MIPS64® P6600 Multiprocessing System Software User's Guide

Document Number: MD01138
Revision 01.23
May 31, 2017

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

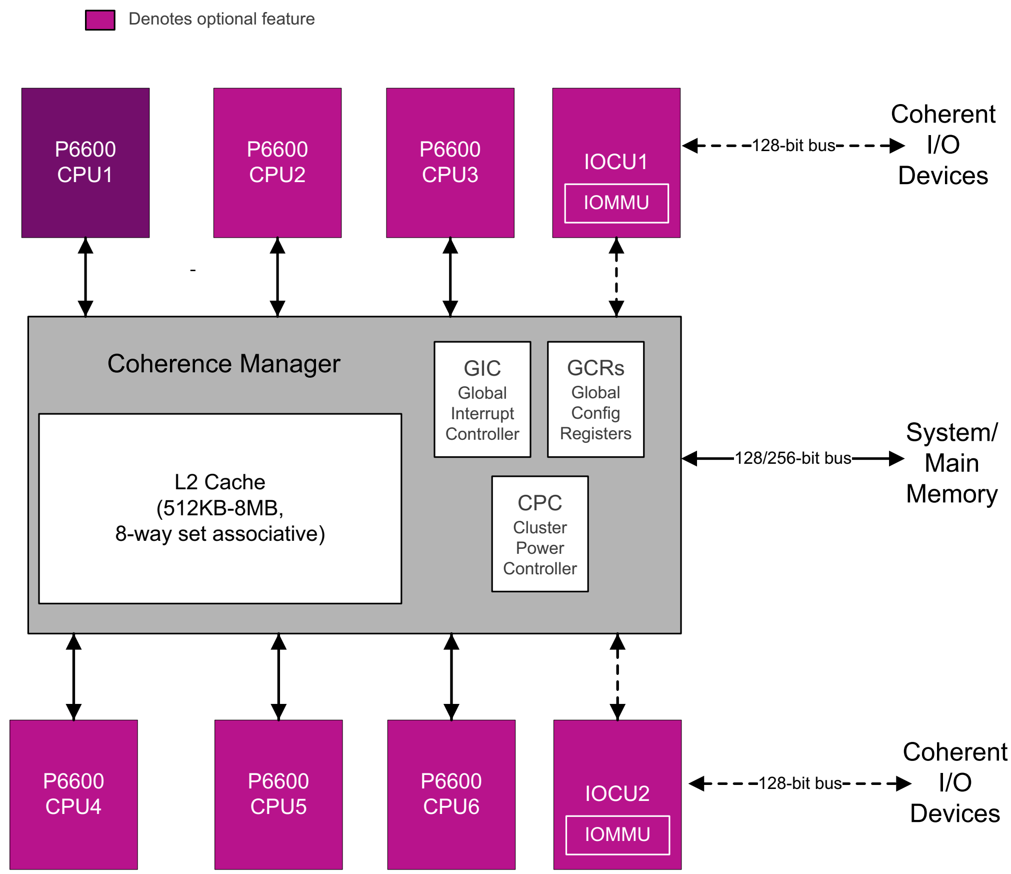
Chapter 1

Overview of the P6600 Architecture

The P6600™ series of high performance multi-core microprocessor cores provides best in class power efficiency for use in system-on-chip (SoC) applications. The P6600 Multiprocessing System (MPS) combines a deep pipeline with multi-issue out-of order-execution to deliver outstanding computational throughput. The P6600 provides full virtualization support. The P6600 Multiprocessing System is fully configurable/synthesizable and contains up to six MIPS64® P6600 CPU cores, a system level Coherence Manager with integrated L2 cache, a coherent I/O port (IOCU), and optional floating point unit with SIMD functionality.

Figure 1.1 shows a block diagram of the P6600 Multiprocessing System (MPS). In the P6600 Multiprocessing System, the Coherence Manager (CM2) with the integrated L2 cache streamlines the dataflow. Multi-CPU coherence is handled in hardware by the Coherence Manager. The I/O Coherence Unit (IOCU) supports hardware I/O coherence by bridging a non-coherent OCP I/O interconnect to the Coherence Manager (CM2) and handling ordering requirements. The Global Interrupt Controller (GIC) handles the distribution of interrupts between and among the CPUs. Under software controlled power management, the Cluster Power Controller (CPC) can gate off the clocks and/or voltage supply to idle cores.

Figure 1.1 P6600 Multiprocessing System Block Diagram



1.1 P6600 Features

P6600 Multiprocessor System is feature rich with the most current MIPS64 architecture, new CPU and system level features designed for the performance and features required for tomorrow's mainstream connected consumer electronics including smart phones, tablets, connected TVs and set-top boxes.

1.1.1 MIPS Architecture

P6600 Multiprocessing System has three key architecture features that sets the core's foundation.

1.1.1.1 MIPS64™ Release 6 Architecture

MIPS64® architecture, an industry standard, is the foundation of the P6600 product offering.
MIPS64 architecture provides a solid high-performance foundation by incorporating powerful features, standardizing privileged mode instructions, supporting past ISAs, and provides a seamless upgrade path from the MIPS32 architecture. MIPS64 is based on a fixed-length, regularly encoded instruction set, and it uses a load/store data model. It is streamlined to support optimized execution of high-level languages. Arithmetic and logic operations use a three-operand format, allowing compilers to optimize complex expressions formulation. Availability of 32 general-purpose registers enables compilers to further optimize code generation by keeping frequently accessed data in registers.

MIPS64 provides backward compatibility, standardizing privileged mode, and memory management, and provides the information through the configuration registers. The MIPS64 architecture enables real-time operating systems and application code to be implemented once and reused.

1.1.1.2 MIPS® SIMD Architecture

SIMD (Single Instruction Multiple Data), important technology for modern CPU designs that improves performance by allowing efficient parallel processing of vector operations. A non-programmable hardware aids the CPU and GPU by handling heavy-duty multimedia codecs, the MIPS® SIMD Architecture (MSA) technology incorporates a software-programmable solution into the CPU to handle emerging codecs or a small number of functions not covered by dedicated hardware. This programmable solution allows for increased system flexibility. In addition, the MSA is designed to accelerate many compute-intensive applications by enabling generic compiler support.

1.1.1.3 MIPS® Virtualization

To address security, privacy and reliability concerns in a wide range of devices, MIPS has added hardware supported virtualization technology into P6600 core. The hardware virtualization support enables processors to be OmniShield-ready. OmniShield is security technology which ensures that applications that need to be secure are effectively and reliably isolated from each other, as well as protected from non-secure applications.

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

1.1.2 System Level Features

- Up to six coherent MIPS64 P6600 CPU cores
- Superscalar, variable-length, out -of-order data return
- Support for power management with multiple power domains
- Cluster Power Controller (CPC) to shut down idle CPU cores to save power
- Hardware I/O coherence unit (IOCU)
- Hardware Virtualization Module Support
- Cache-to-cache data transfers
- Speculative memory reads to reduce latency
- Integrated 8-way set associative L2 cache controller supporting 512 KB to 8 MB cache sizes
- Shared L2 cache controller supporting 512 KB to 8 MB cache sizes
- Separate clock ratios on memory and IOCU OCP ports
- Clock ratio of 1:1 between Core, CM2, and L2 cache
- SOC system interface supports OCP version 2.1 protocol with 32- or 40-bit address and 128-bit or 256-bit data paths
- EJTAG Debug port supporting multi-processor debugging
- MIPS PDtrace
- Full scan design achieves test coverage in excess of 99% with memory BIST for internal SRAM arrays

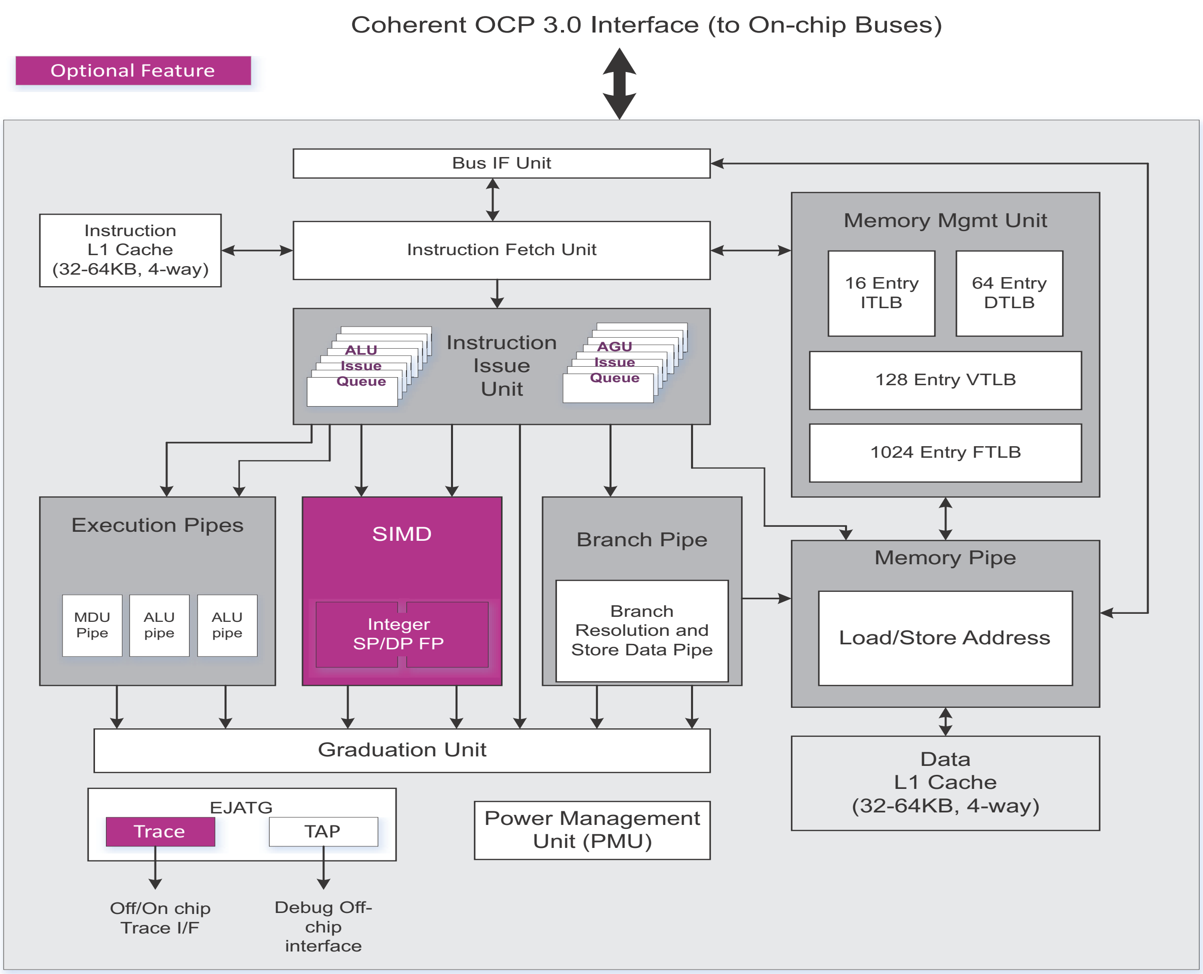
1.1.3 CPU Core Level Features

- 40-bit addressing
- Quad issue integer and dual issue 128-bit (integer/floating point) execution pipes
- Sophisticated branch prediction with fully associative Level 1 BTB
- Floating Point Unit with SIMD support and Out-Of-Order (OOO) execution
- Virtualization support
- Instruction Fetch Unit (IFU) with 4 instructions fetched per cycle
- Programmable Memory Management Unit with large first-level ITLB/DTLB backed by fast on-core second-level variable page size TLB (VTLB) and fixed page size TLB (FTLB):
- L1 Instruction and Data Caches can be configured as 32 or 64 KB per cache

1.2 P6600 CPU Core

Figure 1.2 shows a block diagram of a single P6600 core. The logic blocks in this diagram are described in the following sections.

Figure 1.2 P6600™ Core Block Diagram



For more information on the P6600 core in a multiprocessing environment, refer to Section 1.3 "Multiprocessing System".

1.2.1 Instruction Fetch Unit

The Instruction Fetch Unit (IFU) fetches instructions from the instruction cache and supplies them to the Instruction Issue Unit (IIU). The IFU can fetch up to four MIPS64 instructions at a time from the 4-way associative instruction cache. Instructions can also be fetched immediately from refill buffers in the event of an instruction cache miss.

The IFU employs sophisticated branch prediction and instruction supply strategies. The main predictor consists of three 2048-entry global branch history tables (BHT) that are indexed by different combinations of instruction PC and global history. A proprietary scheme is used to combine information from the three arrays to make a branch direction prediction.

The IFU also has a hardware-based return prediction stack to predict subroutine return addresses.The main predictor corrects target mispredicts from lower-level predictors without paying a full branch resolution penalty. The IFU supports fully out-of-order branch resolution.

The IFU has a 16-entry micro-Instruction TLB (ITLB) used to translate the virtual address into a physical address and used to compare against tags in the instruction cache to determine a hit. Refer to Section 1.2.6 "Memory Management Unit (MMU)" for more information.

A 24-entry instruction buffer decouples the instruction fetch from the execution. To maximize performance, some 'bonding' (or concatenation) of instructions is done at this stage while other types of instruction 'bonding' are performed downstream.

The IFU can also be configured to allow for hardware prefetching of cache lines on a miss. This mechanism provides excellent performance without incurring the area, power and latency costs of more overly complicated branch or instruction prefetch strategies.

The Global History register is internal to the IFU block and supports a novel history computation scheme that factors different information into the history for different kinds of control transfer instructions.

The P6600 level 1 (L1) instruction cache incorporates 'next fetch way' hit prediction logic. This allows the IFU to power on only those cache tag and data arrays that will provide the final instruction bytes and contributes to low power consumption.

1.2.2 Instruction Issue Unit (IIU)

The Instruction Issue Unit (IIU) is responsible for receiving instructions from the IFU and dispatching them to the out-of-order instruction scheduling windows and global instruction tracking window at a rate of 4 instructions per cycle.

The IIU tracks dynamic data flow dependencies between operations and issues them to the various pipes as efficiently as possible. Two schedulers service the various integer pipes.

The schedulers employ multiple dependency wake-up and pick schemes to enable age-based scheduling at high frequency. These two schedulers provide superior performance and power characteristics.

The IIU helps to 'bond' load and store operations whereby two 32-bit loads or 64-bit or stores to adjacent locations are 'bonded' or concatenated into one 64-bit or 128-bit memory access. This allows a factor of two improvement in certain memory intensive codes.

The IIU also keeps track of the progress of each instruction through the pipeline, updating the availability of operands in the 'rename map' and in all dependent instructions. Renamed instructions are steered to the most appropriate schedulers, taking opcode and other information into account.

The IIU also keeps track of global pipeline flushes, adjusting the rename map and other control structures to deal with interrupts, exceptions and other unexpected changes of control.

1.2.3 Graduation Unit (GRU)

The Graduation Unit (GRU) is responsible for committing execution results and releasing buffers and resources used by these instructions. The GRU is also responsible for evaluating the exception conditions reported by execution units and taking the appropriate exception. Asynchronous interrupts are funneled into the GRU, which prioritizes those events with existing conditions and takes the appropriate interrupt.

After processing the exception conditions, the GRU performs the following functions:

- Destination register(s) are updated and the completion buffers are released.
- Graduation information is sent to the IIU so it can update the rename maps to reflect the state of execution results (such as GPRs).
- Resolved branch information is sent to the IFU so that branch history tables can be updated and if needed, a pipeline redirect can be initiated. If sequential control flow is aborted for any reason, the GRU signals all core units to flush and recover microarchitectural state. After recovery is complete, it allows the IIU to resume dispatching instructions.

1.2.4 Level 1 Instruction Cache

The Level-1 (L1) instruction cache is configurable at 32 or 64 KB in size and is organized as 4-way set associative. Up to four instruction cache misses can be outstanding. The instruction cache is virtually indexed and physically tagged to make the data access independent of virtual to physical address translation.

Each instruction cache entry contains a tag portion, a data portion, and a way select portion.

An instruction tag entry holds 21 - 29 bits of physical address, a valid bit, a lock bit, and a parity bit. The data entry consists of 256 bits (8 MIPS64 instructions) of data and 32 bits of parity for a total of 288 bits. The way-select entry contains a 6 bit least-recently-used (LRU) field.

The P6600 core supports instruction-cache locking. Cache locking allows critical code segments to be locked into the cache on a “per-line” basis, enabling the system programmer to maximize the performance of the system cache.

The cache-locking function is always available on all instruction-cache entries. Entries can be marked as locked or unlocked on a per entry basis using the CACHE instruction.

The P6600 core implements virtual aliasing for the instruction cache, although this function can be disabled by the user.

1.2.5 Level 1 Data Cache

The Level 1 (L1) data cache is configurable at 32 or 64 KB in size. It is also organized as 4-way set-associative. Data cache misses are non-blocking and up to nine misses may be outstanding. The data cache is virtually indexed and physically tagged to make the data access independent of virtual-to-physical address translation. To achieve the highest possible frequencies using commercially available SRAM generators, cache access and hit determination are spread across three pipeline stages, dedicating an entire cycle for the SRAM access.

Each instruction cache entry contains a tag portion, a data portion, a way-select portion, and a dirty status portion.

- A data tag entry holds 21 bits of physical address in 32-bit addressing mode (29 bits in 40-bit addressing mode), a valid bit, a state bit, and a parity bit, making a total of 24 - 32 bits per tag entry.

- The data entry consists of 256 bits consisting of 32 bytes of data of data and 32 bits of parity for a total of 288 bits. The way-select entry contains a 6 bit least-recently-used (LRU) field, a 4-bit lock field, and a 4-bit lock parity field for a total of 14 bits.
- The Dirty state entry contains a 4-bit dirty field and a 4-bit dirty parity field.

The P6600 core supports a data-cache locking mechanism identical to that used in the instruction cache. Critical data segments are locked into the cache on a "per-line" basis. The locked contents can be updated on a store hit, but are not selected for replacement on a cache miss.

The P6600 core implements virtual aliasing for the data cache. This function is managed in hardware and is transparent to the user.

1.2.6 Memory Management Unit (MMU)

The P6600 core's Memory Management Unit (MMU) is primarily responsible for converting virtual addresses to physical addresses and providing attribute information for different segments of memory. The P6600 MMU contains the following Translation Lookaside Buffer (TLB) types:

- Instruction TLB (ITLB)
- Data TLB (DTLB)
- Variable Page Size Translation Lookaside Buffer (VTLB)
- Fixed Page Size Translation Lookaside Buffer (FTLB)

1.2.6.1 Instruction TLB (ITLB)

The ITLB is a 16-entry high speed TLB dedicated to performing translations for the instruction stream. The ITLB maps only 4 KB or 16 KB pages. Larger pages are split into smaller pages of one of these two sizes and installed in the ITLB.

The ITLB is managed by hardware and is transparent to software. The larger VTLB and FTLB structures are used as a backup structure for the ITLB. If a fetch address cannot be translated by the ITLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the ITLB for future use.

1.2.6.2 Data TLB (DTLB)

The DTLB is a 32-entry high speed TLB dedicated to performing translations for the data stream. The DTLB maps only 4 KB or 16 KB pages. Larger pages are split into one of these configured sizes and installed in the DTLB.

The DTLB is managed by hardware and is transparent to software. The larger VTLB and FTLB structures are used as a backup structure for the DTLB. If a fetch address cannot be translated by the DTLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the DTLB for future use.

1.2.6.3 Variable Page Size TLB (VTLB)

The VTLB is a fully associative variable translation lookaside buffer with 64 dual entries that can map variable size pages from 4KB to 256MB. When an instruction address is calculated, the virtual address is first compared to the contents of the ITLB and DTLB. If the address is not found in either the ITLB or DTLB, the VTLB/FTLB is accessed. If the entry is found in the VTLB, that entry is then written into the ITLB or DTLB. If the address is not found in the VTLB, a software TLB exception is taken. For data accesses, the virtual address is looked up in the VTLB only, and a miss causes a TLB exception.

1.2.6.4 Fixed Page Size TLB (FTLB)

The FTLB is 512 dual entries organized as 128 sets and 4 ways. Each set of each way contains dual data RAM entries and one tag RAM entry. If the tag RAM contents match the requested address, either the low or high RAM location of the dual data RAM is accessed depending on the state of the most-significant-bit (MSB) of the offset portion of the virtual address (VPN2). Each RAM location can only map a fixed page size, which is configurable to 4KB or 16KB.

1.2.6.5 Enhanced Virtual Address

The P6600 core supports a programmable memory segmentation scheme called Enhanced Virtual Address (EVA). EVA allows for more efficient use of 32-bit address space. Traditional MIPS virtual memory support divides up the virtual address space into fixed segments, each with fixed attributes and access privileges. Such a scheme limits the amount of physical memory available to 0.5GB, the size of kernel segment 0 (*kseg0)*.

1.2.6.6 Virtualization Support

Virtualization defines a set of extensions to the MIPS64 Architecture for efficient implementation of virtualized systems.

Virtualization is enabled by software. The key element is a control program known as a Virtual Machine Monitor (VMM) or hypervisor. The hypervisor is in full control of machine resources at all times.

The hypervisor is responsible for managing access to sensitive resources, maintaining the expected behavior for each VM, and sharing resources between multiple VMs.

In a traditional operating system, the kernel (or supervisor) typically runs at a higher level of privilege than user applications. The kernel provides a protected virtual-memory environment for each user application, inter-process communications, IO device sharing and transparent context switching. The hypervisor performs the same basic functions in a virtualized system, except that the hypervisor's clients are full operating systems rather than user applications.

The virtual machine execution environment created and managed by the hypervisor consists of the full Instruction Set Architecture (ISA), including all Privileged Resource Architecture (PRA) facilities, and any device-specific or board-specific peripherals and associated registers. It appears to each guest operating system as if it is running on a real machine with full and exclusive control.

The Virtualization Module enables full virtualization, and is intended to allow VM scheduling to take place while meeting real-time requirements, and to minimize costs of context switching between VMs.

1.2.7 Execution Pipelines

The P6600 core contains the following execution pipelines:

- Arithmetic Logic Pipeline
- Multiply-Divide Pipeline
- Memory Pipeline

- Branch Pipeline
- Two FPU3 Pipelines

Each of these execution units is described in the following subsections. Instructions intended for the arithmetic logic pipeline are driven by the out-of-order ALU Decode and Dispatch queue inside the Instruction Issue Unit (IIU) as shown in Figure 1.2. The other four pipelines are driven by the out-of-order Address Generation unit (AGU) Decode and Dispatch queue also located in the IIU.

1.2.7.1 Arithmetic Logic Pipeline

The arithmetic unit pipeline consists of one execution unit, called the ALU (Arithmetic Logic Unit), which performs integer instructions such as adds, shifts and bit-wise logical operations with a single cycle latency. If the IIU decodes a single-cycle instruction, it is usually sent to the ALU dispatch queue that feeds the arithmetic unit pipeline. This pipeline also contributes to performing 'bonded' loads. Refer to Section 1.2.2 "Instruction Issue Unit (IIU)"for a definition of instruction 'bonding'.

1.2.7.2 Multiply/Divide Pipeline

The multiply/divide pipeline executes integer multiplies, integer divides, and integer multiply-accumulate instructions. The multiply/divide pipeline incorporates a new very high-speed integer divider.

The MDU consists of a 64-bit multiplier, result/accumulation registers, a divide state machine, and all necessary multiplexers and control logic.

The MDU supports execution of one multiply or multiply-accumulate operation every clock cycle whereas divides can be executed as fast as one every four cycles.

1.2.7.3 Memory Pipeline

The memory pipeline primarily contains the LSU (Load Store Unit). The LSU is responsible for interfacing with the AGU dispatch queue (see Figure 1.2) and processing load/store instructions to read/write data from data caches and downstream memory.

It is capable of handling loads and stores issued out-of-order. The LSU has the ability to receive loads and stores in almost any order enables very high performance compared to an in-order machine. Such instruction-level parallelism allows maximum utilization of the memory pipe resources with minimal area and power.

The LSU can execute loads and stores at twice the rate of regular operations by concatenating data from two 32-bit or 64-bit memory to form a single 64-bit or 128-bit entity, respectively. This 'bonding' of instructions allows the LSU to provide almost all the benefits of dual memory access pipes without incurring the area and power costs of multiple tag, data and TLB structures.

The memory pipeline receives instructions from the Instruction Issue Unit (IIU) and interfaces to the L1 data cache. Loads are non-blocking in the P6600 core. Loads that miss in the data cache are allowed to proceed with their destination register marked unavailable. Consumers of this destination register are held back and replayed as needed after the cache miss has been serviced by the downstream memory subsystem, which includes the high performance L2 cache.

Graduated load misses and store hits and misses are sent in order to the Load/Store Graduation Buffer (LSGB). The LSGB has corresponding data and address buffers to hold all relevant attributes.

An 8-entry Fill Store Buffer (FSB) tracks outstanding fill or copy-back requests. It fills the data cache at the rate of 128-bits per cycle when an incoming line is completely received. Each FSB entry can hold an entire cache line.

The Load Data Queue (LDQ) keeps track of outstanding load misses and forwards the critical data to the main pipe as soon as it becomes available.

Hardware anti-aliasing allows using the core with operating systems that do not support software page coloring. The fully-associative DTLB operates a clock earlier in the LSU pipeline, making use of fast add-and-compare logic to enable virtual address to physical address translations that do not require the area and power expense of virtual tagging. All of this is done completely transparent to software.

1.2.7.4 Branch Pipeline

The Branch pipeline performs the following functions:

- Executes Branch and Jump instructions
- Performs Branch resolution
- Performs Jump resolution
- Sends the redirect to the Instruction Fetch Unit (IFU)
- Performs a write-back to the Link registers

1.2.7.5 Floating Point Pipelines

The optional Floating Point Unit with SIMD contains two execution pipelines. One pipeline executes SIMD logical operations (ops), SIMD integer adds. The FP compares and stores. The other pipeline executes SIMD integer multiplies, SIMD vector shuffles, FP adds, FP multiplies, and FP divides.

For more information, refer to Section 1.2.12 "Floating Point Unit".

1.2.8 Bus Interface (BIU)

The BIU controls a 128-bit interface to the CM2. The interface implements the Open Core Protocol (OCP).

1.2.8.1 Write Buffer

The BIU contains a merging write buffer. This buffer stores and combines write transactions before issuing them to the external interface. The write buffer is organized as eight, 32-byte buffers. Each buffer can contain data from a single 32-byte aligned block of memory.

When using the write-through cache policy or performing uncached accelerated writes, the write buffer significantly reduces the number of write transactions on the external interface and reduces the amount of stalling in the core caused by the issuance of multiple writes in a short period of time.

The write buffer also holds eviction data for write-back lines. The load-store unit extracts dirty data from the cache and sends it to the BIU. In the BIU, the dirty data is gathered in the write buffer and sent out as a bursted write.

For uncached accelerated writes, the write buffer can gather multiple writes together and then perform a bursted write in order to increase the efficiency of the bus.

Gathering of uncached accelerated stores can start on any arbitrary address and can be combined in any order within a cache line. Uncached accelerated stores that do not meet the conditions required to start gathering are treated like regular uncached stores.

1.2.9 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation and cache protocols, the exception control system, the processor's diagnostic capability, the operating modes, and whether interrupts are enabled or disabled. Configuration information, such as cache size and associativity, and the presence of features like a floating point unit, are also available by accessing the CP0 registers.

CP0 also contains the state used for identifying and managing exceptions. Exceptions can be caused by a variety of sources, including boundary cases in data, external events, or program errors.

1.2.10 Interrupt Handling

The P6600 core supports six hardware interrupts, two software interrupts, a timer interrupt, and a performance counter interrupt. These interrupts can be used in any of three interrupt modes, as defined in the MIPS64 Architecture:

- Interrupt compatibility mode.
- Vectored Interrupt (VI) mode. Adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt.
- External Interrupt Controller (EIC) mode. Provides support for an external interrupt controller that handles prioritization and vectoring of interrupts.

1.2.11 Modes of Operation

The P6600 core supports four modes of operation:

- Two user modes (guest and root), most often used for application programs.
- Two supervisor modes (guest and root) provides an intermediate privilege level with access to the *ksseg* (kernel supervisor segment) address space.
- Two kernel modes (guest and root), typically used for handling exceptions and operating system kernel functions, including CP0 management and I/O device accesses.
- Debug mode is used during system bring-up and software development. Refer to Section 1.2.14 "EJTAG Debug Support" for more information on debug mode.

1.2.12 Floating Point Unit

The P6600 core features an optional IEEE 754 compliant 3rd generation Floating Point Unit with SIMD.[1]

The FPU3 contains thirty-two, 128-bit vector registers shared between SIMD and MIPS64 instructions.

SIMD instructions enable:

1. Requires separate MIPS license.

- Efficient vector parallel arithmetic operations on integer, fixed-point and floating-point data.
- Operations on absolute value operands.
- Rounding and saturation options available.
- Full precision multiply and multiply-add.
- Conversions between integer, floating-point, and fixed-point data.
- Complete set of vector-level compare and branch instructions with no condition flag.
- Vector (1D) and array (2D) shuffle operations.
- Typed load and store instructions for endian-independent operation.

The FPU3 with SIMD is fully synthesizable and operates at the same clock speed as the CPU. The IIU can issue up to two instructions per cycle to the FPU3.

The FPU3 contains two execution pipelines for floating point and SIMD instruction execution. These pipelines operate in parallel with the integer core and do not stall when the integer pipeline stalls. This allows long-running FPU3/SIMD operations such as divide or square root, to be partially masked by system stall and/or other integer unit instructions.

An out-of-order scheduler in the FPU3 issues instructions to the two execution units. The exception model is 'precise' at all times.

1.2.13 P6600 Core Power Management

The P6600 core offers several power management features, that support low-power designs, such as active power management and power-down modes of operation. The P6600 core is a static design that supports slowing or halting the clocks to reduce system power consumption during idle periods.

You can also use the Cluster Power Controller (CPC) to control your power management. Refer to "Cluster Power Controller (CPC)" on page 37 for more details.

1.2.13.1 Instruction-Controlled Power Management

The Instruction Controlled power-down mode is invoked through execution of an instruction. When the WAIT instruction is executed, the internal clock is suspended; however, the internal timer and some of the input pins continue to run. When the CPU is in this instruction-controlled power management mode, any interrupt, NMI, or reset condition causes the CPU to exit this mode and resume normal operation.

The P6600 core asserts a sleep signal whenever it has entered low-power mode (sleep mode). The core enters sleep mode when all bus transactions are complete and there are no running instructions.

The WAIT instruction can put the processor in a mode where no instructions are running. When the WAIT instruction is seen by the Instruction Fetch Unit (IFU), subsequent instruction fetches are stopped. The WAIT instruction is dispatched down the pipe and graduated. Upon graduation of the WAIT, the GRU waits for the processor to reach a quiescent state and allows the processor to enter sleep mode.

1.2.14 EJTAG Debug Support

The P6600 core includes an Enhanced JTAG (EJTAG) block for use in software debugging of application and kernel code. For this purpose, in addition to standard user/supervisor/kernel modes of operation, the P6600 core provides a Debug mode.

Debug mode is entered when a debug exception occurs and continues until a debug exception return instruction is executed. During this time, the processor executes the debug exception handler routine.

The EJTAG interface operates through the Test Access Port (TAP), a serial communication port used for transferring test data in and out of the P6600 core. In addition to the standard JTAG instructions, special instructions defined in the EJTAG specification define which registers are selected and how they are used.

There are several types of simple hardware breakpoints defined in the EJTAG specification. These breakpoints stop the normal operation of the CPU and force the system into debug mode.

During synthesis, the P6600 core can be configured to support the following breakpoint options:

- Zero instruction, zero data breakpoints
- Four instruction, two data breakpoints

Instruction breaks occur on instruction fetch operations, and the break is set on the virtual address. Instruction breaks can also be made on the ASID value used by the MMU. A mask can be applied to the virtual address to set breakpoints on a range of instructions.

Data breakpoints occur on load and/or store transactions. Breakpoints are set on virtual address and address space identifier (ASID) values, similar to the Instruction breakpoint. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to the virtual address, ASID value, and the load/store value.

1.2.14.1 Fast Debug Channel

The P6600 CPU includes the EJTAG Fast Debug Channel (FDC) for efficient bi-directional data transfer between the CPU and the debug probe. Data is transferred serially via the TAP interface. A pair of memory- mapped FIFOs buffer the data, isolating software running on the CPU from the actual data transfer. Software can configure the FDC block to generate an interrupt based on the FIFO occupancy or can poll the status.

1.2.14.2 PDtrace

The P5600 core includes trace support for real-time tracing of instruction addresses, data addresses, data values, performance counters, and processor pipeline inefficiencies. The trace information is collected in an on-chip or off-chip memory, for post-capture processing by trace regeneration software. Software-only control of trace is possible in addition to probe-based control.

An on-chip trace memory may be configured in size from 256B to 8 MB; it is accessed either through load instructions or the existing EJTAG TAP interface, which requires no additional chip pins.

Off-chip trace is managed with the PIB2 (2nd-generation Probe Interface Block) hardware that ships with the product. It provides a selectable trace port width of 4, 8, or 16 pins plus DDR clock. Trace data is streamed on these pins and captured using the MIPS Navigator™ Pro probe Other supported probes include DA-net and Joyner.

1.3 Multiprocessing System

The Multiprocessing System (MPS) consists of the logic modules —CPC, CM2, IOCU, GIC, and GCR—shown in Figure 1.1. Each block is described throughout this section. In additional the clocking and debugging features are also described in this section

1.3.1 Cluster Power Controller (CPC)

Individual CPUs within the cluster can have their clock and/or power gated off when they are not in use. This gating is managed by the Cluster Power Controller (CPC). The CPC handles the power shutdown and ramp-up of all CPUs in the cluster. Any P6600 CPU that supports power-gating features is managed by the CPC.

The CPC also organizes power-cycling of the CM2 dependent on the individual core status and shutdown policy. Reset and root-level clock gating of individual CPUs are considered part of this sequencing.

1.3.1.1 Reset Control

The reset input of the system resets the Cluster Power Controller (CPC). Reset sideband signals are required to qualify a reset as system cold, or warm start. Pin settings determine the course of action for each core after a CPC reset.

- Remain in powered-down
- Go into clock-off mode
- Power-up and start execution

In case of a system cold start, after reset is released, the CPC powers up the P6600 CPUs as directed in the CPC cold start configuration pins. If at least one CPU has been chosen to be powered up on system cold start, the CM2 is also powered up.

When supply rail conditions of power gated CPUs have reached a nominal level, the CPC will enable clocks and schedule reset sequences for those CPUs and the coherence manager.

At a warm start reset, the CPC brings all power domains into their cold start configuration. However, to ensure power integrity for all domains, the CPC ensures that domain isolation is raised before power is gated off. Domains that were previously powered and are configured to power up at cold start remain powered and go through a reset sequence.

Within a warm start reset, sideband signals are also used to qualify if coherence manager status registers and GIC watch dog timers are to be reset or remain unchanged. The CPC, after power up of any CPU, provides a test logic reset sequence per domain to initialize TAP logic.

There are memory-mapped registers that can set the value for each CPU's *SI_ExceptionBase* pins. This allows different boot vectors to be specified for each of the cores so they can execute unique code if required. Each of the cores will have a unique CPU number, so it is also possible to use the same boot vector and branch based on that.

1.3.2 Coherence Manager 2 (CM2)

The Coherence Manager with integrated L2 cache (CM2) is responsible for establishing the global ordering of requests and for collecting the intervention responses and sending the correct data back to the requester. A high-level view of the request/response flow through the CM2 is shown in Figure 1.3. Each of the blocks is described in more detail in the following subsections.

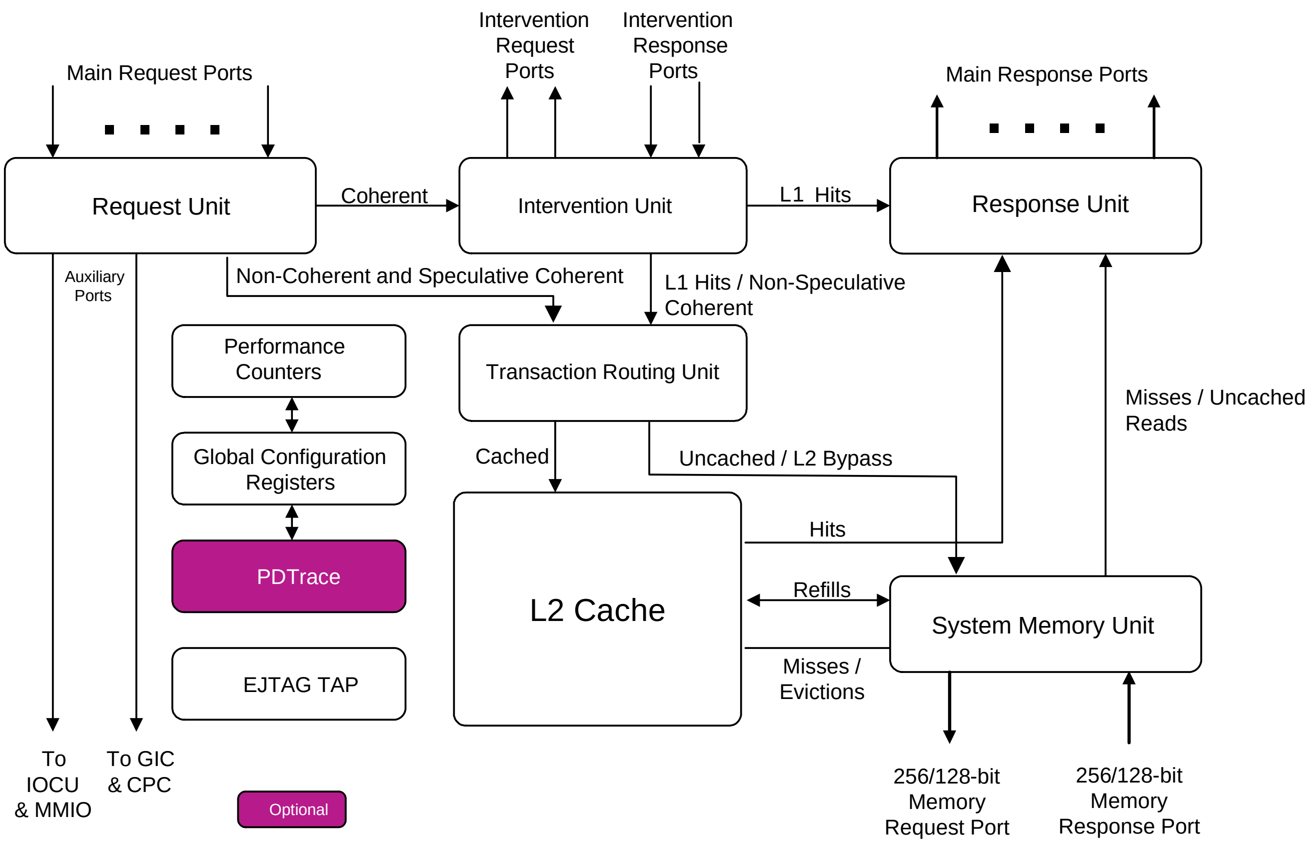
1.3.2.1 Request Unit (RQU)

The Request Unit (RQU) receives OCP bus transactions from multiple CPU cores and/or I/O ports, serializes the transactions and routes them to the Intervention Unit (IVU), Transaction Routing Unit (TRU), or an auxiliary port used to access a configuration registers or memory-mapped IO. The routing is based on the transaction type, the transaction address, and the CM2's programmable address map.

1.3.2.2 Intervention Unit (IVU)

The Intervention Unit (IVU) interrogates the L1 data caches by placing requests on the intervention OCP interfaces. Each processor responds with the state of the corresponding cache line. If the processor has the corresponding data in its L1 data cache, it provides the data with its response. If the original request was a read, the IVU routes the data to the original requestor via the Response Unit (RSU).

Figure 1.3 Coherence Manager 2 (CM2) with Integrated L2 Cache Block Diagram



The IVU gathers the responses from each of the agents and manages the following actions:

- Speculative reads are resolved (confirmed or cancelled).
- Memory reads that are required because they were not speculative are issued to the Transaction Routing Unit (TRU).
- Modified data returned from the CPU is sent to the TRU to be written back to the L2 cache or memory.
- Data returned from the CPU is forwarded to the Response Unit (RSU) to be returned to the requester.

- The MESI state in which the line is installed by the requesting CPU is determined (the "install state"). If there are no other CPUs with the data, a Shared request is upgraded to Exclusive.

Each device updates its cache state for the intervention and responds when the state transition has completed. The previous state of the line is indicated in the response. If a read type intervention hits on a line that the CPU has in a Modified or Exclusive state, the CPU returns the cache line with its response. A cache-less device, such as the IOCU, does not require an intervention port.

1.3.2.3 System Memory Unit (SMU)

The System Memory Unit (SMU) provides the interface to the memory OCP port. For an L2 refill, the SMU reads the data from an internal buffer and issues the refill request to the L2 pipeline.

1.3.2.4 Response Unit (RSU)

The RSU takes responses from the SMU, L2, IVU, or auxiliary port and places them on the appropriate OCP interface. Data from the L2 or SMU is buffered inside a buffer associated with each RSU port.

When a coherent read receives an intervention hit in the MODIFIED or EXCLUSIVE state, the Intervention Unit (IVU) provides the data to the RSU. The RSU then returns the data to the requesting core.

1.3.2.5 Transaction Routing Unit

The Transaction Routing Unit (TRU) arbitrates between requests from the RQU and IVU, and routes requests to either the L2 or the SMU. The TRU also contains the request and intervention data buffers which are written directly from the RQU and IVU, respectively. The TRU reads the appropriate write buffer when it processes the corresponding write request.

1.3.2.6 Level 2 Cache

The unified L2 cache holds both instruction and data references and contains a 7-stage pipeline to achieve high frequencies with low power while using commercially available SRAM generators.

Cache read misses are non-blocking; that is, the L2 can continue to process cache accesses while up to 15 misses are outstanding. The cache is physically indexed and physical tagged.

- *L2 Cache Configuration* provides the following L2 cache configuration options: 512KB, 1MB, 2MB, 4MB, and 8MB

- *L2 Pipeline Tasks* manages the flow of data to and from the L2 cache. The L2 pipeline performs the following tasks:

  - Accesses the tags and data RAMs located in the memory block (MEM).

  - Returns data to the RSU for cache hits.

  - Issues L2 miss requests.

  - Issues L2 write and eviction requests.

  - Returns L2 write data to the SMU. The SMU issues refill requests to the L2 for installation of data for L2 allocations

- *L2 Cache* Features are

- Supports write-back operation.
- Pseudo-LRU replacement algorithm
- Programmable wait state generator to accommodate a wide variety of SRAMs.
- L2 prefetcher. Hardware recognizes streams of sequential accesses and prefetches memory data into the L2 cache.
- Operates at same clock frequency as CPU.
- Cache line locking support
- ECC support for resilience to soft errors
- Single-bit error correction and 2-bit error detection support for Tag and Data arrays
- Single bit detection only for WS array
- Bypass mode
- Fully static design: minimum frequency is 0 MHz
- Sleep mode
- Memory BIST for internal SRAM arrays, with support for integrated (March C+, IFA-13) or custom BIST controller.

1.3.2.7 CM2 Configuration Registers

The Registers block (GCR) contains the control and status registers for the CM2. It also contains registers that control the Trace Funnel, EJTAG TAP state machine, and other multi-core features.

1.3.2.8 Performance Counter Unit

The CM2 implements a Performance Counter Unit (PERF) that contains the performance counters and associated logic.

1.3.2.9 Coherence Manager Performance

The CM2 has a number of high performance features:

- 256-bit wide internal data paths throughout the CM2
- 128-bit or 256-bit wide system OCP interface
- Integrated L2 cache provides low latency for L2 cache hits
- CM2 and L2 can process up to 1 request per cycle
- *Cache to Cache transfers*: If a read request hits in another L1 cache in the EXCLUSIVE or MODIFIED state, it will return the data to the CM2 and it will be forwarded to the requesting CPU, thus reducing latency on the miss.

- *Speculative Reads*: Coherent read requests are forwarded to the L2 cache before they are looked up in the other caches. This is speculating that the cache line will not be found in another CPU's L1 cache.

1.3.3 I/O Coherence Unit (IOCU)

Hardware I/O coherence is provided by the I/O Coherence Unit (IOCU), which maintains I/O coherence of the caches in all coherent CPUs in the cluster.

The IOCU acts as an interface block between the Coherence Manager (CM2) and I/O devices. Reads and writes from I/O devices may access the L1 and L2 caches by passing through the IOCU and the CM2. Each request from an I/O device may be marked as coherent, non-coherent cached, or uncached. Coherent requests access the L1 and L2 caches. Non-coherent cached requests access only the L2 cache. Uncached requests bypass both the L1 and L2 caches and are routed to main memory.

The IOCU also provides a legacy (without coherent extensions) OCP slave interface to the I/O interconnect for I/O devices to read and write system memory. The design also includes an OCP Master port to the I/O interconnect that allows the CPUs to access registers and memory on the I/O devices.

The IOCU design provides several features for easier integration:

- Supports incremental bursts up to 256 bytes (16 beats of 128b data) on I/O side. These requests are split into cache-line- sized requests on the CM side
- Read responses with different TagIDs may be returned out-of-order
- Integrated I/O Memory Management Unit (IOMMU)

In addition, the IOCU contains the following features used to enforce transaction ordering.

- Writes are issued to the CM in the order they were received.
- The CM provides an acknowledge (ACK) signal to the IOCU when writes are "visible" (guaranteed that a subsequent CPU read will receive that data).
  - Non-coherent write is acknowledged after serialization
  - Coherent write is acknowledged after intervention complete on all CPUs

1.3.3.1 Software I/O Coherence

For cases where system redesign to accommodate hardware I/O coherence is not feasible, the CPUs and Coherence Manager provide support for an efficient software-managed I/O coherence. This support is through the globalization of hit-type CACHE instructions.

When a coherent address is used for the L1 CACHE operations, the CPU makes a corresponding coherent request. The CM2 sends interventions for the request to all of the CPUs, allowing all of the L1 caches to be maintained together. The basic software coherence routines developed for single CPU systems can be reused with minimal modifications.

1.3.4 Global Interrupt Controller

The Global Interrupt Controller (GIC) handles the distribution of interrupts between and among the CPUs in the cluster. This block has the following features:

- Software interface through relocatable memory-mapped address range.
- Configurable number of system interrupts - from 128 to 1256.
- Support for different interrupt types:
  - Level-sensitive: active high or low.
  - Edge-sensitive: positive-, negative-, or double-edge sensitive.
- Virtualization support allows each interrupt to be mapped to Guest or Root.
- Ability to mask and control routing of interrupts to a particular CPU.
- Support for NMI routing.
- Standardized mechanism for sending inter-processor interrupts.

1.3.5 Global Configuration Registers (GCR)

The Global Configuration Registers (GCR) are a set of memory-mapped registers that are used to configure and control various aspects of the Coherence Manager and the coherence scheme.

1.3.5.1 Inter-CPU Debug Breaks

The MPS includes registers that enable cooperative debugging across all CPUs. Each core features a debug output signal that indicates it has entered debug mode (possibly through a debug breakpoint). Registers are defined that allow CPUs to be placed into debug groups such that whenever one CPU within the group enters debug mode, a debug interrupt is sent to all CPUs within the group, causing them to also enter debug mode and stop executing non-debug mode instructions.

1.3.5.2 CM2 Control Registers

Control registers in the CM2 allow software to configure and control various aspects of the operation of the CM2. Some of the control options include:

- *Address map*: the base address for the GCR and GIC address ranges can be specified. An additional four address ranges can be defined as well. These control whether non-coherent requests go to memory or to memory-mapped I/O. A default can also be selected for addresses that do not fall within any range.
- *Error reporting and control*: Logs information about errors detected by the CM2 and controls how errors are handled (ignored, interrupt, etc.).
- *Control Options*: Various features of the CM2 can be disabled or configured. Examples of this are disabling speculative reads and preventing ReadShared requests from being upgraded to Exclusive.

1.4 Clocking Options

The P6600 core has the following clock domains:

- Cluster domain — This is the main clock domain, and includes all P6600 cores (including optional FPU3) and the CM2 (including Coherence Manager, Global Interrupt Controller, Cluster Power Controller, trace funnel, IOCU, and L2 cache).
- System Domain - The OCP port connecting to the SOC and the rest of the memory subsystem may operate at a ratio of the cluster domain. Supported ratios are 1:1, 1:2, 1:3, 1:4, 1:5, and 1:10.
- TAP domain - This is a low-speed clock domain for the EJTAG TAP controller
- IO Domain - This is the OCP port connecting the IOCU to the I/O Subsystem. This clock may operate at a ratio of the CM2 domain. Supported ratios are the same as the system domain.

1.5 Design For Test (DFT) Features

The P6600 core provides the following test for determining the integrity of the core.

- Internal Scan: The P6600 core supports full mux-based scan for maximum test coverage, with a configurable number of scan chains. ATPG test coverage can exceed 99%, depending on standard cell libraries and configuration options.
- Memory BIST: The P6600 core provides an integrated memory BIST solution for testing of all internal SRAMs.

Memory BIST can also be inserted with a CAD tool or other user-specified method. Wrapper modules and signal buses of configurable width are provided within the core to facilitate this approach.

1.6 Configuration Options

The P6600 provides a number of configuration options as shown in Table 1.1. These are options available to you to select for your P6600 configuration.

Table 1.1 P6600 Multiprocessing System Configuration Options

Parameter	Configurable Options
Number of Cores	1, 2, 3, 4, 5, or 6
L1 Instruction Cache	32 or 64 KB
L1 Data Cache	32 or 64 KB
MIPS64 + SIMD	None or MIPS64 + SIMD
System Interrupts	128 or 256
L2 Cache	512 KB, 1 MB, 2 MB, 4 MB, or 8 MB
Physical Address Bits	40
Location of Boot Exception Vector	Configurable
External Interface Type	OCP or AXI
External Interface Width	128- or 256-bit

Chapter 2

CP0 Registers

The P6600 Multiprocessing System Control Coprocessor (CP0) provides the register interface to the P6600 core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it, referred to as its *register number.* A separate *select number* is used to differentiate additional registers within the *register number.* For example, as shown in the table below, there are eight configuration registers (Selects) within register number 16. If the *select number* is omitted, it is zero.

This chapter contains the following sections:

- Section 2.1 "CP0 Register Summary"
- Section 2.2 "CP0 Register Descriptions"

2.1 CP0 Register Summary

The following two subsections show the CP0 register set grouped by function and grouped by number.

2.1.1 CP0 Registers Grouped by Function

The CP0 registers set are divided into the register groups shown in Table 2.1. Note that assembly programmers modifying certain CP0 registers or register fields must clear any execution or instruction hazards created by the modification.

The following table provides a functional listing of the CP0 registers. Click on a Name column entry to provide a link to the desired register.

Table 2.1 P6600 CP0 Registers Grouped by Function

Category	Register Name	Register Number	Register Select	Location in Document
CPU Configuration and Status	Config	16	0	Section 2.2.1.1 on page 52
	Config1	16	1	Section 2.2.1.2 on page 54
	Config2	16	2	Section 2.2.1.3 on page 57
	Config3	16	3	Section 2.2.1.4 on page 58
	Config4	16	4	Section 2.2.1.5 on page 60
	Config5	16	5	Section 2.2.1.6 on page 62
	Config6	16	6	Section 2.2.1.7 on page 64
	Config7	16	7	Section 2.2.1.8 on page 67
	PRId	15	0	Section 2.2.1.9 on page 71
	EBase	15	1	Section 2.2.1.10 on page 71
	Status	12	0	Section 2.2.1.11 on page 73
	IntCtl	12	1	Section 2.2.1.12 on page 76
TLB Management	Index	0	0	Section 2.2.2.1 on page 79
	EntryLo0	2	0	Section 2.2.2.2 on page 80
	EntryLo1	3	0	
	EntryHi	10	0	Section 2.2.2.3 on page 82
	Context	4	0	Section 2.2.2.4 on page 84
	ContextConfig	4	1	Section 2.2.2.5 on page 85
	XContext	20	0	Section 2.2.2.6 on page 86
	XContextConfig	4	3	Section 2.2.2.7 on page 87
	PageMask	5	0	Section 2.2.2.8 on page 88
	PageGrain	5	1	Section 2.2.2.9 on page 89
	Wired	6	0	Section 2.2.2.10 on page 91
	BadVAddr	8	0	Section 2.2.2.11 on page 91
	PWBase	5	5	Section 2.2.2.12 on page 92
	PWField	5	6	Section 2.2.2.13 on page 93
	PWSize	5	7	Section 2.2.2.14 on page 95
	PWCtl	6	6	Section 2.2.2.15 on page 97
Exception Control	Cause	13	0	Section 2.2.3.1 on page 100
	EPC	14	0	Section 2.2.3.2 on page 104
	ErrorEPC	30	0	Section 2.2.3.3 on page 104
	BadInstr	8	1	Section 2.2.3.4 on page 105
	BadInstrP	8	2	Section 2.2.3.5 on page 106
Timer	Count	9	0	Section 2.2.4.1 on page 107
	Compare	11	0	Section 2.2.4.2 on page 107

Table 2.1 P6600 CP0 Registers Grouped by Function *(continued)*

Category	Register Name	Register Number	Register Select	Location in Document
Cache Management	ITagLo	28	0	Section 2.2.5.1 on page 108
	ITagHi	29	0	Section 2.2.5.2 on page 110
	IDataLo	28	1	Section 2.2.5.3 on page 111
	IDataHi	29	1	Section 2.2.5.4 on page 111
	DTagLo	28	2	Section 2.2.5.5 on page 112
	DDataLo	28	3	Section 2.2.5.6 on page 115
	L23TagLo	28	4	Section 2.2.5.7 on page 116
	L23DataLo	28	5	Section 2.2.5.8 on page 117
	L23DataHi	29	5	Section 2.2.5.9 on page 118
	ErrCtl	26	0	Section 2.2.5.10 on page 118
	CacheErr	27	0	Section 2.2.5.11 on page 120
Shadow Registers	SRSCtl	12	2	Section 2.2.6.1 on page 121
Performance Monitoring	PerfCtl0	25	0	Section 2.2.7.1 on page 123
	PerfCtl1	25	2	
	PerfCtl2	25	4	
	PerfCtl3	25	6	
	PerfCnt0	25	1	Section 2.2.7.2 on page 132
	PerfCnt1	25	3	
	PerfCnt2	25	5	
	PerfCnt3	25	7	
Debug	Debug	23	0	Section 2.2.8.1 on page 132
	DEPC	24	0	Section 2.2.8.2 on page 135
	DESAVE	31	0	Section 2.2.8.3 on page 136
	WatchLo0	18	0	Section 2.2.8.4 on page 136
	WatchLo1	18	1	
	WatchLo2	18	2	
	WatchLo3	18	3	
	WatchHi0	19	0	Section 2.2.8.5 on page 137
	WatchHi1	19	1	
	WatchHi2	19	2	
	WatchHi3	19	3	

Table 2.1 P6600 CP0 Registers Grouped by Function *(continued)*

Category	Register Name	Register Number	Register Select	Location in Document
PDTrace	TraceControl	23	1	Section 2.2.9.1 on page 138
	TraceControl2	23	2	Section 2.2.9.2 on page 140
	TraceControl3	24	2	Section 2.2.9.3 on page 142
	UserTraceData1	23	3	Section 2.2.9.4 on page 143
	UserTraceData2	24	3	Section 2.2.9.5 on page 144
	TraceIBPC	23	4	Section 2.2.9.6 on page 144
	TraceDBPC	23	5	Section 2.2.9.7 on page 145
User Mode Support	HWREna	7	0	Section 2.2.10.1 on page 147
	UserLocal	4	2	Section 2.2.10.2 on page 148
Kernel Mode Support	KScratch1	31	2	Section 2.2.11 on page 150
	KScratch2	31	3	
	KScratch3	31	4	
	KScratch4	31	5	
	KScratch5	31	6	
	KScratch6	31	7	
Memory Mapped	CDMMBase	15	2	Section 2.2.12.1 on page 152
	CMGCRBase	15	3	Section 2.2.12.2 on page 153
Virtualization	GuestCtl0	12	6	Section 2.2.13.1 on page 154
	GuestCtl1	10	4	Section 2.2.13.2 on page 158
	GuestCtl2	10	5	Section 2.2.13.3 on page 159
	GuestCtl0Ext	11	4	Section 2.2.13.4 on page 161
	GTOffset	12	7	Section 2.2.13.5 on page 163
Memory Accessibility Attribute	MAAR	17	1	Section 2.2.14.1 on page 165
	MARRI	17	2	Section 2.2.14.2 on page 168
Memory Segmentation	SegCtl0 - SegCtl2	5	2 - 4	Section 2.2.15 on page 169

2.1.2 CP0 Registers Grouped by Number

The following table provides a numerical listing of the P6600 CP0 registers. Click on a Name column entry to provide a link to the desired register.

Table 2.2 CP0 Registers Grouped by Number

Register			Function	Location
Num	Sel	Name		
0	0	Index	Index into the TLB array	Section 2.2.2.1
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages.	Section 2.2.2.2
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages.	
4	0	Context	Pointer to page table entry in memory.	Section 2.2.2.4
4	1	ContextConfig	Defines the bits of the Context register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted.	Section 2.2.2.5
4	2	UserLocal	User information that can be written by privileged software and read via RDHWR register 29	Section 2.2.10.2
4	3	XContextConfig	Defines the bits of the XContext register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted.	Section 2.2.2.7
5	0	PageMask	PageMask controls the variable page sizes in TLB entries.	Section 2.2.2.8
5	1	PageGrain	PageGrain controls the granularity of the page sizes in TLB entries.	Section 2.2.2.8
5	5	PWBase	Hardware page table walker base address register.	Section 2.2.2.12
5	6	PWField	Hardware page table walker field configuration register.	Section 2.2.2.13
5	7	PWSize	Hardware page table walker size register.	Section 2.2.2.14
6	0	Wired	Controls the number of fixed (“wired”) TLB entries. This register is reserved if the TLB is not implemented.	Section 2.2.2.10
6	6	PWCtl	Hardware page table walker configuration register.	Section 2.2.2.15
7	0	HWREna	Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode.	Section 2.2.10.1
8	0	BadVAddr	Reports the address for the most recent address-related exception.	Section 2.2.2.11
8	1	BadInstr	Captures the most recent instruction that caused the exception.	Section 2.2.3.4
8	2	BadInstrP	Used in conjunction with the BadInstr register. Contains the prior branch instruction, when the faulting instruction is in a branch delay slot.	Section 2.2.3.5
9	0	Count	Processor cycle count.	Section 2.2.4.1
10	0	EntryHi	High-order portion of the TLB entry. This register is reserved if the TLB is not implemented.	Section 2.2.2.3
10	4	GuestCtl1	Guest ID register used in Virtualization.	Section 2.2.13.2
10	5	GuestCtl2	Guest interrupt-related register used in virtualization.	Section 2.2.13.3
11	0	Compare	Timer interrupt control.	Section 2.2.4.2
11	4	GuestCtl0Ext	Extension of guest control register used in virtualization.	Section 2.2.13.4
12	0	Status	Processor status and control.	Section 2.2.1.11

Table 2.2 CP0 Registers Grouped by Number *(continued)*

Register			Function	Location
Num	Sel	Name		
12	1	IntCtl	Setup for interrupt vector and interrupt priority features.	Section 2.2.1.12
12	2	SRSCtl	Shadow register set control.	Section 2.2.6.1
12	6	GuestCtl0	Guest mode control register used in virtualization.	Section 2.2.13.1
12	7	GTOffset	Guest timer offset register used in virtualization.	Section 2.2.13.5
13	0	Cause	Cause of last exception.	Section 2.2.3.1
14	0	EPC	Program counter at last exception.	Section 2.2.3.2
15	0	PRId	Processor identification and revision.	Section 2.2.1.9
15	1	EBase	Exception base address.	Section 2.2.1.10
15	2	CDMMBase	Common Device Memory Map Base Address.	Section 2.2.12.1
15	3	CMGCRBase	Defines the 36-bit physical base address for the memory-mapped Coherency Manager Global Configuration Register (CMGCR) space.	Section 2.2.12.1
16	0	Config	Configuration register.	Section 2.2.1.1
16	1	Config1	Configuration for MMU, caches etc.	Section 2.2.1.2
16	2	Config2	Configuration for MMU, caches etc.	Section 2.2.1.3
16	3	Config3	Interrupt and ASE capabilities	Section 2.2.1.4
16	4	Config4	Indicates presence of Config5 register	Section 2.2.1.5
16	5	Config5	Provides information on EVA and cache error exception vector.	Section 2.2.1.6
16	5	Config6	Provides information about the presence of optional extensions to the base MIPS64 architecture.	Section 2.2.1.7
16	7	Config7	P6600 family-specific configuration register.	Section 2.2.1.8
17	1	MAAR	Memory accessibility attribute register.	Section 2.2.14.1
17	2	MARRI	Memory accessibility attribute index register.	Section 2.2.14.2
18	0	WatchLo0	Watchpoint address associated with instruction watchpoint 0.	Section 2.2.8.4
18	1	WatchLo1	Watchpoint address associated with instruction watchpoint 1.	
18	2	WatchLo2	Watchpoint address associated with data watchpoints 0.	
18	3	WatchLo3	Watchpoint address associated with data watchpoints 1.	
19	0	WatchHi0	Watchpoint ASID and Mask associated with instruction watchpoint 0.	Section 2.2.8.5
19	1	WatchHi1	Watchpoint ASID and Mask associated with instruction watchpoint 1.	
19	2	WatchHi2	Watchpoint ASID and Mask associated with data watchpoint 0.	
19	3	WatchHi3	Watchpoint ASID and Mask associated with data watchpoint 1.	
20	0	XContext	Pointer to page table entry in memory.	Section 2.2.2.6
23	0	Debug	EJTAG Debug register.	Section 2.2.8.1
23	1	TraceControl	PDTrace control register 1.	Section 2.2.9.1
23	2	TraceControl2	PDTrace control register 2.	Section 2.2.9.2
23	3	UserTraceData1	PDTrace user trace data 1.	Section 2.2.9.4
23	4	TraceIBPC	Trace instruction breakpoint condition.	Section 2.2.9.6
23	5	TraceDBPC	Trace data breakpoint condition.	Section 2.2.9.7

Table 2.2 CP0 Registers Grouped by Number *(continued)*

Register			Function	Location
Num	Sel	Name		
24	0	DEPC	Restart address from last EJTAG debug exception.	Section 2.2.8.2
24	2	TraceControl3	PDTrace Control 3.	Section 2.2.9.3
24	3	UserTraceData2	PDTrace user trace data 2.	Section 2.2.9.5
25	0	PerfCtl0	Performance counter 0 control.	Section 2.2.7.1
25	1	PerfCnt0	Performance counter 0 count.	Section 2.2.7.2
25	2	PerfCtl1	Performance counter 1 control.	Section 2.2.7.1
25	3	PerfCnt1	Performance counter 1 count.	Section 2.2.7.2
25	4	PerfCtl2	Performance counter 2 control.	Section 2.2.7.1
25	5	PerfCnt2	Performance counter 2 count.	Section 2.2.7.2
25	6	PerfCtl3	Performance counter 3 control.	Section 2.2.7.1
25	7	PerfCnt3	Performance counter 3 count.	Section 2.2.7.2
26	0	ErrCtl	Software test enable of way-select and Data RAM arrays for I-Cache and D-Cache.	Section 2.2.5.10
27	0	CacheErr	Records information about cache parity errors	Section 2.2.5.11
28	0	ITagLo	Cache tag read/write interface for I-cache.	Section 2.2.5.1
28	1	IDataLo	Low-order data read/write interface for I-cache.	Section 2.2.5.3
28	2	DTagLo	Cache tag read/write interface for D-cache.	Section 2.2.5.5
28	3	DDataLo	Low-order data read/write interface for D-cache.	Section 2.2.5.6
28	4	L23TagLo	Cache tag read/write interface for L2-cache.	Section 2.2.5.7
28	5	L23DataLo	Low-order data read/write interface for L2-cache.	Section 2.2.5.8
29	0	ITagHi	Cache tag read/write interface for I-cache, upper 32 bits.	Section 2.2.5.1
29	1	IDataHi	High-order data read/write interface for I-cache.	Section 2.2.5.4
29	5	L23DataHi	High-order data read/write interface for L2-cache.	Section 2.2.5.9
30	0	ErrorEPC	Program counter at last error.	Section 2.2.3.3
31	0	DESAVE	Debug handler scratchpad register.	Section 2.2.8.3
31	2	KScratch1	Kernel scratch pad register 1.	Section 2.2.11
31	3	KScratch2	Kernel scratch pad register 2.	Section 2.2.11
31	4	KScratch3	Kernel scratch pad register 3.	Section 2.2.11
31	5	KScratch4	Kernel scratch pad register 4.	Section 2.2.11
31	6	KScratch5	Kernel scratch pad register 5.	Section 2.2.11
31	7	KScratch6	Kernel scratch pad register 6.	Section 2.2.11

2.2 CP0 Register Descriptions

The following subsections describe the CP0 registers listed in Table 2.1 above.

