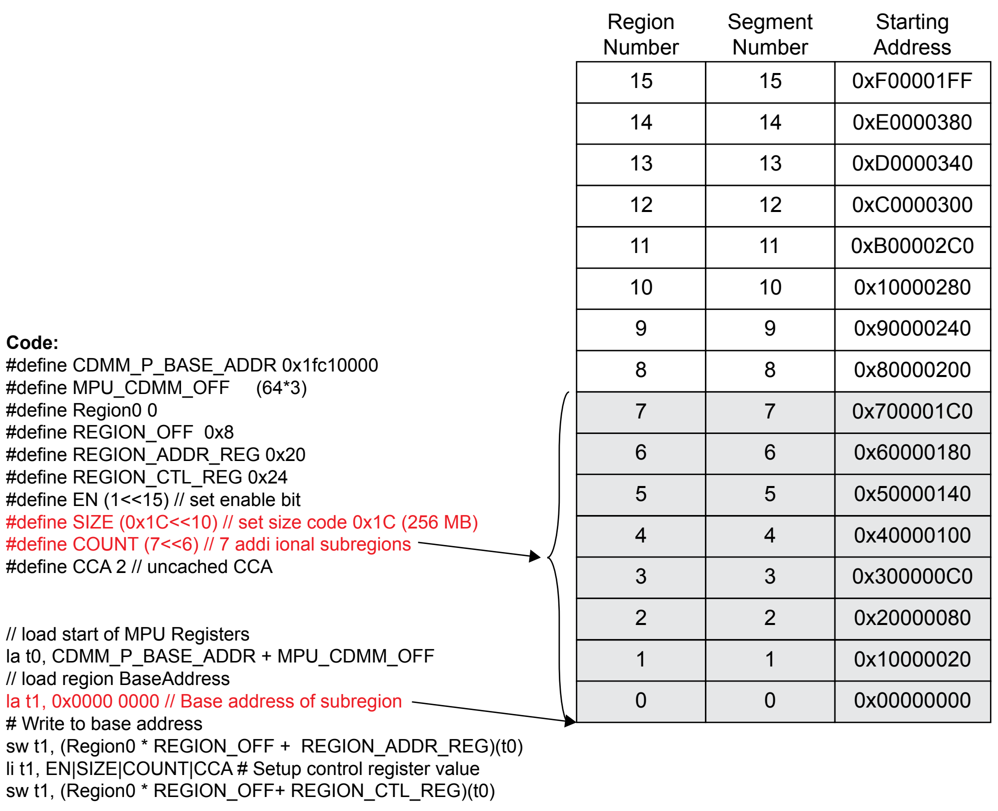
Code:

```
#define CDMM_P_BASE_ADDR 0x1fc10000
#define MPU_CDMM_OFF     (64*3)
#define Region1 1
#define REGION_OFF  0x8
#define REGION_ADDR_REG 0x20
#define REGION_CTL_REG 0x24
#define EN (1<<15) // set enable bit
#define SIZE (6<<10) //set size code 6 (64 bytes)
#define COUNT (5<<6) // 5 addi ional sub-regions
#define CCA 2 // uncached CCA


// load start of MPU Registers
la t0, CDMM_P_BASE_ADDR + MPU_CDMM_OFF
// load region BaseAddress
la t1, 0x1000 0200 // Base address of subregion
# Write to base address
sw t1, (Region0 * REGION_OFF +  REGION_ADDR_REG)(t0)
li t1, EN|SIZE|COUNT|CCA # Setup control register value
sw t1, (Region1 * REGION_OFF+ REGION_CTL_REG)(t0)
```

Subregion Number	Offset	Address
15	0x3C0	0x100003C0
14	0x380	0x10000380
13	0x340	0x10000340
12	0x300	0x10000300
11	0x2C0	0x100002C0
10	0x280	0x10000280
9	0x240	0x10000240
8	0x200	0x10000200
7	0x1C0	0x100001C0
6	0x180	0x10000180
5	0x140	0x10000140
4	0x100	0x10000100
3	0x0C0	0x100000C0
2	0x080	0x10000080
1	0x040	0x10000040
0	0x000	0x10000000

Region Example Code 4

This example creates a region that spans segments. A region covers the entire 4 GB of address space by using a 256 MB subregion and a starting address of 0; 16 subregions equal 4 GB. 7 subregions are enabled, covering the lower 2 GB of memory.



Code:

```
#define CDMM_P_BASE_ADDR 0x1fc10000
#define MPU_CDMM_OFF     (64*3)
#define Region0 0
#define REGION_OFF  0x8
#define REGION_ADDR_REG 0x20
#define REGION_CTL_REG 0x24
#define EN (1<<15) // set enable bit
#define SIZE (0x1C<<10) // set size code 0x1C (256 MB)
#define COUNT (7<<6) // 7 addi ional subregions
#define CCA 2 // uncached CCA


// load start of MPU Registers
la t0, CDMM_P_BASE_ADDR + MPU_CDMM_OFF
// load region BaseAddress
la t1, 0x0000 0000 // Base address of subregion
# Write to base address
sw t1, (Region0 * REGION_OFF +  REGION_ADDR_REG)(t0)
li t1, EN|SIZE|COUNT|CCA # Setup control register value
sw t1, (Region0 * REGION_OFF+ REGION_CTL_REG)(t0)
```

Region Number	Segment Number	Starting Address
15	15	0xF00001FF
14	14	0xE0000380
13	13	0xD0000340
12	12	0xC0000300
11	11	0xB00002C0
10	10	0x10000280
9	9	0x90000240
8	8	0x80000200
7	7	0x700001C0
6	6	0x60000180
5	5	0x50000140
4	4	0x40000100
3	3	0x300000C0
2	2	0x20000080
1	1	0x10000020
0	0	0x00000000

Number of MPU Regions

It may be useful to know how many regions are available. The MPU_Config register stores the number of regions in the NumRegions field. This value is static and is pre-built into the core. The core can be built with 8, 12, 16, 20, 24, 28, or 32 regions.

Table 23: Finding the Number of MPU Regions

Register Fields		MPU_Config Register	Read/Write	Reset State
Name	Bits	(CDMM address + MPU Offset + 0x0008)		
NumRegions	4:0	Number of protected regions implemented. (Actual = NumRegion +1)	R	Preset

4.7 MPU and Segment Control

When the MPU is configured, segment-control registers are not supported. However, some capabilities, such as BEV relocation, are retained.

The following table highlights the differences and supported capabilities in these modes.

Table 24: TLB, FMT, and MPU Capabilities

	TLB	FMT	MPU
MMU Type	tlb	fmt	mpu-seg
MPU Support Configured	None	MPU-segment is de-configured MPU-region is configured	MPU-segment and MPU-regions are enabled
MPU Segment	N/A	Disabled permanently	Enabled permanently
MPU Region	N/A	Default is off. It is possible to turn on MPU Regions.	Default is off. When MPU Regions are turned on: • In overlapping regions and segments, CCA, XI, RI, and WI are defined by MPU-region takes priority over MPU-segments. • VA-to-PA mapping is still VA=PA regardless of MPU-region being enabled or disabled.
SegCtl, EVA, Overlay	Yes	No	No
BEV Relocation	Yes	Yes	Yes

K0, K23, KU, CV	K23 = RO - 0
	KU = RO - 0
	K0 = R/W – 2
	Config5.K = R/W – 0
	Config5.CV = R/W - 0
		K23 = R/W - 2
		KU = R/W - 2
		K0 = R/W – 2
		Config5.K = RO – 0
		Config5.CV = RO - 0
			K23 = RO - 0
			KU = RO - 0
			K0 = RO – 0
			Config5.K = RO – 0
			Config5.CV = RO - 1

	TLB	FMT	MPU
Notes	Standard TLB configuration with Segment Control Register support. Fully supports: • EVAReset • LegacyUserExceptionBase • BEVExceptionBaseMask • BEVExceptionBasePA • BEVExceptionBase	The core operates in FMT mapping mode. EVAReset (Core-Local Reset Exception Extended Base Register 0x30) is expected to be 0. LegacyUseExceptionBase is controllable by software. If this bit is 0, core boots at bfc0_0000 or legacy boot. BEVExceptionBaseMask[27:20] is unused. BEVExceptionBasePA[31:29] is unused. This mode does not support PA relocation. BEVExceptionBase[31:12] is used if LegacyUseExceptionBase = 1. Bits [31:30] are tied to 2'b10 because the boot-vector must be located within kseg1.	The core operates in FMT mapping mode. EVAReset (Core-Local Reset Exception Extended Base Register 0x30) is expected to be 0. LegacyUseExceptionBase is controllable by software. If this bit is 0, core boots at bfc0_0000 or legacy boot. BEVExceptionBaseMask[27:20] is unused. BEVExceptionBasePA[31:29] is unused. This mode does not support PA relocation. BEVExceptionBase[31:12] is used if LegacyUseExceptionBase = 1.

5 Caches

The I7200 Multiprocessing System (MPS) 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.

When configured with an MPU, the I7200 L1 instruction and data caches support up to a maximum of 128 KB, improving L1 hit rates and performance.

Table 25: I7200 Cache Configurations

Attribute	L1 Instruction Cache	L1 Data Cache	L2 Cache
Size[4]	4 KB, 8 KB, 16 KB, 32 KB, 64 KB, (or 128 KB with MPU)	4 KB, 8 KB, 16 KB, 32 KB, 64 KB, (or 128 KB with MPU)	0, 128 KB, 256 KB, 512 KB, 1024 KB, 2048 KB, 4096 KB, 8192 KB
Line Size	32 Bytes	32 Bytes	32 or 64 Bytes
Number of Cache Sets	Cache Size/32 B	Cache Size/32B	0, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768
Associativity	4 way	4 way	8 way

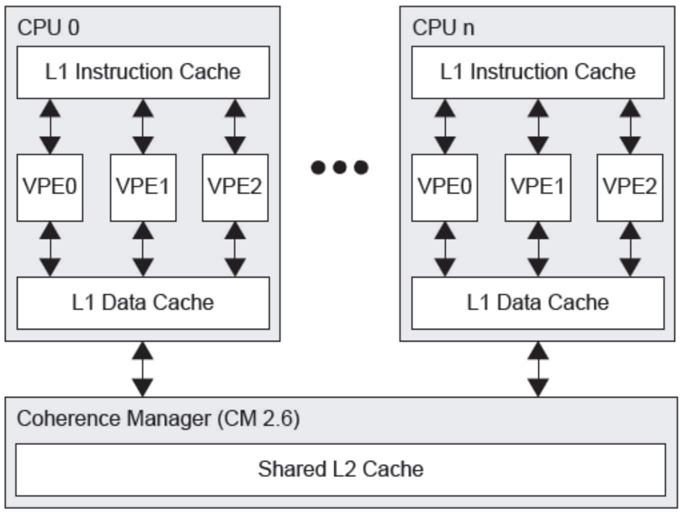
This chapter provides an overview of the cache architecture and a description of the elements that go into programming the caches. A description of the CP0 register interface to each cache is provided, as well as cache initialization code. Other programmable elements include setting up cache coherency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

5.1 Caches Substem Overview

[4] For Linux-based applications, MIPS recommends an optimum L1 cache size of 64 KB, and a minimum L1 cache size of 32 KB.

The following figure shows the relative location of the caches within the I7200 MPS. The L1 instruction and L1 data caches are shared by all VPEs in the same core. The L2 cache is shared by all cores.

Figure 25: I7200 Multiprocessing System Caches



5.1.1 L1 Instruction Cache

The L1 instruction cache contains three arrays: tag, data, and way-select. The L1 instruction cache is virtually indexed, since a virtual address is used to select the appropriate line within each of the three arrays. The caches are physically tagged, as the tag array contains a physical, not virtual, address.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. The way-select array holds information to choose the way to be filled, as well as dirty bits in the case of the data cache.

An instruction cache tag entry consists of the upper bits of the physical address bits, one valid bit for the line, and a lock bit. An instruction cache data entry contains four, 64-bit doublewords in the line, for a total of 32 bytes. All four words in the line are present or not in the data array together, hence the single valid bit stored with the tag. The number of upper address bits depends on the cache size as shown below.

- bits [31:12] for 128 KB, 64 KB, 32 KB, and 16 KB caches
- bits[31:11] for 8 KB cache
- bits[31:10] for 4 KB cache

A way-select entry holds bits choosing the way to be replaced according to a Least Recently Used (LRU) algorithm. The LRU information applies to all the ways and there is one way-select entry for all the ways in the set. The instruction cache only supports reads, hence only LRU entries are stored in the instruction way-select array.

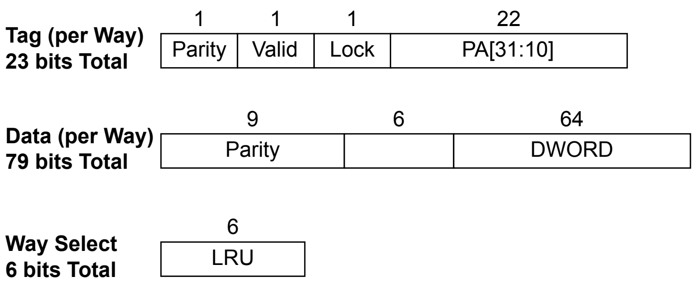
Table 26: L1 Instruction Cache Attributes

Attribute	With Parity[5]
Size[6]	0 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 128 KB
Line Size	32 bytes
Number of Cache Sets	32, 64, 128, 256, or 512
Associativity	4-way

[5] **Only applies if parity is supported.**
[6] For Linux based applications, MIPS recommends a 64 KB L1 instruction cache size, with a minimum size of 32 KB.

Attribute	With Parity[5]
Replacement	LRU
Cache Locking	per line
Data Array	
Read Unit	64b x 4 (no parity), 72b x 4 (parity)
Write Unit	64b x 4 (no parity), 72b x 4 (parity)
Tag Array	
Read Unit	24b x 4 (no parity) 25b x 4 (parity)
Write Unit	24b (no parity) 25b (parity)
Way-Select Array	
Read Unit	6b
Write Unit	1-6b

Figure 26: L1 Instruction Cache Organization



L1 Instruction Cache Virtual Aliasing

The instruction cache on the I7200 core is virtually indexed and physically tagged. The lower bits of the virtual address are used to access the cache arrays and the physical address is used in the tags. Because the way size can be larger than the minimum TLB page size, there is a potential for virtual aliasing. This means that one physical address can exist in multiple indices within the cache, if it is accessed with different virtual addresses. Virtual aliasing comes into effect only for cache sizes that are larger than 16 KB and when using the TLB-based MMU.

In the I7200 core, the $Config7_{IAR}$ bit is set to indicate the existence of instruction cache virtual aliasing hardware based on cache size and MMU type. The core allows a physical address to reside at multiple indices if accessed with different virtual addresses. When an invalidate request is made due to the `CACHE` or `SYNCI` instructions, the core will serially check each possible alias location for the given physical address.

The hardware can be enabled and disabled using the $Config7_{IVAD}$ bit. When this bit is cleared, the hardware used to remove instruction cache virtual aliasing is enabled. In this case the virtual aliasing is managed in hardware. No software interaction is required. When the $Config7_{IVAD}$ bit is set, the virtual aliasing hardware is disabled. This can be done when software ensures that no cache aliases are possible, for example when using a minimum TLB page size of 16 KB. In cases where the TLB page size is less than 16 KB, it is up to software to manage virtual aliasing within the instruction cache.

L1 Instruction Cache Line Locking

The I7200 core does not support the locking of all 4 ways of either cache at a particular index. If all 4 ways of the cache at a given index are locked by either Fetch and Lock or Index Store Tag `CACHE` instructions, subsequent cache misses at that cache index will displace one of the locked lines.

[5] **Only applies if parity is supported.**

Locking lines in the caches is somewhat counter to the idea of coherence. If a line is locked into a particular cache, it is expected that any processes utilizing that data will be locked to that processor and coherence is not needed. Based on this usage model, locking coherent lines into the cache is not recommended. However, should this occur, the CPU adheres to the following rules:

- `SYNCI` instructions are user-mode instructions. Because locking is a kernel mode feature (requires the `CACHE` instruction), `SYNCI` is not allowed to unlock cache lines. This applies to both local and globalized `SYNCI` instructions.
- Locking overrides coherence. Intervention requests from other CPUs and I/O devices that match on a locked line are treated as misses.
- Self-intervention requests for globalized `CACHE` instructions are allowed to affect a locked line. This is done primarily for handling lock and unlock requests for kseg0 addresses when kseg0 is being treated coherently.

L1 Instruction Cache Memory Coherence Issues

The I7200 core supports software cache coherency in a multi-CPU cluster. Software must explicitly manage instruction cache coherence via the `CACHE` or `SYNCI` instructions to invalidate a line and pick up new data from L2 cache or main memory. These operations are globalized—if the address used in the operation has a coherent CCA, the request will be sent to all instruction caches in the cluster.

In the I7200 core, the hardware does not automatically keep the instruction caches coherent with the data caches. Doing so requires many additional cache lookups and would likely require the instruction cache tag array to be duplicated as well. For many types of code, this would be of small benefit, and the added area and power costs would not make sense. Further, the existing non-coherent cores from MIPS do not keep the I-Cache coherent with the D-Cache, so the code already exists for software I-Cache coherence where it is required. Globalized `CACHE` and `SYNCI` instructions ease the task of software I-Cache coherence. Existing, single-CPU routines that push dirty data out of the data cache and invalidate stale instruction cache lines using hit-type `CACHE` or `SYNCI` instructions can be globalized, and the coherence can be handled for all of the instruction caches in parallel.

Software I-Cache Coherence (JVM, Self-modifying Code)

The CPU does not support hardware I-Cache coherence, so code that modifies the instruction stream must clean up the instruction cache. This is equivalent to what is currently required on uniprocessor systems that also do not have a coherent I-Cache. The recommended `SYNCI` sequence shown below will also work for coherent addresses:

```
SW instn_address
SYNCI instn_address
SYNC
JR.HB instn_address
NOP
```

L1 Instruction Software Cache Management

The L1 instruction cache is not fully “coherent” and requires OS intervention at times. The `CACHE` instruction is the building block of such OS interventions, and is required for correct handling of DMA data and for cache initialization. Historically, the `CACHE` instruction also had a role when writing instructions. Unless the programmer takes the appropriate action, those instructions may only be in the D-cache and would need them to be fetched through the I-cache at the appropriate time. Wherever possible, use the `SYNCI` instruction for this purpose.

A cache operation instruction is written `cache op, s(rs)` where `s(rs)` is an address format (register plus immediate), written as for a load/store instruction. Cache operations are privileged and can only run in kernel mode (`SYNCI` works in user mode).

Table 27: Fields in the Encoding of a CACHE Instruction

31 26	25 21	20 18 17 16	15	14 11	10	9 8	7 0
101001	op	rs	s[8	0111	0	01	s[7:0]

The op field packs together a 5-bit field. The lower 2 bits of this field (17:16) select which cache to work on:

- 00—L1 I-cache
- 01—L1 D-cache
- 10—Reserved
- 11—L2 cache

The upper 3-bits of the OP field encodes a command to be carried out on the line the instruction selects.

The `CACHE` instruction come in three varieties which differ in how they pick the cache entry (the "cache line") they will work on:

- **Hit-type cache operation:** presents an address (just like a load/store), which is looked up in the cache. If this location is in the cache (it "hits") the cache operation is carried out on the enclosing line. If this location is not in the cache, nothing happens.
- **Address-type cache operation:** presents an address of some memory data, which is processed just like a cached access—if the cache was previously invalid the data is fetched from memory.
- **Index-type cache operation:** as many low bits of the address as are required are used to select the byte within the cache line, then the cache line address inside one of the four cache ways, and then the way. The size of the cache (contained within the Config1 register) to know exactly where the field boundaries are located. The address is used as follows:

31		5	4 0
Unused	Way 1-0	Index	byte-within-line

Note: The MIPS32 specification allows the CPU designer to select whether to derive the index from the virtual or physical address. For index-type operations, MIPS recommends using a kseg0 address, so that the virtual and physical address are the same. This also avoids a potential of cache aliasing.

5.1.2 L1 Data Cache

The L1 data cache is similar to the instruction cache, with a few key differences.

- The dirty bit is part of the way-select RAM.
- To handle store bytes, the data array is byte-accessible, and the data parity is 1 bit per byte (if parity is supported).
- If ECC is supported, an ECC code is generated across a 32b word. Reads and writes are 32 or 64b. Sub-word stores are handled by doing a read-modify-write sequence.

Like the L1 instruction cache, the L1 data cache is virtually indexed, because a virtual address is used to select the appropriate line within each of the arrays. The cache is physically tagged, as the tag array contains a physical, not virtual, address.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. The way-select array holds information to choose the way to be filled, as well as dirty bits in the case of the data cache.

A tag entry consists of the upper bits of the physical address bits [31:10], a valid bit, and a lock bit. A data entry contains the four, 64-bit doublewords in the line, for a total of 32 bytes. All four words in the line are present or not in the data array together, hence the single valid bit stored with the tag. Once a valid line is resident in the cache, byte, halfword, triple-byte, word, or doubleword stores can update all or a portion of the words in that line. The tag and data entries are repeated for each of the 4 lines in the set.

A way-select entry holds bits choosing the way to be replaced according to a Least Recently Used (LRU) algorithm. The LRU information applies to all the ways and there is one way-select entry for all the ways in the set.

Table 28: L1 Data Cache Organization

Attribute	Without Parity	With Parity[7]	With ECC[8]
Size[9]	0 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 128 KB	0 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 128 KB	0 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 128 KB
Line Size	32-byte	32-byte	32-byte
Number of Cache Sets	32, 64, 128, 256, or 512	32, 64, 128, 256, or 512	32, 64, 128, 256, or 512
Associativity	4-way	4-way	4-way
Replacement	LRU	LRU	LRU
Cache Locking	per line	per line	per line
Data Array			
Read Unit	64b x 4	72b x 4	78b x 4
Write Unit	8b	9b	39b
Tag Array			
Read Unit	24b x 4	25b x 4	31b x 4
Write Unit	24b	25b	31b
Way-Select Array			
Read Unit	10b	14b	22b
Write Unit	1-10b	1-14b	1-22b

L1 Data Cache Virtual Aliasing

The data cache on the I7200 core is virtually indexed and physically tagged. The lower bits of the virtual address are used to access the cache arrays and the physical address is used in the tags. Because the way size can be larger than the minimum TLB page size, there is a potential for virtual aliasing. This means that one physical address can exist in multiple indices within the cache, if it is accessed with different virtual addresses.

The following table indicates the conditions under which virtual aliasing can occur.

Table 29: L1 Data Cache Virtual Aliasing Conditions

Cache Size	MMU Page Size	Way Size	Aliasing Can Occur	Hardware Aliasing Fix Required
32 KB	4 KB	8 K	Yes	Yes
64 KB	4 KB	16 K	Yes	Yes
32 KB	≥ 16 KB	8 K	No	No
64 KB	≥ 16 KB	16 K	No	No

In the I7200 core, the read-only Config7$_{AR}$ bit determines whether the data cache virtual aliasing hardware is enabled based on the build-time configuration. Note that for some of the configuration options in the

[7] **Only applies if parity is supported.**
[8] **Only applies if ECC is supported.**
[9] For Linux based applications, MIPS recommends a 64 KB L1 instruction cache size, with a minimum size of 32 KB.

table above, the hardware aliasing fix (HWAF) is required. As such, it is incumbent upon the designer to select the HWAF option at build time. The selection of this option causes hardware to set the Config7$_{AR}$ bit.

L1 Data Cache Line Locking

The mechanism for line locking in the L1 data cache is identical to that of the L1 instruction cache. For more information, refer to *L1 Instruction Cache Line Locking* on page 65.

L1 Data Cache Memory Coherence Protocol

The I7200 core supports cache coherency in a multi-CPU cluster using Cache Coherence Attributes (CCAs) specified on a per cache-line basis and an Intervention Port containing coherent requests by all CPUs in the system. Each I7200 core monitors its Intervention Port and updates the state of its cache lines (valid, lock, and dirty tag bits) accordingly.

The L1 data caches utilize a standard MESI protocol. Each cache line will be in one of the following four states:

- **Invalid:** The line is not present in this cache.
- **Shared:** This cache has a read-only copy of the line. The line may be present in other L1 data caches, also in a shared state. The line will have the same value as it does in the L2 cache or memory.
- **Exclusive:** This cache has a copy of the line with the right to modify. The line is not present in other L1 data caches. The line is still clean - consistent with the value in L2 cache or memory.
- **Modified:** This cache has a dirty copy of the line. The line is not present in other L1 data caches. This is the only up-to-date copy of the data in the system (the value in the L2 cache or memory is stale).

The `SYNC` instruction may also be useful to software in enforcing memory coherence, because it flushes the write buffers.

Some of the basic characteristics of the coherence protocol are summarized below. Coherence can occur on the data cache.

- **Writeback cache:** Uses a writeback cache to ensure high performance
- **Cache-line based:** Coherence and ownership is maintained per 32-byte cache line
- **Snoopy protocol:** Each CPU snoops the stream of transactions and updates its cache state accordingly
- **Invalidate:** A line is invalidated from the cache (possibly with a writeback to memory) when a store from another processor is seen.

5.1.3 L1 Instruction and Data Cache Software Testing

Typically, the cache RAM arrays will be tested using BIST. It is, however, possible for software running on the processor to test all of the arrays. Of course, testing of the I-cache arrays should be done from an uncacheable space with interrupts disabled in order to maintain the cache contents. There are multiple methods for testing these arrays in software, some of which are described in the following subsections

L1 Instruction Cache Tag Array

The L1 instruction cache tag array can be tested via the Index Load Tag and Index Store Tag varieties of the `CACHE` instruction. An Index Store Tag writes the contents of the ITagLo and ITagHi registers into the selected tag entry. An Index Load Tag reads the selected tag entry into the ITagLo and ITagHi registers.

If parity is implemented, the parity bits can be tested as normal bits by setting the PO (parity override) bit in the ErrCtl register. This will override the parity calculation and use the parity bits in ITagLo and ItagHi as the parity values.

Instruction Cache Data Array

This array can be tested using the Index Store Data and Index Load Tag varieties of the `CACHE` instruction. The Index Store Data variety is enabled by setting the WST bit in the ErrCtl register.

If parity is implemented, the parity bits in the array can be tested by setting the PO bit in the ErrCtl register. This will use the PI field in ErrCtl instead of calculating the parity on a write.

The rest of the data bits are read/written to/from the IDataLo and IDataHi registers.

Instruction Cache Way Select Array

The testing of this array is done with via Index Load Tag and Index Store Tag `CACHE` instructions. By setting the WST bit in the ErrCtl register, these operations will read and write the WS array instead of the tag array.

L1 Data Cache Tag Array

The L1 data cache tag array can be tested via the Index Load Tag and Index Store Tag varieties of the `CACHE` instruction. An Index Store Tag writes the contents of the DTagLo register into the selected tag entry. An Index Load Tag will read the selected tag entry into the DTagLo register.

If parity is implemented, the parity bits can be tested as normal bits by setting the PO (parity override) bit in the ErrCtl register. This will override the parity calculation and use the parity bits in DTagLo as the parity values.

If ECC is implemented, the ECC bits can be tested as normal bits by setting the PO (parity override) bit in the ErrCtl register. This overrides the ECC calculation and uses the ECC bits in the DTagHi register as the ECC values.

Duplicate Data Cache Tag Array

This array can be tested via the Index Load Tag and Index Store Tag varieties of the `CACHE` instruction. To access the duplicate tags, set both the WST and SPR bits of ErrCtl. Index Store Tag will write the contents of the TagLo register into the selected tag entry. Index Load Tag will read the selected tag entry into the TagLo. In normal mode, with WST and SPR cleared, IndexStoreTags will write into both the primary and duplicate tags, while IndexLoadTags will read the primary tag.

If parity is implemented, the parity bit can be tested as a normal bit by setting the PO bit in the ErrCtl register. This will override the parity calculation and write P bit in TagLo as the parity value.

Data Cache Data Array

This array can be tested using the Index Store Tag `CACHE`, `SW`, and `LW` instructions. First, use Index Store Tag to set the initial state of the tags to valid with a known physical address (PA). Write the array using SW instructions to the PAs that are resident in the cache. The value can then be read using LW instructions and compared to the expected data.

If parity is implemented, the parity bits can be implicitly tested using this mechanism. The parity bits can be explicitly tested by setting the PO bit in ErrCtl and using Index Store Data and Index Load Tag `CACHE` operations. The parity bits (one bit per byte) are read/written to/from the PD field in ErrCtl. Unlike the I-cache, the DataHi register is not used, and only 32b of data is read/written per operation.

Data Cache Way Select Array

The dirty and LRU bits can be tested using the same mechanism as the I-cache WS array.

5.1.4 Batch Cache Operation

Traditional MIPS cache control instructions are achieved using the CACHE, PREF, and SYNCI instructions. These instructions operate on a single memory location (either word or cache-line). To invalidate and write-back multiple cache lines (8 cache-line blocks), the I7200 core L1 data and instruction caches have batch flush controls. CP0 includes two registers to facilitate these batch operations:

- CP0[22][0]: Batch Cache Operation Configuration Register - BatchCacheOpControl
- CP0[22][1]: Batch Cache Operation Status Register - BatchCacheOpStatus (per TC)

A typical operation starts by using MTC0 to program the BatchCacheOpControl register with the following information:

- Starting virtual address.
- Number of subsequent 8 cache-line blocks.
- Desired operation: invalidate, write-back, or a combination of these operations.
- Cache on which to operate (I-cache or D-cache).

Write 1'b1 into the PAGE_START_ADDR field to initiate the procedure. The program can monitor the BatchCacheOpStatus register to determine whether the batch operation has completed successfully. At the end of the operation, the program should read RESULT to determine whether the batch operation has completed without failure or exceptions (e.g., no interrupts, cache errors, or TLB and MPU related exceptions).

Instead of operating on a virtual address range (determined by the BatchCacheOpControl register), the program can operate on the entire cache: an Index operation flushes the entire cache, a Page operation performs a normal address flush.

The batch flush operation has the following known and expected limitations:

- If a register/field is not initialized (i.e., has a value of X), and a batch operation was attempted, the result is UNPREDICTABLE.
- Batch operation is not atomic: interrupts, cache error exceptions, and bus errors can terminate the operation. The core does not automatically restart this operation. The software is required to restart this operation by resetting the batch cache operation register fields and reinitiating the batch operation.
- This operation blocks the pipeline. Other TCs and VPEs within the core do not execute, which prevents another TC or VPE from performing another batch cache operation.
- The batch operation does not operate beyond a single page. It begins with the starting address and traverses for the number of cache-lines determined by the BatchCacheOpControl register. If a TLB or MPU exception is detected, the batch operation stops and the error status is reported in the RESULT register field.
- This operation does not cause TLB or MPU exceptions (because MTC0 does not exert TLB or MPU exceptions). Software should ensure that the target page is accessible and in the TLB or MPU region.
- This operation does not cause address-related exceptions; breakpoints and watchpoint exceptions are ignored.
- Unlike the CACHE instruction, this feature does not operate on SPRAMs.

5.1.5 L2 Cache

The L2 cache processes transactions that are not serviced by the L1 cache. L2 is generally larger than the L1 cache, but slower, due to the use of higher-density memories.

The L2 communicates with external memory via an AMBA AXI interface. The L2 cache is integrated into the Coherence Manager (CM), which reduces both latency and complexity.

The L2 also communicates with the CPU(s) through the performance counter interface, error reporting interface, and other side band signals. In addition to these interfaces, the L2 has the clock, reset, and bypass signals as well as some static input signals that can be used to configure it for different operating modes.

The following table shows the list of possible L2 cache configurations.

Table 30: Valid and Invalid L2 Cache Configurations

Line Size	Sets per Way	Number of Ways	Total L2 Cache Size	Valid L2 Cache Configuration	Notes
32 bytes	512	8	128 KBytes	Yes	
32 bytes	1024	8	256 KBytes	Yes	
32 bytes	2048	8	512 KBytes	Yes	
32 bytes	4096	8	1 MByte	Yes	
32 bytes	8192	8	2 MBytes	Yes	
32 bytes	16384	8	4 MBytes	Yes	
32 bytes	32768	8	8 MBytes	Yes	
64 bytes	64	8	32 KBytes	Yes	No ECC
64 bytes	128	8	64 KBytes	Yes	No ECC
64 bytes	256	8	128 KBytes	Yes	No ECC
64 bytes	512	8	256 KBytes	Yes	
64 bytes	1024	8	512 KBytes	Yes	
64 bytes	2048	8	1 MByte	Yes	
64 bytes	4096	8	2 MBytes	Yes	
64 bytes	8192	8	4 MBytes	Yes	
64 bytes	16384	8	8 MBytes	Yes	
64 bytes	32768	8	16 MBytes	No	32768 sets/way valid only with 32 byte line size

0K L2 Cache Option

The I7200 MPS contains a 0K cache option that is selected during IP configuration. If the Enable L2 option is selected, the cache size field can be used to select the cache size between 128 KB and 8 MB. If this option is not selected, the L2 cache is disabled and the cache size field is not available (0K option).

32K and 64K L2 Cache Options

The I7200 MPS contains 32K and 64K cache options that are selected during IP configuration. These options are supported only when the L2 cache is configured with a 64-byte line size and without ECC.

Cacheable vs. Uncacheable vs. Uncached Accelerated

The L2 cache supports cacheable and uncacheable accesses. This information also is conveyed on the AxCACHE field. Cacheable operations access the cache memories, whereas an uncached access bypasses the L2 cache arrays and is sent directly to the main memory.

Uncached accelerated accesses are treated the same way as non-accelerated uncached accesses. This CCA enables uncached transactions to better utilize bus bandwidth via burst transactions. L2 supports single-beat as well as 4-beat burst uncacheable transactions for both read and write operations.

Sleep Mode Using the WAIT Instruction

In addition to slowing down or stopping the primary cm_clk input, software may initiate low-power Sleep Mode via the execution of the WAIT instruction in the processor.

When the processor enters into Sleep Mode, it will assert SI_Sleep. The SI_Sleep drives the SI<n>_Sleep input to the L2. All cores should assert SI<n>_Sleep. The L2 then enters a low-power state and asserts the L2_Sleep output when all outstanding bus activity has completed. Most clocks in the L2 will be stopped, but a handful of flops remain active to sense the wake up call from the processor, which is the deassertion of SI<n>_Sleep.

Power is reduced because the global clock goes to the vast majority of flops within the L2, which are held idle during this period. There is no bus activity while the L2 is in sleep mode, so the system bus logic which interfaces to the L2 could be placed into a low power state as well.

When the L2 samples SI<n>_Sleep asserted and there is no activity in the L2, the L2 asserts L2_Sleep two cm_clks later. Any activity in the L2 will delay the start of L2_Sleep assertion.

Internal Dynamic Sleep Mode

When there is no activity at the input pins of the L2 cache and all pending transactions from the CPU are completed, the L2 cache will eventually empty. When this occurs, the L2 cache turns off the l2_clk signal after some small delay.

Besides programmable power control using the WAIT instruction to induce sleep mode, the L2 is also equipped with dynamic global clock gating. When there are no pending transactions in the L2 cache, the L2 shuts down the majority of internal clocks to save power. This action is not programmable.

.

Bypass mode

Bypassmode is a test/bringup feature that causes the L2 cache to forward all requests received from the CM directly to the system memory interface. Entering or exiting from Bypass Mode other than at reset requires flushing of the L2 cache while running from uncached memory to restore the L2 cache state to a stable state. In bypass mode, all requests are forwarded to the system as received including L2 CACHE instructions.

Note: Bypass mode is strictly a debug feature and is not intended to be a normal mode of operation. It is not intended for active switching during normal operation.

L2 Cache Fetch and Lock

In the L2 cache, each line in a way can be locked independently. If a line is locked it will not be evicted. Software is not allowed to lock all available ways at the same cache index, since L2 would be unable to refill any other addresses at that index.

If the requested address is not contained in the L2 cache, the line is refilled and then locked in the cache. The LRU bits in the WS array are updated to make the fetched way most-recently-used. The dirty bit and the dirty parity bit are set to clean.

On a hit the L2 cache line is locked and the operation retires. The LRU bits or the dirty bits are not affected.

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, for the 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.

L2 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.

5.2 Cache Coherency Attributes

The I7200 core defines a set of Cache Coherency Attributes (CCA).

The I7200 core supports the following cacheability attributes:

- *Uncached (0x2)* : 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.
- *Non-coherent Writeback With Write Allocation (0x3)*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is in the cache. If it is, the cache contents are updated, but main memory is not written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. Hence, the

allocation policy on a cache miss is read- or write-allocate. Data stores will update the appropriate dirty bit in the 'dirty' array to indicate that the line contains modified data. When a line with dirty data is displaced from the cache, it is written back to memory.

- *Coherent Write-back With Write Allocation, Exclusive (0x4)*: This attribute is similar to Coherent Write-back With Write Allocation, Exclusive on Write, except that load misses bring data into the cache in the exclusive state rather than the shared state. This can be used if data is not shared and will eventually be written. This can reduce bus traffic, because the line does not have to be refetched in an exclusive state when a store is done.
- *Coherent Write-back With Write Allocation, Exclusive on Write (0x5)*: Use coherent data. Load misses will bring the data into the cache in a shared state. Multiple caches can contain data in the shared state. Stores will 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 will be invalidated and brought back into the cache in an exclusive state.
- *Uncached Accelerated (0x7)* : Uncached stores are gathered together for more efficient bus utilization.

5.3 Register Interface

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

5.3.1 L1 Instruction Cache CP0 Register Interface

The I7200 core uses CP0 registers for instruction cache operations.

Table 31: Instruction Cache CP0 Register Interface

CP0 Registers	CP0 Number
Config1	16.1
CacheErr	27.0
ITagLo	28.0
ITagHi	29.0
IDataLo	28.1
IDataHi	29.1

Config1 Register (CP0 register 16, Select 1)

The $Config1_{IS}$ field (bits 24:22) indicates the number of sets per way in the instruction cache.

The $Config1_{IL}$ field (bits 21:19) indicates the line size for the instruction cache. The I7200 L1 instruction cache supports a fixed line size of 32 bytes as indicated by a default value of 4 for this field.

The $Config1_{IA}$ field (bits 18:16) indicates the set associativity for the instruction cache. The I7200 L1 instruction cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

CacheErr Register (CP0 register 27, Select 0)

The CacheErr register is a read-only register used to determine the status of a cache error. The upper two bits of this register ($CacheErr_{EREC}$) indicate whether the contents of the register pertain to an L1 instruction cache error, an L1 data cache error, a TLB error, or an external error. This register provides information such as:

- L1 data versus L2 data cache error
- Tag RAM versus Data RAM error
- External snoop request indication in multi-core systems

- Indicates coherent L1 cache error in another CPU in a multi-core system
- Fatal/non-fatal error indication
- Indicates if the error affects the Scratchpad RAM
- Indicates the cache index or Scratchpad RAM index of the double word entry where the error occurred

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

These registers are a staging location for cache tag information being read/written with cache load-tag/ store-tag operations.

The interpretation of this register changes depending on the setting of the $ErrCtl_{WST}$ and $ErrCtl_{SPR}$ bits.

- Default cache interface mode ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 0)
- Diagnostic "way select test mode" ($ErrCtl_{WST}$ = 1, $ErrCtl_{SPR}$ = 0)
- For scratchpad memory setup ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 1)

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

Staging registers for special cache instruction which loads or stores data from or to the cache line. Two registers (IDataHi, IDataLo) are needed, because the I7200 core loads I-cache data at least 64 bits at a time. This register stores the lower 32 bits of the load data.

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

Staging registers for special cache instruction which loads or stores data from or to the cache line. Two registers (IDataHi, IDataLo) are needed, because the I7200 core loads I-cache data at least 64 bits at a time. This register stores the upper 32 bits of the load data.

5.3.2 L1 Data Cache CP0 Register Interface

The I7200 core uses CP0 registers for data cache operations.

Table 32: Data Cache CP0 Register Interface

CP0 Registers	CP0 Number
Config1	16.1
CacheErr	27.0
DTagLo	28.2
DTagHi	29.1
DDataLo	28.3

Config1 Register (CP0 register 16, Select 1)

The $Config1_{DS}$ field (bits 15:13) indicates the number of sets per way in the data cache.

The $Config1_{DL}$ field (bits 12:10) indicates the line size for the data cache. The I7200 L1 data cache supports a fixed line size of 32 bytes as indicated by a default value of 4 for this field.

The $Config1_{DA}$ field (bits 9:7) indicates the set associativity for the data cache. The I7200 L1 data cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

CacheErr Register (CP0 register 27, Select 0)

The CacheErr register is a read-only register used to determine the status of a cache error. The upper two bits of this register ($CacheErr_{EREC}$) indicate whether the contents of the register pertain to an L1 instruction cache error, an L1 data cache error, a TLB error, or an external error.

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

These registers are a staging location for cache tag information being read/written with cache load-tag/ store-tag operations.

In a multi-core system, the D-cache has five logical memory arrays associated with this DTagLo register. The tag RAM stores tags and other state bits with special attention to the needs of the CPU. The duplicate tag RAM also stores tags and state, but is optimized for the needs of interventions. Both of these arrays are set-associative (4-way). The Dirty RAM and duplicate Dirty RAM store the dirty bits (indicating modified data) for CPU and intervention uses, and each combine their ways together in a single entry per set. The WS RAM combines the dirty and LRU data in a single entry per set. Accessing these arrays for index cache loads and stores is controlled by using three bits in the ErrCtl register to create modes that allow the correct access to these arrays.