2.2.1 CPU Configuration and Status Registers

This section contains the following CPU Configuration and Status registers.

2.2.1.1 Device Configuration — Config (CP0 Register 16, Select 0)

The main role of the *Config* register is to be a read-only repository of information about the P6600 core resources, encoded so as to be useful to operating system initialization code.

Figure 2.1 Config Register Format

31	30 28	27 25	24	23	22	21	20 19	18	17	16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	ISP	DSP	UDI	SB	0	MM	0	BM	BE	AT	AR	MT	0	VI	K0

Table 2.3 Field Descriptions for Config Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	This bit is hardwired to '1' to indicate the presence of the *Config1* register.	R	1
K23	30:28	These fields are unused in the P6600 core since the TLB structure is supported. They should be written as zero only.	R/W	0x0
KU	27:25		R/W	0x0
ISP	24	Instruction Scratch Pad RAM present. This bit is always 0 in the P6600 core. 0: Instruction scratch pad RAM *(*ISPRAM*)* is not implemented. 1: Instruction scratch pad RAM *(*ISPRAM*)* is implemented.	R	0
DSP	23	Data Scratch Pad RAM present. This bit is always 0 in the P6600 core. 0: Data scratch pad RAM *(*DSPRAM*)* is not implemented. 1: Data scratch pad RAM *(*DSPRAM*)* is implemented.	R	0
UDI	22	User-Defined Instruction. This bit is always 0 in the P6600 core. 0: The P6600 core does not contain user-defined "CorExtend" instructions. 1: The P6600 core contains user-defined "CorExtend" instructions.	R	0
SB	21	Read-only "SimpleBE" bus mode indicator, which reflects the P6600 input signal *SI_SimpleBE*. 0: No reserved byte enabled on the OCP interface. 1: Only simple byte enables allows on the OCP interface. If set by hardware, the P6600 core will only do simple partial-word transfers on its OCP interface; that is, the only partial-word transfers will be byte, aligned half-word, and aligned word. If zero, it may generate partial-word transfers with an arbitrary set of bytes enabled. This generates less requests, but may not be supported by all downstream devices.	R	Externally Set
0	20:19	Must be written as zero; returns zero on read.	R	0
MM	18	Write Merge.This bit indicates whether write-through merging is enabled in the 32-byte collapsing write buffer. 0: No merging allowed 1: Merging allowed Setting this bit allows writes resulting from separate store instructions in write-through mode to be merged into a single transaction at the interface. The state of this bit does not affect cache writebacks (which are always whole blocks together) or uncached writes (which are never merged).	R/W	1
0	17	Must be written as zero; returns zero on read.	R	0
BM	16	Burst Mode. 0: Sequential burst mode 1: Subblock burst mode This bit reads 0 when the bus uses sequential burst ordering and reads 1 when it uses sub-block burst ordering. This bit is set by the inputsignal *SI_SBlock* signal to match the system controller.	R	0
BE	15	Endian mode. 0: Little endian 1: Big endian This bit is written by hardware based on the state of the *SI_Endian* input pin.	R	Externally Set

Table 2.3 Field Descriptions for Config Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
AT	14:13	Architecture type implemented by the processor. This field is always 0x2 to indicate the MIPS64 architecture.	R	0x2
AR	12:10	Architecture release. 0x2: Release 6 This bit always reads 2 to reflect Release 6 of the MIPS64 architecture.	R	0x2
MT	9:7	MMU type: This field is encoded as follows. For Root mode, this field has a default value of 3'b001. In Guest mode, the Root can write the Guest.Config.MT field with a value of 3'b001 or 3'b100 depending on whether an FTLB is implemented. 000: Reserved 001: VTLB Only 010 - 011: Reserved 100: VTLB + FTLB 101 - 111: Reserved	R	0x1 or 0x4
0	6:4	Must be written as zero; returns zero on read.	R	0
VI	3	Virtually indexed. This bit is set by hardware and is 0 to indicate that the L1 instruction cache is physically tagged.	R	0
K0	2:0	Kseg0 coherency attribute of the page. See Table 2.19 for the field encoding.	R/W	2

2.2.1.2 Device Configuration 1 — Config1 (CP0 Register 16, Select 1)

The Config1 register provides information such as the size of the VTLB and the L1 instruction and data cache parameters. It also contains a series of single bits that indicate the presence of selected logic units on the P6600 core.

Figure 2.2 Config1 Register Format

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMUSize	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 2.4 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Continuation bit, set to 1 to indicate that the *Config2* register is implemented.	R	1
MMUSize	30:25	The size of the TLB array (the array has MMUSize + 1 entries). Refer to the *Config4* register for more information. In Root mode, this field has a default value of 0x3F. In Guest mode, the Root can write the Guest.Config1.MMUSize field with another default value depending on the size of the MMU.	R	0x3F
IS	24:22	L1 Instruction cache number of sets per way. This field indicates the number of sets per way in the L1 instruction cache. The number of sets is multiplied by the number of ways and the line size to derive the cache size. In this case, the number of sets defines the cache size since the line size and number of ways in the P6600 core are fixed. This field is encoded as follows: 000 - 001: Reserved 010: 256 sets per way (equates to 32 KByte instruction cache) 011: 512 sets per way (equates to 64 KByte instruction cache) 100 - 111: Reserved Because the line size and associativity are fixed for the P6600 instruction cache as defined in the IL and IA fields below, the IS field is used to determine the overall cache size as follows: If this field is set to 2, the instruction cache size would be: 256 sets/way x 32 bytes/line x 4 sets per way = 32 KBytes. If this field is set to 3, the instruction cache size would be: 512 sets/way x 32 bytes/line x 4 sets per way = 64 KBytes.	R	Preset
IL	21:19	L1 Instruction cache line size. In the P6600 core, the instruction cache line size is fixed at 32 bytes. As such, this field is encoded as follows: 000 - 011: Reserved 100: 32 byte line size 101 - 111: Reserved	R	Preset
IA	18:16	L1 Instruction cache associativity. In the P6600 core, the instruction cache associativity is fixed at 4 ways. As such, this field is encoded as follows: 000 - 010: Reserved 011: 4-ways 100 - 111: Reserved A default value of 3 indicates a 4-way set associative instruction cache. Refer to the IS field above to determine how to calculate the size of the L1 instruction cache.	R	3

Table 2.4 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
DS	15:13	L1 Data cache number of sets per way. This field indicates the number of sets per way in the L1 data cache and is encoded as follows: The number of sets is multiplied by the number of ways and the line size to derive the cache size. In this case, the number of sets defines the cache size since the line size and number of ways in the P6600 core are fixed. This field is encoded as follows: 000 - 001: Reserved 010: 256 sets per way (equates to 32 KByte instruction cache) 011: 512 sets per way (equates to 64 KByte instruction cache) 100 - 111: Reserved Because the line size and associativity are fixed for the P6600 data cache as defined in the DL and DA fields below, the DS field is used to determine the overall cache size as follows: If this field is set to 2, the data cache size would be: 256 sets/way x 32 bytes/line x 4 sets per way = 32 KBytes. If this field is set to 3, the data cache size would be: 512 sets/way x 32 bytes/line x 4 sets per way = 64 KBytes.	R	Preset
DL	12:10	L1 data cache line size. In theP6600 core, the data cache line size is fixed at 32 bytes. As such, this field is encoded as follows: 000 - 011: Reserved 100: 32 byte line size 101 - 111: Reserved	R	Preset
DA	9:7	L1 data cache associativity. In the P6600 core, the data cache associativity is fixed at 4 ways. As such, this field is encoded as follows: 000 - 010: Reserved 011: 4-ways 100 - 111: Reserved A default value of 3 indicates a 4-way set associative data cache.	R	3
C2	6	This bit is always 0 to indicate that a coprocessor 2 is not supported.	R	Preset
MD	5	MDMX Application Specific Extension (ASE). A logic ‘0’ indicates that the MDMX ASE is not implemented in the floating point unit (FPU) of the P6600 core.	R	0
PC	4	Performance counter enable. There are four performance counters implemented in the P6600 core. For the Root version of this register, this bit is always a logic ‘1’. For the Guest version of this register, this bit can be cleared by the root using the MTGC0 instruction. Refer to the *PerfCtl0-3* and *PerfCnt0-3* registers for more information.	R	1
WR	3	Watchpoint registers present. This bit always reads 1 because the P6600 core always contains watchpoint registers. Refer to the *WatchLo 0-3*/*WatchHi 0-3* registers in Section 2.2.8.4 “Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)”.	R	1
CA	2	MIPS16e present. This bit always reads 0 to indicate the MIPS16e compressed-code instruction set is not available.	R	0
EP	1	EJTAG unit present. This bit always reads 1 as the EJTAG debug unit is provided on the P6600 core.	R	1
FP	0	Floating Point Unit present. This bit is set to indicate that a floating point unit is present. The floating point unit is optional on the P6600 core.	R	Preset

2.2.1.3 Device Configuration 2 — Config2 (CP0 Register 16, Select 2)

The *Config2* register provides information about the size and organization of L2 and L3 caches. The *Config2* register also has fields that indicate the presence of some extensions to the base MIPS64 architecture.

An L3 cache can be used with the P6600 core. However, the core does not support passing of the L3 configuration information via the Config2 register. As such, the TU, TS, TL and TA bits of this register, which handle L3 operations, are not used and are all tied to 0. Information on L3 transfers may be available in an implementation specific register elsewhere in the system.

Figure 2.3 Config2 Register Format

31	30 28	27 24	23 20	19 16	15 13	12	11 8	7 4	3 0
M	TU	TS	TL	TA	SU	L2B	SS	SL	SA

Table 2.5 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config3 register.	R	1
TU	30:28	An L3 cache can be used with the P6600 core. However, the core does not support passing of the L3 configuration data via the Config2 register. As such, the TU, TS, TL and TA bits of this register, which report L3 information, are not used and are all tied to 0. Details of the L3 configuration may be available in an implementation specific register elsewhere in the system.	R	0
TS	27:24		R	0
TL	23:20		R	0
TA	19:16		R	0
SU	15:13	These bits are reserved in the P6600 core and is always 0.	R	0
L2B	12	L2 cache bypass. Setting this bit disables or bypasses the L2 cache. Setting this bit also forces $Config2_{SL}$ to 0. Based on this information, most operating system code will conclude that there is no L2 cache in the system. Setting this bit forces hardware to drive a series of internal handshake signals between the core to the CM2, placing the L2 cache into bypass mode. When this bit is set through a write operation, a subsequent read of this bit will not indicate a logic 1 until the L2 has asserted its internal handshake signal, indicating that it has been bypassed.	R/W	0

Table 2.5 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
SS	11:8	L2 cache number of sets per way. This field indicates the number of sets per way in the L2 cache of the Coherent Processing System (CPS) and is written by hardware at reset based on the state of the *L2_Sets[3:0]* signals. At IP configuration time, the user selects the cache size and the line size. Hardware then takes this information and selects the appropriate number of sets. See the example formulas below for determining the number of sets based on cache and line size. This field is encoded as follows: 0x0 - 0x3: Reserved 0x4: 1024 sets per way 0x5: 2048 sets per way 0x6: 4096 sets per way 0x7: 8192 sets per way 0x8: 16384 sets per way 0x9: 32768 sets per way 0xA- 0xF: Reserved For example: If this field is set to 0x4, the SL field is set to 0x5, and the SA field is set to 0x4, the L2 cache size would be: 1024 sets/way x 64 bytes/line x 8 ways = 512 KBytes Conversely, if this field is set to 0x9, the SL field is set to 0x4, and the SA field is set to 0x4, the L2 cache size would be: 32768 sets/way x 32 bytes/line x 8 ways = 8 MBytes	R	Preset
SL	7:4	L2 cache line size. The L2 cache line sizes can be configured at 32 or 64 bytes. This field is written by hardware at reset based on the state of the *L2_LineSize[3:0]* signals. These signals are driven based on the customer's line size choice during IP configuration. As such, this field is encoded as follows: 0x0 - 0x3: Reserved 0x4: 32 byte line size 0x5: 64 byte line size 0x6 - 0xF: Reserved	R	Preset
SA	3:0	L2 cache associativity. In the P6600 core, the L2 cache associativity is fixed at 8 ways. This field is written by hardware at reset based on the state of the *L2_Assoc[3:0]* signals. As such, this field is encoded as follows: 0x0 - 0x6: Reserved 0x7: 8-way set associative 0x8 - 0xF: Reserved	R	0x7

2.2.1.4 Device Configuration 3 — Config3 (CP0 Register 16, Select 3)

Config3 provides information about the presence of optional extensions to the base MIPS64 architecture in addition to those specified in *Config2*. All fields in the *Config3* register are read-only.

If Virtualization is supported ($Config3_{VZ}$ = 1), and GuestID is supported, then explicit invalid TLB entry support (EHINV) is required in order for a Guest to be able to detect invalid entries in the Guest TLB.

Figure 2.4 Config3 Register Format

31	30	29	28	27	26	25	24	23	22 16
M	BPG	CMGCR	MSAP	BP	BI	SC	PW	VZ	0

15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ISA	ULRI	RXI	DSP2P	DSPP	CTXTC	0	LPA	VEIC	VInt	SP	CDMM	MT	SM	TL

Table 2.6 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. This bit is always one to indicate the presence of *Config4*.	R	1
BPG	30	Big pages. This bit is always 1 to indicate that TLB pages larger than 256 MB are supported and that the CP0 PageMask Register is 64-bits wide.	R	1
CMGCR	29	Reads 1 to indicate that the Coherence Manager has a Global Configuration Register Space and the CMGCRBase cop0 register is implemented.	R	1
MSAP	28	MIPS SIMD architecture implemented. This bit indicates if the MIPS SIMD architecture is implemented and is encoded as follows: 0: MSA module not implemented 1: MSA module is implemented	R	Preset
BP	27	*BadInstrP* register implemented. This bit indicates whether the faulting prior branch instruction word register is present. This bit is always set in the P6600 core to indicate the presence of the *BadInstrP* register.	R	1
BI	26	*BadInstr* register implemented. This bit indicates whether the faulting branch instruction word register is present. This bit is always set in the P6600 core to indicate the presence of the *BadInstr* register.	R	1
SC	25	Segment Control implemented. This bit indicates whether the Segment Control registers *SegCtl0*, *SegCtl1* and *SegCtl2* are present. This bit is always 1 in the P6600 core.	R	1
PW	24	HardWare page table walk implemented. This bit indicates whether the page table walking registers PWBase, PWField and PWSize are present. This bit is encoded as follows: 0: Page table walking not implemented. 1: Page table walking is implemented	R	Preset
VZ	23	Virtualization Module implemented. This bit indicates whether the Virtualization Module is implemented. This bit is always 1 for the P6600 core. 0: Virtualization module not implemented 1: Virtualization module is implemented	R	1
0	22:16	Must be written as zero; returns zero on read.	R	0
ISA	15:14	Indicates the instruction set availability. This bit is always 0 to indicate MIPS64.	R	0
ULRI	13	Reads 1 to indicate that the *UserLocal* Register is implemented.	R	1
RXI	12	Reads 1 to indicate that the *RIE* and *XIE* fields exist in the *PageGrain* register.	R	1
DSP2P	11	Indicates the MIPS DSP ASE revision. This bit is ignored in the P6600 core. 0: Revision 1 (DSP R1) 1: Revision 2 (DSP R2)	R	0

Table 2.6 Field Descriptions for Config3 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
DSPP	10	Reads 1 to indicate that the MIPS DSP ASE extension is implemented. This bit is always 0 in the P6600 core.	R	0
CTXTC	9	Reads 1 to indicate the *ContextConfig* register is implemented. The width of the *BadVPN2* field in the *Context* register depends on the contents of the *ContextConfig* register.	R	1
0	8	Must be written as zero; returns zero on read.	R	0
LPA	7	Large physical address support is implemented, and the PageGrain register exists. The following Coprocessor 0 fields and associated control are present if this bit is a 1: • Modifications to the EntryLo0/1, EntryHi, and BadVaddr registers to support 40-bit physical addresses of the P6600. • Modifications to other optional COP0 registers with PA: LLAddr, ITagLo and DTagLo. • PageGrain • Config5.MVH	R	1
VEIC	6	Support for an external interrupt controller. This bit is set or cleared by hardware depending on whether the EIC option was selected at build time. 0: Support for EIC mode not supported. 1: Support of EIC mode supported. The value of this bit is set by the static input, *SI_EICPresent*. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not	R	Externally Set
VInt	5	Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented. On the P6600 core, this bit reads 1 to indicate the CPU can handle vectored interrupts.	R	1
SP	4	Reads 0 to indicate the CPU does not support 1 Kbyte TLB pages.	R	0
CDMM	3	Reads 1 to indicate the Common Device Memory Map (CDMM) feature is implemented, as well as the *CDMMBase* register is present.	R	1
MT	2	Reads 0 to indicate the P6600 core does not include the MIPS MT module.	R	0
SM	1	Reads 0 to indicate the P6600 does not include the instructions of the SmartMIPS ASE.	R	0
TL	1	Reads 1 to indicate PDTrace is supported.	R	0

2.2.1.5 Device Configuration 4 — Config4 (CP0 Register 16, Select 4)

The *Config4* register encodes additional capabilities such as TLBINV instruction support and the number of kernel scratch registers.

Figure 2.5 Config4 Register Format

31	30 29	28	27 24	23 16	15 13	12 8	7 4	3 0
M	IE	AE	VTLB SizeExt	KScrExist	0	FTLB Page Size	FTLB Ways	FTLB Sets

Table 2.7 Field Descriptions for Config4 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. This bit is one to indicate the presence of *Config5*.	R	1
IE	30:29	TLBINV instruction support. For this field, the P6600 core only returns the following encoding. 10: TLBINV, TLBINVF instruction supported, $EntryHi_{EHINV}$ supported. TLBINV, TLBINVF instruction operate on one TLB entry.	R	0x2
AE	28	If this bit is set, then *EntryHI.ASID* is extended to 10 bits.	R	Preset
VTLBSizeExt	27:24	VTLB size extension. This field is used to extend the size of the VTLB. This field is always concatenated to the left of the most-significant bit of the $Config1_{MMUSize}$. In the P6600 core the VTLB size is fixed. Hence this field is not used.	R	Preset
KScrExist	23:16	Indicates how many scratch registers are available to kernel-mode software within CP0 Register 31. In the P6600 architecture, six kernel scratch registers are included at register selects 2 - 7. Each bit represents a select for CP0 Register 31. Bit 16 represents Select 0, Bit 23 represents Select 7. If the bit is set, the associated scratch register is implemented and available for kernel-mode software. Therefore, this field contains a value of 0xFC (8'b11111100). This indicates that bits 18 - 23 are set, corresponding to selects 2 - 7. These registers are used by the kernel for temporary storage of information. Refer to Section 2.2.11, "Kernel Mode Support Registers" on page 150 for more information.	R	0xFC
0	15:13	Reserved. Must be written as zero. Ignored on reads.	R	0
FTLB Page Size	12:8	Indicates the Page Size of the FTLB Array Entries. The FTLB must be flushed of any valid entries before this register field value is changed by software. The FTLB behavior is UNDEFINED if there are valid FTLB entries which were not all programmed using a common page size. This field is encoded as follows: 00000: Reserved 00001: 4 KB 00010: 16 KB 00011 - 11111: Reserved	R/W	0x01
FTLB Ways	7:4	Indicates the set associativity of the FTLB array, which is fixed at 4 in the P6600 architecture. This field is encoded as follows: 0000 - 0001: Reserved 0010: 4 way 0011 - 1111: Reserved	R	0x2
FTLB Sets	3:0	Indicates the number of sets per way within the FTLB array, which is fixed at 128 in the P6600 architecture. This field is encoded as follows: 0000 - 0110: Reserved 0111: 128 sets 1000 - 1111: Reserved	R	0x7

2.2.1.6 Device Configuration 5 — Config5 (CP0 Register 16, Select 5)

The *Config5* register encodes additional capabilities for the address mode programming and cache error exceptions.

Figure 2.6 Config5 Register Format

31	30	29	28	27	26–16
M	K	CV	EVA	MSAEN	0

15–14	13	12	11	10	9	8	7	6	5	4	3	2–1	0
0	XNP	0	DEC	L2C	UFE	FRE	0	SBRI	MVH	LLB	MRP	0	NFE

Table 2.8 Field Descriptions for Config5 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. Even though the *Config6 and Config7* registers are used in the P6600 Multiprocessing System, they are both defined as implementation-specific registers. As such, this bit is zero and is not used to indicate the presence of *Config6*.	R	0
K	30	This bit effects the cache coherency attributes, the boot exception vector overlay, and the location of the exception vector as follows: When this bit is cleared, the following events occur: 1. The $Config_{K0}$ 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. 3. The exception vectors are forced to reside in kseg0/kseg1 by ignoring the state of bits 31:30 of the *EBase* register as well as the *SI_ExceptionBase[31:30]* pins and forcing them to a value of 2’b10. When this bit is set, the following events occur: 1: The $Config_{k0}$ field is ignored and the cache coherency attributes are derived from the 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 *SI_ExceptionBase[31:30]* signals and used to place the exception vectors anywhere within virtual address space.	R/W	0
CV	29	Cache error exception vector control. Disables logic forcing use of kseg1 region in the event of a Cache Error exception when $Status_{BEV}$ = 0. When the CV bit is cleared, bits 31:30 of the *EBase* Register are fixed with the value 2’b10 to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments. Bit 29 of exception base address will be forced to 1 on Cache Error exceptions so the exception handler will be executed from the uncached kseg1 segment. When the CV bit is set, the ExcBase field is expanded to include bits 31:30 to facilitate programmable memory segmentation.	R/W	0
EVA	28	This bit is always a logic one to indicate support for enhanced virtual address (EVA).	R	1

Table 2.8 Field Descriptions for Config5 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
MSAEN	27	MIPS SIMD architecture (MSA) enable. This bit is encoded as follows: 0: MSA instructions and registers are disabled. Executing an MSA instruction causes a MSA disabled exception. 1: MSA instructions and registers are enabled.	R/W	0
Reserved	26:14	Reserved. Must be written as zero. Ignored on reads.	R	0
XNP	13	Extended LL/SC family of instructions. The LLX/SCX family of instructions is required for Release 6 Double-Width atomic support. This support is provided by extending the capability of legacy LL/SC instructions. 0: LLX/SCX instruction family supported 1: LLX/SCX instruction family not supported This bit is always 1 in the P6600 core. This bit can be read in user mode by setting the XNP bit in the HWREna CP0 register. Refer to Section 2.2.10.1, "Hardware Enable — HWREna (CP0 Register 7, Select 0)".	R	1
0	12	Reserved. Must be written as zero. Ignored on reads.	R	0
DEC	11	Dual Endian Capability. Determines endian capability of processor.If both modes are supported, then the processor will initially boot in little-endian mode always. Thereafter, software can force a change in endian mode by setting a bit in a memory-mapped external register. The endian mode change will only take effect on a subsequent reset. For current endian state, software should read Config.BE. 0: Only Little-Endian mode supported. Any implementation must support Little-endian mode. 1: Both Little and Big-Endian modes supported.	R	1
L2C	10	Indicates presence of COP0 Config2. 0: Config2 present. Software can read Config2 to determine L2/L3 cache configuration. 1: Config2 not present. Replaced by memory mapped register that software can read instead.	R	0
UFE	9	Enable for user mode access to Config5.FRE. User mode can conditionally access Config5.FRE using CTC1 and CFC1 instructions. 0: An attempt by the user to read/write Config5.FRE causes a Reserved Instruction exception. 1: User is allowed to write Config5.FRE (only) using CTC1, and read Config5.FRE (only) using CFC1. A kernel can access Config5 using MTC0/MFC0. Config5.UFE applies also to kernel use of CFC1/CTC1. Config5.UFE is reserved if: FIR.FREP is 0 or Config1.FP=0.	R/W	0
FRE	8	Enable for user mode to emulate Status.FR = 0 handling on an FPU with Status.FR hardwired to 1. User mode can conditionally access Config5.FRE using the CTC1 and CFC1 instructions. Release 6 eliminates the Status.FR = 0. If Status.UFE = 0, which is always the case in the P6600 core, then FRE always equals 0. 0: Instructions impacted by Config5.FRE do not generate additional exception conditions. 1: The following instructions cause a Reserved Instruction exception: - All single-precision FP arithmetic instructions. - All LWC1 and MTC1 instructions. - All SWC1 and MFC1 instructions. COP1 branches are not affected by Config5.FRE.	R/W	0

Table 2.8 Field Descriptions for Config5 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
0	7	Reserved. Must be written as zero. Ignored on reads.	R	0
SBRI	6	SDBBP instruction Reserved Instruction control. The purpose of this field is to restrict availability of SDBBP to kernel mode operation. It prevents user (and supervisor) code from entering Debug mode using SDBBP. 0: SDBBP instruction executes as defined prior to Release 6. 1: SDBBP instruction can only be executed in kernel mode. User or supervisor execution of SDBBP causes a Reserved Instruction exception.	R/W	0
MVH	5	Move To/From High COP0 (MTHC0/MFHC0) instructions. These instruction are not used in the P6600, hence this bit is always 0. 0: MTHC0 and MFHC0 instructions are not supported. 1: MTHC0 and MFHC0 instruction are supported.	R	0
LLB	4	Load-Linked Bit software support present. Features enabled by setting this bit are recommended if Virtualization is supported ($Config3_{VZ}$ = 1). This bit is set by hardware to indicate support for LLB and is encoded as follows: 0: LLB functionality is not supported. 1: LLB functionality is supported. When this bit is set, the following features are supported. • ERETNC instruction added. • CP0 $LLAddr_{LLB}$ bit must be set. • LLbit is software accessible through the LLADDR[0] bit in the LLADDR register.	R	1
MRP	3	COP0 Memory Accessibility Attribute Registers, MAAR and MAARI, present. This bit is encoded as follows: 0: MAAR and MAARI not present. 1. MAAR and MAARI present. Software may program these registers to apply additional attributes to fetch, load, or store accesses to memory/IO address ranges.	R	1
0	2:1	Reserved. Must be written as zero. Ignored on reads.	R	0
NFE	0	Nested fault. Setting this bit indicates that the nested fault feature exists. The nested fault allows recognition of faulting behavior within an exception handler.	R	0

2.2.1.7 Device Configuration 6 — Config6 (CP0 Register 16, Select 6)

Config6 provides information about the presence of optional extensions to the base MIPS64 architecture. Note that this register is implemented only by the root context and not by the guest context.

Figure 2.7 Config6 Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17 16
0	FPDSR	DOPC	0	DSFW	DWP	DL1B	DNPE	ODTG	ODDG	DLSB	DFIS	HITLB	HDTLB	FTLBP

15	14	13	12 10	9 1	0
FLTBEn	SPCD	0	IFUPerfCtl	0	JRCD

Table 2.9 Field Descriptions for Config6 Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:30	Reserved. Write as zero. Ignored on reads.	R	0
FPDSR	30	Floating point disable square root. 0: Enable floating point divide and square root 1: Disable floating point divide and square root	R/W	0
DOPC	29	Opcode cache disable. Setting this bit indicates that the opcode cache is disabled. 0: Opcode cache is enabled. 1: Opcode cache is disabled.	R/W	0
0	28	Reserved. Write as zero. Ignored on reads.	R	0
DSFW	27	Disable superforwarding. 0: Enable superforwarding. 1: Disable superforwarding.	R/W	0
DWP	26	Disable IFU way prediction. 0: Enable IFU way prediction. 1: Disable IFU way prediction.	R/W	0
DL1B	25	Disable L1 branch target buffer. 0: Enable L1 branch target buffer. 1: Disable L1 branch target buffer.	R/W	0
DNPE	24	Disable NOP elimination. 0: Enable NOP elimination. 1: Disable NOP elimination.	R/W	0
ODTG	23	Override data cache tag clock gater. 0: Enable data cache tag clock gating. 1: Override data cache tag clock gating. Enable the clock to data cache tag array always.	R/W	0
ODDG	22	Override data cache data clock gater. 0: Enable data cache data clock gating. 1: Override data cache data clock gating. Enable the clock to data cache data array always.	R/W	0
DLSB	21	Disable load/store bonding. 0: Enable load/store bonding. 1: Disable load/store bonding.	R/W	0
DFIS	20	Disable ‘cracking’. 0: Enable cracking. 1: Disable cracking.	R/W	0
HITLB	19	Half size instruction TLB (ITLB). When this bit is set, the ITLB becomes half of its current size. 0: Full size ITLB. 1: Half size ITLB.	R/W	0
HDTLB	18	Half size data TLB (DTLB). When this bit is set, the DTLB becomes half of its current size. 0: Full size DTLB. 1: Half size DTLB.	R/W	0

Table 2.9 Field Descriptions for Config6 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
FTLBP	17:16	FTLB probability. On a TLBWR instruction, if the *PageMask* register matches the FTLB page size, the write would be done to the FTLB. Otherwise it would go to the FTLB. However, for systems that use only a single page size, the FTLB would be used and most of the FTLB would be unused. This field allows some TLBWR instruction to go to the VTLB instead of the FTLB whenever the *PageMask* register matches the FTLB page size. If the contents of the *PageMask* register do not match the FTLB page size, the TLBWR instruction goes to the VTLB. 0: FTLB only. All TLBWR instructions go to the FTLB. 1: FTLB:VTLB = 15:1. For every 16 TLBWR instructions, 15 go to the FTLB and 1 goes to the VTLB. 2: FTLB:VTLB = 7:1. For every 8 TLBWR instructions, 7 go to the FTLB and 1 goes to the VTLB. 3: FTLB:VTLB = 3:1. For every 4 TLBWR instructions, 3 go to the FTLB and 1 goes to the VTLB.	R/W	0
FTLBEn	15	FTLB enable. Setting this bit indicates that the FTLB is enabled. 0: FTLB is disabled. 1: FTLB is enabled.	R/W	0
SPCD	14	Sleep state performance counter disable. When this bit is set, the performance counter P6600 clocks are prevented from shutting down. The primary use of this bit is to keep performance counters alive when the P6600 core is in sleep mode. 0: Performance counters are enabled in sleep mode. 1: Performance counters are disabled in sleep mode.	R/W	0
0	13	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.9 Field Descriptions for Config6 Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>IFUPerfCtl</td><td>12:10</td><td>IFU Performance Control. This field encodes IFU events that provide debug and performance information for the IFU pipeline and is encoded as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>000</td><td>IDU is accepting instructions, but IFU is not providing any.</td></tr>
<tr><td>001</td><td>A control transfer instruction such as a branch or jump causes lost IDU bandwidth.</td></tr>
<tr><td>010</td><td>A stalled instruction such as an unpredicted jump must wait for an address and thus causes lost IDU bandwidth.</td></tr>
<tr><td>011</td><td>Cache prediction was correct.</td></tr>
<tr><td>100</td><td>Cache prediction was incorrect.</td></tr>
<tr><td>101</td><td>Cache did not predict due to invalid JR cache entry, or the instruction tag miscompared with tag in JR cache.</td></tr>
<tr><td>110</td><td>Unimplemented.</td></tr>
<tr><td>111</td><td>Condition branch was taken.</td></tr>
</table>
Lost IDU bandwidth occurs when the IDU is accepting instructions, but instructions are not being provided by the IFU. The count of these events can be seen via Performance Counters 0 or 3, and the event number 11. In order to view the IFU Perf Ctl events, the Performance Counter Control needs to be programmed accordingly See Table 2.64, "Performance Counter Events and Codes" for general information on event number 11.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>9:1</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>JRCD</td><td>0</td><td>Jump register cache prediction disable. Setting this bit disables the Jump Register (JR) target address prediction.
0: JR cache target address prediction is enabled.
1: JR cache target address prediction is not enabled.</td><td>R/W</td><td>0</td></tr>
</table>

2.2.1.8 Device Configuration 7 — Config7 (CP0 Register 16, Select 7)

This register controls machine-specific features of the P6600 core. A few of them are for hardware interface adaptation, but most are for chip or system test only. They default to a "safe" value. Most software, including bootstrap software, can and should ignore these registers unless specifically advised to use them. Note that in the P6600 Multiprocessing System, this register is implemented only by the root context and not by the guest context.

Figure 2.8 Config7 Register Format

31	30	29	28	27	26 25 24	23	22	21	20 19	18	17	16
WII	FPFS	IHB	0	SEHB	0	DGHR	SG	SUI	0	HCI	0	AR

15 13	12 11	10	9	8	7	6	5	4	3	2	1	0
0	PREF	IAR	IVAD	ES	0	CP1IO	0	ULB	BP	RPS	BHT	SL

Table 2.10 Field Descriptions for Config7 Register

Name	Bit(s)	Description	Read/ Write	Reset State
WII	31	Wait IE Ignore. When this bit is set, an interrupt will unblock a **`wait`** instruction, even if $Status_{IE}$ is preventing the interrupt from being taken. If *WII* reads 0, the P6600 core remains in the wait condition forever if entered with interrupts disabled. If set to 1, it allows operating system code to avoid complex race conditions.	R	1
FPFS	30	Fast prepare for store. When this bit is set, **`pref 31`** will behave as specified, i.e., the prefetch instruction will only validate the data tag but not write 0's into the data cache. By default, this bit will be 0 and **`pref 31`** will behave like **`pref 30`**. This means that **`pref 31`** will validate the data tag and write 0's into the data cache array for the specified line.	R/W	0
IHB	29	Implicit hazard barrier. If *IHB* = 1, the following behavior will be true: • When the P6600 sees any explicit/implicit **`mtc0`**(**`cache`**, **`ll, mtc0`**, **`tlbop`**, **`eret`**, **`deret`**, **`sync`**`-in-debug-mode`, **`di`**, **`ei`**) followed by any implicit **`mfc0`** (**`ehb, mfc0, eret, deret`**, **`di`**, **`ei`**), the pipeline will behave as if an **`ehb`** is introduced implicitly prior to executing the **`mfc0`**. This ensures all state modification by **`mtc0`** is completely seen by **`mfc0`**. • Any **`jalr r31`**, **`jr r31`** instruction seen by the CPU when CP0 is usable (i.e CU0=1 or Kernel or Debug mode as defined in the PRA) will automagically treat those instructions as **`jalr.hb`** and **`jr.hb`**. If *IHB* = 0, the following behavior will be true: • Programmer is responsible for resolving hazards and put **`ehb`** or .**`hb`** where appropriate. Prior cores may have used some number of **`nops`** or **`ssnops`** to ensure that the effect of a CP0 modifying instruction is seen by a CP0 read instruction, but the P6600 core cannot guarantee such behavior with a small number of **`nops/ssnops`**. Per Release3, the programmer is expected to put in an explicit **`ehb`** or **`.hb`** where needed. If there is reason to believe that the programmer has not done this, then this bit can be enabled to get correct operation.	R/W	0
0	28	Reserved. Write as zero. Ignored on reads.	R	0
SEHB	27	Slow EHB. An experimental mode to accelerate CP0 sequences using the **`ehb`** instruction. If this bit is set, **`ehb`** will block issue of instructions from the instruction buffer until all older instructions have graduated and the pipe is empty. By default, **`ehb`** will block issue of instructions from the instruction buffer only if there are pending explicit CP0-modifying instructions in the pipe.	R/W	0

Table 2.10 Field Descriptions for Config7 Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>0</td><td>26:24</td><td>Reserved for future use.</td><td>R/W</td><td>0</td></tr>
<tr><td>DGHR</td><td>23</td><td>Disables the use of any global history in the branch predictor.</td><td>R/W</td><td>0</td></tr>
<tr><td>SG</td><td>22</td><td>Set 1 to allow only one instruction to graduate per cycle. This has a negative impact on performance and should only be used for test purposes.</td><td>R/W</td><td>0</td></tr>
<tr><td>SUI</td><td>21</td><td>Strict Uncached Instruction (SUI) policy control.
When this bit is set, hardware runs uncached instructions strictly in order and (as far as possible) unpipelined. This will cause a significant performance degradation as it will introduce a bubble equivalent to the depth of the pipeline between each instruction. Only the branch-delay-slot instruction of a branch is fetched without this bubble.
The advantage is that the CPU will not wander off speculatively fetching unwanted instructions from a (perhaps slow) boot memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>20:19</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>HCI</td><td>18</td><td>Hardware Cache Initialization: Indicates that a cache does not require initialization by software. This bit will most likely only be set on simulation-only cache models and not on real hardware.</td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>17</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>AR</td><td>16</td><td>Alias removed. Hardware sets this bit to indicate that the L1 data cache is configured to avoid cache aliases.</td><td>R</td><td>1</td></tr>
<tr><td>0</td><td>15:13</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>PREF</td><td>12:11</td><td>These two bits control the extent of prefetching of instructions into the instruction cache as indicated. This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>Prefetch 0 cache lines on an I-cache miss in addition to fetching the missing cache line. i.e. Disable I-cache prefetching.</td></tr>
<tr><td>01</td><td>Prefetch 1 cache line (sequential next line) on an I-cache miss in addition to fetching the missing cache line.</td></tr>
<tr><td>10</td><td>Reserved.</td></tr>
<tr><td>11</td><td>Prefetch 2 cache lines (sequential next 2 lines) on an I-cache miss in addition to fetching the missing cache line.</td></tr>
</table></td><td>R/W</td><td>01</td></tr>
<tr><td>IAR</td><td>10</td><td>Instruction Alias Removed.
Indicates that the P6600 core has hardware support to remove instruction cache aliasing. The virtual aliasing hardware can be disabled via the IVAD bit described below. The instruction cache virtual aliasing hardware is always present in the P6600 core.</td><td>R</td><td>1</td></tr>
</table>

Table 2.10 Field Descriptions for Config7 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
IVAD	9	Instruction Virtual Aliasing disabled. The hardware required to resolve instruction cache virtual aliasing is always present in the P6600 core as noted by the default state of the *IAR* bit shown above. However, software can toggle the *IVAD* bit to enable or disable the virtual aliasing hardware for the instruction cache. Setting this bit disables the hardware alias removal on the instruction cache. If this bit is cleared, the **`CACHE Hit Invalidate`** and **`SYNCI`** instructions look up all possible aliased locations and invalidate the given cache line in all of them. This bit is Read-only if *IAR* = 0 and can only be written when *IAR* = 1.	R/W	0
ES	8	Externalize **`sync`**. If this bit is set, and if the downstream device (toward memory) is capable of accepting SYNCs (indicated by the pin *SI_SyncTxEn*), the **`sync`** instruction causes a SYNC-specific transaction to go out on the external bus. If this bit is cleared or if *SI_SyncTxEn* is deasserted, no transaction will go out, but all SYNC handling internal to the CPU will nevertheless be performed. The **`sync`** instruction is signalled on the P6600's OCP interface as an "ordering barrier" transaction. The transaction is an extension to the OCP standards, and system controllers which don't support it typically under-decode it as a read from the boot ROM area. But that's going to be quite slow, so set this bit only if your system understands the synchronizing transaction. When this bit is read, the value returned depends on the state of the *SI_SyncTxEn* pin. If *SI_SyncTxEn* is 0, a value of 0 is returned. If *SI_SyncTxEn* is 1, the value returned is the last value that was written to this bit.	R	1
0	7	Reserved. Write as zero. Ignored on reads.	R	0
CP1IO	6	CP1 instruction order. By default, data sent from the P6600 core to a coprocessor block may be sent in an order reflecting the internal pipeline execution sequence. Set this bit to arrange that data will be sent only in instruction order to the FPU.	R/W	0
0	5	Reserved. Write as zero. Ignored on reads.	R	0
ULB	4	Uncached load blocking. Set to 1 to make all uncached loads blocking (a program usually only blocks when it uses the data which is loaded).	R/W	0
BP	3	Branch prediction. When set, no branch prediction is done, and all branches stall.	R/W	0
RPS	2	Return prediction stack. When set, the return address branch predictor is disabled, so **`jr $31`** is treated just like any other jump register. An instruction fetch stalls after the branch delay slot, until the jump instruction reaches the Address Generation pipeline and can provide the right address.	R/W	0
BHT	1	Branch history table. When set, the branch history table is disabled and all branches are predicted taken. This bit is don't care if $Config7_{BP}$ is set.	R/W	0
SL	0	Scheduled loads. When set, non-blocking loads are disabled. Normally the P6600 core continues after a load instruction, even if it misses in the D-cache, until the data is used. When this bit is set, the CPU stalls on any D-cache load miss.	R/W	0

2.2.1.9 Processor ID — PRId (CP0 Register 15, Select 0)

The Processor Identification (*PRId*) register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturing options, processor identification, and revision level of the processor.

Figure 2.9 PRId Register Format

31 24	23 16	15 8	7 0
CoOpt	CoID	ProcType	Rev

Table 2.11 Field Descriptions for PRId Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>CoOpt</td><td>31:24</td><td>Company Option. Should be a number between 0 and 127— higher values are reserved by MIPS Technologies.</td><td>R</td><td>Preset</td></tr>
<tr><td>CoID</td><td>23:16</td><td>Company ID. Identifies the company that designed or manufactured the processor. In the P6600, this field contains a value of 1 to indicate MIPS Technologies, Inc.</td><td>R</td><td>0x01</td></tr>
<tr><td>ProcType</td><td>15:8</td><td>Processor ID. Identifies the type of processor. This field allows software to distinguish between the various types of processors from MIPS Technologies. The value of this field is 0xA4 for the P6600 core.</td><td>R</td><td>0xA4</td></tr>
<tr><td>Rev</td><td>7:0</td><td>The revision number of the P6600 design. This field allows software to distinguish between one revision and another of the same processor type.
This field is broken up into the following three subfields:
<table>
<tr><th>Bit(s)</th><th>Name</th><th>Meaning</th></tr>
<tr><td>7:5</td><td>Major Revision</td><td>This number is increased on major revisions of the P6600 core.</td></tr>
<tr><td>4:2</td><td>Minor Revision</td><td>This number is increased on each incremental revision of the processor and reset on each new major revision.</td></tr>
<tr><td>1:0</td><td>Patch Level</td><td>If a patch is made to modify an older revision of the processor, this field will be incremented.</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
</table>

2.2.1.10 Exception Base Address — EBase (CP0 Register 15, Select 1)

The 64-bit *EBase* register is a read/write register containing the base address of the exception vectors used when *StatusBEV* equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different. Bits 63:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from the fixed defaults when $Status_{BEV}$ is 1, or for any EJTAG Debug exception.

The size of the ExcBase field depends on the state of the WG bit. At reset, the WG bit is cleared by default. In this case, the ExcBase field is comprised of bits 29:12. Bits 63:30 of the EBase Register are not writeable and retain their previous state. This is shown in Figure 2.10.

When the WG bit is set, bits 63:30 of the ExcBase field become writeable and are used to relocate the ExcBase field to other segments. This is shown in Figure 2.11. Note that if the WG bit is set by software (allowing bits 63:30 to become part of the ExcBase field) and then cleared, bits 63:30 can no longer be written by software and the state of these bits remains unchanged for any writes after WG was cleared.

If the value of the exception base register is to be changed, this must be done with *StatusBEV* equal to 1. The operation of the processor is **UNDEFINED** if the exception base field is written with a different value when *StatusBEV* is 0.

Combining bits 63:12 with the *Exception Base* field allows the base address of the exception vectors to be placed at any 16 Kbyte page boundary.

Figure 2.10 EBase Register Format — WG = 0

63 30	29 12	11	10	9 0
Fill (not writable)	ExcBase	WG	0	CPUNum

Figure 2.11 EBase Register Format — WG = 1

63 12	11	10	9 0
ExcBase	WG	0	CPUNum

Table 2.12 Field Descriptions for EBase Register

Name	Bit(s)	Description	Read/ Write	Reset State
Fill	63:30	When the WG bit is cleared, this field is not writable by software and retains its previous value.	R/W	Undefined
ExcBase	29:12 or 63:12	Exception Base Address. The size and behavior of this field depends on the state of the WG bit. When the WG bit is set, the ExcBase field includes bits 63:12. When the WG bit is cleared, bits 63:30 are not writable and the exception base address is stored in bits 29:12. Bits 31:30 default to a value of 2'b10, forcing the exception vector into kseg0/kseg1 address space to maintain 32-bit backward compatibility. Setting *EBase* in any CPU to a unique value allows that CPU can have its own unique exception handlers. This field should be written only when $Status_{BEV}$ is set so that any exception will be handled through the ROM entry points.	R/W	0x8000.0 or 0xF.FFFF. FFF8.0000
WG	11	Write gate. When the WG bit is set, the ExcBase field is expanded to include bits 31:30 of the *EBase* register to facilitate programmable memory segmentation controlled by the *SegCtl0* through *SegCtl2* registers. When the WG bit is cleared, bits 31:30 of the *EBase* register are not writeable and remain unchanged from the last time that WG was cleared.	R/W	Externally Set
0	10	Reserved. Write as zero. Ignored on reads.	R	0
CPUNum	9:0	This field contains an identifier that will be unique among the CPU's in a multi-processor system. The value in this field is set by the *SI_CPUNum[9:0]* static input pins to the P6600 core.	R	Externally Set

2.2.1.11 Status (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor. Fields in this register and the CP0 Debug register combine to create operating modes for the processor. Selected bits are encoded as follows to place the processor into one of the operating modes. Refer to the MMU chapter for more information on the various operating modes. A brief summary is provided below.

Table 2.13 Operating Mode Encoding

$Status_{IE}$	$Status_{ERL}$	$Status_{EXL}$	$Status_{KSU}$	$Debug_{DM}$	Mode of Operation
1	0	0	x	0	Individual interrupts can be disabled/enabled using the $Status_{IM7-0}$ mask bits.
x	0	0	2'b2	0	*User Mode*. In user mode, the CPU has access only to the mapped kuseg address region.
x	0	0	2'b1	0	*Supervisor Mode*. In supervisor mode, the CPU has access to the top half of the kseg2 region (sometimes known as kseg3), but no access to CP0 registers or most kernel memory.
x	x	x	2'b0	0	*Kernel addressing mode*. In this mode, a TLB miss goes to the TLB Refill Handler.
x	x	1	x	0	*Kernel addressing mode*. In this mode, a TLB miss goes to the TLB Refill Handler.
x	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.
x	x	x	x	1	*Debug Mode*. In debug mode, the processor has full access to all resources that are available in Kernel Mode operation, in addition to those provided by EJTAG.

Figure 2.12 shows the format of the *Status* Register; Table 2.14 describes the *Status* register fields.

Figure 2.12 Status Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 8	7		5	4 3	2	1	0
CU3	CU2	CU1	RW	0	FR	RE	MX	PX	BEV	0	SR	NMI	0	CEE	0	IM7-0	KX	SX	UX	KSU	ERL	EXL	IE

Table 2.14 Field Descriptions for Status Register

Name	Bit(s)	Description	Read/ Write	Reset State
CU3	31	Coprocessor 3 usable. Because the P6600 core does not support a coprocessor 3, $Status_{CU3}$ is hardwired to zero.	R	0
CU2	30	Coprocessor 2 usable. Because the P6600 core does not support a coprocessor 2, $Status_{CU2}$ is hardwired to zero.	R	0
CU1	29	Coprocessor 1 Usable. Controls access to coprocessor 1. 0: Access not allowed. 1: Access allowed. *CU1* is most often used for a floating-point unit. When no coprocessor 1 is present, this bit is read-only and reads zero.	R/W	Undefined
RW	28	Read/write field. This bit can be written by software without side-effects. A use case is for the kernel to set this bit to signify that the exception condition is due to user code, prior to saving Status to the stack in memory. This bit is not used by the P6600 core hardware.	R/W	Undefined
0	27	Reserved. Write as zero. Ignored on reads.	R	0
FR	26	Floating Register. This bit is used to indicate the floating-point register mode for 64-bit floating point units: This bit is encoded as follows: 0: Floating point registers can contain any 32-bit data type. 64-bit data types are stored in even-odd pairs of registers. 1: Floating point registers can contain any data type. If the P6600 core is equipped with an optional FPU, set this bit to 0 for MIPS I compatibility mode, which allows for 16 real FP registers, with 16 odd FP register numbers reserved for access to the high-order bits of double-precision values.	R	1
0	25	Reserved. Write as zero. Ignored on reads.	R	0
MX	24	MIPS DSP Extension. Enables access to DSP ASE resources. This bit is always 0 in the P6600 core. 0: Access not allowed. 1: Access allowed. An attempt to execute any DSP ASE instruction before when this bit is 0 will cause a *DSP State Disabled* exception. The state of this bit is reflected in $Config3_{DSPP}$.	R	0
PX	23	Enables access to 64-bit operations in User mode, without enabling 64-bit addressing. 0: Access not allowed 1: Access allowed	R	0
BEV	22	Boot Exception Vector. Controls the location of exception vectors: 0: Normal. Refer to the EBase register for more information 1: Bootstrap When set to 1, all exception entry points are relocated to near the reset start address.	R/W	1
0	21	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.14 Field Descriptions for Status Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
SR	20	Soft Reset. The P6600 core only supports a full external reset, so this bit is not used and always reads zero.	R	0
NMI	19	Indicates that the entry through the reset exception vector was due to an NMI. 0: Not NMI (reset) 1: NMI Software can only write a 0 to this bit to clear it and cannot force a 0 to 1 transition. As such, a write of 1 to this bit is ignored.	R/W0	1 for NMI 0 otherwise
0	18:16	Reserved. Write as zero. Ignored on reads.	R	0
IM7-0	15:8	Interrupt Mask. Bitwise interrupt enables for the eight interrupt conditions. The state of these bits is visible in $Cause_{IP7-0}$, except in EIC mode. External Interrupt Controller (EIC) mode is activated when the $Config3_{VEIC}$ is set by hardware at reset based on the state of the *SI_EICPresent* signal. If this bit is set by hardware, software should set the $Cause_{IV}$ bit, then write a non-zero "vector spacing" in the VS bit of the *IntCtl* register. In EIC mode, *IM7-2* is used as a 6-bit $Status_{IPL}$ (Interrupt Priority Level) field. An interrupt is only triggered when the interrupt controller presents an interrupt code which is numerically higher than the current value of $Status_{IPL}$. $Status_{IM1-0}$ always acts as a bitwise mask for the two software interrupt bits programmable in $CauseI_{P1-0}$.	R/W	Undefined
KX	7	Setting this bit enables the following: • Access to 64-bit Kernel Segments • Use of the XTLB Refill Vector for references to Kernel Segments This bit is encoded as follows: 0: Access to 64-bit Kernel Segments is disabled; the TLB Refill Vector is used for references to Kernel Segments. 1: Access to 64-bit Kernel Segments is enabled; the XTLB Refill Vector is used for references to Kernel Segments.	R/W	0
SX	6	Setting this bit enables the following: • Access to 64-bit Supervisor Segments • Use of the XTLB Refill Vector for references to Supervisor Segments This bit is encoded as follows: 0: Access to 64-bit Supervisor Segments is disabled; the TLB Refill Vector is used for references to Supervisor Segments. 1: Access to 64-bit Supervisor Segments is enabled; the XTLB Refill Vector is used for references to Supervisor Segments. In the P6600 core, a write of 1 to this register is ignored when KX = 0.	R/W	0
UX	5	Setting this bit enables the following: • Access to 64-bit User Segments • Use of the XTLB Refill Vector for references to User Segments This bit is encoded as follows: 0: Access to 64-bit User Segments is disabled; the TLB Refill Vector is used for references to User Segments. 1: Access to 64-bit User Segments is enabled; the XTLB Refill Vector is used for references to User Segments. In the P6600 core, a write of 1 to this register is ignored when KX = 0 or SX = 0.	R/W	0

Table 2.14 Field Descriptions for Status Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
KSU	4:3	These bits denote the processor's operating mode. 2'b00: Kernel Mode 2'b01: Supervisor Mode 2'b10: User Mode. 2'b11: Reserved A value of 2'b11 in this field is an illegal value that will drop the entire write operation. Note that the processor can also be in Kernel mode if *ERL* or *EXL* is set, regardless of the state of these bits.	R/W	2'b00
ERL	2	Error Level; Set by the processor when a Reset, NMI, or Cache Error exception is taken. 0: Normal level 1: Error level When *ERL* is set: • The processor is running in kernel mode • Interrupts are disabled • The ERET instruction will use the return address held in *ErrorEPC* instead of *EPC* • When ERL = 1 in the Status register, the segment kuseg (legacy) or xkseg0 (EVA) is treated as an unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address, and does not include the ASID field.	R/W	1
EXL	1	Exception Level; Set by the processor when any exception other than Reset, Cache Error, or NMI exception is taken. 0: Normal level 1: Exception level When EXL is set: • The processor is running in Kernel Mode. • Hardware and software interrupts are disabled. • TLB Refill exceptions use the general exception vector instead of the TLB Refill vector. When an exception occurs and *EXL* is set, a nested TLB Refill exception is sent to the general exception handler (rather than to its dedicated handler) and the values in *EPC* and $Cause_{BD}$ are not overwritten. The result is that, after returning from the second exception, the processor jumps back to the code that was executing before the first exception occurred.	R/W	0
IE	0	Interrupt Enable. Acts as the master enable for software and hardware interrupts. 0: Interrupts are disabled 1: Interrupts are enabled This bit can be written using the `di`/`ei` instructions.	R/W	0

2.2.1.12 Interrupt Control — IntCtl (CP0 Register 12, Select 1)

The *IntCtl* register controls the interrupt capabilities of the *P6600* core, including vectored interrupts and support for an external interrupt controller.

Figure 2.13 IntCtl Register Format

31 29	28 26	25 23	22 10	9 5	4 0
IPTI	IPPCI	IPFDCI	0	VS	0

Table 2.15 Field Descriptions for IntCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
IPTI	31:29	For *Interrupt Compatibility* and *Vectored Interrupt* modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider $Cause_{TI}$ for a potential interrupt. This field is encoded as shown in Table 2.16, "Encoding of IPTI, IPPCI, and IPFDCI Fields". The value of this bit is set by the static input, *SI_IPTI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_TimerInt* signal is attached. The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.	R	Externally Set
IPPCI	28:26	For *Interrupt Compatibility* and *Vectored Interrupt* modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider $Cause_{PCI}$ for a potential interrupt. This field is encoded as shown in Table 2.16, "Encoding of IPTI, IPPCI, and IPFDCI Fields". The value of this bit is set by the static input *SI_IPPCI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_PCInt* signal is attached. The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.	R	Externally Set
IPFDCI	25:23	For *Interrupt Compatibility* and *Vectored Interrupt* modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider $Cause_{FDCI}$ for a potential interrupt. This field is encoded as shown in Table 2.16, "Encoding of IPTI, IPPCI, and IPFDCI Fields". The value of this bit is set by the static input, *SI_IPFDCI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_FDCInt* signal is attached. The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.	R	Externally Set
0	22:10	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.15 Field Descriptions for IntCtl Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>VS</td><td>9:5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by $Config3_{VInt}$ or $Config3_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th>VS Field Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr>
<tr><td>0x00</td><td>0x000</td><td>0</td></tr>
<tr><td>0x01</td><td>0x020</td><td>32</td></tr>
<tr><td>0x02</td><td>0x040</td><td>64</td></tr>
<tr><td>0x04</td><td>0x080</td><td>128</td></tr>
<tr><td>0x08</td><td>0x100</td><td>256</td></tr>
<tr><td>0x10</td><td>0x200</td><td>512</td></tr>
</table>
All other values are reserved. The operation of the processor is UNDEFINED if a reserved value is written to this field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>4:0</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
</table>

Table 2.16 Encoding of IPTI, IPPCI, and IPFDCI Fields

Encoding	IP bit	Hardware Interrupt Source
0	0	Reserved
1	1	Reserved
2	2	HW0
3	3	HW1
4	4	HW2
5	5	HW3
6	6	HW4
7	7	HW5

2.2.2 TLB Management Registers

This section contains the following TLB management registers.

2.2.2.1 Index (CP0 Register 0, Select 0)

Index is used as the TLB index when reading or writing the TLB with **TLBR/TLBWI/TLBINV/TLBINVF** respectively. It is also set by a TLB probe (**TLBP**) instruction to return the location of an address match in the TLB.

During execution of a **TLBR** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be read. Hardware then uses this information to perform the read operation.

During execution of a **TLBWI**, **TLBINV**, or **TLBINVF** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be written or invalidated. Hardware then uses this information to perform the respective write or invalidate operation.

Prior to executing a **TLBP** instruction, the VPN to be searched should have been written to the VPN2 field in the *EntryHi* register. During the **TLBP** instruction, hardware searches the TLB array for a match to the VPN stored in the *EntryHi* register. If a match is found, hardware writes the index into the *Index* field of this register.

The *P* bit of this register is set by hardware to indicate that a match was not found. If this bit is not set, software can then read the corresponding index from this register.

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the *Index* register.

Note that when virtualization is enabled, there is one *Index* register for Root and one for each Guest.

Figure 2.14 Index Register Format

31	30 ... 10	9 ... 0
P	0	Index (VTLB and FTLB)

Table 2.17 Field Descriptions for Index Register

Name	Bit(s)	Description	Read/ Write	Reset State
P	31	Probe Failure. This bit is automatically set when a **`TLBP`** search of the TLB fails to find a matching entry. The following rules apply when accessing this bit: 1. Root can only set Root.Index.P value to 1 (and not clear it) using the MTC0 instruction. 2, Guest can only set Guest.Index.P value to 1 (and not clear it) using the MTC0 instruction. 3. Root can both set and clear Guest.Index.P value using the MTGC0 instruction.	WO or R/W (See descr)	0
0	30:10	Must be written as zero; returns zero on reads.	0	0
Index	9:0	An index into the TLB used for **`TLBR, TLBWI, TLBINV`** and **`TLBINVF`** instructions. This field is set by the **`TLBP`** instruction when it finds a matching entry.	R/W	0

2.2.2.2 EntryLo0 - EntryLo1 (CP0 Registers 2 and 3, Select 0)

The pair of *EntryLo* registers act as the interface between the TLB and the **`TLBP, TLBR, TLBWI,`** and **`TLBWR`** instructions. These registers store the contents of a TLB entry. Each entry maps a pair of pages. The *EntryLo0* and *EntryLo1* register store even and odd numbered virtual pages respectively. These registers are read during a **`TLBWR`** or **`TBLW`**I instruction, and written by a **`TLBR`** instruction. They are not used for any other purpose.

Software may determine the value of PABITS by writing all ones to the EntryLo0 or EntryLo1 registers and reading the value back. Bits read as "1" from the PFN field allow software to determine the boundary between the PFN and Fill fields to calculate the value of PABITS.

The contents of the EntryLo0 and EntryLo1 registers are not defined after an address error exception and some fields may be modified by hardware during the address error exception sequence. Software writes of the EntryHi register (via MTC0) do not cause the implicit update of address-related fields in the BadVAddr or Context registers.

Figure 2.15 EntryLo0 and EntryLo1 Register Format

63	62	61 … 34	33 … 32
RI	XI	0	PFNX

31 … 30	29 … 6	5 … 3	2	1	0
PFNX	PFN	C	D	V	G

Table 2.18 Field Descriptions for EntryLo0 and EntryLo1 Registers

Name	Bit(s)	Description	Read/ Write	Reset State
RI	63	Read Inhibit. If this bit is set in a TLB entry, any attempt to read data on the virtual page causes either a TLB Invalid or a TLBRI exception, even if the V (Valid) bit is set. The RI bit is writable only if the RIE bit of the *PageGrain* register is set. For more information, refer to Section 2.2.2.9, "Page Granularity — PageGrain (CP0 Register 5, Select 1)". If the RIE bit of the *PageGrain* register is not set, the RI bit of *Entry 0* and *Entry 1* are set to zero on any write to the register, regardless of the value written.	R/W	0
XI	62	Execute Inhibit. If this bit is set in a TLB entry, any attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes a TLB Invalid or a TLBXI exception, even if the V (Valid) bit is set. The XI bit is writable only if the XIE bit of the *PageGrain* register is set. For more information, refer to Section 2.2.2.9, "Page Granularity — PageGrain (CP0 Register 5, Select 1)". If the XIE bit of the *PageGrain* register not set, the XI bit of TLB Entry 0 - 1 is set to zero on any write to the register, regardless of the value written.	R/W	0
Fill	61:34	These bits are ignored on writes and return 0 on reads.	R/W	0
PFNX	33:30	Page Frame Number Extension. This field is used to extend the size of the PFN. This field is concatenated with the PFN field to form the full page frame number corresponding to the physical address, thereby providing up to 40 bits of physical address. If the processor is not enabled to support 1KB pages (Config3SP = 0 or PageGrainESP = 0), the combined PFNX || PFN fields corresponds to 0b00 || bits PABITS-1..12 of the physical address (the field is unshifted and the upper two bits must be written as zero). The boundaries of this field change as a function of the value of PABITS. If support for large physical addresses is enabled (Config3.LPA = 1 or PageGrain.ELPA = 1), this field is R/W and can be written by software. If support for large physical addresses is not enabled (Config3.LPA = 0 or PageGrain.ELPA = 0), this field is read-only. In that case, the PFNX bits are ignored on write and return 0 on read.	R/W or R	Undefined
PFN	29:6	The 24 bits of *PFN*, together with the 4-bit PFNX field and 12 bits of in-page address, make up a 40-bit physical address. The PFNX field in bits 33:30 of this register is appended to the upper bits of the PFN to create the extended address.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 2.19.	R/W	Undefined
D	2	The "Dirty" flag. 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 zero, 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.	R/W	Undefined

Table 2.18 Field Descriptions for EntryLo0 and EntryLo1 Registers

Name	Bit(s)	Description	Read/ Write	Reset State
V	1	The “Valid” flag. Indicates that the TLB entry, and thus the virtual page mapping, are valid. If this bit is a set, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a *TLB Invalid* exception. This bit can be used to make just one of a pair of pages valid.	R/W	Undefined
G	0	The “Global” bit. On a TLB write, the logical AND of the G bits in both the *Entry 0* and *Entry 1* registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both *Entry 0* and *Entry 1* reflect the state of the TLB G bit.	R/W	Undefined

Table 2.19 Cache Coherency Attributes Encoding of the C Field

C[5:3] / K0[2:0][1]	Name	Cache Coherency Attribute
0	—	Reserved
1	—	Reserved
2	UC	Uncached, non-coherent
3	WB	Cacheable, non-coherent, write-back, write allocate
4	CWBE	Cacheable, coherent, write-back, write-allocate, read misses request Exclusive
5	CWB	Cacheable, coherent, write-back, write-allocate, read misses request Shared
6	—	Reserved
7	UCA	Uncached Accelerated, non-coherent

1. State of the K0 field at bits 2:0 of the Config register. See Section 2.2.1.1 “Device Configuration — Config (CP0 Register 16, Select 0)”

2.2.2.3 EntryHi (CP0 Register 10, Select 0)

The *EntryHi* register contains the upper portion of the virtual address match information used for TLB read, write, and access operations. The remaining information is stored in the *EntryLo0* and *EntryLo1* registers described in Section 2.2.2.2 “EntryLo0 - EntryLo1 (CP0 Registers 2 and 3, Select 0)”.

A TLB exception (TLB Refill, XTLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, or TLB Modified) causes bits $VA_{47:13}$ of the virtual address to be written into the *VPN2* field of the *EntryHi* register and VA[63:62] to be written to the Region (*R)* field of the *EntryHi* register. A TLBR instruction writes the *EntryHi* register with the corresponding fields from the selected TLB entry. The *ASID* field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match.

Because the *ASID* and *EHINV* fields are overwritten by a TLBR instruction, software must save and restore the value of *ASID* around use of the TLBR. This is especially important in TLB Invalid and TLB Modified exceptions, and in other memory management software.

The *VPN2* field of the *EntryHi* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence. Software writes of the *EntryHi* register (via MTC0) do not cause the implicit write of address-related fields in the *BadVAddr* and *Context* registers.

The $EntryHi_{EHINV}$ field has been added to support explicit invalidation of TLB entries via the **`TLBWI`** instruction. When $EntryHi_{EHINV}$ = 1, the **`TLBWI`** instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with a TLB entry to the invalid state. When $EntryHi_{EHINV}$ = 1, only the *Index* register is required to be valid. Behavior of the TLBWR instruction is unmodified by $EntryHi_{EHINV}$. The **`TLBR`** instruction copies the EHINV bit from the TLB Entry to $EntryHI_{EHINV}$. Note that execution of the **`TLBP`** instruction does not change this value.

Figure 2.16 EntryHi Register Format

63 62	61 … 48	47 … 32
R	0	VPN2

31 … 13	12 11	10	9 8	7 … 0
VPN2	0	EHINV	0	ASID

Table 2.20 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
R	63:62	Virtual memory region, corresponding to VA[63:62]. This field is encoded as follows: 00: xuseg: user address region 01: xsseg: supervisor address region. If Supervisor Mode is not implemented, this encoding is reserved 10: Reserved 11: xkseg: kernel address region This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	0
0	61:48	Reserved. Write as zero. Ignored on reads.	R	0
VPN2	47:13	$EntryHi_{VPN2}$ is the virtual address to be matched on a **`TLBP`**. This field consists of $VA_{39:13}$ of the virtual address (virtual page number / 2). It is also the virtual address to be written into the TLB on a **`TLBWI`** and **`TLBWR,`** and the destination of the virtual address on a **`TLBR`**. On a TLB-related exception, the VPN2 field is automatically set to the virtual address that was being translated when the exception occurred. This field is written by software before a **`TLBP`** or **`TLBWI`** and written by hardware in all other cases.	R/W	Undefined
0	12:11	Reserved. Write as zero. Ignored on reads.	R	0
EHINV	10	TLBWI invalidate enable. When this bit is set, the TLBWI instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. When this bit is set, the *PageMask* and *EntryLo0/EntryLo1* registers do not needto be valid. Only the *Index* register is required to be valid. This bit is ignored on a **`TLBWR`** instruction.	R/W	0
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.20 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
ASID	7:0	Address space identifier. This field is used to stage data to and from the TLB, but in normal running software it's also the source of the current "ASID" value, used to extend the virtual address and help to map address translations for the current process. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field. This field supports up to 256 unique ASID values, consisting of a virtual tag that is in addition to the 32-bit address.	R/W	0

2.2.2.4 Context (CP0 Register 4, Select 0)

The 64-bit *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *Context* register duplicates some of the information provided in the *BadVAddr* register but is organized in such a way that the operating system can directly reference an 8-byte page table entry (PTE) in memory.

The *BadVPN2* field of the *Context* register is not defined after an address error exception, and this field may be modified by hardware during the address error exception sequence.

The pointer implemented by the *Context* register can point to any power-of-two-sized PTE structure within memory. This allows the TLB refill handler to use the pointer without additional shifting and masking steps. For example, if the low-order bit of the PTEBase field is 20, the page table entry (PTE) structure occurs on a 1M boundary. If the low-order bit is 21, PTE structure occurs on a 2M boundary, etc. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

A TLB exception (Refill, Invalid, Modified, Read Inhibit, Execute Inhibit) causes the virtual address to be written to a variable range of bits, defined as (X-1):Y of the *Context* register. This range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 63:X, Y-1:0 are read/write to software and are unaffected by the exception.

For example, if X = 23 and Y = 4, i.e. bits 22:4 are set in *ContextConfig*, the behavior is identical to the standard MIPS32 *Context* register (bits 22:4 are filled with $VA_{31:13}$). Although the fields have been made variable in size and interpretation, the MIPS32 nomenclature is retained. Bits 63:X are referred to as the *PTEBase* field, and bits X-1:Y are referred to as *BadVPN2*.

The value of the *Context* register is **UNPREDICTABLE** following a modification of the contents of the *ContextConfig* register. After the *ContextConfig* register is modified, software should write the PTEBase field of the *Context* register. However, note that the contents of the BadVPN2 field will not be valid until the next TLB exception.

Figure 2.17 shows the format of the *Context* Register; Table 2.21 describes the *Context* register fields.

Figure 2.17 Context Register Format

63 … X	X-1 … Y	Y-1 … 0
PTEBase	BadVPN2	PTEBaseLow

Table 2.21 Context Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
PTEBase	Variable, 63:X	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception. The size of the BadVPN2 field is determined by number of contiguous 'ones' in the VirtualIndex field of the *ContextConfig* register described below. If the VirtualIndex field is all 'ones', then the BadVPN2 field is comprised of bits 22:2. If the VirtualIndex field is all 'zero', then there is no BadVPN and the PTEBase and PTEBase low fields are merged together to form a single 32-bit PTEBase value.	R/W	Undefined
BadVPN2	Variable, (X-1):Y	This field is written by hardware on a TLB exception. It contains bits $VA_{31:32-X+Y}$ of the virtual address that caused the exception.	R	Undefined
PTEBaseLow	Variable, (Y-1):0	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception.	R/W	Undefined

2.2.2.5 Context Configuration — ContextConfig (CP0 Register 4, Select 1)

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected *BadVPN2* field of the *Context* register are read/write to software and serve as the *PTEBase* field. Bits below the selected *BadVPN2* field of the *Context* register serve as the *PTEBaseLow* field.

Software writes a set of contiguous ones to the *VirtualIndex* field of the *ContextConfig* register. Hardware then determines which bits of this register are high and low. The highest order bit that is a logic '1' serves as the MSB of the *BadVPN2* field of the *Context* register. The lowest order bit that is a logic '1' serves as the LSB of the*BadVPN2* field of the *Context* register. A value of all zero's in the *VirtualIndex* field means that the full 32 bits of the *Context* register are R/W for software and are unaffected by TLB exceptions.

Figure 2.18 shows the formats of the *ContextConfig* register; Table 2.22 describes the *ContextConfig* register fields.

Figure 2.18 ContextConfig Register Format

31 ... 23	22 ... 2	1 ... 0
0	VirtualIndex	0

Table 2.22 ContextConfig Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	31:23	Ignored on write; returns zero on read.	R	0x00
VirtualIndex	22:2	A mask of 0 to 21 contiguous 1 bits in this field causes the corresponding bits of the *Context* register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is **UNDEFINED** if non-contiguous 1 bits are written into the register field. Note that it is the responsibility of software to ensure that this field is written with contiguous ones because if non-contiguous 1 bits are written, no exception will be taken.	R/W	0x1F_FFFC
0	1:0	Ignored on write; returns zero on read.	R	0

2.2.2.6 XContext Register (CP0 Register 20, Select 0)

The *XContext* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *XContext* register is primarily intended for use with the XTLB Refill handler, but is also loaded by hardware on a TLB Refill. However, it is unlikely to be useful to software in the TLB Refill Handler. The *XContext* register duplicates some of the information provided in the *BadVaddr* register. The size of the *BadVPN2* field, indicated by the X-1:Y parameter in the figure below, depends on the number of consecutive ones in the *XContextConfig* register.

Figure 2.19 shows the format of the *XContext* register.

Figure 2.19 XContext Register Format

63 ... X	X-1 ... Y	Y-1 ... 0
PTEBase	BadVPN2	PTEBaseLow

Table 2.23 XContext Register Field Descriptions when Config3.CTXTC = 1

Fields		Description	Read / Write	Reset State
Name	**Bits**			
PTEBase	63:X	This field is for use by the operating system and is normally written with a value that allows the operating system to use the XContext Register as a pointer to an array of data structures in memory corresponding to the address region containing the virtual address which caused the exception. Note that the lower 2-bits of PTEBase are always 0.	R/W	Undefined

Table 2.23 XContext Register Field Descriptions when Config3.CTXTC = 1 *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
BadVPN2	X-1:Y	This field is written by hardware on a TLB exception. It contains the virtual address that caused the exception. The upper and lower bound of this field is determined by the consecutive number of 1's in the *XContextConfig* register.	R	Undefined
PTEBaseLow	(Y-1):0	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception.	R/W	Undefined

2.2.2.7 XContext Configuration — XContextConfig (CP0 Register 4, Select 3)

The *XContextConfig* register defines the bits of the *XContext* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected *BadVPN2* field of the *Context* register are read/write to software and serve as the *PTEBase* field. Bits below the selected *BadVPN2* field of the *Context* register serve as the *PTEBaseLow* field.

Software writes a set of contiguous ones to the *VirtualIndex* field of the *XContextConfig* register. Hardware then determines which bits of this register are high and low. The highest order bit that is a logic '1' serves as the MSB of the *BadVPN2* field of the *XContext* register. The lowest order bit that is a logic '1' serves as the LSB of the *BadVPN2* field of the *XContext* register. A value of all zero's in the *VirtualIndex* field means that the full 32 bits of the *XContext* register are R/W for software and are unaffected by TLB exceptions.

Figure 2.18 shows the formats of the *XContextConfig* register; Table 2.22 describes the *XContextConfig* register fields.

Figure 2.20 XContextConfig Register Format

63 — 39	38 — 2	1 — 0
0	VirtualIndex	0

Table 2.24 XContextConfig Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	63:39	Ignored on write; returns zero on read.	R	0x00

Table 2.24 XContextConfig Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
VirtualIndex	38:2	A mask of 0 to 37 contiguous 1 bits in this field causes the corresponding bits of the *XContext* register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is **UNDEFINED** if non-contiguous 1 bits are written into the register field. Note that it is the responsibility of software to ensure that this field is written with contiguous ones because if non-contiguous 1 bits are written, no exception will be taken.	R/W	0x1F_FFFF_FFFC
0	1:0	Ignored on write; returns zero on read.	R	0

2.2.2.8 PageMask (CP0 Register 5, Select 0)

Every TLB entry has an independent virtual-address mask that allows it to ignore some address bits when deciding to match. By selectively ignoring lower page addresses, the entry can be made to match all the addresses in a "page" larger than 4KB.

Software can determine the maximum page size supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. Note that the bits are read in pairs, so bits 14:13 are read first and can have only a value of 00 or 11. If they are both 11, bits 16:15 are read, and so on.

The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 2.26, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

Figure 2.21 PageMask Register Format

63 — 33	32 — 13	12 — 0
0	Mask	0

Table 2.25 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:33	Ignored on write; returns zero on read.	R	0
Mask	32:13	The mask field is a bit mask in which a logic “1” indicates that the corresponding bit of the virtual address should not participate in the TLB match. Note that only a restricted range of *PageMask* values are legal (i.e., with "1"s filling the $PageMask_{Mask}$ field from low bits upward, two at a time). Maximum page size is 4 GB. The legal values for this field are shown in Table 2.26 below.	R/W	Undefined
0	12:0	Ignored on write; returns zero on read.	R	0

Table 2.26 PageMask Register Values

PageMask Register Value	Size of Each Output Page
0x0000_0000_0000.6000	16 Kbytes
0x0000_0000_0001.E000	64 Kbytes
0x0000_0000_0007.E000	256 Kbytes
0x0000_0000_001F.E000	1 Mbyte
0x0000_0000_007F.E000	4 Mbytes
0x0000_0000_01FF.E000	16 Mbytes
0x0000_0000_07FF.E000	64 Mbytes
0x0000_0000_1FFF.E000	256 Mbytes
0x0000_0000_7FFF.E000	1 Gbytes
0x0000_0001_FFFF.E000	4 Gbytes

Software may determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 2.26, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

2.2.2.9 Page Granularity — PageGrain (CP0 Register 5, Select 1)

The PageGrain register is a read/write register used for XI/RI TLB protection bits. Figure 2.22 shows the format of the PageGrain register. Table 2.27 describes the PageGrain register fields.

Figure 2.22 PageGrain Register Format

31	30	29	28	27	26 — 5	4 — 0
RIE	XIE	ELPA	ESP	IEC	0	MCAUSE

Table 2.27 Field Descriptions for PageGrain Register

Name	Bit(s)	Description	Read/ Write	Reset State
RIE	31	Read inhibit enable. This bit is always 1 to indicate that the RI bit of the *Entry0* and *Entry1* registers is enabled.	R	1
XIE	30	Execute inhibit enable. This bit is always 1 to indicate that the XI bit of the *Entry0* and *Entry1* registers is enabled.	R	1
ELPA	29	Enables support for large physical addresses. This field is encoded as follows: 0: Large physical address support is disabled. 1: Large physical address support is enabled. If this bit is a 1, the following changes occur to coprocessor 0 registers: • The PFNX field of the EntryLo0 and EntryLo1 registers is writable and concatenated with the PFN field to form the full page frame number. • Access to optional COP0 registers with PA extension, LLAddr, TagLo is defined. If this bit is a 0 and $Config3_{LPA}$=1, then writes to above registers or fields are ignored and reads return 0.	R/W	0
ESP	28	This bit is always 0 as 1K pages are not supported. This bit must be written with 0.	R	0
IEC	27	Enables unique exception codes for the Read-Inhibit and Execute-Inhibit exceptions. This bit is always 1 to indicate that Read-Inhibit exceptions use the TLBRI exception code, and that Execute-Inhibit exceptions use the TLBXI exception code.	R	1
0	26:5	Reserved. Ignored on write; returns zero on read.	R	0
MCAUSE	4:0	Machine Check Cause. Only valid after a Machine Check Exception. This field indicates the cause of the machine check exception and it encoded as follows: 0x0: No Machine Check Reported 0x1: Multiple Hit in TLB(s). 0x2: Multiple Hits in TLB(s) for speculative accesses. The value in EPC might not point to the faulting instruction. 0x3: For Dual VTLB and FTLB. A page with EntryHi.EHINV=0 is written into FTLB and PageMask is not set to a page size that is supported by the FTLB. 0x4: For Dual VTLB and FTLB. A page with EntryHi.EHINV=0 is written into FTLB but the VPN2 field is not consistent with the TLB set selected by the Index register. 0x5: For Hardware Page Table Walker and Dual Page Mode of Directory Level PTEs - first PTE accessed from memory has PTEVld bit set but second PTE accessed from memory does not have PTEVld bit set. 0x6: For Hardware Page Table Walker and derived Huge Page size is power-of-4 but Dual Page mode not implemented. 0x7 - 0x31: Reserved.	R	0

2.2.2.10 Wired (CP0 Register 6, Select 0)

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 2.28. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

Note that wired entries in the TLB 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 VTLB are wired. Release 6 adds the Limit field. The intent of a non-zero value for this field is to place a limit on the number of wired entries in a TLB such that non-wired entries may be shared. If the Limit field is greater than 0, and software attempts to wire an entry greater than the value programmed into the Limit field, the write is ignored. The *Wired* register is reset to zero by a Reset exception.

Hardware will drop any attempt to write the Wired.Wired field with a value greater than either the number stored in the Limit field, or the number of VTLB entries.*Wired* can be set to a non-zero value to prevent the random replacement of up to 63 VTLB pages.

Figure 2.23 Wired Register Format

31 — 21	20 — 16	15 — 6	5 — 0
0	Limit	0	Wired

Table 2.28 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:21	Ignored on write; returns zero on read.	R	0
Limit	20:16	Limit field. This field indicates the maximum number of entries that can be wired, which in the P6600 core is 31. Values above 31 are ignored and the value in this field is truncated to 0x1F. However, if the value in the Limit field is 0, hardware will allow all writes to the Wired field as long as the value being written is less than the total number of TLB entries.	R	0x1F
0	15:6	Ignored on write; returns zero on read.	R	0
Wired	5:0	Defines the number of wired dual entries in the VTLB. A value of 0 in this field indicates that no TLB entries are hard wired. A value of 0x1F indicates that all 31 VTLB entries are hard wired. This field is encoded as follows: 0x00: 0 VTLB entries are hardwired 0x01: 1 VTLB entry is hardwired 0x02: 2 VTLB entries are hardwired ...... 0x1F: 31 VTLB entries are hardwired	R/W	0

2.2.2.11 Bad Virtual Address — BadVAddr (CP0 Register 8, Select 0)

The 64-bit *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions:

- Address error (AdEL or AdES)

- TLB Refill
- TLB Invalid (TLBL, TLBS)
- TLB Read Inhibit (TLBRI)
- TLB Execute Inhibit (TLBXI)
- TLB Modified

The *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.

There is more information about this register in the notes to the $Cause_{ExcCode}$ field.

Figure 2.24 BadVAddr Register Format

63 … 0
BadVAddr

Table 2.29 BadVAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
BadVAddr	63:0	Bad virtual address. This register stores the virtual address that causes one of the TLB exceptions listed above.	R	Undefined

2.2.2.12 PWBase Register (CP0 Register 5, Select 5)

The *PWBase* register contains the Page Table Base virtual address, used as the starting point for hardware page table walking. It is used in combination with the *PWField* and *PWSize* registers. The existence of this register is indicated when $Config3_{PW}$ = 1. For more information on page table walking, refer to Chapter 3 of this manual.

Figure 2.25 shows the format of the *PWBase* register; Table 2.30 describes the *PWBase* register fields.

Figure 2.25 PWBase Register Format

63 … 0
PWBase

Table 2.30 PWBase Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
PWBase	63:0	Page Table Base address pointer.	R/W	0

2.2.2.13 PWField Register (CP0 Register 5, Select 6)

The *PWField* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase* and *PWSize* registers.

The hardware page walker supports multi-level page tables - up to four directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system consists of multiple levels, the lowest level being the Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address. The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in the faulting address. This *PWField* register only exists if $Config3_{PW}$ = 1.

If a synchronous exception condition is detected on a read operation during hardware page-table walking, the automated process is aborted and a TLB Refill exception is taken.

Figure 2.26 shows the formats of the *PWField* Register; Table 2.31 describes the *PWField* register fields.

Figure 2.26 PWField Register Format

63 — 38	37 — 32
0	BDI

31 — 30	29 — 24	23 — 18	17 — 12	11 — 6	5 — 0
0	GDI	UDI	MDI	PTI	PTEI

Table 2.31 PWField Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	63:38	Must be written as zero; returns zero on read.	R	0
BDI	37:32	Base Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Base Directory. The number of index bits is specified by PWSize.BDW.	R	0x0
GDI	29:24	Global Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Global Directory. The number of index bits is specified by $PWSize_{GDW}$. This register must contain a value greater than 0x0C at all times. The entire write is dropped if the write value to this field is less than 12 decimal.	R/W	0xC

Table 2.31 PWField Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
UDI	23:18	Upper Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Upper Directory. The number of index bits is specified by $PWSize_{UDW}$. This register must contain a value greater than 0x0C at all times. The entire write is dropped if the write value to this field is less than 12 decimal.	R/W	0xC
MDI	17:12	Middle Directory index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Middle Directory. The number of index bits is specified by $PWSize_{MDW}$. This register must contain a value greater than 0x0C at all times. The entire write is dropped if the write value to this field is less than 12 decimal.	R/W	0xC
PTI	11:6	Page Table index. Least significant bit of the index field extracted from the faulting address, which is used to index into the Page Table. The number of index bits is specified by $PWSize_{PTW}$. This register must contain a value greater than 0x0C at all times. The entire write is dropped if the write value to this field is less than 12 decimal.	R/W	0xC
PTEI	5:0	Page Table Entry shift. Specifies the logical right shift and rotation which will be applied to Page Table Entry values loaded by hardware page table walking. The entire PTE is logically right shifted by *PTEI*-2 bits first. The purpose of this shift is to remove the software-only bits from what will be written into the TLB entry. Then the two least-significant bits of the shifted value are rotated into position for the RI and XI protection bit locations within the TLB entry. A value of 2 means rotate the right-most 2 bits into the RI/XI bit positions for the TLB entry. A value of 3 means logical shift right by 1 bit the entire PTE and then rotate the right-most 2 bits into the RI/XI positions for the TLB entry. A value of 4 means logical shift right by 2bits the entire PTE and then rotate the right-most 2 bits into the RI/XI positions for the TLB entry. In the P6600 core, the values of 1 and 0 in this field are RESERVED and should not be used; the operation of the page table walker is **UNPREDICTABLE** for these cases. The set of available non-zero shifts is implementation-dependent. Software can discover the available values by writing this field. If the requested shift value is not available, *PTEI* will remain unchanged. A shift of zero must be implemented.	R/W	0x2

Note that the *PTEI* field can be incorrectly programmed so that the entire PFN, C, V, G TLB fields are overwritten with zeros by the logical right shift operation. The intention of this facility is to only remove the SW-only bits of the PTE from the value which will be later written into the TLB.

2.2.2.14 PWSize Register (CP0 Register 5, Select 7)

The 64-bit *PWSize* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase* and *PWField* registers. For more information on the page table walker, refer to Chapter 3 of this manual.

The hardware page walk feature supports multi-level page tables - up to four directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system contains multiple levels, the lowest of which are Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address *BadVAddr*. The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in *BadVAddr*.

Index values used to access Directories are multiplied by the 32-bit native pointer size for the refill. When $PWSize_{PS}$ = 0, the native pointer size is 32 bits (2 bit left shift), and hardware page table walking is applied only when the TLB exception would be taken. When $PWSize_{PS}$ = 1, the native pointer size is 64 bits (3 bit left shift), and hardware page table walking is applied only when a TLB Refill exception would be taken.

The index value used to access the Page Table is multiplied by the native pointer size. An additional multiplier (left shift value) can be specified using the $PWSize_{PTEW}$ field. This allows space to be allocated in the Page Table structure for software-managed fields.

This register only exists if $Config3_{PW}$ = 1.

Figure 2.27 shows the formats of the *PWSize* Register; Table 2.32 describes the *PWSize* register fields.

Figure 2.27 PWSize Register Format

63 … 38	37 … 32
0	BDW

31	30	29 … 24	23 … 18	17 … 12	11 … 6	5 … 0
0	PS	GDW	UDW	MDW	PTW	PTEW

Table 2.32 PWSize Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	63:38	Must be written as zero; returns zero on read.	0	0
BDW	37:32	Base Directory index. This field is encoded as follows: 0: No read is performed using the base directory index. 0x01 - 0x3F: The number of bits to be extracted from BadVAddr to create an index into the base directory. The least significant bit of the field is specified by the PWField.BDI field.	R	0x0

Table 2.32 PWSize Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31	Must be written as zero; returns zero on read.	0	0
PS	30	Pointer Size. This field determines whether the pointer is loaded with 32-bit aligned addresses or 64-bit aligned address and is encoded as follows: 0: 32-bit pointer size. Pointers within Directories are loaded as 32-bit addresses. Hardware Page Table Walking is activated only for 32-bit address regions, when the TLB Refill vector would be used. 1: 64-bit pointer size. Pointers within Directories are loaded as 64-bit addresses. Hardware Page Table Walking is activated only for 64-bit address regions, when the XTLB Refill vector would be used.	R/W	0
GDW	29:24	Global Directory index width. This field is encoded as follows: 0: No read is performed using Global Directory index. 0x01 - 0x 3F: A non-zero number in this field indicates the number of bits to be extracted from *BadVAddr* to create an index into the Global Directory. The least significant bit of the field is specified by $PWField_{GDI}$.	R/W	0
UDW	23:18	Upper Directory index width. 0: No read is performed using Upper Directory index. 0x01 - 0x 3F: A non-zero number in this field indicates the number of bits to be extracted from *BadVAddr* to create an index into the Upper Directory. The least significant bit of the field is specified by $PWField_{UDI}$.	R/W	0
MDW	17:12	Middle Directory index width. 0: No read is performed using Middle Directory index. 0x01 - 0x 3F: A non-zero number in this field indicates the number of bits to be extracted from *BadVAddr* to create an index into the Middle Directory. The least significant bit of the field is specified by $PWField_{MDI}$.	R/W	0
PTW	11:6	Page Table index width. This field is encoded as follows: 0: UNPREDICTABLE. A value of 0 in this field causes unpredictable behavior. This field should have a non-zero value. 1: Number of bits to be extracted from *BadVAddr* to create an index into the Page Table. The least significant bit of the field is specified by $PWField_{PTI}$. Note that a write of 0 to this bit causes the entire write to be dropped.	R/W	1
PTEW	5:0	Specifies the left shift applied to the Page Table index, in addition to the shift required to account for the native data size of the machine. In the P6600 core, the PTEW field cannot be set to value 1 if $PWSize_{PS}$ = 1. In addition, if $PWSize_{PTEW}$ is already set to 1 and $PWSize_{PS}$ is changed from 0 to 1, hardware forces the $PWSize_{PTEW}$ field to a value to 0 (as a side-effect of updating $PWSize_{PS}$ to 1). Therefore, if $PWSize_{PS}$ = 0, then PTEW can be set to 1, else it is always 0.	R/W	0

Table 2.33 describes valid *PWSize* $_{PS/PTEW}$ and $PWCtl_{HugePg}$ settings.

Table 2.33 PS/PTEW Usage

$PWSize_{PS}$	$PWCtl_{HugePg}$	$PWSize_{PTEW}$	Pointer Addressing	Directory Pointer Size	Non-leaf PTE Size	Leaf PTE SIze	Suggested Use Case
0	0	0	32b	32b	N/A	32b	32-bit Compatibility
0	0	1	32b	32b	N/A	64b	32-bit PA 32-bit Compatibility
0	1	0	32b	32b	32b	32b	32-bit with Huge Page Compatibility
0	1	1	32b	64b	64b	64b	32-bit with Huge Pages and PA 32-bit Compatibility
1	0	0	64b	64b	N/A	64b	64-bit Base
1	0	1	64b	64b	N/A	128b	64-bit with Extended PTE
1	1	0	64b	64b	64b	64b	64-bit with Huge Pages
1	1	1	64b	128b	128b	128b	64-bit with Huge Pages and Extended PTE

2.2.2.15 PWCtl Register (CP0 Register 6, Select 6)

The 32-bit *PWCtl* register configures hardware page table walking for TLB refills. It is used in combination with the *PWBase, PWField* and *PWSize* registers. Hardware page table walking is disabled when $PWCtl_{PWEn}$ = 0.

The hardware page walker feature supports multi-level page tables - up to four directory levels plus one page table level. The lowest level of any page table system is an array of Page Table Entries (PTEs). This array is known as a Page Table (PT) and is indexed using bits from the faulting address. A single-level page table system contains only a single Page Table.

A multi-level page table system supports multiple levels, the lowest of which are Page Table Entries. Levels above the lowest Page Table level are known as Directories. A directory consists of an array of pointers. Each pointer in a directory is either to another directory or to a Page Table.

The Page Table and the Directories are indexed by bits extracted from the faulting address *BadVAddr*. The *PWBase* register contains the base address of the first Directory or Page Table which will be accessed. The *PWSize* register specifies the number of index bits to be used for each level. The *PWField* register specifies the location of the index fields in *BadVAddr*. The existence of this register is denoted when $Config3_{PW}$ = 1.

Figure 2.28 shows the formats of the *PWCtl* Register; Table 2.34 describes the *PWCtl* register fields.

Figure 2.28 PWCtl Register Format

31	30	29	28	27	26	25 … 8	7	6	5 … 0
PWEn	PWDirExt	0	XK	XS	XU	0	DPH	HugePg	Psn

Table 2.34 PWCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
PWEn	31	Hardware Page Table walker enable If this bit is set, then the Hardware Page Table is enabled.	R/W	0
PWDirExt	30	PW Indices - PWField and PWSize - extended for 4th directory level - the Base level.	R	0
0	29	Reserved, Must be written as zero; returns zero on read.	R	0
XK	28	XKSEG kernel address space management. This bit is encoded as follows: 0: xkseg misses generate a TLB miss exception. The hardware page walk is not initiated. 1: The page table walker handles xkseg.	R/W	0
XS	27	XSSEG supervisor address space management. This bit is encoded as follows: 0: xsseg misses generate a TLB miss exception. The hardware page walk is not initiated. 1: The page table walker handles xsseg accesses.	R/W	0
XU	26	XUSEG user address space management. This bit is encoded as follows: 0: xuseg misses generate a TLB miss exception. The hardware page walk is not initiated. 1: The page table walker handles xuseg accesses.	R/W	0
0	25:8	Reserved, Must be written as zero; returns zero on read.	R	0
DPH	7	Dual Page format of Huge Page support. This bit is only used when *HugePg* = 1. If *DPH* bit is set, then a Huge Page PTE can represent a power-of-4 memory region or a 2x power-of-4 memory region. For the first case, one PTE is used for even TLB page and the adjacent PTE is used for the odd PTE. For the latter case, the Hardware will synthesize the physical addresses for both the even and odd TLB pages from the single PTE entry. If *DPH* bit is clear, then a Huge Page PTE can only represent a region that is 2 x power-of-4 in size. For this case, the Hardware will synthesize the physical addresses for both the even and odd TLB pages from the single PTE entry.	R/W	0
HugePg	6	Huge Page PTE supported in Directory levels. If this bit is set, then Huge Page PTE in non-leaf table (i.e., directory level) is supported.	R/W	0
PSn	5:0	Bit position of *PTEvld* in Huge Page PTE. Only used when *HugePg* field is set.	R/W	0

Software enables Huge Pages by setting $PWCtl_{HugePg}$ = 1. Software can disable Huge Pages by setting $PWCtl_{HugePg}$ = 0. The 6-bit *PWCtlPsn* field indicates the starting bit position for *PTEvld* up to bit 64 in the PTE. Software can determine the supported range by writing ones to *PWCtlPsn*, then reading the value.

Table 2.35 describes how the *HugePg* field is used to denote whether Huge Pages are supported or not.

Table 2.35 HugePg Field and Huge Page configurations

$PWCTL_{HugePg}$	Type of Entry		Rsvd Field in Non-leaf entry	Comment
	Non-Leaf	Leaf		
0	Always Pointer PTE_{PTEVld} not used	Always PTE PTE_{PTEVld} not used	X	No Huge-Page Support
1	PTE_{PTEVld} = 0 means Pointer PTE_{PTEVld} = 1 means Huge Page	Always PTE PTE_{PTEVld} not used	Must be 0	Huge-Page Support

Table 2.36 describes how Huge Pages are represented in the Directory Levels.

Table 2.36 Huge Page representation in Directory Levels

$PWCTL_{DPH}$	Size of Huge Page		Comment
	Power of 4	non-Power of 4	
0	Not Allowed If encountered, HW Page Walker aborts and TLB Refill exception is taken.	Allowed Even TLB page and Odd TLB page entries both derived from single PTE	Huge-Page region can only be 2x power-of-4
1	Allowed Two PTEs are read from memory by the HW Page Walker to be used for the Even and Odd TLB page entries.	Allowed Even TLB page and Odd TLB page entries both derived from single PTE	Huge-Page region can be any power-of-2 (either power of 4 or 2x power-of-4)

Table 2.37 describes the usage of the *XK, XS, and XU* fields is used to indicate the hardware page walker capability.

Table 2.37 PWCtl XK/XS/XU Register Field Configurations

Register Fields			VA Bits Prepended to Global Directory Index	Hardware Walker Capability
$PWCTL_{XK}$	$PWCTL_{XS}$	$PWCTL_{XU}$		
0	0	0	None	Disabled
0	0	1	None	xuseg
0	1	0	---	Reserved. Not supported in the P6600 core.
0	1	1	62	xuseg and xsseg
1	0	0	---	Reserved. Not supported in the P6600 core.
1	0	1	63	xuseg and xkseg
1	1	0	---	Reserved. Not supported in the P6600 core.
1	1	1	63:62	xuseg, xsseg, xkseg

2.2.3 Exception Control Registers

This section contains the following exception control registers.

- Section 2.2.3.1, "Cause (CP0 Register 13, Select 0)" on page 100
- Section 2.2.3.2, "Exception Program Counter — EPC (CP0 Register 14, Select 0)" on page 104
- Section 2.2.3.3, "Error Exception Program Counter — ErrorEPC (CP0 Register 30, Select 0)" on page 104
- Section 2.2.3.4, "BadInstr Register (CP0 Register 8, Select 1)" on page 105
- Section 2.2.3.5, "BadInstrP Register (CP0 Register 8, Select 2)" on page 106

Also refer to the Interrupt Control register in Section 2.2.1.12, "Interrupt Control — IntCtl (CP0 Register 12, Select 1)" on page 76.

2.2.3.1 Cause (CP0 Register 13, Select 0)

The *Cause* register describes the cause of the most recent exception and controls software interrupt requests and the vector through which interrupts are dispatched. With the exception of the *IP1:0*, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. *IP7:2* are interpreted as the Requested Interrupt Priority Level (RIPL) in External Interrupt Controller (EIC) interrupt mode.

Figure 2.29 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21	20 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	FDCI	0	IP7-2	IP1-0	0	ExcCode	0

Table 2.38 Field Descriptions for Cause Register

Name	Bit(s)	Description	Read/ Write	Reset State
BD	31	Indicates whether the last exception taken occurred in a branch delay slot. 0: Exception taken was not in delay slot 1: Exception taken was in delay slot The processor updates *BD* only if the *EXL* bit in the *Status* register was zero when the exception occurred. If the exception occurred in a branch delay slot, the exception program counter (*EPC)* is set to restart execution at the branch. Software should read this bit to determine if the exception was taken in a delay slot.	R	Undefined
TI	30	Timer Interrupt. Denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types) 0: No timer interrupt is pending 1: Timer interrupt is pending Hardware sets this bit based on the state of the external *SI_TimerInt* signal. See also the descriptions of the *Count* and *Compare* registers.	R	Undefined

Table 2.38 Field Descriptions for Cause Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
CE	29:28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is **UNPREDICTABLE** for all exceptions except Coprocessor Unusable. 00: Coprocessor 0 01: Coprocessor 1 10: Coprocessor 2 (not supported in P6600) 11: Coprocessor 3 (not supported in P6600)	R	Undefined
DC	27	Disable *Count* register. In some power-sensitive applications, the *Count* register is not used but may still be the source of some noticeable power dissipation. This bit allows the *Count* register to be stopped in such situations. For example, this can be useful during low-power operation following a **`wait`** instruction. 0: Enable counting of *Count* register 1: Disable counting of *Count* register	R/W	0
PCI	26	Performance Counter Interrupt. Indicates whether a performance counter interrupt is pending (analogous to the *IP* bits for other interrupt types). 0: No performance counter interrupt is pending 1: Performance counter interrupt is pending See also the description of the *PerfCnt* registers.	R	Undefined
0	25:24	Reserved. Write as zero. Ignored on reads.	R	0
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector: 0: Use the general exception vector (0x180) 1: Use the special interrupt vector (0x200) When the *IV* bit in the *Cause* register is 1 and the *BEV* bit in the *Status* register is 0, the special interrupt vector represents the base of the vector interrupt table.	R/W	Undefined
WP	22	Indicates that a watch exception was deferred because either the $Status_{EXL}$ bit or the $Status_{ERL}$ bit was a logic ‘1’ at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated when $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software should never write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is **UNPREDICTABLE** whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once $Status_{EXL}$ and $Status_{ERL}$ are both zero. Software should clear this bit, but never set it. It is set by hardware.	R/W	Undefined
FDCI	21	Fast Debug Channel Interrupt: This bit denotes whether an FDC interrupt is pending (analogous to the *IP* bits for other interrupt types). 0: No FDC interrupt is pending 1: FDC interrupt is pending This bit is set by hardware based on the state of the external *SI_FDCInt* signal.	R	Undefined
0	20:16	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.38 Field Descriptions for Cause Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>IP7-2
RIPL</td><td>15:10</td><td>Indicates an interrupt is pending.
If External Interrupt Controller (EIC) mode is disabled ($Config3_{VEIC} = 0$), timer interrupts are combined in a system-dependent way with any hardware interrupt. Each bit of this field maps to an individual hardware interrupt.
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr>
<tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr>
<tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr>
<tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr>
<tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr>
<tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr>
</table>
If EIC interrupt mode is enabled ($Config3_{VEIC} = 1$), these bits take on a different meaning and are interpreted as the Requested Interrupt Priority Level (RIPL) field.
When EIC interrupt mode is enabled, this field (RIPL) contains the encoded (0 - 63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.</td><td>R</td><td>Undefined</td></tr>
<tr><td>IP1-0</td><td>9:8</td><td>Controls the request for software interrupts:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr>
<tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr>
</table>
These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits are driven onto the external SI_SWInt[1:0] bus.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>0</td><td>7</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>ExcCode</td><td>6:2</td><td>Encodes the cause of the last exception as described in Table 2.39.</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>1:0</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
</table>

Table 2.39 Exception Code Values in ExcCode Field of Cause Register

Value (decimal)	Value (hex)	Code	Description
0	0x0	Int	Interrupt
1	0x1	Mod	Store, but page marked as read-only in the TLB
2	0x2	TLBL	Load or fetch, but page not present or marked as invalid in the TLB
3	0x3	TLBS	Store, but page not present or marked as invalid in the TLB
4	0x4	AdEL	Address error on load/fetch or store respectively. Address is either wrongly aligned, or a privilege violation.
5	0x5	AdES	
6	0x6	IBE	Bus error signaled on instruction fetch
7	0x7	DBE	Bus error signaled on load/store (imprecise)

Table 2.39 Exception Code Values in ExcCode Field of Cause Register *(continued)*

Value (decimal)	Value (hex)	Code	Description
8	0x8	Sys	System call, i.e. `syscall` instruction executed.
9	0x9	Bp	Breakpoint, i.e. `break` instruction executed. If an SDBBP instruction is executed while the processor is running in EJTAG Debug Mode, this value is written to the *Debug*$_{DExcCode}$ field to denote an SDBBP in Debug mode.
10	0xA	RI	Reserved instruction. Instruction code not recognized (or not legal)
11	0xB	CpU	Coprocessor Unusable Exception. Instruction code was for a co-processor which is not enabled in *Status*$_{CU3-0}$.
12	0xC	Ov	Overflow exception. Overflow from a trapping variant of integer arithmetic instructions.
13	0xD	Tr	Trap exception. Condition met on one of the conditional trap instructions `teq` etc.
14	0xE	MSAFPE	MSA floating point unit exception.
15	0xF	FPE	Floating point unit exception — more details in the FPU control/status registers.
16	0x10	TLBPAR	TLB parity error exception.
17 - 18	0x11 - 0x12	-	Available for implementation-dependent use.
19	0x13	TLBRI	TLB read inhibit exception.
20	0x14	TLBXI	TLB execute inhibit exception.
21	0x15	MDADi	MSADi exception.
22	0x16	-	Reserved.
23	0x17	WATCH	Instruction or data reference matched a watchpoint. Refer to *WatchHi*/*WatchLo* address.
24	0x18	MCheck	Machine check exception.
25	0x19	-	Reserved
26	0x1A	DSPDis	DSP ASE not enabled or not present exception. This exception occurs when trying to run an instruction from the MIPS DSP ASE, but the ASE is either not enabled or not available. If this exception occurs and the DSP ASE is present in the system, check the state of the *Status*$_{MX}$ bit to make sure it is set to ‘1’. This value is not used in the P6600 core.
27	0x1B	GE	Hypervisor Exception (Guest Exit). GE is set to 1 in following cases: - Hypervisor-intervention exception occurred during guest mode execution. - Hypercall executed in root mode GuestCtl0$_{GExcCode}$ contains additional cause information.
28 29	0x1C - 0x1D	-	Reserved.
30	0x1E	CacheErr	Parity/ECC error occurred somewhere in the P6600 core, on either an instruction fetch, load, or cache refill. This exception does not occur during normal operation, but can occur while in debug mode. Refer to Section 2.2.8.1 “Debug (CP0 Register 23, Select 0)” for more information.
31	0x1F	-	Reserved.

2.2.3.2 Exception Program Counter — EPC (CP0 Register 14, Select 0)

Following an exception other than an error or debug exception, the 64-bit *Exception Program Counter* (*EPC*) contains the address at which processing resumes after the exception has been serviced (the corresponding debug and error exception use *DEPC* and *ErrorEPC* respectively).

Unless the *EXL* bit in the *Status* register is set (indicating, among other things, that interrupts are disabled), the processor writes the *EPC* register when an exception occurs.

- For synchronous (precise) exceptions, *EPC* contains either:
  - The virtual address of the instruction that was the direct cause of the exception, or
  - The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set.
- For asynchronous (imprecise) exceptions, *EPC* contains the address of the instruction at which to resume execution.

The processor reads the *EPC* register as the result of execution of the **`eret`** instruction.

Figure 2.30 EPC Register Format

63 ... 0
EPC

Table 2.40 EPC Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EPC	63:0	Exception Program Counter.	R/W	Undefined

2.2.3.3 Error Exception Program Counter — ErrorEPC (CP0 Register 30, Select 0)

The 64-bit *ErrorEPC* register is a read/write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

This full 32-bit register is filled with the restart address on a cache error exception or any kind of CPU reset — in fact, any exception which sets $Status_{ERL}$ and leaves the CPU in "error mode".

Figure 2.31 ErrorEPC Register Format

63 ... 0
ErrorEPC

Table 2.41 ErrorEPC Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
ErrorEPC	63:0	Error Exception Program Counter.	R/W	Undefined

2.2.3.4 BadInstr Register (CP0 Register 8, Select 1)

The 32-bit *BadInstr* register is a read-only register that captures the most recent instruction which caused one of the following exceptions:

- Instruction validity

  Coprocessor Unusable, Reserved Instruction

- Execution Exception

  Integer Overflow, Trap, System Call, Breakpoint, Floating Point, Coprocessor 2 exception

- Addressing

  Address Error, TLB or XTLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstr* register is provided to allow acceleration of instruction emulation. The *BadInstr* register is only set by exceptions which are synchronous to an instruction. The *BadInstr* register is not set by Interrupts, NMI, Machine check, Bus Error or Cache Error exceptions. The *BadInstr* register is not set by Watch or EJTAG exceptions.

When a synchronous exception occurs for which there is no valid instruction word (for example TLB Refill - Instruction Fetch), the value stored in *BadInstr* is **UNPREDICTABLE**. Presence of the *BadInstr* register is indicated by the $Config3_{BI}$ bit.

Figure 2.32 shows the format of the *BadInstr* register; Table 2.42 describes the *BadInstr* register fields.

Figure 2.32 BadInstr Register Format

31 … 0
BadInstr

Table 2.42 BadInstr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
BadInstr	31:0	Faulting instruction word. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined

2.2.3.5 BadInstrP Register (CP0 Register 8, Select 2)

The 32-bit *BadInstrP* register is used in conjunction with the *BadInstr* register. The *BadInstrP* register contains the prior branch instruction, when the faulting instruction is in a branch delay slot.

The *BadInstrP* register is updated for these exceptions:

- Instruction validity

  Coprocessor Unusable, Reserved Instruction

- Execution Exception

  Integer Overflow, Trap, System Call, Breakpoint, Floating Point, Coprocessor 2 exception

- Addressing

  Address Error, TLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstrP* register is provided to allow acceleration of instruction emulation. The *BadInstrP* register is only set by exceptions which are synchronous to an instruction. The *BadInstrP* register is not set by Interrupts, NMI, Machine check, Bus Error or Cache Error exceptions. The *BadInstr* register is not set by Watch or EJTAG exceptions.

When a synchronous exception occurs and the faulting instruction is not in a branch delay slot, then the value stored in *BadInstrP* is **UNPREDICTABLE**. Presence of the *BadInstrP* register is indicated by the $Config3_{BP}$ bit.

Figure 2.33 shows the proposed format of the *BadInstrP* register; Table 2.43 describes the *BadInstrP* register fields.

Figure 2.33 BadInstrP Register Format

31 ... 0
BadInstrP

Table 2.43 BadInstrP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
BadInstrP	31:0	Prior branch instruction. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined

2.2.4 Timer Registers

This section contains the following timer registers.

- Section 2.2.4.1, "Count (CP0 Register 9, Select 0)" on page 107
- Section 2.2.4.2, "Compare (CP0 Register 11, Select 0)" on page 107

2.2.4.1 Count (CP0 Register 9, Select 0)

The 32-bit *Count* register acts as a timer, incrementing at a constant rate. Incrementing of this register occurs whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. When enabled by clearing the *DC* bit in the *Cause* register, the counter increments every other clock (half the clock rate).

The *Count* may be stopped in either of the following two circumstances.

- Some implementations may stop *Count* in the low-power mode, for example, through the **`wait`** instruction, but only if the $Cause_{DC}$ flag is set to 1.

- When the device is in debug mode, the *Count* register can be stopped by setting $Debug_{CountDM}$. By writing the $Count_{DM}$ bit, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

The *Count* field starts counting from whatever value is loaded into it. However, OS timers are usually implemented by leaving *Count* free-running and writing *Compare* as necessary. This counter rolls over when reaching it maximum value.

By writing the $Count_{DM}$ bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

Figure 2.34 Count Register Format

31 ... 0
Count

Table 2.44 Count Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
Count	31:0	Interval counter.	R/W	Undefined

2.2.4.2 Compare (CP0 Register 11, Select 0)

The 32-bit *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. When the value of the *Count* register equals the value of the *Compare* register, the *SI_TimerInt* output pin is asserted. *SI_TimerInt* remains asserted until the *Compare* register is written.

The *SI_TimerInt* output can be fed back into the P6600 core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 in order to set interrupt bit *IP*(7) in the *Cause* register.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. As a side effect, writing a value to this register clears the timer interrupt.

Figure 2.35 Compare Register Format

31 ... 0
Compare

Table 2.45 Compare Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Compare	31:0	Interval count compare value.	R/W	0xFFFF_FFFF

2.2.5 Cache Management Registers

This section contains the following cache management registers.

- Section 2.2.5.1, "Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)" on page 108
- Section 2.2.5.2, "Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)" on page 110
- Section 2.2.5.3, "Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)" on page 111
- Section 2.2.5.4, "Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)" on page 111
- Section 2.2.5.5, "Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)" on page 112
- Section 2.2.5.6, "Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)" on page 115
- Section 2.2.5.7, "Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)" on page 116
- Section 2.2.5.8, "Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)" on page 117
- Section 2.2.5.9, "Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)" on page 118
- Section 2.2.5.10, "ErrCtl (CP0 Register 26, Select 0)" on page 118
- Section 2.2.5.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)" on page 120

2.2.5.1 Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)

The 64-bit *ITagLo* register acts as the interface to the instruction cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *ITagLo* register as the source of tag information.

When the *WST* bit of the *ErrCtl* register is asserted, this register becomes the interface to the way-selection RAM. In this mode, the fields are redefined to give appropriate access the contents of the WS array instead of the Tag array.

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 s of $ErrCtl_{WST}$ and $ErrCtl_{SPR}$.

- Default cache interface mode (*ErrCtl*$_{WST}$ = 0)
- Diagnostic "way select test mode" (*ErrCtl*$_{WST}$ = 1)

See the diagrams below for a description.

ITagLo (ErrCtl$_{WST}$ = 0)

In this mode, this register is a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations—routinely used in cache initialization.

Figure 2.36 ITagLo Register Format (ErrCtl$_{WST}$ = 0)

63 — 40	39 — 32
0	PTagLo

31 — 12	11 — 8	7	6	5	4 — 1	0
PTagLo	0	V	0	L	0	P

Table 2.46 Field Descriptions for ITagLo Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:40	Must be written as zero; returns zero on read.	R	0
PTagLo	39:12	The cache address tag, which is a physical address because the P6600's caches are physically tagged. It holds bits 40:16 of the physical address. The low-order 16 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
0	11:8	Must be written as zero; returns zero on read.	R	0
V	7	Set to 1 if this cache entry is valid (set to zero to initialize the cache).	R/W	Undefined
0	6	Must be written as zero; returns zero on read.	R	0
L	5	Specifies the lock bit for the cache tag. This bit is set to lock this cache entry, preventing it from being replaced by another line when a cache miss occurs. When this bit is set, and the V bit is set, the corresponding cache line will not be replaced by the cache replacement algorithm. This can be used for critical data that must not be removed from the cache. However, this can reduce the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
0	4:1	Must be written as zero; returns zero on read.	R	0
P	0	Parity bit over the cache tag entries. This bit is updated with tag array parity on CACHE Index Load Tag operations and used as tag array parity on Index Store Tag operations when the *PO* bit of the *ErrCtl* register is set.	R/W	Undefined

ITagLo-WST (ErrCtlWST = 1)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access the data in these fields either by **`cache`** load-tag or store-tag operations when *ErrCtl*$_{WST}$ is set.

Figure 2.37 ITagLo Register Format ($ErrCtl_{WST}$ = 1)

63 – 32
0

31 – 16	15 – 10	9 – 0
0	LRU	0

Table 2.47 Field Descriptions for ITagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:16	Must be written as zero; returns zero on read.	R/W	Undefined
LRU	15:10	LRU bits. This field contains the value read from the WS array after a CACHE Index Load WS operation. It is used to store into the WS array during CACHE Index Store WS operations. When reading or writing the tag in way-select test mode (that is, with $ErrCtl_{WST}$ set), this field reads or writes the LRU ("least recently used") state bits, held in the way-select RAM.	R/W	Undefined
0	9:8	Must be written as zero; returns zero on read.	R	0

2.2.5.2 Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)

This register represents the I-cache Predecode bits and is intended for diagnostic use only.

Figure 2.38 ITagHi Register Format

31 – 25	24 – 18	17 – 11	10 – 4	3	2	1	0
PREC_67	PREC_45	PREC_23	PREC_01	P_67	P_45	P_23	P_01

Table 2.48 Field Descriptions for ITagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
PREC_67	31:25	P6600 family cores do some decoding of instructions when they're loaded into the I-cache, which helps speed instruction dispatch. When you use `cache` tag load/store instructions, you see that information here. The individual *PREC* fields hold precode information for pairs of adjacent instructions in the I-cache line, and the *P* fields hold parity over them.	R/W	Undefined
PREC_45	24:18		R/W	Undefined
PREC_23	17:11		R/W	Undefined
PREC_01	10:4		R/W	Undefined
P_67	3		R/W	Undefined
P_45	2		R/W	Undefined
P_23	1		R/W	Undefined
P_01	0		R/W	Undefined

2.2.5.3 Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)

The *IDataLo* register is a register that acts as the interface to the instruction cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *IDataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *IDataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction.

Two registers (*IDataHi*, *IDataLo*) are needed, because the P6600 core loads I-cache data at least 64 bits at a time.

Figure 2.39 IDataLo Register Format

31 0
DATA

Table 2.49 IDataLo Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DATA	31:0	Low-order data read from the cache data array.	R/W	Undefined

2.2.5.4 Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)

The *IDataHi* register is a register that acts as the interface to the cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *IDataHi* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *IDataHi* can be written to the cache data array by doing an Index Store Data CACHE instruction.

Because the interface to the I-cache only operates on pairs of instructions, two registers (*IDataHi*, *IDataLo*) are needed because the P6600 core loads I-cache data at least 64-bits at a time. The high instruction is written into the *IDataHi* register. Note that *IDataHi* and *IDataLo* reflect the memory ordering of the instructions. Depending on the endianness of the system, Instruction0 belongs in either *IDataHi* (BigEndian) or *IDataLo* (LittleEndian) and vice versa for Instruction1.

Figure 2.40 IDataHi Register Format

31 0
DATA

Table 2.50 IDataHi Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DATA	31:0	High-order data read from the cache data array.	R/W	Undefined

2.2.5.5 Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)

The 64-bit *DTagLo* register acts as the interface to the data cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *DTagLo* register as the source of tag information.

When the *WST* bit of the *ErrCtl* register is asserted, this register becomes the interface to the way-selection RAM. In this mode, the fields are redefined to give appropriate access the contents of the WS array instead of the Tag array.

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

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 intervention uses, and each combine their ways together in a single entry per set. The WS RAM also combines the lock 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.

Note that the P6600 core does not implement the *DTagHi* register.

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

- Default cache interface mode ($ErrCtl_{WST}$ = 0, $ErrCtl_{DYT}$ = 0)
- Diagnostic "way select test mode" ($ErrCtl_{WST}$ = 1, $ErrCtl_{DYT}$ = 0)
- Diagnostic "dirty array test mode" ($ErrCtl_{WST}$ = 0, $ErrCtl_{DYT}$ = 1)

For all modes, the data RAM, tag RAM, WS RAM, and duplicate tag RAM are read. In addition, for duplicate tag array test mode, the duplicate tag RAM is also read, and for duplicate dirty array test mode, the duplicate Dirty RAM is read. Table 2.51 shows which RAMs are accessed for each mode for Loads and Stores.

Table 2.51 Summary of D-cache RAM accesses for Index Loads and Stores

Index Cacheop	Mode		RAM Being Accessed					
	WST	DYT	Primary Tag RAM	WS RAM	Data RAM	Dirty RAM	Duplicate Tag RAM	Duplicate Dirty RAM
Tag Store	0	0	WR	partial WR	RD	—	WR	—
Tag Load	0	0	RD	RD	RD	RD	—	—
Tag Store	1	0	—	partial WR	RD	—	—	—
Tag Load	1	0	RD	RD	RD	RD	—	—
Data Store	1	0	—	—	WR	—	—	—
Tag Store	0	1	—	—	RD	WR	—	WR
Tag Load	0	1	RD	RD	RD	RD	—	—

DTagLo ($ErrCtl_{WST}$ = 0, $ErrCtl_{DYT}$ = 0)

In this mode, this register is a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations—routinely used in cache initialization. For stores in this mode, the tag RAM, WS RAM, and duplicate tag RAM are written. Also for stores, the $ErrCtl_{PO}$ bit controls whether the tag RAM is written with P bit or with generated parity; the other RAMs written in this mode always use generated parity.

Figure 2.41 DTagLo Register Format ($ErrCtl_{WST}$ = 0, $ErrCtl_{DYT}$ = 0)

63 40	39 12	11	10 8	7	6	5	4 1	0
Unused	PTagLo (40-bit address mode)	VA11	0	V	E	L	0	P

Table 2.52 Field Descriptions for DTagLo Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:40	Unused	R/W	Undefined
PTagLo	39:12	The cache address tag — a physical address because the P6600 caches are physically tagged. It holds bits 39:12 of the physical address. The low 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
VA11	11	This bit always gets the virtual address bit [11] of the tag if the index load tag cache instruction is executed.	R/W	Undefined
0	10:8	Reserved. Write as zero. Ignored on reads.	R	0
V	7	Valid entry: This bit is set if this cache entry is valid (set zero to initialize the cache). Index Load: load from V field in primary tag RAM Index Store: store to V field in primary and duplicate tag RAM	R/W	Undefined
E	6	Exclusive entry: This bit is set if this cache entry is exclusive (set zero to initialize the cache). Index Load: load from E field in primary tag RAM Index Store: store to E field in primary tag RAM	R/W	Undefined
L	5	Locked entry: This bit is set to lock this cache entry, preventing it from being replaced by another line when there's a cache miss. Done when you have data so critical that it must be in the cache: it's quite costly, reducing the efficiency of the cache for memory data competing for space at this index. Index Load: load from appropriate way of L field in WS RAM Index Store: store to appropriate way of L and LP field in WS RAM, and if V is set, make selected way MRU in WS RAM; also, store to L field of duplicate tag RAM.	R/W	Undefined
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
P	0	Parity bit over the PTAG, E, and V bits of the cache tag entries Index Load: load from P field in primary tag RAM Index Store: possible write value for the P field of the primary tag RAM; write this bit if *ErrCtl.PO* = 1, else generate; parity written to other RAMs is generated.	R/W	Undefined

DTagLo-WST($ErrCtl_{WST}$ = 1, $ErrCtl_{DYT}$ = 0)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **`cache`** load-tag/store-tag operations when *$ErrCtl_{WST}$* is set: then you get the data in these fields. For stores in this mode, the WS RAM is written. Also for stores, the $ErrCtl_{PO}$ bit controls whether the WS RAM is written with LP bits or with generated parity; the other RAMs written in this mode always use generated parity. Also for stores, the LP and L fields only have the appropriate way written in the WS RAM. It is software's responsibility to maintain consistency with the value of the L field written into the duplicate tag RAM.

Figure 2.42 DTagLo Register Format ($ErrCtl_{WST}$ = 1, $ErrCtl_{DYT}$ = 0)

63–32
0

31–24	23–20	19–16	15–10	9–0
0	LP	L	LRU	0

Table 2.53 Field Descriptions for DTagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:24	Reserved. Write as zero. Ignored on reads.	R	0
LP	23:20	Parity for Cache-line locking control bits, held in the way select RAM. Each bit of this field is a parity bit for the corresponding bit in the L field. Index Load: load from LP field of WS RAM Index Store: store to appropriate way of LP field of WS RAM if *$ErrCtl_{PO}$*=1, else generate;	R/W	Undefined
L	19:16	Cache-line locking control bits, held in the way select RAM. Index Load: load from L field of WS RAM Index Store: store to appropriate way of L field of WS RAM.	R/W	Undefined
LRU	15:10	When reading or writing the tag in way select test mode (that is, with *$ErrCtl_{WST}$* set) this field reads or writes the LRU ("least recently used") state bits, held in the way select RAM. Index Load: load from LRU field of WS RAM Index Store: store to LRU field of WS RAM	R/W	Undefined
0	9:0	Reserved. Write as zero. Ignored on reads.	R	0

DTagLo-DYT ($ErrCtl_{WST}$ = 0, $ErrCtl_{DYT}$ = 1)

The dirty RAM is another slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **`cache`** load-tag/store-tag operations when *$ErrCtl_{DYT}$* is set: then you get the data in these fields. For stores, the Dirty RAM is written. For stores, the Dirty RAM and duplicate Dirty RAM are written. Also for stores, the *$ErrCtl_{PO}$* bit controls whether the Dirty RAM is written with DP bits or with generated parity; the other RAMs written in this mode always use generated parity.

Figure 2.43 DTagLo-DYT Register Format

63 … 32
0

31 … 24	23 … 20	19 … 16	15 … 12	11 … 10	9 … 0
0	DP	D	0	A	0

Table 2.54 Field Descriptions for DTagLo-DYT Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:24	Reserved. Write as zero. Ignored on reads.	R	0
DP	23:20	Parity for Cache line "dirty" bits. Index Load: load from DP field of Dirty RAM Index Store: store to DP field of Dirty RAM if $ErrCtl_{PO}$=1, else generate;	R/W	Undefined
D	19:16	Cache line "dirty" bits. Index Load: load from D field of Dirty RAM Index Store: store to D field of Dirty RAM	R/W	Undefined
0	15:12	Reserved. Write as zero. Ignored on reads.	R	0
A	11:10	Cache line "alias" bits. Index Load: load from A field of Dirty RAM Index Store: store 0 and A[10] to A field of Dirty RAM	R/W	Undefined
0	9:0	Reserved. Write as zero. Ignored on reads.	R	0

2.2.5.6 Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)

In the P6600 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.

The *DDataLo* register acts as the interface to the data cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DDataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *DDataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction.

Figure 2.44 DDataLo Register Format

31 … 0
DATA

Table 2.55 DDataLo Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DATA	31:0	Low-order data read from the cache data array.	R/W	Undefined

2.2.5.7 Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)

The *L23TagLo* register acts as the interface to the L2 or L3 cache tag array. The L2 and L3 Index Store Tag and Index Load Tag operations of the CACHE instruction use the *L23TagLo* register as the source of tag information. Note that the P6600 CPU does not implement the *L23TagHi* register.

Figure 2.45 and Table 2.56 describe the fields of L23TagLo as interpreted by the L2 during Index Load Tag and Index Store Tag cache-ops. In Figure 2.46, the Tag field is always left justified so system address bit 31 is at L23TagLo[31].

Figure 2.45 L23TagLo Register (Tag Accesses)

63 … 40	39 … 32
0	Tag

31 … 14	13 … 9	8	7	6	5	4 … 0
Tag	0	TP	V	D	L	Parity

Table 2.56 L23TagLo Register (Tag Accesses) Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
0	63:40	Reserved. Write as zero. Ignored on reads.	R	0
Tag	39:14	Tag.	R/W	Undefined
0	13:9	Reserved. Write as zero. Ignored on reads.	R/W	Undefined
TP	8	Total Parity.	R/W	Undefined
V	7	Valid.	R/W	Undefined
D	6	Dirty.	R/W	Undefined
L	5	Lock.	R/W	Undefined
Parity	4:0	Parity.	R/W	Undefined

Figure 2.46 L23TagLo Register (WS Accesses)

63 … 32
0

31 … 24	23 … 16	15 … 9	8 … 0
DP	D	LRU	0

Table 2.57 L23TagLo Register (WS Accesses) Field Descriptions

Fields				
Name	Bits	Description	Read/ Write	Reset State
0	63:32	Reserved. Write as zero. Ignored on reads.	R/W	Undefined
DP	31:24	Dirty Parity.	R/W	Undefined
D	23:16	Dirty.	R/W	Undefined
LRU	15:9	LRU algorithm. For Cache-Ops that access the LRU field, the associativity impacts the number of LRU bits present and how they affect line replacement and refill. The P6600 core supports an 8-way set associative L2 cache. The 8-way configuration uses all bits of the LRU field (15:9), but since it is a pseudo-LRU algorithm, the value of the LRU field does not directly correspond to the least-to-most order of the 8 ways.	R/W	Undefined
0	8:0	Reserved. Write as zero. Ignored on reads.	R/W	Undefined

2.2.5.8 Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)

The *L23DataLo* register is a register that acts as the interface to the L2 or L3 cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *L23DataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *L23DataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction.

The core can be configured without L2/L3 cache support. In this case, this register will be a read-only register that reads as 0.

On P6600 family cores, test software can read or write cache data using a `cache` index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the `cache` instruction.

Figure 2.47 L23DataLo Register Format

31 … 0
DATA

Table 2.58 L23DataLo Register Field Description

Fields				
Name	Bit(s)	Description	Read / Write	Reset State
DATA	31:0	Low-order data read from the cache data array.	R/W	Undefined

2.2.5.9 Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)

On P6600 family cores, test software can read or write cache data using a **`cache`** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 2.48 L23DataHi Register Format

31 … 0
DATA

Table 2.59 L23DataHi Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DATA	31:0	High-order data read from the cache data array.	R/W	Undefined

2.2.5.10 ErrCtl (CP0 Register 26, Select 0)

Most of the fields of this register are for test software only. The MIPS64 architecture defines this register as implementation-dependent, but most CPUs put the parity-enable control in the top bit. So running OS software is well advised to set this register to `0x8000.0000` to enable cache parity checking, or to zero to disable parity checking.