The interpretation of this register changes depending on the settings of $ErrCtl_{WST}$, $ErrCtl_{DYT}$, and $ErrCtl_{SPR}$.

L1 Data Cache TagHi Register (CP0 register 29, Select 2)

The DTagHi register is used to store ECC error information (if ECC is supported) for the L1 data cache and DSPRAM memories. The bit assignments of the register depends on the type of memory being accessed. On a DSPRAM ECC error, bits 19:0 contain ECC error information. On a L1 data cache tag or data error, bits 16:10 contains data RAM ECC information, and bits 6:0 contain tag RAM error information.

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

In the I7200 core, software can read or write cache data using a cache index load tag/index store data instruction. Which word of the cache line is transferred depends on the low address fed to the cache instruction.

Note: The I7200 core does not implement the DDataHi register.

5.3.3 L2 Cache CM GCR Control Registers

The I7200 Coherency Manager (CM) uses the GCR registers for L2 cache operations. Note that these registers are located in CM address space. 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.

Note: Refer to the I7200 CM2.6 Registers HTML file for detailed information on the GCR registers.

5.4 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.

5.4.1 L1 Instruction Cache Initialization

The following sample boot code provides an example initialization routine for the instruction cache.

```
// ====== Initialize L1 instruction cache

li r4, 0x8        // invalidate whole icache
mtc0 r4, $22, 0   // C0_BCOP
ehb
```

5.4.2 L1 Data Cache Initialization

The following sample boot code provides an example initialization routine for the data cache.

```
// ====== Initialize L1 data cache
Initial_DCache:
    li      r4, 0x9    //invalidate whole dcache
    mtc0    r4, $22, 0 //C0_BCOP
    ehb
```

5.4.3 L2 Cache Initialization

The following sample boot code provides an example initialization routine for the L2 cache.

```
// ====== Initialize CM and L2 cache

#define GCR_L2SM_COP 0x0620
#define L2SM_COP_MODE_SHIFT 5
#define L2SM_COP_MODE_BITS 1
#define GCR_CONFIG_ADDR 0x1fbf8000     // Boot address of the GCR registers change as needed

li    t4, GCR_CONFIG_ADDR

li    t2, L2SM_COP_CMD_START          // NOTE command type is 0x0: Index WB Inv/Index Inv
                                      // (Full cache flush) and Start Command "1" is first
                                      // 2 bits
sw    t2, GCR_L2SM_COP(t4)            // Write L2 Configuration register
sync
lw    t3, GCR_L2SM_COP(t4)
ext   t3, t3, L2SM_COP_MODE_SHIFT, L2SM_COP_MODE_BITS
bnez  t3, L2_Running
```

5.5 Setting the Memory Space Cache Coherency

For non-MPU cores, 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 I7200 the CCA is set to coherent because all cached access for the I7200 are coherent.

```
LEAF(change_k0_cca)
   // NOTE! This code must be executed in KSEG1 (not KSGE0 uncached)
   // Set CCA for kseg0 to cacheable
   mfc0     $14, C0_CONFIG        // read C0_Config
   li       $15, 5                // CCA for coherent cores (fall through)

   set_kseg0_cca:
   ins       $14, $15, 0, 3       // insert K0
   mtc0      $14, C0_CONFIG       // write C0_Config
   jalrc.hb  zero, ra

   END(change_k0_cca)
```

For MPU Cores the cache attribute is set in the segment configuration registers:

```
#define CDMM_P_BASE_ADDR   0x1fc10000            // physical address of the CDMM Register
                                                 // (Change to suit your core)
#define MPU_CDMM_OFFSET   (64*3)
#define MPU_SegmentControl0 0x10
li   a0, CDMM_P_BASE_ADDR
li   a1, (0x2)<<24|(0x2)<<16|(0x2)<<8|(0x5)      // Segment 0 set to CWB (Cacheable, coherent,
                                                 // write-back, write-allocate, read misses
                                                 // request Shared)
sw   a1, MPU_CDMM_OFFSET+MPU_SegmentControl0(a0) // Segment 0 address 0~0x0FFFFFFF to cacheable
sync
```

6 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 (internal events), or external to the core (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 *Exception Processing Overview* on page 79.

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

6.1 Exception Processing Overview

The I7200 core includes support for three interrupt modes:

- *Interrupt Compatibility mode:* The behavior of the I7200 core is identical to the behavior of an implementation of Release 1 of the Architecture.
- *Vectored Interrupt (VI) mode:* Adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt.
- *External Interrupt Controller (EIC) mode:* 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 I7200 core defaults to Interrupt Compatibility mode.

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.

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

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.

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.

6.2 Exception Vector Locations

Historically in MIPS processors, the boot exception vector (BEV) is at a fixed location in both virtual and physical memory; it is mapped from a virtual address of 0xBFC0_0000 to a physical address of 0x1FC0_0000. For an MPU core, BEV is hard-coded into the core using the Configuration Options for CM Exception Vector setting in the core configuration GUI. The numbers in the following table are offsets to that address.

Table 33: Exception Vectors

The VPE has 2 modes: boot mode (offset shown in column 2) and normal running mode after the boot process has been completed (offset shown in column 3).

Exception Type	CP0[STATUS][BEV] = 1 Boot Exception Vector Offset	STATUS[BEV] = 0 CP0[EBase][WG] = 1 CP0[EBase][ExcBase] offset
Reset/NMI	0x0000	NA
TLB Miss	0x0200	0x0000
Cache Error	0x0300	0x0100 (CP0[Config5][CV] = 1)
General Exception	0x0380	0x0180
Interrupt (IV=1)	0x0400	0x0200
Debug 0	0x0480	NA
Debug 1	0xff200200	NA

Upon power-up of the I7200 core, the VPE is in boot mode. The CP0[Status][BEV] register's mode bit is preset to 1 at power on (BEV=1) so that when the core powers up VPE0 fetches the first instruction of the boot code from the Boot Exception Vector address. The boot code must be linked to start at this address.

All memory is uncached and directly mapped. Virtual program memory addresses from 0x0000 0000 to 0x7fff ffff are mapped directly to physical memory 0x0000 0000 to 0x7fff ffff. Only the lower 2 GB are available for use, therefore, the physical address on the boot device must be located in this range. The core can be switched to MPU mode at any time, usually during the boot process. In MPU mode, the VPE can access the complete 32-bit address range, (4 GB).

At power up (BEV=1), the exception vectors are offset from the Boot Exception Vector. The exception routines must be linked to the Boot Exception Vector offsets as shown in *Table 33: Exception Vectors* on page 80.

After the boot code finishes all of its tasks it usually sets CP0[Status][BEV]=0. The VPE then switches from using the Boot Exception Vectors to using Exception vectors starting at the Exception Base address. The boot code should program this Exception Base address into CP0[EBase][ExcBase].

The Exception Base address defaults to a restricted address range (corresponding to 2 memory segments, KSEG0 and KSEG1) for all MIPS processors. These memory segments range from 0x8000 0000 to 0xbfff ffff. This restricted range is enforced by gating the top 2 bits (30, 31) of the exception base address to 1 0 respectively. This setting is usually too restrictive for an MPU core, and can be overridden by clearing the Write Gate bit, CP0[EBase][WG]. Once this bit is cleared, bits 30 and 31 of CP0[EBase][ExcBase] are writable and the Exception Base address can be placed on a 4K boundary anywhere in memory.

The Cache Error exception vector also defaults to a restricted address for all MIPS processors. That address corresponds to an address located in the KSEG1 memory segment. KSEG1 is an uncached segment that cannot be changed to cacheable, which is appropriate for a Cache Error routine. The vector address is 0xa000 0100. This setting is usually too restrictive for a MPU core and can be overridden by clearing the Cache Vector bit CP0[Config5][CV]. Once the Cache Vector bit is cleared, the Cache Error vector is located at offset 0x100 from the Exception Base Address.

Note: The Cache Error Exception address must be uncached. The MPU Super segment that contains the Exception vectors could be set to cached. You should program a sub-region overlay to ensure that the cache error address range offset from 0x100 to 0x200 is uncached. See the MPU chapter for more details on region programming.

Related Concepts

Refer to the MPU chapter for more information on MPU programming. on page 47

6.3 Defining the Exception Vector Locations

The location of the exception vector in the I7200 core depends on the operating mode. If the core is in the legacy setting, the exception vector location is the same as in previous MIPS processors. However, if the core is configured for Enhanced Virtual Address (EVA), the exception vector can effectively be placed anywhere within kernel address space.

The EVAReset bit determines the addressing scheme and whether the device boots up in the legacy setting or the EVA setting. The legacy setting is defined as having the traditional MIPS virtual memory map used in previous generation processors. The EVA setting places the device in the enhanced virtual address configuration, where the initial size and function of each segment in the virtual memory map is determined from the segmentation control registers (SegCtl0 - SegCtl2).

If the EVAReset bit is not set at reset, the I7200 core comes up in the legacy configuration and hardware takes the following actions:

- The CONFIG5.K bit becomes read-write and is programmed by hardware to a value of 0 to indicate the legacy configuration. In this case, the cache coherency attributes for the kseg0 segment are derived from the Config.K0 field as described in the previous subsection. In addition to selecting the location of the cache coherency attributes, the CONFIG5.K bit also causes hardware to generate two boot exception overlay segments, one for kseg0 and one for kseg1.
- Hardware programs the CP0 memory segmentation registers (SegCtl0 - SegCtl2) for the legacy setting. Note that these registers are new in the I7200 core and are not used by legacy software. However, they are used by hardware during normal operation, so their default values should not bechanged.

If the EVAReset bit is asserted at reset, the I7200 core comes up in the EVA configuration (default is xkseg0 space = 3 GB) and hardware takes the following actions:

- The CONFIG5.K bit becomes read-only and is forced to a value of 1 to indicate the EVA configuration. In this case, the CONFIG.K0 field is ignored and is no longer used to determine the kseg0 cache coherency attributes (CCA). Rather, the values in bits 2:0 (segments 0, 2, and 4) and bits 18:16 (segments 1, 3, and 5) of the SegCtl0-SegCtl2registersare used to define the CCA for each memory segment. In this case, hardware generates only one BEV overlay segment.
- Hardware sets the CP0 memory segmentation registers (SegCtl0 - SegCtl2) for theEVA configuration.

When the LegacyUseExceptionBase bit is 0 and the Config5.K bit is cleared, the device is in legacy mode. In this mode the exception vector location defaults of 0xBFC0_0000 and the Core Local Reset Exception Base Register bits [31:12] are ignored.

When the LegacyUseExceptionBase is set and the Config5.K bit is cleared, the device is still in legacy mode, but the Core Local Reset Exception Base Register bits [29:12] are used to indicate the location of the exception vector. Bits 31:30 are forced to a value of 2'b10, placing the exception vector somewhere in kseg0/kseg1 space.

If the Config5.K bit is set, the device is in EVA mode. In this case theLegacyUseExceptionBase bit is ignored and the Core Local Reset Exception Base Register bits [31:12] are used to derive the location of the exception vector.

The function of the Config5.Kbit and the LegacyUseExceptionBase bit are shown in the following table. For more information on EVA mode, refer to *Memory Management Unit* on page 17.

Table 34: LegacyUseExceptionBase bit and CONFIG5.K Encoding

CONFIG5.K Bit	LegacyUse ExceptionBase bit	Condition	Action
0	0	Legacy Mode SI_ExceptionBase[31:12] pins are not used.	Use default BEV location of 0xBFC0_0000.
0	1	Legacy Mode Legacy Mode Use only Core Local Reset Exception Base Register bits [29:12] for the BEV base location. Bits 31:30 are forced to a value of 2'b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.	The BEV location is determined as follows: Core Local Reset Exception Base Register bits [31:12] = 2'b10, Core Local Reset Exception Base Register bits [29:12], 12'b0 Bits 31:30 are forced to a value of 2'b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.
1	Don't care	EVA Mode Use Core Local Reset Exception Base Register bits [31:12].	The Core Local Reset Exception Base Register bits [31:12] are used directly to derive the BEV location. The LegacyUseExceptionBase bit is ignored.

Another degree of flexibility in the selection of the vector base address, for use when StatusBEV equals 1, is provided via LegacyUseExceptionBase, Core Local Reset Exception Base Register bits [31:12], and Core-Local Reset Exception Extended Base Register BEVExceptionBaseMask field [27:20].

In the legacy setting, when the SI_UseExceptionBase pin is 0, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in *Table 35: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase bit Is Not Set* on page 83. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In the I7200 core, software is allowed to specify the vector base address via the EBase register for exceptions that occur when StatusBEV equals 0. *Table 35: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase bit Is Not Set* on page 83 shows the vector base address when the core is in legacy setting and the SI_UseExceptionBase pin is 0.

Table 36: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase Is Set on page 83 shows the vector base addresses when the core is in legacy setting and the LegacyUseExceptionBase bit is set. As can be seen in *Table 36: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase Is Set* on page 83, when the LegacyUseExceptionBase bit is set, the exception vectors for cases where StatusBEV = 0 are not affected.

Table 35: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase bit Is Not Set

Exception	Status$_{BEV}$[10]	
	0	1
Reset, NMI	0xBFC0.0000	
EJTAG Debug (with ProbEn = 0, in the EJTAG_Control_register and DCR.RDVec=0)	0xBFC0.0480	
EJTAG Debug (with ProbEn = 0, in the EJTAG_Control_register and DCR.RDVec=1)	DebugVectorAddr$_{[31:7]}$ || 2b0000000	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	EBase$_{31..30}$ || 1 || EBase$_{28..12}$ || 0x000 Note that EBase$_{31.\ 30}$ have the fixed value of 2b'10	0xBFC0.0300
Other	EBase31..12 || 0x000 Note that EBase31..30 have the fixed value of 2'b10	0xBFC0.0200

In legacy mode, when the LegacyUseExceptionBase bit is not set, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in the following table.

Table 36: Exception Vector Base Addresses: Legacy Mode, LegacyUseExceptionBase Is Set

Exception	Status$_{BEV}$[11]	
	0	1
Reset, NMI	0b10 || Core Local Reset Exception Base Register bits [29:12] || 0x000	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register and DCR.RDVec=0)	0b10 || Core Local Reset Exception Base Register bits [29:12] || 0x480	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register and DCR.RDVec=1)	DebugVectorAddr[31:7] || 2b0000000	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	EBase$_{31..30}$ || 1 || EBase$_{28..12}$ || 0x000 Note that EBase$_{31.\ 30}$ have the fixed value 2'b10. Exception vector resides in kseg1.	0b101 || Core Local Reset Exception Base Register bits [28:12] || 0x300 Exception vector resides in kseg1.
Other	EBase$_{31..12}$ || 0x000 Note that EBase$_{31.\ 30}$ have the fixed value 2'b10 Exception vector resides in kseg0/kseg1.	0b10 || Core Local Reset Exception Base Register bits [29:12] || 0x200 Exception vector resides in kseg0/ kseg1.

Table 37: Exception Vector Offsets on page 84 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 6.25* (on *page 346*) shows the offset

[10] **"||" denotes bit string concatenation.**
[11] **"||" denotes bit string concatenation.**

from the base address in the case where StatusBEV = 0 and CauseIV = 1. *Table 39: Exception Vectors* on page 84 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 IntCtlVS = 0.

Table 37: Exception Vector Offsets

Exception	Vector Offset
TLB Refill, EXL = 0	0x000
General Exception	0x180
Interrupt, CauseIV = 1	0x200 (In Release 3 implementations, this is the base of the vectored interrupt table when Status_{BEV} = 0)
Reset, NMI	None (uses reset base address)

In EVA mode, when the LegacyUseExceptionBase bit is ignored and the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a location determined by the programming of the three Segment Control registers (SegCtl0 - SegCtl2), as shown in the following table.

Table 38: Exception Vector Base Addresses — EVA Mode

Exception	Status_{BEV} [12]	
	0	1
Reset, NMI	Core Local Reset Exception Base Register bits [31:12] || 0x000	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register and DCR.RDVec=0)	Core Local Reset Exception Base Register bits [31:12] || 0x480	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register and DCR.RDVec=1)	DebugVectorAddr[31:7] || 2b0000000	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	$\text{EBase}_{31..12}$ || 0x000	Core Local Reset Exception Base Register bits [31:12] || 0x300 (Forced uncached)
Other	$\text{EBase}_{31..12}$ || 0x000	Core Local Reset Exception Base Register bits [31:12] || 0x200

Table 39: Exception Vectors

Exception	Config_K	Legacy Use Exception Base	Status_{BEV}[1]	Status_{EXL}	Cause_{IV}	EJTAG ProbEN	Vector (IntCtl_{VS} = 0)[14]
Reset, NMI	0	0	x	x	x	x	0xBFC0.0000
Reset, NMI	0	1	x	x	x	x	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x000

12 "||" denotes bit string concatenation.
13 'x' denotes don't care
14 || denotes bit string concatenation

Exception	Config$_K$	Legacy Use Exception Base	Status$_{BEV}$[1]	Status$_{EXL}$	Cause$_{IV}$	EJTAG ProbEN	Vector (IntCtl$_{VS}$ = 0)[14]
Reset, NMI	1	x	x	x	x	x	Core Local Reset Exception Base Register bits [31:12] || 0x000
EJTAG Debug	0	0	x	x	x	0	0xBFC0.0480 (if DCR.RDVec=0) DebugVectorAddr[31:7] || 2b0000000 (if DCR.RDVec=1)
EJTAG Debug	0	1	x	x	x	0	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x480 (if DCR.RDVec = 0) DebugVectorAddr[31:7] || 2b0000000 (if DCR.RDVec = 1)
EJTAG Debug	1	x	x	x	x	0	Core Local Reset Exception Base Register bits [31:12] || 0x480 (if DCR.RDVec = 0) DebugVectorAddr[31:7] || 2b0000000 (if DCR.RDVec = 1)
EJTAG Debug	x	x	x	x	x	1	0xFF20.0200
TLB Refill	x	x	0	0	x	x	EBase[31:12] || 0x000
TLB Refill	x	x	0	1	x	x	EBase[31:12] || 0x180
TLB Refill	0	0	1	0	x	x	0xBFC0.0200
TLB Refill	0	1	1	0	x	x	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x200
TLB Refill	1	x	1	0	x	x	Core Local Reset Exception Base Register bits [31:12] || 0x200
TLB Refill	0	0	1	1	x	x	0xBFC0.0380
TLB Refill	0	1	1	1	x	x	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x380
TLB Refill	1	x	1	1	x	x	Core Local Reset Exception Base Register bits [31:12] || 0x380
Cache Error	0	x	0	x	x	x	EBase[31:30] || 0b1 || EBase[28:12] || 0x100
Cache Error	1	x	0	x	x	x	0xBFC0.0100 (Config5CV = 0)
Cache Error	1	x	0	x	x	x	EBase[31:12] || 0x100 (Config5CV = 1)
Cache Error	0	0	1	x	x	x	0xBFC0.0300
Cache Error	0	1	1	x	x	x	2'b101 || Core Local Reset Exception Base Register bits [28:12] || 0x300
Cache Error	1	x	1	x	x	x	Core Local Reset Exception Base Register bits [31:12] || 0x300
Interrupt	x	x	0	0	0	x	EBase[31:12] || 0x180

[13] **'x' denotes don't care**

[14] **\|\| denotes bit string concatenation**

Exception	**Config$_K$**	**Legacy Use Exception Base**	**Status$_{BEV}$**[1]	**Status$_{EXL}$**	**Cause$_{IV}$**	**EJTAG ProbEN**	**Vector (IntCtl$_{VS}$ = 0)**[14]
Interrupt	x	x	0	0	1	x	EBase[31:12] || 0x200
Interrupt	0	0	1	0	0	x	0xBFC0.0380
Interrupt	0	1	1	0	0	x	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x380
Interrupt	1	x	1	0	0	x	Core Local Reset Exception Base Register bits [31:12] || 0x380
Interrupt	0	0	1	0	1	x	0xBFC0.0400
Interrupt	0	1	1	0	1	x	2'b10 || SI_ExceptionBase[29:12] || 0x400
Interrupt	1	x	1	0	1	x	Core Local Reset Exception Base Register bits [31:12] || 0x400
All others	x	x	0	x	x	x	EBase[31:12] || 0x180
All others	0	0	1	x	x	x	0xBFC0.0380
All others	0	1	1	x	x	x	2'b10 || Core Local Reset Exception Base Register bits [29:12] || 0x380
All others	1	x	1	x	x	x	Core Local Reset Exception Base Register bits [31:12] || 0x380

6.4 Core-Level Exception Priorities

The following table 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 40: Priority of Exceptions

Cause.ExcCode Field Encoding		**Exception**	**Description**
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.
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.

[13] 'x' denotes don't care
[14] || denotes bit string concatenation

Cause.ExcCode Field Encoding		Exception	Description
Decimal	Hex		
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 Send to Group 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.
N/A	N/A	DDBLImpr DDBSImpr	Debug Data Break Load/Store. Imprecise. When this exception occurs, hardware sets the CP0 Debug.DDBLImpr bit if the error occurred during a load, or the Debug.DDBSImpr bit if the error occurred during a store.
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.
0	0x00	Interrupt	An enabled interrupt occurred.
23	0x17	Deferred Watch	A 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, hardware sets the WP bit in the CP0 Cause register. In addition, hardware sets the I, R, or W bits in the CP0.WatchHi register depending on whether the exception occurred during a fetch (I), a load (R), or a store (W).
N/A	N/A	DIB	A 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 DIP bit of the CP0 Debug register.
23	0x17	WATCH - Instruction Fetch	A watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.
4	0x04	AdEL	Instruction fetch address alignment error. A non-word-aligned address was loaded into the PC in the current mode.
2	0x02	TLBL/XTLBL Refill - instruction fetch or load	TLB/XTLB refill - Instruction fetch or load. A TLB miss occurred on an instruction fetch or a data load.
29	0x1d	MPUL	Fetch MPU miss. This exception is at the same priority level as TLBL above and is only taken in an MPU is implemented.
20	0x14	TLBXI	TLB Execute Inhibit. An instruction fetch matched a valid TLB entry which had the XI bit set.

Cause.ExcCode Field Encoding		Exception	Description
Decimal	Hex		
30	0x1E	I-cache Error - instruction fetch	A Cache error occurred on an instruction fetch.
6	0x6	IBE	From Instruction Fetch Unit (IFU) bus error.
30	0x1E	D$/L2$ Error	Both of these errors are signaled as data cache errors.
7	0x07	DBE	Bus error signaled on load/store (imprecise)
N/A	N/A	SDBBP	A debug breakpoint (SDDBP instruction) was executed. When a SDBBP exception occurs, hardware writes the DBp bit of the CP0 Debug register.
8	0x08	Sys (Validity exception) [15]	Execution of SYSCALL instruction.
9	0x09	Bp (Validity exception)[15]	Execution of BREAK instruction.
11	0x0B	CpU (Validity exception)[15]	Execution of a coprocessor instruction for a coprocessor that is not enabled. The I7200 core supports the CP0 and CP1 coprocessors.
26	0x1A	DSPDis (Execution exception)[16]	DSP ASE state disabled.
10	0x0A	RI (Execution exception)[16]	Execution of a Reserved Instruction.
15	0x0F	FPE (Execution exception)[16]	Floating Point exception. [17]
0x11	0x0B	C2E (Execution exception)[16]	Coprocessor 2 unusable exception.
16	0x10	ISI (Execution exception)[16]	Implementation specific Coprocessor 2 exception.
12	0x0C	Ov (Execution exception)[16]	Execution of an arithmetic instruction that overflowed.
13	0x0D	Tr (Execution exception)[16]	Execution of a trap (when trap condition is true).
25	0x19	MT_ov (Execution exception)[16]	Thread overflow condition, where a TC allocation request cannot be satisfied.
25	0x19	MT_under (Execution exception)[16]	Thread underflow condition, where the termination and deallocation of a thread leaves no TCs activated on a VPE.
25	0x19	MT_invalid (Execution excep- tion)[16]	Invalid qualifier condition, where a YIELD instruction specifies an invalid condition for resuming execution.
25	0x19	MT_yield_sched (Execution exception)[16]	YIELD scheduler exception condition, where a valid YIELD instruction could have caused a rescheduling of a TC, and the YIELD intercept bit is set.

15 All of the Validity exceptions have the same priority level.
16 All of the execution exceptions have the same priority.
17 The I7200 core does not support the FPU.

Cause.ExcCode Field Encoding		Exception	Description
Decimal	Hex		
N/A	N/A	DDBL / DDBS	Precise Debug Data Address 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 address 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.
23	0x17	WATCH - data access	A watch address match was detected on the address referenced by a load or store.
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.
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.
2	0x02	TLBL/XTLBL refill - data access	Load TLB miss. A TLB miss occurred on a data access.
29	0x1D	MPUL	Load MPU miss. This exception is at the same priority level as the TLBL load miss above and is only taken if an MPU is implemented.
3	0x03	TLBS	Store TLB miss. Store TLB hit to page with V=0. This exception is at the same priority level as the MPUS load miss below and is only taken if an MMU is implemented.
29	0x1D	MPUS	Store MPU miss. This exception is at the same priority level as the TLBS load miss above and is only taken if an MPU is implemented.
19	0x13	TLBRI	TLB Read Inhibit. Occurs when there is an attempt to access a page table whose RI bit is set.
1	0x1	TLB Mod	Store to TLB page with D = 0.
25	0x19	MT_GSS (Thread exception	Gating storage scheduler exception, where a gating storage load or store would have been blocked and caused a rescheduling or a TC, and the GS intercept bit is set. Note that both thread exception have the same priority.
25	0x19	MT_GS	(Thread exception) Gating storage exception condition, where implentation-dependent logic associated with gating or inter-thread communication (ITC) storage requires software intervention. Note that both thread exception have the same priority.

6.5 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 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 general 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.

6.6 Debug Exception Processing

All debug exceptions have the same basic processing flow

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted.
- The*DSS*, *DBp*, *DDBL*, *DDBS*, *DIB,* and *DINT* bits in the *Debug* register are updated appropriately, depending on the debug exception type.
- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.
- The *DM* bit in the *Debug* register is set to1.
- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case.

A unique debug exception is indicated through the *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB* and *DINT* bits (D* bits [5:0]) in the *Debug* register.

The location of the debug exception vector is determined by the ProbTrap bit in the OCI Control register (OCR) and the RDVec bit in the Debug Control register (DCR), as shown in in the following table.

Table 41: Debug Exception Vector Address

ProbTrap bit in OCR Register	RDVec bit in DCR Register	Debug Exception Vector Address
0	0	0480 (offset from the Boot Exception Vector address)
0	1	DebugVectorAddr31..7 || 0000000
1	0	0xFF20 0200 in dmseg
1	1	

The value in the optional drseg register DebugVectorAddr (offset 0x00020) is used as the debug exception vector when the OCR ProbTrap bit is 0 and when enabled through the optional RDVec control bit in the Debug Control Register (DCR).

Table 42: DebugVectorAddr Register Format

31	30	29 7	6 1	0
1	0	DebugVectorOffset	0	IM

Table 43: DebugVectorAddr Register Field Descriptions

Name	Bits	Description	Read/Write	Reset State
1	31	Ignored on write; returns one on read.	R	1

Name	Bits	Description	Read/Write	Reset State
DebugVectorOffset	29:7	Programmable Debug Exception Vector Offset.	R/W	Preset to 0x7F8009
IM	0	This bit is ignored for nanoMIPS, which does not have ISA mode.	R	0
0	30,6:1	Ignored on write; returns zero on read.	R	0

Bits 31..30 of the DebugVectorAddr register are fixed with the value 0b10, and the addition of the base address and the exception offset is done inhibiting a carry between bit 29 and bit 30 of the final exception address. The combination of these two restrictions forces the final exception address to be in the kseg0 or kseg1 unmapped virtual address segments. For cache error exceptions, bit 29 is forced to a 1 in the ultimate exception base address, so that this exception always runs in the kseg1 unmapped, uncached virtual address segment.

6.7 Interrupt Mode Code Examples

The I7200 supports three interrupts modes: interrupt compatibilty mode, Vectored Interrupt (VI) mode, and External Interrupt Controller (EIC) mode. The following subsections show provide examples of interrupt handlers for each of these modes.

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 BEV = 0).

The following core shows 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 */
```

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:

- $\text{Config3}_{\text{VInt}} = 1$
- $\text{Config3}_{\text{VEIC}} = 0$
- $\text{IntCtl}_{\text{VS}} \neq 0$
- $\text{Cause}_{\text{IV}} = 1$
- $\text{Status}_{\text{BEV}} = 0$

In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer, performance counter, and fast debug channel interrupts are combined in a system-dependent way (external to the CPU) with the hardware interrupts (the interrupt with which they are combined is indicated by the *IntCtlIPTI*/*IPCI*/*IPFDCI* fields) to provide the appropriate relative priority of the those interrupts with that of the hardware interrupts. The processor interrupt logic ANDs each of the *CauseIP* bits with the corresponding *StatusIM* bits. If any of these values is 1, and if interrupts are enabled (*StatusIE* = 1, *StatusEXL* = 0, and *StatusERL* = 0), an interrupt is signaled and a priority encoder scans the values in the order shown in the following table.

Table 6.24 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated from	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
Lowest Priority		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
	Software	SW1	IP1 and IM1	1
		SW0	IP0 and IM0	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 */
/* NOTE: K0 and K1 should not be used until interrupts are disabled again */
/* 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 */
```

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. The priority is based on the interrupt number: a higher number indicates a higher priority. Interrupt numbers range from 1 to 63, so the EIC can send 63 possible interrupts. A 0 in the RIPL field of the Cause register means an interrupt is pending. A 0 in the IPL field in the status register represents no interrupts are masked.

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

- $\text{Config3}_{\text{VEIC}} = 1$
- $\text{IntCtl}_{\text{VS}} \neq 0$
- $\text{Cause}_{\text{IV}} = 1$
- $\text{Status}_{\text{BEV}} = 0$

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 `IVexception` 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 $_{\text{RIPL}}$ to Status $_{\text{IPL}}$ 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.
 */
```

Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with *IntCtl*VS to create the interrupt offset, which is added to 0x200 to create the exception vector offset. For VI mode, the vector number is in the range 0..7, inclusive. For EIC interrupt mode, the vector number is in the range 1..63, inclusive (0 being the encoding for “no interrupt”). The *IntCtl*VS field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility mode. A non-zero value enables vectored interrupts. The following table shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl*VS field.

Table 44: Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of $IntCtl_{VS}$ Field				
	5'b00001	5'b00010	5'b00100	5'b01000	5'b10000
0	0x0200	0x0200	0x0200	0x0200	0x0200
1	0x0220	0x0240	0x0280	0x0300	0x0400
2	0x0240	0x0280	0x0300	0x0400	0x0600
3	0x0260	0x02C0	0x0380	0x0500	0x0800
4	0x0280	0x0300	0x0400	0x0600	0x0A00
5	0x02A0	0x0340	0x0480	0x0700	0x0C00
6	0x02C0	0x0380	0x0500	0x0800	0x0E00
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
• • •					
61	0x09A0	0x1140	0x2080	0x3F00	0x7C00
62	0x09C0	0x1180	0x2100	0x4000	0x7E00
63	0x09E0	0x11C0	0x2180	0x4100	0x8000

The general equation for the exception vector offset for a vectored interrupt is:

vectorOffset $\leftarrow$ 0x200 + (vectorNumber × ($IntCtl_{VS}$ || 0b00000))

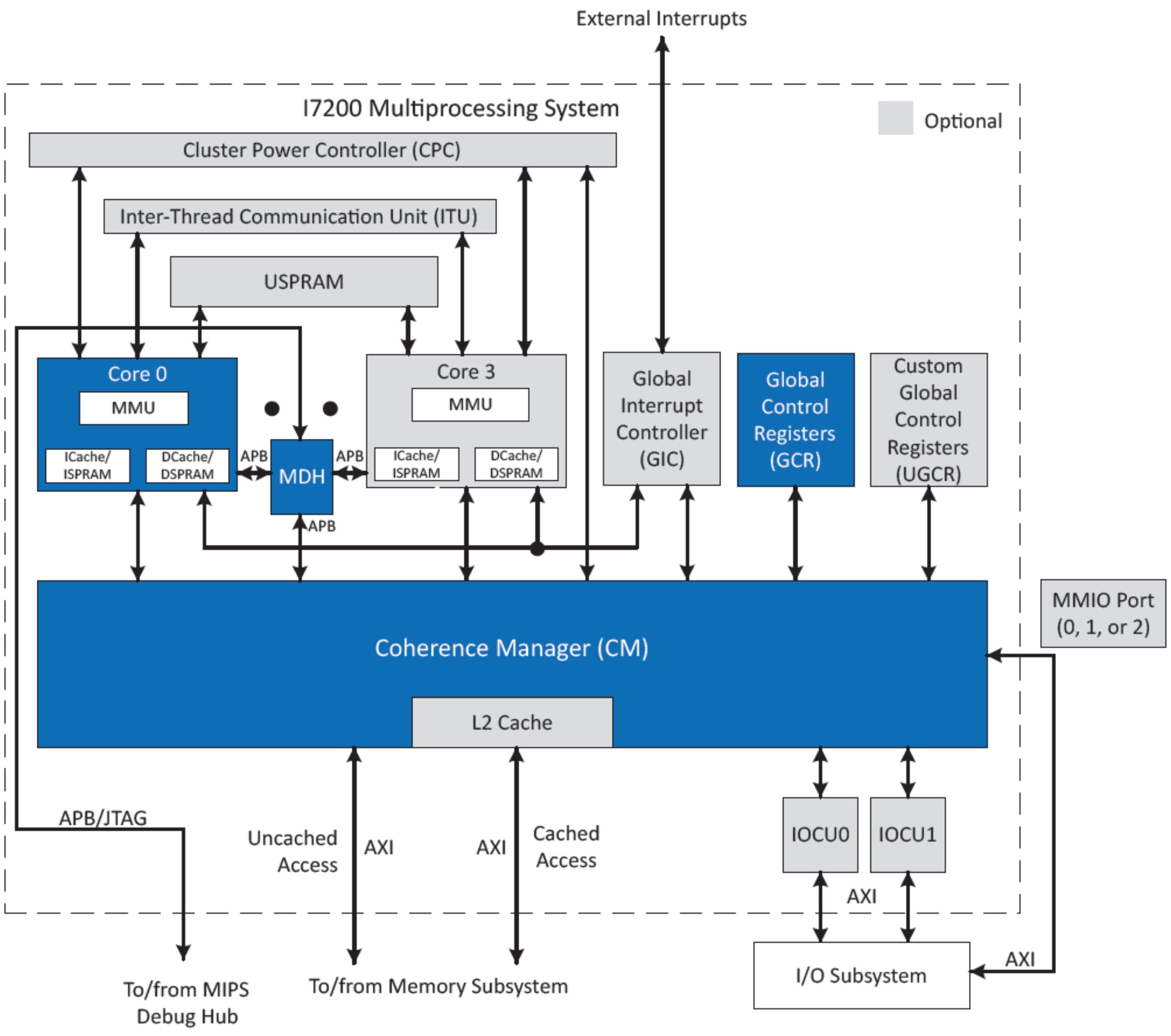
7 Coherence Manager

The Coherence Manager (CM) communicates with all cores and other devices in the I7200 Multiprocessing System (MPS), as well as coherent devices external to the I7200 MPS, to achieve system-wide coherence.

A MESI-based coherence protocol is used to maintain coherence among the L1 data caches of each I7200 core, the L2 cache and I/O coherence units (IOCUs).

The CM contains numerous ports that allow the various system peripherals to communicate with the CM. The ports connected to the CM are shown in the following figure.

Figure 27: CM Interface Ports



This chapter covers the Coherence Manager's Global Control Block. Other blocks are covered in separate chapters.

7.1 Global Control Registers (GCR)

The GCR register block contains memory mapped registers that provide:

- System information
- Settings that control the behaviour of the system
- Programmable address for the other controllers in the system such as the Cluster Power Controller (CPC), Global interrupt Controller (GIC) and User Defined Custom region .

GCR memory-mapped registers are accessed in root mode. All registers in the Global Control Block are 32 bits wide and should only be accessed using 32-bit uncached loads and stores. Reads from unpopulated registers in the GCR address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

The rest of this chapter describes how to perform programming tasks using the GCRs. The code examples use include files (`cm2.h` and `m32c0.h`) provided with MIPS Codescape SDK GNU tools. They can be found in the `include/mips` directory.

GCR Base Address

The core implementer sets the GCR Base Address, which can be anywhere in the memory map that is aligned to a 32 Kbyte boundary. The address defaults to read/write but can be forced to read only mode by the core implementer. The CMGCR Base Register (CP0 Register 15, Select 3) is a mirror of the GCR_BASE Register that is located within the GCR register space at offset 0x0008.

Table 45: CMGRBase Register

Register Fields		CMGCRBase Register	Reset State
Name	Bits	(CP0 Register 15, Select 3)	
CMGCR_BASE_ADDR	27:11	Bits 31:15 of the base physical address of the memory mapped Coherence Manager GCR registers. This register field reflects the value of the GCR_BASE field within the memory-mapped Coherence Manager GCR Base Register. The number of implemented physical address bits is implementation-specific. For the unimplemented address bits - writes are ignored, returns zero on read.	Preset

The following example code reads the value of the CMGCR:

```
#define C0_CMGCR  $15,3
#define getcmgcrg() \
({ unsigned int __value; \
__asm__ __volatile ("mfc0 %0, $%1, 0" : "=d" (__value) : "i" (C0_CMGCR)); \
__value;})
unsigned int x;
x = getcmgcrg() ;
```

GCR Address Blocks

The GCR address space has a total size of 32 Kbytes, which is divided into 8 Kbyte blocks as shown in the table below. These sub-blocks are described in detail later in this chapter.

Table 46: GCR Sub-Blocks

GCR Base Offset	Description
0x0000 - 0x1FFF	Global Control Block. Contains registers pertaining to the global system functionality.
0x2000 - 0x3FFF	Core-Local Control Block (aliased for each CPU core). Contains registers pertaining to the core issuing the request. Each CPU has its own copy of registers within this block.
0x4000 - 0x5FFF	Core-Other Control Block (aliased for each CPU core). This block of addresses gives each Core a window into another CPU's Core-Local Control Block. Before accessing this space, the Core-Other Addressing Register in the Core Local Control Block must be set with the CoreNum of the target core.
0x6000 - 0x7FFF	Global Debug Block. Contains global registers useful in debugging the MPS.

For more details, refer to *#unique_72/unique_72_Connect_42_table_core_other_address*

7.2 Programming the GCRs

The Global Control Block is a set of memory-mapped controller registers that are used to configure and control various aspects of the coherence scheme and Coherence Manager. This block:

- Provides information on system configuration.
- Configures address map locations of the Global Interrupt Controller, Cluster Power Controller, and the Custom Region.
- Configures the non-coherent areas of memory to main memory or MMIO devices.
- Controls the coherency of the default shared memory region.
- Controls the handling and reporting of Coherence Manager errors.
- Controls other options of the Coherence Manager.

The Global Control Block registers are referred to as the Global Configuration Registers or GCR.

7.2.1 Finding the Number of Regions, IOCUs, and Cores in the System

The first register in the global block section is the global configuration register. This register is a read-only register that gives you information on the configuration of your system as shown in the table below.

Table 47: Global Configuration Register

Register Fields		Global Configuration Register		Reset State
Name	Bits	(GCR_CONFIG Offset 0x0000)		
NUMMMIO	22:20	Total number of MMIO ports in the system. 0 - 2 MMIOs are currently supported.		IP Configuration Value
NUM_ADDR_REGIONS	19:16	Total number of CM Address Regions. Note: only 0, 4, 6, or Address Regions are currently supported.		IP Configuration Value
		Encoding	**Meaning**	
		0x0	0 Address Regions (IOCU)	
		0x4	4 Address Regions - Standard	

Register Fields		**Global Configuration Register**		**Reset State**
Name	**Bits**	**(GCR_CONFIG Offset 0x0000)**		
		0x6	6 Address Regions - 4 Standard plus 2 Attribute Only	
		0x8	8 Address Regions - 4 Standard plus Attribute Only	
NUMIOCU	11:8	Total number of IOCUs in the system. Note: 0 - 2 IOCU are currently supported.		IP Configuration Value
PCORES	7:0	Total number of cores in the system, not including the IOCUs.		IP Configuration Value

The following example code shows how to get PCORES into register t2 and NUMIOCU into register t3.

```
li    t0, GCR_CONFIG_ADDR
lw    t1, GCR_CONFIG(t0)                    // Load GCR_CONFIG
ext   t2, t1, PCORES_SHIFT, PCORES_BITS     // Extract PCORES
ext   t3, t1, NUMIOCU_SHIFT, NUMIOCU_BITS   // Extract NUMIOCU.
```

7.2.2 Restricting Access to GCRs from Cores

The Global GCR Access Privilege Register's CM_ACCESS_EN field is used to restrict access to the GCRs from a core. There are 6 requestors that could access the GCRs. Bits 0 – 3 correspond to cores 0 – 3. Bits 4 and 5 correspond to IOCU 0 and 1.

Table 48: Global GCR Access Privilege Register

Register Fields		**Global GCR Access Privilege Register**	**Reset State**
Name	**Bits**	**(GCR_ACCESS Offset 0x0020)**	
CM_ACCESS_EN	7:0	Each bit in this field represents a coherent requester. If the bit is set, that requester is able to write to the GCR registers (this includes all registers within the Global, Core-Local, Core-Other, and Global Debug control blocks. The GIC is always writable by all requestors). If the bit is clear, any write request from that requestor to the GCR registers (Global, Core-Local, Core-Other, or Global Debug control blocks) will be dropped.	0xff

The following example code shows how to restrict access to core 0.

```
li   t0, GCR_CONFIG_ADDR
li   t1, 1
sw   t1, GCR_ACCESS(t0)        // GCR_ACCESS
```

7.2.3 Programming Controller Base Addresses

There are 3 controllers: Global Interrupt Controller, Cluster Power Controller, and an optional Custom controller. The base address of these controllers is programmed into GCR registers discussed in this section. These can be placed anywhere in physical memory naturally aligned to their size.

Programming the Global Interrupt Controller Base Address

The GIC is an optional component. To see if your system contains a GIC, read the GIC_EX field in the Global Interrupt Controller Status Register. Because the GIC_EX field is the only field and 1 indicates that there is a GIC, simply check whether the register value is 0.

Table 49: GIC Status Register

Register Fields		GIC Status Register	Reset State
Name	**Bits**	**(GCR_GIC_STATUS Offset 0x00D0)**	
GIC_EX	0	If this bit is set, the GIC is connected to the CM.	1

The GIC base address and enable bit are located in the GIC base address register.

Table 50: GIC Base Address Register

Register Fields		GIC Base Address Register	Reset State
Name	**Bits**	**(GCR_GIC_BASE Offset 0x0080)**	
GIC_BASE_ADDR	31:17	This field sets the physical base address of the 128 KB GIC.	0
GIC_EN	0	If this bit is set, the GIC address region is enabled. This bit cannot be set to 1 if GIC_EX = 0, indicating that a GIC is not attached to the CM.	0

The following example code performs a GIC check and sets the base address.

```
// Change the next line to a value you want the GIC controller to be (128 KB aligned)
#define GIC_P_BASE_ADDR   0x1bdc0000     // physical address of the GIC

li      t0, GCR_CONFIG_ADDR
lw      t1, GCR_GIC_STATUS(t0)
beqzc   t1, no_gic                       // branch around set base address if 0
// set the GIC address
li      t1, GIC_P_BASE_ADDR | 1          // Physical address + enable bit
sw      t1, GCR_GIC_BASE (t0)
no_gic:
```

Programming the Cluster Power Controller Base Address

The CPC is an optional component. To see if your system contains a CPC, read the CPC_EX field in the GIC Status Register. Because the CPC_EX field is the only field and 1 indicates that there is a CPC, simply check whether the register value is 0. The CPC base address and enable bit are located in the Cluster Power Controller base address register.

Table 51: Cluster Power Controller Address Register

Register Fields		Cluster Power Controller Base Address Register	Reset State
Name	**Bits**	**(GCR_CPC_BASE Offset 0x0088)**	
CPC_BASE_ADDR	31:15	This field sets the base address of the 32K Cluster Power Controller.	0
CPC_EN	0	If this bit is set, the CPC address region is enabled. This bit cannot be set if 1 CPC_EX = 0, indicating that a CPC is not attached to the CM.	0

The following example code performs a CPC check and sets the base address.

```
// Change the next line to a value you want the CPC controller to be (32 KB aligned)
#define CPC_P_BASE_ADDR  0x1bde0000       // physical address of the CPC

li       t0, GCR_CONFIG_ADDR
lw       t1, GCR_CPC_STATUS(t0)
beqzc    t1, no_cpc                       // branch around set base address if 0
// set the CPC address
li       t1, CPC_P_BASE_ADDR | 1          // Physical address + enable bit
sw       t1, GCR_CPC_BASE(t0)
no_cpc:
```

Programming the Global Custom Base Register

The CM supports an optional Custom (user-defined) GCRs. To see if your system contains Custom GCR registers, read the GGU_EX field in the Custom Status Register. Because the GGU_EX field is the only field and a 1 indicates that there are Custom GCR registers, simply check whether the register value is 0. The Custom base address and enable bit are located in the Custom base register.

Table 52: GCR Custom Base Register

Register Fields		GCR Custom Base Register	Reset State
Name	**Bits**	**(GCR_CUSTOM_BASE Offset 0x0060)**	
GCR_CUSTOM_BASE	31:16	This field sets the base address of the 64 KB GCR custom user-defined block of the I7200 MPS.	Undefined
GGU_EN	0	If this bit is set, the Custom GCR address region is enabled. This bit cannot be set to 1 if GGU_EX = 0, indicating that a custom GCR is not attached to the CM.	0

The following example code performs a custom base register check and sets the base address.

```
// Change the next line to a value you want the Custom Base Registers to be (64 KB aligned)
#define GGU_P_BASE_ADDR   0x1bdf0000     // physical address of the GGU

li      t0, GCR_CONFIG_ADDR
lw      t1, GCR_CUSTOM_STATUS (t0)
beqz    t1, no_gcr                     // branch around set base address if 0
// set the GGU address
li      t1, GGU_P_BASE_ADDR | 1        // Physical address + enable bit
sw      t1, GCR_CUSTOM_BASE(t0)
no_gcr:
```

Programming MMIO Address Regions

An Uncached or Uncached Accelerated (UCA) request from CPU cores can be mapped to Memeory Mapped I/O (MMIO) using the address region registers (GCR_REG*_BASE and GCR_REG*_MASK) in the CM GCR block. An MMIO request initiated by an I7200 CPU that targets a MMIO port is routed from the core, to the CM, to the MMIO port. The corresponding responses are passed from the MMIO port back to the CM and back to the requesting core. When the CM routes out the MMIO requests from the CPU cores to a target MMIO port, CM hardware maintains the order of read and write requests for each CPU core.

If the system does not have MMIO, the request goes to the main memory, regardless of the region configuration.

Note: The MMIO address space is expected to be accessed using an Uncached or Uncached Accelerated (UCA) CCA. I7200 CPU requests targeting MMIO are routed through the uncached pipeline of the CM and the CM sends those requests out on the external MMIO AXI port. The corresponding responses from the MMIO bus are captured by the CM. The CM then routes those responses back to the requesting CPU core.

Programming Address Regions

Programming the Default Address Region

Requests with addresses that do not map to any of the registers blocks within CM or with the ranges in address region registers fall into the default address region. For the default address region, the GCR base register determines whether the cache coherency attribute default override is enabled. It also determines the CCA if the override is enabled and the memory port (memory or MMIO) that is associated with the default region.

Table 53: GCR Base Register

Register Fields		GCR Base Register		Reset State
Name	**Bits**	**(GCR_BASE Offset 0x0008)**		
CCA_DEFAULT_ OVERRIDE_VALUE	7:5	Used with CCA_DEFAULT_OVERRIDE_ENABLE to force the CCA value for transactions on the L2/system memory interface. Refer to the CCA_DEFAULT_OVERRIDE_ENABLE field. This field is only applicable for the cached pipe in dual-pipe CM configurations		0
		Encoding	**Meaning**	
		0x0	Write through	
		0x2	Uncached	
		0x3	Write-back non-coherent	
		0x4	Mapped to WB	
		0x5	Mapped to WB	
		0x7	Uncached accelerated	
CCA_DEFAULT_ OVERRIDE_ENABLE	4	If the CCA_DEFAULT_OVERRIDE_ENABLE is set to 1 and CM_DEFAULT_TARGET is set to memory, transactions with addresses that do not map to any region have a CCA value set to the CCA_DEFAULT_OVERRIDE_VALUE when driven to the L2 or memory. This field is only applicable for the cached pipe in dual-pipe CM configurations.		0
CM_DEFAULT_TARGET	1:0	Determines the target device for the default memory region. This is only applicable for the uncached pipe in dual-pipe CM configurations. If there are no MMIO ports present in the system, the target is always memory.		Signal SI_CM_DEFAULT _TARGET
		Encoding	**Meaning**	
		0	Memory	
		2	First MMIO	
		3	Second MMIO	

The following example code programs the default region to disable the L2 cache.

```
li    t0, GCR_CONFIG_ADDR
lw    t1, GCR_BASE(t0)         // Read GCR_BASE
li    t2, 0x50                 // Enable CCA and set to uncached
ins   t1, t2, 0, 8             // Insert bits
sw    t1, GCR_BASE(t0)         // Write GCR_BASE
```

The following example code programs the default region to enable to L2 cache.

```
li    t0, GCR_CONFIG_ADDR
lw    t1, GCR_BASE(t0)         // Read GCR_BASE
ins   t1, zero, 0, 8           // CCA Override disabled
sw    t1, GCR_BASE(t0)         // Write GCR_BASE
```

Programming Variable Size Address Regions

There can be up to four programmable variable size address regions for mapping the MMIO and memory. Like the default region, these regions can be used to override the CCA value between the L2 cache and memory. The number of regions is determined at IP configuration time. If MMIO is not present, the regions registers can still be used for CCA overrides to main memory. There are two registers for each address region, a base address register and an address mask register. The following table shows the offsets from the GCR base address for the four possible regions.

Table 54: Address Region Register Offsets

Region	Base Address Register Offset	Address Mask Register offset
0	0x0090	0x0098
1	0x00A0	0x00A8
2	0x00B0	0x00B8
3	0x00C0	0x00C8

The region base address register is used to program the region address. The base address is aligned to a 64 Kbyte boundary so the lower 16 bits of the address are always 0.

Table 55: Region Base Address Register

Register Fields		Region Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)	Reset State
Name	Bits		
CM_REGION_BASE_ADDR	31:16	This field sets the base address of the Coherence Memory Region.	0

The region address mask register is used to program the size of the region, the CCA override, and the device (MMIO or memory).

Table 56: Region Address Mask Register

Register Fields		Region Address Mask Register (GCR_REGx_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)		Reset State
Name	Bits			
CM_Region_Address_Mask	31:16	This field is used to set the size of the CM region. The only allowed values in this register are contiguous sets of leading 0x1s. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xfff0 is allowed, but the value 0xffef is not allowed).		Undefined
CCA_OVERRIDE_VALUE	7:5	Used with CCA_OVERRIDE_ENABLE to force the CCA value for transactions on the L2 or system memory interface. This field is only applicable for the cached pipe in dual-pipe CM configurations.		0
		Encoding	**Meaning**	
		0x0	Write through	
		0x2	Uncached	
		0x3	Write-back	
		0x4	Mapped to WB (CWBE)	
		0x5	Mapped to WB (CWB)	
		0x7	Uncached accelerated	

Register Fields		Region Address Mask Register	Reset State
Name	**Bits**	**(GCR_REGx_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)**	
CCA_OVERRIDE_ENABLE	4	If CCA_OVERRIDE_ENABLE is set to 1 and CM_REGION_TARGET is set to memory, transactions with addresses that map to this region have a CCA value set to CCA_OVERRIDE_VALUE when driven to the L2 or memory. This field is only applicable for the cached pipe in dual-pipe CM configurations.	0
CM_Region_TARGET	1:0	Maps this region to the specified device. This field is only applicable for the uncached pipe in dual-pipe CM configurations. **Encoding** | **Meaning** 0 | Disabled 1 | memory 2 | First MMIO 3 | Second MMIO	0

CM_Region_Address_Mask determines the size of the region. This field is used along with its equivalent CM region base address register. The request physical address (result of a load, store or fetch) is logically ANDed with the value of this register and ANDed with the value of the region's base address register to see if the requested address is in range of this region. If both ANDed results match, the request is routed to its CM_Region_TARGET.

The CCA_OVERRIDE_VALUE overrides the CCA for the L2 to memory if the CCA_OVERRIDE_ENABLE is also set.

The CM_Region_TARGET determines the device that the request is directed to memory or MMIO.

Programming Attribute-Only Address Regions

Attribute-only regions allow the cache coherency attributes for that region to be modified, but they cannot be used to select between memory and MMIO as the target.

0, 2, or 4 variable size attribute-only addresses can override the CCA value driver to the L2 cache and memory. The number of attribute-only regions is determined at IP configuration time. Each region has two registers: the attribute-only base address register and the attribute-only address mask register. The following table shows the offsets from the attribute-only GCR base address for the 4 possible regions.

Table 57: Attribute-Only Region Register Offsets

Region	Base Address Register Offset	Address Mask Register Offset
0	0x0190	0x0198
1	0x01A0	0x01A8
2	0x0220	0x0218
3	0x0220	0x0228

The Attribute-Only Region Base Address Register is used to program the base address. The base address is aligned to a 64 Kbyte boundary so the 16 bits of the address are always 0.

Table 58: Attribute-Only Region Address Register

Register Fields		Attribute-Only Region Address Register	Reset State
Name	**Bits**	**(GCR_REGn_ATTR_BASE Offsets 0x0190, 0x01A0, 0x0210, 0x0220)**	
CM_REGION_BASE_ADDR	31:16	This field sets the base address of the Coherence Memory Region.	0

The Attribute-Only Region Address Mask register is used to program the size of the region and the CCA override.

Table 59: Attribute-Only Region Address Mask Register

Register Fields		Attribute-Only Region Address Mask Register	Reset State
Name	**Bits**	**(GCR_REGn_ATTR_MASK Offsets 0x0198, 0x1A8, 0x218, 0x228)**	
CM_REGION_ADDR_MASK	31:16	This field is used to set the size of the CM region. The only allowed values in this register are contiguous sets of leading 0x1s. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xfff0 is allowed, but the value 0xffef is not allowed). This field is only applicable for the cached pipe in dual-pipe CM configurations.	Undefined
CCA_OVERRIDE_VALUE	7:5	Used with CCA_OVERRIDE_ENABLE to force the CCA value for transactions on the L2 or system memory interface. **Encoding** / **Meaning** 0x0 / Write through 0x2 / Uncached 0x3 / Write-back 0x4 / Mapped to WB (CWBE) 0x5 / Mapped to WB (CWB) 0x7 / Uncached accelerated	0
CCA_OVERRIDE_ENABLE	4	If CCA_OVERRIDE_ENABLE is set to 1 and CM_REGION_TARGET is set to memory, transactions with addresses that map to this region have a CCA value set to CCA_OVERRIDE_VALUE when driven to the L2 or memory. This field is only applicable for the cached pipe in dual-pipe CM configurations.	0

7.2.4 CM Error Detection

If error detection is supported, the CM detects, reports, and handles several types of errors that may be caused by errant software or hardware soft or hard errors.

Table 60: CM Error Types

CM Interrupt # (Type)	Error Name	Description
1	GC_WR_ERR	GIC or GCR register write accessed through cache address instead of an uncached address.

CM Interrupt # (Type)	Error Name	Description
2	GC_RD_ERR	GIC or GCR register read accessed through cache address instead of an uncached address.
3	COH_WR_ERR	Coherent write error caused by GIC, GCR, or MMIO areas accessed with a coherent address (address with a CCA of 4 or 5) instead of an uncached accessed.
4	COH_RD_ERR	Coherent read error caused by GIC, GCR or MMIO areas accessed with a coherent address (address with a CCA of 4 or 5) instead of an uncached accessed.
8	MEM_WR_RESP_ERR	Write response to the memory port with slave or decode error.
9	MEM_UC_WR_RESP_ERR	Write response to the uncached memory port (if implemented) with slave or decode error.
10	MMIO0_WR_RESP_ERR	Write response to MMIO port 0 with slave or decode error.
11	MMIO1_WR_RESP_ERR	Write response to MMIO port 1 with slave or decode error.
17	INTVN_WR_ERR	Request does not require a response and: one core responded with M and one or more cores responded with E, or S or One core responded with E and one or more cores responded with S or Multiple cores responded with data.
18	INTVN_RD_ERR	Request requires a response and: one core responded with M and one or more cores responded with E, or S or one core responded with E and one or more cores responded with S or multiple cores responded with data.
24[18]	L2_RD_UNCORR	An uncorrectable parity/ECC error occurred during a read to an L2 RAM.
25[18]	L2_WR_UNCORR	An uncorrectable parity/ECC error occurred during a write to an L2 RAM.
26[18]	L2_CORR	A correctable parity/ECC error occurred during an access to an L2 RAM.

The first 7 errors are invalid requests to the GCR, GIC, or MMIO: two errors for invalid intervention responses due to inconsistent L1 cache states and 3 errors due to L2 RAM parity errors.

If these registers already have valid error information and a second error is detected, the second error type is captured in the CM Error Multiple Register.

In addition to reporting the error an interrupt or error response or a normal response can be generated.

[18] Numbers 24 – 26 are L2 cache errors. To accommodate L2 cache sizes greater than 1MB, when the index field is too small in the CP0 CacheErr register to hold all index tags, the information is captured in the CM2 Error GCRs. The previous CP0 CacheErr functionality is preserved for L2 cache sizes of 1MB and less.

Enabling CM Interrupts

The CM Error Mask Register can be programmed to enable a CM interrupt. Each bit in this register enables the corresponding CM interrupt.

Table 61: CM Error Mask Register

Register Fields		Global CM Error Mask Register (GCR_ERROR_MASK Offset 0x0040)	Reset State
Name	**Bits**		
CM_ERROR_MASK	31:0	Each bit in this field represents an error type. If the bit is set, an interrupt is generated if an error of that type is detected. If the bit is set, the transaction for read errors completes with OK response to avoid double reporting of the error.	0x000A_002A Enables error numbers 2, 4, 5, 17, 19 (write errors cause interrupts; read errors provide error response).

Determine the Cause of a CM Error

The CM Error Cause Register contains information on the CM error number, CM_ERROR_TYPE, and an information field (CM_ERROR_INFO) that gives more precise information on the cause.

Table 62: Error Cause Register

Register Fields		Global CM Error Cause Register (GCR_ERROR_CAUSE Offset 0x0048)	Reset State
Name	**Bits**		
CM_ ERROR_TYPE	31:27	Indicates type of error detected. When CM_ERROR_TYPE is zero, no errors have been detected. When CM_ERROR_TYPE is non-zero, another error will not be reloaded until a power-on reset or this field is written to 0.	0
CM_ERROR_INFO	26:0	Information about the error. Based on type.	Undefined

Error Codes 1 - 7

If the decimal value in the CM_ERROR_TYPE field is between 1 and 15, the ERROR_INFO field in the Global CM Error Cause register is organized as shown as follows.

Table 63: ERROR_INFO Field State for Error Types 1 - 7

Bits	Meaning
26:18	Reserved
17:15	CCA
14:12	Target Region (0: MEM, 1:GCR, 2: GIC, 3: MMIO, 5: CPC)
11:7	OCP MCmd (see following table)
6:3	Source TagID
2:0	Source Port

The OCP MCmd field is further encodes as shown in the following table.

Table 64: MCmd (Bits 11:7) Encoding for CM_ERROR_INFO

MCmd Encoding	Description
0x01	Legacy Write
0x02	Legacy Read
0x08	Coherent Read Own
0x09	Coherent Read Share
0x0A	Coherent Read Discard
0x0B	Coherent Ready Share Always
0x0C	Coherent Upgrade
0x0D	Coherent Writeback
0x10	Coherent Copyback
0x11	Coherent Copyback Invalidate
0x12	Coherent Invalidate
0x13	Coherent Write Invalidate
0x14	Coherent Completion Sync

Error Codes 8 - 15

If the decimal value in the CM_ERROR_TYPE field is between 8 and 15, the ERROR_INFO field in the *Global Config* register is organized as shown in the following table.

Table 65: ERROR_INFO Field State for Error Types 8 - 15

Bit	Meaning
26:10	Reserved
9:8	Error type: • 2'b01 Unexpected EXOK • 2'b10 SLVERR • 2'b11 DECERR
7	Address available
6:3	Source TagID
2:0	Source port

Error Codes 16 - 23

If the decimal value in the CM_ERROR_TYPE field is between 16 and 23, the ERROR_INFO field in the *Global Config* register is organized as shown in the following table.

Table 66: ERROR_INFO Field State for Error Types 16 - 23

Bit	Meaning
26:21	Reserved
20:19	Coherent state from core 3 (see *Table 67: Coherent State Values for Error Types 16 - 23* on page 109)
18	Intervention SResp from core 3 (see *Table 68: Intervention SResp Values for Error Types 16 - 23* on page 109)

Bit	Meaning
17:16	Coherent state from core 2 (see *Table 67: Coherent State Values for Error Types 16 - 23* on page 109)
15	Intervention SResp from core 2 (see *Table 68: Intervention SResp Values for Error Types 16 - 23* on page 109)
14:13	Coherent state from core 1 (see *Table 67: Coherent State Values for Error Types 16 - 23* on page 109)
12	Intervention SResp from core 1 (see *Table 68: Intervention SResp Values for Error Types 16 - 23* on page 109)
11:10	Coherent state from core 0 (see *Table 67: Coherent State Values for Error Types 16 - 23* on page 109)
9	Intervention SResp from core 0 (see *Table 68: Intervention SResp Values for Error Types 16 - 23* on page 109)
8	Request was from a Store Conditional
7:3	OCP MCmd (see *Table 64: MCmd (Bits 11:7) Encoding for CM_ERROR_INFO* on page 108)
2:0	Source port

The following table shows the encoding for the coherent state errors for bits 20:19, 17:16, 14:13, and 11:10.

Table 67: Coherent State Values for Error Types 16 - 23

Encoding	Meaning
0	Invalid
1	Shared
2	Modified
3	Exclusive

The following table shows the encoding for the Intervention Sresp errors for bits 18,15, 12, and 9.

Table 68: Intervention SResp Values for Error Types 16 - 23

Encoding	Meaning
0	OK
1	Data (DVA)

Error Codes 24 - 26

If the decimal value in the CM_ERROR_TYPE field is between 24 and 26, the ERROR_INFO field in the Global Config register is organized as shown in the following table.

Table 69: ERROR_INFO Field State for Error Types 24 - 26

Bit	Meaning
26:24	Reserved (zero)
23	Multiple Uncorrectable

Bit	Meaning
22:18	Instruction[4:0] associated with the error see *Table 70: Instructions for Error Types 24 - 26* on page 110
17:16	Array type[1:0]: 00 = None 01 = Tag RAM single/double ECC error 10 = Data RAM single/double ECC error 11 = WS RAM uncorrectable dirty parity
15:12	DWord[3:0] with error, Array type = 2 only
11:9	Way[2:0] associated with the error
8	Multi-way error for Tag or WS RAM
7:0	Syndrome associated with Tag or WS way, or Syndrome associated with Data DWord

The instruction associated with the error is encoded into bits 22:18 of the ERROR_INFO field. The encoding for these bits is shown in the following table.

Table 70: Instructions for Error Types 24 - 26

Bit	Meaning
0x00	L2_NOP
0x01	L2_ERR_CORR
0x02	L2_TAG_INV
0x03	L2_WS_CLEAN
0x04	L2_RD_MDYFY_WR
0x05	L2_WS_MRU
0x06	L2_EVICT_LN2
0x08	L2_EVICT
0x09	L2_REFL
0x0A	L2_RD
0x0B	L2_WR
0x0C	L2_EVICT_MRU
0x0D	L2_SYNC
0x0E	L2_REFL_ERR
0x10	L2_INDX_WB_INV
0x11	L2_INDX_LD_TAG
0x12	L2_INDX_ST_TAG
0x13	L2_INDX_ST_DATA
0x14	L2_INDX_ST_ECC
0x18	L2_FTCH_AND_LCK
0x19	L2_HIT_INV
0x1A	L2_HIT_WB_INV
0x1B	L2_HIT_WB

7.2.5 Programming Individual Core Coherency Configuration

This section describes how to program a core's local GCR configuration. Two blocks of memory mapped registers are used to configure a core's local GCR: the Core-Local block and the Core-Other block.

- The Core-Local block is used to change the *local* GCR configuration for the core that is currently executing.
- The Core-Other block is used to change the local GCR configuration for *other* cores in the system.

These two blocks are offset from the base address of the Global Configuration Registers.

Table 71: Core-Local And Core-Other Offset from GCR Base

GCR Base Offset	Description
0x2000 - 0x3FFF	Core-Local Control Block (aliased for each CPU core). Contains registers pertaining to the core issuing the request. Each CPU has its own copy of registers within this block.
0x4000 - 0x5FFF	Core-Other Control Block (aliased for each CPU core). This block of addresses gives each core a window into another CPU's Core-Local Control Block. Before accessing this space, the Core-Other_Addressing Register in the VPE Local Control Block must be set with the CORENum of the target core.

Table 72: Core-Local and Core-Other Register Offsets

Register Offset	Name	Description
0x0008	Core Local Coherence Control Register	Controls which coherent intervention transactions apply to the local core.
0x0010	Core Local Config Register	Indicates the number of VPEs in this core, etc.
0x0018	Core Other Addressing Register (only in Core-Local block)	Used to access the registers of another core.
0x0020	Core Local Reset Exception Base Register	Sets the Reset Exception Base for the local core.
0x0028	Core Local Identification Register	Indicates the I7200 number of the local core.

Programming Access to another Cores Local Registers

There is 1 register in the Core-Local block that is not present in the Core-Other block and that is the Core-Other Addressing register. This register is used to select the core number that will be the target for reads (loads) and writes (stores) access to the Core-Other Control block. To access another cores local registers, write that core's number to the Core-Other Addressing Register then use the Core-Other control block offset to access the registers.

Table 73: Core-Other Addressing Register

Register Fields		Core-Other Addressing Register (GCR_CL_OTHER Offset 0x0018)	Reset State
Name	Bits		
CoreNum	31:16	CoreNum of the register set to be accessed in the Core-Other address space.	0

The following example code selects core 1 as the target core for the Core-Other block accesses:

```
li      t0, GCR_CONFIG_ADDR       // load GCR Base address
li      t1, 1                     // load a 1 for Core #1
sll      t1, t1, 16               // Shift Core number CoreNum field
// Write the Core Other Addressing Register
```

```
sw     t1, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_OTHER)(t0)
```

Programming a Coherency Domain

The Coherence Control Register puts the local core into coherent mode and enables the local core to be coherent with other requestors (core or IOCU) in the system.

Table 74: Core-Local Coherence Control Register

Register Fields		Core Local Coherence Control Register (GCR_Cx_COHERENCE Offset 0x0008)		Reset State
Name	**Bits**			
COH_DOMAIN_EN	7:0	Each bit in this field represents a coherent requester within the CPS. Setting a bit within this field enables interventions to this core from that requester. The requestor bit that represents the local core is used to enable or disable coherence mode in the local core. Changing the coherence mode for a local core from 0x1 to 0x0 can only be done after flushing and invalidating all the cache lines in the core; otherwise, the system behavior is UNDEFINED.		0x0
		Encoding	**Meaning**	
		Bit 0	Core 0	
		Bit 1	Core 1	
		Bit 2	Core 2	
		Bit 3	Core 3	
		Bit 4	IOCU 0	
		Bit 5	IOCU 1	

The following example core enables Core 0 coherency by setting bit 0 and enables coherency with cores 1 – 3 (assuming that the code is executing from core 0).

```
li    t0, GCR_CONFIG_ADDR      // load GCR Base address
li    t1, 0x0f                 // load mask for Core 0 and Cores 1 - 3
// Write it to the Core-Local Coherence Register
sw    t1, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_COHERENCE) (t0)
// Reading the Core-Local Coherence Register and issuing a sync insures write has taken effect
lw    t1, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_COHERENCE) (t0)
sync
```

To enable coherency on another core, first write the other core's number to the Core-Other Addressing Register. Then use CORE_OTHER_CONTROL_BLOCK instead of CORE_LOCAL_CONTROL_BLOCK in the example code.

Finding the number of VPEs on a Core

Sometimes it is desirable to write portable code. Instead of using a static number for VPEs software can look it up in the Core Local Config Register. The number of VPEs for the core that is executing is in the PVPE field.

Table 75: Core Local Config Register

Register Fields		Core Local Config Register (GCR_Cx_CONFIG Offset 0x0010)		Reset State
Name	**Bits**			
PVPE	9:0	**Encoding**	**Meaning**	IP Configuration Value

Register Fields		Core Local Config Register		Reset State
Name	**Bits**	**(GCR_Cx_CONFIG Offset 0x0010)**		
		0x0	1 VPE	
		0x1	2 VPEs	
		0x2	3 VPEs	

The number of VPEs on a core for the I7200 does not vary, therefore, there is no need to read the value from other cores.

Finding the Local Core Number

You can find the number on which software is executing by reading the CoreNum field of the Core-Local Identification Register.

Table 76: Core Local Identification Register

Register Fields		Core-Local Identification Register	Reset State
Name	**Bits**	**(GCR_Cx_ID Offset 0x0028)**	
CoreNum	31:0	This number is used as an index to the registers within the GCR when accessing the Core-local control block for this core.	

Programming the Boot Exception Vector (BEV)

To facilitate the BEV overlay scheme, a number of pins were added to the I7200 core that allow the user to select the boot overlay parameters at build time. In addition, the CM provides two registers in the Core-Local address space that allow the boot exception vector for each core to be located anywhere in physical memory.

The initial state of the default values selected by the user at build time are registered inside the Coherence Manager (CM) block using two Core-Local Configuration Registers. There are two GCR registers used per core. Each core has its own pair of these GCR registers and its own set of BEV related pins. This allows each core to be programmed in a different manner and independently from one another.

7.3 CM Performance Counters

7.3.1 CM Performance Counter Functionality

Performance characteristics of the CM can be measured via the CM performance counters. Two sets of identical programmable 32-bit performance counters in addition to a 32-bit cycle counter are implemented. The counters are controlled and accessed via GCR registers located in the Global debug block at offset 0x6000.

The counters are started by writing a 1 to the P0_CountOn, P1_CountOn and Cycl_Cnt_CountOn bits in the CM Performance Counter Control Register. Each counter can be reset to 0, and the corresponding overflow bit (P0_Overflow, P1_Overflow, Cyc_Cnt_Overflow) is reset to 0 prior to the start of counting by writing a 1 to the P0_Reset, P1_Reset and Cycl_Cnt_Reset bits in the same access that sets the corresponding start bits. This functionality allows all three counters to be reset and started with a single GCR write. The CM Performance Counter Control Register also controls how a counter overflow is handled. If the Perf_Ovf_Stop bit is set to 1, then all CM Performance counters will stop when one of the counters (including the Cycle Counter) reaches its maximum value of 0xFFFFFFFF. If instead the Perf_Ovf_Stop bit is set to 0, when a counter overflows, it rolls over and continues counting from 0.

If the Perf_Int_En bit is set to 1, an interrupt is generated when one of the counters (including the cycle counter) reaches its maximum value of 0xFFFFFFFF. The CM asserts the CM_PCInt signal which generates an interrupt only if the System Integrator has connected CM_PCInt to one bit of SI_CMInt.

When a performance counter overflows, the corresponding bit is automatically set in the CM Performance Counter Overflow Status Register. A status bit is cleared by writing a 1 to it.

The event to be counted by each performance counter is designated by the event number set in the Event_Sel_0 and Event_Sel_1 fields of the CM Performance Counter Event Selection Register. The events corresponding to the event numbers are listed and described in *Table 77: CM Performance Counter Event Types* on page 116. Each event is further specified by the CM Performance Counter Qualifier Register. The meaning of the CM Performance Counter Qualifier Register is different for each event. The column labeled “Qualifier” in *Table 77: CM Performance Counter Event Types* on page 116 shows the qualifiers that can be specified for each event. For example, the qualifiers for the Request_Count event (Event 0) are the request port, CCA, Burst Length, Command, and Target. The details of the qualifiers for the Request_Count event are defined in *Table 78: CM Performance Counter Request Count Qualifier* on page 117.

The qualifiers for some events are composed of several groups. A performance counter will increment if the specified event occurs and the qualifier criteria is matched in all groups. For example, assume the Event_Sel_0 field in the CM Performance Counter Event Selection Register is set to 0 (Request_Count). This event occurs when the CM serializes a request. However, the performance counter for this event will only count if the request meets the criteria programmed in all 5 groups in the Request Qualifier (see *Table 78: CM Performance Counter Request Count Qualifier* on page 117):

```
   The port that issued the request has the corresponding Request Port qualifier bit
   set to 1
AND
   The Cacheability attribute (CCA) for the request has the corresponding CCA
   qualifier bit set to 1
AND
   The Burst Length of the request (in dwords) has the corresponding qualifier bit set
   to 1
AND
   The OCP MCmd Type for the request has the corresponding Request Command qualifier
   bit set to 1
AND
   The target of the request has the corresponding Target qualifier bit set to 1
```

Multiple bits within a qualification group may be set. In this case, the OR of all bits set within the group. For example, by setting the request port qualifier for Port 0 and Port 1, then a request will be counted if it originated from Port 0 or Port 1.

A qualifier group can be set to “don’t care” by setting all bits within the group to 1. For example, to have performance counter 0 count all requests from port 1, program the CM Performance Counter Event Selection Register and CM Performance Counter Qualifier 0 Register as follows:

```
Set Event_Sel_0 to 0 (Request_Count)
Set Request Port Qualifer bit to 1 for Port 1
Set Requeset Port Qualifier bits to 0 for all other Ports
Set all other qualifer bits to 1 (causing the CCA, Burst Length, Command and Target
   to be ignored)
```

The two counters can be programmed to count a different event or the same event with different qualifiers. For example, to measure the ratio of requests from Port 1 vs. all Ports, set program Counter 0 to count requests from Port 1 (see previous example) and program Counter 1 to count all request from all Ports by setting Event_Sel_1 to 0 (Request_Count) and set all bits in the CM Performance Counter Qualifier 1 Register to 1.

The cycle counter can be used to calculate the average rates of specified events. Continuing the above example, assuming the cycle counter is reset, started, and stopped simultaneously with the two performance counters, then the rate of requests from port 1 and all ports can be easily computed (value of each performance counter / value in cycle counter).

7.3.2 CM Performance Counter Usage Models

There are several model for using the CM performance counters. This section discusses 3 possible models:

- Periodic Sampling - take many measurement samples of specific duration
- Stop and Interrupt when counter overflows - counters run until one overflows, then interrupt CPU
- Large count capability - enables unrestricted sample periods

One model for making performance measurements is for the software to set up and gather samples for a set period of time. The code sequence could follow the following steps:

```
start:
   Write CM Event and Qualifier Registers for particular event of interest
   Write CM Performance Counter Control Register to reset and start counters
      Perf_Int_En = 0 (no interrupt on overflow)
      Perf_Ovf_Stop = 0(no stop on overflow).
      P1_Reset = 1, P1_CountOn = 1
      P0_Reset = 1, P0_CountOn = 1
      Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
   Wait for some relatively small period of time (i.e., 2 seconds)
   Write CM Performance Counter Control Register to stop counters
      P1_Counton = 0, P0_CountOn=0, Cycl_Cnt_CountOn = 0
   Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
   If more events, go to start (or if measuring same counter go to step 2 instead)
```

A second CM performance counter usage model involves setting up the counters to stop and interrupt on overflow. This runs the counters until one of the counters (usually the cycle counter) reaches the 32-bit limit. An example of such a code sequence is:

```
start:
   Write CM Event and Qualifier Registers for particular event of interest
   Write CM Performance Counter Control Register to reset and start counters
      Perf_Int_En = 1 (interrupt on overflow)
      Perf_Ovf_Stop = 1(stop on overflow).
      P1_Reset = 1, P1_CountOn = 1
      P0_Reset = 1, P0_CountOn = 1
      Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
   When interrupt occurs:
   Read CM Performance Counter Status Register
   Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
   Write CM Performance Counter Control Register to reset counters
      (clears status register and interrupt)
      P0_Reset = 1, P1_Reset = 1, Cycl_Cnt_Reset = 1
   If more events, go to start (or if measuring same counter go to step 2 instead)
```

If larger counts than can fit into the 32-bit counters are required, the counters can be set up to interrupt, but not stop, on overflow. Memory variables can then count the number of overflows, as shown below:

```
start:
Write CM Event and Qualifier Registers for particular event of interest
Write CM Performance Counter Control Register to reset and start counters
   Perf_Int_En = 1 (interrupt on overflow)
   Perf_Ovf_Stop = 0 (do not stop on overflow).
   P1_Reset = 1, P1_CountOn = 1
   P0_Reset = 1, P0_CountOn = 1
   Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
When interrupt occurs:
<status>=Read CM Performance Counter Status Register
Increment <overflow_count>[counter] for each counter with <status> = 1
Write <status> to CM Performance Counter Status Register to clear interrupt
When run limit is reached then :
Write CM Performance Counter Control Register to stop counters
   P1_Counton = 0, P0_CountOn=0, Cycl_Cnt_CountOn = 0
Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
Write CM Performance Counter Control Register to reset counters
   (clears status register and interrupt)
   P0_Reset = 1, P1_Reset = 1, Cycl_Cnt_Reset = 1
If more events, go to start (or if measuring same counter go to step 2 instead)
```

In the above model, the final counts are calculated for each counter by multiplying <overflow_count>[counter] by 4G and adding the final values in the performance counter register.

7.3.3 CM Performance Counter Event Types and Qualifiers

Table 77: CM Performance Counter Event Types

Event #	Related Events	Use	Qualifiers	Description/Comments
0	Request_Count	Measuring Load	Request Port Request CCA Request Cmd Request Length Request Target See *Table 78: CM Performance Counter Request Count Qualifier* on page 117.	Can be used in conjunction with a cycle count to determine number of requests received in a given period of time.
1	Coh_Req_Resp	Track coherent requests or responses, and measure sharing	Intervention State Speculation Intervention Cmd Store Conditional See *Table 79: CM Performance Counter Coherent Request/Response Qualifier* on page 118.	Gives a count of the specified coherent request and response types.
2	L2_WR_Data_Ut	L2 Write Data Bus Usage	Accept State See *Table 80: CM Performance Counter Accept State Qualifier* on page 120.	Counts number of cycles the L2/Memory write data bus is occupied. The qualifier determines if stall cycles are counted or not.
3	L2_Cmd_Util	L2 Command Bus Usage	Accept State See *Table 80: CM Performance Counter Accept State Qualifier* on page 120.	Counts number of cycles the L2/ Memory command data bus is occupied. The qualifier determines if stall cycles are counted or not.
4	L2_RD_Data_Uti	L2 Read Data Bus Usage	None	Counts number of cycles the L2/Memory read data bus is occupied.
5	Sharing_Miss	Sharing Frequency	Request Source Port Data Source Port See *Table 81: CM Performance Counter CM Data Source Qualifier* on page 120.	Counts source of data for coherent read requests only (i.e., CohReadShare, CohReadDiscard, CohReadOwn, and CohReadAlways). Useful to determine how many cache misses were satisfied by other processors.
6	RSU_Util	RSU Usage	Port to measure Response Type See *Table 82: CM Performance Counter CM Port Response Qualifier* on page 121.	Counts number of d-words on the processor/iocu read data bus. A counter can only measure one port at a time. The port number is specified as the qualifier.
8	L2_Util	L2 Pipeline Usage	L2 Pipeline starts See *Table 83: L2 Utilization Qualifier* on page 121.	Counts starts into the TA stage of the L2 pipeline.
9	L2_Hit	L2 Hit/Miss Usage	Hit/Miss Type Source Port See *Table 84: L2 Hit Qualifier* on page 121.	Counts different types of L2 Cache Hits and Misses, crossed with Source Port ID.
10	RD_req	Measuring read load	Request Port Cacheability Target See *Table 86: CM Performance Counter Read Request and Latency Qualifier* on page 123	Gives count of matching read requests made.

Event #	Related Events	Use	Qualifiers	Description/Comments
11	RD_req_latency	Measuring read latency	Request port Cacheability Target See *Table 86: CM Performance Counter Read Request and Latency Qualifier* on page 123	Gives a count of total cycles all the matching read requests had to wait before giving responses to CPUs or IOCUs, i.e., total read latency. For example if 4 matching reads had to wait for 10 cycles each to get a response, then this counter will contain the value 40 at the end. In conjunction with RD_req, the average read latency can be determined by RD_req_latency count / RD_req count.
16	IOCU_Request	IOCU Request	CM Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type See *Table 85: IOCU Performance Counter Request Count* on page 122.	Counts requests receive by the IOCU. The CM receives a sideband signal, SI_CMP_IOC_PerfInfo from the IOCU as described in *Table 85: IOCU Performance Counter Request Count* on page 122.
17	IOCU1_Request	2nd IOCU Request	Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type. See *Table 85: IOCU Performance Counter Request Count* on page 122.	Counts requests receive by the 2nd IOCU. The CM receives a sideband signal, SI_CMP_IOC1_PerfInfo from the 2nd IOCU as described in *Table 85: IOCU Performance Counter Request Count* on page 122.

Table 78: CM Performance Counter Request Count Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31	Request Port	Port 7	Request originated from port 7
30		Port 6	Request originated from port 6
29		Port 5	Request originated from port 5
28		Port 4	Request originated from port 4
27		Port 3	Request originated from port 3
26		Port 2	Request originated from port 2
25		Port 1	Request originated from port 1
24		Port 0	Request originated from port 0
23	Request CCA[19]	WT	Request had Write Through Cacheability Attribute
22		UC/UCA	Request had Uncached Cacheability Attribute
21		WB	Request had Cached (non-coherent) Attribute
20		CWBE	Request had Coherent (Exclusive) Attribute
19		CWB	Request had Coherent (Shared) Attribute

[19] CCA qualifier group is ignored on non-coherent cache-ops.

Bit	Qualifier Group	Qualifier Value	Description/Comments
18	Burst Length[20] (# of dwords)	1 dword	Request was for 1 dword of data Note: This counts the burst length as seen by the Coherent Manager. Requests from the I/O Subsystem may be longer, but the IOCU may break these into multiple smaller requests.
17		2 dwords	Request was for 2 dwords of data See Note for 1 dword.
16		4 dwords	Request was for 4 dwords of data See Note for 1 dword
15	Request Command	Legacy WR	Request is a legacy Write command. This is used for all noncoherent writes. Note: When a processor is in coherent mode, L1 cache writebacks are always considered coherent, so they result in a cohWriteBack command, not a WR command.
14		Legacy RD	Request is a legacy Read command. This is used for all non-coherent reads, including code fetches.
13		CohReadShare CohReadShareAlwa	Request is a coherent read share generated by the processor on a load that misses its L1 cache. Currently CohReadShareAlways is unused.
12		CohReadOwn	Request is a coherent read own generated by the processor on a store that misses its L1 cache.
11		CohReadDiscard	Request is a coherent read discard generated by the IOCU for coherent requests.
10		CohUpgrade	Request is a coherent upgrade request generated by the the processor on a store that hits a shared line in its L1 cache.
9		CohWriiteBack	Request is coherent writeback generated by the processor when evicting a line from the L1 cache. The line may have been installed in the cache from a coherent or non-coherent transaction.
8		CohWriteInval (Partial Line)	Request is a coherent write invalidate (not a full line of data) generated by the IOCU.
7		CohWriteInval (Full Line)	Request is a coherent write invalidate (full line of data) generated by the IOCU.
6		CohInvalidate	Request is an invalidate request from a processor executing a PREF Prepare for Store or a CACHE Hit Invalidate.
5		CohCopyBack	Request from a processor executing a CACHE hit writeback
4		CohCopyBackInv	Request from a processor executing a CACHE hit CACHE WriteBackInvalidate
3		CohCompletionSync	Request is from a processor executing a SYNC instruction
2	Target	Memory	Request targets memory (coherent or non-coherent)
1		GCR/GIC/CPC	Request targets the Interrupt controller or Global Control Registers
0		MMIO	Request targets Memory Mapped I/O space

Table 79: CM Performance Counter Coherent Request/Response Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:25	Reserved		

[20] Burst Length only used when Request Command is Legacy Read, Legacy Write, CohReadDiscard or CohWriteInval.

Bit	Qualifier Group	Qualifier Value	Description/Comments
24	Intervention State	Exclusive with data	A processor has an exclusive copy in its L1 cache and returned data (all commands except CohInvalidate)
23		Exclusive with no data	A processor has an exclusive copy in its L1 cache but no data was returned (occurs on a CohInvalidate)
22		Modified with data	A processor has a modified copy in its L1 cache and returned data (all commands except CohInvalidate)
21		Modified with no data	A processor has a modified copy in its L1 cache but no data was returned (occurs on a CohInvalidate)
20		Shared	One or more processors have a shared copy in its L1 cache
19		Invalid	No processor has a copy of the data in its L1 cache
18	Speculation	Speculate	Requestwas a CohReadShare, CohReadOwn, CohReadDiscard or CohReadAlways and the CM issued a speculative read request to L2/Memory. This qualifier group is ignored when the request is not one of the commands listed above.
17		No Speculate	Requestwas a CohReadShare, CohReadOwn, CohReadDiscard or CohReadAlways and the CM did not issue a speculative read request to L2/ Memory. This qualifier group is ignored when the request is not one of the commands listed above.
16	Intervention Cmd	Reserved	Currently a don't care.
15		Reserved	Currently a don't care.
14		CohReadShare	Request is a coherent read share generated by the processor on a load that misses its L1 cache.
13		CohReadShareAlways	Currently CohReadShareAlways is unused.
12		CohReadOwn	Request is a coherent read own generated by the processor on a store that misses its L1 cache.
11	Intervention Cmd (cont.)	CohReadDiscard	Request is a coherent read discard generated by the IOCU for coherent requests.
10		CohUpgrade (OK Response)	Request is a coherent upgrade request generated by the processor on a store that hits a shared line in its L1 cache. There is no intervening request to the same line so an OK response is given.
9		CohUpgrade (Data Response)	Request is a coherent upgrade request generated by the processor on a store that hits a shared line in its L1 cache. There is an interveningrequest to the same line so a data response is given.
8		CohWriteBack	Request is coherent writeback generated by the processor when evicting a line from the L1 cache. The line may have been installed in the cache from a coherent or non-coherent transaction.
7		CohWriteInval (Partial Line)	Request is a coherent write invalidate (not a full line of data) generated by the IOCU.
6		CohWriteInval (Full Line)	Request is a coherent write invalidate (full line of data) generated by the IOCU.

Bit	Qualifier Group	Qualifier Value	Description/Comments
5		CohInvalidate	Request is an invalidate request from a processor executing a PREF Prepare for Store or a CACHE Hit Invalidate.
4		CohCopyBack	Request from a processor executing a CACHE hit writeback
3		CohCopyBackInv	Request from a processor executing a CACHE hit CACHE WriteBackInvalidate
2	Store Conditional (only used when cmd is CohUpgrade or CohReadOwn)	Not due to a Store Conditional	CohUpgrade or CohReadOwn is not due to a store conditional instruction. This qualifier group is ignored when thecommand is not a CohUpgrade or CohReadOwn.
1		Store Conditional that was not Cancelled	CohUpgrade or CohReadOwn is due a store conditional instruction and the intervention was not cancelled. This qualifier group is ignored when the command is not a CohUpgrade or CohReadOwn.
0		Store Conditional that was Cancelled	CohUpgrade or CohReadOwn is due a store conditional instruction and the intervention was cancelled due to livelock avoidance scheme. This qualifier group is ignored when the command is not a CohUpgrade or CohReadOwn.

Table 80: CM Performance Counter Accept State Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:1	Reserved		
0	Accept State	Count_Stalls	Setting this value to 0 for the L2_WR_Data_Util or L2_Cmd_Util events cause a count of cycles when a data word or command is accepted by the L2/ Memory. Setting this value to 1 for L2_WR_Data_Util or L2_Cmd_Util cause a count of cycles when a data word or command is valid on the bus, i.e., the count includes cycles where the command or data bus is stalled.

Table 81: CM Performance Counter CM Data Source Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:15	Reserved		
14	Request Port	7	Request originated from port 7
13		6	Request originated from port 6
12		5	Request originated from port 5
11		4	Request originated from port 4
10		3	Request originated from port 3
9		2	Request originated from port 2
8		1	Request originated from port 1
7		0	Request originated from port 0

Bit	Qualifier Group	Qualifier Value	Description/Comments
6	Response Port	5	Data returned by processor connected to port 5
5		4	Data returned by processor connected to port 4
4		3	Data returned by processor connected to port 3
3		2	Data returned by processor connected to port 2
2		1	Data returned by processor connected to port 1
1		0	Data returned by processor connected to port 0
0		L2/Mem	Data returned by L2/Memory

Table 82: CM Performance Counter CM Port Response Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:6	Reserved		
5	Response Type	Read Data Response	Response was a dword of data.
4		Write Acknowledge Response	Response was a write acknowledge (DVA response for a write).
3		OK Response	Response was an OK response (due to a CohUpgrade).
2:0	Port Number	Port to measure	Encoded value of port number to measure. For example, a value of 2 will only count responses on response port 2.

Table 83: L2 Utilization Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:6	Reserved		
5	Pipeline Start Type	L2 Pipeline start was stalled	Any type of pipeline request start (new, replay,refill) was refused due to a stall (ram or global stall)
4		L2 Pipeline start is taken	Use to calculate L2 utilization Any type of pipeline request start (new, replay,refill)
3		New request waiting for Sync to clear	A new request is waiting to be dispatched to the L2 until a preceeding Sync has guaranteed ordering
2		New L2 request stalled	New request to the L2 was not accepted due to a stall (ram or global stall)
1		New L2 request denied	New request to the L2 was not accepted due to replay, refill, or a stall.
0		New L2 request started	Use to calculate L2 bandwidth

Table 84: L2 Hit Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:22	Reserved		

Bit	Qualifier Group	Qualifier Value	Description/Comments
21	Partial Write Merge (write misses only)	Partial write merged	Partial write was merged into the SRT.
20		Partial write not merged	Partial write was not merged into the SRT.
19	Allocation (for Write orRead misses only)	Line allocated	Amiss caused an allocation by the L2. This occurs either for a full line write miss or a read miss, depending on the L2 allocation policy.
18		Line not allocated	A miss did not cause an allocation by the L2.
17	Hit/Miss Type (these are mutially exclusive)	Other	Index L2 cacheop or Fetch & Lock.
16		Non-index cache-op hit	Non-index L2 cacheop hit the L2 cache.
15		Non-index cache-op miss	Non-index L2 cacheop missed the L2 cache.
14		Write hit without RMW	Write hit and update the L2 without a read modify write operation. When the data byte enables of all DWords are either all 0 or all 1, an L2 write operation is performed without doing a read modify write operation.
13		Write hit with RMW	Write hit that requires a read modify write operation. When the data byte enable of any DWord is not all 1, a read modify write operation is required.
12		Write miss, no memory read	Write miss that does not require a memory read request.
11		Write miss requiring memory read	Write miss that does require a memory read request.
10		Read into CRQ	Read matched a pending L2 miss. Data is returned when the pending line is refilled. It is not a Read hit or a Read miss.
9		Read hit	Read hit the L2 cache.
8		Read miss	Read missed the L2 cache. Either allocates or reads through to memory, depending on the L2 allocation policy.
7	Source Port	7	Request originated from port 7
6		6	Request originated from port 6
5		5	Request originated from port 5
4		4	Request originated from port 4
3		3	Request originated from port 3
2		2	Request originated from port 2
1		1	Request originated from port 1
0		0	Request originated from port 0

Table 85: IOCU Performance Counter Request Count

Bit	Qualifier Group	Qualifier Value	Description/Comments
31	Reserved		
30:27	Transaction ID	TID	Value of IC Tag ID to match when the All_TID qualifier bit is set to 0. This field is unused when All_TID is 1.

Bit	Qualifier Group	Qualifier Value	Description/Comments
26		All_TID	If 1 then the all values of IC Tag ID will match. If 0 then only transactions with IC Tag ID equal to the TID specified above will match.
25	CM Transaction Count	65-129	Request resulted in 65-129 CM transactions.
24		33-64	Request resulted in 33-64 CM transactions.
23		17-32	Request resulted in 17-32 CM transactions.
22		9-16	Request resulted in 9-16 CM transactions.
21		5-8	Request resulted in 5-8 CM transactions.
20		3-4	Request resulted in 3-4 CM transactions.
19		2	Request resulted in 2 CM transactions.
18		1	Request resulted in 1 CM transaction.
17	Burst Length	129-256	IC Burst Length is 129-256 qwords.
16		65-128	IC Burst Length is 65-128 qwords.
15		33-64	IC Burst Length is 33-64 qwords.
14		17-32	IC Burst Length is 17-32 qwords.
13		9-16	IC Burst Length is 9-16 qwords.
12		5-8	IC Burst Length is 5-8 qwords.
11		3-4	IC Burst Length is 3-4 qwords.
10		2	IC Burst Length is 2 qwords.
9		1	IC Burst Length is 1 qword.
8	L2 Allocation	L2 Allocation	Request will cause an L2 allocation.
7		No L2 Allocation	Request will not cause an L2 allocation.
6	Posted[21]	*Non-posted Write*	*Write is non-posted. Not used on reads.*
5		*Posted Write*	*Write is posted. Not used on reads.*
4	Cacheability	Uncached	Request is uncached.
3		Cached	Request is Cached, non-coherent.
2		Coherent	Request is Coherent.
1	Request Type	Read	Request is a read.
0		Write	Request is a write.

Table 86: CM Performance Counter Read Request and Latency Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:30	Reserved		
29	Request Port	Port 5	Request originated from IOCU 1
28		Port 4	Request originated from IOCU 0
27		Port 3	Request originated from CPU 3
26		Port 2	Request originated from CPU 2
25		Port 1	Request originated from CPU 1
24		Port 0	Request originated from CPU 0

[21] This qualifier field is redundant when configured with AXI bus for IC port as writes are always non-posted with AXI bus.

Bit	Qualifier Group	Qualifier Value	Description/Comments
23:5	Reserved		
4	Cacheability	Cached	Requests with Cached CCA values. Ie CCA = 3,4,5
3		Uncached	Requests with Uncached CCA values. Ie CCA = 2,7
2	Target	Memory	Cached (Coherent or non-coherent) and Uncached requests targeting memory.
1		GCR/GIC/CPC	Uncached requests targeting CM registers in GCR, GIC and CPC.
0		MMIO	Uncached requests targetting MMIO ports.

8 Power Mangement and the Cluster Power Controller

This chapter describes the Cluster Power Controller (CPC) included in the I7200 Multiprocessing System. The CPC organizes bootstrap, reset, tree root clock gating, and power gating of CPUs. The CPC also manages power cycling, reset, and clock gating of the Coherence Manager, dependent on the individual core status and shutdown policy.

8.1 About the Cluster Power Controller

The CPC works with the power management features of the individual I7200 cores to provide a comprehensive power management scheme. The CPC manages static leakage and dynamic power consumption based on system-level power states assigned to the individual components of the I7200 Multiprocessing System. As such, the CPC acts as a programmable platform peripheral, which is accessible through cluster CPU software and SOC-level hardware protocols.

The CPC is an integral part of the coherent cluster and is designed to bootstrap, reset, tree root clock-gate and power-gate cluster CPUs and the Coherence Manager. Implementors may or may not choose to support some or all of the physical features the CPC is architected to control. The following physical power-management features can be selected independently:

- **Power gating of selected CPUs and/or the CM:** Supported by industry-standard physical design flows, supply voltage of individual power domains can be switched on-chip. Currently, the Unified Power Format (UPF) is provided for a seamless front to back-end design flow. Besides UPF-compliant EDA tools, standard cell libraries are required to provide power-gating header or footer cells, as well as isolate-high and isolate-low cells to separate unpowered domains from their active surroundings. The CPC provides a front-end RTL simulation environment and diagnostics to verify power-gating behavior.
- **Tree root clock gating:** Independent of CPU internal power-management features such as register-bank level clock gating and the sleep and doze modes, the CPC provides controls to gate clocks directly at or after the PLL to quiesse the entire clock tree of a CPU. CPC clock-gating signals are designed to bridge large clock insertion delays and are controlled through system-level power states.

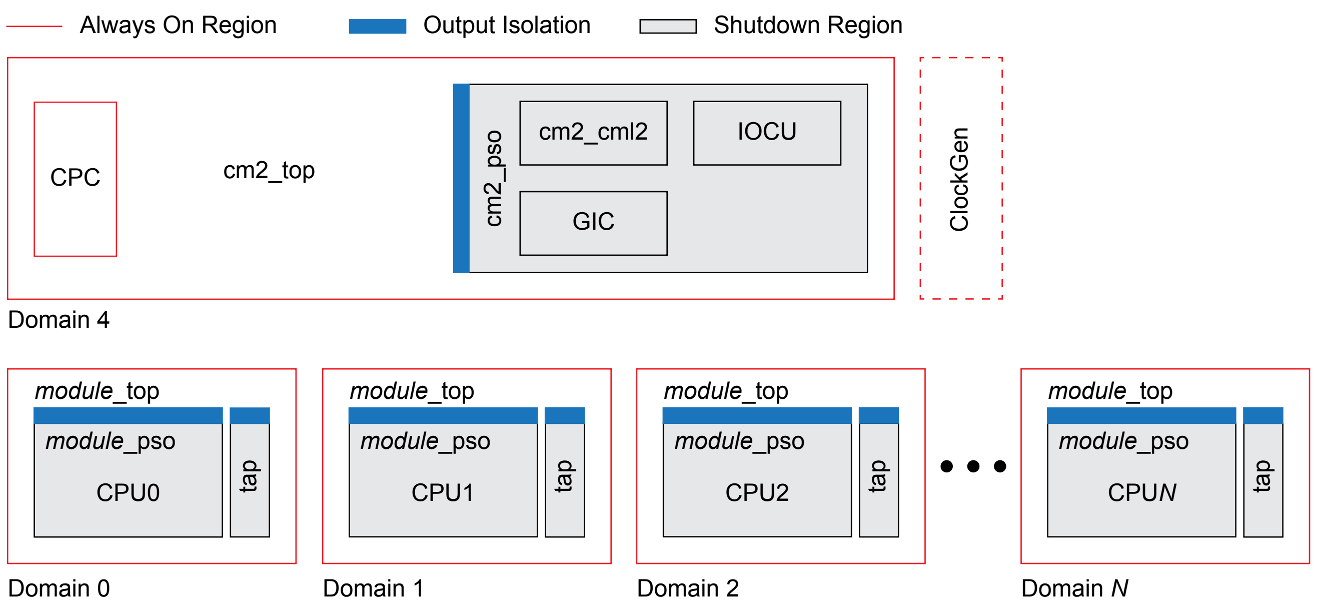
In addition to power-management functions, the CPC also acts as reset and bootstrap controller of the Multiprocessing System (MPS) to initialize cores as they become operational, or re-initialize them upon system-level requests. The CPC also facilitates debug probe access to cores by detecting the connection of a probe and enabling cores to respond to debug interrupt requests.

8.1.1 I7200 Power Domains

To power gate each core individually, the I7200 core has independently controlled power domains. RTL simulation (as well as physical implementation of the CPS) support five distinct domains, cpu0-N and the

Coherence Manager. These components are intended to be implemented with power rail switch cells to allow shutdown.

Figure 28: I7200 Power Domains



Each controllable domain also is required to drive isolation values towards the system. This ensures proper logic values from shutdown domain boundaries into powered surroundings.

The CPS top level can be implemented to belong to a voltage scaled supply domain. This enables dynamic voltage and frequency scaling over the full CPS with shutdown features for individual subdomains.

With shutdown of all cores, the Coherence Manager becomes inactive unless IOCU traffic is requested. The CPC provides programmable power down for these components.

8.1.2 Operating Level Transitions

To reach power-down and clock-off mode, software and hardware are required to go through a sequence of steps on each operating level to reach the next level.

Coherent to Non-Coherent Mode Transition

To leave the coherent domain and operate independently or prepare for shutdown, follow this sequence:

1. Switch to non-coherent CCA.
2. Flush dirty data from data cache using IndexWritebackInvalidate CACHE instruction on all lines in the cache.
3. If the instruction cache contains lines that are expected to be maintained by software as coherent (via globalized CACHE instructions), and the CPU is not going to go through a reset sequence, the instruction cache should be flushed using IndexInvalidate CACHE instructions.
4. Write GCR_CL_COHERENCE (Core Local GCR address 0x0008). Write 0 to all bits except bit for "self", which should stay set to 1. This step is required so that the core can issue a coherent SYNC (step 6) to make sure all previous interventions are complete.
5. Read GCR_CL_COHERENCE (ensures step 4 has completed).
6. Issue Coherent SYNC (intervention-only SYNC is fine).
7. Write 0 to GCR_CL_COHERENCE to completely remove core from coherence domain.

8. Read GCR_CL_COHERENCE to ensure step 7 is complete.

Non-Coherent to Coherent Mode Transition

An independently operating core becomes a member of a coherent cluster.

1. Caches must be initialized first (since last reset)
2. There should be no data in the caches that will later be accessed coherently. Non-coherent data is treated as exclusive/modified, which can lead to violations of the coherence protocol if other caches have copies of the data.
3. The GCR local coherence control register is programmed to add the core to the coherent domain.
4. Switch to coherent Cache Coherence Attribute (CCA).
5. Regular coherent programs can now start on this core.

Non-Coherent to Power Down Mode Transition

A core that is not a member of a coherent domain is powered down.

Note: When a probe is detected, the CPC will prevent power down to preserve the connectivity of the TAP scan chain. A power-down command instead causes the core to enter clock off mode.

1. The GIC might be programmed to re-route interrupts away from this core.
2. The CPC must be programmed to enter power-down mode.
3. Core outputs are held inactive towards the CM. Completion of pending bus traffic is awaited and start of new traffic prevented using the *SI_LPReq* protocol.
4. The CPC initiates the clock and power shutdown micro-sequence.

Non-Coherent to Clock Off Mode Transition

A core is disconnected from bus and stops operation. Dynamic power consumption is removed.

1. Programming a CPC ClkOff command disables the clock tree root for this core.
2. Core outputs are held inactive towards the CM. Completion of pending bus traffic is awaited and start of new traffic prevented using the *SI_LPReq* protocol.
3. The GIC might be programmed to re-route interrupts for this core to others.

Clock Off to Power Down Mode Transition

Power supply is removed from a disconnected core. Dynamic and leakage power is removed.

1. The CPC must be programmed to enter power-off mode.
2. The CPC initiates the clock and power shutdown micro-sequence.

Clock Off to Non-Coherent Mode Transition

A disconnected core is reconnected to the bus and starts operation.

1. The CPC command register is programmed to bring the core back on-line. A CPC_PwrUp command lets the core resume operation immediately, or, if a Reset command is given, go through a reset sequence before becoming operational.
2. If the core bus was isolated due to earlier power modes, this isolation is removed.
3. The clock is applied and the core starts executing instructions.

PowerDown to Non-Coherent Mode Transition

A core is powered up and becomes operational.

1. The GCR local coherence control register must be set inactive for this core. Powering up into a coherent state with uninitialized caches may corrupt coherent data.
2. Software on another core can send a PwrUp or Reset command for this core or an SOC hardware signal can request for the CPC to schedule a power-up sequence targeting non-coherent mode.
3. The CPC schedules a power-up sequence and the core becomes operational outside the coherent domain. After the core becomes operational, execution continues at the boot vector provided while power-up mode reset.

   Note: Reset is not automatically applied unless the core really was in the power-down state prior to a PwrUp command or hardware PwrUp signal.
4. The GIC might be reprogrammed to perform interrupt routing to this core.

8.2 CPC Register Programming

These sections describe some of the programming functions that can be performed via the CPC registers.

8.2.1 CPC 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 the following table, 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 CM.

Table 87: 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 is visible to all CPUs.
0x2000 - 0x3FFF	8 KB	**Core-Local Control Block**. Aliased for each I7200 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 I7200 core. This block of addresses gives each Core a window into another core’s Local Control Block. Before accessing this space, the Core-Other_Addressing Register in the Local Control Block must be set to the CORENum of the target core.

Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in *Table 87: CPC Address Map (Relative to GCR_CPC_BASE[31:15])* on page 128 are relative to a CPC base address and can be located anywhere in physical memory. The base address is a 17-bit value that is programmed into the GCR_BASE_ADDR field of the GCR_CPC_BASE register located at offset address 0x0088 in the Global Control Block of the CM registers.

To determine the physical address of each block listed in *Table 87: CPC Address Map (Relative to GCR_CPC_BASE[31:15])* on page 128, the base address written to the GCR_CPC_BASE register. This value would be added to the CPC block offset ranges to derive the absolute physical address as shown in *Table 88: Example Physical Address Calculation of the CPC Register Blocks* on page 129. (An example base address of 0x1BDE_0 is used for these calculations.)

Table 88: Example Physical Address Calculation of the CPC Register Blocks

Description	Example Base Address	GCR Block Offset	Absolute Physical Address	Size (bytes)
CPC Global Control Block.	0x1BDE_0	0x0000 - 0x1FFF	0x1BDE_ 0000 - 0x1BDE_1FFF	8 KB
CPC Core-Local Control Block.	0x1BDE_0	0x2000 - 0x3FFF	0x1BDE_ 2000 - 0x1BDE_3FFF	8 KB
CPC Core-Other Control Block.	0x1BDE_0	0x4000 - 0x5FFF	0x1BDE_ 4000 - 0x1BDE_5FFF	8 KB

Register Offsets Relative to the Block Offsets

In addition to the block offsets, the register offsets provided in each register description of this chapter are relative to the block offsets shown in *Table 88: Example Physical Address Calculation of the CPC Register Blocks* on page 129. To determine the physical address of each register, the base address programmed into the GCR_CPC_BASE register is added to the corresponding CPC block offset plus the actual register offset to derive the absolute physical address as shown in in the following table. This example uses a base address of 0x1BDE_0.

Table 89: Absolute Address of Individual CPC Global Control Block Registers

Global Control Register	MIPS Default Base	Global Register Block Offset	Global Register Offset	Absolute Physical Address
CPC Access Privilege.	0x1BDE_0	0x0000	0x0000	0x1BDE_0000
CPC Global Sequence Delay.	0x1BDE_0	0x0000	0x0008	0x1BDE_0008
CPC Rail Delay.	0x1BDE_0	0x0000	0x0010	0x1BDE_0010
CPC Reset Length.	0x1BDE_0	0x0000	0x0018	0x1BDE_0018
CPC Revision.	0x1BDE_0	0x0000	0x0020	0x1BDE_0020

The following table shows the absolute physical addresses for the CPC Core-Local block using an example base address of 0x1BDE_0.

Table 90: Absolute Address of Individual CPC Core-Local Block Registers

Global Control Register	MIPS Default Base	Global Register Block Offset	Global Register Offset	Absolute Physical Address
CPC Core-Local Command.	0x1BDE_0	0x2000	0x0000	0x1BDE_2000
CPC Core-Local Status and Configuration.	0x1BDE_0	0x2000	0x0008	0x1BDE_2008
CPC Core-Other Addressing.	0x1BDE_0	0x2000	0x0010	0x1BDE_2010

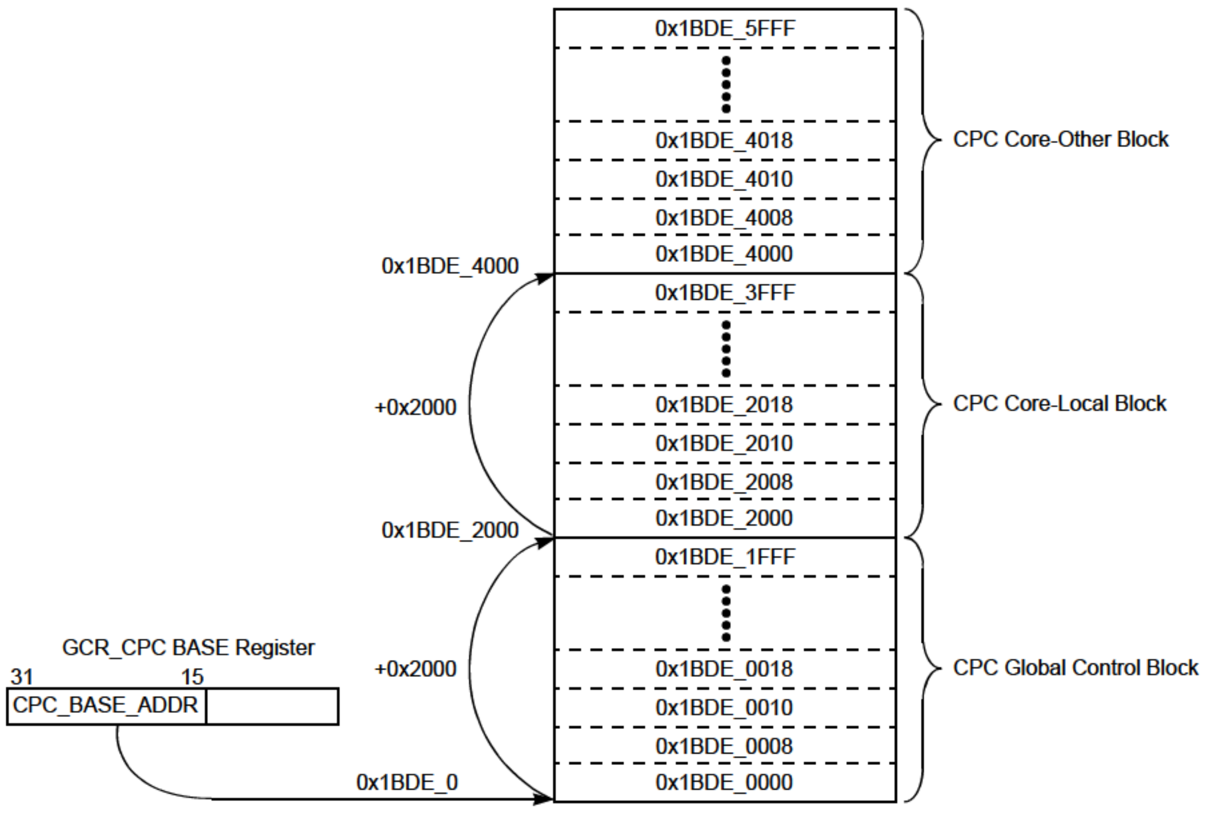
The following table shows the absolute physical addresses for the CPC Core-Other block using an example base address of 0x1BDE_0.

Table 91: Absolute Address of Individual CPC Core-Other Block Registers

Global Control Register	MIPS Default Base	Global Register Block Offset	Global Register Offset	Absolute Physical Address
CPC Core-Other Command.	0x1BDE_0	0x4000	0x0000	0x1BDE_4000
CPC Core-Other Status and Configuration.	0x1BDE_0	0x4000	0x0008	0x1BDE_4008

The following figure shows this concept (using a base address of 0x1BDE_0).

Figure 29: CPC Register Addressing Scheme Using an Example Base Address of 0x1BDE_0



Global Control Block Register Map

All registers in the Global Control Block are 32 bits wide and should only be accessed using aligned 32-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.

Table 92: Global Control Block Register Map (Relative to Global Control Block Offset)

Register Offset in Block	Name	Type	Description
0x000	CPC Global GCR Access Privilege Register (CPC_ACCESS_REG)	R/W	Controls which cores can modify the CPC registers.
0x008	CPC Global Sequence Delay Counter (CPC_SEQDEL_REG)	R/W	Time between microsteps of a CPC domain sequencer in CPC clock cycles.

Register Offset in Block	Name	Type	Description
0x010	CPC Global Rail Delay Counter Register (CPC_RAIL_REG)	R/W	Rail power-up timer to delay CPS sequencer progress until the gated rail has stabilized.
0x018	CPC Global Reset Width Counter Register (CPC_RESETLEN_REG)	R/W	Duration of any domain reset sequence.
0x020	CPC Global Revision Register (CPC_REVISION_REG)	R	RTL revision of CPC.
0x028 0x0F8	CPC Global RESERVED registers.	-	Reserved for future extensions.

Local and Core-Other Control Blocks

All registers in the CPC Local Control Block are 32 bits wide and should only be accessed using aligned 32-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 I7200 MPS. These registers can also be accessed from other cores by first writing the CPC Core Other Addressing Register (in the Core-Local Control Block) with the proper CoreNum and then accessing these registers using the Core Other address space.

The register offsets shown are relative to the offsets shown in the following table.

Table 93: Core-Local Block Register Map

Register Offset in Block	Name	Type	Description
0x000	CPC Local Command Register (CPC_CL_CMD_REG)	R/W	Places a new CPC domain state command into this individual domain sequencer. This register is not available within the CM sequencer. Writes to the CM CMD register are ignored while reads will return zero.
0x008	CPC Local Status and Configuration register (CPC_CL_STAT_CONF_REG)	R/W	Individual domain power status and domain configuration register. Reflects domain micro-sequencer execution. Initiates micro-sequencer after status register programming. Reflects command execution status.
0x010	CPC Core Other Addressing Register (CPC_CL_OTHER_REG)	R/W R/O for CM	Used to access local registers of another core.
0x018 0x0F8	CPC Local RESERVED registers	-	Reserved for future extensions.

The register offsets shown are relative to the offsets shown in the following table.

Table 94: Core-Other Block Register Map

Register Offset in Block	Name	Type	Description
0x000	CPC Local Command Register (*CPC_CO_CMD_REG*)	R/W	Places a new CPC domain state command into this individual domain sequencer. This register is not available within the CM sequencer. Writes to the CM CMD register are ignored while reads will return zero.
0x008	CPC Local Status and Configuration register (*CPC_CO_STAT_CONF_REG*)	R/W	Individual domain power status and domain configuration register. Reflects domain micro-sequencer execution. Initiates micro-sequencer after status register programming. Reflects command execution status.
0x010	CPC Core Other Addressing Register (*CPC_CO_OTHER_REG*)	R/W R/O for CM	Used to access local registers of another core.
0x018 0x0F8	CPC Local RESERVED registers	-	For Future Extensions

9 Global Interrupt Controller

This chapter describes the optional Global Interrupt Controller (GIC) included in the I7200 Multiprocessing System. The GIC can control up to 256 external interrupt sources in multiples of 8. This chapter describes how software controls the configuration and use of the GIC.

The optional GIC is selected at IP Configuration time. The GIC handles the distribution of interrupts between and among the CPUs in the cluster. On a multithreaded CPU with multiple Virtual Processing Elements (VPEs), each VPE has its own set of interrupt inputs. The GIC has the ability to route interrupts to each VPE independently.

9.1 GIC Terminology

In the context of the GIC, the term ‘Processor’ will be used to refer to a virtual processor in the I7200 core. The I7200 core can contain up to three VPEs per core.

The following table shows the processor numbering. If a core/VPE is not present in the system, that processor number is reserved.

Table 95: Processor Numbering

Processor Number	Core Number	VPE Number
0	0	0
1	0	1
2	0	2
3	Reserved	Reserved
4	1	0
5	1	1
6	1	2
7	Reserved	Reserved
8	2	0
9	2	1
10	2	2
11	Reserved	Reserved
12	3	0
13	3	1
14	3	2
15	Reserved	Reserved

9.2 GIC Features

To provide support for a multiprocessor environment, the GIC design includes the following features:

- Accepts interrupts from up to 256 external sources.
- Supports active-high, active-low, rising-edge triggered, falling-edge triggered, and dual-edge triggered interrupt signaling.
- Distributes/partitions the interrupt sources among the available cores and VPEs.

- Steers any interrupt source to any VPE interrupt input (Interrupt pin, NMI, yield qualifier).
- Allows any VPE to interrupt any other VPE.
- Backward compatible with pre-defined MIPS Technologies interrupt modes (legacy, vectored, and EIC).
- Scalable for both the number of interrupt sources as well as the number of VPE in the system.
- Able to integrate interrupt messages from peripherals such as PCI-Express.
- Supports multithreading, as defined in the MIPS MT-ASE and implemented by the I7200 cores, including routing interrupts to an individual Virtual Processing Element (VPE) within a CPU core and routing interrupts to the Yield Qualifier input pins of the CPU.
- Hardware assist features are software-configurable at run-time.
- Provides interval and watchdog timers.

9.3 GIC Address Map Overview

An I7200 Multiprocessing System can contain up to four cores and twelve VPEs. To avoid the large address space needed for VPE-specific register sets, an aliasing address scheme is used.

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

In the I7200 Multiprocessing System, any VPE can access the registers of any other VPE by using the VPE-Other address spaces. Software must write the VPE-Other Addressing Register before accessing these address spaces. The value of this register is used by hardware to index the appropriate subset of the control registers.

Two address windows are made available to the programmer:

- A window for the local VPE (as specified by the VPE number information).
- A second window for an other VPE that allows a VPE to access the register set belonging to another VPE. The other VPE is specified by first writing the VPE-Other Addressing Register in the local VPE address space.

The User-Mode Visible section 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.

The following table shows the GIC address map.

Table 96: GIC Address Space

Segment	Base Offset	Addressing Method	Address Space Size	Virtual Address Space Type
Shared Section Offset	0x00000	Offset relative to GCR_GIC_Base	32 KB	Kernel
VPE-Local Section Offset	0x08000	Offset relative to GCR_GIC_Base + using VPE number as Index	16 KB	Kernel
VPE-Other Section Offset	0x0C000	Offset relative to GCR_GIC_Base + using VPE-Other Addressing Register as Index	16 KB	Kernel

Segment	Base Offset	Addressing Method	Address Space Size	Virtual Address Space Type
User-Mode Visible Section Offset	0x10000	Offset relative to GCR_GIC_Base	64 KB	User

As shown in the previous table, the GIC address space is divided into four types:

- **Shared section:** The external interrupt sources are registered, masked, and assigned to a particular VPE and interrupt pin. This section is used by all VPEs and all cores in the system.
- **VPE-Local section:** Interrupts local to a VPE are registered, masked, and assigned to a particular interrupt pin. If External Interrupt Controller Mode (EIC) mode is used for a particular VPE, the EIC encoder is instantiated here.
- **VPE-Other section:** The local VPE can access the VPE-Local section of another VPE by which the interrupt can be registered, masked, and asigned to a particular interrupt pin of the other VPE. One VPE can setup the GIC for all VPEs in the system using this section.
- **User Mode Visible section:** Contains the GIC Hi/Lo counters accessible in user mode for quick user mode access. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers.

In the GIC, the Shared, VPE-Local, and VPE-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.

A separate 64 KB 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 registers that are aliased into this space are the shared GIC_SH_CounterLo and GIC_SH_CounterHi registers.

GIC Base Address

The GIC base address is a 15-bit value that is programmed into the GIC_BASE_ADDR field of the GCR GIC Base register located at offset address 0x0080 in the Global Control Block of the CM registers. Refer to the GCR_GIC_BASE Register in *Coherence Manager* on page 96 for more information on this register.

Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in *Table 96: GIC Address Space* on page 134 are relative to a GIC base address and can be located anywhere in physical memory. To determine the physical address of each block listed in *Table 97: Example Physical Address Calculation of the GIC Register Blocks* on page 135, the base address written to the GCR_GIC_BASE Register would be added to the GIC block offset ranges to derive the absolute physical address as shown in the following table. Note that an example base address of 0x1BDC_0 is used for these calculations.

Table 97: Example Physical Address Calculation of the GIC Register Blocks

Description	Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)
GIC Shared Control Block	0x1BDC_0	+	0x0000	=	0x1BDC_ 0000	32 KB
GIC Core-Local Control Block	0x1BDC_0	+	0x8000	=	0x1BDC_ 8000	16 KB
GIC Core-Other Control Block	0x1BDC_0	+	0xC000	=	0x1BDC_ C000	16 KB
User-Mode Visible Block	0x1BDC_0	+	0x10000	=	0x1BDD_ 0000	64 KB

Register Offsets Relative to the Block Offsets

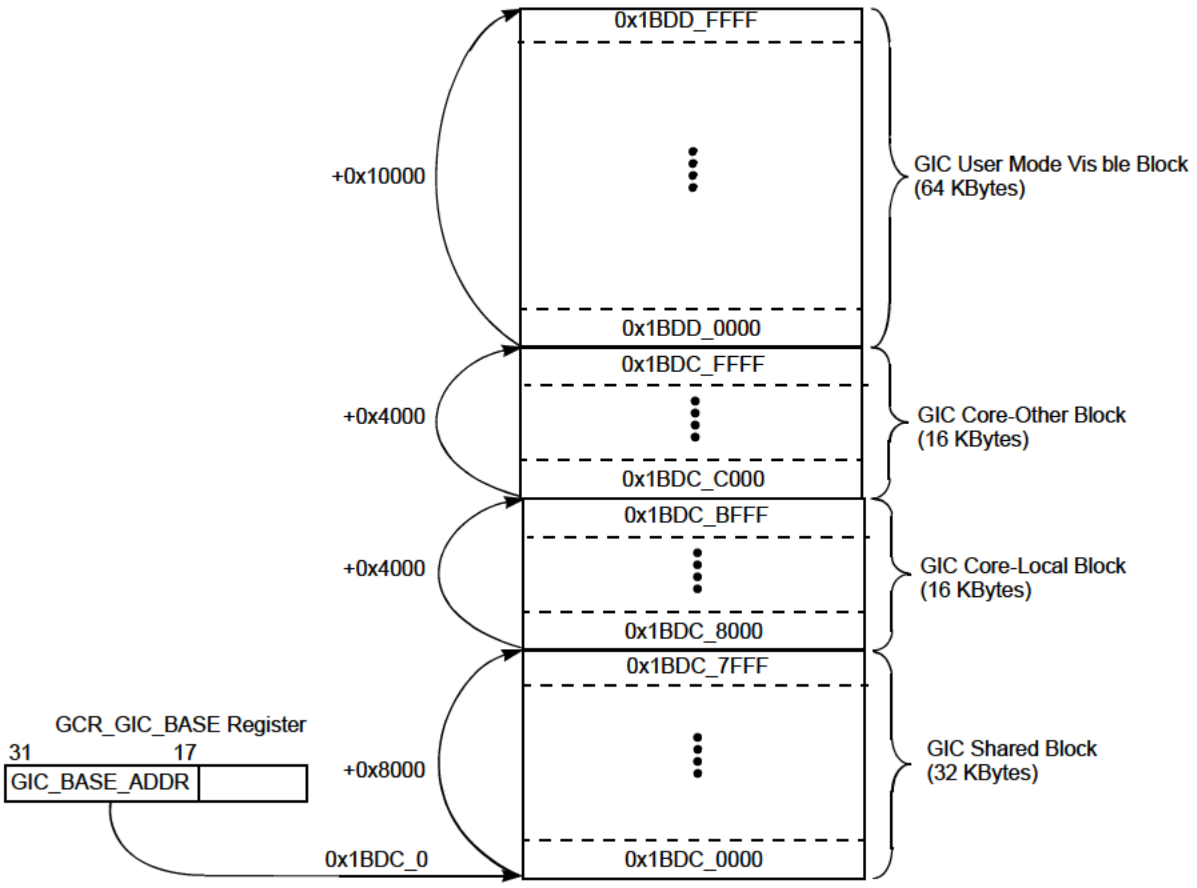
To determine the physical address of each register, the base address programmed into the GCR_GIC_BASE register is added to the corresponding GIC block offset described above, plus the actual register offset to derive the absolute physical address as shown in the following table. This table shows the physical address for the first few registers of the GIC Shared block and uses an example base address of 0x1BDC_0.

Table 98: Absolute Address of Individual GIC Shared Block Registers

Global Control Register	MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address
GIC Config	0x1BDC_0	+	0x000	+	0x000	=	0x1BDC_0000
GIC CounterLo	0x1BDC_0	+	0x000	+	0x010	=	0x1BDC_0010
GIC CounterHi	0x1BDC_0	+	0x000	+	0x014	=	0x1BDC_0014
GIC Revision	0x1BDC_0	+	0x000	+	0x020	=	0x1BDC_0020
CPC Interrupt Polarity 0	0x1BDC_0	+	0x000	+	0x100	=	0x1BDC_0100
...	...	+	...	+	...	=	...

This concept is described in the following figure using an example base address of 0x1BDC_0.

Figure 30: GIC Register Addressing Scheme Using an Example Base Address of 0x1BDC_0



9.4 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 interrupt external interrupts to specific processors
- Enabling or disabling interrupts
- Inter-processor interrupts
- Local device interrupt configuration

9.4.1 Setting the GIC Base Address and Enabling the GIC

As described in *GIC Base Address* on page 135, the base address for the memory mapped registers of the GIC is set using the GIC_BASE_ADDR field of the GCR_GIC_BASE Register. This field is normally programmed by the boot code. To enable the GIC the GIC_EN bit must be set in this same register.

The following example code shows how to set the GIC base address and enable the GIC at the same time.