Figure 2.49 Error Control Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 12	11 … 4	3 … 0
PE	PO	WST	0	PCO	0	LBE	WABE	L2P	PCD	DYT	SE	FE	0	PI	PD

Table 2.60 Field Descriptions for ErrCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
PE	31	This bit is set to 1 to enable cache parity checking and is encoded as follows: 0: Parity disabled 1: Parity enabled	R/W	0
PO	30	Parity Overwrite. Set 1 to set the parity bit regardless of parity computation, which is only for diagnostic/test purposes. After setting this bit you can use **`cache IndexStoreTag`** to set the cache data parity to the value currently in *PI* (for I-cache) or *PD* (for D-cache), while the tag parity is forcefully set from *ITagLo*/*DTagLo*. 0 = User calculated parity 1 = Override calculated parity	R/W	0
WST	29	Write to 1 for test mode for **`cache IndexLoadTag`**/ **`cache IndexStoreTag`** instructions, which then read/write the cache's internal *way-selection RAM* instead of the cache tags.	R/W	0
0	28	Reserved. Write as zero. Ignored on reads. This bit should never be set.	R/W	0

Table 2.60 Field Descriptions for ErrCtl Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
PCO	27	Precode override. Used for diagnostic/test of the instruction cache. When this bit is set, then the precode values in the *ITagHi* register are used instead of the hardware generated precode values. This applies to index store data cacheop operations.	R/W	0
0	26	Reserved. Write as zero. Ignored on reads.	R	0
LBE	25	Indicates whether a bus error (the last one, if there's been more than one) was triggered by a load or a write-allocate respectively. A write-allocate is where a cacheable write has missed in the cache, and the cache has read the line from memory. Where both a load and write-allocate are waiting on the same cache-line refill, both could be set. These bits are "sticky", remaining set until explicitly written zero.	R/W0	Undefined
WABE	24		R/W0	Undefined
L2P	23	L2 cache parity enable. Indicates whether parity is enabled on the L2Cache if present. If the L2 cache is not present, this bit has no meaning. 0: L2 cache present, L2 parity disabled 1: L2 cache present, L2 parity enabled	R/W	0
PCD	22	Precode Disable. When set, **`cache IndexStoreTag`** instructions do not update the corresponding precode field and precode parity in the instruction cache tag array.	R/W	0
DYT	21	Setting this bit allows **`cache`** load/store data operations to work on the "dirty array" — the slice of cache memory which holds the "dirty"/"stored-into" bits.	R/W	0
SE	20	Indicates that a second cache or TLB error was detected before the first error was processed. This is an unrecoverable error. This bit is set when a cache error is detected while the *FE* bit is set. This bit is cleared on reset or when a cache error is detected with *FE* cleared.	R	0
FE	19	Indicates that this is the first cache or TLB error and therefore potentially recoverable. Error handling software should clear this bit when the error has been processed. This bit is set by hardware and cleared by software on reset. Refer to the *SE* bit description for implications of this bit. Note that software can only write a 0 to this bit. A write value of 1 will not have any effect.	R	0
0	18:12	Reserved. Write as zero. Ignored on reads.	R	0
PI	11:4	Parity bits per double-word (two instructions) of data being read/written to the instruction cache data when the *PO* bit is set. During a read of IDataHi and IDataLo registers, the parity bits are stored here. This field is updated by hardware on every instruction fetch and also during a CacheOp store. During a CacheOp store, this field can be used for instruction cache data parity error injection apart from the Instruction cache store index. During a CacheOp read, this field can be used to check/read the instruction cache parity bits and also for storing the parity bits when an index load tag is executed.	R/W	Undefined
PD	3:0	Parity bits being read/written to the data cache when *PO* is set.	R/W	0x0

2.2.5.11 Cache Error — CacheErr (CP0 Register 27, Select 0)

Read-only register used to analyze the details of a parity error. The FTLB parity error sets the EREC field to ‘b11, and also sets either the ED or ET bits indicating a data or tag parity error (not both) and then updates the index and way fields. The other bits are left as 0. Note that the index field contains the FTLB set and not the index value from the Index CP0 register.

Figure 2.50 CacheErr Register Format

31 30	29	28	27	26	25	24	23	22	21 19	18	17	16 0
EREC	ED	ET	ES	EE	EB	EF	0	EW	Way	DR	0	Index

Table 2.61 Field Descriptions for CacheErr Register

Name	Bit(s)	Description	Read/ Write	Reset State
EREC	31:30	This 2-bit field indicates the block where the error occurred and is encoded as follows: 00: L1 instruction cache error 01: External cache error 10: L1 data cache error 11: FTLB parity error The FTLB parity error sets the *EREC* field to ‘b11, and sets either the *ED* or *ET* bits indicating a data or tag parity error (not both). It also updates the *Index* (bits 16:0) and *Way* (bits 21:19) fields. The other bits are left as 0. Note that the index field contains the FTLB set and not the index value from the CP0 Index register.	R	Undefined
ED	29	The encoding of these two bits depends on the state of the EREC field above. If the state of this field contains an encoding of 00, 01, or 10, indicating a cache error, the encoding of this field is as shown below. 00: No tag or data RAM error detected 01: Primary tag RAM error 10: Data RAM error 11: Duplicate tag RAM error A parity error in the FTLB tag sets the ET bit (28), while a parity error in the FTLB data sets the ED bit (29). One or both of these bits may be set.	R	Undefined
ET	28		R	Undefined
ES	27	Error source. In a multi-core system, this bit reads 0 if the error was caused by one of the cores and 1 if the error was caused by an external snoop request. In a single-core system, this bit is not used.	R	Undefined
EE	26	Error external: In a multi-core system, this bit indicates that a parity error was seen on a coherent L1 cache in another CPU. In a single-core system, this bit is not used.	R	Undefined
EB/EM	25	If *EREC* equals 0 indicating an error in the L1 cache, this bit is *EB,* indicating an error in Both caches. If data and instruction-fetch errors are reported on the same instruction, it is unrecoverable. If so, the rest of the register reports on the instruction-fetch error. If *EREC* equals 1, indicating an error in the L2 cache, this bit is *EM,* indicating there are errors in multiple locations in the cache.	R	Undefined

Table 2.61 Field Descriptions for CacheErr Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
EF	24	Unrecoverable (fatal) error (other than the *EB* type above). Some parity errors can be fixed by invalidating the cache line and relying on good data from memory. However, if this bit is set, it indicates the error cannot be fixed. Here are some possible scenarios of when the EF bit might be set by hardware: • Dirty parity error in dirty line being displaced from cache • Line being displaced from cache has a tag parity error. • The line being displaced from cache tag indicates it has been written by the CPU since it was obtained from memory (the line is "dirty" and needs a write-back), but it has a data parity error. • Writeback store miss and $CacheErr_{EW}$ error. • At least one more cache error happened concurrently with or after this one, but before the original error reached the cache error exception handler. • If *EREC* equals 0, and a second L2 error occurs when an earlier L2 error is pending.	R	Undefined
0	23	Reserved. Write as zero. Ignored on reads.	R	Undefined
EW	22	Parity error on way-selection RAM array.	R	Undefined
Way	21:19	If *EREC* equals 0, bit 19 is unused. Bits 21:20 indicate the way-number of the cache entry where the error occurred. If *EREC* equals 1, indicating an L2 or higher-level cache error, bits 21:19 indicate the way-number of the cache entry where the error occurred. On a FTLB error, bits 20:19 indicate the number of ways in each set. Bit 21 is not used on a FTLB error.	R	Undefined
DR	18	A 1 bit indicates that the reported error affected the cache-line "dirty" bits. This bit is only meaningful in case of an L1 data cache access.	R	Undefined
Index	16:0	The cache index or Scratchpad RAM index of the double word entry where the error occurred. The way of the faulty cache is written by hardware in the Way field. The CacheErr bits [16:0] represents the Address index bits [19:3]. The index-type `cache` instruction will need an "index" with the way bits glued on top of this cache-entry field; you know how to put that together, because the shape of the cache is defined in the *Config1-2* registers. On a TLB error, this field indicates the number of sets in the FTLB. The number of bits is implementation dependent and is always right-justified in the Index field.	R	Undefined

2.2.6 Shadow Control Registers

Although the P6600 Multiprocessing System does not support thread contexts or shadow registers, the Shadow Register Set Control (SRSCtl) register is implemented to allow software to read this register to determine that shadow registers are not implemented.

2.2.6.1 SRSCtl Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor.

Figure 2.51 SRSCtl Register Format

31 30	29 26	25 0
0	HSS	0

Table 2.62 SRSCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Must be written as zeros; returns zero on read.	0	0
HSS	29:26	Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented. Possible values of this field for the P6600 core are: 0x0: One shadow register set present 0x1 - 0xF: Reserved	R	Preset
0	25:0	Must be written as zeros; returns zero on read.	0	0

2.2.7 Performance Monitoring Registers

This section contains the following performance monitoring registers.

- Section 2.2.7.1, "Performance Counter Control 0 - 3 — PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6)" on page 123
- Section 2.2.7.2, "Performance Counter 0 - 3 — PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7)" on page 132

2.2.7.1 Performance Counter Control 0 - 3 — PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6)

Cores in the P6600 family provide four performance counters that provide the capability to count events or cycles for use in performance analysis. Each performance counter consists of a pair of registers: a 32-bit control register (*PerfCtl*) and a 32-bit counter register (*PerfCnt*).

Performance counters can be configured to count implementation-dependent events or cycles under a specified set of conditions that are determined by the performance counter's control register. The counter register increments once for each enabled event; when the most-significant bit of the counter register is a one (the counter overflows), and the counter is enabled, the performance counter optionally requests an interrupt.

The *IE* flag in the performance counter control register is used to enable an interrupt to be signalled when bit 31 of the corresponding counter overflows. The OR of all the performance counter register interrupts becomes the CPU output *SI_PCI*, which is typically fed back into an interrupt input, conventionally identified by $IntCtl_{IPPCI}$. However, systems using more sophisticated interrupt controllers may feed the performance counter interrupt into the interrupt controller.

Figure 2.52 PerfCtl0-3 Register Format

31	30	29 25	24 23	22 13	12 5	4	3	2	1	0
M	W		EC	0	Event	IE	U	S	K	EXL

Table 2.63 Field Descriptions for PerfCtl0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Set to 1 if there is another *PerfCtl* register after this one. This field is set for *PerfCtl0-2* and cleared on *PerfCtl3*.	R	1 for PerfCnt 0 - 2 0 for PerfCnt 3
W	30	Specifies the width of the corresponding Counter register as follows: 0: 32-bit counter width 1: 64-bit counter width	R	0
0	29:25	Reserved. Must be written as zeros; returns zeros on reads.	R	0
EC	24:23	Event Class. Root only. Reserved, read-only 0 in all other contexts. The P6600 may detect the existence of this feature by writing a non-zero value to the field and reading. If value read is 0, then EC is not supported. This field is encoded as follows: 00: Root events counted (default). Active in Root context. 01: Root intervention events counted, Active in Root context. 10: Guest events counted. Active in Guest context. 11: Guest events plus Root intervention events counted. Active in Guest context. Root will only assign encoding if it wants to give Guest visibility into Root intervention events. Root events are those that occur when $GuestCtl0_{GM} = 0$. Root intervention events are those that occur when $GuestCtl0_{GM} = 1$ and $!(Root.Status_{EXL} = 0$ and $Root.Status_{ERL} = 0$ and $Root.Debug_{DM} = 0)$ Guest events are those that occur when $GuestCtl0_{GM} = 1$ and $Root.Status_{EXL} = 0$ and $Root.Status_{ERL} = 0$ and $Root.Debug_{DM} = 0$ For the case of root intervention mode, $PerfCtl_{U/S/K/EXL}$ are ignored as $Root.Status_{EXL}=1$ and root must be in kernel mode. An implementation must qualify existing performance counter events with the value of EC. For example, if an event is “Instructions Graduated” and EC = 0, then only instructions graduated in root mode are counted.	R/W	Undefined
0	22:13	Reserved. Must be written as zeros; returns zeros on reads.	R	0
Event	12:5	Determines which event to count. Available events are listed in Table 2.64, "Performance Counter Events and Codes".	R/W	Undefined
IE	4	Set to cause an interrupt when the counter overflows into bit 31. This caneither be used to implement an extended count or (by presetting the counter appropriately) to notify software after a certain number of events have occurred.	R/W	0
U	3	Count events in User mode. When this bit is set, events can be counted in User mode.	R/W	Undefined
S	2	Count events in Supervisor mode. When this bit is set, events can be counted in Supervisor mode.	R/W	Undefined
K	1	Count events in Kernel mode. When this bit is set, events can be counted in Kernel mode.	R/W	Undefined

Table 2.63 Field Descriptions for PerfCtl0-3 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
EXL	0	Count events in Exception mode. When this bit is set, events can be counted in Exception mode (when *StatusEXL* is set).	R/W	Undefined

Table 2.64 provides a list of performance counter events as encoded into the *Event* field in bits 12:5. Note that events 128 and above are root intervention events, meaning they are only counted if PerfCtl[0-3].EC = 2'b01 of 2'b11. Hypercall instructions are also included when EC = 2'b01 or 2'b11. These events are not visible when EC = 2'b10.

Table 2.64 Performance Counter Events and Codes

Event Number	Counter 0/2	Counter 1/3
0	Cycles	
1	Instructions graduated	
2	`jr $31` (return) instructions whose target is predicted.	`jr $31` (return) predicted but guessed wrong.
3	Cycles where no instruction is fetched because it has no "next address" candidate. This includes stalls due to register indirect jumps such as `jr,` stalls following a `wait` or `eret` Redirect Stall cycles due to: • Stalls due to register indirect jumps including non-predicted JR $31. • Stalls due to ERET, WAIT instructions. • Stalls due to IFU determined exception. and stalls dues to exceptions from instruction fetch	`jr $31` (return) instructions fetched and **not** predicted using RPS
4	ITLB accesses.	ITLB misses, which result in an MMU access. ITLB misses seen at the ID stage (this is the same for MMU instruction accesses). It is possible that a pending ITLB is killed before accessing the MMU.
5	Reserved	Reserved
6	Instruction Cache accesses. P6600 cores have a 128-bit connection to the I-cache and fetch 4 instructions every access. This counts **every** such access, including accesses for instructions which are eventually discarded. For example, following a branch which is incorrectly predicted, the P6600 core will continue to fetch instructions, which will eventually get thrown away.	Instruction cache misses. Includes misses resulting from fetch-ahead and speculation.
7	Cycles where no instruction is fetched because we missed in the I-cache. I-cache miss stall cycles. This includes the cycles where the IFU state machine for a given TC is in the miss state. It is possible that multiple TCs requesting the same line will all count the same miss cycles.	Number of fetches restricted due to MAAR.
8	Uncached Instruction Fetch stall cycles. Cycles where no instruction is fetched because we are waiting for an I-fetch from uncached memory.	Reserved

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
9	Number of IFU fetch stalls due to lack of credits on the IBUF interface.	Valid fetch slots killed due to taken branches/jumps or stalling instructions.
10	Reserved in single-core environments In a multi-core environment, store misses transitioning to I->M or S->M	Reserved in single-core environments In a multi-core environment, load misses transitioning to I->S or I->E
11	Cycles IFU-IDU gate is closed due to mispredicted branch. This counts the time from when IEU closes the gate to when GRU opens.	Cycles IFU-IDU gate is open but no instructions fetched by IFU. May be overridden by changing *Config6.IFUPerfSel* field. See Table 2.9, "Field Descriptions for Config6 Register" for a description of the other overloading events.
12	Cycles IFU-IDU gate is closed due to other reasons: • MTC0/MFC0 sequence in pipe • EHB • DD_DR_DS is blocked	Reserved in single-core environments. In a multi-core environment, intervention hits.
13	Number of cycles where no instruction is inserted in DDQ0 because it is full.	Number of cycles where no instruction is inserted in DDQ1 because it is full.
14	Number of cycles where no instructions can be issued because there are no completion buffer ID's.	Reserved.
15	Reserved.	Cycles where no instructions can be added to the issue pool, because we have filled the coprocessor 1's shelves used for coprocessor 1 instructions.
16 - 17	Reserved	Reserved
18	Cycles when three instructions are issued.	Cycles when four instructions are issued.
19	Reserved	Reserved
20	Cycles when only one instruction is issued.	Cycles when two instructions are issued.
21	Number of **`jr`** (not **`$31`**) instructions mispredicted at graduation.	Number of **`jr $31`** instructions graduated.
22	Number of graduated JAR/JALR.HB	D-cache line refill (not LD/ST misses)
23	Counts the number of speculative loads. Pairs of loads or stores that are bonded count as one.	Speculative data cache accesses and instruction cache Cacheops. Pairs of loads or stores that are bonded count as one.
24	Number of data cache misses at graduation.	D-cache misses. This count is per instruction at graduation and includes load, store, prefetch, **`synci`** and address based cacheops.
25	JTLB translation fails on d-side (data side as opposed to instruction side) accesses. This pertains to graduated instructions only.	Reserved
26	Load/store instruction redirects, which happen when the load/store follows too closely on a possibly matching cacheop. Load/Store generated replays - typically, a load following a CacheOp that has matches the Index match of the CacheOp.	Reserved

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
27	LSGB graduation blocked cycles. Reasons for block: • CP1/2 store data not ready • SYNC, SYNCI at the head • **`sc`** at the head • CACHEOP at the head FSB, LDQ, WBB, or ITU FIFO full.	LSGB graduation that does not result in a request going out on the bus. Reasons include: • Misses at integer pipe graduation turn into hit. • Miss merges with outstanding fill request.
28	L2 cache writebacks	L2 cache accesses
29	L2 cache misses	L2 cache miss cycles
30	Cycles Fill Store Buffer (FSB) are full and cause a pipe stall	Cycles Fill Store Buffer (FSB) > 1/2 full
31	Cycles Load Data Queue (LDQ) are full and cause a pipe stall	Cycles Load Data Queue (LDQ) > 1/2 full
32	Cycles Writeback Buffer (WBB) are full and cause a pipe stall	Cycles Writeback Buffer (WBB) > 1/2 full
33	Not used in single-core environments. In a multi-core environment, counts requests that will receive data from the Coherence Manager.	Not used in single-core environments. In a multi-core environment, request latency to first data word of data from the Coherence Manager.
34	Reserved in single-core environments. In a multi-core environment, invalidate intervention hits.	Reserved in single-core environments. In a multi-core environment, all invalidate interventions.
35	Replays following optimistic issue of instruction dependent on load which missed. Counted only when the dependent instruction graduates. Reserved.	Floating Point Load instructions graduated.
36	`jr` (not `$31`) instructions graduated.	`jr $31` mispredicted at graduation.
37	Integer Branch instructions graduated.	Floating Point Branch instructions graduated.
38	Branch likely instructions graduated.	Mispredicted Branch likely instructions graduated.
39	Conditional branches graduated.	Mispredicted Conditional branches graduated.
40	Integer instructions graduated (includes **`nop, ssnop, ehb`** as well as all arithmetic, logic, shift and extract type operations).	Floating Point instructions graduated (but not counting Floating Point load/store).
41	Loads graduated. Bonded load/store counted as 2.	Stores graduated. Bonded load/store counted as 2.
42	**`j/jal`** graduated.	Reserved.
43	no-ops graduated.	integer multiply/divides graduated.
44	Reserved	Reserved
45	Reserved	Reserved
46	Uncached loads graduated.	Uncached stores graduated.
47	Reserved in single-core environments. In a multi-core environment, writebacks due to evictions.	Reserved in single-core environments. In a multi-core environment, writebacks due to any reason.
48	Reserved in single-core environments. In a multi-core environment, count of all invalidates (M,E,S)->I	Reserved in single-core environments. In a multi-core environment, count of transitions from (I,S)->E.

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
49	EJTAG instruction triggers.	EJTAG data triggers.
50	CP1 branches mispredicted.	Reserved
51	`sc` instructions graduated.	`sc` instructions failed.
52	`prefetch` instructions graduated at the top of LSGB.	`prefetch` instructions which did nothing, because they hit in the cache.
53	Cycles where no instructions graduated.	Cacheable load misses in TI. Includes floating point and fast path loads.
54	Cycles where one instruction graduated.	Cycles where two instructions graduated.
55	GFifo blocked cycles.	Floating point stores graduated.
56	GFifo blocked due to TLB or Cacheop.	Number of cycles no instructions graduated from the time the pipe was flushed because of a replay until the first new instruction graduates. This is an indicator graduation bandwidth loss due to replay. Often times this replay is a result of event 25 and therefore an indicator of bandwidth lost due to cache miss.
57	Mispredicted branch instruction graduations without the delay slot (in the same cycle).	Cycles waiting for delay slot to graduate on a mispredicted branch.
58	Exceptions taken.	Replays initiated from graduation.
59	Indicates the load/store graduation buffer (LSGB) is full.	Indicates the load/store graduation buffer (LSGB) is half full.
60	Reserved in single-core environments. In a multi-core environment, state transition from S->M (coherent and non-coh).	Reserved in single-core environments. In a multi-core environment, state transitions from (M,E)->S.
61	Reserved in single-core environments. In a multi-core environment, request latency to self-intervention.	Reserved in single-core environments. In a multi-core environment, count of requests that will receive self-intervention.
62	Prediction buffer full causing IFU stall.	Reserved.
63	L2 single-bit errors detected.	Reserved in single-core environments. In a multi-core environment, all interventions.
64	SI_Event[0] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[0]* pin to an event to be counted.	SI_Event[1] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[1]* pin to an event to be counted.
65	SI_Event[2] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[2]* pin to an event to be counted.	SI_Event[3] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[3]* pin to an event to be counted.
66	SI_Event[4] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[4]* pin to an event to be counted.	SI_Event[5] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[5]* pin to an event to be counted.
67	SI_Event[7] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[7]* pin to an event to be counted.	SI_Event[8] - Implementation-specific system event. The system integrator of the P6600 core may connect the *SI_PCEvent[8]* pin to an event to be counted.
68	All OCP requests accepted.	All OCP cacheable requests accepted.
69	OCP read requests accepted.	OCP cacheable read requests accepted.

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
70	OCP write requests accepted.	OCP cacheable write requests accepted.
71	Reserved	OCP write data sent.
72	Reserved	OCP read data received.
73	Reserved in single-core environments. In a multi-core environment, OCP Intervention write data stalled (valid but not accepted).	Reserved in single-core environments. In a multi-core environment, OCP Intervention write data valid (accepted or not).
74	Cycles Fill Store Buffer (FSB) < 1/4 full.	Cycles Fill Store Buffer (FSB) 1/4 to 1/2 full.
75	Cycles Load Data Queue (LDQ) < 1/4 full.	Cycles Load Data Queue (LDQ) 1/4 to 1/2 full.
76	Cycles Writeback Buffer (WBB) < 1/4 full.	Cycles Writeback Buffer (WBB) 1/4 to 1/2 full.
77	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect without IFU predecode-based prediction, causing a redirect or replay. Measures the number of true hits for the Return Prediction Stack (RPS) portion of the L1BTB.	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect without IFU predecode-based prediction causing a redirect or replay. Measures the number of true hits for the branch portion of the L1BTB.
78	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect with IFU predecode-based prediction causing a redirect or replay. Measures the number of mis-predicts for the Return Prediction Stack (RPS) portion of the L1BTB.	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect with IFU predecode-based prediction causing a redirect or replay. Measures the number of mis-predicts for the branch portion of the L1BTB.
79	Counts the number of writes to the Return Prediction Stack (RPS) portion of the L1 Branch Target Buffer (L1BTB) with no L1BTB hit (cold miss).	Counts the number of writes to the branch portion of the L1 Branch Target Buffer (L1BTB) with no L1BTB hit (cold miss).
80	Number of L1 Branch Target Buffer masked hits due to lack of credit for DS.	Number of L1 Branch Target Buffer masked hits due to lack of credit for target.
81	Number of NFW or L1 Branch Target Buffer mis-predicts for instruction cache way-hit prediction.	Reserved
82 - 83	Reserved	Reserved
84	Counts the number of times a Write-Back Buffer (WBB) entry is newly allocated for an Uncached Accelerated (UCA) store and there is one UCA store already active in the WBB.	Counts the number of times a Write-Back Buffer (WBB) entry is newly allocated for an Uncached Accelerated (UCA) store and there are two UCA stores already active in the WBB.
85	Number of times an uncached instruction arrives at BIU while there is an actively gathering UCA buffer.	Reserved
86	Reserved	Reserved
87	Number of stall cycles due to the lack of load/store queue (LSQ) ID.	Number of stall cycles due to the lack of IID.
88	Reserved.	Reserved.
89	Number of cycles when no FP instructions are dispatched.	Number of cycles when no integer instructions are dispatched.
90	Number of cycles when one FP instruction is dispatched.	Number of cycles when one integer instruction is dispatched.
91	Number of cycles when two FP instructions are dispatched.	Number of cycles when two integer instructions are dispatched.

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
92 - 93	Reserved	Reserved
94	Number of cycles when three instructions are issued.	Number of cycles when four instructions are issued.
95 - 96	Reserved	Reserved
97	Number of instructions issued on AGU port from DDQ1.	Number of instructions issued on BSU port from DDQ1.
98	Number of instructions issued on MDU/ALU2 port from DDQ1.	Number of instructions issued on ALU1 port from DDQ0.
99	Number of DTLB accesses (speculative).	Number of DTLB misses (speculative).
100	Data side hits in the VTLB/FTLB. This includes FTLB and VTLB hits and unmapped region accesses.	Instruction side hits in the VTLB/FTLB. This includes FTLB and VTLB hits and unmapped region accesses.
101	Number of data side hits in the VTLB/FTLB in an unmapped region.	Number of instruction side hits in the VTLB/FTLB in an unmapped region.
102	Number of instruction side hits in the VTLB.	Number of instruction side hits in the FTLB.
103	Number of data side hits in the VTLB.	Number of data side hits in the FTLB.
104	Number of TLBWR writes to the VTLB.	Number of TLBWR writes to the FTLB.
105	Number of DTLB hits to the half of *EntryLo* that caused a fill (speculative).	Number of DTLB hits to the half of *EntryLo* that did not cause a fill (speculative).
106	Number of pairs of bonded stores at graduation.	Number of pairs of bonded loads at graduation.
107	Reserved	Speculative count of ‘over-eager’ loads that hit a store without the data being available.
108	Number of times a load is not issued because it is tagged by the ‘over-eager’ predictor.	Reserved
109	Speculative count of incorrectly bonded loads and stores.	Reserved
110	Number of misaligned loads that graduated.	Number of misaligned stores that graduated.
111 - 112	Reserved	Reserved
113	Number of cycles where one FP/MSA opcode is issued.	Number of cycles where FPU/MSA sent F2I strobes.
114	Number of cycles where two FP/MSA opcodes are issued.	Number of cycles where FPU/MSA received I2F strobes
115	Number of data-side unmapped XKPhys accesses.	Number of instruction-side unmapped XKPhys accesses.
116	Number of cycles where one FP/MSA opcode is retired.	Number of cycles where FPU/MSA received I2F load strobes.
117	Number of cycles where two FP/MSA opcodes are simultaneously retired.	Number of cycles where FPU/MSA received I2F bonded load strobes.
118	Number of cycles where FPU/MSA shelf is full.	Number of cycles where FPU/MSA slowly returning credits.
119	Number of load and stores graduated with VA[13:12] != PA[13:12]. Misaligned stores counted as two.	Reserved
120	Number of Number of noRFO stores graduated.	Number of times noRFO detected.

Table 2.64 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
121	Number of refetches for integer misaligned instructions.	Number of refetches for MSA misaligned instructions.
122	Number of doubleword bonded speculative loads.	Number of doubleword bonded speculative stores.
123	Number of quadword bonded speculative loads.	Number of quadword bonded speculative stores.
124 - 125	Reserved	Reserved
126	Hardware table walker (HTW) abort due to HTW access denied to XKSeg (XK = 0).	Hardware table walker (HTW) abort due to HTW access denied to XSSeg (XS = 0).
127	Reserved	Reserved
128	Number of root exceptions taken in guest mode.	Number of guest mode to root mode transitions.
129	Number of GSFC exceptions.	Number of GHFC exceptions.
130	Number of GPSI exceptions.	Number of GRIR exceptions.
131	Number of Hypercall exceptions.	Number of guest-related root TLB exceptions taken when GuestCtl0.GExcCode = GVA.
132	Number of root TLB exceptions caused by instruction-side guest translation requests.	Number of root TLB exceptions caused by data-side guest translation requests.
133	Number of root writes that set the Guest.Cause.TI bit to 1.	Number of root writes to Guest.PerfCnt that set the Guest.Cause.PCI bit to 1.
134	Number of guest accesses to the Watch registers that cause GPSI when virtually shared.	Number of guest accesses to the PerfCnt and PerfCtl registers that cause GPSI when virtually shared.
135	Number of interrupts that cause a guest exit in EIC mode.	Number of interrupts that cause a guest exit in non-EIC mode.
136	Number of data side hardware page table walks aborted due to an exception or branch mispredict related to an older instruction.	Number of instruction side hardware page table walks aborted due to an exception or branch mispredict related to an older instruction.
137	Number of instruction or data side hardware page table walks aborted because a related table walk load has missed in the main TLB.	Number of instruction or data side hardware page table walks aborted because a related table walk load has caused an exception, including a TLB refill.
138	An instruction or data side hardware page table walk has been initiated.	Reserved
139	Number of dependent instructions replayed in ALU2/MDU due to load miss.	Number of dependent instructions replayed in CTI pipe due to load miss.
140	Number of dependent instructions replayed in ALU1 pipe due to load miss.	Number of dependent instructions replayed in AGEN pipe due to load miss.
138 - 255	Reserved	Reserved

2.2.7.2 Performance Counter 0 - 3 — PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7)

General purpose event counters, which operate as directed by *PerfCtl0-3*.

Figure 2.53 Performance Counter 0 - 3 Register

31 ... 0
Counter

Table 2.65 Performance Counter 0 - 3 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Counter	31:0	Counter value.	R/W	Undefined

2.2.8 Debug Registers

This section contains the following debug registers.

- Section 2.2.8.1, "Debug (CP0 Register 23, Select 0)" on page 132
- Section 2.2.8.2, "Debug Exception Program Counter — DEPC (CP0 Register 24, Select 0)" on page 135
- Section 2.2.8.3, "Debug Save — DESAVE (CP0 Register 31, Select 0)" on page 136
- Section 2.2.8.4, "Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)" on page 136
- Section 2.2.8.5, "Watch High 0 - 3 — WatchHi0-3 (CP0 Register 19, Select 0-3)" on page 137

2.2.8.1 Debug (CP0 Register 23, Select 0)

The *Debug* register provides control and status information while in debug mode. During normal operation (non-debug mode), this register may not be written at all, and only the *DM* bit and the *EJTAGver* field returns valid data.

The read-only bits are updated by hardware every time the debug exception is taken, or when a normal exception is taken when already in debug mode (a "nested exception"). Not all fields are valid in both circumstances: *Halt* and *Doze* are not defined after a nested exception, and the nested-exception-type field *DExcCode* is undefined from a debug exception.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT* are updated on both debug exceptions and on exceptions in debug modes
- *DExcCode* is updated on exceptions in debug mode, and is undefined after a debug exception
- *Halt* and *Doze* are updated on a debug exception, and are undefined after an exception in debug mode
- *DBD* is updated on both debug and on exceptions in debug modes

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g. *EJTAGver* and *DM*.

Figure 2.54 Debug Register Format

31	30	29	28	27	26	25	24	23	22	21	20
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI

19	18	17 15	14 10	9	8	7 6	5	4	3	2	1	0
DDBSImpr	DDBLImpr	EJTAGver	DExcCode	NoSSt	SSt	0	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 2.66 Field Descriptions for Debug Register

Name	Bit(s)	Description	Read/ Write	Reset State
DBD	31	Indicates if the last debug exception or exception in debug mode occurred in a branch delay slot. 0: Not in delay slot 1: In delay slot When set to 1, the Debug Exception Program Counter (*DEPC)* points to the branch instruction, which is usually the correct place to restart.	R	Preset
DM	30	Indicates if the processor is operating in debug mode. 0: Processor is operating in non-debug mode 1: Processor is operating in debug mode In debug mode, this bit is set on any debug exception and is cleared by **`deret`**.	R	0
NoDCR	29	Indicates if the dseg memory segment and a memory-mapped *DCR* register is present. 0: dseg address space is present 1: dseg address space is not present	R	0
LSNM	28	Controls access of load/store between dseg and main memory. 0: Load/stores in dseg address range goes to dseg 1: Load/stores in dseg address range goes to main memory Setting this bit causes debug-mode accesses to dseg addresses to be sent to system memory. This makes most of the EJTAG unit's control systems unavailable, so will probably only be done around a particular load/store.	R/W	0
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred. 0: Processor not in low power mode when debug exception occurred 1: Processor in low power mode when debug exception occurred Before the debug exception, CPU was in one of the reduced power mode.	R	0
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred. 0: Internal system bus clock running 1: Internal system bus clock stopped Before the debug exception, the CPU was stopped — probably asleep following a **`wait`** instruction.	R	1
CountDM	25	Controls or indicates the Count register behavior in debug mode. 0: Count register stopped in debug mode 1: Count register is running in debug mode	R/W	1

Table 2.66 Field Descriptions for Debug Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
IBusEP	24	These "pending exception" flags remember exception events caused by instructions run in debug mode, but which have not yet occurred because they are imprecise and $Debug_{IEXI}$ is set. Note that you can write a 1 to any of these at any time, so they survive writes to the whole *Debug* register; but a write of zero to a field is ignored. They remain set until $Debug_{IEXI}$ is cleared explicitly, or implicitly by a **`deret`**. If the **`deret`** clears the bit, the exception is taken and the pending bit cleared. *IBusEP* remembers a bus error on an instruction fetch. This exception is precise, so it cannot occur and the field is always zero. *MCheckP* machine check condition (usually an illegal TLB update). . The machine check can be either precise or imprecise depending on the type of error. Refer to the Machine Check exception in the Exception chapter for more information. *CacheEP* indicates a precise cache parity error is pending. Data access Bus Error exception Pending: *DBusEP* remembers a bus error on a data access. Set when an data bus error event occurs or if a 1 is written to the bit by software. Cleared when a Data Bus Error exception is taken by the processor, and by reset. If *DBusEP* is set when *IEXI* is cleared, a Data Bus Error exception is taken by the processor, and *DBusEP* is cleared	R	0
MCheckP	23		R	0
CacheEP	22		R/W	0
DBusEP	21		R/W	0
IEXI	20	Imprecise Error eXception Inhibit. Set to 1 to defer imprecise exceptions. By default, this bit is set on entry to debug mode and cleared on exit. The deferred exception returns when and if this bit is cleared, and until then the occurrence of the imprecise exception can be observed in the “pending exception” flags described in bits 24:21 above.	R/W	0
DDBSImpr	19	Imprecise store breakpoint. *DEPC* probably points to an instruction some time later in the sequence than the store which triggered the breakpoint.	R	Preset
DDBLImpr	18	Imprecise load breakpoint. *DEPC* probably points to an instruction some time later in the sequence than the store which triggered the breakpoint. The debugger or user (or both) have to cope as best they can.	R	Preset
EJTAGver	17:15	These read-only bits encode the revision of the EJTAG specification to which this implementation conforms. The legal values are. 110: Version 6.0 All other values are reserved.	R	6
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. Following initial entry to debug mode, this field is undefined. The subsequent value will be one of those defined in $Cause_{ExcCode}$. See Table 2.39 for a list of values. Value is undefined after a debug exception.	R	Preset
NoSSt	9	Indicates whether the single-step feature controllable by the *SSt* bit is available in this implementation. This read-only bit is always zero on the P6600 core because single-step is implemented.	R	0
SSt	8	Controls if debug single step exception is enabled. 0 = No debug single-step exception enabled 1 = Debug single-step exception enabled	R/W	0
R	7:6	Reserved. Must be written as zeros; returns zeros on reads.	R	0
DINT	5	Indicates that a debug interrupt exception (from EJTAG pin) occurred. Cleared on exception in debug mode. 0: No debug interrupt exception 1: Debug interrupt exception	R	Preset

Table 2.66 Field Descriptions for Debug Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
DIB	4	Instruction breakpoint. This bit is set by hardware when an instruction breakpoint occurs. 0: No debug exception breakpoint 1: Debug exception breakpoint occurred	R	Preset
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. 0: No debug data exception on a store 1: Debug instruction exception on a store	R	Preset
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. 0: No debug data exception on a load 1: Debug instruction exception on a load	R	Preset
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. 0: No debug software breakpoint exception 1: Debug software breakpoint exception	R	Preset
DSS	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode. 0: No debug single-step exception 1: Debug single-step exception	R	Preset

2.2.8.2 Debug Exception Program Counter — DEPC (CP0 Register 24, Select 0)

The 64-bit Debug Exception Program Counter (DEPC) points to the instruction to restart when a **`deret`** is executed to exit debug mode. When *$Debug_{DBD}$* is set, it means that the "real" return address is in a branch delay slot, and *DEPC* points to the preceding branch.

**Figure 2.55 DEPC Register Format **

63 … 0
DEPC

Table 2.67 DEPC Register Formats

Field		Description	Read / Write	Reset
Name	Bit(s)			
DEPC	31:0	The *DEPC* register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, then the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the **`deret`** instruction causes a jump to the address in the *DEPC*.	R/W	Preset

2.2.8.3 Debug Save — DESAVE (CP0 Register 31, Select 0)

Software-only register, with no hardware effect. Provided because the debug exception handler can't use the *k0-1* GP registers, used by ordinary exception handlers to bootstrap themselves: but a debug handler can save a GPR into *DESAVE*, and then use that GPR register in code which saves everything else.

Figure 2.56 DeSave Register Format

63 … 0
DESAVE

Table 2.68 DeSave Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DESAVE	63:0	Debug exception save contents.	SO	Undefined

2.2.8.4 Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)

Used in conjunction with *WatchHi0-3* respectively, each of these registers carries the virtual address and what-to-match fields for a CP0 watchpoint. *WatchLo0-1* are used for instruction side accesses and *WatchLo2-3* are used for data side accesses. The bit assignments for each of the *WatchLo* registers is identical. Hence, only one register is shown below.

Figure 2.57 WatchLo0-3 Register Format

63 … 3	2	1	0
VAddr	I	R	W

Table 2.69 Field Descriptions for WatchLo0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
VAddr	63:3	The address to match on, with a resolution of a doubleword.	R/W	Undefined
I	2	Accesses to match: I = Instruction fetches. This bit is always 0 in the P6600 core. R = Reads (loads) W = Writes (stores) In the P6600 core, the *I* bit of this field (bit 2) is always 0 for WatchLo registers 2 and 3, but is R/W and can be programmed for WatchLo registers 0 and 1. *WatchLo0-1*$_R$ and *WatchLo0-1*$_W$ are fixed to zero as the P6600 core does not implement load/store watches.	R/W	0
R	1		R/W	0
W	0		R/W	0

2.2.8.5 Watch High 0 - 3 — WatchHi0-3 (CP0 Register 19, Select 0-3)

These registers provide the interface to a debug facility that causes an exception if an instruction or data access matches the address specified in the registers. Watch exceptions are not taken ifthe CPU is already in exception mode (that is if $Status_{EXL}$ or $Status_{ERL}$ is already set).

Watch events which trigger in exception mode are remembered, and result in a "deferred" exception, taken as soon as the CPU leaves exception mode.

WatchHi0-1 are used for instruction side accesses and *WatchHi2-3* are used for data side accesses.

This CP0 watchpoint system is independent of the EJTAG debug system (which provides more sophisticated hardware breakpoints).

The *WatchLo0-3* registers hold the address to match, while *WatchHi0-3* hold a bundle of control fields.

Figure 2.58 WatchHi0-3 Register Format

31	30	29 28	27 24	23 16	15 12	11 3	2	1	0
M	G	WM	0	ASID	0	Mask	I	R	W

Table 2.70 Field Descriptions for WatchHi0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	The $WatchHi0\text{-}3_{M}$ bit is set whenever there is one more watchpoint register pair to find. Software can use these four bits (starting with *WatchHi0*) to determine how many watchpoints there are. This field is set for *WatchHi0-2* and cleared on *WatchHi3*.	R	1 (*WatchHi0-2*) 0 (*WatchHi3*)
G	30	If the $WatchHi0\text{-}3_{G}$ bit is set, any address that matches that specified in the corresponding *WatchLo* register causes a watch exception. If this bit is zero, the *ASID* field of the *WatchHi* register must match the *ASID* field of the *EntryHi* register to cause a watch exception.	R/W	Undefined
WM	29:28	Virtualization support. This bit is used for root management of the Watch functionality. This field is reserved and read as 0 for Guest*WatchHi*, or if such functionality is unimplemented. Software can determine existence of this feature by writing then reading this field.	R/W	0
0	27:24	Reserved. Write as zero. Ignored on reads.	R	0
ASID	23:16	$WatchHi0\text{-}3_{ASID}$ matches addresses from a particular address space (the "ASID" is like that in TLB entries) — except that you can set $WatchHi0\text{-}3_{G}$ ("global") to match the address in any address space. The match a particular address, the $WatchHi0\text{-}3_{G}$ bit is cleared and the $WatchHi0\text{-}3_{ASID}$ value is used to ensure that the match is to the correct address space. If the If the $WatchHi0\text{-}3_{G}$ bit is set, the address is always matched, regardless of the $WatchHi0\text{-}3_{ASID}$ value.	R/W	Undefined
0	15:12	Reserved. Write as zero. Ignored on reads.	R	0
Mask	11:3	Watch mask. This field marks the corresponding $WatchLo0\text{-}3_{VAddr}$ address bits to be ignored when deciding whether this is a match.	R/W	Undefined

Table 2.70 Field Descriptions for WatchHi0-3 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
I	2	Watch exception type. These bits indicate what type of access (if any) matched after a watch exception. I = Instruction fetches R = Reads (loads) W = Writes (stores) Write a 1 to any of these bits in order to *clear* it (and therefore prevent the exception from immediately happening again). This behavior is unusual among CP0 registers, but it is quite convenient: to clear a watchpoint of all the exception causes you've seen, just read the value of *WatchHi0-3* and write it back again. $\mathit{WatchHi0\text{-}1}_R$ and $\mathit{WatchHi0\text{-}1}_W$ should always read 0 and $\mathit{WatchHi2\text{-}3}_I$ should always read 0	W1C	Undefined
R	1		W1C	0
W	0		W1C	0

2.2.9 PDTrace Registers

This section contains the following MIPS PDTrace registers.

2.2.9.1 Trace Control Register — TraceControl (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

Figure 2.59 TraceControl Register Format

31	30	29	28	27	26	25	24	23	22	21	20–13	12–5	4	3	2	1	0
TS	UT	0	Ineff	TB	IO	D	E	K	S	U	ASID_M	ASID	G	TFCR	TLSM	TIM	On

Table 2.71 TraceControl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
TS	31	The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in the *TraceControl* register.	R/W	0
UT	30	This bit has been deprecated and is no longer used since there are now two explicit trace registers, *UserTraceData1* and *UserTraceData2*. This bit is tied to 0 internally.	R	0
0	29	Reserved. Must be written as zero; returns zero on read.	0	0
Ineff	28	When set to 1, core-specific inefficiency tracing is enabled, and core-specific trace information is included in the trace stream. The inefficiency code replaces an “NI” and is interpreted in the trace stream with an expanded InsComp (Instruction Completion Indicator). The InsComp is expanded from 3b to 4b for all trace formats.	R/W	0
TB	27	Trace All Branch. When set to 1, this tells the processor to trace the PC value for all branches taken, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
IO	26	Inhibit Overflow. This signal is used to indicate to the P6600 trace logic that slow but complete tracing is desired. Hence, the P6600 tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full, so that no trace records are ever lost.	R/W	Undefined
D	25	Debug mode. When set to one, this enables tracing in debug mode. For a trace to be enabled in Debug mode, the *On* bit must also be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in debug mode.	R/W	Undefined
E	24	Exception mode. When set to one, tracing is enabled in Exception mode. For a trace to be enabled in Exception mode, the *On* bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Exception Mode.	R/W	Undefined
K	23	Kernel mode. When set to one, enables tracing in Kernel mode. For a trace to be enabled in Kernel mode, the *On* bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Kernel Mode.	R/W	Undefined
S	22	Supervisor mode. When set to one, tracing is enabled in Supervisor Mode. For a trace to be enabled in Supervisor mode, the On bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Supervisor Mode, regardless of other bits. If the processor does not implement Supervisor Mode, this bit is ignored on write and returns zero on read.	R/W	Undefined

Table 2.71 TraceControl Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
U	21	User mode. When set to one, tracing is enabled in User mode. For a trace to be enabled in User mode, the *On* bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in User Mode, regardless of the setting of other bits.	R/W	Undefined
ASID_M	20:13	ASID mask. This is a mask value applied to the ASID comparison (done when the *G* bit is zero). A “1” in any bit in this field inhibits the corresponding *ASID* bit from participating in the match. As such, a value of zero in this field compares all bits of ASID. Note that the ability to mask the *ASID* value is not available in the hardware signal bit; it is only available via the software control register.	R/W	Undefined
ASID	12:5	Address space identifier. This field stores the *ASID* field to match when the *G* bit is zero. When the *G* bit is one, this field is ignored.	R/W	Undefined
G	4	Global enable. When set, tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true.	R/W	Undefined
TFCR	3	When set, this bit indicates to the PDtrace interface that the optional *Fcr* bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the function call (or return) instruction must also be traced.	R/W	Undefined
TLSM	2	Load/Store Miss trace. When set, this indicates to the PDtrace interface that information about data cache misses should be traced. If PC, load/store address, and data tracing are disabled (see the TraceControl2Mode field), the full PC and load/store address are traced for data cache misses. If load/store data tracing is enabled, the LSM bit must be traced in the appropriate trace format. Note that data cache miss information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
TIM	1	Trace IM bit. When set, this indicates to the PDtrace interface that the optional *IM* bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the instruction that missed in the I-cache must be traced. Note that instruction cache miss information is only traced if tracing is actually enabled in the current mode.	R/W	Undefined
On	0	This is the master trace enable switch in software control. When zero, tracing is always disabled. When set to one, tracing is enabled whenever the other enabling functions are also true.	R/W	0

2.2.9.2 Trace Control 2 Register — TraceControl2 (CP0 Register 23, Select 2)

The *TraceControl2* register provides additional control and status information. Note that some fields in the *TraceControl2* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB). As such, these fields in the *TraceControl2* register will not have valid values until the TCB asserts these values.

This register is only implemented if the MIPS Trace capability is present.

Figure 2.60 TraceControl2 Register Format

31 30	29 10	9 7	6 5	4	3	2 0
SyPExt	R	Mode	ValidModes	TBI	TBU	SyP

Table 2.72 TraceControl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
SyPExt	31:30	Sync period extension. Extension to the *SyP* (sync period) field for implementations that need higher numbers of cycles between synchronization events. The value of *SyP* is extended by assuming that these two bits are juxtaposed to the left of the three bits of *SyP* ($SypExt_{SyP}$). When only *SyP* was used to specify the synchronization period, the value was 2x, where x was computed from *SyP* by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits just obtained by the juxtaposition of *SyPExt* and *SyP*. Sync period values greater than 2^{31} are UNPREDICTABLE. That is all values greater than 11010 (26 + 5 = 31) are UNPREDICTABLE. With SyPExt bits, a sync period range of 25 to 2^{31} cycles can be obtained.	R/W	0
R	29:10	Reserved. Write as zero. Ignored on reads.	R	0
Mode	9:7	When tracing is turned on, these five bits specify what information is to be traced by the core. Each bit turns on tracing of a specific tracing mode when that bit value is a 1. If the corresponding bit is 0, then the corresponding trace (shown in the table below) is not traced by the processor. Each bit is this field is encoded as follows: Bit 7: PC Bit 8: Load address Bit 9: Store address	R/W	Undefined
ValidModes	6:5	This field specifies the subset of tracing that is supported by the processor. This field is encoded as follows: 01: PC and load and store address tracing only All other values are invalid.	R	2’b01
TBI	4	This bit indicates how many trace buffers are implemented by the TCB, as follows. 0: Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented. 1: Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the traces is currently written.	R	Undefined
TBU	3	This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the $TraceControl2_{SyP}$ field. 0: Trace data is being sent to an on-chip trace buffer 1: Trace Data is being sent to an off-chip trace buffer This bit is loaded from $TCBCONTROLB_{OfC}$.	R	Undefined

Table 2.72 TraceControl2 Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
SyP	2:0	The period (in cycles) to which the internal synchronization counter is reset when tracing is started, or when the synchronization counter has overflowed. This field is encoded as follows. 000: 2^5 001: 2^6 010: 2^7 011: 2^8 100: 2^9 101: 2^{10} 110: 2^{11} 111: 2^{12} This field is loaded from *TCBCONTROLA*$_{SyP}$.	R	Undefined

2.2.9.3 Trace Control 3 Register — TraceControl3 (CP0 Register 24, Select 2)

The *TraceControl3* register provides additional control and status information. This register is only implemented if the PDtrace capability is present.

Figure 2.61 TraceControl3 Register Format

31 30	29	28	27	26 23	22	21 14	13	12	11	10	9	8	7 3	3	1	0
0	UPR	0	MSA	0	GV	GuestID	PeCOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC	0	TRIDLE	TRPAD	0

Table 2.73 TraceControl3 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	31:30	Reserved. Must be written as zeros; returns zeros on reads.	R	0
UPR	29	Indicates that for 128 bit load/ stores (MSA, if tracing of 128 bit MSA ld/st is not implemented (see bit TraceControl3.MSA) and bonded 2x64) only the lower 64 bits are traced.	R	1
0	28	Reserved. Must be written as zeros; returns zeros on reads.	R	0
MSA	27	128 bit MSA load/store data trace not implemented (see the UPR bit 29).	R	0
0	26:23	Reserved. Must be written as zeros; returns zeros on reads.	R	0
GV	22	Enable trace for all GuestIDs or only 1 GuestID. 0: Trace enabled for all Guests 1: Trace enabled only for Guest specified by TCBControlEGuestID	R/W	0

Table 2.73 TraceControl3 Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
GuestID	21:14	The GuestID field to match when tracing. If GuestCtl0.G1 = 1, the number of active bits in this register field matches the number of writeable bits in GuestCtl.ID register field and the rest of the bits of this field are read-only as zero. If GuestCtl0.G1 = 0, then only the right-most bit of this register field is writeable and the rest of the bits of this field are read-only as zero. A value of 0 represents Root execution while non-zero represents Guest execution.	R/W	Undefined
PeCOvf	13	Performance counter overflow. Setting this bit enables the trace control logic to trace a performance counter overflow.	R/W	0
PeCFCR	12	Performance counter function/call return. Setting this bit enables the trace control logic to trace a function call/return condition or an exception handler entry.	R/W	0
PeCBP	11	Performance counter hardware breakpoint. Setting this bit enables the trace control logic to trace a hardware breakpoint condition.	R/W	0
PeCSync	10	Performance counter synchronization counter expiration. Setting this bit enables the trace control logic to trace a synchronization counter expiration condition.	R/W	0
PeCE	9	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1. This bit is used under software control. When trace is controlled by an external probe, this enabling is done via $TraceControl3_{PeCE}$.	R/W	0
PeC	8	Specifies whether or not Performance Control Tracing is implemented. This bit is always set to 1 in the P6600 processor.	R	1
0	7:3	Reserved. Must be written as zeros; returns zeros on reads.	R	0
TrIDLE	2	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware.	R/W	0
TRPAD	1	Trace RAM Access Disable. Disables program software access to the on-chip trace RAM using load/store instructions. This bit is loaded from $TCBCONTROLB_{TRPAD}$.	R/W	0
0	0	Reserved. Must be written as zeros; returns zeros on reads.	R	0

2.2.9.4 User Trace Data 1 Register — UserTraceData1 (CP0 Register 23, Select 3)

A software write to any bits in the *UserTraceData1* register triggers a trace record to be written with a type indicator TU1.

This register is only implemented if the MIPS Trace capability is present.

Figure 2.62 User Trace Data 1 Register Format

63 … 0
Data

Table 2.74 User Trace Data 1 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	63:0	Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory.	R/W	0

2.2.9.5 User Trace Data 2 Register — UserDataTrace2 (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData2* register triggers a trace record to be written with a type indicator TU2.

These register are only implemented if the MIPS Trace capability is present.

Figure 2.63 User Trace Data 2 Register Format

63 … 0
Data

Table 2.75 User Trace Data 2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	63:0	Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory.	R/W	0

2.2.9.6 Trace Instruction Breakpoint Condition Register — TraceIBPC (CP0 Register 23, Select 4)

The *TraceIBPC* register is used to control start and stop of tracing using an EJTAG Instruction Hardware breakpoint. The Instruction Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present.

Figure 2.64 TraceIBPC Register Format

31 30	29	28	27 12	11 9	8 6	5 3	2 0
0	PCT	IE	0	$IBPC_3$	$IBPC_2$	$IBPC_1$	$IBPC_0$

Table 2.76 TraceIBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG instruction breakpoint match occurs. 0: Disables performance counter trigger signal from instruction breakpoints 1: Enables performance trigger signals from instruction breakpoints	R/W	0
IE	28	Used to specify whether or not the trigger signal from EJTAG instruction breakpoint should trigger tracing functions. 0: Disables trigger signals from instruction breakpoints 1: Enables trigger signals from instruction breakpoints	R/W	0
0	27:12	Reserved. Must be written as zeros; returns zeros on reads.	R	0
IBPC3 *IBPC2* *IBPC1* *IBPC0*	11:9 9:6 5:3 2:0	The four 3-bit fields are decoded to enable different tracing modes. Table 2.78 shows the possible interpretations. Each set of 3 bits represents the encoding for the instruction breakpoint *n* in the EJTAG implementation, if it exists. If the breakpoint does not exist, then the bits are reserved, read as zero, and writes are ignored.	R/W	0

2.2.9.7 Trace Data Breakpoint Condition Register — TraceDBPC (CP0 Register 23, Select 5)

The *TraceDBPC* register is used to control start and stop of tracing using an EJTAG Data Hardware breakpoint. The Data Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present.

Figure 2.65 TraceDBPC Register Format

31–30	29	28	27–6	5–3	2–0
0	PCT	DE	0	$DBPC_1$	$DBPC_0$

Table 2.77 TraceDBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG data breakpoint match occurs. 0: Disables performance counter trigger signal from data breakpoints 1: Enables performance trigger signals from data breakpoints	R/W	0
DE	28	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions. 0: Disables trigger signals from data breakpoints 1: Enables trigger signals from data breakpoints	R/W	0
0	27:26	Reserved. Must be written as zeros; returns zeros on reads.	R	0

Table 2.77 TraceDBPC Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
DBPC0 *DBPC1*	2:0 5:3	The two 3-bit fields are decoded to enable different tracing modes. Table 2.78 shows the possible interpretations. Each set of 3 bits represents the encoding for the data breakpoint *n* in the EJTAG implementation, if it exists. If the breakpoint does not exist then the bits are reserved, read as zero and writes are ignored.	R/W	0

Table 2.78 BreakPoint Control Modes: IBPC and DBPC

Value	Trigger Action	Description
000	Unconditional Trace Stop	Unconditionally stop tracing if tracing was turned on. If tracing is already off, then there is no effect.
001	Unconditional Trace Start	Unconditionally start tracing if tracing was turned off. If tracing is already turned on, then there is no effect.
010	None	Reserved for future implementations.
011	Unconditional Trace Start (core and CM)	Unconditionally start tracing in both coreand coherence manager if tracing was turned off. If tracing is already turned on, then there is no effect.
100	Identical to trigger condition 000, and in addition, dump the full performance counter values into the trace stream	If tracing is currently on, dump the full values of all the implemented performance counters into the trace stream, and turn tracing off. If tracing is already off, then there is no effect.
101	Identical to trigger condition 001, and in addition, also dump the full performance counter values into the trace stream	Unconditionally start tracing if tracing was turned off. If tracing is already turned on, then there is no effect. In both cases, dump the full values of all the implemented performance counters into the trace stream.
110	Not used	Reserved for future implementations.
111	Unconditional Trace Start (core and CM), and in addition, dump the full performance counter values into the trace stream	Unconditionally start tracing in both coreand coherence manager if tracing was turned off. If tracing is already turned on, then there is no effect. Dump the full values of all the implemented performance counters into the trace stream.

2.2.10 User Mode Support Registers

This section contains the following hardware access registers.

- Section 2.2.10.1, "Hardware Enable — HWREna (CP0 Register 7, Select 0)" on page 147
- Section 2.2.10.2, "UserLocal (CP0 Register 4, Select 2)" on page 148
- Section 2.2.10.3, "LLAddr Register (CP0 Register 17, Select 0)" on page 149

2.2.10.1 Hardware Enable — HWREna (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the **rdhwr** instruction when that instruction is executed in user mode.

The low-order four bits [3:0] control access to the four registers required by the MIPS64® architecture standard. The two high-order bits [31:30] are available for implementation-dependent use.

Using the *HWREna* register, privileged software may select which of the hardware registers are accessible via the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

Software may determine which registers are implemented by writing all ones to the *HWREna* register, then reading the value back. If a bit reads back as a one, the processor implements that hardware register.

Figure 2.66 HWREna Register Format

31 30	29	28 ... 6	5	4	3	2	1	0
Impl	UL	0	XNP	PerfCnt	CCRes	CC	SYNCI_Step	CPUNum

Table 2.79 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/ Write	Reset State
Impl	31:30	These bits control access to implementation-dependent hardware registers. These registers are not currently implemented in any P6600 family processor. Attempts to access these bits results in a Reserved Instruction Exception.	R	0
UL	29	*UserLocal* register present. This register provides read access to the coprocessor 0 *UserLocal* register. Set this bit to 1 to permit user programs to obtain the value of the *UserLocal* CP0 register using **rdhwr 29.**	R/W	0
0	28:4	Ignored on write; returns zero on read.	R	0
XNP	5	When set, this bit provides read access to the coprocessor 0 Config5.XNP register bit. Set this bit to 1 to permit user programs to obtain the value of the Config5.XNP CP0 register field using rdhwr 5. See Config5.XNP.	R/W	0
PerfCnt	4	Performance Counter Pair. Even *sel* selects the Control register, while odd *sel* selects the Counter register in the pair.	R/W	0

Table 2.79 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/ Write	Reset State
CCRes	3	Resolution of the *Count* register. This value denotes the number of cycles between updates of the *Count* register. Setting this bit allows selected instructions to read the *Count* register. For example, if this bit is set, the execution of a user-mode **rdhwr 3** instruction read the interval at which the *Count* register increments. This field is encoded as follows: 0: Count register increments every cycle 1: Count register increments every second cycle 2: Count register increments every third cycle etc.	R/W	0
CC	2	*Count* register present. This register provides read access to the coprocessor 0 *Count* Register. Set this bit to 1 so a user-mode **`rdhwr 2`** can read out the value of the *Count* register.	R/W	0
SYNCI_Step	1	L1 cache line size. Setting this bit allows hardware to read the line size of the L1 cache. This field is used in conjunction **`synci`** instruction. See that instruction's description for the use of this value. In the typical implementation, this value should be zero if there are no caches in the system that must be synchronized (either because there are no caches, or because the instruction cache tracks writes to the data cache). In other cases, the return value should be the smallest line size of the caches that must be synchronized. For the P6600 core, the SYNCI_Step value is 32 since the line size is 32 bytes. Set this bit to 1 so that a user-mode **`rdhwr 1`** can read the cache line size (actually, the smaller of the L1 I-cache line size and D-cache line size). That line size determines the step between successive uses of the **`synci`** instruction, which does the cache manipulation necessary to ensure that the CPU can correctly execute the instructions.	R/W	0
CPUNum	0	This register provides read access to the coprocessor 0 $EBase_{CPUNum}$ field. Set this bit 1 so a user-mode **`rdhwr 0`** reads out the CPU ID number.	R/W	0

2.2.10.2 UserLocal (CP0 Register 4, Select 2)

UserLocal is a read-write 64-bit register that is not interpreted by the hardware and conditionally readable by software. This register is suitable for a kernel-maintained ID whose value can be read by user-level code with **`rdhwr 29`**, as long as $HWRENA_{UL}$ is set.

The presence of the *UserLocal* register is indicated by $Config3_{ULRI}$ = 1.

Figure 2.67 UserLocal Register Format

63 ... 0
UserLocal

Table 2.80 UserLocal Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
UserLocal	63:0	Software information that is not interpreted by hardware.	R/W	Undefined

2.2.10.3 LLAddr Register (CP0 Register 17, Select 0)

The *LLAddr* register stores the physical address (to the enclosing 32-byte block) of the target location of any LL/SC sequence. This register is readable purely for diagnostic reasons. This register is used by the hardware to properly handle LL/SC sequences by monitoring if the memory location has potentially been written between the LL and SC instructions.

Figure 2.68 LLAddr Register Format

63 ... 36	35 ... 32
0	PAddr

31 ... 1	0
PAddr	LLB

Table 2.81 LLAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
0	63:36	Unused bits. For these bits, writes are ignored and reads return zero.	R	Undefined
PAddr	35:1	Bits [39:5] of address used by last the LL instruction. LLAddr[1] is always aligned to PA[5], which implies PAddr is always 32-byte aligned.	R	Undefined
LLB	0	Load-Linked bit. The LL instruction sets this bit when executed. The SC instructions and other hardware events may clear the LLB bit. This bit allows the LL bit to be software accessible. Software can never write 1 to LL bit. In this case, the state of LLAddr.LLB must remain unchanged. Software may clear LL bit by writing a 0 to LLAddr..LLB.	R/W	0

2.2.11 Kernel Mode Support Registers

This section contains the following 64-bit kernel scratch registers.

- KScratch1 (CP0 Register 31, Select 2)
- KScratch2 (CP0 Register 31, Select 3)
- KScratch3 (CP0 Register 31, Select 4)
- KScratch4 (CP0 Register 31, Select 5)
- KScratch5 (CP0 Register 31, Select 6)
- KScratch6 (CP0 Register 31, Select 7)

The presence of *KScratch* registers is indicated by the $Config4_{KScrExist}$ field (bits 23:18). Six *KScratch* registers are required in the MIPSr6 architecture and reside at CP0 register 31, selects 2 - 7. As such, the various bits of the *KScrExist* field are used to identify the presence of the *KScratch* registers as shown in the table below.

Table 2.82 KScratch Register Map

CP0 Config4 Register Bit	Bit Name	Indicates the Presence of	KScratch Register Location
18	KScrExist[2]	KSratch1 register	CP0 register 31, select 2
19	KScrExist[3]	KSratch2 register	CP0 register 31, select 3
20	KScrExist[4]	KSratch3 register	CP0 register 31, select 4
21	KScrExist[5]	KSratch4 register	CP0 register 31, select 5
22	KScrExist[6]	KSratch5 register	CP0 register 31, select 6
23	KScrExist[7]	KSratch6 register	CP0 register 31, select 7

Each of the KScratch registers listed above have an identical bit orientation as shown below.

KScratch1 - KScratch6 are read-write 64-bit registers used by the kernel for temporary storage of information.

The presence of the *KScratch* registers is indicated by $Config4_{KScrExist[7:2]}$ = 1'b1 as shown in Table 2.82 above.

Figure 2.69 KScratch 1 - 6 Register Format

63 … 0
KScratch

Table 2.83 KScratch 1 - 6 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
KScratch	63:0	Used by the kernel for temporary storage of information.	R/W	Undefined

2.2.12 Memory Mapped Registers

This section contains the following memory mapped registers.

- Section 2.2.12.1, "Common Device Memory Map Base Address — CDMMBase (CP0 Register 15, Select 2)" on page 152
- Section 2.2.12.2, "Coherency Manager Global Configuration Register Base Address — CMGCRBase (CP0 Register 15, Select 3)" on page 153

2.2.12.1 Common Device Memory Map Base Address — CDMMBase (CP0 Register 15, Select 2)

The 32-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if $Config3_{CDMM}$ is set to one.

Figure 2.70 shows the format of the *CDMMBase* register, and Table 2.84 describes the register fields.

Figure 2.70 CDMMBase Register

63 … 40	35 … 11	10	9	8 … 0
0	CDMM_UPPER_ADDR	EN	CI	CDMMSize

Table 2.84 CDMMBase Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	63:36	Unimplemented physical address bits. Writes are ignored, returns 0 on read.	R	0
CDMM_UPPER_ADDR	35:11	Bits 39:15 of the base physical address of the common device memory-mapped registers.	R/W	Undefined
EN	10	Enables the CDMM region. If this bit is cleared, memory requests to this address region go to regular system memory. If this bit is set, memory requests to this region go to the CDMM logic. 0: CDMM region is disabled. 1: CDMM region is enabled.	R/W	0
CI	9	If set to 1 by hardware, this bit indicates that the first 64-byte Device Register Block (DRB) of the CDMM is reserved for additional registers which manage CDMM region behavior and are not IO device registers. This bit is always 0 in the P6600 core since additional I/O device registers are not implemented.	R	0
CDMMSize	8:0	This field represents the number of 64-byte Device Register Blocks (DRB) instantiated in the P6600 core. 0x000: 1 DRB 0x001: 2 DRB's 0x010: 3 DRB's ... 0x1FF: 512 DRB's	R	2

2.2.12.2 Coherency Manager Global Configuration Register Base Address — CMGCRBase (CP0 Register 15, Select 3)

This register is used in a multi-core environment and defines the 36-bit physical base address for the memory-mapped Coherency Manager Global Configuration Register (CMGCR) space. This register only exists if $Config3_{CMGCR}$ is set.

Figure 2.71 shows the format of the *CMGCRBase* register, and Table 2.85 describes the register fields.

Figure 2.71 CMGCRBase Register

63 — 36	35 — 11	10 — 0
0	CMGCR_BASE_ADDR	0

Table 2.85 CMGCRBase Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	63:36	Unimplemented physical address bits. Writes are ignored, returns 0 on read	R	0
CMGCR_ BASE_ADDR	35:11	Bits 39:15 of the base physical address of the memory mapped Coherency Manager Global Configuration registers. The number of implemented physical address bits is implementation-specific. For the unimplemented address bits, writes are ignored, reads return zero. The reset value is set when the core is configured using the Configuration GUI.	R	Preset
0	10:0	Must be written as zero; returns zero on read	R	0

2.2.13 Virtualization Registers

This section contains the set of register used to control Virtualization on the P6600 core. The Virtualization Module extends the MIPS64 architecture with a set of new instructions and machine state, and makes backward-compatible modifications to existing MIPS32 features.