```
#define GCR_CONFIG_ADDR 0xbfbf8000     // GCR registers base address (address should
                                       // be changed to match your I7200)
#define GCR_GIC_BASE 0x0080            // Offset of GIC Base address register from
                                       // GCR base address
#define GIC_P_BASE_ADDR 0x1bdc0000     // physical address of the GIC (address should
                                       // be changed to match your I7200)
li a1, GCR_CONFIG_ADDR + GCR_GIC_BASE  // Address of the GIC address register
li a0, GIC_P_BASE_ADDR | 1             // Physical address of the GIC + enable bit
sw a0, 0(a1)                           // Write to GCR[GIC base address register]
```

9.4.2 Configuring Interrupt Sources

The triggering of interrupts is configured through several registers in the GIC that are shared by all processors. All processors can access these registers but in practice these registers are usually programmed at boot time by processor 0. 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 32 bits so each register controls 32 interrupt sources. The first register in each group controls interrupts 0 - 31, the next 32 - 63 and so on. Since there can be 256 interrupt sources there could be 8 registers in each group. There are enough of these registers in each group to control the number of interrupt sources implemented. The number of interrupt sources is a fixed value configured at core build time. This number can be determined by reading the NUMINTERRUPTS field of the "GIC Configuration Register", GIC_SH_CONFIG.

The following example code determines the number of interrupt sources in your core:

```
// extracting number of interrupt slices:
#define GIC_SH_CONFIG 0x0000
#define NUMINTERRUPTS 16
#define NUMINTERRUPTS_S 8

li   a1, GIC_BASE_ADDR                       // load virtual base address of the GIC
```

```
                                                // registers NOTE: must be uncached address
lw   a0, GIC_SH_CONFIG(a1)                      // dereference GIC_SH_CONFIG
ext  a0, NUMINTERRUPTS, NUMINTERRUPTS_S         // NUMINTERRUPTS (actually slices - 1)
```

After the code executes, a0 contains how many groups of 8 plus 1 the core has. 0 indicates 1 group of 8, 1 indicates 2 groups of 8, and so on.

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 99: Selecting Interrupt Polarity, Edge Sensitivity, and Triggering* on page 139. 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.

Trigger Type Register Group

The trigger type register group is made up of shared "Global Interrupt Trigger Type Registers", GIC_SH_TRIG. The trigger type can be set to level or edge sensitive. Setting the source bit configures the source to be edge sensitive and clearing it configures it to be level sensitive. For example to set the interrupt source 32 to edge sensitive bit 0 of the second GIC_SH_TRIG Register should be set.

The following example code shows how to set interrupt source 31 to edge sensitive:

```
#define GIC_SH_TRIG31_0 0x0180 // offset from the GIC base address for trigger bits
                               // for interrupt sources 0 - 31
li a1, GIC_BASE_ADDR           // load virtual base address of the GIC registers
                               // NOTE: must be uncached address
li a0, 0x80000000              // interrupt source 31 (bit 31)
sw a0, GIC_SH_TRIG31_0(a1)     // (edge sensitive)
```

Edge Type Register Group

The edge type register group is made up of shared "Global Dual Edge Registers", GIC_SH_DUAL. This register group is used if the Trigger type described in the last section is set to edge sensitive and has no effect if the trigger type is level sensitive. The edge type can be either single or dual edge. Setting the source bit configures the source to be dual edge and clearing it configures it to be single edge. For example, to set interrupt source 32 to dual edge sensitive bit 0 of the second Global Dual Edge Registers should be set.

The following example code shows how to program interrupt source 31 to be dual edge sensitive:

```
#define GIC_SH_DUAL31_0 0x0200    // offset from the GIC base address for dual bits for
                                  // interrupt sources 0 - 31
li a1, GIC_BASE_ADDR              // load virtual base address of the GIC registers
                                  // NOTE: must be uncached address
li a0, 0x80000000                 // interrupt source 31 (bit 31)
sw a0, GIC_SH_DUAL31_0(a1)        // Dual
```

Polarity Type Register Group

The polarity register group is made up of shared "Global Interrupt Polarity Registers", GIC_SH_POL. This register group is used to determine the polarity sensitivity of the source.

- If the interrupt source type is level sensitive then setting the source bit configures the source 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.

The following example code shows how to program interrupt source 31 for high/raise polarity:

```
#define GIC_SH_POL31_0 0x0100    // offset from the GIC base address for polarity bits
                                 // for interrupt sources 0 - 31
li    a1, GIC_BASE_ADDR          // load virtual base address of the GIC registers
                                 // NOTE: must be uncached address
li    a0, 0x80000000             // interrupt source 31 (bit 31)
sw    a0, GIC_SH_POL31_0(a1)     // (high/rise for 31)
```

Table 99: 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 inter- rupts occur on both the rising and falling edges of the signal.

9.4.3 Interrupt Routing

The routing of interrupts to a specific input on a specific processor is controlled by the setting of 2 registers.

- Global Interrupt Map to Processor register, GIC_SH_MAP_VPE—Maps the interrupt to a VPE.
- Global Interrupt Map to Pin Register, GIC_SH_MAP_PIN—Maps interrupt to a specific signal on a VPE.

There is one of each of these 32 bit registers for each external interrupt source. The mapping of external interrupt pins and the registers that control them is listed in the following table.

Table 100: External Interrupt Mapping

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
0	0x2000	GIC_SH_MAP0_VPE	248	0x3F00	GIC_SH_MAP248_VPE
	0x0500	GIC_SH_MAP0_PIN		0x08E0	GIC_SH_MAP248_PIN
1	0x2020	GIC_SH_MAP1_VPE	249	0x3F20	GIC_SH_MAP249_VPE
	0x0504	GIC_SH_MAP1_PIN		0x08E4	GIC_SH_MAP249_PIN
2	0x2040	GIC_SH_MAP2_VPE	250	0x3F40	GIC_SH_MAP250_VPE
	0x0508	GIC_SH_MAP2_PIN		0x08E8	GIC_SH_MAP250_PIN
3	0x2060	GIC_SH_MAP3_VPE	251	0x3F60	GIC_SH_MAP251_VPE
	0x050C	GIC_SH_MAP3_PIN		0x08EC	GIC_SH_MAP251_PIN
4	0x2080	GIC_SH_MAP4_VPE	252	0x3F80	GIC_SH_MAP252_VPE
	0x0510	GIC_SH_MAP4_PIN		0x08F0	GIC_SH_MAP252_PIN
5	0x20A0	GIC_SH_MAP5_VPE	253	0x3FA0	GIC_SH_MAP253_VPE
	0x0514	GIC_SH_MAP5_PIN		0x08F4	GIC_SH_MAP253_PIN
6	0x20C0	GIC_SH_MAP6_VPE	254	0x3FC0	GIC_SH_MAP254_VPE
	0x0518	GIC_SH_MAP6_PIN		0x08F8	GIC_SH_MAP254_PIN
7	0x20E0	GIC_SH_MAP7_VPE	255	0x3FE0	GIC_SH_MAP255_VPE

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
	0x051C	GIC_SH_MAP7_PIN		0x08FC	GIC_SH_MAP255_PIN
8 - 247	0x2100 - 0x3EE0	GIC_SH_MAP8_VPE - GIC_SH_MAP247_VPE			
	0x0520 - 0x08DC	GIC_SH_MAP8_PIN - GIC_SH_MAP247_PIN			

Mapping an Interrupt Source to a VPE

There is one shared Global Interrupt Map to VPE Register, GIC_SH_MAP_VPE for each interrupt source that maps that source to a processor. Bit 0 would map the interrupt source to processor 0; bit 1 would map the interrupt to processor 1 and so on. Setting any bit in this register causes the interrupt source to be routed to the corresponding processor. For all GIC_SH_MAPi_VP registers, only one bit may be set at a time. That is, an interrupt source will be routed to one and only one VPE.

The following example code maps interrupt source 31 to processor 0, which is core 0 VPE 0 as shown in *Table 95: Processor Numbering* on page 133:

```
#define GIC_SH_MAP0_VPE31_0 0x2000    // register offset from GIC base
#define GIC_SH_MAP_SPACER 0x20        // space between registers
li a1, GIC_BASE_ADDR
a0, 1                                 // set bit 0 for CORE0 or for MT vpe0
sw a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 31) (a1)  // source 31 to VPE 0
```

Mapping and Interrupt Source to a Specific VPE Pin

There is one shared "Global Interrupt Map to Pin Register", GIC_SH_MAP_PIN for each external interrupt source that further maps that source to a specific signal on the VPE. There are two bits that control the type of signals that can be assigned to the interrupt source.

- If set, the MAP_TO_PIN bit will map the external interrupt source to Interrupt Pending bits in the CP0 Cause register of the local processor. The actual Interrupt Pending value is set in the MAP field of this register.
- Note that in EIC mode, the MAP Field of this register contains the encoded value of the number (0 -63). For example, a value of 0x20 asserts Interrupt 32 (decimal). For vectored interrupt mode, only values of 0x0 through 0x5 should be used.
- If set, the MAP_TO_NMI bit will map the external interrupt source to the NMI bit in the CP0 Status register. This in essence will cause the processor to soft boot using the boot exception vector as the start of the interrupt routine.
- MAP_TO_YQ bit determines that the source is a Yield Qualifier. The actual Yield Qualifier setting is set in the MAP field of the register.

Mapping an Interrupt Source to a Register Set

Each processor has one register per interrupt source used when the processor is in EIC mode to map the interrupt source to a register set. This is the EIC Shadow Set Register, GIC_VPEi_EICSS, located in the GIC local and other sections.

The first register corresponds to interrupt source 0; the second to interrupt source 1 and so on. The EIC_SS field is set to the register set number.

9.4.4 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 processors.

There are four 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 32 bits so each controls 32 interrupt sources. The first register in each group would control interrupts sources 0 - 31, the next 32 - 63 and so on. Since there can be 256 interrupt sources there could be 8 registers in each group. There are enough of these registers in each group to control the number of interrupt sources implemented. The number of interrupt sources is a fixed value configured at core build time. This number can be determined by reading the NUMINTERRUPTS field of the GIC Configuration Register, GIC_SH_CONFIG.

Refer to *Configuring Interrupt Sources* on page 137 for an example that shows how to read the NUMINTERRUPTS field.

Enabling External Interrupts

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.

The following example code shows the enabling of interrupts 24 through 31:

```
li a0, 0xff000000
sw a0, GIC_SH_SMASK31_00(a1) // (enable 24..31)
```

Disabling External Interrupts

The GIC Reset Mask register group is used to disable external interrupts. It is made up of GIC Reset Mask Registers, GIC_SH_RMASK. For synchronization purposes; this is a write only register. Setting the source bit disables the interrupt.

The following example code disables interrupts sources for interrupts 24 through 31:

```
li a0, 0xff000000
sw a0, GIC_SH_RMASK31_0(a1) // (disable 24..31)
```

Determining the Enabled or Disabled Interrupt State

The GIC Mask register group is used to determine if an external interrupt is enabled. It is made up of GIC Mask Registers, GIC_SH_MASK. For synchronization purposes; this is a read only register. If a bit is set the corresponding interrupt source is enable. If it is clear the corresponding interrupt is disabled.

Polling for an Active Interrupt

The GIC Pending register group is used to determine if a external interrupt is active. It is made up of GIC Pending Registers, GIC_PEND_MASK. This is a read only register. If a bit is set the corresponding interrupt source is active. If it is clear the corresponding interrupt is inactive.

9.4.5 Inter-processor Interrupts

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 source must be configured to be edge sensitive.

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.

Setting a bit in the Write Edge 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 VPE 0 wants to interrupt VPE 1, the number of the interrupt to be used is programmed into the GIC_SH_WEDGE31_0 register. The selected interrupt must be mapped to the target VPE (VPE1 in this example) using the GIC_SH_MAPi_VPE register).

For example, assume VPE 0 wants to toggle interrupt 40. In this case, software writes a value of 0x28 into the GIC_SH_WEDGE31_0 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 VPE 1, not VPE 0. The GIC routing logic then routes interrupt 40 onto the appropriate VPE 1 interrupt pins.

The following C code example that delivers a interprocessor interrupt to any given VPE:

```
#define GIC_SH_WEDGE_ADDR *((volatile unsigned int*) (0x1bdc0280))
// offset 0x280 of the GIC address
//
// set_ipi(): Send an inter-processor interrupt to the specified cpu.
//
void set_ipi(int VPE_num) {
  GIC_SH_WEDGE_ADDR = 0x80000000 + FIRST_IPI + VPE_num ; // Use external interrupts 32..39 for
 ipi
}
```

Inter-Processor Interrupt Code Example

The following example shows how to set up interrupt sources 32 through 39 for inter-processor interrupts. The following table shows the #define settings.

Setting Interrupt Sources 32 Through 39

#define	Value	Description
GIC_BASE_ADDR	0xbbdc0000	Virtual Base memory address of the GIC memory mapped registers
GIC_P_BASE_ADDR	0x1bdc0000	Physical Base address of the GIC memory mapped registers
GIC_SH_RMASK63_32	0x0304	Offset into the GIC registers for the GIC Reset Mask Register
GIC_SH_POL63_32	0x0104	Offset into the GIC registers for the GIC Reset Polarity Register
GIC_SH_TRIG63_32	0x0184	Offset into the GIC registers for the GIC Trigger Register
GIC_SH_SMASK63_32	0x0384	Offset into the GIC registers for the GIC Set Mask Register
GCR_CONFIG_ADDR	0xbfbf8000	Virtual base memory address of the Global Configuration Register
GCR_GIC_BASE	0x0080	Offset int the GCR of the GIC base Address

#define	Value	Description
GIC_SH_MAP0_VPE31_0	0x2000	Offset into the GIC for first map register
GIC_SH_MAP_SPACER	0x20	Spacing between map registers

```
// First load GIC base address into the GCR and enable the GIC
li   a1, GCR_CONFIG_ADDR + GCR_GIC_BASE // load the address of the GIC Base Address register
li   a0, (GIC_P_BASE_ADDR | 1) // Physical address + enable
sw   a0, 0(a1) // Store the Physical address of the GIC and the enable
               // bit to the GCR

// Configure the source pins for inter-processor interrupts
li   a1, GIC_BASE_ADDR // load GIC base address
li   a0, 0xff // load bits for interrupts 32..39 lower 8 bits of 2nd group)
sw   a0, GIC_SH_RMASK63_32(a1) // (disable interrupts 32..39)
sw   a0, GIC_SH_TRIG63_32(a1) // (set source to be edge sensitive for interrupts 32..39)
sw   a0, GIC_SH_POL63_32(a1) // (set Polarity to rising edge for interrupts32..39)
sw   a0, GIC_SH_SMASK63_32(a1)// (enable interrupts 32..39)

// Map interrupts to a processor

// The register offset into the GIC for the MAP TO VPE register is obtained by multiplying the
// interrupt number by the spacing size (GIC_SH_MAP_SPACER) and adding the offset for the Global
// Interrupt Map to VPE Registers (GIC_SH_MAP0_VPE31_0).

li   a0, 1 // set bit 0 processor 0

// Map Source 32 processor 0
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 32)(a1)
sll  a0, a0, 1 // set bit 1 for processor 1

// Source 33 to processor 1
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 33)(a1)
sll  a0, a0, 1 // set bit 2 for processor 2

// Source 34 to processor 2
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 34)(a1)
sll  a0, a0, 1 // set bit 3 for processor 3 or for MT vpe3

// Source 35 to processor 3
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 35)(a1)
sll  a0, a0, 1 // set bit 4 for processor 4

// Source 36 to processor 4
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 36)(a1)
sll  a0, a0, 1 // set bit 5 for processor 5

// Source 37 to processor 5
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 37)(a1)
sll  a0, a0, 1 // set bit 6 for processor 6

// Source 38 to processor 6
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 38)(a1)
sll a0, a0, 1 // set bit 7 for processor 7

// Source 39 to processor 7
sw   a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 39)(a1)
```

At this point the Map-to-Pin Registers could be used to map each interrupt source to Interrupt Pending bits in the CP0 Cause register of a processor. The default values for the Map to Pin registers are the MAP_TO_PIN bit is set and the MAP field is cleared. This example does not change the default values therefore the interrupts are mapped to IP2, Hardware Interrupt 0.

Example of Sending an Inter-Processor Interrupt

The following is a C coding example of sending an inter-processor interrupt.

#define	Value	Description
GIC_SH_WEDGE	*((volatile unsigned int*) (0x1bdc0280))	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 ;
```

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 will be 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 System VPE Number
addiu   k1, 0x20                // Offset to base of IPI interrupts
sw      k1, 0(k0)               // Clear this IPI.
```

9.4.6 Local Device Interrupt 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 registers, GIC_VPE_CompareLo/Hi of a processor with a global counter, GIC_SH_CounterLo/Hi in the GIC and activates an interrupt when they match.
- **GIC Watchdog Timer:** A 32 bit decrementing counter, GIC_VPE_WD_COUNT that can be used as liveliness signal for a processor.

GIC Interval Timer

The interval timer is similar to the CP0 Count/Compare timer within each processor. The difference is the GIC CounterLo/Hi register is global to the CPS so all processors will have the same time reference.

Both the interval count and interval compare values are 8 bytes wide and are made up of 2 (Lo/HI) registers. For each Lo register overflow the Hi register is incremented. If the Hi register overflows, both registers rollover to 0.

Counter Registers

The counter registers, GIC_SH_CounterLo/Hi are 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. These counter registers are 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_CounterHi register. In the GIC design, this field is fixed at a value of 0x8, indicating a total counter size of 64-bits.

The shared counter registers are defined as follows:

- GIC_SH_CounterLo register is used with the GIC_SH_CounterHi register. Sets the lower 32-bits of the starting count value.
- GIC_SH_CounterHi register is used with the GIC_SH_CounterLo register. Sets the upper 32-bits of the starting count value.

Compare Registers

The compare registers, GIC_VPE_CompareLo/Hi are 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 either GIC_VPE_CompareLo/Hi register. The compare registers are defined as follows:

- GIC_VPEi_CompareLo register is used with the GIC_VPEi_CompareHi register to set the count value at which an internal interrupt is generated.
- GIC_VPEi_CompareHi register is used with the GIC_VPi_CompareLo register to set the count value at which an internal interrupt is generated.

Determining the Counter Width

The counter used for GIC internal interrupt generation has a minimum width of 32 bits, meaning that all of the GIC_SH_CounterLo register is used. In the GIC design, the width of the GIC_SH_CounterHi register is also fixed at 32 bits as indicated by a value of 0x8 in the 4-bit COUNTBITS field in the GIC_SH_CONFIG register. To derive the total width of the counter, the following formula isused:

32 + COUNTBITS x 4

Where:

‘32’ is the width of the GIC_SH_CounterLo register and ‘COUNTBITS’ is the value in the COUNTBITS field of the GIC_SH_CONFIG register.

Since the COUNTBITS field contains a fixed value of 0x8, the overall width of the counter would be:

32 + 8 x 4 = 64 bits

In the GIC design, the COUNTBITS field is fixed at a value of 0x8, indicating a total counter size of 64-bits.

GIC Watchdog Timer

Each core supports a Watchdog timer that is controlled by the following three registers:

- The GIC Watchdog Timer Configuration Register, GIC_COREi_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_COREi_WD_INITIAL, is local to each processor and is used to set the timer interval.
- The Watchdog Timer Count Register, GIC_VPEi_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_COREi_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. Setting this bit to 1 will cause it to continue counting down in these states. Usually this bit is left unset.
- Clearing the Debug bit (default value) causes the counter to stop the count when the processor enters debug mode. When set the count continues counting down. Usually this bit is left unset.
- The TYPE field in bits 3:1 of this register determines what happens when the timer reaches 0.

GIC Watchdog Timer Modes

Encoding	Mode	Behavior
0x2	One Trip	An interrupt is asserted and the timer stops.

Encoding	Mode	Behavior
0x1	Second Countdown	An interrupt is asserted and the timer reloads. If the timer expires for the second time before being reloaded again all processors in the CPS will be reset. This mode provides a way to distinguish between a software hang and a hardware hang. Usually the Watchdog Timer Interrupt is routed to NMI. This will cause the processor to soft reboot. In this mode that is what happens when the timer expires the first time so if this was a software hang during the reboot the 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 so when the count expires the second time a reset signal will be sent to all processors in the system.
0x3	Programmable Interval Timer	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 WDEN bit disables the timer and when it is set it enables the timer. Writing WDEN with a 1 triggers a reloads the GIC_VPE_WD_COUNT register with the value in the GIC_COREi_WD_INITIAL register. Refer to "Watchdog Timer Config Register" for more information.

Watchdog Timer Initial Count Register

The Watchdog Timer Initial Count Register, GIC_COREi_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 WDEN bit in the GIC_COREi_WD_CONFIG register and the countdown value loaded into this register and then the counter enabled by setting the WDEN bit. Refer to "Watchdog Timer Initial Count Register" for more information.

Watchdog Timer Count Register

The Watchdog Timer Count Register, GIC_VPE_WD_COUNT is a read only register local to each processor that contains the current value of the countdown. This register is reloaded with the value in the GIC_COREi_WD_INITIAL register each time the WDEN bit in the GIC_COREi_WD_CONFIG register is set.

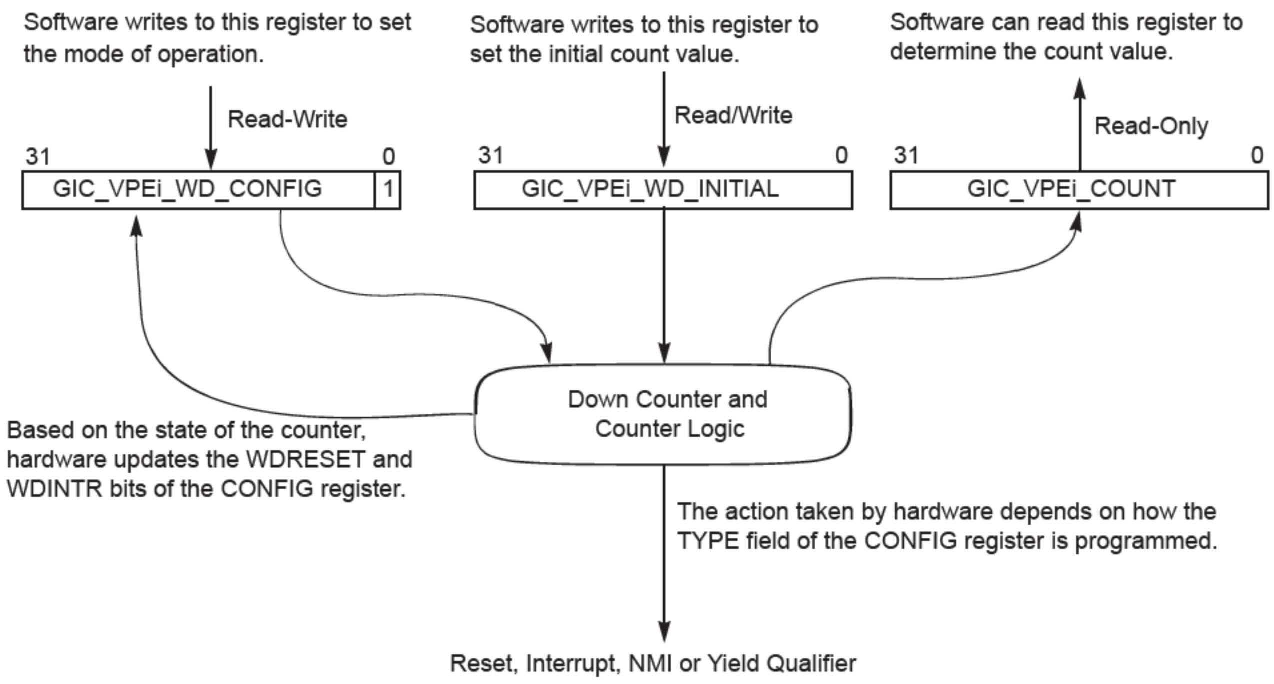
Refer to "Watchdog Timer Count Register" for more information.

Configuring the Watchdog Timer

Software can configure the WatchDog timer with a starting count value by programming the WatchDog Timer Initial Count register (GIC_VPEi_WD_INITIAL) located at offset address 0x0098. Refer to “Watchdog Timer Initial Count Register” for more information.

Software can read the state of the count at any time by reading the the WatchDog Timer Count register (GIC_VPEi_WD_COUNT) located at offset address 0x0094. Refer to “Watchdog Timer Count Register” for more information.

Figure 31: Local Watchdog Timer Interrupt Count Configuration


Software writes to this register to set the mode of operation.
Read-Write
31
0
GIC_VPEi_WD_CONFIG
1
Software writes to this register to set the initial count value.
Read/Write
31
0
GIC_VPEi_WD_INITIAL
Software can read this register to determine the count value.
Read-Only
31
0
GIC_VPEi_COUNT
Down Counter and Counter Logic
Based on the state of the counter, hardware updates the WDRESET and WDINTR bits of the CONFIG register.
The action taken by hardware depends on how the TYPE field of the CONFIG register is programmed.
Reset, Interrupt, NMI or Yield Qualifier

Watchdog Timer Masking and Mapping

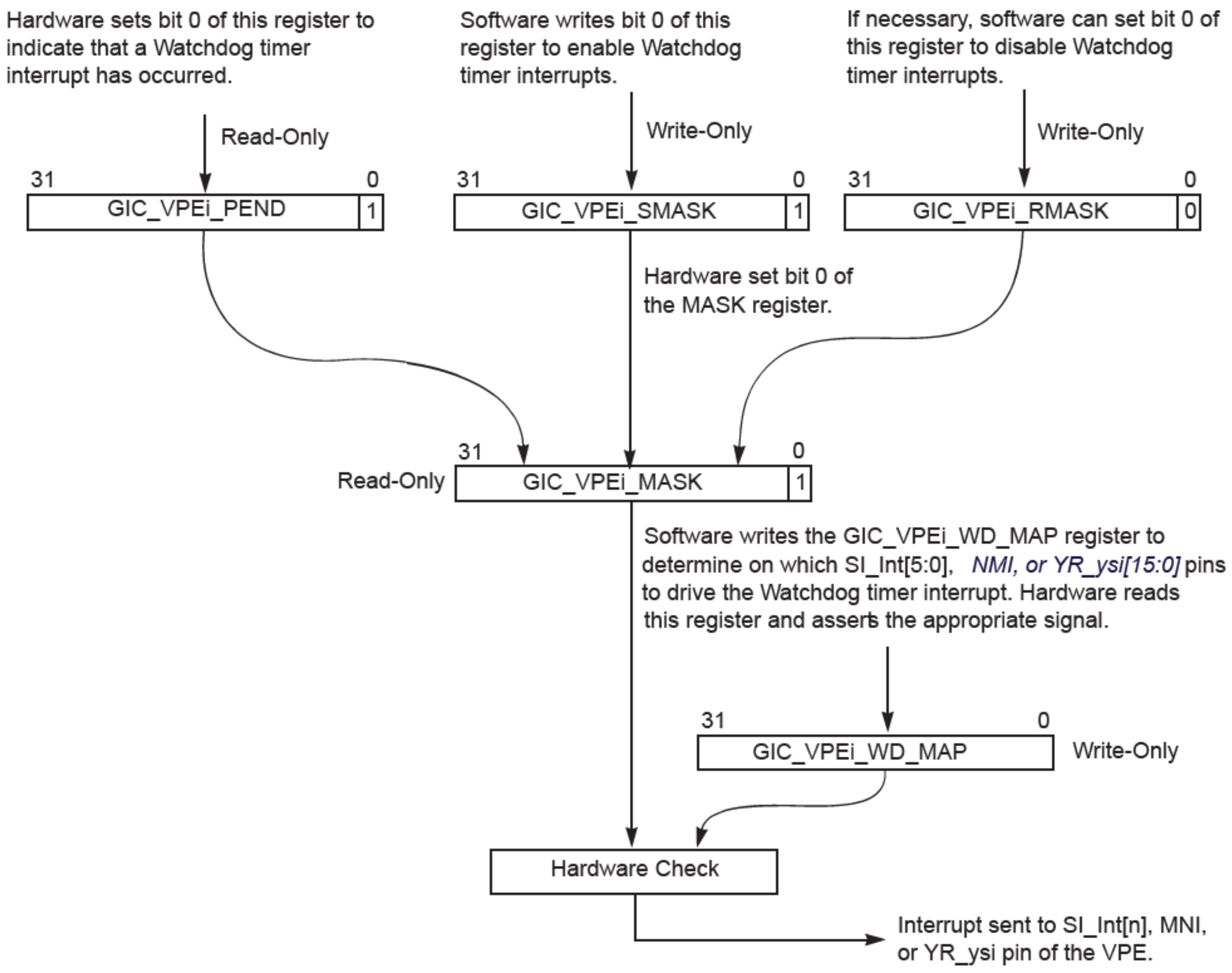
The previous figure shows the process used to configure the Watchdog timer. Once a Watchdog timer interrupt is generated (output of the figure), hardware sets bit 0 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 0 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the Watchdog timer interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_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 determine 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], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. For example, if software programs this field with a value of 0x2, then the Watchdog timer interrupt will be driven into SI_Int[2]. In non-EIC mode, only encodings 0 - 5 are valid.

In EIC mode, the VPE encodes this field to support up to 64 interrupts. For example, if 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 32: Watchdog Timer Interrupt Masking and Mapping in the GIC


Hardware sets bit 0 of this register to indicate that a Watchdog timer interrupt has occurred.
Software writes bit 0 of this register to enable Watchdog timer interrupts.
If necessary, software can set bit 0 of this register to disable Watchdog timer interrupts.
Read-Only
Write-Only
Write-Only
31
0
GIC_VPEi_PEND
1
31
0
GIC_VPEi_SMASK
1
31
0
GIC_VPEi_RMASK
0
Hardware set bit 0 of the MASK register.
31
0
Read-Only
GIC_VPEi_MASK
1
Software writes the GIC_VPEi_WD_MAP register to determine on which SI_Int[5:0], NMI, or YR_ysi[15:0] pins to drive the Watchdog timer interrupt. Hardware reads this register and asserts the appropriate signal.
31
0
GIC_VPEi_WD_MAP
Write-Only
Hardware Check
Interrupt sent to SI_Int[n], MNI, or YR_ysi pin of the VPE.

Watchdog Timer and Debug Mode

Under certain conditions, software may want to suspend Watchdog timer operation while the I7200 Multiprocessing System is in debug mode. This can be accomplished by clearing the DEBUGMODE_CTRL 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 this bit is set by software, entering debug mode has no effect on the Watchdog timer counting process.

Watchdog Timer and Low Power Mode

Under certain conditions, software may want to suspend Watchdog timer operation while the I7200 Multiprocessing System is in low power mode. This can be accomplished by clearing the WAITMODE_CTRL 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 software, entering low power mode has no effect on the Watchdog timer counting process.

9.4.7 Local Interrupt Routing

Routability of Local Interrupts

Local interrupts (except for the Watchdog timer, GIC Interval Timer and software interrupts) can be hardwired to local pins when the CPS is configured or can be more flexible and left to software to route the local interrupts to local pins on the processor. The Local Interrupt Control Register, GIC_VPEi_CTL, reports the routable state of the local interrupts. If the bit for the particular interrupt is set then the interrupt is routable within the GIC. The following table describes the behavior if not set.

Bits 4:1 of the GIC_VPEi_CTL register determines the routing of the following interrupts. In the I7200 GIC design, these bits are hard-wired to 1. Note that Software Interrupts from the VPE are routed internally by the CPU in vectored interrupt mode, and are only routed through the GIC when the GIC is in EIC mode, regardless of the GIC_VPEi_CTL register.

GIC_COREi_CTL Register Fields

Bit Field Name	Behavior if cleared
FDC_ROUTABLE	The CPU Fast Debug Channel Interrupt is hardwired to one of the SI_Int pins as described by the CPU's COP0 IntCtlIPFDCI register field.
SWINT_ROUTABLE	The CPU SW Interrupts are routed back to the CPU directly.
PERFCOUNT_ROUTABLE	The CPU Performance Counter Interrupt is hardwired to one of SI_Int pins as described by the CPU's COP0 IntCtlIPPCI register field.
TIMER_ROUTABLE	The CPU Timer Interrupt is hardwired to one of the SI_Int pins, as described by the CPU's COP0 IntCtlIPTI register field

Routing Local Interrupts

If a local interrupt is routable it can be routed to a local signal of the local processor, much the same as an external interrupt.

There is a Local Interrupt Map to Pin Register for each local interrupt source that further maps the local interrupt to a specific input on the processor. There are three bits, MAP_TO_PIN, MAP_TO_NMI, and MAP_TO_YQ 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.

- If set the MAP_TO_PIN bit will map the local interrupt source to Interrupt Pending bits in the CP0 Cause register of the processor. The actual Interrupt Pending bit is set in the MAP field of this register. The MAP Field of this register contains the encoded value of the number (0 -63). For example, a value of 0x20 asserts Interrupt 32 (decimal). For vectored interrupt mode, only use values of 0x0 to 0x5.
- If set the MAP_TO_NMI bit will map the local interrupt source to the NMI bit in the CP0 Status register. This in essence will cause the processor to soft boot using the boot exception vector as the start of the interrupt routine.
- If set the MAP_TO_YQ bit will map the local interrupt source to an MT Yield Qualifier pin. The actual Yield Qualifier setting is set in the MAP field of the register.

Each of these interrupt types is described in the following subsections. The following table lists the registers and associated bits that would be programmed to facilitate each type of interrupt described above.

Table 101: Local Interrupt Masking and Mapping Register Usage Per Interrupt Type

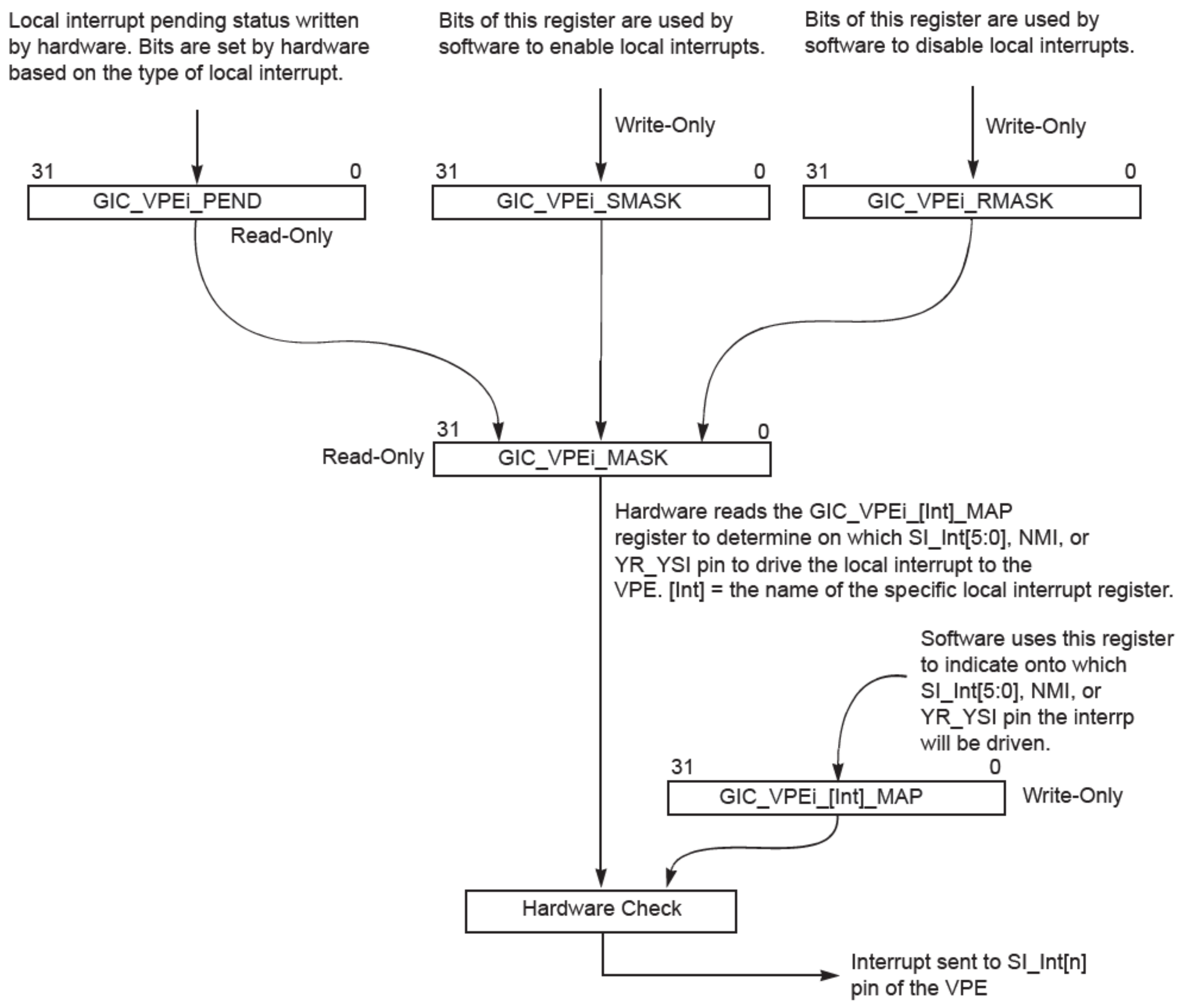
Interrupt	Register Name	Offset	Bits Used	Function
WatchDog	GIC_VPEi_PEND	0x0004	0	Set by hardware on a local WatchDog timer interrupt.

Interrupt	Register Name	Offset	Bits Used	Function
	GIC_VPEi_MASK	0x0008	0	Set by hardware based on the state of bit 0 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	0	Used by software to disable WatchDog timer interrupts.
	GIC_VPEi_SMASK	0x0010	0	Used by software to enable WatchDog timer interrupts.
	GIC_VPEi_WD_MAP	0x0040	31, 5:0	Used by software to map the WatchDog timer interrupt to one of the SI_Int[5:0] pins of the I7200 VPE.
Count and Compare	GIC_VPEi_PEND	0x0004	1	Set by hardware on a local Count/Compare interrupt.
	GIC_VPEi_MASK	0x0008	1	Set by hardware based on the state of bit 1 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	1	Used by software to disable Count/Compare interrupts.
	GIC_VPEi_SMASK	0x0010	1	Used by software to enable Count/Compare interrupts.
	GIC_VPEi_ COMPARE_MAP	0x044	31, 5:0	Used by software to map the Count/Compare interrupt to one of the SI_Int[5:0] pins of the I7200 VPE.
Timer	GIC_VPEi_PEND	0x0004	2	Set by hardware on a local timer interrupt.
	GIC_VPEi_MASK	0x0008	2	Set by hardware based on the state of bit 2 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	2	Used by software to disable timer interrupts.
	GIC_VPEi_SMASK	0x0010	2	Used by software to enable timer interrupts.
	GIC_VPEi_ TIMER_MAP	0x048	31, 5:0	Used by software to map the timer interrupt to one of the SI_Int[5:0] pins of the I7200 VPE.
Performance Counter	GIC_VPEi_PEND	0x0004	3	Set by hardware on a performance counter interrupt.
	GIC_VPEi_MASK	0x0008	3	Set by hardware based on the state of bit 3 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	3	Used by software to disable performance counter interrupts.
	GIC_VPEi_SMASK	0x0010	3	Used by software to enable performance counter interrupts.

Interrupt	Register Name	Offset	Bits Used	Function
	GIC_VPEi_ PERFCTR_MAP	0x0050	31, 5:0	Used by software to map the performance counter interrupt to one of the SI_Int[5:0] pins of the I7200 VPE.
Software Interrupt 0	GIC_VPEi_PEND	0x0004	4	Set by hardware on a software interrupt 0 occurrence.
	GIC_VPEi_MASK	0x0008	4	Set by hardware based on the state of bit 4 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	4	Used by software to disable software interrupt 0 interrupts.
	GIC_VPEi_SMASK	0x0010	4	Used by software to enable software interrupt 0 interrupts.
	GIC_VPEi_ SWInt0_MAP	0x0054	31, 5:0	Used by software to map software interrupt 0 to one of the SI_Int[5:0] pins of the I7200 VPE.
Software Interrupt 1	GIC_VPEi_PEND	0x0004	5	Set by hardware on a software interrupt 1 occurrence.
	GIC_VPEi_MASK	0x0008	5	Set by hardware based on the state of bit 5 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	5	Used by software to disable software interrupt 1 interrupts.
	GIC_VPEi_SMASK	0x0010	5	Used by software to enable software interrupt 1 interrupts.
	GIC_VPEi_ SWInt1_MAP	0x0058	31, 5:0	Used by software to map software interrupt 1 to one of the SI_Int[5:0] pins of the I7200 VPE.
Fast Debug Channel	GIC_VPEi_PEND	0x0004	6	Set by hardware on a Fast Debug Channel (FDC) interrupt.
	GIC_VPEi_MASK	0x0008	6	Set by hardware based on the state of bit 6 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_VPEi_RMASK	0x000C	6	Used by software to disable FDC interrupts.
	GIC_VPEi_SMASK	0x0010	6	Used by software to enable FDC interrupts.
	GIC_VPEi_FDC_MAP	0x004C	31, 5:0	Used by software to map the FDC interrupt to one of the SI_Int[5:0] pins of the I7200 VPE.

The general overview of the local interrupt pending, masking, and mapping process is shown in the following figure.

Figure 33: Local Interrupt Masking and Mapping in the GIC


Local interrupt pending status written by hardware. Bits are set by hardware based on the type of local interrupt.
Bits of this register are used by software to enable local interrupts.
Bits of this register are used by software to disable local interrupts.
Write-Only
Write-Only
31
0
GIC_VPEi_PEND
Read-Only
GIC_VPEi_SMASK
GIC_VPEi_RMASK
Read-Only
GIC_VPEi_MASK
Hardware reads the GIC_VPEi_[Int]_MAP register to determine on which SI_Int[5:0], NMI, or YR_YSI pin to drive the local interrupt to the VPE. [Int] = the name of the specific local interrupt register.
Software uses this register to indicate onto which SI_Int[5:0], NMI, or YR_YSI pin the interrp will be driven.
GIC_VPEi_[Int]_MAP
Write-Only
Hardware Check
Interrupt sent to SI_Int[n] pin of the VPE

Watchdog Timer Interrupts

For more information, refer to *GIC Watchdog Timer* on page 145.

Count and Compare Interrupts

A count and compare interrupt occurs when the contents of the of GIC_VPEi_CompareLo and GIC_VPEi_CompareHi registers match the contents of GIC_SH_CounterLo and GIC_SH_CounterHi, the Count/Compare interrupt is triggered. Refer to the Counter Based Interrupt Example in *Local Device Interrupt Configuration* on page 144 for more information.

When a count and compare interrupt is generated, hardware sets bit 1 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 1 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the count and compare interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_RMASK) at offset address 0x000C. Software sets bit 1 of the SMASK register to enable the count and compare interrupt, or it can set bit 1 of the RMASK register to disable count and compare interrupts.

Once hardware has determined the masking characteristics of the interrupt, it uses the Count/Compare Map-to-Pin register at offset address 0x0044 to determine which SI_Int[5:0], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In vectored interrupt mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. In this mode, only encodings 0 - 5 are valid. In EIC mode, the VPE encodes this field to support up to 63 interrupts. For example, if software programs this field with a value of 0x20, then the WatchDog timer interrupt corresponds to interrupt level 32. This encoded value is then driven onto SI_Int[5:0].

Timer Interrupts

When a timer interrupt is generated, hardware sets bit 2 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 2 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the timer interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_RMASK) at offset address 0x000C. Software sets bit 2 of the SMASK register to enable the timer interrupt, or it can set bit 2 of the RMASK register to disable timer interrupts.

Once hardware has determined the masking characteristics of the interrupt, it uses the Timer Map-to-Pin register at offset address 0x0048 to determine which SI_Int[5:0], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the VPE encodes this field to support up to 63 interrupts.

The following example code sets the local timer interrupt to interrupt vector 5

```
#define GIC_BASE_ADDR 0x1bdc0000 // MPU Direct address address of the GIC
                                 // (may be different for your configuration)
#define GIC_CORE_LOCAL_SECTION_OFFSET 0x8000
#define GIC_COREL_CTL 0x0000
#define TIMER_ROUTABLE_SHIFT 1
#define TIMER_ROUTABLE_BITS 1
#define GIC_COREL_TIMER_MAP 0x0048

// Initialize configuration of for vpe interrupts
li   a1, (GIC_BASE_ADDR | GIC_CORE_LOCAL_SECTION_OFFSET)
lw   a3, GIC_COREL_CTL(a1)

map_timer_int:
ext  a0, a3, TIMER_ROUTABLE_BITS, TIMER_ROUTABLE_SHIFT
beqz a0, map_perfcount_int       // not routable go to configuration of performance
                                 // counter interrupts
li   a0, 0x80000005                // Int5 is selected for timer routing
sw   a0, GIC_COREL_TIMER_MAP(a1)
```

Performance Counter Interrupts

When a timer interrupt is generated, hardware sets bit 3 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 3 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the performance counter interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_RMASK) at offset address 0x000C. Software sets bit 3 of the SMASK register to enable the performance counter interrupt, or it can set bit 3 of the RMASK register to disable timer interrupts.

Once hardware has determined the masking characteristics of the interrupt, it uses the Performance Counter Map-to-Pin register at offset address 0x0050 to determine which SI_Int[5:0], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the VPE encodes this field to support up to 63 interrupts.

The following example code sets the performance counter interrupts to local interrupt vector 4. Note this code follows the previous code example for the timer interrupt.