The Virtualization Module is designed to enable full virtualization of operating systems and allows for the execution of guest Operating Systems in a fully virtualized environment. Software can determine if the Virtualization Module is implemented by checking the state of the VZ bit in the *Config3* CP0 register.

The Virtualization Module is supported by the following CP0 register.

- Section 2.2.13.1, "GuestCtl0 Register (CP0 Register 12, Select 6)"
- Section 2.2.13.2, "GuestCtl1 Register (CP0 Register 10, Select 4)"
- Section 2.2.13.3, "GuestCtl2 Register (CP0 Register 10, Select 5)"
- Section 2.2.13.4, "GuestCtl0Ext Register (CP0 Register 11, Select 4)"
- Section 2.2.13.5, "GTOffset Register (CP0 Register 12, Select 7)"

2.2.13.1 GuestCtl0 Register (CP0 Register 12, Select 6)

The *GuestCtl0* register contains control bits that indicate whether the base mode of the processor is guest mode or root mode, plus additional bits controlling guest mode access to privileged resources. The *GuestCtl0* register is accessible only in root mode.

Note on behaviour of $GuestCtl0_{DRG/RAD}$: These R/W fields define additional functions for the Guest and Root TLBs. Both must be interpreted together. An implementation does not have to support all valid combinations. Root software can test supported combinations by writing then reading legal values. Legal values for (RAD,DRG)={00,01,11}.

Figure 2.72 shows the format of the Virtualization Module *GuestCtl0* register; Table 2.86 describes the *GuestCtl0* register fields.

Figure 2.72 GuestCtl0 Register Format

31	30	29	28	27 26	25	24	23	22	21 20	19	18	17 16	15 10	9	8	7	6 2	1	0
GM	RI	MC	CP0	AT	GT	CG	CF	G1	Impl	G0E	PT	ASE	PIP	RAD	DRG	G2	GExcCode	S FC2	S FC1

Table 2.86 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
GM	31	Guest Mode The processor is in guest mode when GM = 1 and the following bits are all zero:$Root.Status_{EXL}$= 0, $Root.Status_{ERL}$ = 0, and $Root.Debug_{DM}$ = 0.	R/W	0
RI	30	Guest Reserved Instruction Redirect. This field is encoded as follows: 0: Reserved Instruction exceptions during guest-mode execution are taken in guest mode. 1: Reserved Instruction exceptions during guest-mode execution result in a Guest Reserved Instruction Redirect exception, taken in root mode.	R/W	0
MC	29	Guest Mode-Change exception enable. The purpose of this enable is to provide Root software control over certain mode-changing events within guest context that may be frequent in guest context by causing Field Change exceptions. This field is encoded as follows: 0: During guest mode execution a hardware initiated change to $Guest.Status_{EXL}$ will not trigger a Guest Hardware Field Change Exception. During guest mode execution, a software initiated change to $Guest.Status_{UM/KSU}$ will not trigger a Guest Software Field Change Exception. 1: During guest mode execution a hardware initiated change to $Guest.Status_{EXL}$ will trigger a Guest Hardware Field Change Exception. During guest mode execution, a software initiated change to $Guest.Status_{UM/KSU}$ will trigger a Guest Software Field Change Exception.	R/W	0

Table 2.86 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CP0	28	Guest access to coprocessor 0. This field is encoded as follows: 0: Guest-kernel use of any Guest Privileged Sensitive Instruction will trigger a Guest Privileged Sensitive Instruction exception. E.g., Guest use of TLBWI always causes GPSI if CP0 = 0. 1: Guest-kernel use of selective Guest Privileged Sensitive Instructions is permitted, subject to all other exception conditions. Eg., Guest use of TLBWI only causes GPSI if $GuestCtl0_{AT}$!=3 while CP0 = 1. The CP0 bit has no other effect on the operation of coprocessor 0 in guest mode.	R/W	0
AT	27:26	Guest Address Translation control This field indicates which entity has control over the guest MMU. In the P6600 core the value of this field is always 0x3, indicating that the Guest MMU is under Guest control. Guest and Root MMU are both implemented and active in hardware. Guest TLB resources include: • TLB related instructions - TLBWR, TLBWI, TLBR, TLBP, TLBINV, TLBINVF. • Supporting Registers - *Index, Random, EntryLo0, EntryLo1, EntryHi, Context, XContext, ContextConfig, PageMask, PageGrain, SegCtl0, SegCtl1, SegCtl2*, *PWBase, PWField, PWSize, PWCtl*. If the Guest TLB resources (excluding Index, Random, EntryLo0, EntryLo1, Context, XContext, ContextConfig, PageMask and EntryHi) are under Root control ($GuestCtl0_{AT}$ = 1), Guest use of these instructions or access to any of these registers triggers a Guest Privileged Sensitive Instruction exception, allowing Root to control Guest address translation directly. In default mode ($GuestCtl0_{AT}$ = 3), the Guest TLB resources are active under Guest control.	R	0x3
GT	25	Timer register access. This register is encoded as follows: 0: Guest-kernel access to *Count* or *Compare* registers, or a read from CC with RDHWR will trigger a Guest Privileged Sensitive Instruction exception. 1: Guest kernel read access from *Count* and guest-kernel read or write access to *Compare* is permitted. Guest reads from CC using RDHWR are permitted in any mode. The GT bit has no other effect on the operation of timers in guest mode.	R/W	0
CG	24	Cache Instruction Guest-mode enable. This register is encoded as follows: 0: A Guest Privileged Sensitive Instruction exception will result from use the CACHE, CACHEE instruction. 1: The CACHE, CACHEE instruction can be used with an Effective Address Operand type of ‘Address’. A Guest Privileged Sensitive Instruction exception will result from use of any other Effective Address Operand type.	R/W	0

Table 2.86 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CF	23	Config register access. This register is encoded as follows: 0: Guest-kernel write access to *Config0-7* triggers a Guest Privileged Sensitive Instruction exception. 1: Guest-kernel access to *Config0-7* is permitted. The CF bit has no other effect on the operation of *Config* register fields in Guest mode.	R/W	0
G1	22	*GuestCtl1* register implemented. Set by hardware. This register is encoded as follows: 0: Unimplemented 1: Implemented	R	Preset
Impl	21:20	Implementation defined. These bits are implementation dependent and not defined by the architecture. If not implemented, they must be ignored on write and read as zero. If implemented and if modifying the behavior of the processor, it must be defined in such a way that correct behavior is preserved if software, with no knowledge of these bits, reads the *GuestCtl0* register, modifies another field, and writes the updated value back to the *GuestCtl0* register.	R/W	0
G0E	19	*GuestCtl0Ext* register implemented. Set by hardware. This register is encoded as follows: 0: Unimplemented 1: Implemented	R	1
PT	18	Defines the existence of the Pending Interrupt Pass-through feature. This register is encoded as follows: 0: $GuestCtl0_{PIP}$ not supported. $GuestCtl0_{PIP}$ is a reserved field. All external interrupts are processed via Root intervention. 1: $GuestCtl0_{PIP}$ supported. Interrupts may be assigned to Root or Guest.	R	1
ASE	17:16	Reserved for MCU Module Pending Interrupt Pass-through. This field is not used in the P6600 core and is always zero.	0	0
PIP	15:10	Pending Interrupt Pass-through. In non-EIC mode, controls how external interrupts are passed through to the guest CP0 context. Interpreted as a bit mask and applies 1:1 to $Guest.Cause_{IP}[7:2]$. $GuestCtl1_{PIP}$ may be extended by $GuestCtl1_{ASE}$. Existence of the PIP feature is defined by the $GuestCtl0_{PT}$ field. This field is encoded as follows: 0: Corresponding interrupt request is not visible in guest context. 1: Corresponding interrupt request is visible in guest context.	R/W	0
RAD	9	RAD, or "Root ASID Dealias" mode determines the means that a Virtualized MMU implementation uses Root ASID to dealias different contexts. This field is encoded as follows: 0: GuestID used to de-alias both Guest and Root TLB entries. 1: Root ASID is used to de-alias Root TLB entries, while Guest TLB contains only one context at any given time.	R	0

Table 2.86 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
DRG	8	DRG, or "Direct Root to Guest" access determines whether an implementation provides root kernel the means to access guest entries directly in the Root TLB for access to guest memory. This bit is always 0 in the P6600 as root software cannot access guest entries directly.	R0	0
G2	7	*GuestCtl2* register implemented. Set by hardware. This bit is always set to 1 in the P6600 core.	R	preset
GExCode	6:2	Hypervisor exception cause code. Described in Table 2.87. This field is UNDEFINED on a root exception.	R	Undefined
SFC2	1	Guest Software Field Change exception enable for $Guest.Status_{CU[2]}$. The purpose of this enable is to provide Root software control over guest COP2 enable related Field Change exception. This bit is not used and is always 0 in the P6600 as COP2 is not supported.	R	0
SFC1	0	Guest Software Field Change exception enable for $Guest.Status_{CU[1]}$. The purpose of this enable is to provide Root software control over guest COP1 enable related Field Change exception. Guest software may utilize $Status_{CU1}$ for COP1 specific context switching. This bit is encoded as follows: 0: GSFC exception taken if CU[1] is modified by guest. 1: GSFC exception not taken if CU[1] modified by guest.	R/W	0

Table 2.87 describes the cause codes use for GExcCode.

Table 2.87 GuestCtl0 GExcCode values

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
0	0x00	GPSI	Guest Privileged Sensitive instruction. Taken when execution of a Guest Privileged Sensitive Instruction was attempted from guest-kernel mode, but the instruction was not enabled for guest-kernel mode.
1	0x01	GSFC	Guest Software Field Change event.
2	0x02	HC	Hypercall.
3	0x03	GRR	Guest Reserved Instruction Redirect. A Reserved Instruction or MDMX Unusable exception would be taken in guest mode. When $GuestCtl0_{RI}$=1, this root-mode exception is raised before the guest-mode exception can be taken.
4 - 7	0x4 - 0x7	IMP	Available for implementation specific use.
8	0x08	GVA	Guest mode initiated Root TLB exception has Guest Virtual Address available. Set when a Guest mode initiated TLB translation results in a Root TLB related exception occurring in Root mode and the Guest Physical Address is not available.
9	0x09	GHFC	Guest Hardware Field Change event.
10	0x0A	GPA	Guest mode initiated Root TLB exception has Guest Physical Address available. Set when a Guest mode initiated TLB translation results in a Root TLB related exception occurring in Root mode and the Guest Physical Address is available.

Table 2.87 GuestCtl0 GExcCode values

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
11 - 31	0xB - 0x1F	-	Reserved

2.2.13.2 GuestCtl1 Register (CP0 Register 10, Select 4)

The *GuestCtl1* register defines GuestID control fields for Root ($GuestCtl1_{RID}$) and Guest ($GuestCtl1_{ID}$) which may be used in the context of TLB instructions, instruction and data address translation. The $GuestCtl1_{RID}$ field additionally is written by the processor on a TLBR or TLBGR instruction in Root mode, then containing the GuestID read from the TLB entry. A TLBR executed in Guest mode does not cause a write to either $GuestCtl1_{ID}$ and $GuestCtl1_{RID}$.

GuestCtl1 is optional and thus the use of GuestID is optional in the context of TLB instructions, instruction and data address translation. The *GuestCtl1* register only exists in Root Context. A GuestID value of 0 is reserved for Root. The primary purpose of the GuestID is to provide a unique component of the Guest/Root TLB entry eliminating TLB invalidation overhead on virtual machine level context switch.

A system implementing a GuestID is required to support a guest identifier field (GID) in each Guest and Root TLB entry. This GuestID field within the TLB is not accessible to the Guest. While operating in guest context, the behavior of guest TLB operations is constrained by the $GuestCtl1_{ID}$ field so that only guest TLB entries with a matching GID field are considered.

The actual number of bits usable in the $GuestCtl1_{ID}$ and $GuestCtl1_{RID}$ fields is implementation dependent. Software may determine the usable size of these fields by writing all ones and reading the value back. The size of $GuestCtl1_{ID}$ and $GuestCtl1_{RID}$ must be equal.

Figure 2.73 shows the format of the Virtualization Module *GuestCtl1* register; Table 2.88 describes the *GuestCtl1* register fields.

Figure 2.73 GuestCtl1 Register Format

31 - 24	23 - 16	15 - 8	7 - 0
EID	RID	0	ID

Table 2.88 GuestCtl1 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
EID	31:24	External Interrupt Controller Guest ID. Required if an External Interrupt Controller (EIC) is supported. A guest interrupt which is posted by the EIC to the root interrupt bus, must cause the Guest ID of the root interrupt bus to be registered in EID once the interrupt is taken. This field is read-only and set by hardware.	R	0
RID	23:16	Root control GuestID. Used by root TLB operations, and when $GuestCtl0_{DRG}$ = 1 in Root mode. Legal values for this field are 0x00 - 0x0F. A value greater than 0x0F causes the entire write operation to be dropped.	R/W	0
0	15:8	Must be written as zero; returns zero on read.	R	0
ID	7:0	Guest control GuestID. Identifies resident guest. Applies to guest address translation. A value greater than 0x0F causes the entire write operation to be dropped.	R/W	0

2.2.13.3 GuestCtl2 Register (CP0 Register 10, Select 5)

The *GuestCtl2* register is optional in an implementation. It is only required if support for Virtual Interrupts in non-EIC mode is included in an implementation. Alternatively, if EIC mode is supported, then *GuestCtl2* is required.

GuestCtl2 is present if $GuestCtl2_{G2} = 1$.

Figure 2.74 shows the format of the Virtualization Module *GuestCtl2* register in non-EIC mode. Table 2.89 describes the non-EIC mode *GuestCtl2* register fields.

Figure 2.75 shows the format of the Virtualization Module *GuestCtl2* register in EIC mode. Table 2.90 describes the EIC mode *GuestCtl2* register fields.

Figure 2.74 GuestCtl2 Register Format for non-EIC Mode

31 30	29 24	23 18	17 16	15 10	9 0
ASEHC	HC	0	ASEVIP	VIP	0

Figure 2.75 GuestCtl2 Register Format for EIC Mode

31 30	29 24	23 16	15 0
ASE	GRIPL	0	GVEC

Table 2.89 non-EIC mode GuestCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
ASEHC	31:30	MCU Module extension for HC. Must be written as zero; returns zero on read.	R	0
HC	29:24	Hardware Clear for $GuestCtl2_{VIP}$ This set of bits maps one to one to $GuestCtl2_{VIP}$. This field is encoded as follows. 0: The deassertion of related external interrupt (IRQ[n]) has no effect on $GuestCtl2_{VIP}$[n]. Root software must write zero to $GuestCtl2_{VIP}$[n] to clear the virtual interrupt. 1: The deassertion of related external interrupt (IRQ[n]) causes $GuestCtl2_{VIP}$[n] to be cleared by hardware. In the case of HC = 0, $Guest.Cause_{IP}$[n+2] could continue to be asserted due to an external interrupt when $GuestCtl2_{VIP}$[n] is cleared by software. Source of external interrupt must be serviced appropriately. Root software can write then read this field to determine the supported configuration.	R/W	0
0	25:18	Must be written as zero; returns zero on read.	R	0

Table 2.89 non-EIC mode GuestCtl2 Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
ASEVIP	17:16	MCU Module extension for VIP. Must be written as zero; returns zero on read.	R	0
VIP	15:10	Virtual Interrupt Pending. The VIP field is used by root to inject virtual interrupts into Guest context. VIP[5:0] maps to *Guest.Status*$_{IP}$[7:2]. VIP effects *Guest.Status*$_{IP}$ in the following manner: 0: *Guest.Status*$_{IP}$[n+2] cannot be asserted due to VIP[n], though it may be asserted by an external interrupt IRQ[n]. n = 5:0. 1: *Guest.Status*$_{IP}$[n+2] must at least be asserted due to VIP[n]. It may also be asserted by a concurrent external interrupt. n=5:0.	R/W	0
0	9:0	Must be written as zero; returns zero on read.	R0	0

Table 2.90 EIC mode GuestCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
ASE	31:30	MCU Module extension for GRIPL. This field is not used by the P6600 core, and must be written as zero; returns zero on read.	R	0
GRIPL	29:24	Guest RIPL This field is written only when an interrupt received on the root interrupt bus for a guest is taken. The RIPL(Requested Interrupt Priority Level) sent by EIC on the root interrupt bus is written to this field. Root software can write the field if it needs to modify the EIC value before assigning to guest. It may also clear this field to prevent a transition to guest mode from causing an interrupt if this field was set with a non-zero value earlier.	R/W	0
GEICSS	21:18	Guest EICSS This field is written only when an interrupt received on the root interrupt bus for a guest is taken. The EICSS (External Interrupt Controller Shadow Set) sent by EIC on the root interrupt bus is written to this field Root software can write the field if it needs to modify the EIC value before assigning to guest.	R/W	Undefined
0	23:16	Must be written as zero; returns zero on read.	R	0

Table 2.90 EIC mode GuestCtl2 Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
GVEC	15:0	Guest Vector This field is written only when an interrupt is received on the root interrupt bus for a guest. The Vector Offset (or Number) sent by EIC on the root interrupt bus is written to this field. GVEC is not loaded into any guest CP0 field, but is used to generate an interrupt vector in guest mode using the root interrupt bus vector and not the guest interrupt bus vector. This will only occur if the interrupt was first taken in root mode. It is recommended that root software use write access only to restore context, not to modify the value delivered by the EIC.	R/W	0

2.2.13.4 GuestCtl0Ext Register (CP0 Register 11, Select 4)

$GuestCtl0_{G0E}$ should be read by software to determine if *GuestCtl0Ext* is implemented.

Figure 2.76 shows the format of the Virtualization Module *GuestCtl0Ext* register. Table 2.91 describes the *GuestCtl0Ext* register fields.

Figure 2.76 GuestCtl0Ext Register Format

31 … 10	9 … 8	7 … 6	5	4	3	2	1	0
0	RPW	NCC	0	CGI	FCD	OG	BG	MG

Table 2.91 GuestCtl0Ext Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:6	Must be written as zero, returns zero on read.	R0	0
RPW	9:8	Root Page Walk configuration. Determines whether Root COP0 Page Walk registers are used for GPA to RPA or RVA to RPA translations, or both. This field is encoded as follows: 00: Pagewalk, if enabled, is enabled for both. Root software is responsible for restoring COP0 Page Walk related registers on context switch between root and guest. 01: Reserved 10: Pagewalk in root context is enabled for guest GPA to RPA translation. Root miss in root TLB causes an exception. 11: Pagewalk in root context is enabled for root RVA to RPA translation. Guest miss in root TLB causes a root exception. Note that the 10 encoding is reserved for internal use. As such, software should never program this field with a value of 2'b10 as it will cause the entire write operation to be dropped.	R/W	0
NCC	7:6	Nested Cache Coherency Attributes Determines whether guest CCA is modified by root CCA in 2nd step of guest address translation. This field is encoded as follows: 00: Guest CCA is independent of root CCA. 01: Guest CCA is modified by root CCA. 10: Guest CCA is passed through without being modified by the root CCA. 11: Reserved The P6600 supports encoding 2'b10 of this field. The P6600 core converts unsupported CCAs to supported CCAs. CCA conversion must only be carried out on the effective CCA after the result of combining guest and root CCAs (GuestVA -> GuestPA -> RootPA). For RootVA -> RootPA translations, the effective CCA is the CCA from the root TLB entry.	R	10
0	5	Must be written as zero, returns zero on read.	R0	0
CGI	4	Related to $GuestCtl0_{CG}$. Allows execution of CACHE, CACHEE Index Invalidate operations in guest mode. This field is encoded as follows: 0: Definition of $GuestCtl0_{CG}$ does not change. 1: If $GuestCtl0_{CG}$ =1 and $GuestCtl0Ext_{CGI}$ =1, then all CACHE, CACHEE Index Invalidate (code 0xb000) operations may execute in guest mode without causing a GPSI.	R/W	0

Table 2.91 GuestCtl0Ext Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
FCD	3	Disables Guest Software/Hardware Field Change Exceptions (GSFC/GHFC). This mode is useful for an implementation with root software that is not a full-featured hypervisor. For e.g., the software may just support memory protection, but may not require protection of CP0 state. If FCD = 1, then hardware must treat guest write, in case of GSFC, and hardware events, in case of GHFC. This bit is encoded as follows: 0: GSFC or GHFC event will cause exception. 1: GSFC or GHFC event will not cause exception.	R/W	0
OG	2	Other GPSI Enable. Applies to *UserLocal, HWREna*, *LLAddr,* and *KScratch1* through *KScratch6*. This bit is encoded as follows: 0: GPSI not enabled for these registers unless $\text{GuestCtl0}_{\text{CP0}}$=0. 1: GPSI enabled for these registers.	R/W	0
BG	1	Bad register GPSI Enable. Applies to *BadVAddr, BadInstr,* and *BadInstrP*. This field is encoded as follows: 0: GPSI not enabled for these registers unless $\text{GuestCtl0}_{\text{CP0}}$=0. 1: GPSI enabled for these registers.	R/W	0
MG	0	MMU GPSI Enable. Applies to *Index, EntryLo0, EntryLo1, Context, ContextConfig, XContextConfig*, *PageMask,* and *EntryHi*. This field is encoded as follows: 0: GPSI not enabled for these registers unless $\text{GuestCtl0}_{\text{CP0}}$=0. 1: GPSI enabled for these registers.	R/W	0

2.2.13.5 GTOffset Register (CP0 Register 12, Select 7)

Timekeeping within the guest context is controlled by root mode. The guest time value is generated by adding the two's complement offset in the *Root.GTOffset* register to the root timer in value *Root.Count*.

The guest time value is used to generate timer interrupts within the guest context, by comparison with the *Guest.Compare* register. The guest time value can be read from the *Guest.Count* register. Guest writes to the *Guest.Count* register always result in a Guest Privileged Sensitive Instruction exception.

The number of bits supported in *GTOffset* is implementation dependent but must be non-zero. It is recommended that a minimum of 16 bits be implemented. Root software can check the number of implemented bits by writing all ones and then reading. Unimplemented bits will return zero.

Figure 2.77 shows the Virtualization Module format of the *GTOffset* register; Table 2.92 describes the *GTOffset* register fields.

Figure 2.77 GTOffset Register Format

31	0
GTOffset	

Table 2.92 GTOffset Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
GTOffset	31:0	Two's complement offset from *Root.Count*.	R/W	0

2.2.14 Memory Accessibility Attribute Registers

The 64-bit Memory Accessibility Attribute registers (MAAR) and the 64-bit Memory Accessibility Attribute register Index (MAARI) define the accessibility attributes of memory regions.

The MAAR register defines whether an instruction fetch or data load/store can speculatively access a memory region within the address bounds specified by MAAR. The *MAARI* register is used to specify a *MAAR* register number that may be accessed by software with an MTC0 or MFC0 instruction. Prior to access by MTC0 or MFC0, software must set the $MAARI_{INDEX}$ field to the appropriate value.

MAAR Register Pairs

The P6600 core contains three pairs of MAAR registers, each of which are indexed using the MAAR Index (MAARI) register located at CP0 Register 17, Sel 2. Each MAAR register pair consists of a 64-bit even and an odd register. The three MAAR register pairs are as follows, where 'O' indicates the odd register of the pair and 'E' indicates the even register; MAAR0O / MAAR0E, MAAR1O / MAAR1E, and MAAR2O / MAAR2E.

The MAARI register must be initialized with the appropriate MAAR register number before the MAAR can be accessed with an MTC0 or MFC0 instruction. An EHB instruction is required to be placed in between the write to MAARI and the subsequent execution of a MTC0 or MFC0 instruction that specifies the MAAR.

The P6600 core implements three pairs of MAAR registers. The presence of a *MAAR* register pair can be detected by software through $Config5_{MRP}$.

3-Pair MAAR Implementation

The following pseudo-code shows a 3-pair MAAR implementation to determine speculation. Software must set the logical valid to 1 of each register in the pair to enable a MAAR pair. It may however, clear any one logical valid of the pair to invalidate the whole MAAR pair. Once both logical values are set to 1, hardware factors in the speculate attribute of only the upper MAAR register with even index. The logical valid is determined as described in the pseudo-code below.

```
speculateCCA ¬ 0 // default is not to speculate
// Modify speculate attribute as per CCA of memory access
// Cached CCA and UCA speculates
if ((CCA == “cached”) or (CCA == “uncached-accelerated (UCA)”))
        speculateCCA ¬ 1
endif
```

```
// Now factor in MAAR
MAARmatch ¬ 0
speculateMAAR ¬ 1
// Example of 40-bit PA is 64KB aligned
PA_Align ¬ PA[39:16]
for (i=0; i<6; i=i+2) // assume 3 pairs

                // Factor in XPA (Extended Physical Addressing)
                MAAR[i]V = MAAR[i]VL and (MAAR[i]VH or not PageGrainELPA)
                MAAR[i+1]V = MAAR[i+1]VL and (MAAR[i+1]VH or not PageGrainELPA)
                if (MAAR[i]V and MAAR[i+1]V) // both logical valids must be set to 1
                        if ((MAAR[i][35:12] >= PA_Align) && // upper bound
                                (MAAR[i+1][35:12] <= PA_Align)) // lower bound
                        speculateMAAR ¬ speculate_MAAR and MAAR[i]_S
                        MAARmatch ¬ 1
                endif
        endif
endfor

// if no MAAR is valid, or no MAAR match occurs, then speculateMAAR ¬ 0 speculate ¬ speculateMAAR and
// speculateCCA and MAARmatch
```

Programming the State of the MAAR / MAARI Register Pair

Software must follow the described method for reprogramming the state of a MAAR pair.

- Disable the MAAR pair by clearing MAAR.VL and MAAR.VH. Accesses to the MAAR region become non-speculative.
- Program $PageGrain_{ELPA}$ as needed.
- Set MAAR.VL along with other fields in MAAR[63:0]

2.2.14.1 Memory Accessibility Attribute Register (CP0 Register 17, Select 1)

The Memory Accessibility Attribute Register (*MAAR)* is a read/write register defines the accessibility attributes of memory regions. In particular, *MAAR* defines whether an instruction fetch or data load/store can speculatively access a memory region within the address bounds specified by *MAAR*.

The purpose of the MAAR register is to control speculation on load or fetch access to memory and I/O addresses. A load is considered speculative if it accesses memory prior to its being the oldest instruction to retire. A fetch typically always speculates on access to memory, while never speculating to I/O.

If the *MAAR* function yields a valid attribute, it will only override any equivalent attribute determined through other means, if it provides a more conservative outcome. For example, if the MMU yields a cacheable CCA, but *MAAR* yields a speculate attribute set to 0, then the access should not speculate as determined by the *MAAR* result. Similarly, if the MMU yields an uncacheable CCA, but *MAAR* yields a speculate attribute set to 1, then the access should not speculate.

The CCA of a memory access now defines speculation, along with *MAAR*. A memory access with a cacheable CCA is allowed to speculate. A memory access with uncacheable CCA on the other hand is not allowed to speculate unless the uncacheable CCA = 7 (UCA) is used. The final speculative attribute is a combination of the CCA and *MAAR* as described above.

The address range specified by a *MAAR* may be used to specify an attribute for any region of the address space, whether memory (DRAM) or memory-mapped I/O.

Note that the *MAARI* register must be initialized with the appropriate *MAARI* register number before the *MAAR* is accessed with an MTC0 or MFC0 instruction. An EHB instruction is required to be placed between the write to *MAARI* and subsequent execution of MTC0 or MFC0 that specifies the *MAAR*.

The MAAR register has the following properties:

- If all MAAR instances are invalid, then no speculation is allowed. This allows the MAAR initialization to occur at any point of time without the risk of execution speculative (bad path) loads or fetches from issuing to IO addresses, with the tradeoff possibly being lower performance.
- If any MAAR region enables speculation, then accesses to physical addresses outside this MAAR region must be non-speculative, unless the physical address of the access matches against a MAAR region with speculation enabled. This access can then speculate.
- MAAR overlap is allowed: This allows non-speculative MAAR region to overlap a speculative MAAR region. For e.g., with this property, a non-speculative region can be overlayed on a speculative DRAM region with the use of just two MAAR pairs.

For software to enable a speculative region out of reset, it should first initialize MAARxO[63:0] and then MAARxE[63:32].

Figure 2.78 shows the format of the *MAAR* register; Table 2.93 describes the *MAAR* register fields.

Figure 2.78 MAAR Register Format

63 … 36	35 … 12	11 … 2	1	0
0	ADDR	0	S	V

Table 2.93 MAAR Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	63:36	Reserved. Writes are ignored, read as 0.	R	0

Table 2.93 MAAR Register Field Descriptions *(continued)*

Fields		Description	Read/Write	Reset State
Name	Bits			
ADDR	35:12	Address bounds. *ADDR* must always specify a physical address. *MAAR* regions are at least 64KB-aligned, and thus the least-significant bit of *ADDR* is equal to PA[16]. If the register specifies the upper bound, then any sourced address must be less than or equal to *ADDR*. If the register specifies the lower bound, then any sourced address must be greater than or equal to *ADDR*. See *MAAR Index* (CP0 Register 17, Select 2) for the method of determining which register is upper or lower in a pair. *MAAR*[12] = PA[16]. This allows the *MAAR* register to specify 40 bits of PA, where *MAAR*[35] = PA[39]. The lower 16 bits of the PA are not specified in this register since the MAAR regions must be 64 KB aligned.	R/W	Undefined
0	11:2	Reserved. Writes are ignored, read as 0.	R	0
S	1	Speculate. If an access is qualified as non-speculative, it must be the oldest unretired instruction in the processor before being allowed to access memory or memory-mapped regions. This field is encoded as follows: 0: Instruction fetch or data load/store that matches MAAR register pair address range is never allowed to speculatively access address range. 1: Instruction fetch or data load/store that matches MAAR register pair address range may be allowed to speculate. *MAAR* regions are allowed to overlap. The cumulative speculative attribute for overlapping regions is determined by ANDing individual valid *MAAR* pair speculation attributes.	R/W	Undefined
V	0	MAAR register valid. This field is encoded as follows: 0: MAAR register is not valid and should not modify the behavior of any instruction fetch or data load/store. 1: MAAR register is valid and may modify behavior of any instruction fetch or data load/store that falls within the range of addresses specified by the MAAR register pair. If either valid bit of the *MAAR* register pair is set to 0, then the pair is assumed invalid and thus will not modify the behavior of any memory access. Software may thus invalidate one register of the *MAAR* pair to invalidate the *MAAR* comparison.	R/W	0

Table 2.94 shows how the valid attribute for a *MAAR* pair is determined from the cumulative individual *MAAR* register valids.

Table 2.94 Valid Determination for MAAR Pair

$MAAR[i]_V$ where i is even	$MAAR[i+1]_V$	Result
0	0	Result is invalid
0	1	Result is invalid
1	0	Result is invalid
1	1	Result is valid

Table 2.95 shows how the speculate attribute for a *MAAR* pair is determined by the cumulative individual speculate attributes.

Table 2.95 Speculate Determination for MAAR Pair

$MAAR[i]_S$ where i is even	$MAAR[i+1]_S$	Result
1	0/1	Valid access may speculate
0	0/1	Valid access may never speculate

2.2.14.2 Memory Accessibility Attribute Register Index (CP0 Register 17, Select 2)

The *MAAR Index* register is used in conjunction with *MAAR* registers (CP0 Register 17, Select 1). Multiple *MAAR* registers may be implemented - *MAAR Index* is used to specify a *MAAR* register number that may be accessed by software with an MTC0 or MFC0 instruction. Prior to access by MTC0 or MFC0, software must set $MAARI_{INDEX}$ to the appropriate value.

Figure 2.79 shows the format of the *MAAR Index* register; Table 2.96 describes the *MAAR Index* register fields.

The presence of *MAARI* can be detected by software through $Config5_{MRP}$.

Figure 2.79 MAAR Index Register Format

63 … 6	5 … 0
0	INDEX

Table 2.96 MAARI Index Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	63:6	Reserved. Writes are ignored, read as 0.	R	0

Table 2.96 MAARI Index Register Field Descriptions *(continued)*

Fields		Description	Read/Write	Reset State
Name	Bits			
INDEX	5:0	MAAR Index. The number of *MAAR* registers is greater than 1. *INDEX* specifies the *MAAR* register to access. MAAR registers are paired. The least-significant bit of *INDEX* is encoded as follows to indicate which register of the pair is being accessed. 0: This register specifies the upper address bound of the MAAR register pair. 1: This register specifies the lower address bound of the MAAR register pair. Software may write all ones to *INDEX* to determine the maximum value supported. Other than the all ones, if the value written is not supported, then *INDEX* is unchanged from its previous value since the write is dropped. The register range is always contiguous and starts at value 0.	R/W	0

2.2.15 Memory Segmentation Registers

Programmable segmentation is a backward compatible mode in the P6600 that allows for the virtual address space segments to be programmed with different access modes and attributes when operating in 32-bit mode. 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 Table 2.97. For more information, refer to Section 2.6 of the MMU chapter of this manual.

Table 2.97 Programmable Segmentation Register Interface

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range (Hex)
SegCtl2	Register 5 Select 4	CFG5	31:16	0.0 GB to 1.0 GB	0x0000_0000_0000_0000 - 0x0000_0000_3FFF_FFFF
		CFG4	15:0	1.0 GB to 2.0 GB	0x0000_0000_4000_0000 - 0x0000_0000_7FFF_FFFF
SegCtl1	Register 5 Select 3	CFG3	31:16	2.0 GB to 2.5 GB	0xFFFF_FFFF_8000_0000 - 0xFFFF_FFFF_9FFF_FFFF
		CFG2	15:0	2.5 GB to 3.0 GB	0xFFFF_FFFF_A000_0000 - 0xFFFF_FFFF_BFFF_FFFF
SegCtl0	Register 5 Select 2	CFG1	31:16	3.0 GB to 3.5 GB	0xFFFF_FFFF_C000_0000 - 0xFFFF_FFFF_DFFF_FFFF
		CFG0	15:0	3.5 GB to 4.0 GB	0xFFFF_FFFF_E000_0000 - 0xFFFF_FFFF_FFFF_FFFF

Chapter 3

Memory Management Unit

The P6600 core includes a Memory Management Unit (MMU) that translates virtual addresses to physical addresses. The MMU consists of a 16-entry Instruction TLB (ITLB), a 32-entry data TLB (DTLB), 64 dual-entry Variable TLB (VTLB), and a 512 dual-entry Fixed TLB (FTLB).

This chapter contains the following sections:

3.1 Introduction

The MMU translates a virtual address to a physical address before the request is sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. Virtual-to-physical address translation is especially useful for operating systems that must manage physical memory to accommodate multiple tasks active in the same memory, and possibly in the same virtual address space. The MMU also enforces the protection of memory areas and defines the cache protocols.

3.2 Memory Management Unit Architecture

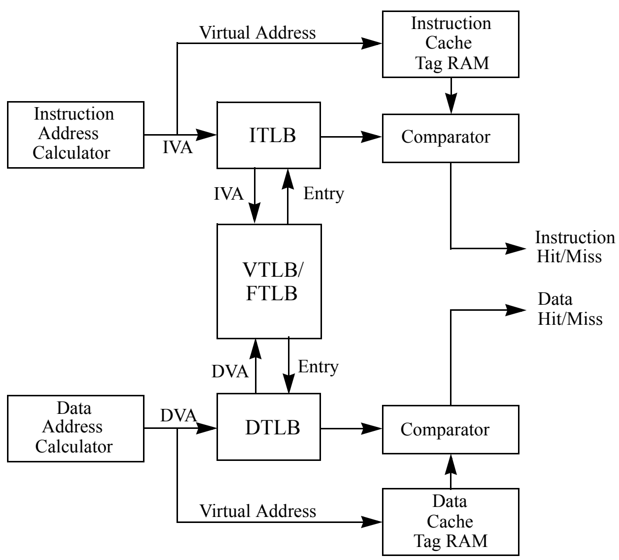
The Memory Management Unit (MMU) in the P6600 core consists of four address-translation lookaside buffers (TLB):

- 16-entry Instruction TLB (ITLB)
- 32 dual-entry Data TLB (DTLB)
- 64 dual-entry Variable Page Size Translation Lookaside Buffer (VTLB)
- Optional 512 dual-entry Fixed Page Size Translation Lookaside Buffer (FTLB)

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

Figure 3.1 shows an overview of the P6600 MMU architecture.

Figure 3.1 Overview of MMU Architecture in the P6600 Core



3.2.1 Instruction TLB (ITLB)

The ITLB is a 16-entry high speed TLB dedicated to performing translations for the instruction stream. The ITLB maps only 4 KB or 16 KB pages. For 4 KB or 16 KB pages, the entire page is mapped in the ITLB. IF the pagesize is larger than 16 KB, then the contents of the larger page are copied into the ITLB on a 16 KB boundary.

The ITLB is managed by hardware and is transparent to software. The larger VTLB/FTLB is used as a backup structure for the ITLB. If a fetch address cannot be translated by the ITLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the ITLB for future use.

The ITLB is functionally invisible to software and is entries are automatically refilled from the VTLB/FTLB when required, and automatically cleared whenever the associated VTLB/FTLB is updated.

3.2.2 Data TLB (DTLB)

The DTLB is a 32 dual-entry high speed TLB dedicated to performing translations for the data stream. The DTLB maps only 4 KB or 16 KB pages. For 4 KB or 16 KB pages, the entire page is mapped in the DTLB.

The DTLB is managed by hardware and is transparent to software. The larger VTLB/FTLB is used as a backup structure for the DTLB. If a load/store address cannot be translated by the DTLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the DTLB for future use.

The DTLB is functionally invisible to software and entries are automatically refilled from the VTLB/FTLB when required, and automatically cleared whenever the associated VTLB/FTLB is updated.

3.2.3 Variable Page Size TLB (VTLB)

The VTLB is a fully associative variable page size translation lookaside buffer with 64 dual entries. The purpose of the VTLB is to translate virtual addresses and their corresponding ASID into a physical memory address. The translation is performed by comparing the upper bits of the virtual address (along with the ASID bits) against each of the entries in the *tag* portion of the VTLB structure. This structure is used to translate both instruction and data virtual addresses.

The VTLB is organized as 64 pairs of even and odd entries. The VTLB implements the following page sizes:

4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, and 256M, 1G, and 4G

The VTLB/FTLB is organized in pairs of page entries to minimize its overall size. Each virtual *tag* entry corresponds to two physical 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 two data entries is used. Since page size 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 selection must be done dynamically during the TLB lookup.

The *PageMask* register is loaded with the desired page size, which is then entered into the TLB when a new entry is written. Thus, operating systems can provide special-purpose maps. For example, a typical frame buffer can be memory-mapped with only one TLB entry. Software can determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back.

The VTLB/FTLB entries are controlled through select CP0 registers. Refer to Section 3.5, "Relationship of TLB Entries and CP0 Registers" for more information.

3.2.4 Fixed Page Size TLB (FTLB)

The 512-entry FTLB is a fixed page size TLB organized as 128 sets and 4-ways. Each set of each way contains dual data RAM entries and one tag RAM entry. If the tag RAM contents matches the requested address, either the low or high RAM location of the dual data RAM is accessed depending on the state of the least-significant-bit (MSB) of the VPN field. Refer to Section 3.5.3, "Address Translation Examples" for more information on VPN2 usage.

The FTLB is organized as 512 pairs of even and odd entries. The FTLB implements the following page sizes:

4K, 16K

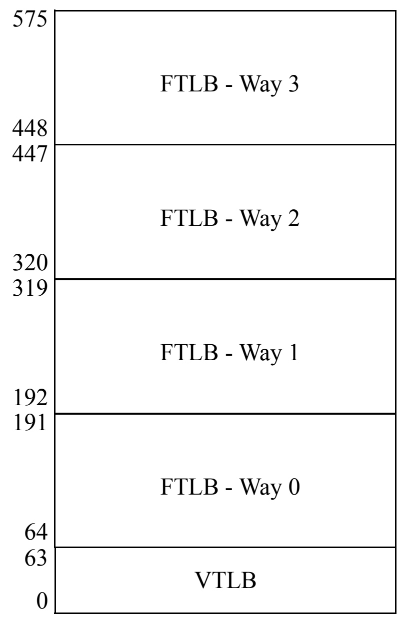
If the FTLB is implemented, the organization is as shown in Table 3.1. Note that all of the entries in the FTLB must be the same page size, either 4K or 16K. The size is determined by the *Config4*$_{FTLB\ Page\ Size}$ field as described in the following table.

Table 3.1 FTLB Configuration Options

FTLB Parameter	Programmable Options	Register Reference
Ways	4 ways	*Config4*$_{FTLB\ Ways}$
Sets	128 sets	*Config4*$_{FTLB\ Sets}$
Page Size	4 KB 16KB	*Config4*$_{FTLB\ Page\ Size}$

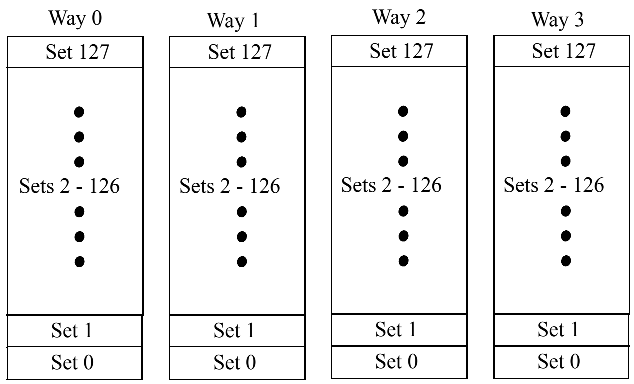
The FTLB resides at the top of the VTLB range as shown in Figure 3.2.

Figure 3.2 P6600 VTLB and FTLB



As shown in Figure 3.3, the 512-entry FTLB contains four ways and 128 sets. Each set of each way contains one dual-entry.

Figure 3.3 FTLB Organization



3.3 MMU Configuration Options

The MMU in the P6600 core can be configured with the following options.

- FTLB enabled/disabled
- MMU type
- MMU size and organization

3.3.1 FTLB Enabled/Disabled

The P6600 core allows software to enable and disable the 512-entry FTLB. This is done via the *FTLBEn* bit in the *Config6* register (CP0 Register 16, Select 6). Depending on how this bit is set, one of the following will occur:

- If the $Config6_{FTLBEn}$ bit is set by software, the FTLB is enabled and the hardware will configure the device accordingly.
- If the $Config6_{FTLBEn}$ bit is cleared by software, the FTLB is disabled. This mode allows the P6600 core to remain backward compatible with existing software. Note that if the $Config6_{FTLBEn}$ bit is cleared, the address translation mechanism acts just like a Joint TLB (JTLB) in previous generation MIPS processors.
- If the $Config6_{FTLBEn}$ bit is not programmed by software, the FTLB is disabled by default because this bit is cleared automatically at reset.

These options are illustrated in the Table 3.2.

Table 3.2 FTLB Enabled of Disabled in the System

$Config6_{FTLBEn}$ **Bit (Set by Software)**	$Config_{MT}$ **Field**[1] **(Set by Hardware)**
1	3’b100 (FTLB Enabled)
0	3’b001 (FTLB Disabled, VTLB Only)

1. See Section 3.3.2, "MMU Type".

Note that the size of the FTLB is fixed at 512 entries. The user cannot implement less than 512 entries if the FTLB is enabled.

3.3.2 MMU Type

The *MT* field of the *Config* register (CP0 Register 16, Select 0) is programmed depending on whether the FTLB is enabled. This is determined by the state of the $Config6_{FTLBEn}$ bit described above. If $Config6_{FTLBEn}$ is cleared, hardware writes a value of 3'b001 to this field. If $Config6_{FTLBEn}$ is set, hardware writes a value of 3'b100 to this field. The kernel code uses this field to determine how to configure the TLB.

The 3-bit $Config_{MT}$ field supports the following two encodings. All other encodings are reserved.

- 3'b001: VTLB only (FTLB disabled)
- 3'b100: VTLB and FTLB present

3.3.3 MMU Size and Organization

The P6600 core uses the following CP0 register fields to determine the size and organization of the MMU. Each of the items below is described in the following subsections.

- Bits 30:25 of the *Config1* register ($Config1_{MMUSIZE}$). Determines VTLB size. The number of VTLB entries is equal to $Config1_{MMUSIZE}$ - 1.
- Bits 12:8 of the *Config4* register ($Config4_{FTLB\ Page\ Size}$). Determines the FTLB page size. If the FTLB is disabled, this field is ignored.
- Bits 7:4 of the *Config4* register ($Config4_{FTLB\ Ways}$). This field determines the number of ways in the FTLB.
- Bits 3:0 of the *Config4* register ($Config4_{FTLB\ Sets}$). This field determines the number of sets per way in the FTLB.

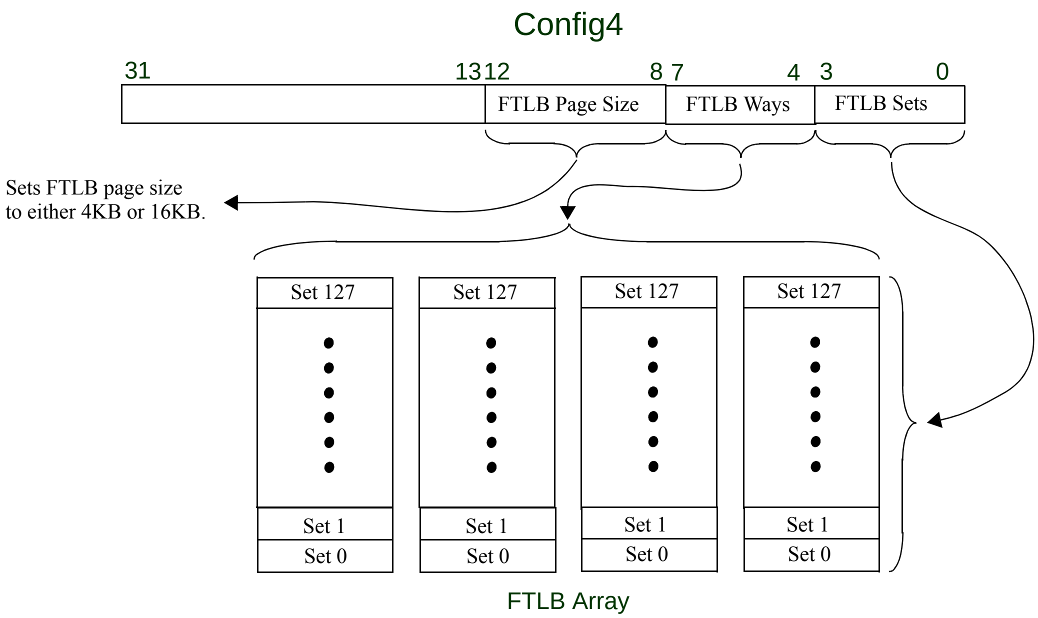
3.3.3.1 Determining VTLB Size

Hardware writes a value of 0x3F into the The 6-bit *MMUSize* field at reset, indicating 64 entries numbered 0 - 63. Note that the number of VTLB entries in the P6600 core is fixed at 64. The user cannot modify this value.

3.3.3.2 FTLB Parameters

Bits 12:0 of this register are used to indicate the FTLB page size ($Config4_{FTLB\ Page\ Size}$), the number of ways ($Config4_{FTLB\ Ways}$), and the number of sets ($Config4_{FTLB\ Sets}$). In the P6600 core, only the FTLB page size is programmable. The number of ways is fixed at 4 and the number of sets is fixed at 128. The page size can be programmed to either 4KB or 16KB pages. This concept is shown in Figure 3.4.

Figure 3.4 Determining the FTLB Characteristics — FTLB Enabled



3.4 Overview of Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB. There is a match when the VPN of the address is the same as the VPN field of the TLB entry after masking out the bits specified by the entries page size, and either:

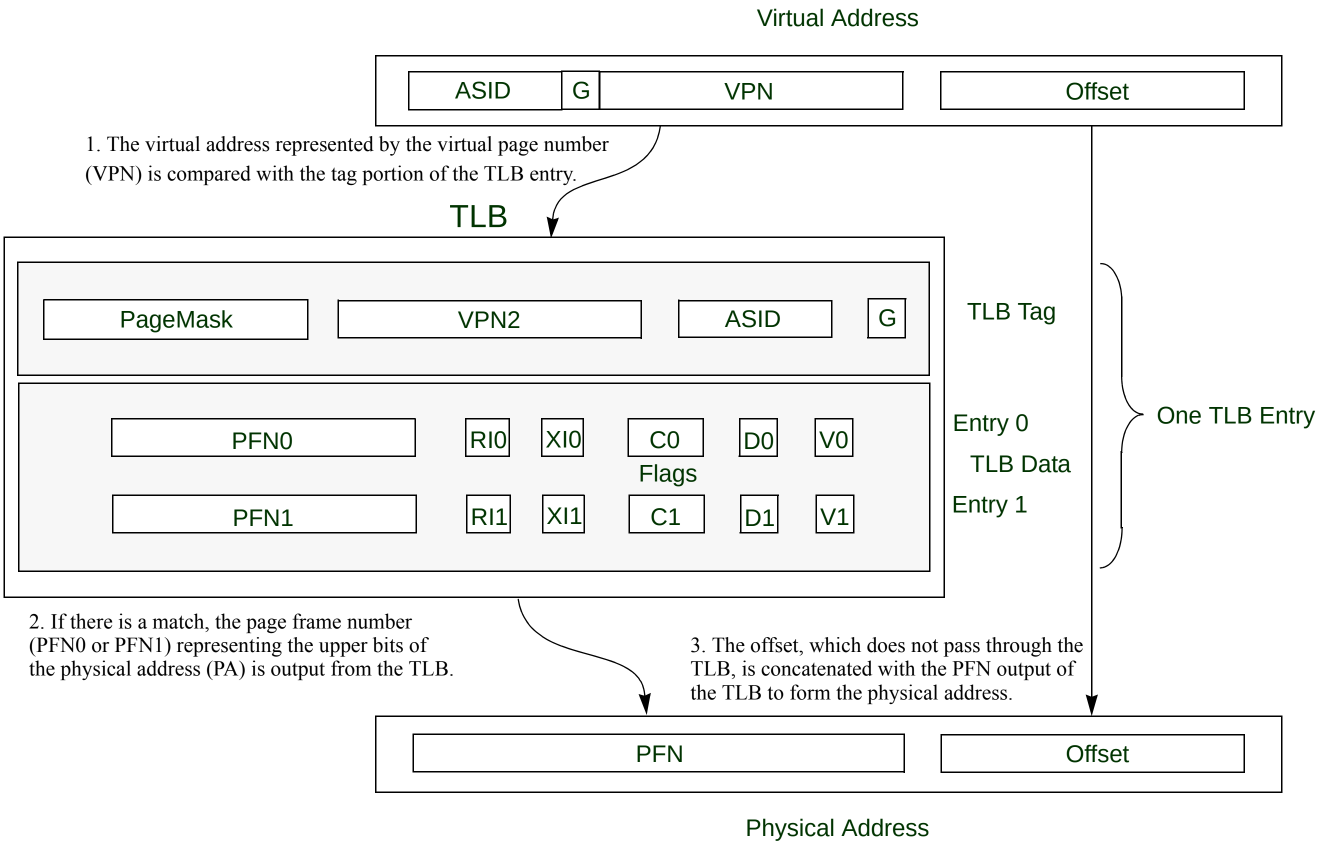
- The Global (G) bit of both the even and odd pages of the TLB entry is set, or
- The Global (G) bit is cleared and the ASID field of the virtual address is the same as the ASID field of the TLB entry

This match is referred to as a TLB *hit*. If there is no match, a TLB *Refill* exception is taken by the processor, and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

Figure 3.5 shows the translation of a virtual address into a physical address. In this figure, the virtual address is extended with an 8-bit ASID, which reduces the frequency of TLB flushes during a context switch. This 8-bit ASID contains the number assigned to that process.

Note that the various register fields used during a TLB translation are managed via CP0 registers as described in Section 3.5, "Relationship of TLB Entries and CP0 Registers".

Figure 3.5 Overview of Virtual to Physical Address Translation



If there is a virtual address match in the TLB, the Physical Frame Number (PFN) is output from the TLB and concatenated with the *Offset* to form the physical address. The *Offset* represents an address within the page frame space. As shown in Figure 3.5, the *Offset* does not pass through the TLB. Note that if the G bit is set, the ASID is ignored and the TLB compares only the VPN portion ofthe virtual address. The G bit is a logical AND of the G bit in the *EntryLo0* and *EntryLo1* registers.

3.4.1 Operating and Addressing Modes

Both the operating mode and the addressing mode of the processor can be selected. The operating mode allows the processor to execute 64-bit operations internally. The addressing mode allows the processor to generate either 32-bit or 64-bit addresses.

3.4.1.1 Operating Modes

The P6600 core can operate in one of the following modes. The mode is determined by the state of the CP0 $Status_{KSU}$ field. Refer to Table 2.13 in Chapter 2 for additional information on the encoding of this field. Note that if the *DM* bit of the *Debug* register is set, the device is placed in debug mode, regardless of the state of the $Status_{KSU}$ field.

Table 3.1 Determining the Operating Mode

Status Register KSU Field	Debug.DM Field	Mode
x	1	Debug mode

Table 3.1 Determining the Operating Mode *(continued)*

Status Register KSU Field	Debug.DM Field	Mode
2’b00	0	Kernel mode
2’b01	0	Supervisor mode
2’b10	0	User mode

Once in the appropriate operating mode, the processor can execute either 32-bit or 64-bit operations. This information can be obtained from the CP0 Status register as shown in the following table.

Table 3.2 Determining the Addressing Mode

Status.KX	Status.SX	Status.UX	Status.PX	Mode
0	0	0	0	32-bit compatibility mode.
1	0	0	0	Access to 64-bit kernel address space is enabled. Uses the XTLB refill exception on a TLB Miss for a kernel address.
1	1	0	0	Access to 64-bit Kernel and 64-bit Supervisor address space enabled. Uses the XTLB refill exception on a TLB Miss for a kernel/supervisor address.
1	1	1	0	Access to 64-bit Kernel/Supervisor/User address space enabled. Uses the XTLB refill exception on a TLB Miss for any mapped address.
1	1	0	1	Access to 64bit Kernel/Supervisor address space enabled. 64-bit operations are enabled in User space, but no access to 64-bit address space. Uses the TLB Refill exception on a TLB Miss.
1	1	1	1	Access to 64bit Kernel/Supervisor/Use address space enabled. Uses the XTLB refill exception on a TLB Miss for any mapped address.

3.4.2 Address Translation in 64-bit Mode

Figure 3.6 shows a flow diagram of the 64-bit address translation process for a 4 KByte page size. In the MIPSr6 architecture, VA[63:62] are used to perform the memory segmentation function to indicate which of the following area of VA space is being accessed.

- Kernel: VA[63:62] = 11
- XKPhys: VA[63:62] = 10
- Supervisor: VA[63:62] = 01
- User: VA[63:62] = 00

In the P6600 core, which implements a 48-bit virtual address, VA[63:62] are appended to the end of the VA and reside in VA[49:48]. The remaining 36 bits of the address (VA[47:12]) represent the virtual page number (VPN) at the segment of memory determined by VA[49:48]. The width of the *Offset* is defined by the page size. For more information, refer to Table 3.14 later in this chapter.

In the figure below, VPN 47:12 represent the virtual address. Bits 49:48 are the Region bits and are used to divide the virtual address space into four segments:

Figure 3.6 64-bit Virtual Address Translation — 4 KB Page Size

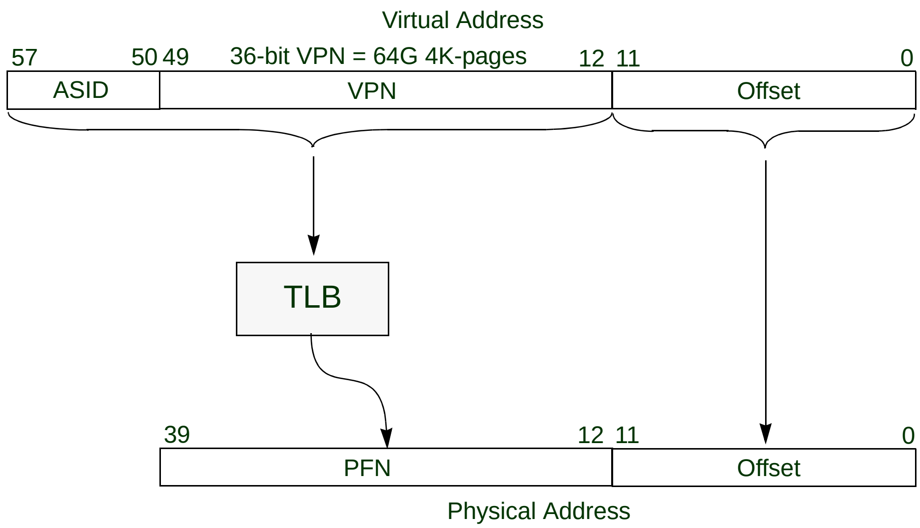
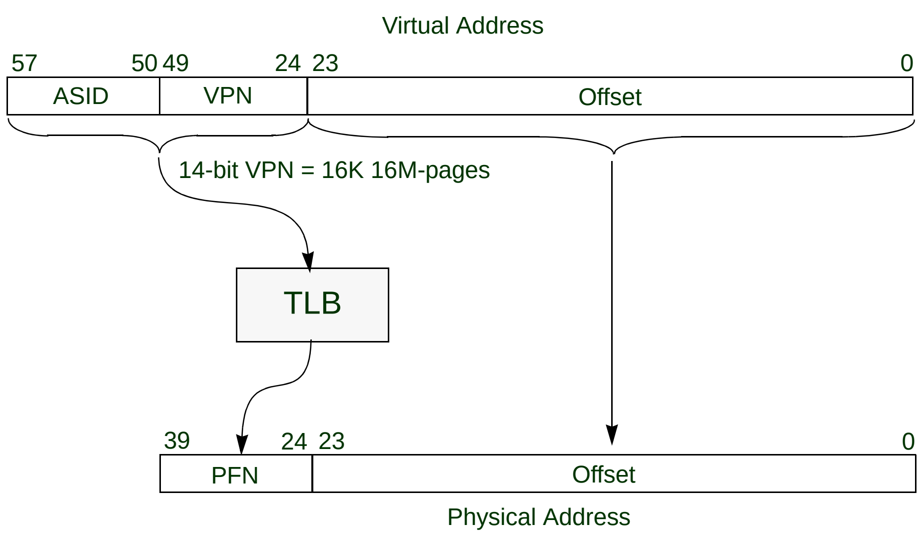


Figure 3.7 shows a flow diagram of the 64-bit address translation process for a 16 MByte page size. The width of the *Offset* is defined by the page size. The remaining bits of the address represent the virtual page number (VPN). Note that the P6600 core can support page sizes up to 4 GB, which yields a 32-bit offset and a 16-bit VPN.

Figure 3.7 64-bit Virtual Address Translation — 16 MB Page Size



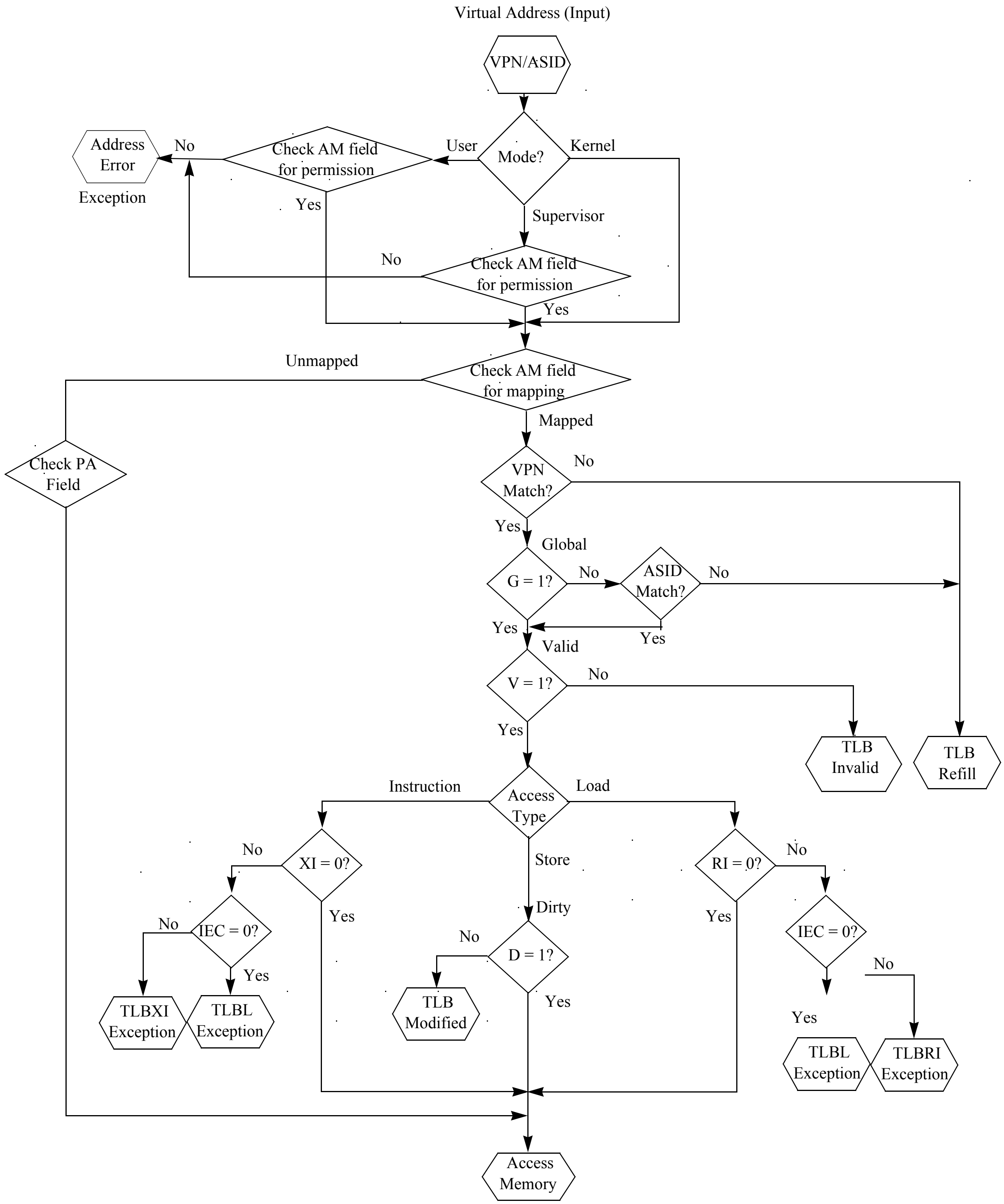
3.4.3 Address Translation in 32-bit Mode

In the P6600 core, all address translations are performed on 64-bit values. To maintain backward compatibility, addresses translation can be done on 32-bit addresses by sign-extending the unused bits 47:32. The 64-bit address space maps to the 32-bit compatibility mode as described in Figure 3.31.

3.4.4 Address Translation Flow

During an address translation, the hardware checks for various conditions such as the addressing mode (user, kernel etc.), access permissions based on the mode, the access type (load/store, etc), and the state of selected bits in the TLB entry. If one or more of the conditions for translation are not met, a TLB exception is taken. This concept is shown in Figure 3.8.

Figure 3.8 Address Translation Flow



3.5 Relationship of TLB Entries and CP0 Registers

Each TLB entry in the VTLB/FTLB consists of a tag portion and dual-data portion as shown in Figure 3.9. In this figure, the following registers are used to manage the TLB entries.

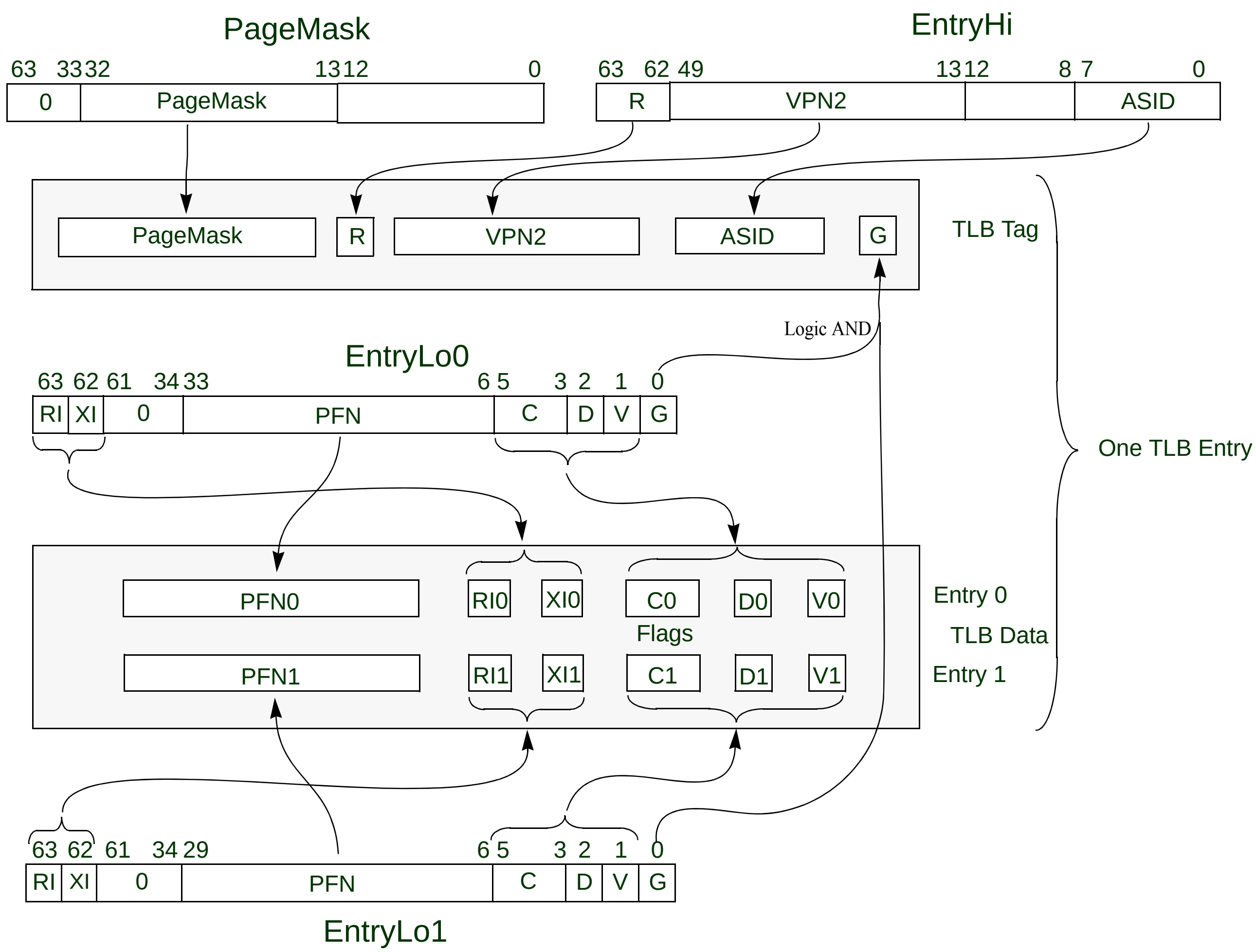
- *EntryLo0* (CP0 Register 2, Select 0)
- *EntryLo1* (CP0 Register 3, Select 0)
- *EntryHi* (CP0 Register 10, Select 0)
- *PageMask* (CP0 Register 5, Select 0)

In order to fill an entry in the VTLB/FTLB, software executes a **TLBWI** or **TLBWR** instruction (see Section 3.17). 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, C0, D0, V0, RI, XI, and G bits are set in the CP0 *EntryLo0* register.
- PFN1, C1, D1, V1, RI, XI, 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 3.9 Relationship Between CP0 Registers and TLB Entries



3.5.1 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 3.9.

3.5.1.1 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 that on a TLB-related exception, the *VPN2* field 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.

3.5.1.2 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.

3.5.1.3 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 VTLB, the P6600 core allows page sizes of 4 Kbytes up to 4 Gbytes in multiples of four. For the FTLB, the P6600 core allows page sizes of 4 Kbytes and 16 Kbytes. 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[32:13] field, and the corresponding page size is shown in Table 3.3. For the PageMask[32:13] field, note that the bits are set two at a time from the least significant bit (LSB) to the most significant bit (MSB).

Table 3.3 PageMask Value and Corresponding Page Size

33-bit PageMask Register Value	PageMask[32:13]	Page Size	Even/Odd Bank Select Bit
0x0_0000_0000	0x00_0000_0000_0000_00	4 KBytes	VAddr[12]
0x0_0000_6000	0x00_0000_0000_0000_11	16 KBytes	VAddr[14]
0x0_0001_E000	0x00_0000_0000_0011_11	64 KBytes	VAddr[16]
0x0_0007_E000	0x00_0000_0000_1111_11	256 KBytes	VAddr[18]
0x0_001F_E000	0x00_0000_0011_1111_11	1 MByte	VAddr[20]
0x0_007F_E000	0x00_0000_1111_1111_11	4 MBytes	VAddr[22]
0x0_01FF_E000	0x00_0011_1111_1111_11	16 MBytes	VAddr[24]
0x0_07FF_E000	0x00_1111_1111_1111_11	64 MBytes	VAddr[26]
0x0_1FFF_E000	0x11_1111_1111_1111_11	256 MBytes	VAddr[28]
0x0_7FFF_E000	0x1111_1111_1111_1111_11	1 GByte	VAddr[30]
0x1_FFFF_E000	0x11_1111_1111_1111_1111_11	4 GBytes	VAddr[32]

Note that the 4 KByte and 16 KByte entries in the above table correspond to the VTLB and the FTLB. All other entries correspond to the VTLB only.

3.5.1.4 Global (G) Bit

The ‘G’ (global) bit in the tag entry is a logical AND between the *G* bits of the *EntryLo0* and *EntryLo1* registers. 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 that since 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.

3.5.2 TLB Data Entry

The data portion of the TLB entry contains the data and associated flag bits for the corresponding tag entry. This section describes each field of the TLB data entry shown in Figure 3.9.

3.5.2.1 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 28-bit *PFN*, together with the lower 12 bits of address that are not translated, make up the 40-bit physical address.

3.5.2.2 Flag Fields (C, D, V, RI, and XI)

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 non-coherent, uncached accelerated, write-back, 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.

RI bit: The 'read inhibit' flag. If this bit is set in a TLB entry, any attempt to read data on the virtual page causes a *TLBRI* exception depending on the state of the $PageGrain_{IEC}$ bit, even if the *V* (Valid) bit is set. Since the $PageGrain_{IEC}$ bit is always set, a *TLBRI* exception is taken. Note that the *RI* bit is writable only if the *RIE* bit of the *PageGrain* register is set.

XI bit: The 'execute inhibit' flag. If this bit is set in a TLB entry, any attempt to fetch an instruction from the virtual page causes a *TLBXI* exception depending on the state of the $PageGrain_{IEC}$ bit, even if the *V* (Valid) bit is set. Since the $PageGrain_{IEC}$ bit is always set, and *TLBXI* exception is taken. Note that the *XI* bit is writable only if the *XIE* bit of the *PageGrain* register is set.

3.5.3 Address Translation Examples

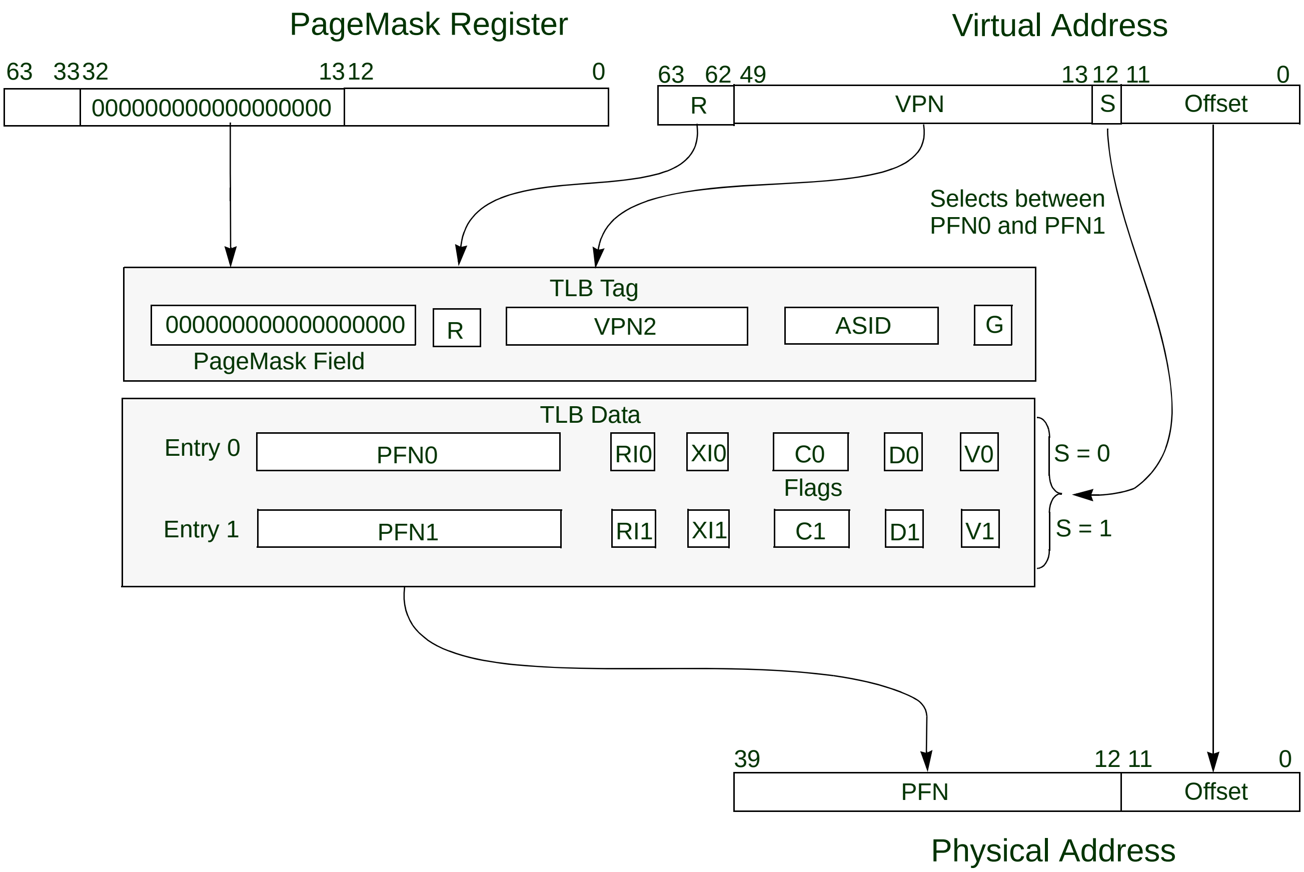
As shown in Figure 3.9, there are two PFN values for each tag match. Which of them is used is determined by the lowest-order bit of the VPN field of the address. So in standard form (using 4 KByte pages) each entry translates an 8 KByte region of virtual address, but each 4Kbyte page can be mapped onto any physical address (with any permission flag bits). This concept is described in the following subsections.

4 KByte Page Size Example

In a 4KB 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 36 bits of the virtual address, along with the Region bits VA[63:62], are used as a pointer to the page table.

The upper 36 bits of virtual address and the Region bits pass through the TLB to generate the corresponding physical address. As described in Section 3.4, the P6600 core implements a dual-entry VTLB/FTLB 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 figure below). In a 4 KByte page example, this equates to bit 12. This is shown in Figure 3.10.

Figure 3.10 Selecting Between PFN0 and PFN1 — 4 KByte Page Size



As shown in Figure 3.10, the *PageMask* field is derived from the *PageMask* register and is used to determine the page size for the application. Since the P6600 core supports VTLB/FTLB page sizes in multiples of four (4 KByte, 16 KByte, 64 KByte, etc. up to 4 GByte), 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 3.3 for a list of valid *PageMask* values.

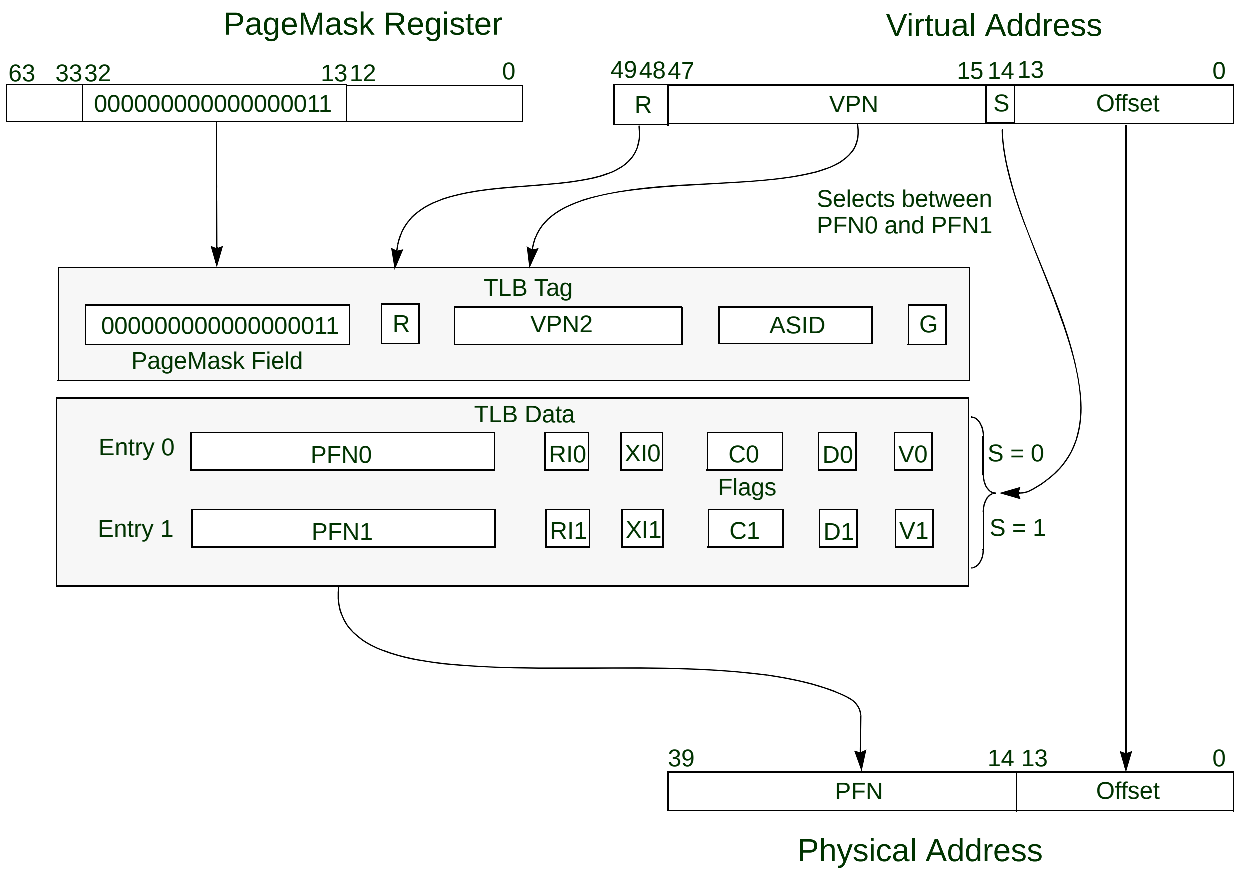
In the above example, all of the PageMask field bits are 0, indicating a 4 KByte page size. For a 16 KByte page size, bits 12 and 13 of the PageMask field would be set. This concept is described below.

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 34 bits of the virtual address, along with the two Region bits VA[63:62], are used as a pointer to the page table.

As described in Section 3.4, the P6600 core implements a dual-entry VTLB/FTLB 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 figure below). In a 16 KByte page example, this equates to bit 14. This is shown in Figure 3.11.

Figure 3.11 Selecting Between PFN0 and PFN1 — 16 KByte Page Size



As shown in Figure 3.11, 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 the above example, the lower 2 bits of the *PageMask* field bits are 11, indicating a 16 KByte page size. Refer to Table 3.3 for a list of valid *PageMask* values.

3.6 Indexing the VTLB and FTLB

In the P6600 core, the VTLB is 64 dual entries, and the FTLB is 512 dual entries. If the FTLB is enabled, a 10-bit value is used to index all 576 dual entries of the VTLB and FTLB. If the FTLB is disabled, a 6-bit value is used to index the 64 dual entries of the VTLB. This is shown in Figure 3.12. This value is stored in the *Index* register (CP0 register 0, Select 0).

Figure 3.12 Index Register Format Depending on TLB Size

31	30 … 6	5 … 0
P	0	Index (VTLB only)

31	30 … 10	9 … 0
P	0	Index (VTLB + FTLB)

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 which fails to find a match for the specified virtual address sets bit 31 of *Index* register.

3.7 Hardware Page Table Walker

Page Table Walking is the process by which a Page Table Entry (PTE) is located in memory. Hardware acceleration for page table walking is an optional feature in the architecture. The mechanism can be used to replace the software handler for the TLB or XTLB Refill condition. The existence of the Hardware Page Walking feature is denoted when $Config3_{PW}$ = 1.

The Hardware Page Table Walker includes the following enhancements to the normal page table entry format.

1. Huge Page support in directories (non-leaf levels of the Page Table hierarchy), and Base Page Size for the (Page Table Entry (PTE) levels (leaf levels of the Page Table hierarchy). This is the baseline definition. Inferred size PTEs are supported at non-leaf levels.

2. A reserved field has been added to PTEs. This field is for future extensions.

A Huge Page may logically be specified in two ways:

1. A Huge Page is a region composed of two power-of-4 pages which have adjacent virtual and physical addresses. Since the even page and the odd page are derived from a single directory entry, they will both inherit the same attributes and all but one of the address bits from the single directory entry. The memory region is divided evenly between the even page and the odd page. The physical address held within the directory entry is aligned to 2 x size of the page (which is a power of 4). This is distinct from *EntryLo0* and *EntryLo1* pairs in the Page Table which are only guaranteed to be adjacent in virtual, but not physical address. They may also have differing page attributes. This method is known as **Adjacent Pages** since the *EntryLo0/1* physical addresses are both derived from one entry and have to be adjacent in the physical address space. This is the default method that is supported by this specification. If an implementation chooses to support Huge Pages in the directory levels, then the Adjacent Page method must be implemented.

2. Where a Huge Page is itself a power-of-4 page, it is handled in exactly the same manner as a Base Page in the Page Table. For this case, one directory entry is used for the even page and the adjacent directory entry is used for the odd page. The physical address held within the directory entry is aligned to the size of the page (which is a power of 4). This method is known as **Dual Pages** since each PFN does not have to be adjacent to each other. If an implementation chooses to support Huge Pages in the directory levels, then the Dual Page method is an additional option.

Examples of power-of-4 regions (start with 1KB and multiply by 4 a number of times): 256MB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB.

Examples of 2x power-of-4 regions (start with 1KB and multiply by 4 a number of times; then multiple by 2) 512MB, 2MB, 8MB, 32MB, 128MB, 512MB, 2GB.

Huge Page Support is optional and is indicated by $PWCtl_{Hugepg}$ = 1. If an Implementation supports Huge Pages in the directory levels, it must support the Adjacent Page method. The Dual Page method is optional if Huge Pages are supported. The implementation of Dual Page method is indicated by $PWCtl_{DPH}$=1.

3.7.1 Multi-Level Page Table Support

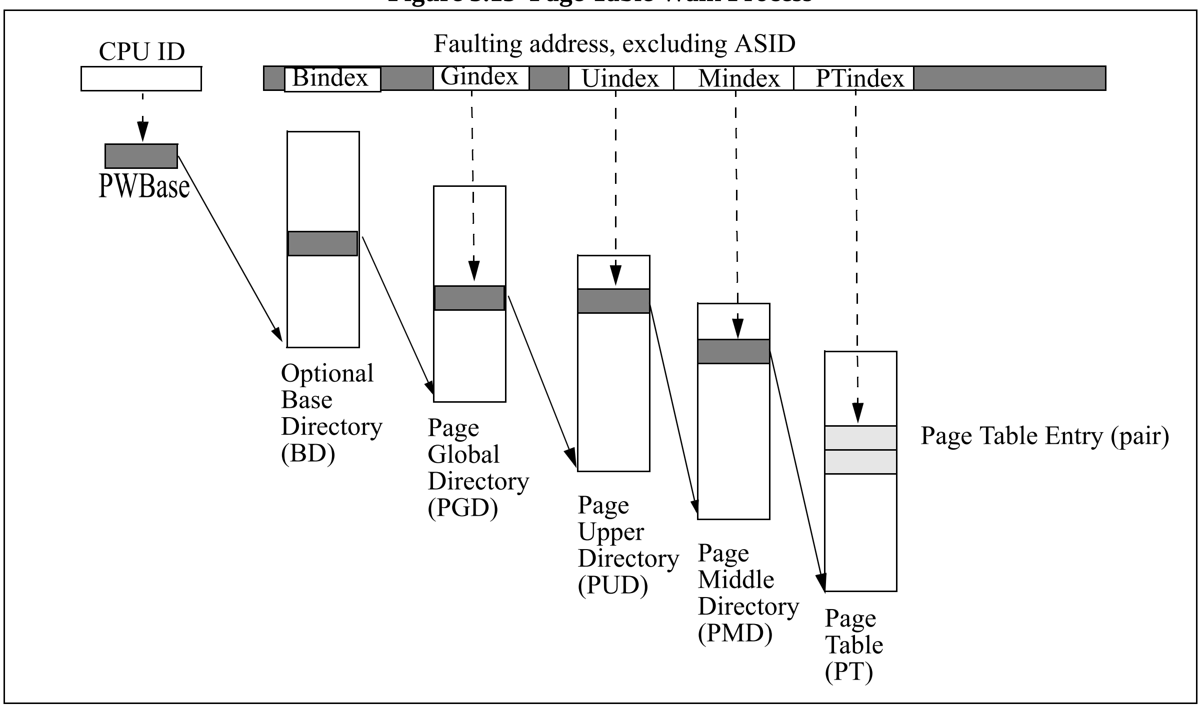
The hardware page table walking system specifies a mechanism for refilling the TLB, independent of the *Context* and XContext registers. Four additional coprocessor 0 registers are added.

- The *PWBase* register specifies the page table base.
- The *PWField* and *PWSize* registers specify address generation for up to four levels of page tables.
- The *PWCtl* register controls the behavior of the Page Table Walker. These registers also configure the separation between Page Table Entries (PTEs) in memory and post-load shifting of PTEs.

A multi-level page table system contains multiple levels, the lowest of which are Page Tables. A Page Table is an array of Page Table Entries. Levels above the Page Tables are known as Directories. A Directory consists of an array of pointers. Each pointer in a Directory is either to another Directory or to a Page Table.

The next figure shows an example of a multi-level page table structure.

Figure 3.13 Page Table Walk Process



Each executing process is typically associated with a separate page table base pointer (*PWBase*). In a uniprocessor system, only one process is active at once. Where multiple CPUs are in use, multiple processes execute simultaneously - thus one page table base pointer is required per CPU. The term 'page table base' refers to the start of a Page Global Directory.

A typical page table structure consists of:

- A *PWBase* register, containing the base of the Page Global Directory.
- Page Global Directories, indexed by upper bits from the faulting address, containing pointers to Page Upper Directories.

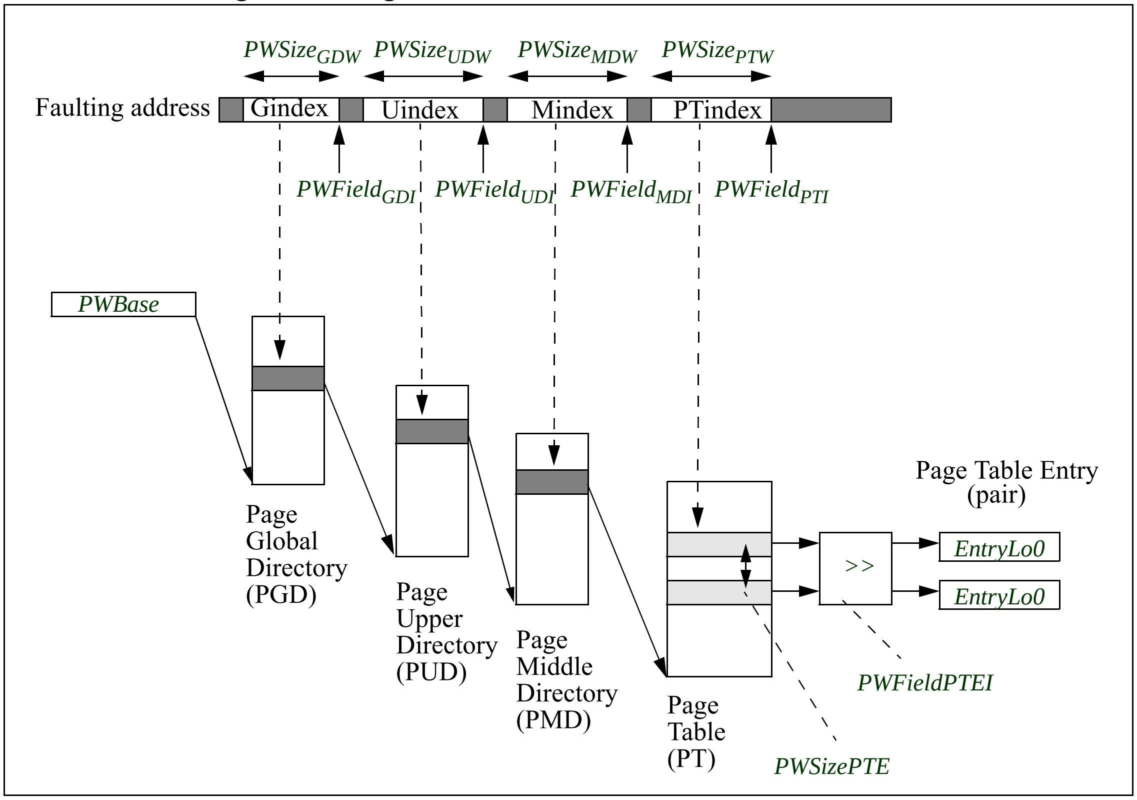
- Page Upper Directories, indexed by bits from the faulting address, containing pointers to Page Middle Directories.

- Page Middle Directories, indexed by bits from the faulting address, containing pointers to Page Tables.

- Page Tables, indexed by bits from the faulting address, containing Page Table Entry (PTE) pairs.

Figure 3.14 shows the registers and fields used by the page table walking scheme for a four level page table structure.

Figure 3.14 Page Table Walk Process and COP0 Control fields



Hardware page table walking is performed when enabled and a TLB or XTLB refill condition is detected.

Memory reads during hardware page table walking are performed as if they were kernel-mode load instructions. Addresses contained in the *PWBase* register and in memory-resident directories are virtual addresses.

Physical addresses and cache attributes are obtained from the Segment Configuration system when $Config3_{SC}$ = 1, or from the default MIPS segment system when $Config3_{SC}$ = 0.

The hardware page walk write should treat the multiple-hit case the same as a TLBWR. Assuming that the write by design cannot detect all duplicates, then a preferred implementation is to invalidate the single duplicate and then write the TLB. A Machine Check exception may subsequently be taken on a TLBP or lookup of TLB.

If a synchronous exception condition is detected during the hardware page table walk, the hardware walking process is aborted and a TLB or XTLB Refill exception will be taken. This includes synchronous exceptions such as Address Error, Precise Debug Data Break and other TLB or XTLB exceptions resulting from accesses to mapped regions.

If an asynchronous exception is detected during the hardware page table walk, the hardware walking process is aborted and the asynchronous exception is taken. This includes asynchronous exceptions such as NMI, Cache Error, and Interrupts. It also includes the asynchronous Machine Check exception which results from multiple matching entries being present in the TLB following a TLB write.

If an exception is detected during the hardware page table walk, the hardware walking process is aborted and the exception is taken. This includes exceptions such as NMI, Cache Error, and Interrupts. It also includes the Machine Check exception which results from multiple matching entries being present in the TLB following a TLB write.

On the 64-bit P6600 core, the hardware page table walk can be used to accelerate TLB or XTLB refills for either 32-bit or 64-bit address regions, but not both. The PWSize.PS field controls whether pointers within directories are treated as 32- or 64-bit addresses.

The selection between TLB and XTLB Refill exception is determined from the faulting address and the UX, SX and KX bits in the Status register.

Hardware page table walking is performed as follows:

1. A temporary pointer is loaded with the contents of the *PWBase* register

2. The native pointer size is set to 4 or 8 bytes (32 or 64 bits) depending on the state of CP0 PWSIZE.PS register field

3. Check if hardware table walk is allowed to walk on a MIPS64 address. Depending on the operating mode one of the following CP0 register bits must be set; PWCtl.XK (kernel), PWCtl.XS (supervisor), PWCtl.XU (user).

4. If the Global Directory is disabled by $PWSize_{GDW} = 0$, skip to the next step.

   - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

   - Extract $PWSize_{GDW}$ bits from the faulting address, with least-significant bit $PWField_{GDI}$. This is the Global Directory index (Gindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Global Directory.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

5. If the Upper Directory is disabled by $PWSize_{UDW} = 0$, skip to the next step.

   - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

   - Extract $PWSize_{UDW}$ bits from the faulting address, with least-significant bit $PWField_{UDI}$. This is the Upper Directory index (Uindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Upper Directory.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

6. If the Middle Directory is disabled by $PWSize_{MDW}$ = 0, skip to the next step.

   - If Huge Pages are supported, check PTEVld bit to determine if entry is PTE. If PTEVld bit is set, write Huge Page into TLB (details left out for brevity, read pseudo-code at end of this section). Page Walking is complete after Huge Page is written to TLB.

   - Extract $PWSize_{MDW}$ bits from the faulting address, with least-significant bit $PWField_{MDI}$. This is the Middle Directory index (Mindex). Logical OR onto the temporary pointer, after multiplying (shifting) by the native pointer size. The result is a pointer to a location within the Middle Directory.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is placed into the temporary pointer. If an exception is detected, abort.

   - The temporary pointer now contains the address of the Page Table to be used.

7. Extract $PWSize_{PTW}$ bits from the faulting address, with least-significant bit $PWField_{PTI}$ This is the Page Table index (PTindex). Multiply (shift) by the native pointer size, then multiply (shift) by the size of the Page Table Entry, specified in $PWSize_{PTEW}$.

   - The temporary pointer now contains the address of the first half of the Page Table Entry.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is logically shifted right by $PWField_{PTEI}$ bits. This is the first half of the Page Table Entry. If an exception is detected, abort.

8. In the temporary pointer, set the bit located at bit location $PWField_{PTEI}$-1.

   - The temporary pointer now contains the address of the second half of the Page Table Entry.

   - Perform a memory read from the address in the temporary pointer, of the native pointer size. The returned value is shifted right by $PWField_{PTEI}$ bits. This is the second half of the Page Table Entry. If an exception is detected, abort.

9. Write the two halves of the Page Table Entry into the TLB, using the same semantics as the TLBWR (TLB write random) instruction.

10. Continue with program execution.

Coprocessor 0 registers which are used by software on a TLB refill exception are unused by the hardware page table walking process. The registers and fields used by software are *BadVAddr*, *EntryHi, PageMask, EntryLo0*, *EntryLo1,,* $Context_{BadVPN2}$, and X$Context_{BadVPN2.}$

3.7.2 PTE and Directory Entry Format

All entries are read from in-memory data structures. There are three types of entries in the baseline definition: Directory Pointer, Huge Page non-leaf PTE of inferred size, and leaf PTE of base size. For options other than baseline, the entry type is a function of the table level and the PTEvld field of an entry. For all but the last level table (leaf level), the PTEvld bit is 0 for directory pointers to the next table and 1 for PTEs. In the leaf table, the entry is always a PTE and the PTEvld bit is not used by Hardware Walker. The $PWCtl_{HugePg}$ register field indicates whether Huge Page non-leaf PTEs are implemented.

All PTEs are shifted right by $PWField_{PTEI}$ -2 (shifting in zeros at the most significant bit) and then rotated right by 2 bits before forming the page-walker equivalents of *EntryLo0* and *EntryLo1* values. These operations are used to remove the Software-only bits and placing the RI and XI protection bits in the proper bit location before writing the TLB. If the RI and XI bits are implemented and enabled, the HW Page Walker feature requires the RI bit to be placed right of the G bit in the PTE memory format. Similarly, it is required that the XI bit to be placed right of the RI bit in the PTE memory format.

Note that the bit position of PTEvld is not fixed at 0. It can be programmed by the $PWCtl_{Psn}$ field. If non-leaf PTE entries are available, there will already be a bit used by the software TLB handler to distinguish non-leaf PTE entries from directory pointers. Normally, the PTEvld bit is configured to point to that software bit within the PTE.

A possible programming error to avoid is placing the PTEvld bit within the Directory Pointer field, as any of those address bits may be set and thus not appropriate to be used to distinguish between a Directory Pointer or a non-leaf PTE.

The following figures show an example of 4-byte pointers or PTE entries. The 4-byte width is configured by having$PWSIze_{PTEW}$=0. In this example, 4bits are used for Software-only flags. The following figures assume a PTE format based on $PWCtl_{Psn}$=0, $PWField_{PTEI}$=6 and a Base Page Size of 4k for simplicity.

Figure 3.15 4-byte Leaf PTE

63 12	11 9	8	7	6	5	4	3..0	Comment
PFN	C	D	V	G	RI	XI	S/W Use	Page Size=Base

Figure 3.16 4-byte Non-Leaf PTE Options

63 16	15 12	11 9	8	7	6	5	4	3..0	Comment
PFN	Reserved (must be 0)	C	D	V	G	RI	XI	S/W Use	Page Size=HgPgSz PTE format in memory

63 16	15 12	11 9	8	7	6	5	4	3..1	0	
PFN	Reserved (must be 0)	C	D	V	G	RI	XI	Unused by HW	PTEvld =1	Page Size=HgPgSz PTE format interpreted by HW Page Walker; PTEvld configured to be at bit 0

63 12	11 1	0	
Dir Pointer 63:12	0	PTEvld =0	Directory Ptr format interpreted by HW Page Walker; PTEvld configured to be at bit 0

After shifting out the software bits (3..0) (shifting in zeros at the most significant bit) and then rotating *RI* and *XI* fields into bits 31:30, the PTE matches the *EntryLo* register format. In the non-Leaf PTE, 4-bits which are just left of the *C* field are reserved for future features.

Figure 3.17 4-Byte Rotated PTE Formats

Comment	63	62	60 … 6	5..3	2	1	0	Comment
Leaf PTE	RI	XI	PFN	C	D	V	G	Page Size=Base

Comment	63	62	60 … 10	9:6	5..3	2	1	0	Comment
Non-leaf PTE	RI	XI	PFN	Reserved (must be 0)	C	D	V	G	Page Size=HgPgSz

The following figures show an example of 8-byte pointers or PTE entries. The 8-byte width is configured by having$PWSize_{PTEW}$=1, or by having $PWSize._{PTEW}$=1.

This example uses 4-bits for Software-only flags. The use of the wider PTE allows for the use of more *PFN* bits to be used for addressing - the 8-byte PTE format is required when more than 32-bits of physical addressing is to be implemented. Both the non-leaf PTE and directory pointer both take 8-bytes of memory space, though only 32-bits are actually used for the memory address. The following figures assume a PTE format based on $PWCtl_{Psn}$=0, $PWField_{PTEI}$=6 and a Base Page Size of 4k for simplicity.

Figure 3.18 8-byte Leaf PTE

63 … 40	39 … 16	15.13	12	11	10	9	8	7..0	Comment
PFN	PFN	C	D	V	G	RI	XI	S/W Use	Page Size=Base

Figure 3.19 8-Byte Non-leaf PTE Options

63 … 40	39 … 20	19 … 16	15.13	12	11	10	9	8	7..0	Comment
PFNX	PFN	Reserved (must be 0)	C	D	V	G	RI	XI	S/W Use	Page Size=HgPgSz PTE format in memory

63 … 40	39 … 20	19 … 16	15.13	12	11	10	9	8	7..1	0	Comment
PNFX	PFN	Reserved (must be 0)	C	D	V	G	RI	XI	Unused by HW	PTEv ld=1	Page Size=HgPgSz PTE format interpreted by HW Page Walker

63 … 12	11 … 1	0	Comment
Directory Ptr	0	PTEv ld=0	Directory Pointer format interpreted by HW Page Walker

After the software bits (7..0) are right shifted away (shifting in zeros at the most significant bit) and the RI and XI fields are rotated to bits 63:62, the PTE matches the *EntryLo* register format. By setting $PWSIze_{PTEW}$=1 to denote 8-byte PTE entries, the shift operation is done on the entire 8 byte PTE, but only the lower 4-bytes are written into the TLB. In the non-Leaf PTE, 4-bits which are just left of the *C* field are reserved for future features.

Figure 3.20 8-Byte Rotated PTE Formats

Comment	63	62	61 53	52 30	29 6	5 3	2	1	0	Comment
Leaf PTE	RI	XI	FILL	PFNX	PFN	C	D	V	G	Page Size=Base

	63	62	61 53	52 30	29 10	9 6	5 3	2	1	0	
Non-leaf PTE	RI	XI	FILL	PFNX	PFN	*Rsvd (must be 0)*	C	D	V	G	Page Size=HgPgSz

Leaf PTEs always occur in pairs (*EntryLo0* and *EntryLo1*). However, non-leaf PTEs (ones which occur in the upper directories) can occur either in pairs (if Dual Page method is enabled) or occur with just one entry (Adjacent Page method).

For the Adjacent Page method, the single non-leaf PTE represent both *EntryLo0* and *EntryLo1* values. When the walker populates the EntryLo registers for a PTE in a directory, the least significant bit above the page size is 0 for *EntryLo0* and 1 for *EntryLo1*. That is, *EntryLo0* and *EntryLo1* represent adjacent physical pages.

For the Dual Page method, the two PTEs are read from the directory level by the Hardware Page Walker.

For Huge Page handling, the size of the Huge Page is inferred from the directory level in which the Huge Page resides. For the Adjacent Page Method, the size of each individual PTE in *EntryLo0* and *EntryLo1* as synthesized from the single Huge Page is always half the inferred size.

If the inferred page size is 2 x power-of-4, then the Adjacent Page Method is used.

If the inferred page size is a power-of-4, then the Dual Page Method is used (if the Dual Page Method is implemented). If the Dual Page method is implemented ($PWCtl_{DPH}$=1), it is implementation-specific whether the PTEVld bit is checked for the second PTE when it is read from memory for writing the second TLB page. The recommended behavior is to check this second PTEVld bit and if it is not set, a Machine Check exception is triggered. The $PageGrain_{MCCause}$ register field is used to differentiate between different types of Machine Check exceptions.

If the inferred Huge Page size is power-of-4, and the Dual Page Methods is not implemented, it is implementation-specific whether a Machine Check is reported.

An example of Huge Page handling follows. It assumes a leaf PTE size of 4KB.

- PMD Huge Page = 2^9 ($PWSize_{PTW}$) * 2^12 ($PWField_{PTI}$) = 2^21 = 2MB. Each EntryLo0/1 page is 1MB, which is a power-of-4 and use the Adjacent Page method.

- PUD Huge Page = 2^10 ($PWSize_{MDW}$) * 2^9 ($PWSize_{PTW}$) * 2^12 ($PWField_{PTI}$) = 2^31 = 2GB. Each EntryLo0/1 page is 1GB, which is a power-of-4 and would use the Adjacent Page method. Note that the index into PMD has been extended to 10 bits from 9 bits. Each PMD table thus has 1K entries instead of the typical 512 entries.

3.7.3 Hardware Page Table Walking Process

The hardware page table walking process is described in pseudocode as follows:

```
/* Perform hardware page table walk
 *
 * Memory accesses are performed using the KERNEL privilege level.
 * Synchronous exceptions detected on memory accesses cause a silent exit
 * from page table walking, resulting in a TLB Refill exception.
```

```
 *
 * Implementations are not required to support page table walk memory
 * accesses from mapped memory regions. When an unsupported access is
 * attempted, a silent exit is taken, resulting in a TLB Refill exception.
 *
 * Note that if an exception is caused by AddressTranslation or LoadMemory
 * functions, the exception is not taken, a silent exit is taken,
 * resulting in a TLB Refill exception.
 *
 * For readability, this pseudo-code does not deal with PTEs of different widths.
 * In reality, implementations will have to deal with the different PTE
 * and directory pointer widths.

 */
subroutine PageTableWalkRefill(vAddr) :


   if (Config3_PW = 0) then
      return(0)  # walker is unimplemented

   if (PWCtl_PWEn=0) then
      return (0)  # walker is disabled

   if !((PWCtl_PWDirExt & PWSize_BDW>0|PWSize_MDW>0)(PWSize_GDW>0|PWSize_UDW>0|PWSize_MDW>0) then
      return (0) # no structure to walk

   if !(PWSize_PS=1 & (PWCtl_XK=1 | PWCtl_XS=1| PWCtl_XU=1))then
      return (0) # no segment to map


         # Initial values
   found  ← 0

   encMask ← 0
   HugePage ← False
   HgPgBDhit ← False
   HgPgGDhit ← False
   HgPgUDhit ← false
   HgPgMDhit ← false

   # Native pointer size
   if PWSize_PS=0 then
   NativeShift   ← 2
   DSize         ← 32
   else
   NativeShift   ← 3
   DSize         ← 64

   # Indices computed from faulting address
   if PWCtl_PWDirExt=1 then
      Bindex    ←(vAddr >> PWField_BDI) and((1<<PWSize_BDW)-1)
      Gindex    ←(vAddr >> PWField_GDI) and((1<<PWSize_GDW)-1)
   else
      tempPointer ←{(vAddr>>PWField_GDI and ((1<<PWSize_GDW)-1)}

      switch ({PWCtl_XK,PWCtl_XS,PWCtl_XU})
         case 001  # xuseg only
            if (vAddr[63] or vAddr[62])=1 then
```

```
            return (0)
            endif
            Gindex ← tempPointer
        case 011  # xuseg & xsseg
            if (vAddr[63] and vAddr[62])=1 then
            return (0)
            endif
            Gindex ← {(vAddr>>62) & 1, tempPointer}
        case 101  # xuseg & xkseg
            if (~vAddr[63] and vAddr[62])=1 then
            return (0)
            endif
            Gindex ← {(vAddr>>63) & 1, tempPointer}
        case 111  # xuseg, xsseg, xkseg
        Gindex ←{(vAddr>>62) and 3, tempPointer}
        default
            return (0)
    end switch
    Uindex ← vAddr >> PWField_UDI and((1<<PWSize_UDW)-1)
    Mindex ← vAddr >> PWField_MDI) and ((1<<PWSize_MDW)-1)
    PTindex ← vAddr >> PWField_PTI) and((1<<PWSize_PTW)-1)

# Offsets into tables
Goffset   ← Gindex << NativeShift
Uoffset   ← Uindex << NativeShift
Moffset   ← Mindex << NativeShift
PToffset0 ←(PTindex >> 1) << (NativeShift + PWSize_PTEW+1)
PToffset1 ← PToffset0 OR (1 << (NativeShift + PWSize_PTEW))

EntryLo0 ← UNPREDICTABLE
EntryLo1 ← UNPREDICTABLE
Context_BadVPN2 ← UNPREDICTABLE
XContext_BadVPN2 ← UNPREDICTABLE

# Starting address - Page Table Base
vAddr ← PWBase

# Global Directory
if (PWSize_GDW > 0) then
    vAddr        ← vAddr or Goffset
    (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
    t            ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

    if (t and (1<<PWCtl_Psn) && PWCtl_Hugpg=1) then # PTEvld is set
        HugePage ← true
        HgPgGDHit ← true
        t  ← t >> PWField_PTEI - 2 // shift entire PTE
        t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
        w ← (PWField_GDI)-1
        if ( ( PWField_GDI and 0x1)=1) // check if index is odd e.g. 2x power of 4
        // generate adjacent page from same PTE for odd TLB page
            lsb ← (1<<w)>> 6
            pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
            pw_EntryLo1 ← t or lsb # lsb=1 odd page
        elseif (PWCtl_DPH = 1)
        // Dual Pages - figure out whether even or odd page loaded first
            OddPageBit = (1 << PWField_GDI)
            if (vAddr and OddPageBit)
```

```
            pw_EntryLo1 ← t
         else
            pw_EntryLo0 ← t
         endif
      // load second PTE from directory for other TLB page
         vAddr2 ←vAddr xor OddPageBit
         (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
         t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
         t  ← t >> PWField_PTEI - 2 // shift entire PTE
         t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
         if (vAddr and OddPageBit)
            pw_EntryLo0 ← t
         else
            pw_EntryLo1 ← t
         endif
      else
         goto ERROR
      endif
      goto REFILL
   else
      vAddr ← t
   endif
endif

# Upper directory
if (PWSize_UDW > 0) then
   vAddr        ← vAddr or Uoffset
   (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
   t            ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

   if (t and (1<<PWCtl_Psn) && PWCtl_Hugpg=1) then# PTEvld is set
      HugePage ← true
      HgPgUDHit ← true
      t  ← t >> PWField_PTEI - 2 // right-shift entire PTE
      t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
      w ← (PWFIELD_UDI)-1
      if ( (PWFIELD_UDI and 0x1)= 0x1) //check if odd e.g. 2x power of 4
      // generate adjacent page from same PTE for odd TLB page
         lsb ← (1<<w)>> 6 // align PA[12] into EntryLo* register bit 6
         pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
         pw_EntryLo1 ← t or lsb # lsb=1 odd page
      elseif (PWCtl_DPH = 1)
      // Dual Pages - figure out whether even or odd page loaded first
         OddPageBit = (1 << PWFIELD_UDI)
         if (vAddr and OddPageBit)
            pw_EntryLo1 ← t
         else
            pw_EntryLo0 ← t
         endif
      // load second PTE from directory for odd TLB page
         vAddr2 ←vAddr xor OddPageBit
         (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
         t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
         t  ← t >> PWField_PTEI - 2 // right-shift entire PTE
         t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
         if (vAddr and OddPageBit)
            pw_EntryLo0 ← t
         else
```

```
            pw_EntryLo1 ← t
         endif
      else
         goto ERROR
      endif
      goto REFILL
   else
      vAddr ← t
   endif
endif

# Middle directory
if (PWSize_MDW > 0) then
   vAddr         ← vAddr OR Moffset
   (pAddr, CCA)  ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
   t             ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)
   if (t and (1<<PWCtl_Psn) && PWCtl_Hugpg=1) then# PTEvld is set
      HugePage ← true
      HgPgMDHit ← true
      t  ← t >> PWField_PTEI - 2 // right-shift entire PTE
      t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
      pw_EntryLo0 ← t # note FILL fields are 0
      w ← (PWField_MDI)-1
      if ( (PWField_MDI and 0x1)= 0x1) // check if odd e.g. 2x power of 4
      // generate adjacent page from same PTE for odd TLB page
      lsb ← (1<<w)>> 6 // align PA[12] into EntryLo* register bit 6
      pw_EntryLo0 ← t and not lsb # lsb=0 even page; note FILL fields are 0
      pw_EntryLo1 ← t or lsb # lsb=1 odd page
      elseif (PWCtl_DPH = 1)
      // Dual Pages - figure out whether even or odd page loaded first
         OddPageBit = (1 << PWField_MDI)
         if (vAddr and OddPageBit)
            pw_EntryLo1 ← t
         else
            pw_EntryLo0 ← t
         endif
      // load second PTE from directory for odd TLB page
         vAddr2 ←vAddr xor (1 << (NativeShift + PWSize_PTEW)
         (pAddr2, CCA2) ←AddressTranslation(vAddr2, DATA, LOAD, KERNEL)
         t  ← LoadMemory(CCA2, DSize, pAddr2, vAddr2, DATA)
         t  ← t >> PWField_PTEI - 2 // right-shift entire PTE
         t  ← ROTRIGHT(t, 2) // 64-bit rotate to place RI/XI bits
         if (vAddr and OddPageBit)
            pw_EntryLo0 ← t
         else
            pw_EntryLo1 ← t
         endif
      else
         goto ERROR
      endif
      goto REFILL
   else
      vAddr ← t
   endif
endif

# Leaf Level Page Table - First half of PTE pair
vAddr         ← vAddr or PToffset0
```

```
    (pAddr, CCA)  ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
    temp0         ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

    # Leaf Level Page Table - Second half of PTE pair
    vAddr         ← vAddr or PToffset1
    (pAddr, CCA)  ← AddressTranslation(vAddr, DATA, LOAD, KERNEL)
    temp1         ← LoadMemory(CCA, DSize, pAddr, vAddr, DATA)

    # Load Page Table Entry pair into TLB
    temp0         ← temp0 >> PWField_PTEI - 2 // right-shift entire PTE
    pw_EntryLo0   ← ROTRIGHT(temp0, 2) // 32-bit rotate to place RI/XI bits

    temp1         ← temp1 >> PWField_PTEI - 2 // right-shift entire PTE
    pw_EntryLo1   ← ROTRIGHT(temp1, 2) // 64-bit rotate to place RI/XI bits

REFILL:
    found ← 1
    m ← (1<<PWField_PTI)-1

    if (HugePage) then
        # Non-power-of-4 page size halved to provide power-of-4 page size.
        # 1st step: Halve page size (1<<(w-1))

        switch ({HgPgBDHit,HgPgGDHit,HgPgUDHit,HgPgMDHit})
            case 1000
                m ← (1<<(PWField_BDI))-1
            case 0100
                m ← (1<<(PWField_GDI))-1
            case 0010
                m ← (1<<(PWField_UDI))-1
            case 0001
                m ← (1<<(PWField_MDI))-1
        end switch
    endif
    # 2nd step: Normalize mask field to 4KB as smallest base (>>12)
    pw_PageMask_Mask ←m>>12


# The hardware page walker inserts a page into the TLB in a manner
# identical to a TLBWR instruction as executed by the software refill handler
    pw_EntryHi = ( vaddr and not 0xfff )| EntryHi_ASID
    TLBWriteRandom(pw_EntryHi, pw_EntryLo0, pw_EntryLo1, pw_PageMask)
    return(found)
    # If an error/exception condition is detected on a page table
    # walk memory access, this function exits with found=0.
    #
    OnError:
        return(0)
endsub
```

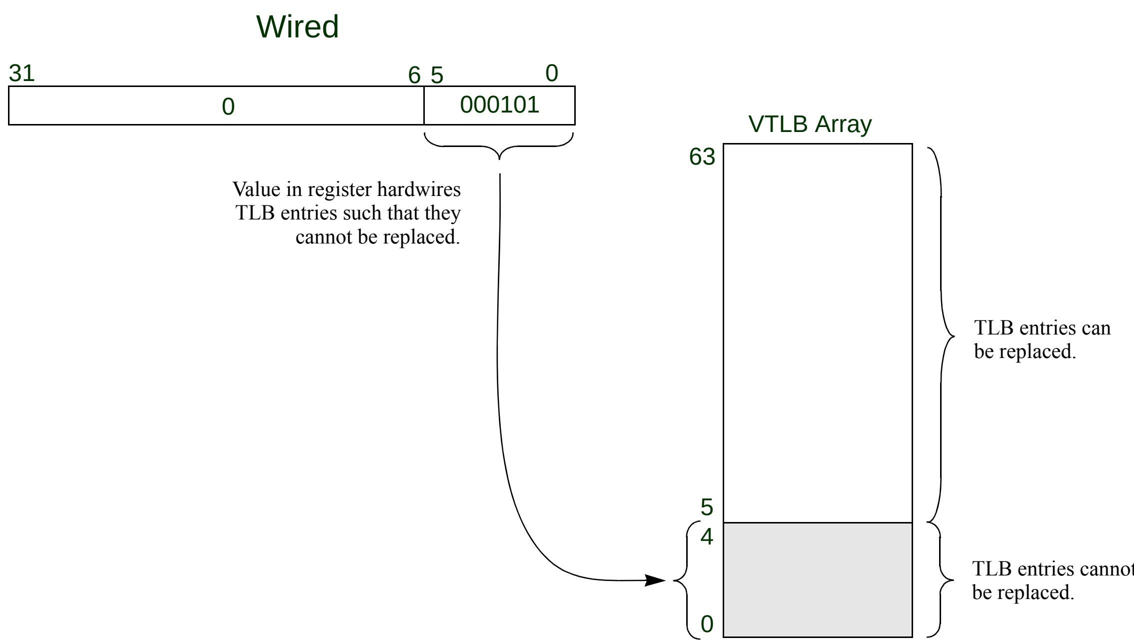
If a page is marked invalid, the hardware refill handler will still fill the page into the TLB. Software can point to invalid PTEs to represent regions that are not mapped. When the Software attempts to use the invalid TLB entry, a TLB invalid exception will be generated.

3.8 Hardwiring VTLB Entries

The P6600 core allows up to 63 entries of the VTLB 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 VTLB. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a **TLBWR** instruction. However, wired entries can be overwritten by a **TLBWI** instruction.

Note that wired entries in the VTLB must be contiguous and start from 0. For example, if the *Wired* field of this register contains a value of 5, this indicates that entries 4, 3, 2, 1, and 0 of the VTLB are wired. The *Wired* register is reset to zero by a Reset exception. Figure 3.21 shows an example of hardwiring the lower 5 entries of the VTLB. A value of 0x0 in the *Wired* register indicates that no entries are hardwired and that all entries are available for replacement.

Figure 3.21 Hardwiring Entries in the VTLB



3.9 FTLB Parity Errors

FTLB parity errors are reported using bits 31:28 of the CP0 *CacheErr* register (CP0, Register 27, Select 0). These read-only bits are set by hardware and are used to report errors within the L1 instruction and data caches, as well as the FTLB. An FTLB parity error can be reported for either the tag portion or the data portion of the array as shown in Table 3.4.

Table 3.4 FLTB Parity Error Reporting in the CacheErr Register

EREC (Bits 31:30)	ED (Bit 29)	ET (Bit 28)	Condition
2'b11	0	0	No FTLB errors
	0	1	FTLB Tag RAM error
	1	0	FTLB Data RAM error
	1	1	N/A[1]

1. It is not possible to set both the ED and ET bits in the P6600 core. Even if there are simultaneous errors in both arrays, the tag error takes precedence and the ET bit is set. In this case the data error is ignored.

Depending on the instruction being executed, hardware may or may not report a parity error for the tag and/or data array of the FTLB. Table 3.5 lists each TLB instruction and whether parity errors are logged for the data and tag arrays.

Table 3.5 FLTB Parity Error Reporting per Instruction

Instruction	Parity Error Checked?	
	FTLB Data Array	FTLB Tag Array
TLBINV	No	Yes
TLBINVF	No	No
TLBR	Yes	Yes
TLBWI	No $EntryHi_{EHINV} = 1$	No $EntryHi_{EHINV} = 1$
	No $EntryHi_{EHINV} = 0$	Yes $EntryHi_{EHINV} = 0$
TLBWR	No	Yes
TLBP	Yes	Yes
Lookup (ITLB or DTLB miss)	Yes	Yes

3.10 FTLB Hashing Scheme and the TLBWI Instruction

When a TLBWI instruction is executed, the following hashing scheme is used to calculate the FTLB index from the VPN2 field of the *EntryHi* register and the Index field of the *Index* register. This scheme is used only when the $EntryHi_{EHINV}$ bit is 0. When $EntryHi_{EHINV} = 1$, hashing is ignored and the indexing of the FLTB is performed entirely in hardware.

When the $EntryHi_{EHINV}$ bit is 0, the VPN2 field in the *EntryHi* register must be consistent with the index value stored in the 10-bit *Index* field of the CP0 *Index* register. This field is used to index the total number of entries in the TLB, which equates to 64 entries in the VTLB and 512 entries in the FTLB for a total of 576 entries. To determine the size of the FTLB, hardware subtracts the VTLB size, which is always 64 entries, from the total number of entries (576) to derive an FTLB size of 512 entries. This number of entries is indexed by the lower 9 bits of the 10-bit Index field.

When the core is configured with an FTLB, the lower 9 bits of the *Index* field are organized as follows:

- Bits 6:0 = FTLB set
- Bits 8:7 = FTLB way

The FTLB set reflected in bits 6:0 of the Index field of the *Index* register ($Index_{Index}$) must be the same as the set number calculated from the VPN2 field of the *EntryHi* register ($EntryHi_{VPN2}$).

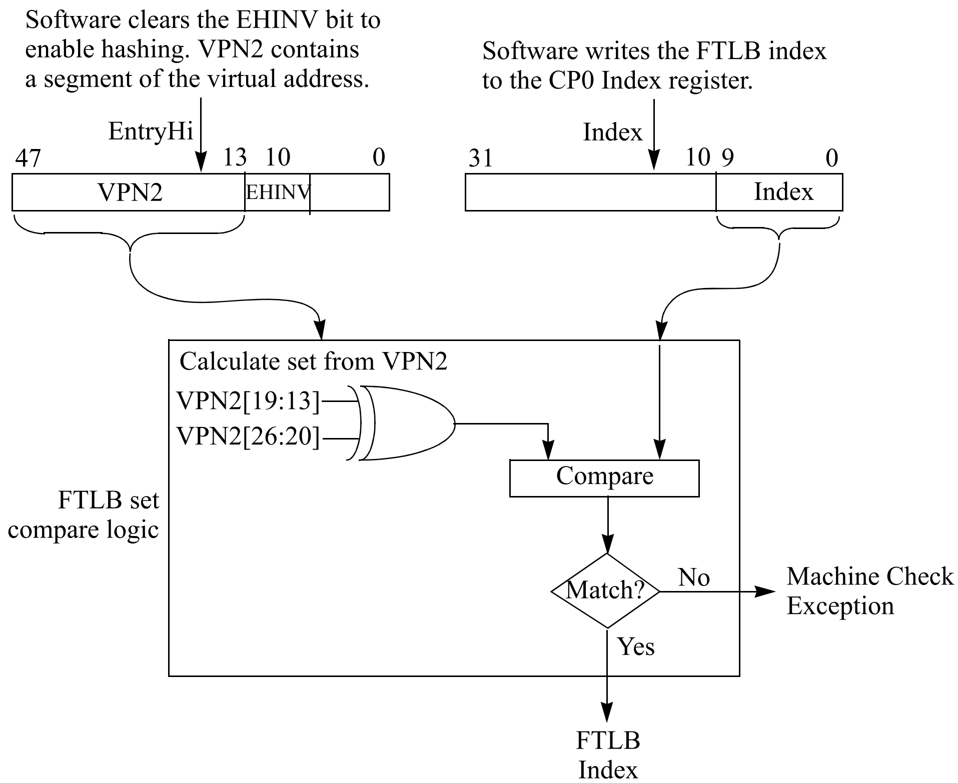
For a 4 KByte page size, the set number is calculated by performing an Exclusive OR (XOR) function of bits [26:20] and bits [19:13] of the $EntryHi_{VPN2}$ field.

For a 16 KByte page size, the set number is calculated by performing an Exclusive OR (XOR) function of bits [28:22] and bits [21:15] of the $EntryHi_{VPN2}$ field.

If the set number calculated from the $EntryHi_{VPN2}$ field as described above matches that stored in bits 6:0 of the *Index* register, the TLBWI instruction is allowed to continue and the FTLB is indexed. If the values do not match, a machine check exception is generated. Refer to Section 5.7.5 of the Exceptions chapter for more information on the machine check exception. Note that the TLBWR instruction does not use this hashing scheme because the indexing is performed exclusively in hardware.

The FTLB hashing scheme for a 4 KByte page size is shown in Figure 3.22. The 16 KByte page size would be identical, except for the range of VPN2 bits that are XOR'ed by hardware as described above. Note that only bits 6:0 of the Index field are compared with the calculated value. Bits 8:7 represent the FTLB way and bypass the compare operation.

Figure 3.22 FTLB Hashing Scheme During a TLB Index Write — 4 KByte Page Size



3.11 TLB Exception Handling

The P6600 core allows for the following types of TLB exceptions.

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

The *Address Error* exceptions (AdEL and AdES) are used in both user mode and supervisor mode.

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

The *TLB Refill* exception is taken on any TLB miss regardless of the operating mode.

The *XTLB Refill* exception is taken on any XTLB miss regardless of the operating mode.

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

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

The *TLB Read Inhibit* exception (TLBRI) is taken when there is a TLB hit during a read operation, the RI bit of the entry is set, and the $PageGrain_{EIC}$ bit is set.

The *TLB Execute Inhibit* exception (TLBXI) is taken when there is a TLB hit during an instruction fetch, the XI bit of the entry is set, and the $PageGrain_{EIC}$ bit is set.

A *TLB Modified* exception is taken whenever there is a TLB hit and the Dirty bit associated with that entry is not set. Note that only occurs on a store instruction and not on a load/fetch instruction.

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

Note that for the **`CacheOp`** and **`SyncI`** instructions, the TLBRI and TLBXI exceptions are not supported.

3.11.1 Overview of TLB Exception Handling Registers

The P6600 core uses three CP0 registers to manage TLB exceptions. The exception flow in terms of these registers is described in Section 3.11.2, "TLB Exception Flow Examples".

- *Context* (CP0 register 4, Select 0): Contains the pointer to an entry in the page table entry (PTE) array.
- *ContextConfig* (CP0 register 4, Select 1): Defines the range of bits used by the *Context* register into which the high order bits of the virtual address causing the TLB exception will be written depending on the page size.
- *BadVAddr* (CP0 register 8, Select 0): Stores the virtual address that caused the exception.

3.11.1.1 Context Register

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. When a TLB exception is taken, hardware performs the bit shifting and manipulation of the value stored in the *BadVAddr* register and places the result into the *BadVPN2* field of the *Context* register. This eliminates software from having to perform this function manually.

A TLB exception causes the virtual address to be written to a variable range of bits, defined as (X-1):Y of the *Context* register. This range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 63:X, Y-1:0 are read/write to software and are unaffected by the exception. Software sets the $ContextConfig_{PTEBase}$ field to point to the base address of a page table in memory. The $ContextConfig_{BadVPN2}$ is derived from the virtual address associated with the exception.

Figure 3.23 shows the format of the *Context* register. Refer to Section 3.11.2, "TLB Exception Flow Examples" for more information on the usage of this register.

Figure 3.23 Context Register Format

63 X	X-1 Y	Y-1 0
PTEBase	BadVPN2	PTEBaseLow

3.11.1.2 ContextConfig Register

The *ContextConfig* register 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.

Software writes a set of contiguous ones to the $ContextConfig_{VirtualIndex}$ field. Hardware then determines which bits of this register are high and low. The highest order bit that is a logic '1' serves as the MSB of the *BadVPN2* field of the *Context* register. The lowest order bit that is a logic '1' serves as the LSB of the *BadVPN2* field of the *Context* register. A value of all zero's in the *VirtualIndex* field means that the full 32 bits of the *Context* register are R/W for software and are unaffected by TLB exceptions.

A value of all ones in the $ContextConfig_{VirtualIndex}$ field means that the full 21 bits of the faulting virtual address will be copied into the context register, making it duplicate the *BadVAddr* register. A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

Figure 3.24 shows the formats of the *ContextConfig* Register. Refer to Section 3.11.2, "TLB Exception Flow Examples" for more information on use of the this register.

Figure 3.24 ContextConfig Register Format

31 — 23	22 — 2	1 — 0
0	VirtualIndex	0

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. It is possible for software to determine which bits are implemented by alternately writing all zeroes and all ones to the register, and reading back the resulting values. Table 3.6 describes some useful *ContextConfig* values. In this table, note that for a page table entry size of 32 bits per page, a total of 64 bits are copied from memory to support the dual-entry structure of the VTLB/FTLB. In this case, the lower 32 bits would be copied to entry 0 of the dual entry structure, and the upper 32 bits would be copied to entry 1 of the structure. The same is true for a page table with 64 bits per page. In this case, 128 bits would be fetched from memory.