```
#define GIC_COREL_PERFCTR_MAP 0x0050
#define PERFCOUNT_ROUTABLE_SHIFT 2
#define PERFCOUNT_ROUTABLE_BITS 1

map_perfcount_int:
ext a0, a3, PERFCOUNT_ROUTABLE_BITS PERFCOUNT_ROUTABLE_SHIFT
beqz a0, done_gic       // not routable go to done
li a0, 0x80000004       // Int4 is selected for performance routing
sw a0, GIC_COREL_PERFCTR_MAP(a1)

done_gic:
```

Software Interrupts

Each VPE provides two software interrupts; 0 and 1. Software interrupts originate from the CPU and are only used in EIC mode. In non-EIC mode they are routed internally.

When software interrupt 0 is generated, hardware sets bit 4 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 4 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the performance counter interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_RMASK) at offset address 0x000C. Software sets bit 4 of the SMASK register to enable the software interrupt 0, or it can set bit 4 of the RMASK register to disable software interrupt 0.

Once hardware has determined the masking characteristics of the interrupt, it uses the Software Interrupt 0 Map-to-Pin register at offset address 0x0054 to determine which SI_Int[5:0], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the VPE encodes this field to support up to 63 interrupts.

The sequence is the same for software interrupt 1, except that bit 5 of each register noted above is set instead of bit 4. In addition, software uses the Software Interrupt 1 Map-to-Pin register at offset address 0x0058 to determine which SI_Int[5:0] pin the interrupt will be driven onto.

Fast Debug Channel Interrupts

When a Fast Debug Channel (FDC) interrupt is generated, hardware sets bit 6 of the Local Interrupt Pending register (GIC_VPEi_PEND) at offset address 0x0004. Hardware then reads the state of bit 6 in the Local Interrupt Mask register (GIC_VPEi_MASK) at offset address 0x0008 to determine whether the fast debug channel interrupt has been masked. The GIC_VPEi_MASK register is a read-only register.

Software can affect the state of this register using the write-only Local Interrupt Set Mask register (GIC_VPEi_SMASK) at offset address 0x0010 and the Local Interrupt Reset Mask register (GIC_VPEi_RMASK) at offset address 0x000C. Software sets bit 6 of the SMASK register to enable the fast debug channel interrupt, or it can set bit 6 of the RMASK register to disable fast debug channel interrupts.

Once hardware has determined the masking characteristics of the interrupt, it uses the Fast Debug Channel Map-to-Pin register at offset address 0x004C to determine which SI_Int[5:0], NMI, or YR_ysi[15:0] pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 VPE interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the I7200 VPE encodes this field to support up to 63 interrupts.

9.4.8 EIC Mode Setting

EIC mode is controlled through software by setting the EIC_MODE bit in the Local interrupt Control Register, GIC_VPE_CTL. Setting this bit enables EIC mode. This bit defaults to 0, vectored interrupt mode.

9.4.9 Enabling, Disabling, and Polling Local Interrupts

The Enabling, Disabling and Polling of local interrupts is configured through several registers in the GIC that are local to each processor.

There are 4 registers for Enabling, Disabling and Polling of local interrupts:

- Enabling an interrupt using the GIC Local Set Mask Registers, GIC_VPE_SMASK
- Disabling an interrupt using the GIC Local Reset Mask Registers, GIC_VPE_RMASK
- Determining the Enable/Disable state of an interrupt state using GIC Local Interrupt Mask Register, GIC_VPE_MASK
- Polling the interrupt active state using the GIC Local Interrupt Pending Register, GIC_VPE_PEND

Enabling External Interrupts

The GIC Local Set Mask Register, GIC_VPE_SMASK, is used to enable individual local interrupts. For synchronization purposes this is a write only register. Setting the bit enables the interrupt. The following table shows which field to set for each local interrupt.

Enabling External Interrupts

Field Name	Interrupt Controlled
FDC_MASK_SET	Fast Debug Channel
SWINT1_MASK_SET	Software interrupt 1
SWINT2_MASK_SET	Software interrupt 2
PERFCOUNT_MASK_SET	Local Performance Counter
TIMER_MASK_SET	CP0 Local Count/Compare Timer
COMPARE_MASK_SET	GIC Local Count/Compare Timer
WD_MASK_SET	Watchdog

Disabling External Interrupts

The GIC Local Reset Mask Register, GIC_VPE_RMASK, is used to disable individual local interrupts. For CPS synchronization purposes this is a write only register. Setting the bit disables the interrupt. The following table shows which field to set for each local interrupt.

Disabling External Interrupts

Field Name	Interrupt Controlled
FDC_RESET_MASK	Fast Debug Channel
SWINT1_RESET_MASK	Software interrupt 1
SWINT2_RESET_MASK	Software interrupt 2
PERFCOUNT_RESET_MASK	Local Performance Counter
TIMER_RESET_MASK	CP0 Local Count/Compare Timer
COMPARE_RESET_MASK	GIC Local Count/Compare Timer
WD_RESET_MASK	Watchdog

Determining the Enabled or Disabled Interrupt state

The GIC Local Mask Register, GIC_VPE_MASK, is used to determine if a local interrupt is enabled. For CPS synchronization purposes this is a read only register. If a bit is set the corresponding interrupt source is enabled. If it is clear the corresponding interrupt is disabled. The following table shows which field corresponds to each local interrupt.

Determining the Enabled of Disabled Interrupt State

Field Name	Interrupt Controlled
FDC_MASK	Fast Debug Channel
SWINT1_MASK	Software interrupt 1
SWINT2_MASK	Software interrupt 2
PERFCOUNT_MASK	Local Performance Counter
TIMER_MASK	CP0 Local Count/Compare Timer
COMPARE_MASK	GIC Local Count/Compare Timer
WD_MASK	Watchdog

Polling for an Active Interrupt

The GIC Pending Register, GIC_VPE_PEND, is used to determine if a external interrupt is active. This is a read only register. If a bit is set the corresponding local interrupt is active. If it is clear the corresponding interrupt is inactive. The following table shows which field corresponds to each local interrupt.

Table 102: Polling for an Active Interrupt

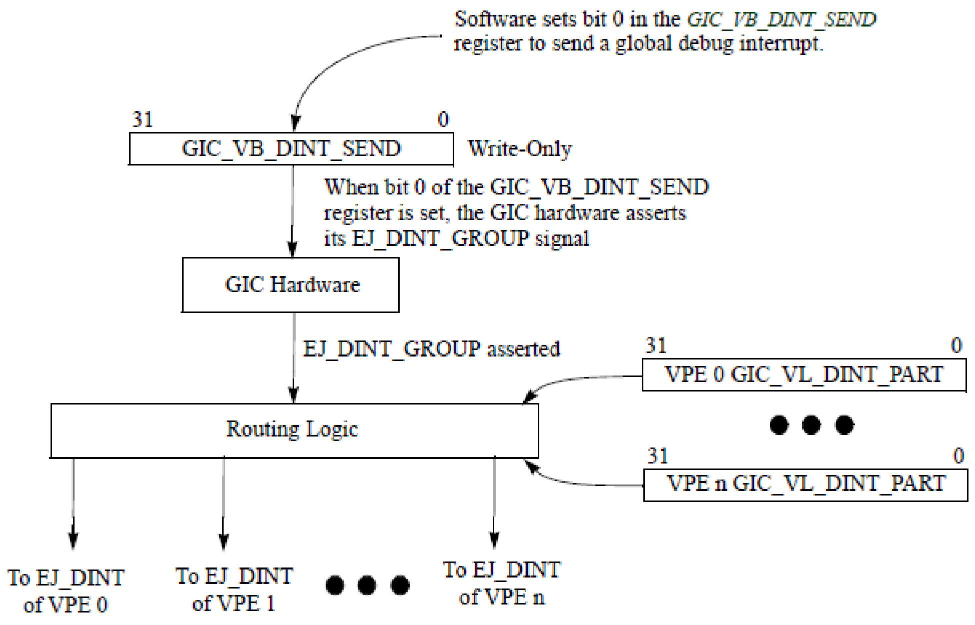
Field Name	Interrupt Controlled
FDC_PEND	Fast Debug Channel
SWINT1_PEND	Software interrupt 1
SWINT2_PEND	Software interrupt 2
PERFCOUNT_PEND	Local Performance Counter
TIMER_PEND	CP0 Local Count/Compare Timer
COMPARE_PEND	GIC Local Count/Compare Timer
WD_PEND	Watchdog

9.4.10 Debug Interrupt Generation

The GIC of the I7200 Multiprocessing System allows software to globally assert a debug interrupt to all VPEs in the system. When the Send_DINT bit of the DINT Send to Group register (GIC_VB_DINT_SEND) is set, the EJ_DINT_GROUP signal of the GIC is asserted. Based on the state of this signal and the VPE-

Local GIC_VL_DINT_PART registers, hardware asserts the EJ_DINT signal of each VPE in the system. This concept is shown in the following figure.

Figure 34: Global Debug Interrupt Generation in the GIC


Software sets bit 0 in the GIC_VB_DINT_SEND register to send a global debug interrupt.
31
0
GIC_VB_DINT_SEND
Write-Only
When bit 0 of the GIC_VB_DINT_SEND register is set, the GIC hardware asserts its EJ_DINT_GROUP signal
GIC Hardware
EJ_DINT_GROUP asserted
31
0
VPE 0 GIC_VL_DINT_PART
31
0
VPE n GIC_VL_DINT_PART
Routing Logic
To EJ_DINT of VPE 0
To EJ_DINT of VPE 1
To EJ_DINT of VPE n

9.5 Shared Register Set

GIC Register Field Types

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For single bit fields, the name is truncated to a single character, which is then shown outside brackets in the Fields|Name column. For the read/write properties of the field, the following notation is used:

Register Field Types

Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	

Notation	Hardware Interpretation	Software Interpretation
R	A field that is either static or is updated only by hardware. If the Reset State of this field is either "0" or "Preset", hardware initializes this field to zero or to the appropriate state, respectively, on power up. If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is "Undefined," software reads of this field result in an **UNPREDICTABLE** value except after a hardware update done under the conditions specified in the description of the field.
W	A field that can be written by software but which can not be read by software. Software reads of this field will return an **UNDEFINED** value.	
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is "Undefined," software must write this field with zero before it is guaranteed to read as zero.

Shared Section Register Map

The register map of the shared section is shown as follows. These registers are accessible by any VPE. For the base address of this block, see *Table 96: GIC Address Space* on page 134.

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCMP address space should return 0x0, and writes to those locations should be silently dropped without generating any exceptions.

The addresses for the registers within the Shared Section of the GIC are calculated as follows:

```
SharedSection_Register_Physical_Address=GIC_baseaddress+SharedSection_baseoffset+Re
 gister_Offset
```

Shared Section Register Map

Name	Description
GIC Config Register (GIC_SH_CONFIG)	Indicates the number of interrupts, number of VPEs, etc.
GIC CounterLo (GIC_SH_CounterLo)	Shared Global Counter.
GIC CounterHi (GIC_SH_CounterHi)	
GIC Revision Register (GIC_RevisionID)	RevisionID of the GIC hardware.

Name	Description
Global Interrupt Polarity Register0 (GIC_SH_POL31_0)	Polarity of the interrupt. For Level Type: 0x0 - Active Low 0x1 - Active High For Single Edge Type: 0x0 - Falling Edge used to set edge register 0x1 - Rising Edge used to set edge register At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Polarity Register1 (GIC_SH_POL63_32)	
Global Interrupt Polarity Register2 (GIC_SH_POL95_64)	
Global Interrupt Polarity Register3 (GIC_SH_POL127_96)	
Global Interrupt Polarity Register4 (GIC_SH_POL159_128)	
Global Interrupt Polarity Register5 (GIC_SH_POL191_160)	
Global Interrupt Polarity Register6 6(GIC_SH_POL223_192)	
Global Interrupt Polarity Register7 (GIC_SH_POL255_224)	
Global Interrupt Trigger Type Register0 (GIC_SH_TRIG31_0)	Edge or Level triggered 0x0 - Level 0x1 - Edge At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Trigger Type Register1 (GIC_SH_TRIG63_32)	
Global Interrupt Trigger Type Register2 (GIC_SH_TRIG95_64)	
Global Interrupt Trigger Type Register3 (GIC_SH_TRIG127_96)	
Global Interrupt Trigger Type Register4 (GIC_SH_TRIG159_128)	
Global Interrupt Trigger Type Register5 (GIC_SH_TRIG191_160)	
Global Interrupt Trigger Type Register6 (GIC_SH_TRIG223_192)	
Global Interrupt Trigger Type Register7 (GIC_SH_TRIG255_224)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL31_0)	Writing a 0x1 to any bit location sets the appropriate external interrupt source to be type dual-edged. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Dual Edge Register (GIC_SH_DUAL63_32)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL95_64)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL127_96)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL159_128)	

Name	Description
Global Interrupt Dual Edge Register (GIC_SH_DUAL191_160)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL223_192)	
Global Interrupt Dual Edge Register (GIC_SH_DUAL255_224)	
Global Interrupt Write Edge Register (GIC_SH_WEDGE)	Used for Interrupt Messages. Writes to this register atomically set or clear a specified bit in the Edge Detect Register.
Global Interrupt Reset Mask Register (GIC_SH_RMASK31_0)	Writing a 0x1 to any bit location masks off (disables) that interrupt. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Reset Mask Register (GIC_SH_RMASK63_32)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK95_64)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK127_96)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK159_128)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK191_160)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK223_192)	
Global Interrupt Reset Mask Register (GIC_SH_RMASK255_224)	
Global Interrupt Set Mask Register (GIC_SH_SMASK31_00)	Writing a 0x1 to any bit location sets the mask (enables) for that interrupt. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Set Mask Register (GIC_SH_SMASK63_32)	
Global Interrupt Set Mask Register (GIC_SH_SMASK95_64)	
Global Interrupt Set Mask Register (GIC_SH_SMASK127_96)	
Global Interrupt Set Mask Register (GIC_SH_SMASK159_128)	
Global Interrupt Set Mask Register (GIC_SH_SMASK191_160)	
Global Interrupt Set Mask Register (GIC_SH_SMASK223_192)	
Global Interrupt Set Mask Register (GIC_SH_SMASK255_224)	

Name	Description
Global Interrupt Mask Register (GIC_SH_MASK31_00)	Shows the enabled global interrupts. If bit N is set, global interrupt N is enabled. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Mask Register (GIC_SH_MASK63_32)	
Global Interrupt Mask Register (GIC_SH_MASK95_64)	
Global Interrupt Mask Register (GIC_SH_MASK127_96)	
Global Interrupt Mask Register (GIC_SH_MASK159_128)	
Global Interrupt Mask Register (GIC_SH_MASK191_160)	
Global Interrupt Mask Register (GIC_SH_MASK223_192)	
Global Interrupt Mask Register (GIC_SH_MASK255_224)	
Global Interrupt Pending Register (GIC_SH_PEND31_00)	Shows the pending global interrupts before masking. If bit N is set, the global interrupt N is pending. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Pending Register (GIC_SH_PEND63_32)	
Global Interrupt Pending Register (GIC_SH_PEND95_64)	
Global Interrupt Pending Register (GIC_SH_PEND127_96)	
Global Interrupt Pending Register (GIC_SH_PEND159_128)	
Global Interrupt Pending Register (GIC_SH_PEND191_160)	
Global Interrupt Pending Register (GIC_SH_PEND223_192)	
Global Interrupt Pending Register (GIC_SH_PEND255_224)	
Global Interrupt Map Src0 to Pin Register (GIC_SH_MAP0_PIN)	Maps this interrupt source to a particular pin - within Int[5:0] or NMI. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
Global Interrupt Map Src1 to Pin Register (GIC_SH_MAP1_PIN)	
Global Interrupt Map Src2 to Pin Register (GIC_SH_MAP2_PIN)	
...	
Global Interrupt Map Src255 to Pin Register (GIC_SH_MAP255_PIN)	

Name	Description
Global Interrupt Map Src0 to VPE Register (GIC_SH_MAP0_VPE31_0)	Assigns this interrupt source to a particular VPE. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources and the number of VPEs.
Global Interrupt Map Src1 to VPE Register (GIC_SH_MAP1_VPE31_0)	
Global Interrupt Map Src2 to VPE Register (GIC_SH_MAP2_VPE31_0)	
....	
Global Interrupt Map Src255 to VPE Register (GIC_SH_MAP255_VPE31_0)	
DINT Send to Group Register (GIC_VB_DINT_SEND)	Sends the DebugInterrupt to the specified VPE.
Reserved for future extensions	Reserved for future extensions.

9.6 GIC User-Mode Visible Section

The Shared, VPE-local, and VPE-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.

A separate 64 KB 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 registers that are aliased into this space are 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

User-Mode Visible Section Register Map

Register Offset	Name	Type	Description
0x0000	GIC CounterLo (GIC_SH_CounterLo)	R	Read-only alias for GIC Shared CounterLo.
0x0004	GIC CounterHi (GIC_SH_CounterHi)	R	Read-only alias for GIC Shared CounterHi.
Any Other Offsets	Reserved		Reserved for future extensions.

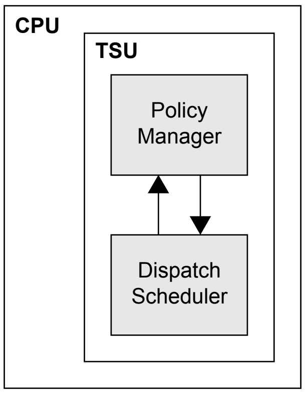
10 Policy Manager

The Policy Manager (PM) provides longer-term hints to the Dispatch Scheduler to achieve the desired system performance allocation. The I7200 core embeds the Policy Manager. The CPU's internal hardware manages the TC/VPE dispatch priorities based on two modes and four priority levels.

10.1 Thread Scheduling Unit

The I7200 core contains a unit called the Thread Scheduling Unit (TSU), which has two submodules: a Dispatch Scheduler and a Policy Manager. Both blocks are internal to the processor core, it cannot be modified by the customer.

Figure 35: TSU Block Diagram



The Dispatch Scheduler (DS) makes cycle-by-cycle choices on which instructions to issue/dispatch. The DS is designed to be as simple as possible.

10.2 Policy Manager Modes

The I7200 code provides the following Policy Manager (PM) modes:

- Quality of Service (QoS)
- Weighted Round-Robin

These modes support thread-scheduling capabilities that are common to many systems.

10.2.1 QoS Mode

In this mode, threads are grouped into high and low priority. High-priority threads have significantly greater resource allocation than low-priority threads. The thread priority level ranges from 3 to 0 (high to low). In QoS mode, TC priority 0 is considered low priority while everything else is considered high priority.

- High-priority threads can be allocated any number of hardware resources without restriction.
- To limit hardware resource consumption by low priority threads, if any low-priority thread occupies at least 1 WBB, FSB, or LDQ entry, subsequent instructions in the low priority threads will not be dispatched..

High-priority threads that are allocatable or activated (not halted, offline, stalled, or blocked) execute before low priority threads. Low-priority threads are only issued if high-priority threads cannot be issued and the low-priority thread is not limited by hardware resource constraints.

While the high-priority threads are runnable, the CPU dispatches and issues instructions from them. Threads are runnable if:

- Instructions are fetched and ready to be executed from the internal instruction buffer
- The thread is active (not halted or in an offline state)
- The thread is not suspended

The CPU reserves all but one of each of the available LDQ entries, FSB entries, and WBB entries for-high priority threads. Low-priority threads use one LDQ, one FSB, or one WBB entry. If the available entries are allocated to low-priority threads, the CPU stops issuing instructions from low-priority threads.

If the CPU has only high-priority threads or only low-priority threads, these threads are treated as equals and instructions are issued in a round-robin style.

In QoS mode, the priorities are arranged as:

- If a thread with priority 3 exists, all priority 3 threads are high priority. All other threads are low priority.
- If no priority 3 exists, and an active thread with priority 2 exists, all priority 2 threads are high priority. All other threads are low priority.
- If no priority 3 or 2 exist and an active thread with priority 1 exists, all priority 1 threads are high priority. All other threads are low priority.
- If multiple threads have the same priority level, the CPU does not differentiate between these them: resource allocation is round robin or first-come-first-serve.

The fetch order is:

1. Emergency mode: These threads have not fetched for a long time and risk starvation. The core gives these threads a one-cycle opportunity to fetch.
2. Round-robin priority 3
3. Round-robin priority 2
4. Round-robin priority 1
5. Empty IBF: Empty instruction buffer. Any thread whose instruction buffer is empty will fetch regardless of priority.
6. Last dispatched thread: If the previous criteria are not met, the next thread to fetch is the same as the last issued thread.
7. Round robin

Stall and block conditions include (but are not limited to):

- EHB hazard resolution
- Execution of ERET, DERET, WAIT, YIELD
- Pending Yield resume
- Pending ITC
- Pending on PAUSE
- Blocked by policy manager
- I-cache CACHE op is pending on another I-cache CACHE op
- COP1 / COP2 dependencies, if present
- MDU execution

Limitations

The QoS mode sacrifices fair-share and performance in low-priority threads to guarantee that high-priority threads are allocated with most of the CPU resources available. Software must be aware of this limitation.

Low-priority threads should not use performance enhancing capabilities such as the UCA cacheability attribute. The UCA CCA mode reserves a WBB entry and attempts to collect write-data before committing the write. Because the low-priority TC is prevented from running while a WBB entry is reserved and a high priority TC is active, the low-priority TC risks starvation until the high-priority TC is deprioritized.

10.2.2 Weighted Round-Robin Policy Mode

The weighted round robin mode attempts to execute high-priority thread instructions at a fixed rate compared to low-priority thread instructions. This rate is fixed in hardware to be 8, 4, 2, or 1, however, it may fluctuate due to pipeline behaviors or external events. CPU resources are allocated on a first-come, first-served basis. If there are only high-priority threads or only low-priority threads in the CPU, these threads are treated equally and instructions are issued in a round-robin style.

Each thread is assigned a priority level, and each priority level is assigned a weight. If threads have the same priority level, the CPU does not differentiate between them and the weight is the same.

- Priority 3 has weight 8
- Priority 2 has weight 4
- Priority 1 has weight 2
- Priority 0 has weight 1

During every 16 cycles, threads in a priority group have an opportunity to issue. Threads in a higher priority group or weight issue more often, but this behavior is dependent on the status of the CPU (such as stalls, synchronization, and exceptions). Threads participating in the dispatch algorithm must be allocatable or activated, not halted, offline, stalled, or blocked. Hardware resources (WBB, LDQ, and FSB) are allocated on a first-come, first-served basis. A thread with more dispatch opportunities may consume more hardware resources.

The fetch order is:

- Emergency mode
- Empty IBF: Empty instruction buffer. If the instruction buffer of a particular thread is empty, this thread fetches.
- Last dispatched thread: If the previous criteria are not met, the next thread to fetch is the same as the last issued thread.
- Round robin (similar to QoS mode)

Micro-architectural stall behaviors do not affect the fetch order.

10.2.3 CP0 Register Interface

The policies and thread priorities are always active. Software must write to MVPControl[16] (POLICY_MODE) to indicate which policy mode to use (POLICY_MODE = 1 for QoS mode and POLICY_MODE = 0 for weighted round robin mode).

A thread's priority is set in the CP0 TCSchedule PRIO bits. If the TCSchedule PRIO_EN bit is set, the PRIO bits can be overridden by the per-TC external pin input, PM_Prio, if the pin value is higher than the PRIO bits.

Instruction scheduling can be halted entirely if the TCSchedule TCHALT_EN bit is set and the external PM_TCHaltTC pin is raised.

The pins and the bits work together as follows:

- PM_TCHaltTC0: Halts TC0 if TC0.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC1: Halts TC1 if TC1.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC2: Halts TC2 if TC2.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC3: Halts TC3 if TC3.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC4: Halts TC4 if TC4.TCSchedule.TCHALT_EN=1

- PM_TCHaltTC5: Halts TC5 if TC5.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC6: Halts TC6 if TC6.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC7: Halts TC7 if TC7.TCSchedule.TCHALT_EN=1
- PM_TCHaltTC8: Halts TC8 if TC8.TCSchedule.TCHALT_EN=1
- PM_Prio0[1:0]: Sets the priority of TC0 if TC0.TCSchedule.PRIO_EN=1, and PM_Prio0[1:0] is higher than TC0.TCSchedule.PRIO[3:2]
- PM_Prio1[1:0]: Sets the priority of TC1 if TC1.TCSchedule.PRIO_EN=1, and PM_Prio1[1:0] is higher than TC1.TCSchedule.PRIO[3:2]
- PM_Prio2[1:0]: Sets the priority of TC2 if TC2.TCSchedule.PRIO_EN=1, and PM_Prio2[1:0] is higher than TC2.TCSchedule.PRIO[3:2]
- PM_Prio3[1:0]: Sets the priority of TC3 if TC3.TCSchedule.PRIO_EN=1, and PM_Prio3[1:0] is higher than TC3.TCSchedule.PRIO[3:2]
- PM_Prio4[1:0]: Sets the priority of TC4 if TC4.TCSchedule.PRIO_EN=1, and PM_Prio4[1:0] is higher than TC4.TCSchedule.PRIO[3:2]
- PM_Prio5[1:0]: Sets the priority of TC5 if TC5.TCSchedule.PRIO_EN=1, and PM_Prio5[1:0] is higher than TC5.TCSchedule.PRIO[3:2]
- PM_Prio6[1:0]: Sets the priority of TC6 if TC6.TCSchedule.PRIO_EN=1, and PM_Prio6[1:0] is higher than TC6.TCSchedule.PRIO[3:2]
- PM_Prio7[1:0]: Sets the priority of TC7 if TC7.TCSchedule.PRIO_EN=1, and PM_Prio7[1:0] is higher than TC7.TCSchedule.PRIO[3:2]
- PM_Prio8[1:0]: Sets the priority of TC8 if TC8.TCSchedule.PRIO_EN=1, and PM_Prio8[1:0] is higher than TC8.TCSchedule.PRIO[3:2]

The Policy Manager is controlled using the TCSchedule register (CP0 Register 2, Select 6).

- [0]: TCHALT_EN. Enables use of the external pins PM_TCHaltTC0 - PM_TCHaltTC8 and halts the corresponding thread.
- [1]: PRIO_EN. Allows the external pins PM_Prio0[1:0] - PM_Prio8[1:0] to adjust thread priorities.
- [3:2]: PRIO. Holds the thread priority level.

11 Inter-Thread Communication Unit

Inter-Thread Communication (ITC) Storage is a gating storage mechanism designed for low-level thread synchronization. Loads and stores to and from gating storage may block until the state of the storage location corresponds to some set of conditions required for completion. A blocked load or store can be precisely aborted if necessary, and restarted later.

In the I7200 core, the ITC storage is provided by the Inter-Thread Communication Unit (ITU). This block of logic resides outside of the core and connects to the core through the gating storage interface. SoC integrators are free to use the MIPS-supplied reference module, or to implement their own ITU module, or to not use ITC at all. This chapter describes the features of the sample ITU block supplied with the I7200 core. This block supports synchronization of TCs across multiple I7200 cores.

11.1 ITC Address Space

The ITC physical address space is defined by two, 32-bit tag registers: ITCAddressMap0 and ITCAddressMap1. These registers are referred to as ITC tags because they are accessed by the `CACHE` IndxLoadTag and IndexStoreTag operations in the same manner as cache tags. The AddressMap registers affect the address mapping of the overall ITU.

The tags in the ITC block are assessed using the `CACHE` instruction with a cache operation of index load tag and index store tag. Before using these operations on the ITC tags, set the ITC bit in the CP0 Error Control register. This setting directs the cache instruction to the ITC AddressMap registers instead of to the L1D-Cache.

- AddressMap0 is accessed using a cacheop with VA offset zero.
- AddressMap1 is accessed using a cacheop with VA offset 0x8.

Together the ITCAddressMap0 and ITCAddressMap1 tag registers specify a 2N aligned block of uncached memory.

- The ITCAddressMap0 register's BaseAddress field specifies the starting address of the ITC memory block.
- The ITCAddressMap1 register's AddrMask determines the size of the memory block, which can be varied from 1 KB to 128 KB.

Within this address space, ITC cells are spread out with a stride specified by the EntryGrain field. Tightly spaced cells save on memory space, but widely spaced cells spread across a number of TLB pages, permitting different cells to be mapped to different processes. The number of cells is specified by the NumEntries field.

Table 103: ITC AddressMap0 Register Format

31 10	9 1	0
BaseAddress	0	En

Table 104: ITC AddressMap1 Register Format

31	30 20	19 17	16 10	9 3	2 0
M	NumEntries	0	AddrMask	0	EntryGrain

Table 105: ITU AddressMap0 Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit			
BaseAddress	31:10	The top [31:10] bits of the ITC Physical Memory Mapped Block.	R/W	Undefined
Unused	9:1	Must be written as zeros; return zeros on read.	0	0
En	0	ITC enable.	R/W	0

Table 106: ITU AddressMap1 Field Descriptions

Register Fields		Address Map Register Tag1		Read/Write	Reset State
Name	Bits	Offset 8			
M	31	This bit indicates if another ITC block is defined along with another pair of ITC AddressMap registers. On the I7200 core, this value is hardcoded to 0.		R	0
NumEntries	30:20	Number of ITC cells present.		R	Preset
Unused	19:17	Must be written as zeros; return zeros on read.		0	0
AddrMask	16:10	Indicates which bits of the BaseAddress field should not participate in determining an ITC memory hit. This field effectively defines the size of the ITC memory block. Setting AddrMask to zero implies a 1 KB ITC address space, setting it to 0x7f implies a 128 KB address space.		R/W	Undefined
Unused	9:3	Must be written as zeros; return zeros on read.		0	0
Entry Grain	2:0	Interval spacing between the ITC cells. Cells are spaced at intervals of $128x2^{EntryGrain}$ bytes.		R/W	Undefined
		Encoding	**Size in bytes**		
		0x0	128		
		0x1	256		
		0x2	512		
		0x3	1024		
		0x4	2048		
		0x5	4096		
		0x6	8192		
		0x7	16384		

Depending on the setting of the AddrMask, NumEntries, and EntryGrain, it is possible that ITC cells do not fill up the entire ITC address block. If for example, two cells are mapped to a 1KB area with a stride of 256B (EntryGrain equal to 0x1), the first cell starts at offset 0x000 and the second at offset 0x100. The remaining two 256B regions starting at offsets 0x200 and 0x300 do not map to any storage. Any access to an address that does not map to an ITC entry will result in undefined behavior. It is also possible to set the ITC registers in a way that makes some of the cells unavailable.

11.2 ITC Storage

The number and type of ITC cells implemented in the ITU is configurable. The possible configurations are: 0, 1, 2, 4, 8, or 16 four-entry FIFOs and 0, 1, 2, 4, 8, or 16 single-entry Semaphores. If the implementation includes both types of cell, the FIFO cells will be grouped before the Semaphore cells. N number of FIFO cells will be located at cell addresses 0 to N-1. M number of Semaphore cells will be located at cell addresses N to N+M-1. The actual physical address is dependent on the base address and cell spacing. See *ITC Address Space* on page 167 for more information on addressing.

The reference ITU supports two kinds of storage cells: four-entry FIFO queues and single-entry Semaphore cells. All ITC cells are composed of the tag and data portions. In the single-entry cells, the data is 32 bits wide. The FIFO cells store four 32-bit data values. Although the memory space allows for 64-bit ITC cells, only the least-significant 32-bit words are present in this implementation. All ITC cells should be accessed as aligned 32-bit memory. Partial-word access such as LH or SB will result in undefined behavior.

11.3 ITC Views

All ITC cells can be accessed in one of 16 ways, called views, using standard load and store instructions. The view is encoded in bits 6:3 of the memory address, such that the successive views of a cell correspond to successive 64-bit-aligned addresses. The following table shows the addresses for the various views and the effects of using each of the views. If the ITC location is of type FIFO, the behavior of some of the views changes, and this is noted in the description of each view.

Table 107: Cell Views

Address[6:3]	View	Description
0x0	Bypass	This view of the ITC location implies that a load or a store does not cause the issuing thread to block and does not affect any of the cells state bits. The operation of SC using this view is undefined. Accesses using Bypass view never result in Gating Storage exceptions. A Bypass view store to a FIFO ITC location overwrites the newest FIFO entry, while a Bypass view load returns the contents of the oldest entry.
0x1	Control	This view of the ITC location can be used to manipulate the tag of the ITC cell. Loads and stores access the entire 32b tag value. Accesses using Control view never cause the issuing thread to block and never result in Gating Storage exceptions. A Control view store to a FIFO location with the *E* bit set will cause the FIFO to reset its read pointer.
0x2	Empty/Full Synchronized	This view of the ITC location implies that a load causes the issuing thread to block if the cell is Empty. Similarly, a store blocks if the cell is full. Accesses using this view cause an automatic update of the *Empty*and *Full*bits to reflect the new state of the cell. The operation of SC using this view is undefined. If the *T* bit is set, then all E/F Synchronized view accesses, success or failure, cause a gated exception trap.

Address[6:3]	View	Description
0x3	Empty/Full Try	This view of the ITC location is similar in nature to the previous E/F Synchronized view in most respects other than the waiting policy on an access failure. It is to be used if the issuing thread can potentially find something else to do and does not wish to be blocked if the access fails. A load with this view returns a value of zero if the cell is Empty, regardless of actual data contained. Otherwise the load behaves as in the E/F Synchronized case. Normal Stores to Full locations through the E/F Try view fail silently to update the contents of the cell, rather than block the thread. SC (Store Conditional) instructions referencing the E/F Try view will indicate success or failure based on whether the ITC store succeeds or fails. If the T bit is set, then all E/F Try view accesses, success or failure, cause a gated exception trap.
0x4	P/V Synchronized	This view of the ITC location does not modify the Empty and Full bits, both of which are assumed to be cleared as part of the cell initialization routine. Loads with this view return the current cell data value if the value is non-zero, and cause an atomic post-decrement of the value. If the cell value is zero, loads block until the cell takes a non-zero value. Normal Stores cause an atomic increment of the cell value, up to a maximum of 0xFFFF at which point the value saturates. Loads check the least significant 16 bits of the cell for a 0x0 irrespective of load size. The operation of SC using this view is undefined. If the T bit is set, then all P/V Synchronized view accesses, success or failure, cause a gated exception trap. P/V Synchronized view accesses are not allowed to FIFO ITC locations.
0x5	P/V Try	This view of the ITC location is similar in nature to the previous P/V Synchronized view in most respects other than the waiting policy on an access failure. It is to be used if the issuing thread can potentially find something else to do and does not wish to be blocked if the access fails. A load with this view returns a value of zero even if the cell con- tains a data value of 0x0. Otherwise the load behaves as in the E/F Synchronized case. Normal stores using this view cause a saturating atomic increment of the cell value (saturating to 0xFFFF), as described for the P/V Synchronized view, and cannot fail. The operation of SC using this view is undefined. If the T bit is set, then all P/V Try view accesses, success or failure, will cause a gated exception trap. P/V Try view accesses are not allowed to FIFO ITC locations.
0x6 - 0xF	Reserved	These views are reserved and should not be used by software.

The Control View of an ITU cell contains the state of that individual cell, i.e., control bits that regulate accesses to that cell. In addition to the E (Empty) and F (Full) fields specified by the MT ASE, the tag contains four implementation-specific fields: T, FIFO, FIFODepth, and FIFOPtr.

- The FIFO and FIFODepth fields indicate whether a cell is a FIFO and its depth.
- The FIFOPtr indicates how many elements are currently in a FIFO; this field is always zero for single-entry cells. The FIFOPtr can be reset by writing 1 into the E field of a FIFO.
- The T field indicates whether a Gating Storage exception should be signaled on an E/F or Proberen/Verhogen (P/V) view access to the cell. In the P/V semaphore, Proberen and Verhogen mean test and increment, respectively.

The following table shows the Control View format.

Table 108: Control View Format

Name	Bit	Description	Read/Write	Reset State
FIFODepth	31:28	Log2 of the cell depth. This field is set to 0x0 for single- entry cells, and to 0x2 for four-entry FIFO cells.	R	Preset
Unused	27:21	Must be written as zeros; return zeros on read.	0	0
FIFOPtr	20:18	This field indicates the number of elements in a FIFO cell, and always reads zero for single-entry Semaphore cells.	R	0
FIFO	17	Indicates the cell type. 1 for FIFO cells and 0 for single-entry Semaphore cells.	R	Preset
T	16	Trap Bit. When set, this bit causes the processor to take a Gating Storage Exception on PV or EF accesses. (Could be used by the OS to reuse the TC.)	R/W	Undefined
Unused	15:2	Must be written as zeros; return zeros on read.	0	0
F	1	Full Bit. This bit indicates that the cell is full.	R/W	Undefined
E	0	Empty Bit. This bit indicates that the cell is empty. Writing 1 to this bit also reset FIFOPtr.	R/W	Undefined

11.4 Programming Examples

This section provides three C code examples for programming the ITU:

- Configuring the ITC block
- Setting up a semaphore cell
- Using the ITC for semaphores

The code uses the following include files for `#defines`:

```
#include <mips/mt.h>
```

The example code uses the following defines:

```
// ITC defines not currently in include files
#define ERRCTL_ITC            0x04000000  /* ITC select for cache opts in ErrCtl register (26,0)
 */
#define ITC_BypassView        0x00000000
#define ITC_ControlView       0x00000008
#define ITC_EmptyFullSyncView 0x00000010
#define ITC_EmptyFullTryView  0x00000018
#define ITC_PVSyncView        0x00000020
#define ITC_PVTryView         0x00000028
#define ITC_En                0x00000001
#define ITC_E                 0x00000001

// ITC configuration defines
// ITC Block must be aligned on a 512 boundary to allow for View bits
unsigned int ITC_Block[1024] __attribute__((aligned(1024)));
unsigned int *ITC_BlockNC;
unsigned int *ITC_FirstPVCell;
#define ITC_AddrMask 0            // 1K address space
#define ITC_EntryGrain 0          // 128 bytes between Entries (Cells)

// Variables for ITC
unsigned int errctlreg;
unsigned int errctlreg_withITC;
unsigned int ITC_Config_Tag0;
unsigned int ITC_Config_Tag8;
unsigned int ITC_CellTag;
unsigned int *ITC_Cell;
```

11.4.1 Configuring the ITC Block

This example code shows how to configure the ITC block. The general steps are:

1. Set the cache controller to access the ITC tags.
2. Set the cell size and spacing.
3. Set the ITC block base address.
4. Return the cache controller to access the cache tags.

Set the Cache Control to Address ITC Tags

Use the CACHE instruction to read or write the ITC block tags. First, set the ITC bit in the Error Control register. This setting directs cache instruction operations to the ITC tags instead of the normal cache tags. Next, use the CACHE instruction to read or write to the ITC block tags.