Table 3.6 Example ContextConfig Values — Single Level Page Table Organization

Value	Page Table Organization	Page Size	Page Table Entry Size	Memory Structure
0x007F_FFF0	Single Level	4K	64 bits/page	128-bit
0x003F_FFF8	Single Level	4K	32 bits/page	64-bit

3.11.1.3 BadVAddr Register

The BadVAddr is a 64-bit read-only register which holds the virtual address which caused the last address-related exception. It is set for the exception types shown at the beginning of Section 3.11, "TLB Exception Handling".

Note that the *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.

Figure 3.25 BadVAddr Register Format

63 — 0
BadVAddr

3.11.2 TLB Exception Flow Examples

The following two examples show the flow of a TLB exception for the single level and dual level page table configurations.

3.11.2.1 Single Level Table Configuration

When a VTLB/FTLB error occurs, hardware writes the most recent virtual address that caused the error into bits 63:0 of the read-only *BadVAddr* register. The number of bits used by hardware to index the page table depends on the page size. For example, with a 4 KByte page size, hardware uses bits 63:13 of the *BadVAddr* register, along with the *PTEBase* field of the *Context* register, to determine the address that caused the exception.

Hardware assembles this information and places the result into the *Context* register. Use of the *Context* and *ContextConfig* registers eliminates software from having to derive the page table index manually. Depending on the page table architecture, software programs the *ContextConfig* register to indicate how many bits of the *BadVAddr* register are used by hardware to program the *Context* register. This determines the size of both the $Context_{BadVPN2}$ and $Context_{PTEBase}$ fields.

The example shown in Figure 3.26 is for a single level table configuration with a 4 KByte page size and 32 bits per page.

When an exception is taken, hardware writes the address that caused the exception into the *BadVAddr* register. Because the page table is single level and the page size is already known to be 4 KBytes, software programs a value of 0x3F_FFF8 into the $ContextConfig_{VirtualIndex}$ field. This value indicates the following information:

- The lower three bits of this value are 0, indicating that a 64-bit memory structure is being accessed. For this 64-bit value, the lower 32 bits are written to the entry 0 of the dual-entry TLB, and the upper 32 bits are written to entry 1 of the same TLB entry. Since the lower 3 bits of this field are zero, bit 3 (the first bit that is set) is used to define the low-order bit of the BadVPN2 field in the *Context* register.
- The highest-order bit that is 1 in this field is bit 21. This indicates that bit 21 is the last bit of the *BadVPN2* field in the *Context* register. As a result, the *PTEBase* field of the *Context* register occupies bits 63:22.

Based on this information, hardware assembles the value in the *Context* register as follows:

- $Context_{PTEBase}$ = bits 63:22. Indicates the base address of the page table in memory. This value is a pointer to the start of the page table in memory.
- $Context_{BadVPN2}$ = bits 21:3. Hardware copies bits 31:13 of the *BadVAddr* register into this field. This 19-bit value is a pointer for up to 1M entries in each page table selected by the $Context_{PTEBase}$ field. Bits 12:0 of the *BadVAddr* register are not used in this case since the page size is 4 KBytes.
- $Context_{PTEBaseLow}$ = bits 2:0. Indicates access to a 64-bit memory location.

Figure 3.26 32-bit TLB Exception Flow Example — Single Level Table, 4 KB Page Size

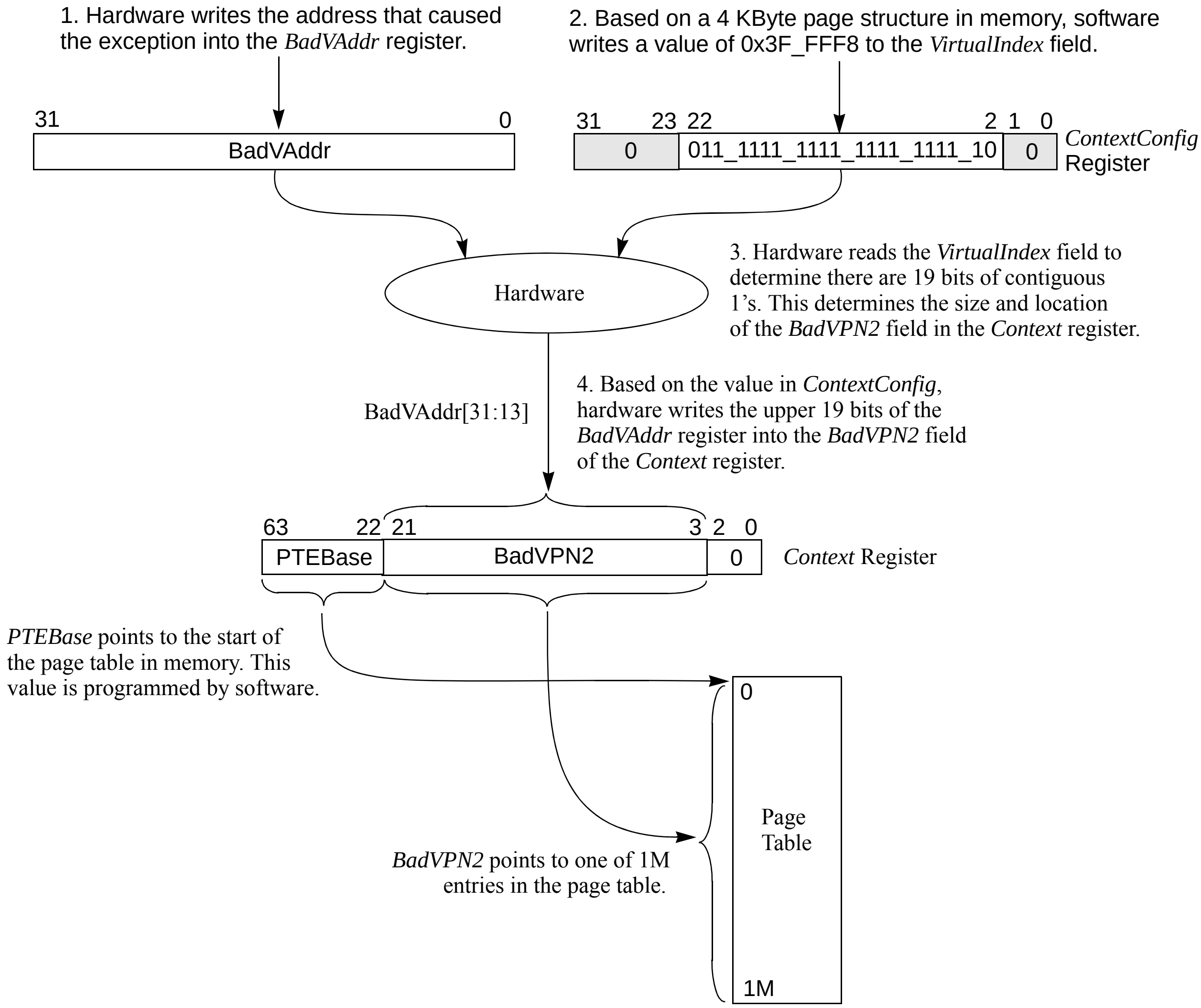
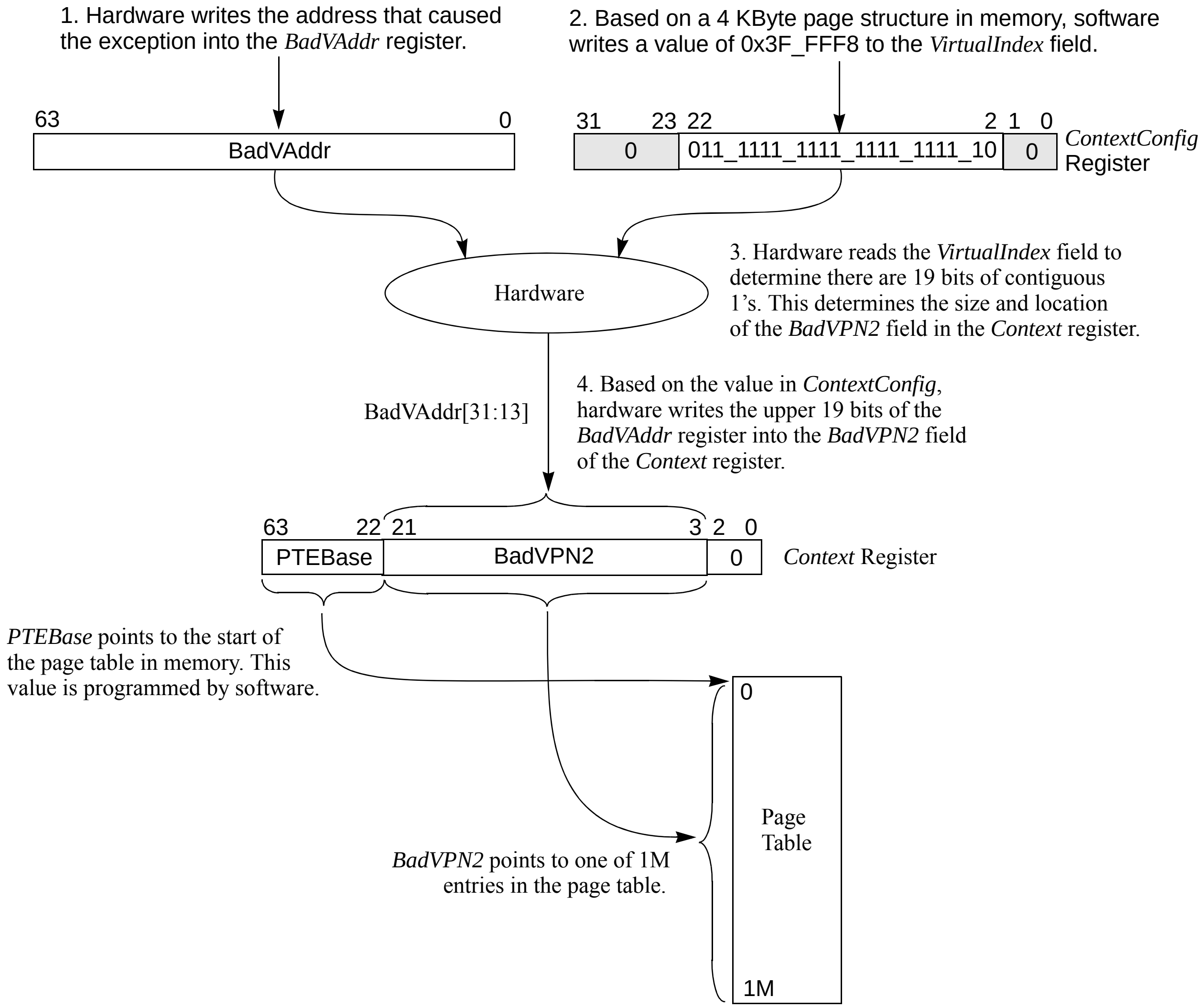


Figure 3.27 64-bit TLB Exception Flow Example — Single Level Table, 4 KB Page Size



3.11.2.2 Dual Level Table Configuration

The TLB exception flow for a dual level page table structure is similar to that of a single level table described in Section 3.11.2.1, "Single Level Table Configuration". The upper bits of *PTEBase* are used to select the location of the first level table in memory. The *BadVPN2* field of the *Context* register is used to index the first level table and acts as a pointer to each of the second level tables in the page table array.

When a VTLB/FTLB error occurs, the most recent virtual address that caused the error is stored in bits 63:0 of the read-only *BadVAddr* register. The number of bits in the *BadVAddr* register used by hardware to index the page table depends on the page size and table organization.

Hardware assembles this information and places the result into the *Context* register. Use of the *Context* and *ContextConfig* registers eliminates software from having to derive the page table index manually. Depending on the page table architecture, software programs the *ContextConfig* register to indicate how many bits of the *BadVAddr* register are used by hardware to program the *Context* register. This determines the size of both the $Context_{BadVPN2}$ and $Context_{PTEBase}$ fields.

The example shown in Figure 3.28 is for a dual level table configuration with a 4 KByte page size and 32 bits per page.

When an exception is taken, hardware writes the address that caused the exception into the 64-bit *BadVAddr* register. Because each table in this example contains 1K entries, software programs a value of 0x00_0FFC into the $ContextConfig_{VirtualIndex}$ field. This value indicates the following information:

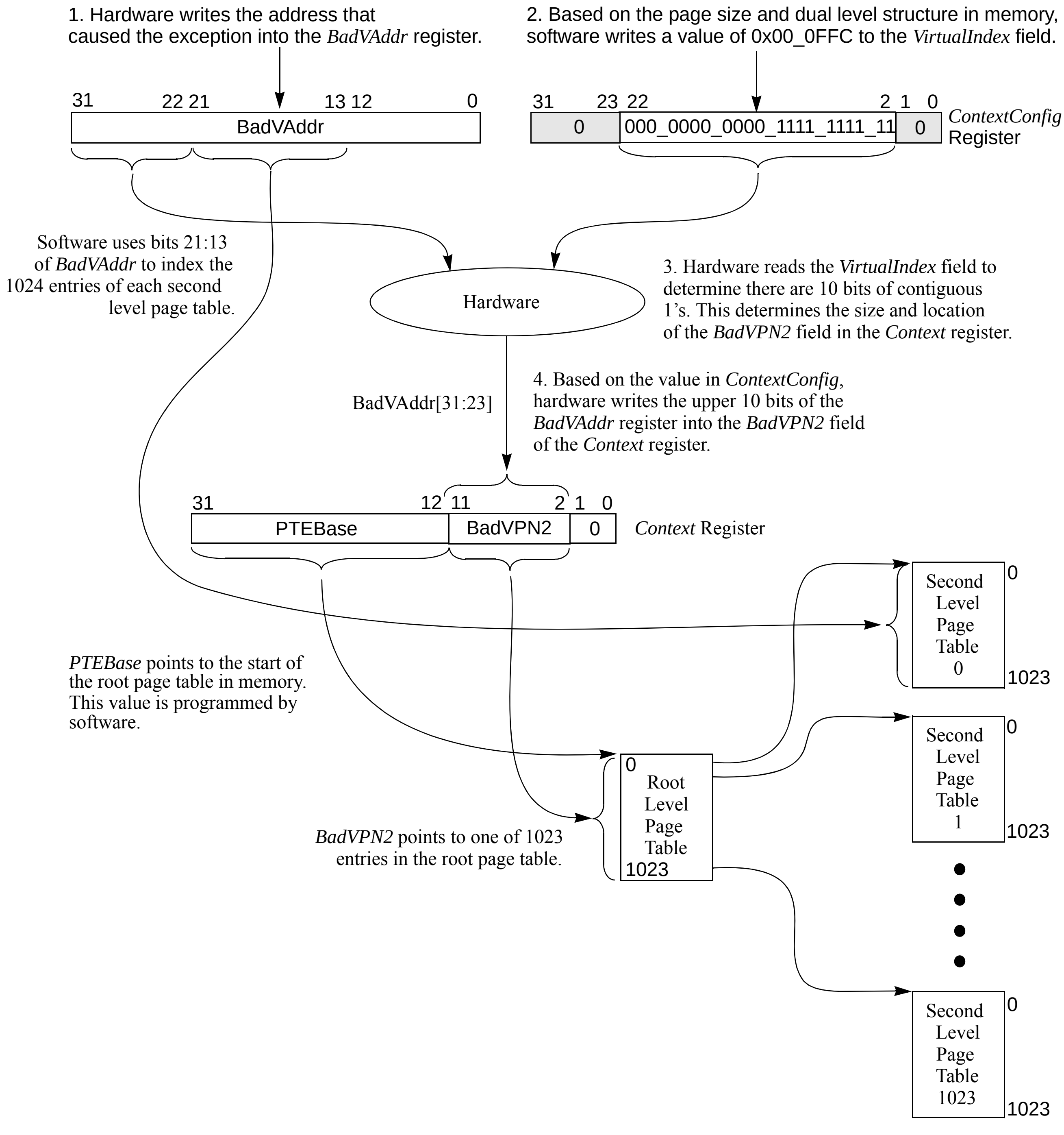
- The lower two bits of this value are 0, indicating that a 32-bit memory structure is being accessed. This also indicates that bit 2 will be the low-order bit for the $Context_{BadVPN2}$ field.
- The highest-order bit that is '1' in the $ContextConfig_{VirtualIndex}$ field is bit 11. This indicates that bit 11 will be the highest-order bit of the $Context_{BadVPN2}$ field. As a result, the $Context_{PTEBase}$ field occupies bits 63:12. This field is used to access the location of the root level page table in memory.

Based on this information, hardware assembles the context register as follows:

- $Context_{PTEBase}$ = bits 63:12. Indicates the base address of the page table in memory. This value is a pointer to the root page table in memory.
- $Context_{BadVPN2}$ = bits 11:2. Based on the state of the $ContextConfig_{VirtualIndex}$ field in this example, hardware copies bits 31:22 of the *BadVAddr* register into this field. This 10-bit value is a pointer to the 1024 entries in the root page table selected by the $Context_{PTEBase}$ field. Bits 12:0 of the *BadVAddr* register are not used in this case since the page size is 4 KBytes.
- $Context_{PTEBaseLow}$ = bits 1:0. Indicates access to a 32-bit memory location.

As stated above, bits 31:22 of the *BadVAddr* register are copied into the *BadVPN2* field of the *Context* register and are used to select one of 1024 entries in the root page table. Each of these entries acts as a pointer to one of the 1024 second level tables. Software uses bits 21:13 of the *BadVAddr* register to index one of 1024 entries in each second level page table.This concept is shown in Figure 3.28.

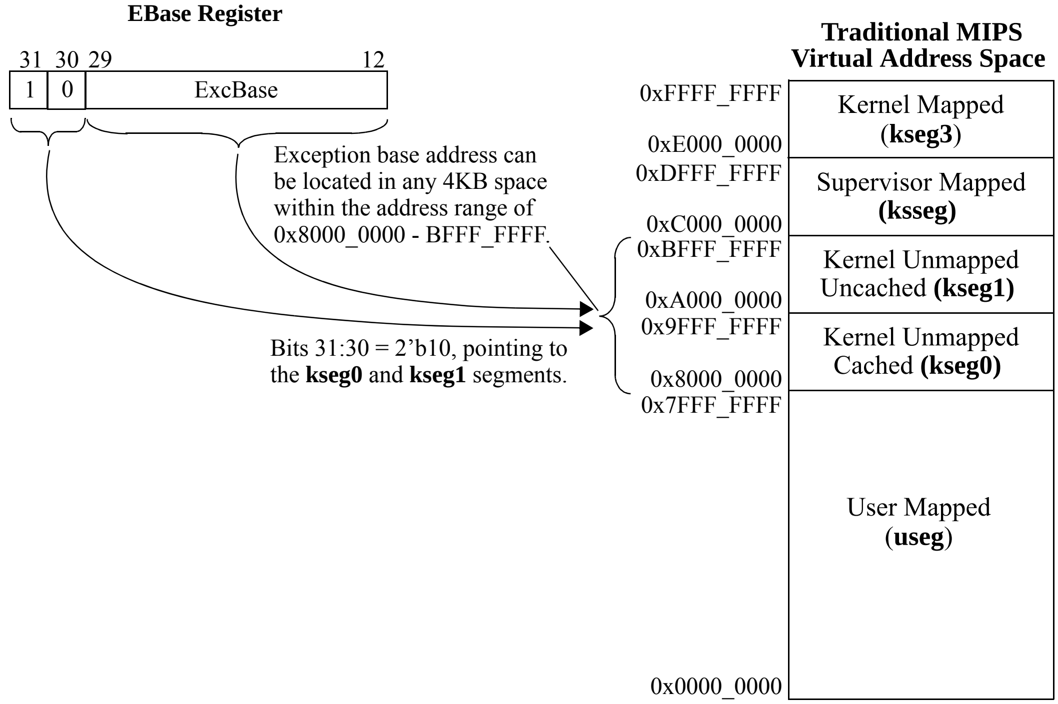
Figure 3.28 32-bit TLB Exception Flow Example — Dual Level Table, 4 KB Page Size



3.12 Exception Base Address Relocation

The P6600 core allows the base address of an exception vector to be relocated. The base address of the exception is stored in the CP0 *EBase* register. In previous generation MIPS32 processors, bits 31:30 of the *EBase* Register were not writeable and had a fixed value of 2'b10 so that the exception handler would be executed from the *kseg0* or *kseg1* segments. This concept is shown in Figure 3.29.

Figure 3.29 Location of 32-bit Exception Vector Base Address in Traditional MIPS Virtual Address Space



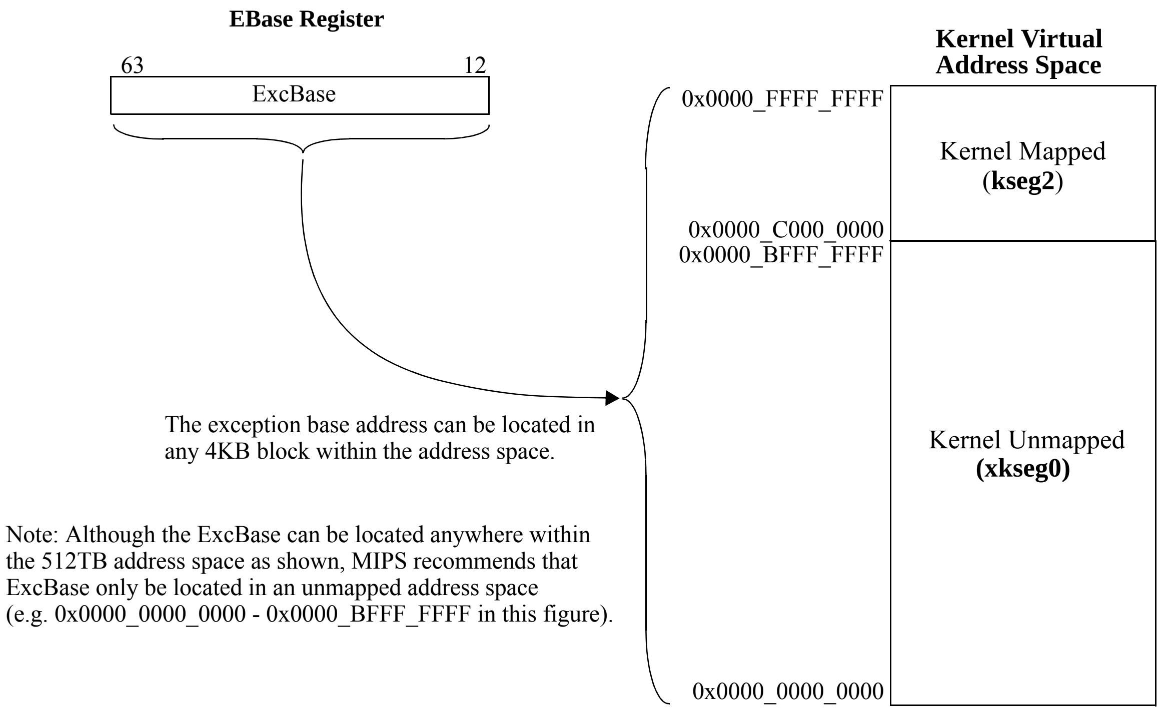
In the P6600 core, the size of the exception base address is determined by the state of the *WG* bit in the CP0 *EBase* register (CP0 register 15, Select 1). At reset, the *WG* bit is cleared by default and bits 31:30 of the *EBase* Register are forced to a value of 2'b10 by hardware as described above. This is shown in Figure 3.29 above.

When the *WG* bit is set, bits 63:30 of the *ExcBase* field become writeable and are used to relocate the exception base address to other areas of memory. This is shown in Figure 3.30.

Note that if the *WG* bit is set by software (allowing bits 31:30 to become part of the *ExcBase* field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after *WG* was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the *EBase* register prior to clearing the *WG* bit if it wants to ensure that future exceptions will be executed from the *kseg0* or *kseg1* segments.

Note that the *WG* bit is different from the *CV* bit in the *Config5* register. Although their functions are similar, the *CV* bit applies only to cache error exceptions, whereas the *WG* bit applies to all exceptions.

Figure 3.30 Location of Exception Vector Base Address in the P6600



3.13 Address Error Detection

This section describes the conditions on which an address error may be taken.

3.13.1 Instruction Address Errors in 64-bit Mode

An address error is taken on an instruction address in 64-bit Mode when any of the following conditions are met.

- Address is reserved/ unavailable
- Address is in Kernel or XKPhys spaces when operating in Supervisor Mode
- Address is in Kernel, XKPhys or Supervisor spaces when operating in User Mode
- Address is not word-unaligned
- Address is in 64-bit Kernel space when Status.KX = 0
- Address is in 64-bit Supervisor space when Status.SX = 0
- Address is in 64-bit User space when Status.UX = 0
- Address is in XKPhys space and bits [47:32] are non-zero when operating in guest mode and *Root.PageGrain.ELPA* = 0.

3.13.2 Instruction Address Errors in 32-bit Mode

An address error is taken on an instruction address in 32-bit mode when any of the following conditions are met.

- Address is in Kernel space when operating in Supervisor Mode
- Address is in Kernel or Supervisor spaces when operating in User Mode
- Address is not word-unaligned
- Address is illegal 32-bit address value

3.13.3 Data Address Errors in 64-bit Mode

A data address error is taken on a data address in 64-bit mode when any of the following conditions are met.

- Address is reserved/ unavailable
- Address is in Kernel or XKPhys spaces when operating in Supervisor Mode
- Address is in Kernel, XKPhys or supervisor spaces when operating in User Mode
- Address crosses 16-KB page boundary with specified data size
- Address is unaligned when instruction is LL, LLD, SC or SCD
- Address is unaligned when cacheability is uncached
- Address is in 64-bit Kernel space when Status.KX = 0
- Address is in 64-bit Supervisor space when Status.SX = 0
- Address is in 64-bit User space when Status.UX = 0
- Address is in XKPhys space and bits [47:32] are non-zero when operating in guest mode and *Root.PageGrain.ELPA* = 0.

3.13.4 Data Address Errors in 32-bit Mode

A data address error is taken on a data address in 32-bit mode when any of the following conditions are met.

- Address is in Kernel space when operating in Supervisor Mode
- Address is in Kernel or Supervisor spaces when operating in User Mode
- Address crosses 16-KB page boundary with specified data size
- Address is unaligned when instruction is LL, LLD, SC or SCD
- Address is unaligned when cacheability is uncached
- Address is illegal 32-bit address value (A legal 32-bit address value is one with natural sign-extension, i.e. VA63:32 = 32{VA31})

3.14 VTLB and FTLB Initialization

This section describes the procedure for VTLB/FTLB initialization.

3.14.1 TLB Initialization Sequence

The following steps are used to initialize the TLB's.

1. Read the 3-bit $Config_{MT}$ field to determine if an FTLB is enabled. If this field is 3'b001, the FTLB is disabled and address translation is performed only in the VTLB. If this field is 3'b100, both the VTLB and the FTLB are

enabled. Refer to the *Config* register in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

2. Read the 6-bit $Config1_{MMUSIZE}$ field to determine the VTLB size. This field has a default of 0x3F, indicating a VTLB size of 64 entries. Refer to the *Config1* register in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

3. Read the *Config4* register to determine the FTLB organization. Bits 12:0 of the *Config4* register store information relating to FTLB organization. Bits 3:0 indicate the number of FTLB ways, bits 7:4 indicate the number of FTLB sets, and bits 12:8 indicate the FTLB page size. Refer to the *Config4* register in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

4. Set the $EntryHi_{EHINV}$ bit to indicate that **TLBWI** invalidate is enabled. When this bit is set, the **TLBWI** instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. This bit is ignored on a **TLBWR** instruction. Refer to the *EntryHi* register in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

5. Write all zero's to the *EntryLo0* and *EntryLo1* registers to initialize them. Refer to the *EntryLo0 and EntryLo1* registers in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

6. Write the appropriate TLB size to the $Index_{INDEX}$ field. The value written depends on whether or not an FTLB is present. If the FTLB is not present, a value of 0x3F is programmed into the lower 6 bits of this register. If the FTLB is present, a value of 0x1FF is programmed into the lower 10 bits of this register and indicates a total of 576 entries (64 VTLB + 512 FTLB). Refer to the *Index* register in the chapter entitled *CP0 Registers of the P6600 Core* for more information.

3.14.2 TLB Initialization Code

The following code snippet can be used to initialize the VTLB and FTLB.

```
************************************************************

/* ... at this point, t0 = index of highest tlb entry in jtlb or ftlb if present */

/*    initialize EntryHi.EHINV=1 */

        li      t1, M_EntryHiEHINV
        mtc0    t1, C0_EntryHi          # set EntryHi.EHINV=1

/* initialize EntryLo0/1 to avoid x's in simulation */

        mtc0    zero, C0_EntryLo0
        mtc0    zero, C0_EntryLo1

/*    invalidate each entry */

10:     mtc0    t0, C0_Index            # Store new index in register
        tlbwi                           # Initialize the TLB entry
        bne     t0, zero, 10b           # Loop if more to do
        addi    t0, t0, -1              # Subtract one from index field

/* clear out EHINV bit again */
```

```
mtc0   zero,C0_EntryHi
```

****

3.15 TLB Duplicate Entries

The VTLB 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. Since the VTLB 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 just invalidated. The new entry gets written. When writing to the TLB, all ways of a single set in the FTLB and all the entries of the VTLB are searched for duplicates. If a large page is written to the VTLB and multiple duplicates exist for that larger page in the FTLB (multiple sets in the FTLB), then not all the duplicates are detected (and invalidated).

3.16 Modes of Operation

The P6600 core can operate in either 32-bit mode, or 64-bit mode. In both of these modes, the core can be accessing Kernel, Supervisor, User, and Debug address spaces. There are three bits in the CP0 Status register that are used to enable access to each of these address spaces as described in the following subsection.

3.16.1 Memory Address Space Access

The KX, SX, and UX bits are used to permit access to the associated kernel, supervisor, user, and memory address spaces.

- KX denotes access to kernel space
- SX denotes access to supervisor space
- UX denotes access to user space

Access to these memory spaces is enabled using bits 7:5 of the CP0 Status register (12, 0). The KX bit has priority over the SX and UX bits, and the SX bit has priority over the UX bit as follows: when KX = 0, SX and UX are forced to 0; when SX = 0, UX is forced to 0.

3.16.1.1 KX Bit

The KX bit (7) in the Status register is used to define Kernel and Debug Modes and permit access to Extended Kernel Segment (XKSeg), 0xC000_0000_0000_0000-0xC000_FFFF_7FFF_FFFF and XKPhys Segments. There are four types of Kernel/Debug modes defined as follows:

- Kernel 32-bit Mode is defined as DM=0 AND (EXL=0 OR ERL=0 OR KSU='b00) AND KX=0.
- Kernel 64-bit Mode is defined as DM=0 AND (EXL=0 OR ERL=0 OR KSU='b00) AND KX=1.
- Debug 32-bit Mode is defined as DM=1 AND KX=0.
- Debug 64-bit Mode is defined as DM=1 AND KX=1.

When KX = 1, access to XKSeg and XKPhys is allowed; when KX = 0, any access to XKSeg and XKPhys causes an Address Error exception.

3.16.1.2 SX Bit

The SX bit (6) in the Status register is used to define Supervisor Modes and permit access to Extended Supervisor Segment (XSSeg), 0x4000_0000_0000_0000-0x4000_FFFF_FFFF_FFFF. There are two types of Supervisor modes defined as follows:

- Supervisor 32-bit Mode is defined as DM=0 AND EXL=0 AND ERL=0 AND KSU='b01 AND SX=0.
- Supervisor 64-bit Mode is defined as DM=0 AND EXL=0 AND ERL=0 AND KSU='b01 AND SX=1.

When SX = 1, access to XSSeg is allowed; when SX = 0, any access to XSSeg causes an Address Error exception.

3.16.1.3 UX Bit

The UX bit (5) in the Status register is used to define User Modes and permit access to Extended User Segment (XUSeg), 0x0000_0000_8000_0000-0x0000_FFFF_FFFF_FFFF. There are two types of User modes defined as follows:

- User 32-bit Mode is defined as DM=0 AND EXL=0 AND ERL=0 AND KSU='b10 AND UX=0.
- User 64-bit Mode is defined as DM=0 AND EXL=0 AND ERL=0 AND KSU='b10 AND UX=1.

When UX = 1, access to XUSeg is allowed; when UX = 0, any access to XUSeg causes an Address Error exception.

3.16.2 32-Bit Mode

The MMU's virtual-to-physical address translation is determined by the mode in which the processor is operating. The P6600 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.

Table 3.7 Selecting the 32-bit Addressing Mode

Mode	Status						Debug	Description
	EXL	ERL	KSU	KX[1]	SX[2]	UX	DM	
User	0	0	2'b2	x	x	0	0	32-bit User addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
Supervisor	0	0	2'b1	x	0	x	0	32-bit Supervisor addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.

Table 3.7 Selecting the 32-bit Addressing Mode

Mode	Status						Debug	Description
	EXL	ERL	KSU	KX[1]	SX[2]	UX	DM	
Kernel	x	x	2'b0	0	x	x	0	32-bit Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	x	1	x				0	32-bit Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	1	x	x				0	32-bit 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.

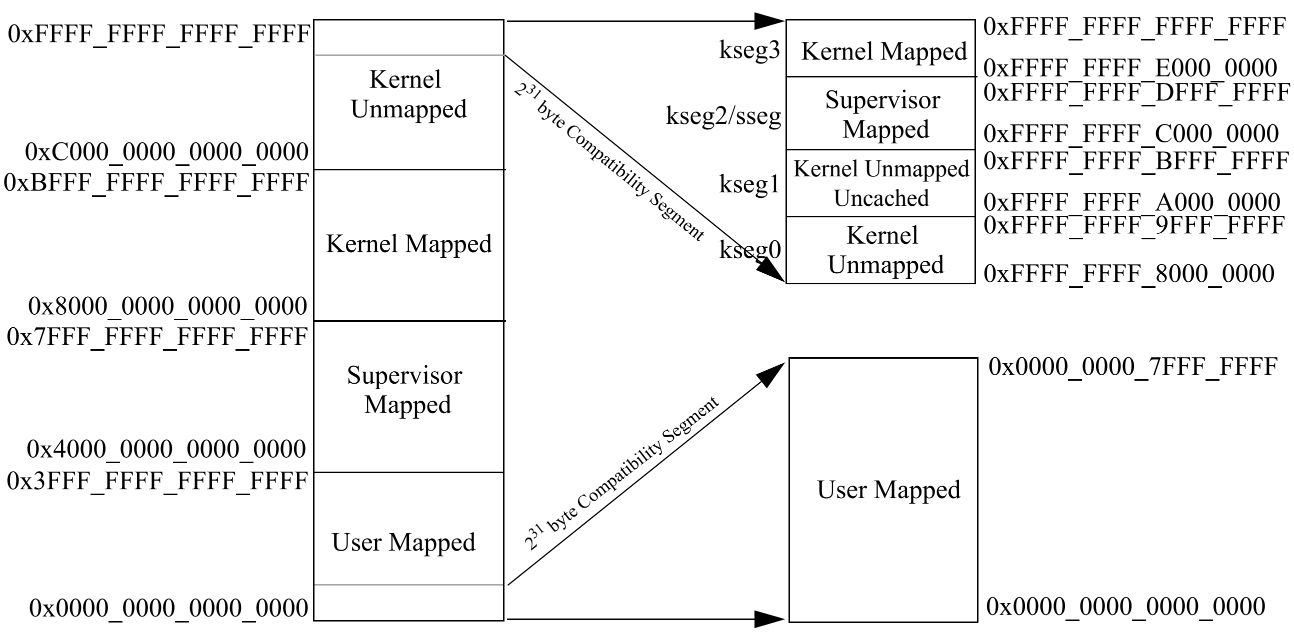
1. When KX = 0, both the SX and UX bits cannot be set.
2. When SX = 0, the UX bit cannot be set.

3.16.2.1 Mapping 64-bit Address Space for 32-bit Addressing

With support for 64-bit operations and address calculation, the P6600 provides support for a 64-bit virtual address space that is sub-divided into four Segments selected by bits 63:62 of the virtual address. To provide compatibility for 32-bit programs, a 2^{32}-byte Compatibility Address Space is defined, separated into two non-contiguous ranges in which the upper 32 bits of the 64-bit address are the sign extension of bit 31. The Compatibility Address Space is further divided similarly into segments selected by bits 31:29 of the virtual address.

Figure 3.31 shows the layout of the Address Spaces, including the Compatibility Address Space and the segmentation of each Address Space.

Figure 3.31 Mapping 64-bit Address Space in 32-bit Mode



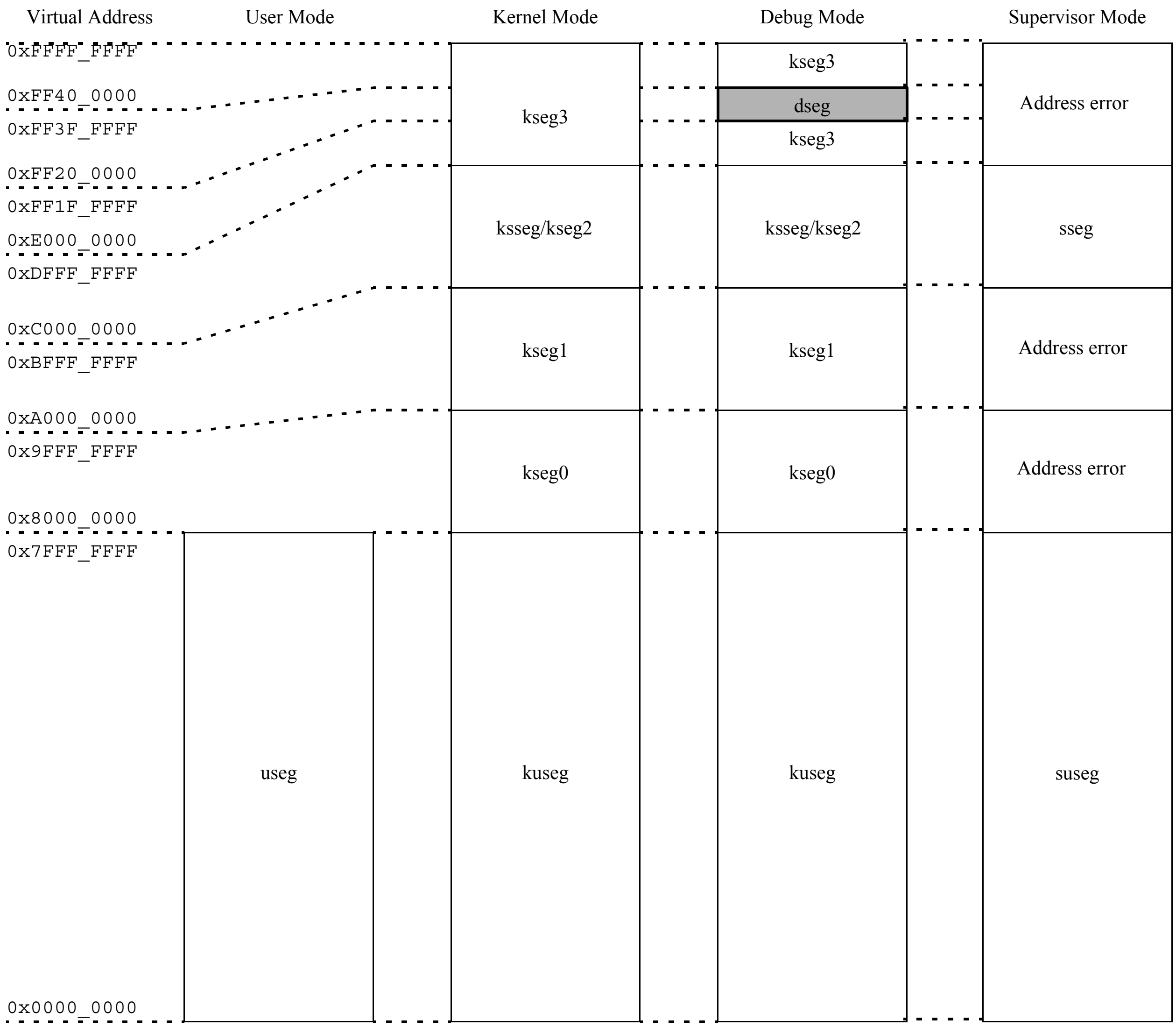
3.16.2.2 Virtual Memory Segments in 32-bit Mode

In the 32-bit mode, the P6600 core supports the traditional MIPS32 virtual address space, which contains fixed address ranges for the various user and kernel segments.

In 32-bit mode, the MIPS64 architecture supports a 4 GByte virtual address space that is partitioned into a number of segments, each characterized by a set of attributes defined by hardware and software. The virtual memory segments are different depending on the mode of operation. Figure 3.32 shows the segmentation for the 4 GByte (2^{32} bytes) virtual memory space, addressed by a 32-bit virtual address, for each of the four modes.

- User mode accesses are limited to a subset of the virtual address space (0x0000_0000_0000_0000 to 0x0000_0000_7FFF_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0xFFFF_FFFF_8000_0000 to 0xFFFF_FFFF_FFFF_FFFF are invalid and cause an exception if accessed.
- Supervisor mode adds access to sseg (0xFFFF_FFFF_C000_0000 to 0xFFFF_FFFF_DFFF_FFFF). kseg0, kseg1, and kseg3 will still cause exceptions if they are accessed.
- In Kernel mode, software has access to the entire address space, as well as all CP0 registers.
- Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as Kernel mode. In addition, while in Debug mode, the CPU has access to the debug segment (dseg). This area overlays part of the kernel segment kseg3. Access to dseg in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

Figure 3.32 Virtual Memory Map — 32-bit Mode

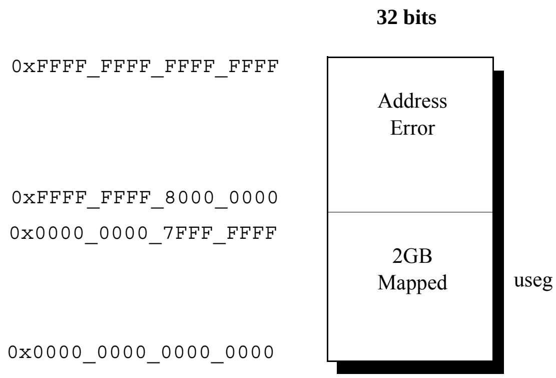


3.16.2.3 32-bit User Mode

In user mode, a single uniform virtual address space, called the user segment (useg), is available. The size of the user segment depends on the virtual addressing mode used.

In the 32-bit mode, the user segment occupies the lower 2 GB of virtual address space. The user segment starts at address 0x0000_0000_0000_0000 and ends at address 0x0000_0000_7FFF_FFFF. Accesses to all other addresses cause an address error exception. This is shown in Figure 3.33.

Figure 3.33 User Mode Virtual Address Space — 32-bit Configuration



The processor operates in 32-bit User mode when the *Status* register contains the following bit values:

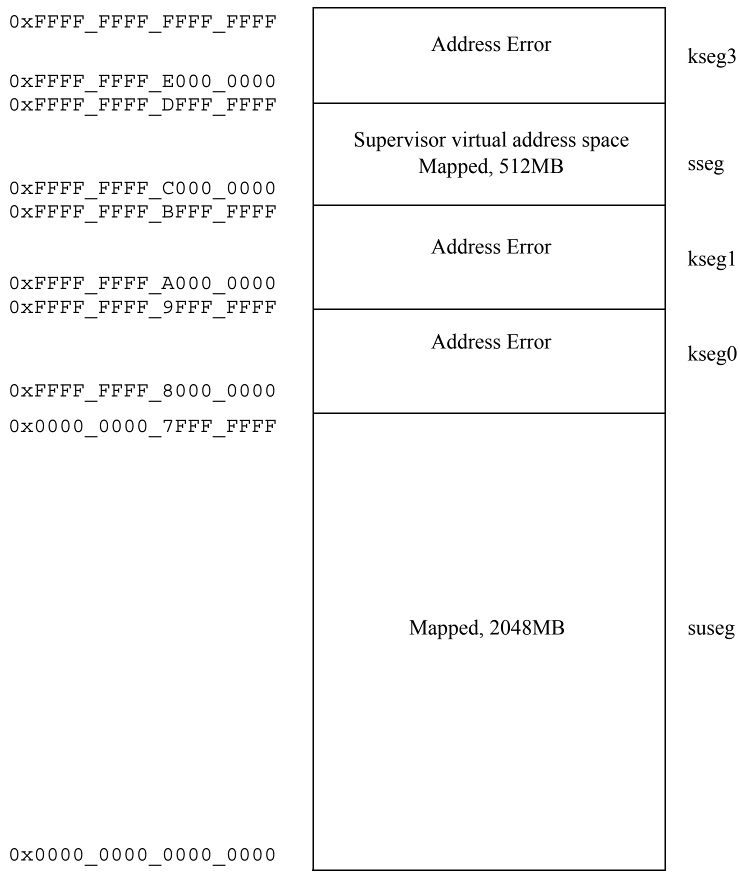
- *KSU* = 0b10
- *EXL* = 0
- *ERL* = 0
- UX = 0

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

3.16.2.4 32-bit Supervisor Mode

Supervisor mode includes a 512 MByte virtual address space called the supervisor segment (sseg). The supervisor-mode virtual address space is shown in Figure 3.34.

Figure 3.34 32-bit Supervisor Mode Virtual Address Space



The supervisor user segment (suseg) begins at address 0x0000_0000 and ends at address 0x7FFF_FFFF. The supervisor segment begins at 0xC000_0000 and ends at 0xDFFF_FFFF. Accesses to all other addresses in Supervisor mode cause an address error exception.

The processor operates in Supervisor mode when the *Status* register contains the following bit values:

- *KSU* = 2’b01
- *EXL* = 0
- *ERL* = 0
- SX = 0

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

Table 3.8 lists the characteristics of the Supervisor mode segments in the 32-bit mode.

Table 3.8 Supervisor Mode Segments — 32-bit Configuration

Address-Bit Value	Status Register Bit Value				Segment Name	Address Range	Segment Size
	EXL	ERL	UM	SM			
32-bit A(31) = 0	0	0	0	1	suseg	0x0000_0000_0000_0000 --> 0x0000_0000_7FFF_FFFF	2 GByte (2^{31} bytes)
32-bit A(31:29) = 3'b110	0	0	0	1	sseg	0xFFFF_FFFF_C000_0000 -> 0xFFFF_FFFF_DFFF_FFFF	512MB (2^{29} bytes)

The system maps all references to s*useg and sseg* through the TLB. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also, bit settings within the TLB entry for the page determine the cacheability of a reference.

3.16.2.5 32-bit Kernel Mode

The processor operates in Kernel mode when the *DM* bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- *KSU* = 2'b00, or
- *ERL* = 1. or
- *EXL* = 1, and
- KX = 0

When a non-debug exception is detected, *EXL* or *ERL* will be set and the processor enters Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears *ERL*, and clears *EXL* if ERL=0. This may return the processor to User mode.

In Kernel mode, a program has access to the entire virtual address space. Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3.35. The characteristics of kernel-mode segments are listed in Table 3.9.

The CPU enters Kernel mode both at reset and when an exception is recognized.

Figure 3.35 Kernel Mode Virtual Address Space — 32-bit Configuration

Table 3.9 Kernel Mode Segments

Address-Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	KSU	EXL	ERL			
A(31) = 0	(KSU = 00_2 or EXL = 1 or ERL = 1) and DM = 0			kuseg	0x0000_0000_0000_0000 through 0x0000_0000_7FFF_FFFF	2 GBytes (2^{31} bytes)
A(31:29) = 3'b100				kseg0	0xFFFF_FFFF_8000_0000 through 0xFFFF_FFFF_9FFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3'b101				kseg1	0xFFFF_FFFF_A000_0000 through 0xFFFF_FFFF_BFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3'b110				ksseg/kseg2	0xFFFF_FFFF_C000_0000 through 0xFFFF_FFFF_DFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3'b111				kseg3	0xFFFF_FFFF_E000_0000 through 0xFFFF_FFFF_FFFF_FFFF	512 MBytes (2^{29} bytes)

Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full 2^{31} bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000_0000_0000 - 0x0000_0000_7FFF_FFFF. For cores with TLBs, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address, and does not include the ASID field.

Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when virtual address bits VA[31:29] are 3'b100, 32-bit kseg0 virtual address space is selected; it is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0xFFFF_FFFF_8000_0000 - 0xFFFF_FFFF_9FFF_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The K0 field of the *Config* register controls cacheability.

Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when virtual address bits VA[31:29] are 3'b101, kseg1 virtual address space is selected. kseg1 is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0xFFFF_FFFF_A000_0000 - 0xFFFF_FFFF_BFFF_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

Kernel Mode, Kernel/Supervisor Space 2 (ksseg/kseg2)

In Kernel mode, when *KSU* = 2'b00, *ERL* = 1, or *EXL* = 1 in the *Status* register, and *DM* = 0 in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are 3'b110, 32-bit kseg2 virtual address space is selected.

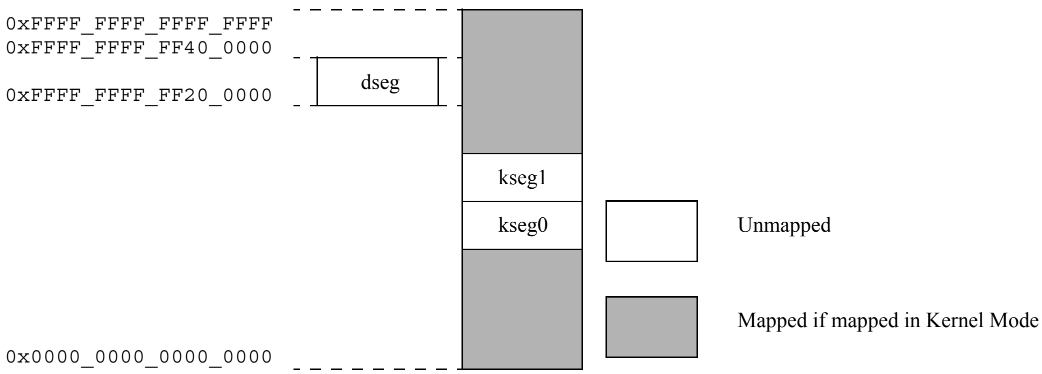
Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when virtual address bits VA[31:29] are 3'b111, the kseg3 virtual address space is selected. The kernel virtual space is located at physical addresses 0xFFFF_FFFF_E000_0000 - 0xFFFF_FFFF_FFFF_FFFF.

3.16.2.6 Debug Mode

Except for kseg3, debug-mode address space is identical to kernel-mode address space with respect to mapped and unmapped areas. In kseg3, a debug segment (dseg) coexists in the virtual address range 0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF3F_FFFF. The layout is shown in Figure 3.36.

Figure 3.36 Debug Mode Virtual Address Space



dseg is subdivided into the dmseg segment at 0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF2F_FFFF, which is used when the debug probe services the memory segment, and the drseg segment at 0xFFFF_FFFF_FF30_0000 to 0xFFFF_FFFF_FF3F_FFFF, which is used when memory-mapped debug registers are accessed. The subdivision and attributes of the segments are shown in Table 3.10.

Accesses to memory that would normally cause an exception in kernel mode cause the CPU to re-enter debug mode via a debug-mode exception. This includes accesses usually causing a TLB exception, with the result that such accesses are not handled by the usual memory-management routines.

The unmapped kseg0 and kseg1 segments from kernel-mode address space are available in debug mode, which allows the debug handler to be executed from uncached, unmapped memory.

Table 3.10 Physical Address and Cache Attributes for dseg, dmseg, and drseg

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFFFF_FFFF_FF20_0000 through 0xFFFF_FFFF_FF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space. drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	Uncached
	drseg	0xFFFF_FFFF_FF30_0000 through 0xFFFF_FFFF_FF3F_FFFF		

Debug Mode, Register (drseg)

The behavior of CPU access to the drseg address range at 0xFFFF_FFFF_FF30_0000 to 0xFFFF_FFFF_FF3F_FFFF is determined as shown in Table 3.11

Table 3.11 CPU Access to drseg

Transaction	LSNM Bit in Debug Register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the *Debug Control* register (*DCR*) to determine which other memory-mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory-mapped register is unpredictable, and writes are ignored to any unimplemented register in drseg. For more information about the *DCR*, refer to Chapter 13, "EJTAG Debug Support".

The allowed access size is limited for the drseg. Only word-size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

Debug Mode, Memory (dmseg)

The conditions for CPU accesses to the dmseg address range (0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF2F_FFFF) are shown in Table 3.12.

Table 3.12 CPU Access to dmseg

Transaction	ProbEn Bit in DCR Register[1]	LSNM Bit in Debug Register	Access
Load / Store	Don't care	1	Kernel mode address space (kseg3)
Fetch	1	Don't care	dmseg
Load / Store	1	0	dmseg
Fetch	0	Don't care	See comments below
Load / Store	0	0	See comments below

1. The NoDCR bit in the CP0 Debug register indicates if the dmseg and drseg address spaces and associated DCR register exists in memory mapped space. The NoDCR bit must be cleared, this DCR register exists. If the bit is set, the register does not exist.

An attempt to access dmseg when the ProbEn bit in the DCR register is 0 should not happen, because debug software is expected to check the state of the ProbEn bit in DCR register before attempting to reference dmseg. If such a reference does occur, the reference hangs until it is satisfied by the probe. The probe must not assume that there will never be a reference to dmseg when the ProbEn bit in the DCR register is 0, because there is an inherent race between the debug software sampling the ProbEn bit as 1, and the probe clearing it to 0.

3.16.3 64-Bit Mode

The MMU's virtual-to-physical address translation is determined by the mode in which the processor is operating. The P6600 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 3.13 Selecting the 64-bit Addressing Mode

	Status						Debug	
Mode	EXL	ERL	KSU	KX	SX	UX	DM	Description
User	0	0	2'b10	x	x	1	0	User addressing mode. In this mode, a TLB miss goes to the XTLB Refill Handler.
Supervisor	0	0	2'b01	x	1	x	0	Supervisor addressing mode. In this mode, a TLB miss goes to the XTLB Refill Handler.
Kernel	x	x	2'b00	1	x	x	0	Kernel addressing mode. In this mode, a TLB miss goesto the XTLB Refill Handler. The core is in the XKPhys address space when VA[63:62] = 2'b11.
	x	1	x	1	x	x	0	Kernel addressing mode. In this mode, a TLB miss goesto the XTLB Refill Handler. The core is in the XKPhys address space when VA[63:62] = 2'b11.
	1	x	x	1	x	x	0	Kernel addressing mode. In this mode, a TLB miss goesto the general exception handler as opposed to the XTLB Refill handler. The core is in the XKPhys address space when VA[63:62] = 2'b11.
Debug	x	x	x	x	x	x	1	Debug mode.

3.16.3.1 Virtual Memory Segments in 64-bit Mode

In the 64-bit mode, the P6600 core supports the full virtual address space, with fixed address ranges for the various segments as shown in Table 3.14. Bits 63:62 of the address determine which of the four address segments is accessed:

- Kernel Segment: VA[63:62] = 11
- XKPhys Segment: VA[63:62] = 10
- Supervisor Segment: VA[63:62] = 01
- User Segment: VA[63:62] = 00

User mode consists of two segments: a 32-bit compatible segment and a 64-bit segment. 32-bit compatible accesses are limited to a subset of the virtual address space 0x0000_0000_0000_0000 to 0x0000_0000_7FFF_FFFF and can be inhibited from accessing CP0 functions. 64-bit compatible accesses can access not only the 32-bit compatible space, but also the 64-bit user segment located at virtual address space 0x0000_0000_8000_0000 to 0x0000_FFFF_FFFF_FFFF. In User mode, virtual addresses 0x0001_0000_0000_0000 to 0x3FFF_FFFF_FFFF_FFFF are reserved as shown in the table below and cause an address error exception if accessed.

The Supervisor mode XKSSeg address space is accessed in 64-bit mode at virtual addresses 0x4000_0000_0000_0000 to 0x4000_FFFF_FFFF_FFFF. In Supervisor mode, virtual addresses 0x4001_0000_0000_0000 to 0x7FFF_FFFF_FFFF_FFFF are reserved as shown in the table below and cause an address error exception if accessed.

XKPhys address space can only be address by the kernel in 64-bit mode and reside at virtual addresses space 0x8000_0000_0000_0000 to 0xBFFF_FFFF_FFFF_FFFF. This address space is split into eight segments. Each segment contains a dedicated CCA value (0 - 7), aswell as a Reserved portion. Accesses to the Reserved portions shown in Table 3.14 cause an address error exception if accessed.

Kernel mode contains both 64-bit and 32-bit compatible segments. The XKseg segment can only be accessed in 64-bit mode and resides at virtual addresses 0xC000_0000_0000_0000 to 0xC000_FFFF_7FFF_FFFF. The Kseg0, Kseg1, SSeg/KSeg2, and Kseg3 segments are all 32-bit compatible. In Kernel mode, software has access to the entire address space (except reserved spaces) shown in Table 3.14, as well as all CP0 registers.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as Kernel mode. In addition, while in Debug mode, the CPU has access to the debug segment (dseg). This area overlays part of the kernel segment kseg3. Access to dseg in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

Table 3.14 MIPS64 Address Space

Segment	Address	Name	Mapping	CCA	Segment Type
Kernel [63:62] = 11	FFFF_FFFF_FFFF_FFFF - FFFF_FFFF_E000_0000	KSeg3	Kernel Mapped	From TLB	32-bit Compatible
	FFFF_FFFF_DFFF_FFFF - FFFF_FFFF_C000_0000	SSeg/Kseg2	Supervisor Mapped	From TLB	32-bit Compatible
	FFFF_FFFF_BFFF_FFFF - FFFF_FFFF_A000_0000	KSeg1	Kernel Unmapped	Uncached	32-bit Compatible
	FFFF_FFFF_9FFF_FFFF - FFFF_FFFF_8000_0000	KSeg0	Kernel Unmapped	From Config.K0	32-bit Compatible
	Reserved				
	C000_FFFF_7FFF_FFFF - C000_0000_0000_0000	XKSeg	Kernel Mapped	From TLB	64-bit
XKPhys [63:62] = 10	Reserved				
	B800_FFFF_FFFF_FFFF - B800_0000_0000_0000	XKPhys	Unmapped	CCA = 7	64-bit
	Reserved				
	B000_FFFF_FFFF_FFFF - B000_0000_0000_0000	XKPhys	Unmapped	CCA = 6	64-bit
	Reserved				
	A800_FFFF_FFFF_FFFF - A800_0000_0000_0000	XKPhys	Unmapped	CCA = 5	64-bit
	Reserved				
	A000_FFFF_FFFF_FFFF - A000_0000_0000_0000	XKPhys	Unmapped	CCA = 4	64-bit
	Reserved				
	9800_FFFF_FFFF_FFFF - 9800_0000_0000_0000	XKPhys	Unmapped	CCA = 3	64-bit
	Reserved				
	9000_FFFF_FFFF_FFFF - 9000_0000_0000_0000	XKPhys	Unmapped	CCA = 2	64-bit
	Reserved				
	8800_FFFF_FFFF_FFFF - 8800_0000_0000_0000	XKPhys	Unmapped	CCA = 1	64-bit
	Reserved				
	8000_FFFF_FFFF_FFFF - 8000_0000_0000_0000	XKPhys	Unmapped	CCA = 0	64-bit

Table 3.14 MIPS64 Address Space

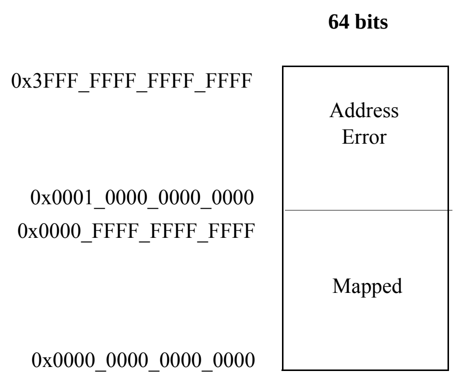
Segment	Address	Name	Mapping	CCA	Segment Type
Supervisor [63:62] = 01	Reserved				
	4000_FFFF_FFFF_FFFF - 4000_0000_0000_0000	XSSeg	Supervisor Mapped	From TLB	64-bit
User [63:62] = 00	Reserved				
	0000_FFFF_FFFF_FFFF - 0000_0000_8000_0000	XUSeg	User Mapped	From TLB	64-bit
	0000_0000_7FFF_FFFF - 0000_0000_0000_0000	USeg	User Mapped	From TLB	32-bit Compatible

3.16.3.2 64-bit User Mode

In 64-bit user mode, a single uniform virtual address space, called the user segment (useg), is available.

The user segment occupies the portion of the virtual address space shown below. The user segment starts at address 0x0000_0000_0000_0000 and ends at address 0x0000_FFFF_FFFF_FFFF. Accesses to addresses 0x0001_0000_0000_0000 and ends at address 0x3FFF_FFFF_FFFF_FFFF cause an address error exception. This is shown in Figure 3.37.

Figure 3.37 User Mode Virtual Address Space — 64-bit Address Mode



The processor operates in User mode when the *Status* register contains the following bit values:

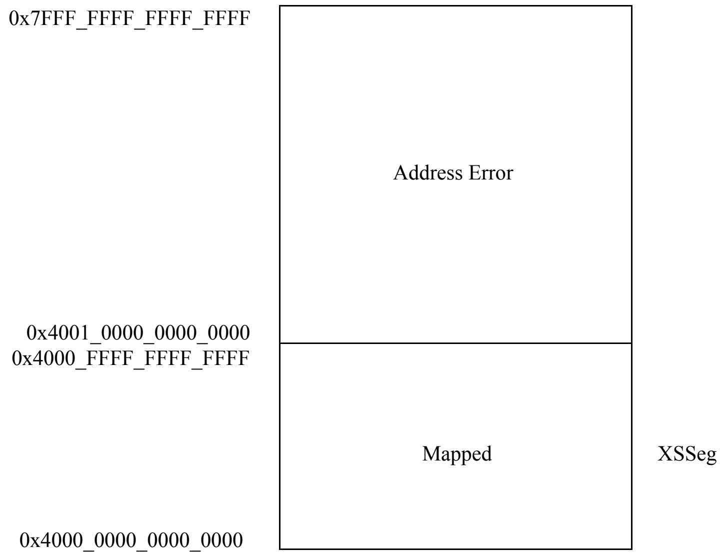
- *KSU* = 2’b10
- *EXL* = 0
- *ERL* = 0
- UX = 1

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

3.16.3.3 64-bit Supervisor Mode

The 64-bit supervisor-mode virtual address space is shown in Figure 3.38. Accesses to addresses 0x4001_0000_0000_0000 - 0x7FFF_FFFF_FFFF_FFFF in Supervisor space cause an address error exception.

Figure 3.38 64-bit Supervisor Mode Virtual Address Space



The accessible supervisor segment begins at address 0x4000_0000_0000_0000 and ends at address 0x4000_FFFF_FFFF_FFFF. The processor operates in Supervisor mode when the *Status* register contains the following bit values:

- *KSU* = 2'b01
- *EXL* = 0
- *ERL* = 0
- SX = 0

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

3.16.3.4 64-bit Kernel Mode

Kernel mode has access to the entire 64-bit address space (except reserved spaces), including supervisor and user mode spaces, and the entire XKPhys address segment. The processor operates in Kernel mode when the *DM* bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

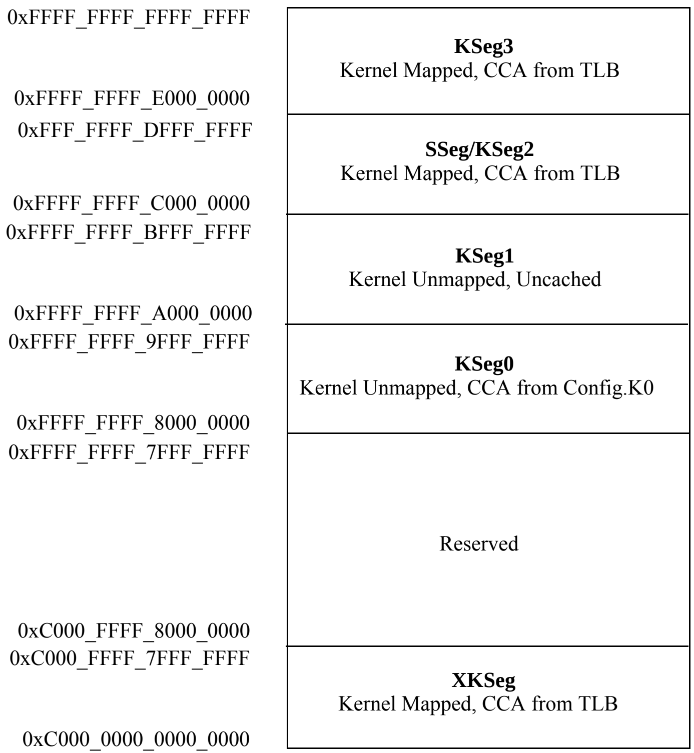
- *KSU* = 2'b00, or
- *ERL* = 1, or
- *EXL* = 1, and
- KX = 1

When a non-debug exception is detected, hardware sets the *EXL* or *ERL* bits in the *Status* register and the processor enters Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears *ERL*, and clears *EXL* if ERL=0. This may return the processor to User mode.