```
// Configure the ITC tags. This is done by using
// cache opts. To set the cache mode for ITC tags
// the ITC bit in the ErrCtl register must be set

errctlreg = mips32_geterrctl();
errctlreg_withITC = errctlreg | ERRCTL_ITC;
mips32_seterrctl(errctlreg_withITC);
```

Configure the Address Mask Bits and Entry Grain

The address mask sets the size of the ITC cell and the Entry Grain sets the distance between the cells. Refer to *Table 106: ITU AddressMap1 Field Descriptions* on page 168 for a detailed description of the fields.

```
// configure Number of entries Address mask bits
// and Entry Grain in ITC tag index 8
// Setup the ITC_Config_Tag8 variable with the Address mask and Grain values
    ITC_Config_Tag8 = ((ITC_AddrMask << 10) | ITC_EntryGrain);
// Write it to the CP0 Data Tag Low Register
    mips32_setdtaglo(ITC_Config_Tag8); // write the tag to the CP0 TagLo register
// Inline Assembly to use the cache instruction operation Index Store Tag
// to write tag 1
// NOTE the 8 offset used to write tag 1
    __asm__ volatile
      ("\
     cache 9, 8($0);  \
     ehb; \
     "
      ) ;
```

Configure the Base Address and Enabling the ITC Block

Configure the ITC tag 0 with the base address of the ITC block and enable the ITC block. Refer to *Table 105: ITU AddressMap0 Field Descriptions* on page 168 for a detailed description of the fields.

The ITC_Block variable was declared as an array to set aside a memory area for the ITC block. It's starting address is used as the ITC block's base address (for convenience in this example, you can choose your own address).

Note: The base address must be a physical address.

kseg0 cacheability is configured via the config0.K0 field (bits[2:0]). To make kseg0 uncached, write the value 0x2 into those bits.

Note: In the I7200 core, config0.K0 defaults to 0x2 after reset. If a TLB mapping is used, the page needs to set its "C" attribute (e.g., via EntryLo bits [5:3] to 0x2 (uncached).

If your core is configured to use an MPU, use the address of the ITC_Block directly.

```
ITC_BlockNC = ITC_Block;
```

If your core is not using an MPU the ITC_Block address should be converted from a virtual address to a physical address by:

- Stripping the top bit if TLB mapping is used.
- Setting the VA bits [31:29] to 0x4 to access it via kseg0 (e.g., use address range 0x80000000 to 0x9FFFFFFF).

```
ITC_BlockNC = (unsigned int *) ((unsigned int)ITC_Block & 0x7fffffff);
```

Next set ITC Tag 0 with the address and enable bits.

```
// setup the tag variable with the address, and the enable bit
ITC_Config_Tag0 = ((unsigned int) ITC_BlockNC | ITC_En);
// Write it to the CP0 Data Tag Low Register
mips32_setdtaglo(ITC_Config_Tag0);
// Inline assemble to use the cache instruction operation Index Store Tag
// to write the tag 0
// NOTE the 0 offset used to write tag 0
__asm__ volatile
     ("\
     cache 9, 0($0); \
     ehb; \
     "
     ) ;
mips32_seterrctl(errctlreg);  // return ErrCtl to it previous state
```

11.4.2 Set up a Semaphore Cell

Now that the ITC Block is enabled, the next step is to enable an ITC Cell using an uncached address to access the cell.

- If your core is configured to use an MPU, the address must be configured to be uncached. Refer to the MPU chapter for more information on using a region for that function.
- If you are not using an MPU, change the address of the cell array from a kseg0 to kseg1 address by changing bit 29 of the address.

  kseg1 is strictly uncached; kseg0 can be uncached, therefore, it can also be used to access the ITU.

```
ITC_BlockNC = (unsigned int *) ((unsigned int)ITC_Block | 0x20000000);
```

Each cell view looks like an offset into the cell. To reset the cell, use the ITC Control View (also known as the cell tag). Refer to *Table 108: Control View Format* on page 171 for a detailed description of the fields.

Note: You do not need to use a CACHE operation to access the cell tag; access it directly by its address.

To obtain the address for the control view:

```
ITC_Cell =(unsigned int *)((unsigned int)ITC_BlockNC | ITC_ControlView);
```

Next set the Empty bit to initialize the cell for use.

```
*ITC_Cell = ITC_E;
```

Finally, increment the ITC cell to allow the first thread to not block. This cell is used as a semaphore; use the PVSyncView to increment the cell.

```
ITC_Cell =(unsigned int *)((unsigned int)ITC_BlockNC | ITC_PVSyncView);
*ITC_Cell = 1;
```

11.4.3 Use the ITC for Semaphores

After completed the previous steps, the cell is ready for use. Read the cell in the code that should be controlled by the semaphore cell:

```
ITC_Cell =(unsigned int *)((unsigned int)ITC_BlockNC |ITC_PVSyncView);
ITC = *ITC_Cell;
```

The read blocks if the cell contains a 0. Execution continues when the cell is incremented.

When the code is finished with the critical section that the semaphore is protecting, write to the cell:

```
*ITC_Cell = ITC;
```

Note: All writes to an ITC requires an explicit to memory barrier to complete (e.g., SYNC).

12 Instruction, Data, and Unified Scratch Pad RAM

The I7200 supports the option of adding high-speed local memory blocks, called Scratch Pad RAM (or SPRAM), when building the core. These blocks provide low-latency storage for critical code or data. SPRAM access speed is similar to that of locked cache lines, but without the impact on cache performance or maintenance.

SPRAM memory blocks are accessible by instruction fetches or data reads/writes.. These blocks are addressed separately from main memory, so it is possible to for the SPRAM addresses to overlap each other and main memory. There can be only one continuous physical address range for each Instruction or Data SPRAM block.

USPRAM is an SPRAM structure that is accessible by all MIPS CPU cores in the I7200 SoC. USPRAM access is shared across Instruction and Data accesses. The latency is longer than caches, but this structure provides a faster alternative to L2 access or data access between cores when coherency is not needed.

The SPRAM array, like the cache arrays, is indexed with a virtual address and the tag comparison is performed using a physical address. Because the SPRAM size can be larger than the 4 KB minimum page size, it is possible to have virtual aliasing in the SPRAM if using a TLB. Virtual aliasing occurs when a single physical address is accessed via two different virtual addresses that can simultaneously reside in memory. This function is not handled by hardware, and programmers must be aware of it. One method is to make one TLB entry that covers the whole SPRAM memory: make the page size for the TLB entry the same size as the SPRAM.

SPRAM blocks can be as small as 4 K and as large as 1 MB. The tags and sizes of all SPRAMs are configured at build time and are not modifiable by software.

Software must ensure an SPRAM entry has been initialized before it is read to avoid reading spurious data. Later sections in this chapter provide code examples.

Implementation notes:

- In most systems the BSS section is cleared when the processor is being initialized or a process is first brought into memory. If you configure your data SPRAM after this clear step to an address that covers the BSS section, you will need to re-clear the area.
- If the area that the Data SPRAM memory is replacing was already in use and it was cached, first flush and invalidate any cache lines that correspond to the physical memory being overlaid. Because SPRAM access supersedes the data cache hit, it is impossible to reference the data cache at the overlaid addresses.
- If you are using a TLB, it is possible to have a virtual address mapped to a cached area and another virtual address mapped through the SPRAM with both using the same physical memory. In this case, both cache and SPRAM memory can have updated values with one not seen by the other. Avoid this scenario.
- SPRAM can be mapped to either cached or uncached space. The address decode and comparison for SPRAM is performed regardless of the cacheability attribute.

12.1 DSPRAM Prediction Buffer

By default, the DSPRAM prediction buffer bit, located at CP0[22] s3 bit[4], is enabled at reset. This allows the core to monitor and remember addresses that hit the DSPRAM. On subsequent DSPRAM access, the core will disable the D-cache to converse power. In TLB configurations, this feature does require the virtually addresses to be mapped contiguously and naturally-aligned to the size of the DSPRAM (software cannot map two non-contiguous or partial pages to the DSPRAM). If this cannot be done, then software must disable the prediction capability by clearing the DSPPB_EN bit at CP0[22]s3 bit[4]. Cores configured with an MPU do not have this restriction.

12.2 SPRAM Examples

The SPRAM configuration size and address information is kept in Tag registers that are read using the CACHE instruction. There are a few simple steps needed to read the ScratchPad RAM configuration of your core.

SPRAM Address

The following code shows how to read the SPRAM size and address.

1. Read the CP0 Configuration Register to check for the existence of the SPRAM.
2. Set the SPR bit in the CP0 Error Control Register to direct the CACHE instruction to the SPRAM.
3. Use the CACHE instruction to read the SPRAM tag, which contains the location of the SPRAM blocks.

The code uses macros defined at the end of this chapter and includes the following files from the gcc toolchain:

- mips/asm.h
- mips/m32c0.h
- mips/regdef.h
- mips/cpu.h

First, read the CP0 Config0 register that contains the Scratchpad existence bits:

Table 109: Config Register (CP0 #16 - Select 0)

31	30 28	27 25	24	23	22	21	20	19	18	17	16	15	14 13	12 10	9 7	6 3	2 0
M	K23	KU	ISP	DSP	UDI	0	MDU	0	0	0	BM	BE	AT	AR	MT	0	K0

```
unsigned int my_Config ;

my_Config = mips32_getconfig0();

if ((my_Config >> 23) & 1) {
    // DSPRAM block0 exists
}

if ((my_Config >> 24) & 1) {
    // ISPRAM block0 exists
}
```

Set the SPR bit in the CP0 Error Control Register so cache operations are directed to the SPRAM controller:

Table 110: Error Control Register (CP0 #26 - Select 0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18 12	11 4	3 0
PE	PO	WST	SPR	0	ITC	LBE	WABE	0	L1ECC	0	SE	PE	0	PI	PD

```
unsigned int my_errctl, my_errctlS ;
my_errctlS = mips32_geterrctl();
// set the SPR bit:
my_errctl = my_errctlS | 0x10000000; // set bit 28
// Write it back to CP0 Error Control
mips32_seterrctl(my_errctl);
```

Read the SPRAM tag registers to get the SPRAM address:

Table 111: DTagLo (CP0 #28 - Select 2 D Tags), ITagLo (CP0 #28 - Select 0 I Tags)

tag	31 20	19 12	11											0
0	Physical Base Address													

Tag 0 contains the physical address bits 12 through 31 of the SPRAM block. These address bits are set to the default values the core was configured with at build time.

Tag 1 contains the number of 4 K sections of the SPRAM block in bits 12 through 19.

```
// Read the base address of the SPRAM
unsigned int my_TagLo;
index_load_tag_i (0);                 // read the SPRAM tag 0 into the I tag lo register
my_TagLo = getitaglo();               // read the I tag lo register into my_TagLo
base_paddr = my_TagLo & 0xfffff000;   // extract the SPRAM base address

// Read the SPRAM size
index_load_tag_i (8);                // read the SPRAM tag 1 into the I tag lo register
my_TagLo = getitaglo();              // read the I tag lo register into my_TagLo
block_size = my_TagLo;               // extract the size of the SPRAM
```

Enabling the Data, Instruction, and Unified SPRAM

The SPRAMs are enabled by setting bits in the CP0 register 22 Select 3.

- Setting bit 0 enables DSPRAM
- Setting bit 1 enables ISPRAM
- Setting bit 2 enables USPRAM

Using Instruction SPRAM

To put your code into the scratchpad, load the instruction SPRAM block from main memory using the CACHE instruction, index store Data. The simplest method is to load the code into main memory and then position the Instruction SPRAM Block directly over it. While the Instruction SPRAM Block is disabled, use the CACHE instruction to copy the code from main memory to the SPRAM block. Finally, enable the Instruction SPRAM and it is ready to use.

Note: Using two data tag registers switches the order of the reads from these registers depending on the endianness.

Using the Data SPRAM

The Data SPRAM block appears to your program as normal physical memory, so you simply need to enable the block. You do not need to have underlying main memory when you are using the Data SPRAM.

Configuring the Data SPRAM at core build time with a DMA interface makes it efficient as a staging area for communicating with complex I/O devices. For example, you can implement a push style I/O in which the device writes incoming data close to the CPU. Another advantage is using Data SPRAM for DMA buffers; it does not have cache management issues such as flushing and invalidating cache lines.

Using the Unified SPRAM

The Unified SPRAM block is similar to the data SPRAM in that it appears to be a single contiguous piece of normal memory accessible by any CCA. It is different in that it can be accessed by both data read/writes and instruction fetches from all cores and DMA.

The USPRAM is configured at build time. Per core access is enabled by the COP0 $22 select 3 bit [2] within each CPU. The USPRAM design is non-coherent; therefore, to ensure data is visible across all cores, software must ensure implement a semaphore or synchronizing system between cores.

Note: Unlike the instruction and data SPRAMs, USPRAM cannot be accessed by the CACHE instruction.

12.3 SPRAM Macros

```
// Use these to access SPRAM configuration tags:
#define Index_Store_Data_I       0x0c  // not defined in m32c0.h
#define index_store_data_i(x) \
({ unsigned int X; \
 X=x; \
asm volatile ("cache %0, 0(%1)" : : "i" (Index_Store_Data_I), "r" (X));})

#define setidatahi(x) \
({ unsigned int X; \
 X=x; \
asm volatile ("mtc0 %0, $%1, 1": : "r" (X), "i" (C0_TAGHI));})

#define setidatalo(x) \
({ unsigned int X; \
 X=x; \
asm volatile ("mtc0 %0, $%1, 1": : "r" (X), "i" (C0_TAGLO));})

#define getidatahi() \
({ unsigned int __value; \
asm volatile ("mfc0 %0, $%1, 1" : "=r" (__value) : "i" (C0_TAGHI)); \
__value;})

#define getidatalo() \
({ unsigned int __value; \
asm volatile ("mfc0 %0, $%1, 1" : "=r" (__value) : "i" (C0_TAGLO)); \
__value;})
```

13 Hardware and Software Initialization

A I7200 core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

13.1 Hardware-Initialized Processor State

The I7200 core is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. Unlike previous MIPS processors, there is no distinction between cold and warm resets (or hard and soft resets). SI_Reset is used for both power-up reset and soft reset.

Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0:

- Random - cleared to maximum value on Reset
- Wired - cleared to 0 on Reset
- $\text{Status}_{\text{BEV}}$ - set to 1 on Reset
- $\text{Status}_{\text{TS}}$ - cleared to 0 on Reset
- $\text{Status}_{\text{NMI}}$ - cleared to 0 on Reset
- $\text{Status}_{\text{ERL}}$ - set to 1 on Reset
- $\text{Status}_{\text{RP}}$ - cleared to 0 on Reset
- $\text{CDMMBase}_{\text{EN}}$ - cleared to 0 on Reset
- $\text{WatchLo}_{\text{I,R,W}}$ - cleared to 0 on Reset
- Config fields related to static inputs - set to input value by Reset
- $\text{Config}_{\text{K0}}$ - set to 010 (uncached) on Reset
- $\text{Config}_{\text{KU}}$ - set to 010 (uncached) on Reset
- $\text{Config}_{\text{K23}}$ - set to 010 (uncached) on Reset
- Debug_{DM}-cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into Debug Mode).
- $\text{Debug}_{\text{LSNM}}$ - cleared to 0 on Reset
- $\text{Debug}_{\text{IBusEP}}$ - cleared to 0 on Reset
- $\text{Debug}_{\text{DBusEP}}$ - cleared to 0 on Reset
- $\text{Debug}_{\text{IEXI}}$ - cleared to 0 on Reset
- $\text{Debug}_{\text{SSt}}$ - cleared to 0 on Reset

TLB Initialization

Each TLB entry has a hidden state bit, which is set by Reset and is cleared when the TLB entry is written. This bit disables matches and prevents TLB Shutdown conditions from being generated by the power-up values in the TLB array (when two or more TLB entries match a single address). This bit is not visible to software.

Bus State Machines

All pending bus transactions are aborted and the state machines in the bus interface unit are reset when a Reset exception is taken.

Statis Configuration Inputs

All static configuration inputs (for example, those defining the bus mode and cache size) should only be changed during Reset.

Fetch Address

Upon Reset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in kseg1, which is unmapped and uncached, so that the TLB and caches do not require hardware initialization.

13.2 Software-Initialized Processor State

Software is required to initialize parts of the device, as described below.

Register File

The register file powers up in an unknown state with the exception of r0, which is always 0. Initializing the rest of the register file is not required for proper operation. Good code will generally not read a register before writing to it, but the boot code can initialize the register file for added safety.

TLB

Because of the hidden bit indicating initialization, the core does not initialize the TLB upon Reset. This is an implementation-specific feature of the I7200 core and cannot be relied upon if writing generic code for MIPS processors.

Caches

The cache tag and data arrays power up to an unknown state and are not affected by reset. Every tag in the cache arrays should be initialized to an invalid state using the CACHE instruction (typically the Index Invalidate function). This can be a long process, especially because the instruction cache initialization must run in an uncached address region.

Coprocessor 0 State

Miscellaneous COP0 states need to be initialized before exiting the boot code. There are various exceptions that are blocked by ERL=1 or EXL=1, and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- **Cause:** WP (Watch Pending), and SW0 and SW1 (Software Interrupts) should be cleared.
- **Config:** K0 shouldbe set to the desired Cache Coherency Algorithm (CCA) prior to accessing kseg0.
- **Count:** Should be set to a known value if timer tnterrupts are used.
- **Compare:** Should be set to a known value if timer tnterrupts are used. Note that the write to Compare will also clear any pending timer interrupts, so Count should be set before Compare to avoid any unexpected interrupts.
- **Status:** Desired state of the device should be set.
- **Other COP0 state:** Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

13.3 Boot and CMP Bringup

After the system is reset and released, all cores configured in hardware to power up will execute their boot sequence. Typically, CPU0 powers up, while all other CPUs are configured to remain powered down. Alternatively, all CPUs can be hardware configured to remain powered down to be awakened through a hardware signal connected to SOC-specific logic.

After system reset, all caches are in an unknown state and must be initialized. It is advisable for core0 to initialize the L2 cache prior to powering up the other cores, but this is not required if other synchronization methods are utilized. For L1 caches, this is expected to be done using IndexStTag ops running on the same CPU. Prior to the data cache being initialized, processing an intervention would cause unpredictable results, potentially corrupting main memory with random data. Thus, the system starts with all of the cores outside the coherence domain until explicitly enabled by software.

```
Core0:
Initialize cop0 state
Initialize L2 Cache
Initialize GCR state
Startup other cores if needed
CoreN:
Initialize L1 Caches
Enable Coherence
Switch to coherent CCA
```

13.4 Hazard Barrier Instructions

When privileged CP0 instructions change the machine state, unexpected behavior can occur if an instruction is deferred out of its normal instruction sequence. However, that behavior can happen because the relevant control register only gets written down the pipeline, or because the changes it makes are sensed by other instructions early in their pipeline sequence. This situation is called a CP0 hazard.

Traditionally, MIPS CPUs the kernel/low-level software engineer had to design sequences that were guaranteed to run correctly, usually by padding the dangerous operation with enough NOP instructions. To help manage pipeline hazards, the I7200 core implements explicit hazard barrier instructions. If a hazard barrier instruction is executed between the instruction that makes the change (the producer) and the instruction that is sensitive to it (the consumer), the change is seen as complete. Hazards can appear when the producer affects the consumer's instruction fetch (instruction hazard) or it can affect the operation of the consuming instruction (execution hazard).

The I7200 core has these hazard barriers:

- EHB deals only with execution hazards
- ERET, ERETNC, and JALRC.HB are barriers to both kinds of hazard

In most implementations, the strong hazard barrier instructions are quite costly, often discarding most or all of the pipeline contents. They should not be used indiscriminately. For efficiency you should use the weaker EHB where it is sufficient. Because some implementations work by holding up execution of all instructions after the barrier, it is preferable to place the barrier just before the consumer, not just after the producer.

The following tables list the execution hazards and the instruction hazards for the I7200 core.

Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. The following table lists possible execution hazards.

Table 112: Execution Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI , TLBINV, TLBINVF	→	Load/store using new TLB entry	TLB entry
MTC0	→	Load/store affected by new state	WatchHi, WatchLo
MTC0	→	MFC0	Any CP0 register
MTC0	→	EI, DI	Status
MTC0	→	RDHWR $3	Count
MTC0	→	Coprocessor instruction execution depends on the new value of $\text{Status}_{\text{CU}}$	$\text{Status}_{\text{CU}}$
MTC0	→	ERET	EPC,DEPC, ErrorEPC
MTC0	→	ERET	Status
EI, DI	→	Interrupted instruction	$\text{Status}_{\text{IE}}$
MTC0	→	Interrupted instruction	Status
MTC0	→	User-defined instruction	$\text{Status}_{\text{ERL}}$, $\text{Status}_{\text{EXL}}$
MTC0	→	Interrupted instruction	$\text{Status}_{\text{IM}}$ (Cause_{IP})
TLBR	→	MFC0	EntryHi, EntryLo0, EntryLo1, PageMask
TLBP	→	MFC0	Index
MTC0	→	RDPGPR, WRPGPR	$\text{SRSCtl}_{\text{PSS}}$
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt
MTC0	→	Instruction affected by change	Any other CP0 register
CACHE	→	MFC0	TagHi, TagLo, DataHi, DataLo
MTC0	→	SC	LLAddr
LL	→	MFC0	LLAddr

Execution Hazards on Memory-Mapped Register

Memory mapped control registers (such as MPU_Config that resides in the CDMM space, or the IBC register in the dseg space) are accessed with uncached SW and LW instructions rather than an MTC0 instruction. To ensure these instructions take effect immediately, a SYNC instruction is required to push the write-data out of the pipeline and internal buffers.

Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. The following table lists the instruction hazards. Because the fetch unit is decoupled from the execution unit, these hazards are rather large. The use of a hazard barrier instructions is required for reliable clearing of instruction hazards.

Table 113: Execution Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI, TLBINV, TLBINVF	→	Instruction fetch using new TLB entry	TLB entry
MTC0	→	Instruction fetch seeing the new value including: • Change to ERL followed by an instruction fetch from the useg segment • Change to ERL or EXL followed by a Watch exception	Status
MTC0	→	Instruction fetch seeing the new value	$\text{EntryHi}_{\text{ASID}}$
MTC0	→	Instruction fetch seeing the new value	WatchHi, WatchLo
Instruction stream write via CACHE	→	Instruction fetch seeing the new instruction stream	Cache entries
Instruction stream write via store	→	Instruction fetch seeing the new instruction stream	Cache entries

14 Multithreading Overview

Multithreading is the ability of a single core in the multi-core I7200 to execute multiple processes or threads concurrently. Multithreading can increase the overall core efficiency. The operating system manages the thread distribution, and is responsible for ensuring that all threads can run simultaneously without interfering with each other.

The I7200 MPS implements the MIPS® Multi-Threading (MT) Application Specific Extension (ASE). The MT ASE provides hardware support for multithreading software applications using Virtual Processing Elements (VPEs) and Thread Contexts (TCs). With multithreading, an I7200 core can execute software applications in fewer cycles than on a typical single-threaded core.

14.1 Thread Context Resource Allocation

Resources are allocated at IP configuration time depending on the number of cores, VPEs, and TCs in the system. The resources common to the core level are:

- Pipeline
- Instruction Fetch Unit
- L1 Caches
- Load Store unit
- Multiply-Divide unit
- Arithmetic logic unit
- Memory
- CP0 registers that are shared by all VPEs
- Up to three VPEs per core

In addition to the core, each VPE has its own:

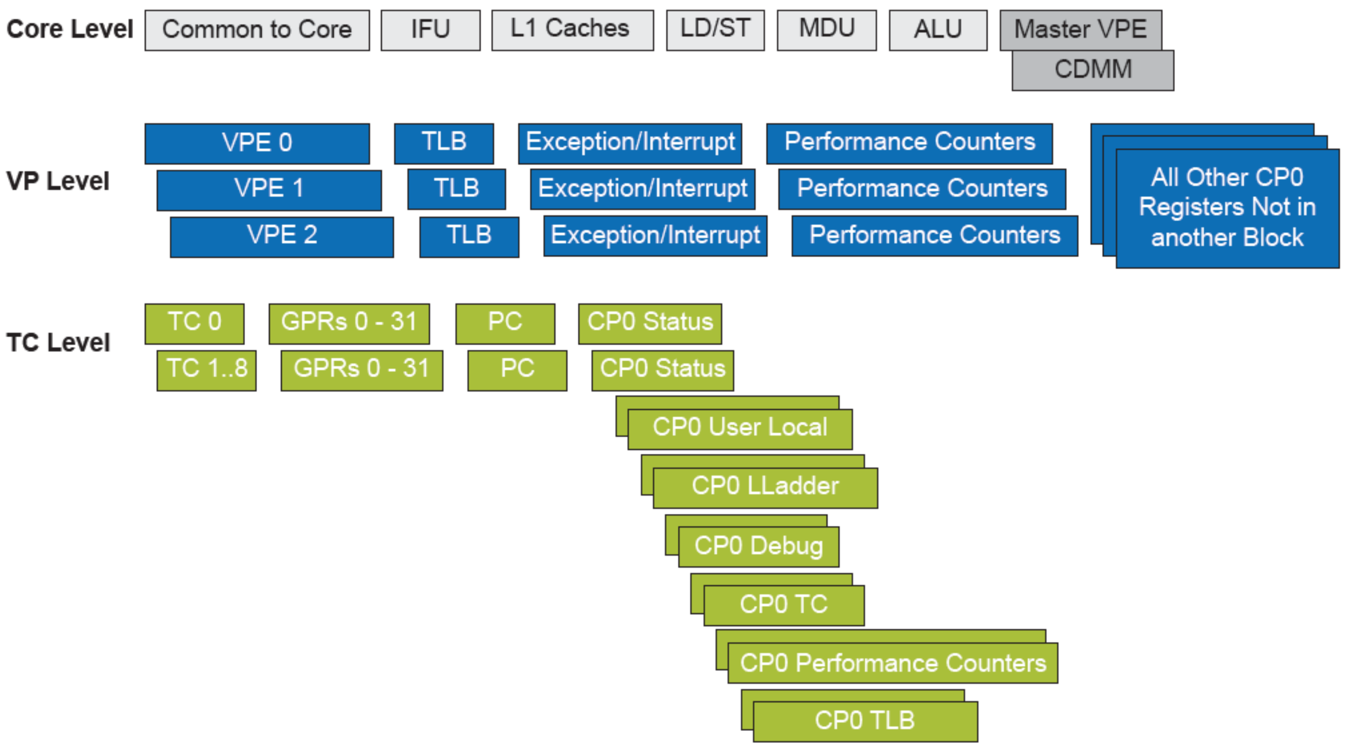
- TLB, if configured
- Exception and interrupt logic
- Performance counters
- All CP0 registers that are not common to the core are duplicated for each VPE. In addition to the Standard MIPS CP0 registers additional registers are defined per-VPE, and are common for all TCs within that VPE.

A core can have up to 9 TCs; each TC is associated with a specific VPE. In addition to the VPE, each TC has its own:

- General-purpose registers (GPRs)
- Internal program counter
- CP0 registers

The following figure shows how resources are distributed within the I7200 at the core, VPE, and TC levels.

Figure 36: I7200 Resource Allocation



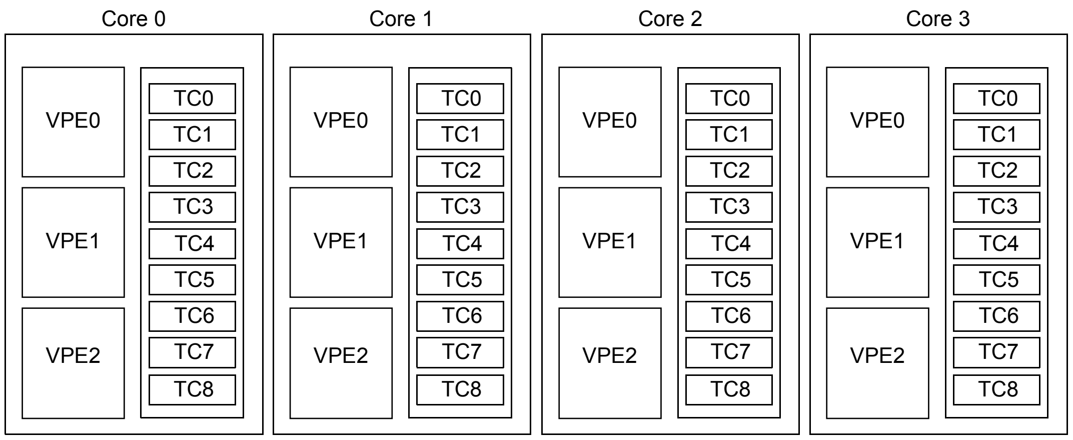
14.2 MT ASE Definitions

The MIPS MT ASE is an application-specific extension of the nanoMIPS32 instruction set and privileged resource architecture. These extensions allow an operating system to manipulate the hardware resources associated within a core with multiple VPEs and threads. The ISA extensions allow for efficient user-mode creation and destruction of threads to avoid OS intervention in typical cases. The MT ASE defines a Thread Context and a Virtual Processor as follows:

- A **Thread Context**, or **TC** is a sequential nanoMIPS32 instruction stream that contains the hardware state necessary to support an execution thread. Each TC includes a set of general-purpose registers (GPRs), a program counter (PC), and some multiplier and coprocessor state information. Each I7200 core can support up to 9 threads.
- A **Virtual Processing Element**, or **VPE**, is an instantiation of the full nanoMIPS32 ISA and Privileged Resource Architecture (PRA), sufficient to run a per-processor operating system image. A VPE can be thought of as an "exception domain" (because exception state and priority apply globally within a VPE) and only one exception can be dispatched at a time on a VPE. Each I7200 core supports up to three VPEs and each VPE must have at least one TC.

The following figure shows the maximum number of cores, virtual processors, and thread contexts the I7200 supports.

Figure 37: Maximum Number of VPEs and TCs in the I7200



14.2.1 Starting Thread Execution

Threads can be directly created in user mode with the FORK instruction. The FORK instruction takes three operands:

- The instruction address at which the new thread will begin execution.
- An arbitrary register value to be passed to the new thread, typically a pointer to a block of thread-specific storage.
- An output parameter, which is the register in the new thread's TC that will receive that second input operand value.

14.2.2 Multithreading Software Design Considerations and the FORK Instruction

Software should consider the following design considerations.

No Implicit Context Copy

Some high-level multithreading software paradigms require that each new thread receive a copy of the parent full register set. Others do not. A MIPS32 user-mode thread context consists of 31 GPRs (register 0 always being 0), the Hi/Lo or ACX/Hi/Lo accumulator, and some coprocessor state. The FORK instruction does not do a copy of the parent processes registers; instead it passes a single GPR value parameter to the newly spawned thread. If the computation requires more than a single value, this value can be a pointer to a context block in memory that contains register and other values needed by the thread computation. A FORK also implicitly propagates some privileged state, such as the contents of the ASID register, but the objective is to minimize the payload of a FORK operation. Minimizing the payload minimizes the hardware implementation cost, but also anticipates multicore remote FORKs, in which information is transmitted between processing elements.

No Value Provided to Forking Thread

Some high-level multithreading paradigms require that thread creation return a value to the *parent* thread performing the FORK operation. This value is use as a handle or tag for future operations that may reference the thread. The FORK instruction does not do this because:

- It would require a register-file write beyond the write of the GPR value parameter to the register file of the new thread, which creates an undesirable constraint on the design of multithreaded register files.

- It creates a name-space problem. In the course of its lifetime, the newly created software thread may end up executing on different register sets of the same CPU due to context switching, and it may even be migrated to some other processing element. Having hardware provide, as an output of the FORK instruction, a system-unique identifier that would follow each new thread, would be possible, but too complex to impose on small, embedded cores.

Where traceability is required, it can be accomplished using software-based memory interactions.

Absolute Virtual Thread Starting Address

The FORK instruction takes as an input operand a register value that is taken to be the starting instruction fetch address for the new thread.

14.2.3 Thread Overflow Exception

Issuing a FORK instruction when there are no free, dynamically allocatable TCs available on the VPE causes a thread *overflow* exception that system software can use to virtualize threads. A program can thus create and use a larger number of software threads than the available compliment of TCs, as long as the OS provides higher level scheduling to swap them in and out.

Issuing a FORK instruction when multi-TC scheduling is inhibited on the issuing VPE does not necessarily result in a failure or exception. As long as a TC can be successfully allocated, it is set up to run by the FORK operation and begins execution once TC scheduling is enabled.

14.2.4 Thread Suspension Using the YIELD Instruction

The YIELD instruction suspends thread execution. It takes as an input operand a descriptor of the circumstances under which the issuing thread should be resumed. If the operand has a value of zero, there are no circumstances under which the thread will resume, and it is de-allocated so that the associated TC may be re-used.

Negative descriptor values are reserved by MIPS for architecturally defined rescheduling conditions. A value of -1 requests that the YIELDing thread be rescheduled without waiting for any specific condition, but allowing other threads to *play through*, according to the implemented thread scheduling scheme. A value of -2 samples the YIELD qualifier inputs to the core without any rescheduling of the thread. Positive descriptor values represent a vector of up to 31 independent *YIELD Qualifier* bits, which are hardware inputs to the processor. This option is only available if a yield manager is configured.

The MIPS MT ASE provides mechanisms for an operating system to intercept and emulate YIELD operations. If a per-VPE enable is set, rescheduling YIELDs trap to the operating system with a designated exception code and subcode identifying a YIELD scheduler intercept. The operating system can:

1. Evaluate the current YIELD qualifier input state
2. Check it against the contents of the register specified by the trapping YIELD instruction
3. Make its own determination whether the YQ input values (or some synthetic value) should be placed in the YIELD’s destination register
4. Restart the TC at the instruction following the YIELD, or decide whether the TC contents should be swapped to memory and replaced with the context of another thread of execution

Each TC has a status bit that is set whenever an instruction is completed for that TC, outside of low-level exception handlers. This bit has multiple software uses, and it is further used by hardware to enable the YIELD scheduler intercept exception. An operating system that wishes to allow a TC to resume and remain blocked on a YIELD after handling a YIELD scheduler intercept exception can clear this *DT* (Dirty Thread) bit before restarting the TC on the YIELD. That particular TC remains blocked until the YIELD qualifiers are satisfied or until some other OS intervention takes place. If the YIELD completes due to the qualifiers being satisfied, the DT bit is set, and the next blocking YIELD issued by that thread traps if the YIELD scheduler intercept exceptions are still enabled.

14.2.5 Additional MT ASE Instructions

While user-mode multithreading is based on the FORK and YIELD primitives, the I7200 MT ASE also includes some privileged instructions to help manage thread and VPE resources.

- **MTTR** is a privileged, COP0 instruction that moves information from a register of the issuing thread to a register of another thread context on the same processor.
- **MFTR** is a privileged, COP0 instruction that moves information from a register of another thread context on the same processor to a register of the issuing thread.
- **EMT** is a privileged, COP0 instruction that atomically enables multithreaded issue on a VPE.
- **EVPE** is a privileged, COP0 instruction that atomically enables multi-VPE issue on a core with multiple VPEs enabled.
- **DMT** is a privileged, COP0 instruction that atomically disables multithreaded issue on a VPE.
- **DVPE** is a privileged, COP0 instruction that atomically disables multi-VPE issue on a core with multiple VPEs enabled.

Note: EMT, DMT, EVPE, and DVPE are all instances of the MFMC0 instruction.

14.3 Multithreading CP0 Registers

Certain privileged resources are required to manage the multithreading capabilities of a VPE. Multithreading CP0 registers are instantiated at the core level, the VPE level, and the TC level.

14.3.1 Per-Core Multithreading Registers

The following registers are instantiated at the core level:

- MVPControl register: Contains control bits for managing multi-VPE processors.
- MVPConf0 and optional MVPConf1 registers: Contain information about global multithreaded processor resources that can be configured at boot time and bound to different VPEs.

14.3.2 Per-VPE Multithreading Registers

The following registers are instantiated at the VPE level:

- VPEControl register: Contains information about the configuration of threads within a VPE.
- VPEConf0 and VPEConf1 registers: Contain per-VPE information about the multithreading resources available to theVPE.
- YQMask register: Allows certain YIELD qualifier bits to be masked, so that an attempt to suspend execution pending that state results in an exception.
- VPESchedule register: Allows for the hardware scheduling algorithms of a processor to be manipulated to guarantee some quality of service to VPEs with hard real-time requirements.
- SRSConf0 - SRSConf4 registers: Allow for run-time binding of TCs to Shadow Register Sets.

14.3.3 Per-TC Multithreading Registers

The following registers are defined to be per-TC:

- **TCStatus register:** Contains privileged resource information per-thread, such as the kernel/user state of the thread, or whether it has access to a coprocessor.
- **TCBind register:** Defines a TC's binding to a VPE.
- **TCRestart register:** Contains the restart fetch and execution address of a TC.
- **TCHalt register:** Allows a TC to be put into or taken out of a halted state with a single register write.
- **TCContext register:** Storage register implemented per-thread, which allows the OS to have instant access to a value, typically a memory pointer such as a kernel stack pointer, that is unique per-thread.

14.4 Thread-Level Exception Processing

By definition, parallelism at the VPE level introduces nothing new in the handling of exceptions for single-threaded VPEs within a multi-VPE core. In the explicit, fine-grained model, however, multiple threads of execution with multiple hardware thread contexts share common system coprocessor resources. This has a number of implications for hardware and software.

Because there is only one Cause register to contain the reason for an exception, a single VPE cannot manage concurrent exceptions. When a synchronous exception is provoked by a thread, as in the case of a TLB miss or a floating-point exception[22], the MIPS32 architecture stipulates that the EXL or ERL bits of the Status register be set, which blocks interrupts and further general exceptions from being taken.

In the MIPS MT ASE, the setting of EXL/ERL also prevents the scheduling of other threads until it is cleared by the exception handler. Short exception handling sequences like TLB miss handlers can reasonably be coded, and re-enable multithreading implicitly with the clearing of EXL by the ERET instruction.

More complex exception handling sequences, such as OS system calls, may explicitly re-enable the concurrent execution of non-privileged application threads by clearing EXL once the the OS has acquired and saved the Cause information.

For a synchronous exception, the TC associated with the instruction stream causing the exception is the one that is associated with the exception:

- If the exception is not bound to a shadow register set, the associated TC is used to execute the exception handler.
- If a shadow register set is used, the associated TC is used as the previous shadow set.

Asynchronous exceptions, such as interrupts, can be associated with any available activated TC, with the restriction that TCs used by real-time service threads may be designated as exempt from use by interrupt service routines by setting a the IXMT per-thread control bit.

If all activated TCs are explicitly blocked via YIELD instructions or uncompleted loads/stores of gating storage locations, asynchronous exceptions, including debug exceptions, must be associated with such a blocked TC. The associated handlers are executed using the previously blocked context, aborting the YIELD or load/store. The VPE resumes execution on an ERET by re-fetching and re-executing the YIELD or load/store. An aborted gating storage load or store must leave the state of the storage location as it would have been had the instruction never been issued.

Debug exceptions are special in several regards with respect to MIPS MT. Like other exceptions, they execute within the context of a specific VPE, but whereas the setting of EXL or ERL by a normal exception inhibits thread scheduling only within the affected VPE, debug mode execution inhibits thread scheduling across all core VPEs. While other asynchronous exceptions, such as interrupts, require a TC that is activated and not halted (though it may have been blocked) to process the exception, an asynchronous debug exception, such as that caused by the assertion of a DINT signal by a probe, can be serviced by any TC bound to the targeted VPE, regardless of its halt or activation state. This makes it possible for debuggers to recover from otherwise completely fatal OS errors, such as halting all TCs.

Note: For further details of debug extensions (such as synchronous halt for multiple VPEs) refer to the MIPS OCI 32-Bit Debug Specification.

14.5 Fine-Grained Multithreading

Finer-grained multithreading can exploit parallelism at levels that cannot be efficiently addressed by OS-level multithreading. The MT ASE implemented by the I7200 core allows threads of execution to be created and destroyed very inexpensively by user-mode code. This requires that the applications or underlying libraries be explicitly built or coded to use the new instructions, and also requires the appropriate OS support.

[22] The I7200 core does not support the FPU.

Fine-grained multithreading in the I7200 core is implemented as an execution model that allows multiple threads to exist within the context of one CPU. Threads share some CPU resources but are able to execute independently.

Each thread is scheduled by the policy manager this is a way of controlling the priority of each thread.

Dedicated Register Set

Each thread has its own set of general-purpose registers. This enables each stage of a multithreaded pipeline to contain instructions from different threads so that execution of those instructions a effects only the registers of the thread the instruction is from.

The CPU scheduling through the policy manager takes advantage of stalls in the CPU pipeline such as a cache miss.

Automatic Fine-Grained Multithreading

Automatic parallelization algorithms can be employed in compilers to generate multithreaded code. This technique is the ultimate means by which a single C or Fortran program can be accelerated in terms of execution clock time.

The necessary compiler techniques exist in the research and high-performance computing communities. They would need to be adapted to MIPS and used in conjunction with the OS support described above for explicitly fine-grained multithreading.

Operating system support for the fine-grained, FORK/YIELD parallelism of the MT ASE should include:

- Context switch code that dynamically checks the number of threads to be saved and restored each time a user task is switched.
- Fault handling code for thread exceptions, which occur when there is an underflow or overflow of the number of available physical TCs.
- Allocation and memory management code for ITC storage, if present, as special memory.

If threads at runtime without OS intervention are to be able to take nested exceptions, it is anticipated that the Thread-Context register value of each TC is unique. The OS start-up code would assign context storage for each TC on a processor, and insert a pointer to it into that thread’s Thread Context register prior to that thread’s being made available for FORK allocation.

14.6 Operating System Support

To provide the most optimum implementation of multithreading, system software should contains the features listed in the following subsections.

14.6.1 Thread Virtualization and Hybrid Scheduling

If more software threads are active in a system than there are TCs available in a MIPS MT VPE, it is necessary to impose a layer of software scheduling on top of the hardware thread scheduling policy of the processor. MIPS MT contains architectural hooks to support this thread virtualization.

Executing a FORK instruction when no dynamically allocatable TCs are free to accept the new instruction stream causes a thread overflow exception. This exception allows an operating system to detect the case of more software than hardware threads in systems where user-mode thread creation is allowed. If software threads are created only by the OS, the OS can track the available resources without an exception.

As long as the number of software threads does not exceed the TC resources available, it is of no consequence from the standpoint of system performance whether a TC remains blocked on a qualified YIELD or a gating storage access. However, when TCs are saturated, it becomes necessary to multiplex the software threads across the available TCs.

This can be achieved using simple scheduling algorithms that time-slice threads, regardless of whether or not they are making forward progress. For high efficiency, it is highly desirable to use blockages as an opportunity to schedule other software threads. The MIPS MT ASE provides the option for a VPE to take an exception whenever a YIELD could cause a rescheduling or whenever a gating storage access blocks.

If blockages will generally be of a short duration, generating exceptions on each blockage may not be desirable; it may be better to allow TCs to be blocked for some period of time before swapping out their contents. An operating system can do this by periodically sampling and clearing the dirty bit associated with each TC, which is set whenever the TC state is modified by instruction execution. If the dirty bit remains clear after a sample interval, it may be deduced that the TC has been blocked for the full interval.

14.6.2 Software Security

If dynamic FORK/YIELD thread creation and resource allocation is in use simultaneously in different security domains, i.e., by multiple applications or by both an OS and an application, there can be a risk of information leakage in the form of register values inherited by an application. It is the responsibility of a secure operating system to manage this risk.

The MIPS MT ASE provides one simple mechanism to facilitate this task; a dirty bit associated with each TC can be cleared by software and is set whenever the context is modified. An OS can initialize all TCs to a known clean state, and clear all associated dirty bits, prior to scheduling a task. On a task switch, dirty TCs must be scrubbed to the clean state before another task can be allowed to allocate and use them.

If a secure operating system wishes to make use of dynamic thread creation and allocation for privileged service threads, the associated TCs must be scrubbed before they are freed for potential use by applications.

The MIPS MT ASE provides no mechanisms to guarantee that two independent, untrusted tasks running concurrently on the same VPE and executing FORK and YIELD instructions, will not exchange TC storage, and thus register values. As such, programs that cannot trust one another should be run on distinct VPE’s.

14.6.3 Manipulating TC Dynamic Allocation Properties

Each TC has an associated DA bit, which makes it available for dynamic allocation by FORK instructions. The interactions of FORK and YIELD with the set of DA bits makes possible several TC management algorithms.

Interrupt-exempt real-time threads may have the DA bit of their associated TC cleared so that a YIELD 0 of the last dynamically allocated thread causes an underflow thread exception on the YIELDing thread without interfering with the realtime thread execution and without leaving the processor in a state where no interrupt-capable TCs are active.

In response to an overflow thread exception on a FORK, where no more DA TCs are available, the OS can, after having saved a copy of the previous values, clear the DA bits of all TCs, so that the next YIELD 0 will cause an underflow thread exception that can be used by the OS to restore DA bits and schedule a replay of the failed FORK.

14.6.4 Virtual Multiprocessor

Most mainstream operating systems implement some form of symmetric multiprocessing (SMP). Several Microsoft operating systems support SMP platforms, as does Linux. Multithreaded applications exist that exploit the parallelism of such platforms, using heavyweight threads provided by the operating system. The MIPS MT ASE is designed to provide maximum leverage to this technology.

To applications software, a multithreaded processor configured as two single-threaded VPEs is indistinguishable from a 2-way SMP multiprocessor. The operating system would have no need to use any of the new instructions or privileged resources defined by the ASE.

Each MIPS MT TC has its own interrupt exemption bit and its own MMU address space identifier (ASID), which allows operating systems to be modified or written to use a symmetric multi-TC (SMTC) model. In this model each TC is treated as an independent processor. Because multiple TCs may share the privileged resources of a single VPE, an SMTC operating system requires additional logic and complexity to coordinate the use of the shared resources. However, the SMTC model allows SMP-like concurrency up to the limit of available TCs.

14.6.5 Master and Slave VPEs

One or more VPEs on a processor may power-up as a master VPE, indicated by the MVP field of the VPConf0 register. A master VPE can access the registers of other VPEs by using MTTR/MFTR instructions, and can, via the DVPE instruction, suspend all other VPEs in a processor.

This master/slave model allows a multi-tasking master application processor VPE running an operating system such as Linux to dispatch real-time processing tasks on another VPE on behalf of various applications. While this could be done using an SMP paradigm, handing work off from one OS to another, MIPS MT also allows it to be done more directly.

A master VPE can take control of another VPE of the same processor at any time. Once a DVPE instruction has been issued by the master VPE:

1. The slave VPEs CP0 privileged resource state can be set up as needed using MTTR instructions targeting TCs that are bound to the slave VPE.
2. The necessary instructions and data can be set up in memory visible to the slave VPE.
3. One or more TCs of the slave VPE can be initialized using MTTR instructions to set up their TCRestart addresses (and their GPR register values, if appropriate).
4. The slave VPE can be dispatched to begin execution using the configured TCs by the master VPE executing an EVPE instruction.

14.7 Programming Example: Starting a Thread

This section provides C and Assembly code examples that show how to start a new thread executing.

1. Turn on virtual processor configuration and turn off threading.
2. Set target TC for next step.
3. Halt thread.
4. Bind TC to VPE.
5. Set stack pointer.
6. Set TC starting address.
7. Unhalt thread.
8. Turn off virtual processor control and turn on threading.

The code uses the following include files for `#defines`:

```
#include <mips/mt.h>
```

```
#include <mips/cpu.h>
```

Note: Register descriptions only cover the effective fields; there may be more fields in the register.

14.7.1 Turn on Virtual Processor Configuration and Disable Virtual Processing

First, the code need puts the processor into a mode in which it can use the CP0 registers to configure the threads to run.

The MVPControl register has 2 fields: VPC and EVP. Setting the VPC field allows write to registers that normally are not writable on a single core MIPS processor. The EVP field is cleared to disable all multi-processing to configure threads.

Table 114: MVPControl Register

Fields		MVPControl - CP0 #0-Sel1	Read/Write	Reset State
Name	Bits			
VPC	1	VPE Configuration State. If set, allows writing to normally read-only configuration register fields on conventional MIPS32 CPUs.	R/W	0
EVP	0	Enable Virtual Processors. If set, execute instructions for all threads on activated VPEs. If cleared, execute instructions only for thread which is running when cleared.	R/W	0

```
mips32_setmvpcontrol((mips32_getmvpcontrol() & ~MVPCONTROL_EVP) | MVPCONTROL_VPC);

Assembly code:
mfc0 t0,c0_mvpcontrol
li t1 2             // load the value for the combined  VPC and EVP fields
ins t0, t1, 0, 2    // insert VPC and EVP fields into MVPControl register value
mtc0 t0,c0_mvpcontrol
ehb
```

14.7.2 Set Target TC

Assuming the thread on which to execute is thread 0, configure the target TC using the TargTC field in the VPEControl register so thread 1 can be configured.

Table 115: VPEControl Register

Fields		VPEControl - CP0 #1-Sel1	Read/Write	Reset State
Name	Bits			
TargTC	7-0	Target TC number to be used on MTTR and MFTR instructions.	R/W	0

```
mips32_mt_settarget(TC1);

Assembly code:
mfc0 t0,c0_vpecontrol     // read the VPEControl Register
li t1, 1                  // load target TC number
ins t0,t1,0, 8            // insert TC number into VPEControl register value
mtc0 t0,c0_vpecontrol     // write new value to VPEControl register
ehb                       // ensure write has completed before continuing
```

The MTTR (move to thread register instruction) and the MFTR (move from thread register instruction) are directed to thread 1.

14.7.3 Halt Target TC

Before continuing, the target thread needs to be halted. Otherwise, the changes being made will be unpredictable. To ensure thread 1 is halted before configuring it, set the H field in the thread’s TCHalt register.

Table 116: TCHalt Register

Fields		TCHalt - CP0 #2-Sel4	Read/Write	Reset State
Name	Bits			
H	0	Thread halted. If set thread is halted and cannot be allocated, activated, or scheduled.	R/W	1

```
mips32_mt_settchalt(TCHALT_H);
```

```
Assembly code:
li t0,1                 // load the H field
mttc0 t0,c0_tchalt      // write the value to the TCHalt register
ehb                     // ensure write has completed before continuing
```

14.7.4 Bind TC to VPE

The TC needs to be bound to a VPE. The TCBind register controls the affiliation of TC to VPE.

Table 117: TCBind Register

Fields		TCBind - CP0 #2-Sel2	Read/Write	Reset State
Name	Bits			
CurVPE	3 - 0	ID number of the VPE to which the TC is bound.	R/W	0

```
mips32_mt_settcbind (VPE0);

Assembly code:
mttc0 zero,c0_tcbind       // write the value to the TCBind register
ehb                        // ensure write has completed before continuing
```

14.7.5 Setting the Target TC's Stack and Global Pointers

Each thread must have its own stack pointer. Threads can share the global pointer so the GP register just needs to be copied from the executing thread.

```
unsigned int TC1_stack[4096] __attribute__((aligned(16)));
unsigned int TC1_stack_top = (unsigned int)TC1_stack + 4080;

// set stack pointer
mips32_mt_setsp(TC1_stack_top);

Assembly code:
// NOTE: replace TC1_stack_top with th address of your stack top for target TC
li   t0, TC1_stack_top       // defined address TC1_stack_top
mttgpr t0,sp                 // write target TC stack pointer

// Set global pointer
mips32_mt_setgp(&_gp);

Assembly code:
mttgpr gp,gp                 // move gp from gp of current thread
```

14.7.6 Setting Starting Function Address

Once the target TC is unhalted and multi-threading is enabled, the CPU fetches the next instruction for the target TC from the target TC's TCRestart Register. The TCRestart Register needs to be loaded with the address of its starting function. In this example startTC1 is the starting function.

Table 118: TCRestart Register

Fields		TCRestart - CP0 #2-Sel3	Read/Write	Reset State
Name	Bits			
Restart address	31 - 0	Address at which execution is started for the TC.	R/W	0

```
mips32_mt_settcrestart(startTC1);

Assembly code
// NOTE: replace _startTC1 with the address of your function
li   t0, _startTC1          // load starting address using function lable
mttc0 t0,c0_tcrestart       // write address to TCRestart register
ehb                         // ensure write has completed before continuing
```

14.7.7 Activate the TC and Make It Dynamically Allocatable

Setting the A field in the TCStatus register activates the thread o the processor knows to fetch instructions for it. The thread also needs to be dynamically allocatable to make it available for use with FORK and YIELD instructions using the TCStatus register. To active the thread, set the A field (activated) in the TCStatus register.

Note: Even though this exaqmple does not use the FORK instruction, the code might use the Yield instruction later, therefore, dynamic thread allocation must be enabled by setting the DA field.

Table 119: TCStatus Register

Fields		TCStatus - CP0 #2-Sel1	Read/Write	Reset State
Name	Bits			
A	13	Activated. If set run instructions for this TC. Also set by FORK and cleared by YIELD $0.	R/W	1
DA	15	Dynamic allocation: If set, the TC can be allocated by FORK or de-allocated by YIELD.	R/W	0

```
mips32_mt_settcstatus(mips32_mt_gettcstatus() | (TCSTATUS_A | TCSTATUS_DA) );

Assembly code:
mftc0 v0,c0_tcstatus      // read the TCStatus register
ori  v0,v0,0xa000         // or in the A and AD bits
mttc0 v0,c0_tcstatus      // write the TCStatus register
ehb                       // ensure write has completed before continuing
```

14.7.8 Unhalt the TC

The last step in configuring the TC is to un-halt it so the TC can be scheduled and instructions can be fetched once multi-threading is enabled. Clearing the H bit in the TCHalt register un-halts the thread. Refer to *Table 116: TCHalt Register* on page 193 for a detailed description of the TCHalt H bit.

```
mips32_mt_settchalt(0);

Assembly code:
mttc0 zero,c0_tchalt    // move the value in the zero register to TCHalt
```

14.7.9 Enable Threading

Enable threading on the VPE by setting the TE bit in the VPEControl register.

Table 120: VPEControl Register

Fields		VPEControl - CP0 #1-Sel1	Read/Write	Reset State
Name	Bits			
TE	15	Thread enable. If cleared only one TC may execute on the VPE.	R/W	0

```
mips32_mt_setvpecontrol(mips32_mt_getvpecontrol() | VPECONTROL_TE);

Assembly code:
mftc0 v0,c0_vpecontrol      // read in the VPEControl register
ori   v0,v0,0x8000          // set the TE bit
mttc0 v0,c0_vpecontrol      // write the VPEControl register
ehb                         // ensure write has completed before continuing
```

14.7.10 Turn Off Configuration Mode and Enable Virtual Processing

The last step before the newly created thread starts execution is to leave configuration mode and enable virtual processing using the MVPControl register (the reverse procedure that started the process). Refer to *Table 114: MVPControl Register* on page 193 for detailed information on the fields.

```
mips32_setmvpcontrol((mips32_getmvpcontrol() & ~MVPCONTROL_VPC) | MVPCONTROL_EVP);

Assembly code:
mfc0 t0,c0_mvpcontrol    // read MVPControl
li   t1 1                // load the value for the combined  VPC and EVP fields
ins  t0, t1, 0, 2        // insert VPC and EVP fields into MVPControl register value
mtc0 t0,c0_mvpcontrol    // write new value to the MVPControl register
ehb                      // ensure write has completed before continuing
```

15 Instruction Delay Cycles

This chapter provides instruction delay cycles for producer and consumer instruction types.

The delay cycle is the minimum time between the time when an instruction issues and the time that a subsequent dependent instruction may issue. For example, and `ADD` instruction has a latency of 1 cycle. Consider the following code sequence:

```
ADD r3, r1, r2
ADD r5, r4, r3
```

In this example the second `ADD` instruction is dependent on the value placed into r3 by the first `ADD` instruction. It may issue one cycle after the first `ADD` instruction issues.

15.1 Instruction Types

15.1.1 Producers

- **Loads:** LB[U12], LB[16], LB[GP], LB[S9], LBU[U12], LBU[16], LBU[GP], LBU[S9], LBUX, LBX, LH[U12], LH[16], LH[GP], LH[S9], LHU[U12], LHU[16], LHU[GP], LHU[S9], LHUX, LHUXS, LHX, LHXS, LL, LLE, LW[U12], LW[16], LW[4X4], LW[GP16], LW[GP], LW[S9], LW[SP], LWM, LWPC, LWX, LWXS[32], LWXS[16], UALH, UALWM, RESTORE[32], RESTORE.JRC[16], RESTORE.JRC[32]
- **Arithmetic:** ADD, ADDIU[32], ADDIU[48], ADDIU[GP48], ADDIU[GP.B], ADDIU[GP.W], ADDIU[R1.SP], ADDIU[R2], ADDIU[RS5], ADDIU[NEG], ADDIUPC[32], ADDIUPC[48], ADDU[32], ADDU[16], ADDU[4X4], ALU20IPC[GP], AND[32], AND[16], ANDI[32], ANDI[16], AU20IPC, CLO, CLZ, EXT, EXTW, INS, LI[16], LI[48], LSA, LU20I, MOVE, MOVE.BALC, MOVEP, MOVEP[REV], MOVN, MOVZ, NOP[32], NOP[16], NOR, NOT[16], OR[32], OR[16], ORI, RESTORE[32], RESTORE.JRC[16], RESTORE.JRC[32], ROTR, ROTRV, SAVE[16], SAVE[32], SEB, SEH, SEQI, SLL[32], SLL[16], SLLV, SLT, SLTI, SLTIU, SLTU, SOV, SRA, SRAV, SRL[32], SRL[16], SRLV, SUB, SUBU[32], SUBU[16], WSBH, XOR[32], XOR[16], XORI
- **Links:** BALC[32], BALC[16], BALRSC, JALRC[32], JALRC[16], JALRC.HB, MOVE.BALC
- **MDU:** DIV, DIVU, MOD, MODU, MHU, MUHU, MUL[32], MUL[4X4], MULU
- **COP:** DI, EI, MFC0
- **Special:** RDPGPR, WRPGPR
- **SC:** SC, SCE

15.1.2 Consumers

- **L/S Base:** CACHE, PREF, PREFX, LB[U12], LB[16], LB[GP], LB[S9], LBU[U12], LBU[16], LBU[GP], LBU[S9], LBUX, LBX, LH[U12], LH[16], LH[GP], LH[S9], LHU[U12], LHU[16], LHU[GP], LHU[S9], LHUX, LHUXS, LHX, LHXS, LL, LLE, LW[U12], LW[16], LW[4X4], LW[GP16], LW[GP], LW[S9], LW[SP], LWM, LWPC, LWX, LWXS[32], LWXS[16], UALH, UALWM, RESTORE[32], RESTORE.JRC[16], RESTORE.JRC[32], SAVE[16], SAVE[32], SB[16], SB[GP], SB[S9], SBX, SC, SCE, SH[U12], SH[16], SH[GP], SH[S9], SHX, SHXS, SW[U12], SW[16], SW[4X4], SW[GP], SW[GP16], SW[S9], SW[SP], SWM, SWPC[48], SWX, SWXS, UASH, UASWM
- **St. Data:** SAVE[16], SAVE[32], SB[16], SB[GP], SB[S9], SBX, SC, SH[U12], SH[16], SH[GP], SH[S9], SHX, SHXS, SW[U12], SW[16], SW[4X4], SW[GP], SW[GP16], SW[S9], SW[SP], SWM, SWPC[48], SWX, SWXS, UASH, UASWM
- **Arithmetic:** ADD, ADDIU[32], ADDIU[48], ADDIU[GP48], ADDIU[GP.B], ADDIU[GP.W], ADDIU[R1.SP], ADDIU[R2], ADDIU[RS5], ADDIU[NEG], ADDIUPC[32], ADDIUPC[48], ADDU[32], ADDU[16], ADDU[4X4], ALU20IPC[GP], AND[32], AND[16], ANDI[32], ANDI[16], AU20IPC, CLO, CLZ, EXT, EXTW, INS, LI[16], LI[48], LSA, LU20I, MOVE, MOVE.BALC, MOVEP, MOVEP[REV], MOVN, MOVZ,

NOP[32], NOP[16], NOR, NOT[16], OR[32], OR[16], ORI, RESTORE[32], RESTORE.JRC[16], RESTORE.JRC[32], ROTR, ROTRV, SAVE[16], SAVE[32], SEB, SEH, SEQI, SLL[32], SLL[16], SLLV, SLT, SLTI, SLTIU, SLTU, SOV, SRA, SRAV, SRL[32], SRL[16], SRLV, SUB, SUBU[32], SUBU[16], WSBH, XOR[32], XOR[16], XORI

- MDU: DIV, DIVU, MOD, MODU, MHU, MUHU, MUL[32], MUL[4X4], MULU
- **Special:** RDPGPR, WRPGPR
- **COP0:** MTC0

15.1.3 Transaction Delay Cycles

The following table shows transaction delay cycles for nanoMIPS on I7200. The instructions have been grouped by category.

Table 121: Transaction Delay Cycles

	Consumer								
		L/S Base	St Data	Arith	MDU	Branch	Jump	Special	COP0	COP1
Producer	Loads	3	2	2	2	2	2	2	2	
	Arith	2	1	1	1	1	1	1	1	
	Links	2	1	1	1	1	1	1	1	
	MDU	6	5	5	5	5	5	4	5	
	COP	3	2	2	2	2	2	2	2	
	Special	2	1	1	1	1	1	1	1	
	SC	3	2	2	2	2	2	2	2	

15.2 MTC0 Instruction Considerations

Any MTC0 instruction that can potentially change the operating mode (kernel, supervisor, user) or context (memory mapping) should be executed before a JALRC.HB instruction to avoid hazards. Instructions following MTC0 - JALRC.HB pair will thus be fetched and executed in the new mode.

Execution of the MTC0 instruction can change the following register bits:

- *Status.ERL:* Changes the mapping of KUSeg memory segment. If the program is being executed in the KUSeg segment, and the MTC0 instruction that modifies the value of the ERL bit is not placed in the delay slot of a JALR.HB instruction, the instructions following the MTC0 instruction may be fetched from a different memory region.
- *Status.ERL, Status.EXL, Status.KSU:* Changes the mode of operation. If the MTC0 instruction that modified the mode is not placed in the delay slot of JALR.HB instruction, the instructions following the MTC0 instruction may be fetched in kernel mode but executed in the new mode.
- *Status.KX, Status.SX, Status.UX:* These bits determines the access privilege to 64-bit memory segments. If the program is being executed in a 64-bit segment and the MTC0 instruction that modified the value of these bits is not placed in the delay slot of JALR.HB instruction, the instructions following the MTC0 instruction may be fetched incorrectly.

Other operating mode changes include writes to the TLB (such as through TLBWR instruction) and writes to the MPU control registers. These operations also require a JALRC.HB instruction hazard barrier to avoid hazards.

16 MIPS On-Chip Instrumentation

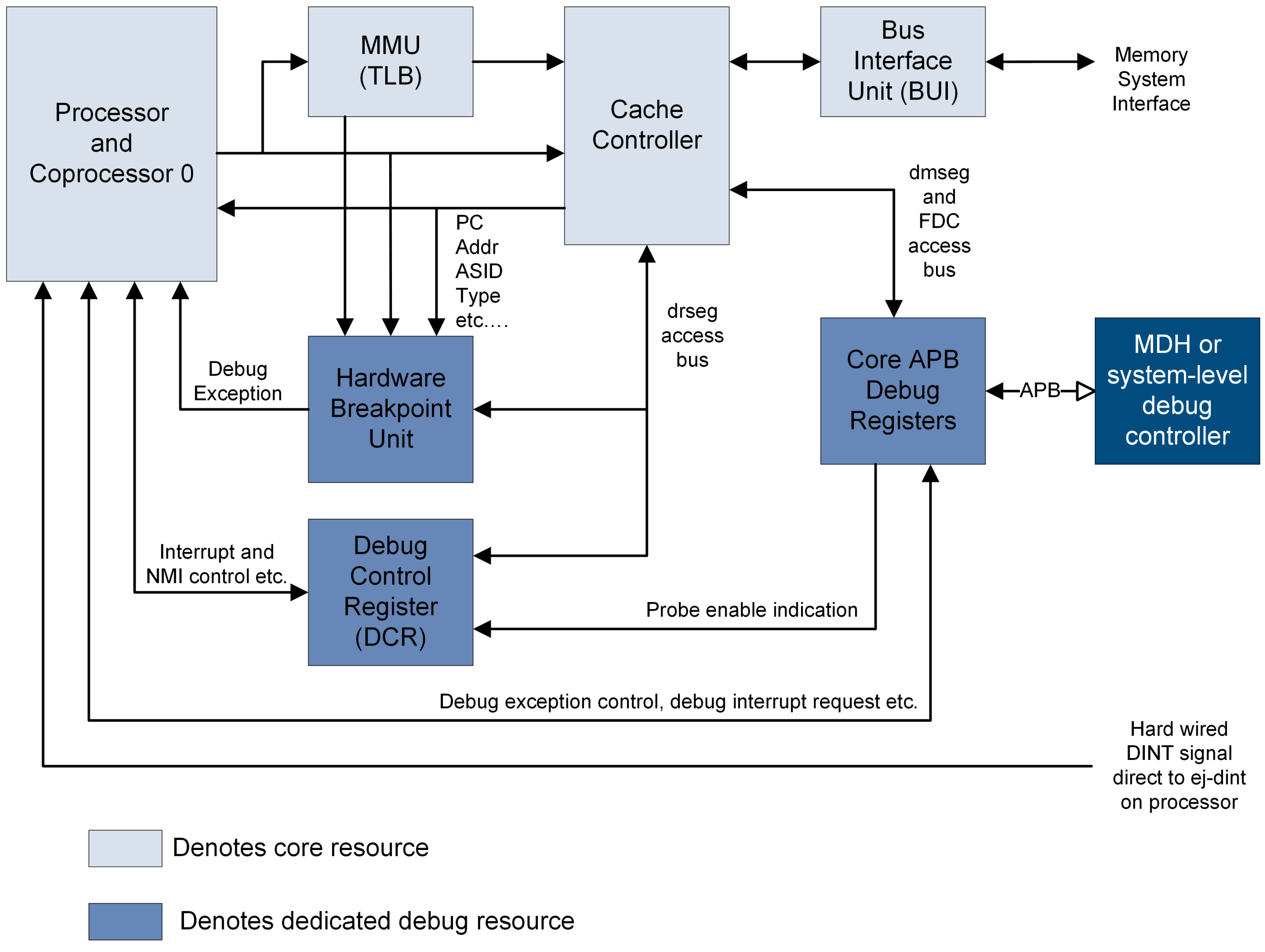
This chapter provides a brief overview of the interface and external debugging environment required to debug MIPS processors that incorporate the MIPS On-Chip Instrumentation (OCI) debug system for multi-core designs. Please refer to the community page link (*https://www.mips.com/develop/tools/*) for information on MIPS probes, Codescape debug tools, SDKs, and documentation.

The I7200 core implements the MIPS OCI 32-Bit Debug System (OCI32). MIPS developed OCI32 to provide comprehensive debugging and performance-monitoring capabilities for single and multicore processor designs that can have multiple Virtual Processing Elements (VPEs) per core and multiple thread contexts (TCs) per VPE.

16.1 I7200 OCI Debug System Overview

The OCI32 debug system consists of several distinct components: debug extensions to the MIPS processor core, the APB access port, the Debug Control Register, and the hardware breakpoint unit. The following figure shows the relationship between these components in a single-core implementation.

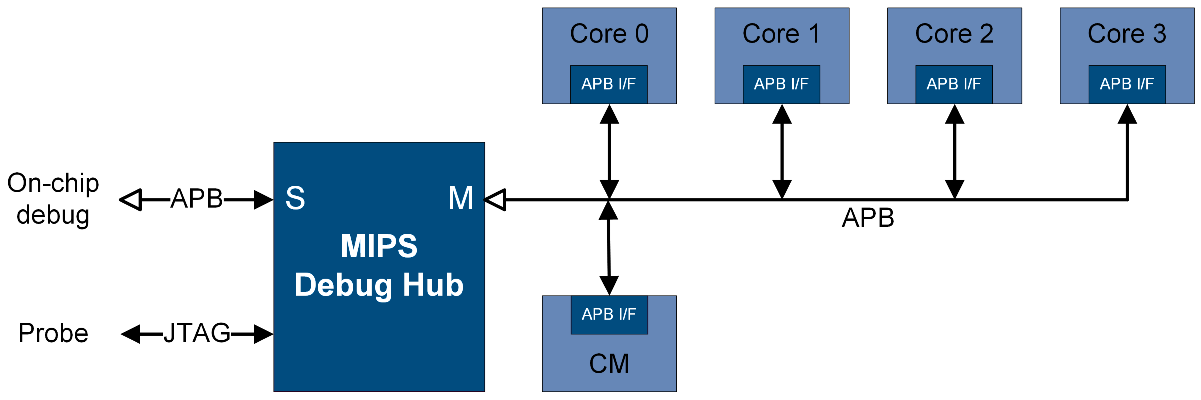
Figure 38: OCI 32-Bit System Block Diagram


Processor and Coprocessor 0
MMU (TLB)
Cache Controller
Bus Interface Unit (BUI)
Memory System Interface
PC Addr ASID Type etc.…
dmseg and FDC access bus
drseg access bus
Debug Exception
Hardware Breakpoint Unit
Core APB Debug Registers
APB
MDH or system-level debug controller
Interrupt and NMI control etc.
Debug Control Register (DCR)
Probe enable indication
Debug exception control, debug interrupt request etc.
Hard wired DINT signal direct to ej-dint on processor
Denotes core resource
Denotes dedicated debug resource

APB - MIPS Debug Hub Interface

An I7200 implementation can incorporate up to four OCI32 compliant cores and a Coherence Manager (CM2.6). Each core and the Coherence Manager incorporate a standard AMBA APB slave port to provide access from an APB-enabled system-level debug controller in a mixed-processor SoC or via a MIPS Debug Hub (MDH). The MDH routes debug control and data signals from a JTAG probe or on-chip APB master device as shown in the following figure.

Figure 39: I7200 Multi-Core Configuration with MDH



APB Debug Port

The APB port is used to access the core APB debug registers. These registers allow a probe or an APB-enabled debug controller to communicate with the rest of the debug module and the core. In an implementation with multiple VPEs, a single APB port provides access to (and within) each core. Each VPE has its own address space and set of APB debug resisters.

The core APB debug registers are used for determining core status, status and control of debug operations, changing debug settings, causing instructions to be executed, and providing other instrumentation functions such as performance analysis.

16.2 APB Map and Address Regions

Table 122: APB Map

A[15:0]	Device	Description
0x0000-0x0FFF	CM2.6	Coherence Manager
0x1000-0x1FFF	Core 0	Core 0
0x2000-0x2FFF	Core 1	Core 1
0x3000-0x3FFF	Core 2	Core 2
0x4000-0x4FFF	Core 3	Core 3

Within each core, the 4K APB address space is divided into ranges for each VPE. A core can have 1, 2, or 3 VPEs. Debug tools read the IDCODE register in each VPE range to determine whether that VPE is present. IDCODE Bit 0 is 1 if the VPE is present. If a VPE is not present, its entire 256-byte range is considered reserved and all registers in the range are 0.

Table 123: APB Address Regions

A[11:0]	Region	Description
0x000-0x0FF	VPE0	Debug registers for VPE0
0x100-0x1FF	VPE1	Debug registers for VPE1
0x200-0x2FF	VPE2	Debug registers for VPE2
0x300-0xFC7	Reserved	Read as 0, writes ignored
0xFC8-0xFFF	ID	ID register block

16.3 Power Management

In the I7200 OCI system, cores do not need to be powered up at all times. Instead, the probe directly accesses CPC registers through the CM APB debug port to monitor and control the power state of each core and the CM.

Note: The MIPS Debug Hub (MDH) should be always be powered on and clocked.

Table 124: APB Behavior for Core States

Core State	Definition	OCI_Control	APB Behavior
Unpowered	SI_TDOIsolate=1	–	SLVERR returned for all reads and writes

Powered, not clocked	SI_TDOIsolate=0,
	SI_LPReq=1
		Sleep=1	• IDCODE, IMPCODE, OCI_CONTROL, DBG_OUT, ID regs accessible
			• SLVERR returned for other reads and writes

Powered, clocked	SI_TDOIsolate=0,
	SI_LPReq=0
		Sleep=0	All registers accessible

When the core is powered but not clocked (state D2 or U2), IDCODE, IMPCODE, OCI_CONTROL, DBG_OUT, and ID register block are accessible. Accessing other registers returns SLVERR.

When the core is powered and clocked (state U5 or U6), all APB registers are accessible.

In transition states U3 and U4, the CPC asserts the core's reset input, which is indicated by OCI_CONTROL.Rocc. When the state reaches U5, reset is deasserted and the probe can clear OCI_CONTROL.Rocc. SI<*core*>_LPReq=0 identifies the clocked and powered state. OCI logic contributes to each core's SI<*core*>_LPAck output by holding off SI<*core*>_LPAck until in-progress APB requests complete.

16.4 CM2.6 Address Regions

Registers that identify the device type and configuration, and indicate reset status within the CM, are addressable through the APB. CPC registers can be accessed directly from the probe via the APB. Within the CPC register blocks, registers are mapped to the same offsets as when accessing from a core.

Table 125: CM2.6 APB Address Regions

A[11:0]	Region	Description
0x000-0x0FF	Debug	CM debug registers

A[11:0]	Region	Description
0x100-0x1FF	CPC global	0x100: CPC_ACCESS_REG 0x108: CPC_SEQDEL_REG 0x110: CPC_RAIL_REG 0x118: CPC_RESETLEN_REG 0x120: CPC_REVISION_REG
0x200-0x2FF	CPC core 0	0x200: CPC_CL_CMD_REG 0x208: CPC_CL_STAT_CONF_REG 0x210: CPC_CL_OTHER_REG
0x300-0x3FF	CPC core 1	
0x400-0x4FF	CPC core 2	
0x500-0x5FF	CPC core 3	
0xFC8-0xFFF	ID	ID register block
Others	Reserved	Read as 0, writes ignored

16.5 CM2.6 Debug Registers

Table 126: CM2.6 APB Debug Registers

A[7:0]	Name	Decription	Function	R/W
0x00	GCR_CONFIG	CM2.6 configuration	Copy of the GCR_CONFIG register	R
0x04	IDCODE	Device ID	Identifies the device manufacturer, part number, and revision	R
0x08	IMPCODE	Implementation code	Identifies the main debug features implemented.	R
0x14	OCI_CONTROL	OCI control register	Provides access to CM debug status and control.	R/W
Others	Reserved	Reserved	Reserved for expansion. Read as 0, writes ignores	–

The following tables define the IDCODE bits.

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Name	Bits	Description	R/W	Reset State
Version	31:28	Identifies the CM version number	R	Externally driven
PartNumber	27:12	Identifies the CM part number	R	Externally driven
ManufID	11:1	JEDEC ID	R	Externally driven
R	0	Reserved	R	1

The following tables define the IMPCODE bits.

31 29	28 14	13 11	10 1	0
DbgVer	0	Type	TypeInfo	0

Name	Bits	Description	R/W	Reset State
DbgVer	31:29	Identifies the OCI version number	R	0
Type	13:11	Type of entity, 2=CM	R	2
TypeInfo	10:1	Unused with Type=CM	R	0

The following tables define the OCI_CONTROL bits.

31	30 23	22	21	20 2	1	0
Rocc	0	Doze	Halt	0	DPD	0

Name	Bits	Description	R/W	Reset State
Rocc	31	CM Reset Occurred since last cleared. Rocc keeps the value 1 as long as reset is applied. The probe clears Rocc by writing 0.	R/W0	1
Doze	22	Unused.	R	0
Halt	21	Unused.	R	0
DPD	1	DisableProbeDebug	R	Externally set

The following table defines the CM2.6 APB Read-Only ID Registers.

Table 127: CM2.6 APB Read-Only ID Registers

A[11:0]	Register Name	Description	Function	Read Value
0xFC8	DEVID	IMG block type	Identifies IMG block type (7=CM)	0x00000007
0xFCC	DEVTYPE	Device type	MAJOR=5 (debug component), SUBTYPE=0 (other)	0x00000005
0xFD0	PIDR4	Peripheral ID part 4	Size=0 (4K), DES_2=4'b0010 (JEDEC)	0x00000002
0xFE0	PIDR0	Peripheral ID part 0	0 (other)	0x00000000
0xFE4	PIDR1	Peripheral ID part 1	DES_0=4'b0111 (JEDEC), PART_1=0	0x00000070
0xFE8	PIDR2	Peripheral ID part 2	Rev=GCR_REV[11:8], DES_1=4'b1010 (JEDEC)	0x000000XA
0xFEC	PIDR3	Peripheral ID part 3	Rev=GCR_REV[3:0], CMOD=0	0x000000X0
0xFF0	CIDR0	Component ID part 0	Fixed value for CoreSight	0x0000000D
0xFF4	CIDR1	Component ID part 1	Fixed value for CoreSight	0x000000E0
0xFF8	CIDR2	Component ID part 2	Fixed value for CoreSight	0x00000005
0xFFC	CIDR3	Component ID part 3	Fixed value for CoreSight	0x000000B1
Others	Reserved	Reserved	Read as 0	0x00000000

16.6 I7200 Core MPS Implementation

This section describes the debug configuration register values and other implementation details for the I7200 Core MPS.

Refer to the MIPS On-Chip Instrumentation 32-Bit Debug Specification, which describes the operation of the OCI32 debug system, for more information.

Debug Version	DBGver in CP0.Debug register or APB.IMCODE: 0 (OCI debug version1)

APB Data Registers	• ID Code (set by customer) • Implementation Code 0x01414800
Hardware Breakpoint Registers	• IBS: 0x48008000 • IBCn: 0x00000040 • DBS: 0x44008000 • DBCn: 0x4000000
Breakpoint Trigger Parameters	• Virtualization (VZ) not implemented • Data breakpoint data address sampling not implemented • Data breakpoint value match implemented • Data breakpoint address range matching not implemented
OCI Control Register	• Bits 13:11 ISAOnDebug: 11 for nanoMIPS • 31:24 = Tx_Count and 23:16 = Rx_Count. Present in I7200
PC/PCsampling	• No IM bit in the PC for nanoMIPS (no ISA mode switching) • PC sampling uses two registers (PCSAMP1 and PCSAMP2)

16.7 More Information on Debug Systems

For more information, refer to the following documents:

- MIPS Debug Hub Technical Reference Manual
- MIPS On-Chip Instrumentation 32-Bit Debug Specification

A Revision History

Revision	Date	Description
01.20	April 30, 2018	• First version released for I7200 Release 1.2.0. • Added information on L2 cache flush and burst operations. • Added DSPRAM Prediction Buffer section. • For EVA, changed references to pins to instead refer to the bit settings. • Minor edits throughout.
01.10	March 31, 2018	• First version released for I7200 Release 1.1.0. • Added Instruction Delay Cycles chapter. • Described limitations for the QoS Policy Manager mode. • Added hazard barrier instruction information. • Updated the section on Programming the Boot Exception Vector (BEV). • Removed the Global Debug Block section • Added a section on CM Performance Counters. • Minor edits throughout.
01.00	January 29, 2018	Initial release.