In Kernel mode, a program has access to the entire virtual address space. Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3.35. The characteristics of kernel-mode segments are listed in Table 3.9.

The CPU enters Kernel mode both at reset and when an exception is recognized.

Figure 3.39 Kernel Mode 64-bit Virtual Address Space



Kernel Mode, Kernel User Space (XKSeg)

The XKSeg segment is accessed under the following conditions:

- The most significant bits of the address (VA[63:62]) are 2’b11, and
- VA[61:48] of the virtual address are all 0’s, and
- The address does not fall in reserved address space of 0xC000_FFFF_8000_0000 to 0xC000_FFFF_FFFF_FFFF

In this configuration, kernel virtual user space is located at addresses 0xC000_0000_0000_0000 - 0xC000_FFFF_7FFF_FFFF. References to XKSeg are kernel mapped and the CCA attributes come from the TLB.

Kernel Mode, Kernel Space 1 (KSeg1)

The KSeg1 segment is accessed under the following conditions:

- The most significant bits of the address (VA[63:62]) are 2’b11, and
- VA[61:32] of the virtual address are all 1’s, and
- VA[31:29] is 3’b101

In this configuration, kernel virtual space 1 is located at addresses 0xFFFF_FFFF_A000_0000 - 0xFFFF_FFFF_BFFF_FFFF. References to XKSeg0 are kernel unmapped and uncached. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

Kernel Mode, Kernel/Supervisor Space 2 (KSSeg/KSeg2)

The KSeg1 segment is accessed under the following conditions:

- The most significant bits of the address (VA[63:62]) are 2'b11, and
- VA[61:32] of the virtual address are all 1's, and
- VA[31:29] is 3'b110

In this configuration, kernel virtual space 2 is located at addresses 0xFFFF_FFFF_C000_0000 - 0xFFFF_FFFF_DFFF_FFFF. References to XKSeg2 are supervisor mapped, and the CCA for this segment is defined by the TLB.

Kernel Mode, Kernel Space 3 (KSeg3)

The KSeg3 segment is accessed under the following conditions:

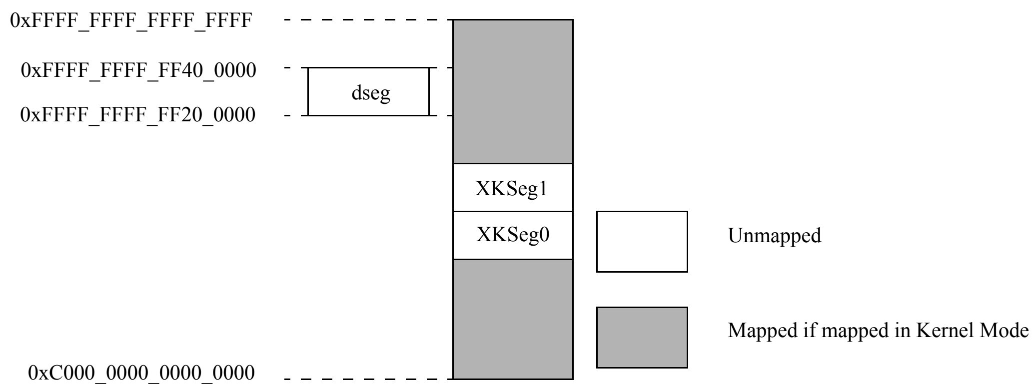
- The most significant bits of the address (VA[63:62]) are 2'b11, and
- VA[61:32] of the virtual address are all 1's, and
- VA[31:29] is 3'b111

In this configuration, kernel virtual space 3 is located at addresses 0xFFFF_FFFF_E000_0000 - 0xFFFF_FFFF_FFFF_FFFF. References to XKSeg3 are kernel mapped, and the CCA for this segment is defined by the TLB.

3.16.3.5 64-bit Debug Mode

Except for XKSeg3, debug-mode address space is identical to kernel-mode address space with respect to mapped and unmapped areas. In XKSeg3, a debug segment (dseg) coexists in the virtual address range 0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF3F_FFFF. The layout is shown in Figure 3.40.

Figure 3.40 64-Bit Debug Mode Virtual Address Space



dseg is subdivided into the dmseg segment at 0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF2F_FFFF, which is used when the debug probe services the memory segment, and the drseg segment at 0xFFFF_FFFF_FF30_0000 to 0xFFFF_FFFF_FF3F_FFFF, which is used when memory-mapped debug registers are accessed. The subdivision and attributes of the segments are shown in Table 3.10.

Accesses to memory that would normally cause an exception in kernel mode cause the CPU to re-enter debug mode via a debug-mode exception. This includes accesses usually causing a TLB exception, with the result that such accesses are not handled by the usual memory-management routines.

The unmapped XKSeg0 and XKSeg1 segments from kernel-mode address space are available in debug mode, which allows the debug handler to be executed from uncached, unmapped memory.

Table 3.15 Physical Address and Cache Attributes for dseg, dmseg, and drseg

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFFFF_FFFF_FF20_0000 through 0xFFFF_FFFF_FF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space. drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	Uncached
	drseg	0xFFFF_FFFF_FF30_0000 through 0xFFFF_FFFF_FF3F_FFFF		

Debug Mode, Register (drseg)

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 3.11

Table 3.16 CPU Access to drseg

Transaction	LSNM Bit in Debug Register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don’t care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the *Debug Control* register (*DCR*) to determine which other memory-mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory-mapped register is unpredictable, and writes are ignored to any unimplemented register in drseg. For more information about the *DCR*, refer to Chapter 13, “EJTAG Debug Support”.

The allowed access size is limited for the drseg. Only word-size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

Debug Mode, Memory (dmseg)

The conditions for CPU accesses to the dmseg address range (0xFFFF_FFFF_FF20_0000 to 0xFFFF_FFFF_FF2F_FFFF) are shown in Table 3.17.

Table 3.17 CPU Access to dmseg

Transaction	ProbEn Bit in DCR Register[1]	LSNM Bit in Debug Register	Access
Load / Store	Don’t care	1	Kernel mode address space (kseg3)
Fetch	1	Don’t care	dmseg
Load / Store	1	0	dmseg

Table 3.17 CPU Access to dmseg

Transaction	ProbEn Bit in DCR Register[1]	LSNM Bit in Debug Register	Access
Fetch	0	Don't care	See comments below
Load / Store	0	0	See comments below

1. The NoDCR bit in the CP0 Debug register indicates if the dmseg and drseg address spaces and associated DCR register exists in memory mapped space. The NoDCR bit must be cleared, this DCR register exists. If the bit is set, the register does not exist.

An attempt to access dmseg when the ProbEn bit in the DCR register is 0 should not happen, because debug software is expected to check the state of the ProbEn bit in DCR register before attempting to reference dmseg. If such a reference does occur, the reference hangs until it is satisfied by the probe. The probe must not assume that there will never be a reference to dmseg when the ProbEn bit in the DCR register is 0, because there is an inherent race between the debug software sampling the ProbEn bit as 1, and the probe clearing it to 0.

3.16.3.6 64-bit XKPhys Address Segment

The Extended Kernel Physical Segment (XKPhys) is divided into a series of eight equal segments, each with a different Cache Coherency Attribute (CCA). The attribute information is stored in the C field of the *EntryLo0* and *EntryLo1* registers.

The eight segments reside within the following address ranges.

- 8000_0000_0000_0000 — 8000_FFFF_FFFF_FFFF: XKPhys0, CCA = 0
- 8800_0000_0000_0000 — 8800_FFFF_FFFF_FFFF: XKPhys1, CCA = 1
- 9000_0000_0000_0000 — 9000_FFFF_FFFF_FFFF: XKPhys2, CCA = 2
- 9800_0000_0000_0000 — 9800_FFFF_FFFF_FFFF: XKPhys3, CCA = 3
- A000_0000_0000_0000 — A000_FFFF_FFFF_FFFF: XKPhys4, CCA = 4
- A800_0000_0000_0000 — A800_FFFF_FFFF_FFFF: XKPhys5, CCA = 5
- B000_0000_0000_0000 — B000_FFFF_FFFF_FFFF: XKPhys6, CCA = 6
- B800_0000_0000_0000 — B800_FFFF_FFFF_FFFF: XKPhys7, CCA = 7

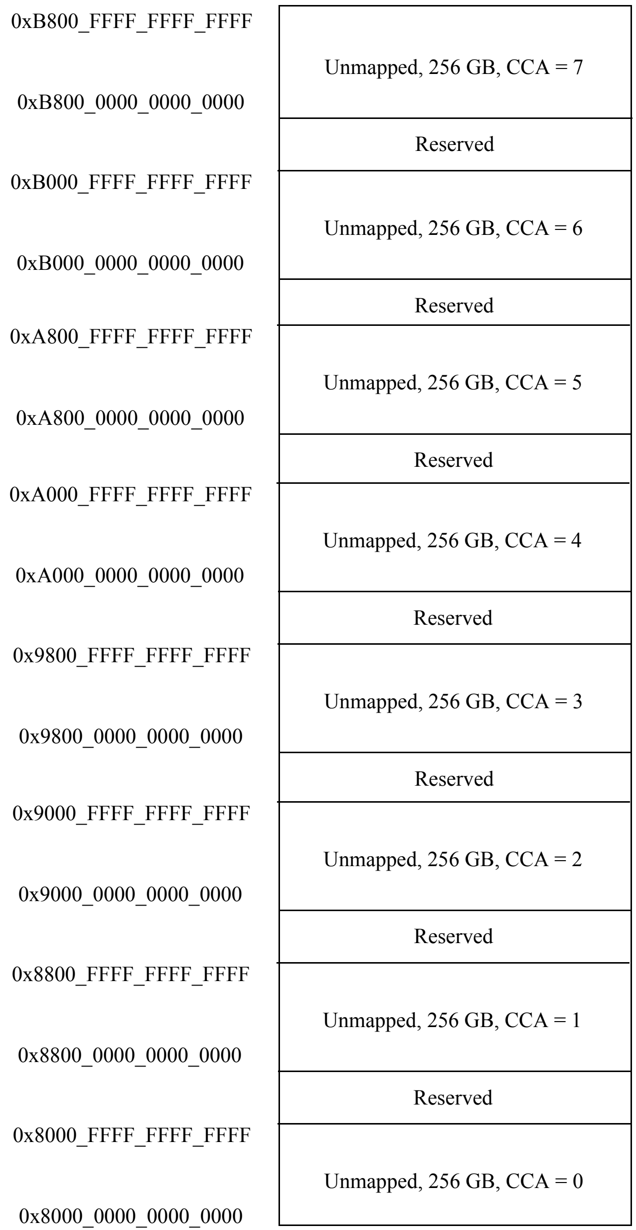
Note that there are empty spaces or gaps between each XKPhys memory segment. These empty spaces are reserved and cause an address error exception if accessed. For example, the address range of 8001_0000_0000_0000 — 87FF_FFFF_FFFF_FFFF is the empty space between the XKPhys0 and XKPhys1 address spaces.

In the P6600 core address space, the following types of accesses are supports;

- Uncached (CCA = 2)
- Cache Coherent Read (CCA = 5)
- Uncached Accelerated (CCA = 7).

All CCA values map to one of these attributes.

Figure 3.41 XKPhys Address Segments in 64-bit Virtual Address Space



3.17 TLB Instructions

Table 3.18 lists the TLB-related instructions implemented in the P6600 core. .

Table 3.18 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 VTLB flush based invalidation of TLB entries.

Refer to the Instructions chapter for more information on the TLB instructions.

Chapter 4

Caches

This chapter describes the caches present in an P6600 core and contains the following sections:

- Section 4.1 "Cache Configurations"
- Section 4.2 "L1 Instruction Cache"
- Section 4.3 "L1 Data Cache"
- Section 4.4 "L1 Instruction and Data Cache Software Testing"
- Section 4.5 "L2 Cache"
- Section 4.6 "The CACHE Instruction"

4.1 Cache Configurations

The P6600 core contains three caches; L1 instruction, L1 data, and shared L2. These caches are non-optional in the P6600 architecture and are always present. The size of each cache can be configured as shown in Table 4.1.

Table 4.1 P6600 Cache Configurations

Attribute	L1 Instruction Cache	L1 Data Cache	L2 Cache
Size[1]	32 KB or 64 KB	32 KB or 64 KB	512 KB 1 MB, 2 MB, 4 MB, or 8 MB
Line Size	32-byte	32-byte	32-byte
Number of Cache Sets	256 or 512	256 or 512	2048, 4096, 8192, 16384, or 32768
Associativity	4 way	4 way	8 way

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

The L1 instruction cache is attached to the Instruction Fetch Unit (IFU) via two 64-bit data paths, allowing for up to four instruction fetches per cycle. The L1 data cache contains two 64-bit data paths, allowing for up to two data read/write operations per cycle. The L2 cache is embedded within the Coherence Manager (CM2) and communicates with external memory via a configurable 128-bit or 256-bit OCP interface.

For more information on the L1 instruction cache, refer to Section 4.2 "L1 Instruction Cache".

For more information on the L1 data cache, refer to Section 4.3 "L1 Data Cache".

For more information on the L2 cache, refer to Section 4.5 "L2 Cache".

4.1.1 Cacheability Attributes

The P6600 core supports the following cacheability attributes:

- *Uncached (code #2)*: Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- *Non-coherent Writeback With Write Allocation (code #3)*: 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 (code #4)*: This attribute is similar to code #5 described below, 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 (code #5)*: 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 (code #7):* Uncached stores are gathered together for more efficient bus utilization.

4.2 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 select the way to be filled.

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.

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 4.2 shows the key characteristics of the L1 instruction cache. Figure 4.1 shows the format of an entry in the three arrays comprising the instruction cache: data, tag, and way-select.

Table 4.2 L1 Instruction Cache Attributes

Attribute	With Parity
Size[1]	32 KB or 64 KB
Line Size	32-byte
Number of Cache Sets	256 or 512
Associativity	4-way
Replacement	LRU
Cache Locking	per line
Data Array	
Read Unit	144b x 4
Write Unit	144b
Tag Array	
Read Unit	63b x 4
Write Unit	63b
Way-Select Array	
Read Unit	6b
Write Unit	1-6b

1. For Linux based applications, MIPS recommends a 64 KB L1 instruction cache size, with a minimum size of 32 KB.

Figure 4.1 L1 Instruction Cache Organization

Tag (per way): (63 bits total)

5	1	1	28	7	7	7	7
Parity	Valid	Lock	PA[39:12]	Precode_67	Precode_45	Precode_23	Precode_01

Data (per way): (144 bits total)

8	8	32	32	32	32
Parity	Parity	word3	word2	word1	word0

Way-Select: (6 bits total)

6
LRU

4.2.1 L1 Instruction Cache Virtual Aliasing

The instruction cache on the P6600 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.

In the P6600 core, the `Config7`$_{IAR}$ bit is always set to indicate the existence of instruction cache virtual aliasing hardware. 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 16KB. 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.

4.2.2 L1 Instruction Cache Precode Bits

In order for the fetch unit to quickly detect branches and jumps when executing code, the instruction cache array contains some additional precode bits. These bits indicate the type and location of branch or jump instructions within a 64b fetch bundle.

4.2.3 L1 Instruction Cache Parity

The instruction cache contains 16 parity bits — one for each byte of the 128 bits of data. The tag array has 5 parity bits for each tag, one for each of the 4 precodefields and one for the physical tag, lock, and valid bits. The LRU array does not have any parity. Instruction cache parity is always present in the instruction cache and cannot be disabled.

4.2.4 L1 Instruction Cache Replacement Policy

The L1 instruction cache replacement policy refers to how a way is chosen to hold an incoming cache line on a miss which will result in a cache fill. The replacement policy is least-recently used (LRU), but excluding any locked ways. The LRU bit(s) in the way-select array encode the order in which ways on that line have been accessed.

On a cache miss, the lock and LRU bits for the tag and way-select entries of the selected line may be used to determine the way which will be chosen.

The LRU field in the way select array is updated as follows:

- On a cache hit, the associated way is updated to be the most recently used. The order of the other ways relative to each another is unchanged.
- On a cache refill, the filled way is updated to be the most recently used.
- On CACHE instructions, the update of the LRU bits depends on the type of operation to be performed:
  - **Index (Writeback) Invalidate**: Least-recently used.
  - **Index Load Tag**: No update.
  - **Index Store Tag,** *WST* **= 0:** Most-recently used if valid bit is set in *TagLo* CP0 register. Least-recently used if valid bit is cleared in *TagLo* CP0 register.
  - **Index Store Tag,** *WST* **= 1:** Update the field with the contents of the *TagLo* CP0 register.
  - **Index Store Data:** No update.
  - **Hit Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
  - **Fill:** Most-recently used.
  - **Hit Writeback:** No update.
  - **Fetch and Lock:** For instruction cache, no update. For data cache, most-recently used.

If all ways are valid, then any locked ways are excluded from consideration for replacement. For the unlocked ways, the LRU bits are used to identify the way which has been used least-recently, and that way is selected for replacement.

4.2.5 L1 Instruction Cache Line Locking

The P6600 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.

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. Since 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.

4.2.6 L1 Instruction Cache Memory Coherence Issues

The P6600 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 P6600 core monitors its Intervention Port and updates the state of its cache lines (valid, lock, and dirty tag bits) accordingly.

The L1 instruction caches utilizes a modified MESI protocol. Each cache line will be in one of the following states:

Invalid: The line is not present in this cache.

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 and is consistent with the value in L2 cache or memory.

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

In the P6600 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.

4.2.7 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
```

4.2.8 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, as described in Section 4.2.11 "Cache Management When Writing Instructions - the "SYNCI" Instruction".

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

31 26	25 21	20 18	17 16	15 0
cache	base	op		offset
47	register	what to do	which cache	

Figure 4.2 Fields in the Encoding of a CACHE Instruction

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	Way1-0	Index	byte-within-line

Note that the MIPS64 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.

4.2.9 L1 Instruction Cache CP0 Register Interface

The P6600 core uses different CP0 registers for instruction cache operations.

Table 4.3 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*

4.2.9.1 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 P6600 L1 instruction cache supports 256 sets per way, which is used to configure a 32 KB cache, or 512 sets per way, which is used to configure a 64 KB cache.

The $Config1_{IL}$ field (bits 21:19) indicates the line size for the instruction cache. The P6600 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 P6600 L1 instruction cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

For more information, refer to Section 2.2.1.2, "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)".

4.2.9.2 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

For more information, refer to Section 2.2.5.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.2.9.3 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 $ErrCtl_{WST}$ bit

- Default cache interface mode ($ErrCtl_{WST}$ = 0)
- Diagnostic "way select test mode" ($ErrCtl_{WST}$ = 1)

For more information, refer to Section 2.2.5.1, "Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)".

4.2.9.4 L1 Instruction Cache TagHi Register (CP0 register 29, Select 0)

This register represents the I-cache pre-decode bits and is intended for diagnostic use only.

For more information, refer to Section 2.2.5.2, "Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)".

4.2.9.5 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 P6600 core loads I-cache data at least 64 bits at a time. This register stores the lower 32 bits of the load data.

For more information, refer to Section 2.2.5.3, "Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)".

4.2.9.6 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 P6600 core loads I-cache data at least 64 bits at a time. This register stores the upper 32 bits of the load data.

For more information, refer to Section 2.2.5.4, "Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)".

4.2.10 L1 Instruction Cache Initialization

The L1 instruction cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user’s, a sample boot code is shown in the following subsection.

4.2.10.1 L1 Instruction Cache Initialization Routine

The following assembly provides an example initialization routine for the instruction cache.

```
/************************************************************************************
init_icache invalidates all Instruction cache entries
************************************************************************************/

LEAF(init_icache)

      // For this Core there is always an instruction cache
      // The IS field determines how many sets there are:
```

```
    // IS = 2 there are 256 sets
    // IS = 3 there are 512 sets
    // $11 set to line size, will be used to increment through the cache tags

    li    $11, 32           # Line size is always 32 bytes.
    mfc0  $10, $16, 1       # Read C0_Config1
    ext    $12, $10, 22, 3  # Extract IS
    li    $14, 2            # Used to test against
    beq   $14, $12, Isets_done# if  IS = 2
    li    $12, 256          # sets = 256
    li    $12, 512          # else sets = 512 Skipped if branch taken
Isets_done:
    lui   $14, 0x8000       # Get a KSeg0 address for cacheops
    // clear the lock bit, valid bit, and the LRF bit
    mtc0    $0, $28         # Clear C0_ITagLo to invalidate entry

next_icache_tag:
    cache   0x8, 0($14)     # Index Store tag Cache opt
    add       $12, -1       # Decrement set counter
    bne     $12, $0, next_icache_tag # Done yet?
    add     $14, $11        # Increment line address by line size
done_icache:

    ins     r31_return_addr, $0, 29, 1
    jr      r31_return_addr
    nop
END(init_icache)
```

4.2.10.2 L1 Instruction Cache Initialization Routine Details

This section provides a detailed description of each line of code in the L1 instruction cache initialization routine described above. Note that this code represents an example of an implementation specific cache initialization. The code is used in specific cache sizes of 32K or 64K, is always part of the P6600 MPS, and always have the L2 cache present. The code example is written with those parameters in mind.

Before use, the cache must be initialized to a known state; that is, all cache entries must be invalidated. This code example initializes the cache, finds the total number of cache sets, then loops through the cache sets using the cache instruction to invalidate each cache set.

```
LEAF (init_icache)

// For this Core there is always an L1 instuction cache
// The IS field determines how many sets there are
// IS = 2 there are 256 sets
// IS = 3 there are 512 sets
// $11 set to line size, will be used to increment through the cache tags

    li    $11, 32           # Line size is always 32 bytes.
```

This instruction cache always has a line size of 32 bytes, 4 ways and can have a size of either 32 KB or 64 KB. The IS field (sets per way) of the *Config1* register will be use to determine the size of the cache. This field can have one of two values. A value of 0x2 indicates a 32 KB cache and a value of 0x3 indicates a 64 KB cache.

```
mfc0  $10, $16, 1       # Read C0_Config1
ext   $12, $10, 22, 3   # Extract IS
li    $14, 2            # Used to test against
```

If the check is true, the code uses the branch delay slot (which is always executed) to set the set iteration value to 256 for a 32 KB cache and then branches ahead to **Isets_done**. If the check is false, the code assumes that the size of the cache is 64 KB. At this point, the code still sets the iteration value to 256 in the branch delay slot, but then falls through and sets it again to 512 for a 64 KB cache.

```
beq   $14, $12, Isets_done # if  IS = 2
li    $12, 256          # sets = 256
li    $12, 512          # else sets = 512 Skipped if branch taken
```

`Isets_done:`

GPR 14 will be used as an index into the cache. It will be set to a virtual address, and then translated to a physical address. Since the address 0x8000_0000 is in kseg0, the CPU will ignore the top bit, so virtual 0x8000_0000 will become physical address 0x0000_0000. Since the cache is physically indexed, the first time through the loop, the cache instruction will write the tag to way 0 index line 0.

The **lui** instruction loads 0x8000 into the upper 16 bits and clears the lower 16 bits of the GPR14 register.

```
lui   $14, 0x8000       # Get a KSeg0 address for cacheops
```

Clearing the tag registers performs two important functions: it sets the Physical Tag address called PTagLo to 0, which ensures the upper physical address bits are zeroed out, and it also clears the valid bit for the set, which ensures that the set is free and may be filled as needed.

The code uses the Move to Coprocessor Zero (MTC0) instruction to move the general purpose register zero, which always contains a zero, to the tag register.

```
// clear the lock bit, valid bit, and the LRF bit

      mtc0  $0, $28     # Clear C0_ITagLo to invalidate entry
```

The **`Cache`** instruction uses the **`Index Store Tag`** operation on the Level 1 instruction cache so the op field is coded with a value of 0x8. The first two bits are 2'b00 for the L1 instruction cache, and the operation code for **`Index Store tag`** is encoded as 3'b010 in bits two, three and four.

`next_icache_tag:`

```
cache 0x8, 0($14) # Index Store tag Cache op
```

The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by breaking down the virtual address argument stored in the base register of the **`Cache`** instruction into several fields.

Bits 14:0 of the Cache Instruction

14 13	12 5	4 0
Way	Page Index	Byte Index

The size of the index field varies according to the size of a cache way. The larger the way, the larger the index. In the table above, the combined byte and page index is 13 bits because each way of the cache is 8K. The way number is always the next two bits following the index.

The code does not explicitly set the way bits. Instead it just increments the virtual address by the cache lines size so the next time through the loop the **`Cache`** instruction will initialize the next set in the cache. Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache because it overflows into the way bits.

At this point all the code needs to do is loop maintenance. First decrement the loop counter (12/t4).

```
add   $12, -1                 # Decrement set counter
```

Then test it to see if it has gotten to zero and if it has not branch back to label one.

```
bne   $12, $0, next_icache_tag # Done yet?
```

The instruction in the branch delay slot, which is always executed, is used to increment the virtual address (14/t6) to the next set in the cache. (11/t3) holds the line size in bytes.

```
add   $14, $11          # Increment line address by line size
```

From this point on, the code can be executed from a cached address. This is easily done by changing the return address from a KSEG1 address to a KSEG0 address by simply inserting a 0 into bit 29 of the address. However, during debugging, this operation will confuse the debugger and you will no longer be able to do source-level debugging. That is why it is commented out here. Once the code has been debugged, the "ins" line can be uncommented.

```
done_icache:

      // Modify return address to kseg0 which is cacheable
      // (for code linked in kseg1.)
      // However it makes it easier to debug if this is not done. So while
      // debugging, this should be commented out.

      ins     r31_return_addr, $0, 29, 1
      jr      r31_return_addr
      nop

END (init_icache)
```

4.2.11 Cache Management When Writing Instructions - the "SYNCI" Instruction

The **`synci`** instruction provides a mechanism available to user-level code for ensuring that previously written instructions are correctly presented for execution (it combines a D-cache writeback with an I-cache invalidate). Use of the **`synci`** instruction is preferred to the traditional alternative of a D-cache writeback followed by an I-cache invalidate.

4.3 L1 Data Cache

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

- In addition to the three arrays (tag, data, and way-select), the L1 data cache also contains a separate dirty array to hold the dirty bits of cache lines.
- The data cache does not contain any precode information.
- To handle store bytes, the data array is byte-accessible, and the data parity is 1 bit per byte.
- The way-select array for the data cache holds the lock bits (and lock parity bits) for each cache line, in addition to the LRU information. The lock bits indicate the cache lines that have been locked using the **`CACHE`** instruction.

Like the L1 instruction cache, the L1 data cache is virtually indexed, since 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 [39:11], 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, half-word, 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 4.4 shows the key characteristics of the data cache. Figure 4.3 shows the format of an entry in the arrays comprising the data cache: tag, data, way-select, and dirty.

Table 4.4 L1 Data Cache Organization

Attribute	With Parity
Size	32 or 64KB
Line Size	32-byte
Number of Cache Sets	256 or 512
Associativity	4-way
Replacement	LRU
Cache Locking	per line
Data Array	
Read Unit	144b x 4
Write Unit	144b
Tag Array	
Read Unit	32b x 4
Write Unit	32b

Table 4.4 L1 Data Cache Organization *(continued)*

Attribute	With Parity
Way-Select Array	
Read Unit	14b
Write Unit	1-14b
Dirty Array	
Read Unit	10b
Write Unit	1-10b

Figure 4.3 L1 Data Cache Organization

Tag (per way): (32 bits total)

1	1	29	1
Parity	State	PA39:11	Valid

Data (per way): (144 bits total)

15	Parity 1 - 14	1	8	Bytes 2 - 14	8	8
Parity	...	Parity	Data15	...	Data1	Data0

Way-Select: (14 bits total)

4	4	6
Lock Parity	Lock	LRU

Dirty (10 bits total)

2	4	4
Reserved	Dirty Parity	Dirty

4.3.1 L1 Data Cache Virtual Aliasing

The data cache on the P6600 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 4.5 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 P6600 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 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.

4.3.2 L1 Data Cache Parity

The L1 cache data parity provides one parity bit for each byte, corresponding to the minimum number of bytes for a store. The tag array has a single parity bit for each tag. The way-select array has separate parity bits to cover each dirty bit, but the LRU bits are not covered by parity Instruction cache parity is always present in the instruction cache and cannot be disabled.

4.3.3 L1 Data Cache Replacement Policy

The replacement policy refers to how a way is chosen to hold an incoming cache line on a miss which will result in a cache fill. The replacement policy is least-recently used (LRU), but excluding any locked ways. The LRU bit(s) in the way-select array encode the order in which ways on that line have been accessed.

On a cache miss, the lock and LRU bits for the tag and way-select entries of the selected line may be used to determine the way which will be chosen.

The LRU field in the way select array is updated as follows:

- On a cache hit, the associated way is updated to be the most recently used. The order of the other ways relative to each another is unchanged.
- On a cache refill, the filled way is updated to be the most recently used.
- On CACHE instructions, the update of the LRU bits depends on the type of operation to be performed:
  - **Index (Writeback) Invalidate**: Least-recently used.
  - **Index Load Tag**: No update.
  - **Index Store Tag,** *WST* **= 0:** Most-recently used if valid bit is set in *TagLo* CP0 register. Least-recently used if valid bit is cleared in *TagLo* CP0 register.
  - **Index Store Tag,** *WST* **= 1:** Update the field with the contents of the *TagLo* CP0 register.
  - **Index Store Data:** No update.
  - **Hit Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
  - **Fill:** Most-recently used.
  - **Hit (Writeback) Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
  - **Hit Writeback:** No update.
  - **Fetch and Lock:** For instruction cache, no update. For data cache, most-recently used.

If all ways are valid, then any locked ways will be excluded from consideration for replacement. For the unlocked ways, the LRU bits are used to identify the way which has been used least-recently, and that way is selected for replacement.

If the way selected for replacement has its dirty bit asserted in the way-select array, then that 32-byte line will be written back to memory before the new fill can occur.

4.3.4 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 Section 4.2.5, "L1 Instruction Cache Line Locking".

4.3.5 L1 Data Cache Memory Coherence Protocol

The P6600 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 P6600 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.

4.3.6 L1 Data Cache Initialization

The L1 data cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user's, a sample boot code is shown in the following subsection.

4.3.6.1 L1 Data Cache Initialization Routine

The following assembly provides an example initialization routine for the data cache.

```
/****************************************************************************
init_dcache invalidates all data cache entries
****************************************************************************/

LEAF (init_dcache)

      // For the P6600 MPSthere is always an L1 data cache
      // The ID field determines how many sets there are
      // DS = 2 there are 256 sets
      // DS = 3 there are 512 sets
      // $11 set to line size, will be used to increment through the cache tags

      li    $11, 32           # Line size is always 32 bytes
      mfc0  $10, $16, 1       # Read C0_Config1
      ext    $12, $10, 13, 3  # Extract DS
      li    $14, 2            # Used to test against
      beq   $14, $12, Dsets_done # if  DS = 2
      li    $12, 256          # sets = 256
      li    $12, 512          # else sets = 512, skipped if branch taken

Dsets_done:

      lui     $14, 0x8000     # Get a KSeg0 address for cacheops
      // clear the lock bit, valid bit, and the LRF bit
      mtc0    $0, $28, 2      # Clear C0_DTagLo to invalidate entry

next_dcache_tag:

      cache 0x9, 0($14)       # Index Store tag Cache opt
      add    $12, -1          # Decrement set counter
      bne   $12, $0, next_dcache_tag # Done yet?
      add    $14, $11         # Increment line address by line size

done_dcache:

       jr      r31_return_addr
      nop

END (init_dcache)
```

4.3.6.2 L1 Data Cache Initialization Routine Details

This section provides a detailed description of each line of code in the initialization routine. The L1 data cache initialization routine is very similar to the L1 instruction cache initialization routine.

```
LEAF(init_dcache)

      // For the P6600 CPS there is always a L1 data cache
      // The DS field determines how many sets there are
```

```
// DS = 2 there are 256 sets
// DS = 3 there are 512 sets
// $11 set to line size, will be used to increment through the cache tags

li    $11, 32           # Line size is always 32 bytes.
```

The data cache always has a line size of 32 bytes and 4 ways, and can have a size of either 32 KB or 64 KB. The DS field (sets per way) of the *Config1* register is used to determine the size of the cache. This field can have one of two values. A value of 0x2 indicates a 32 KB cache and a value of 0x3 indicates a 64 KB cache.

```
mfc0  $10, $16, 1       # Read C0_Config1
ext   $12, $10, 13, 3   # Extract DS
li    $14, 2            # Used to test against
```

If the check is true, the code uses the branch delay slot (which is always executed) to set the set iteration value to 256 for a 32 KB cache and then branches ahead to **Dsets_done**. If the check is false, the code assumes that the size of the cache is 64 KB. At this point, the code still sets the iteration value to 256 in the branch delay slot, but then falls through and sets it again to 512 for a 64 KB cache.

```
beq   $14, $12, Dsets_done # if  DS = 2
li    $12, 256          # sets = 256
li    $12, 512          # else sets = 512 Skipped if branch taken
```

Dsets_done:

GPR 14 will be used as an index into the data cache. It is set to a virtual address and then translated to a physical address. Since the address 0x8000_0000 is in kseg0, the CPU will ignore the top bit, so virtual 0x8000_0000 will become physical address 0x0000_0000. Since the cache is physically indexed, the first time through the loop, the cache instruction will write the tag to way 0 index line 0.

The **lui** instruction loads 0x8000 into the upper 16 bits and clears the lower 16 bits of the GPR14 register.

```
lui   $14, 0x8000       # Get a KSeg0 address for cacheops
```

Clearing the tag registers performs two important functions: it sets the Physical Tag address called PTagLo to 0, which ensures the upper physical address bits are zeroed out, and it also clears the valid bit for the set, which ensures that the set is free and may be filled as needed.

The code uses the Move to Coprocessor zero instruction to move the general purpose register zero, which always contains a zero, to the tag register.

```
// clear the lock bit, valid bit, and the LRF bit
mtc0    $0, $28, 2      # Clear C0_DTagLo to invalidate entry
```

The **`Cache`** instruction uses the **`Index Store Tag`** operation on the Level 1 data cache so the op field is coded with a value of 0x9. The first two bits are 2’b01 for the L1 data cache, and the operation code for **`Index Store tag`** is encoded as 3’b010 in bits two, three and four.

next_dcache_tag:

```
cache 0x9, 0($14) # Index Store tag Cache opt
```

The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by breaking down the virtual address argument stored in the base register of the **`Cache`** instruction into several fields.

Bits 14:0 of the Cache Instruction

Way (14–13)	Page Index (12–5)	Byte Index (4–0)

The size of the index field varies according to the size of a cache way. The larger the way, the larger the index. In the table above, the combined byte and page index is 13 bits because each way of the cache is 8K. The way number is always the next two bits following the index.

The code does not explicitly set the way bits. Instead it just increments the virtual address by the cache line size so the next time through the loop the **`Cache`** instruction will initialize the next set in the cache. Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache because it overflows into the way bits.

At this point all the code needs to do is loop maintenance. First decrement the loop counter (12/t4).

```
add   $12, -1           # Decrement set counter
```

Then test it to see if it has gotten to zero and if not branch back to label one.

```
bne   $12, $0, next_dcache_tag # Done yet?
```

The instruction in the branch delay slot, which is always executed, is used to increment the virtual address (14/t6) to the next set in the cache. (11/t3) holds the line size in bytes

```
add   $14, $11          # Increment line address by line size
```

At this point the Dcache initialization is done.

```
done_dcache:

      jr      r31_return_addr
      nop

END (init_dcache)
```

4.3.7 Data Cache CP0 Register Interface

The P6600 core uses the following CP0 registers for data cache operations.

Table 4.6 Data Cache CP0 Register Interface

CP0 Registers	CP0 number
Config1	*16.1*
CacheErr	*27.0*
DTagLo	*28.2*
DDataLo	*28.3*

4.3.7.1 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 P6600 L1 data cache supports 256 sets per way, which is used to configure a 32 KB cache, or 512 sets per way, which is used to configure a 64 KB cache.

The $Config1_{DL}$ field (bits 12:10) indicates the line size for the data cache. The P6600 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 P6600 L1 data cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

For more information, refer to Section 2.2.1.2, "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)".

4.3.7.2 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.

For more information, refer to Section 2.2.5.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.3.7.3 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 four 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.

Note that the P6600 core does not implement the *DTagHi* register.

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

For more information, refer to Section 2.2.5.5, "Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)".

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

In the P6600 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 that the P6600 core does not implement the *DDataHi* register.

For more information, refer to Section 2.2.5.6, "Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)".

4.4 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 some of the arrays (prediction arrays are not accessible through software). 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.

4.4.1 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.

4.4.2 L1 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.

The Index Store Data instruction can optionally update the corresponding precode field in the tag array.The precode bits in the array are updated if the *PCD* bit in the *ErrCtl* register is zero when executing the Index Store Data instruction. The precode value is generated by the hardware automatically if the PCO bit in the *ErrCtl* register is zero. Otherwise, the corresponding precode value (PREC_01/PREC_23/PREC_45/PREC_67) from the *ITagHi* register is used in updating the tag array.

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.

4.4.3 L1 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.

4.4.4 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.

4.4.5 Duplicate Data Cache Tag Array

This array can be tested via the Index Load Tag and Index Store Tag varieties of the CACHE instruction. In order to access the duplicate tags, the WST and SPR bits of ErrCtl should both be set. 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.

4.4.6 L1 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.

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.

4.4.7 L1 Data Cache Way Select Array

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

4.4.8 L1 Data Cache Dirty Bit Array

The testing of this array is also done through Index Load Tag and Index Store Tag CACHE instructions. By setting the *DYT* bit in the *ErrCtl* register, these operations will read and write the dirty array instead of the tag array.

4.5 L2 Cache

The L2 cache (which is part of the Coherence manager) 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 Open Core Protocol (OCP) interface.

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 which can be used to configure it for different operating modes.

4.5.1 L2 Cache General Features

- 7-stage pipeline. (Optional 8th stage[1] for pipelined memory arrays.)
- 40-bit address paths and 256-bit internal data paths
- Associativity: 8-way
- Cache size: 512 KB, 1 MB, 2 MB, 4 MB, 8 MB
- Line Size: 32 bytes (4 doublewords)
- Locking Support: Yes
- Replacement Algorithm: Pseudo LRU for 8-way
- Write policy: Write Back
- Write miss allocation policy: No-Write-Allocate and Write-Allocate
- Error Checking and Correction (ECC): 2-bit error detection and 1-bit error correction covering the tag and data arrays. 1-bit error detection covering the WS array
- Maximum read misses outstanding: 15
- Out-Of-Order processing (OOO): Yes
- Coherency: Non-coherent
- 256-bit or 128-bit OCP SData/MData width on memory-side OCP interface.
- OCP Burst Size on the memory interface: 1 or 2 with 128-bit OCP data width, 1 with 256-bit OCP
- Bypass Mode Support: In bypass mode, all processor requests are routed to the system. This mode is used only for debug purposes and should not be used during normal operation.
- Multi-cycle Data Rams: 0, 1, 2, or 3 stalls can set Data RAM access times to 1, 2, 3, or 4 clocks.
- Multi-cycle Tag Rams: 0, 1, 2, or 3 stalls can set Tag RAM access times to 1, 2, 3, or 4 clocks.
- Multi-cycle Way-Select Rams: 0, 1, 2, or 3 stalls can set the Way-Select RAM access times to 1, 2, 3, or 4 clocks.

---

1. Build time option. The customer must choose this option if they are using pipelined RAM's in the wrappers instead of standard RAM cells (that are not pipelined in this way).

- Endianness: Independent of endianness

Table 4.7 L2 Cache Attributes

Attribute	With Parity
Size	512 KB, 1 MB, 2 MB, 4 MB, or 8 MB
Line Size	32-byte
Number of Cache Sets	2048, 4096, 8192, 16384 of 32768
Associativity	8 way

In the table above, the associativity of the L2 cache is fixed at 8 ways. As a result, changes to the number of sets per way and the line size determine the overall size of the L2 cache. Table 4.8 shows the list of possible L2 cache configurations.

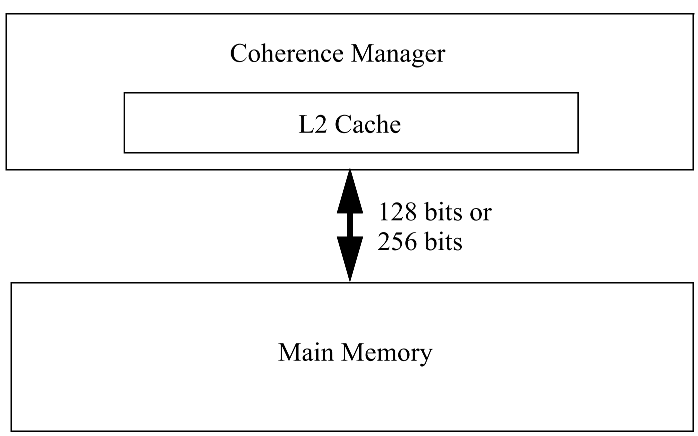
Table 4.8 Valid Cache Configurations

Line Size	Sets per Way	Number of Ways	L2 Cache Size
32 bytes	2048	8	512 KBytes
32 bytes	4096	8	1 MByte
32 bytes	8192	8	2 MByte
32 bytes	16384	8	4 MByte
32 bytes	32768	8	8 MByte

4.5.2 OCP Interface

In the P6600 core, the L2 cache is integrated into the CM2. This integration improves performance by eliminating the OCP interface that originally connected the L2 cache to the CM, or the L2 cache to the CPU depending on configuration. The OCP interface between the CM2 and the memory is programmable for widths of either 128-bit or 256-bit and has a fixed 64-byte line size. This is shown in Figure 4.4.

Figure 4.4 .OCP Interface Between CM2 and Memory



4.5.3 L2 Replacement Policy

The P6600 core uses a pseudo-LRU replacement algorithm. The system memory configuration does not affect the replacement policy.

4.5.4 L2 Allocation Policy

The L2 cache controller always allocates cacheable reads issued by a core. A cacheable write (such as an L1 write-back) issued by a core is never allocated in the L2 cache. Cacheable reads and writes from the IOCU may or may not be allocated into the L2, depending upon signals driven with the request by the IO Subsystem.

4.5.5 Write-Through vs. Write-Back

Write-through and write-back operations are both supported. The L2 decodes *MReqInfo[2:0]* fields and determines which way to handle the write data.

When a write hits in the L2 cache, the data is written into the L2 cache, and alsosent to the main memory when it was write-through type (*MReqInfo[2:0]* = 0).

When a write misses, the no-write-allocation policy is employed in most cases. That is, the write data is forwarded to the main memory without updating the L2 cache contents. However, for the write-back type write with full line data, usually resulting from the L1 D-cache eviction, the L2 supports write-allocate on miss as well as the normal no-allocate policy. This is controlled by the value on *MReqInfo[4]* that is set by the OCP requester. Please refer to the Section 4.5.4 "L2 Allocation Policy" for more details.

4.5.6 Cacheable vs. Uncacheable vs. Uncached Accelerated

The L2 cache supports cacheable and uncacheable accesses. 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.

4.5.7 Cache Aliases

The L2 cache is physically addressed and physically tagged. It is not subject to virtual aliasing.

4.5.8 Performance Counters

The L2 tracks and reports to core the number of the following events.

- the number of cached accesses
- the number of misses
- the number of write backs
- the amount of cycles the L2 is held due to misses
- the number of single bit errors that were corrected

- L2 pipeline utilization — Counts the number of starts into the TA stage of the L2 pipeline
- L2 hit qualifier — Counts different types of L2 cache hits and misses, crossed with the instruction being requested

4.5.9 Sleep Modes

The L2 cache contains two basic sleep modes:

- Instruction controlled sleep mode using the WAIT instruction
- Internal dynamic sleep mode

4.5.9.1 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_L2_Sleep* input to the L2. The L2 then enters a low-power state and asserts the *L2_Sleep* output once all outstanding bus activity has completed. Most clocks in the L2 will be stopped, but a handful of flops will remain active to sense the wake up call from the processor, which is the deassertion of *SI_L2_Sleep*.

Power is reduced since 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_L2_Sleep* asserted and there is no activity in the L2, the L2 will assert *L2_Sleep* two *cm_clks* later. Any activity in the L2 will delay the start of *L2_Sleep* assertion.

When *SI_L2_Sleep* is deasserted, the L2 will deassert *L2_Sleep* and assert *PB_SCmdAccept* two clocks later. If there is a valid *PB_MCmd* waiting at the L2 pins at the *cm_clk*, then the following *cm_clk* will have a coincident internal *l2_clk* edge (clocks are now enabled) and the command that was accepted is launched into the pipeline as indicated by *inst_ta*. The following clock after that will have an *l2_tram_clk* that initiates the tag ram access for that command. Thus, there is a four *cm_clk* latency from *SI_L2_Sleep* deassertion to the start of a tag ram access.

4.5.9.2 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 will turn off the l2_clk signal after some small delay. Only data of value in the CMOS SRAM's retains state.

Beside the WAIT instruction induced sleep mode, the L2 is also equipped with the 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. While the most part of the L2 cache can be turned off, the minimum required logic on the core-side OCP interface remain active. Thus, the L2 cache can accepta new OCP request from core at any time, and this will wake up the whole L2 cache controller.

4.5.10 Bypass Mode

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

Bypass mode is a test/bringup feature that causes the L2 cache to forward all requests received from either the core or the Coherency Manager to the OCP system interface to main memory. 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 and SYNCs.

4.5.11 Reduced L2 Hit Latency

The CM2 integrates the CM and L2 cache into a single, more tightly-coupled component, providing reduced L2 hit latency. Table 4.9 provides the latencies for a read request from a P6600 core to an idle CM2.

- The system is idle prior to this request
- The L2 cache is configured with no L2 Tag RAM or Data RAM stalls
- The L2 is configured with ECC
- L2-to-memory clock ratio is 1:1
- The L2 is configured with non-pipelined Data RAM’s

Table 4.9 CM2 Read Latencies (in core clock cycles)

Request CCA	Cache Hit/Miss	CM2
Coherent (CWB, CWBE)	L1 Miss/L2 Hit	11
	L1 Hit	15
	L1 Miss/L2 Miss	14
Cached/Non-coherent (WB)	L2 Hit	11
	L2 Miss	15
Uncached (UC)	---	12
GCR Read	---	8
Coherent Upgrade	Intervention Response of SHARED	11

4.5.12 L2-only Sync

The CM2 adds the ability to issue a barrier-sync to the L2 without executing a SYNC instruction, thus reducing the latency incurred for the sync. The L2-only sync provides a mechanism to guarantee that a uncached request does not pass previous cached requests in the L2 pipeline. For example, the L2-only SYNC can be used between a L2 HitWB cacheop and a subsequent uncached write to ensure that the uncached write does not pass the writeback from the L2. The following sequence could be used to flush a cache line from the L1 and L2 and then provide a sentinel to a consuming device as follows:

```
L1HitWB (flush L1 data to L2. will be globalized to all cores if coherent)
L2HitWB (flush L2 data to memory. CM2 ensures this does not pass the L1 HitWB)
L2-only SYNC (ensures subsequent uncached write does not pass L2HitWB)
uncached Store (sentinel to consuming device)
consuming device receives sentinel and reads memory
```

The L2-only sync is achieved by executing an Uncached store to an address that maps to the address region specified by the CM2's GCR_L2_ONLY_SYNC_BASE register. When the L2-only SYNC write is ready to be issued to the L2 pipeline the following actions occur:

1) Stop issuing new L2 requests until the L2 pipeline is empty and eviction queue is empty

2) The L2-only sync request is dropped and subsequent L2 requests continue.

Notice that the the L2-only sync does not ensure any ordering in the coherent portion of the CM2.

The CM_L2_ONLY_SYNC_EN in bit 0 of the GCR_L2_ONLY_SYNC_BASE register must be set to a 1 for this feature to be enabled. The address match is performed on a 4KB boundary. An uncached write request address [31:12] that matches the address [31:12] in the GCR_L2_ONLY_SYNC_BASE will cause the CM2 to treat the uncached write request as an L2 only Sync.

The GCR_L2_ONLY_SYNC_BASE register is programmed through the Global Control Block Register Map located at offset 0x0070.

4.5.13 L2 Cache Initialization

The L2 cache controller contains minimal hardware initialization logic. It normally relies on software to fully initialize the L2 arrays. The registers used to support cache initialization are described in Section 4.5.14, "L2 Cache CP0 Interface". For additional information, refer to the *CP0 Registers* chapter of this manual.

The L1 data cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user's, a sample boot code is shown in the following subsection.

4.5.13.1 init_l2u Cache Initialization Routine

The following assembly provides an example initialization routine for the L2 cache.

```
LEAF(init_l2u)
      # Use CCA Override to allow cached execution of L2 init.
      # Check for CCA_Override_Enable by writing a one.
      lw r4_temp_data, 0x0008(r22_gcr_addr) # Read GCR_BASE register
      li r7_temp_mark, 0x50 # CM_DEFAULT_TARGET Memory
      # CCA Override Uncached enabled
      ins r4_temp_data, r7_temp_mark, 0, 8
      sw r4_temp_data, 0x0008(r22_gcr_addr)
      lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
      ext r4_temp_data, r4_temp_data, 4, 1 # Extract CCA_Override_Enable
      bnez r4_temp_data, done_l2 # Skip if CCA Override is implemented.
      nop
      b init_l2u
      nop
END(init_l2u)
```

4.5.13.2 init_l2c Cache Initialization Routine

The code in this function will be called from start.S after the L1 caches have been initialized. It will check to see if the core implements CCA Override. If it does, it will call the code to initialize the L2 cache.

```
LEAF(init_l2c)

      # Skip cached execution if CCA Override is not implemented.
      # If CCA override is not implemented the L2 cache would have already
      # been initialized when init_l2u was called.

      lw r4_temp_data, 0x0008(r22_gcr_addr) # Read GCR_BASE
      bnez r16_core_num, done_l2 # Only done from core 0.
      ext r4_temp_data, r4_temp_data, 4, 1 # CCA_Override_Enable
      beqz r4_temp_data, done_l2
      nop

END(init_l2c)
```

4.5.13.3 init_L2u Initialization Routine Details

This section provides a detailed description of each line of code in the init_l2u initialization routine.

The L2 cache is a system resource used by all cores in the system. Initialization of the L2 cache is done only by Core 0, because it only needs to be done once. The initialization of the L2 cache can be time consuming depending on its size. For example, a 256 KByte cache initializes quicker than an 8 MB cache.

The L2 cache initialization code executes faster if it is being run out of the instruction cache, so ideally the L2 initialization should be done after the L1 instruction cache in core 0 has been initialized. The instruction cache is a per-core resource and not initialized in the system initialization section of the code. Therefore, to be efficient and run the L2 cache initialization code out of the I-cache, the boot code tries to put off L2 cache initialization until the core 0 resources have been initialized. This can only be done if the L2 cache can be disabled before other cores are released to run this boot code. Otherwise there is a danger that other cores will use the L2 cache before it has been initialized by core 0.

The CCA override feature controls the cache attributes for the L2 cache. It allows for the disabling of the L2 cache by enabling the CCA override and setting the CCA to uncached. The CCA override works along with the L2 cache implementation.

The init_l2u function tries to enable the CCA override and set the L2 cache to uncached in the GCR_BASE register, thus disabling it. On systems that do not support CCA override, writes to the CCA override field have no effect, and reading back the GCR_BASE register will not show the CCA override being set.

The code reads the GCR Base register.

```
      lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
```

The next 3 lines of code are used to enable CCA Override and set the L2 cache CCA to uncached.

```
      li r7_temp_mark, 0x50 # CM_DEFAULT_TARGET Memory
      # CCA Override Uncached enabled
```

```
ins r4_temp_data, r7_temp_mark, 0, 8
sw r4_temp_data, 0x0008(r22_gcr_addr)
```

Now the code reads back the GCR_BASE register. If the CCA override bit is set, it means the code above worked, and the L2 cache is set to uncached. In this case, the code skips the initialization for now. The routine will be recalled later once the code is executing out of the L1 instruction cache. If not, the code branches to the init_l2 function, which initializes the L2 cache.

```
lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
ext r4_temp_data, r4_temp_data, 4, 1 # CCA_Override_Enable
bnez r4_temp_data, done_l23 # Skip if CCA Override is implemented.
nop
b init_l2
nop

END(init_l2u)
```

4.5.13.4 init_L2c Initialization Routine Details

This section provides a detailed description of each line of code in the init_l2c initialization routine. The code in this function is called from the start.S function after the L1 caches have been initialized. It checks to see if the core implements CCA Override. If it does, it calls the code to initialize the L2 cache.

In Section 4.5.13.3 the code also checks to see if CCA override was implemented, If it was not, then it initialized the L2 cache while the code was executing in uncached mode, so there is no need to do it again here.

```
LEAF(init_l2c)

      # Skip cached execution if CCA Override is not implemented.
      # If CCA override is not implemented the L2 cache
      # would have already been initialized when init_l2u was called.

      lw r4_temp_data, 0x0008(r22_gcr_addr)     # GCR_BASE
      bnez r16_core_num, done_l2                # Only done from core 0
      ext r4_temp_data, r4_temp_data, 4, 1      # CCA_Override_Enable
      beqz r4_temp_data, done_l23 nop

END(init_l2c)
```

4.5.14 L2 Cache CP0 Interface

The P6600 core uses different CP0 registers for L2 cache operations.

Table 4.10 L2 Cache CP0 Register Interface

CP0 Registers	CP0 number
Config2	*16.2*
ErrCtl	*26.0*
CacheErr	*27.0*
L23TagLo	*28.4*
L23DataLo	*28.5*
L23DataHi	*29.5*

This section describes the base processor core CP0 registers that support the L2 cache. A complete description and bit assignments for each register listed is described in Chapter 2, CP0 Registers.

4.5.14.1 Config2 Register (CP0 register 16, Select 2)

Asserting $Config2_{L2B}$ (bit 12) enables the bypass-mode of the L2 cache. This bit is reflected on the *L2_Bypass* output from the core. When L2 goes into bypass-mode, L2 responds by asserting *L2_Bypassed* output, and the value or *L2_Bypassed* is returned when $Config2_{L2B}$ is read by software. Thus, reading this $Config2_{L2B}$ bit does not read back what was written: it reflects the value of a signal sent back from the L2. The feedback signal, *L2_Bypassed*, will reflect the previously written value with some implementation and clock ratio dependent delay.

Changing the value of $Config2_{L2B}$ field in the middle of the normal operation may cause an unwanted loss of an OCP transaction in the L2 cache. For the safe transition into the L2 bypass-mode, an externalized SYNC before the MTC0 $Config2_{L2B}$ is necessary to make sure all the pending transactions in L2 are completed. And, these instructions should run from the uncached space. It might be also a good idea to check if L2 is really in bypass-mode by reading the $Config2_{L2B}$ field before moving onto the next instructions.

The $Config2_{SS}$ field (bits 11:8) indicates the number of sets per way in the data cache. The P6600 L2 cache supports from 512 up to 32768 sets per way, which is used to configure cache sizes from 256 KBytes to 8 MBytes.

The $Config2_{SL}$ field (bits 7:4) indicates the line size for the L2 cache. The P6600 L2 cache can be configured for a 32-byte or 64 byte line size.

The $Config2_{SA}$ field (bits 3:0) indicates the set associativity for the L2 cache. The P6600 L2 cache is fixed at 8-way set associative as indicated by a default value of 4 for this field.

For more information, refer to Section 2.2.1.3, "Device Configuration 2 — Config2 (CP0 Register 16, Select 2)".

4.5.14.2 Error Control Register (CP0 register 26, Select 0)

ErrorControl.L2P (bit 23) is used to enable L2 ECC checking and correction. This bit is read-only if the L2 has not been built with ECC/Parity support. Specific parity support is enabled using both L2P and ErrorControl.PE (bit 31) as described in Table 4.11. L2P is also reflected on the *L2_ECCEnable* output from the core.

These encodings were chosen such that legacy code which is unaware of L2P, will by default enable L2 ECC logic when it enables L1 parity. For more information, refer to Section 2.2.5.10, "ErrCtl (CP0 Register 26, Select 0)"

Table 4.11 L2_ECC_Enable

PE	L2P	L2_ECCEnable
1	0	1
1	1	0
0	0	0
0	1	1

4.5.14.3 Cache Error Register (CP0 register 27, Select 0)

When the L2 detects an uncorrectable error, CacheError.EC is set, identifying the exception as an L2 error. The Cache Error register stores information such as the cache way where the error was detected, the cache index of the double word in which the error was detected, the cache level at which the error was detected, if the tag RAM was involved, etc.

For more information, refer to Section 2.2.5.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.5.14.4 L23TagLo Register (CP0 register 28, Select 4)

The L23TagLo register contains the contents of the L2 tag array at the location accessed by the L2 Index Load Tag cache-op. It is also used as the source register for the L2 Index Store Tag cache-op.

For more information, refer to Section 2.2.5.7, "Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)".

4.5.14.5 L23DataHi Register(CP0 register 29, Select 5) / L23DataLo Register(CP0 register 28, Select 5)

For the L2 Index Load Tag cache-op, L23DataHi and L23DataLo hold the contents of the doubleword from the L2 data array at the indexed location. (L23DataHi holds the most-significant word and L23DataLo holds the least-significant word). For the L2 Index Load WS cache-op, L23DataHi and L23DataLo each hold the ECC parity of the doubleword from the L2 data array at the indexed location.

These registers are also used for the source data for the Index Store Data cache-op. Finally, L23DataLo is used as the data source for the ECC to be written by the Index Store ECC cache-ops. For more details on the data returned by the L2 on a Index Load Tag/Data cache-op, please refer to Section 4.6 “The CACHE Instruction”.

For more information on the L23DataLo register, refer to Section 2.2.5.8, "Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)". For more information on the L23DataHi register, refer to Section 2.2.5.9, "Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)".

4.5.15 L2 Cache Operations

Cache-ops are used for control operations such as initialization, invalidation, eviction, etc. A brief description of the cache-ops implemented by the L2 are given below:

Index Writeback Invalidate: If the state of the cache line at the specified index is valid and dirty, the line is written back to the memory address specified by the cache tag. After that operation is completed, the state of the cache line is set to invalid. If the line is valid but not dirty, the state of the line is set to invalid.

Index Load Tag: The tag, valid, lock, dirty, parity and LRU bits for the cache line at the specified index are read. The doubleword indexed in the data RAM is also read.

Index Load WS: The LRU, dirty, and dirty parity bits for the cache line at the specified index are read. ECC for the doubleword indexed in the data RAM is also read.

Hit Invalidate: If the cache contains the specified address, the state of that cache line is set to invalid.

Hit Writeback Inv: If the cache contains the specified address and it is valid and dirty, the contents of that line are written back to main memory. After that operation is completed, the state of the cache line is set to invalid. If the line is valid but not dirty, the state of the line is set to invalid.

Hit Writeback: If the cache contains the specified address and it is valid and dirty, the contents of that line are written back to main memory. After the operation is completed, the state of the line is left valid, but the dirty state is cleared.

Index Store Tag: Write the tag for the cache line at the specified index.

Index Store WS: Write the WS array for the cache line at the specified index.

Fetch And Lock: If the cache contains the specified address, lock the line. If the cache does not contain the specified address, refill the line from main memory and then lock the line.

Index Store Data: Write the data and ECC for the cache line at the specified index. Proper ECC is generated for the written data and written into the ECC field.

Index Store ECC: Write the ECC for the cache line at the specified index.

Most CP0 instructions are used rarely, in code which is not timing-critical. But an OS which has to manage caches around I/O operations or otherwise may have to sit in a tight loop issuing hundreds of `cache` operations at a time, so performance can be important.

4.5.15.1 Bus Transaction Equivalence

When the base processor executes an L2 CACHE instruction, the operands and as well as data to be written to CP0 registers is transferred to and from L2. Index Load Tag and Index Load WS generate burst read transactions. All other L2 cache-ops generate single write transactions.

For 64 byte line configurations, bit 5 (the LSB of the Index field) is the selector to which 32 byte half of the 64 byte line is targeted (essentially it becomes an additional DW bit). For *tag* and *ws* type cache-ops, this bit is disregarded and cache-ops with either value of bit 5 impact the exact same tag or ws entry. For data type cache-ops, bit 5 selects which half of the 64 byte cache line is being accessed.

Figure 4.5 Index Encoding for PB_MAddr (1MB, 8-way)

31 ... 23	22 ... 20	19 ... 5	4 ... 3	2 ... 0
Unused	Way	Index	DW	Unused

4.5.15.2 Details of Cache-ops

Table 4.12 indicates the operation and behavior of the L2 cache for each cache-op.

Table 4.12 Cache-ops

Cache-op	Effective Address Operand Type	Operation
Index WB inv/ Indx Inv (OPCODE: 0)	INDEX	• If the state of the cache line at the specified index is valid and dirty, the line is written back to the memory address specified by the cache tag. After that operation is completed, the state of the cache line is set to invalid. • If the line is valid but not dirty, the state of the line is set to invalid • The LRU bits are updated to Least-recently-used. • The dirty bits are updated to clean for that way.
Index Load Tag (OPCODE: 1) ErrCtl.WST = 0	INDEX	• The tag, valid, lock, and parity fields from the tag array for the cache line at the specified index are written into L23TagLo. Furthermore, the dirty bit from the WS array corresponding to the specified index is also written into L23TagLo. (First beat of return data) • For the first beat of return data, the two halves of the 64-bit data bus are identical. • The indexed doubleword is written into {L23DataHi, L23DataLo}. (2nd beat of return data) • ErrCtl.PO is treated as a don't care • The LRU bits are unchanged
Index Load WS (OPCODE: 1) ErrCtl.WST = 1	INDEX	• The dirty, dirty parity, and LRU fields from the WS array for the cache line at the specified index are written into L23TagLo. (First beat of return data) • For the first beat of return data, the two halves of the 64-bit data bus are identical. • The WS data at the indexed location is written into L23TagLo. (First beat of return data) • The indexed doubleword's ECC is written into {L23DataHi, L23DataLo}. (2nd beat of return data) • ErrCtl.PO is treated as a don't care • The LRU bits are unchanged • Data RAM: • The DW ECC to be read in the line is determined by *PB_MAddr[4:3]*
Index Store Tag (OPCODE: 2) ErrCtl.WST = 0	INDEX	• The tag, valid, and lock fields in the Tag array at the indexed location are written from L23TagLo. • If ErrCtl.PO==1, the parity and total parity fields in the Tag array at the indexed location are written from L23TagLo. • If ErrCtl.PO==0, the parity and total parity fields in the Tag array at the indexed location are written with hardware generated values. • If valid==1, the LRU bits in the WS array are updated to make the indexed way most-recently-used. If valid==0, the LRU bits are updated with least-recently-used. • If valid==1, the dirty bit in the WS array at the indexed location is written from L23TagLo. • If valid==0, the dirty bit in the WS array at the indexed location is cleared. • The dirty parity bit in the WS array at the indexed location is written with the correct hardware generated values.
Index Store WS (OPCODE: 2) ErrCtl.WST = 1	INDEX	• The dirty and LRU fields for all 8 ways of the WS array at the indexed location are written from L23TagLo • If ErrCtl.PO==1, the dirty parity fields for all 8 ways of the WS array at the indexed location are written from L23TagLo • If ErrCtl.PO==0, the dirty parity fields for all 8 ways of the WS array at the indexed location are written with hardware generated values

Table 4.12 Cache-ops *(continued)*

Cache-op	Effective Address Operand Type	Operation
Index Store Data (OPCODE: 3) ErrCtl.WST = 0	INDEX	• The doubleword in the data array at the indexed location and doubleword offset is written from {L23DataHi, L23DataLo} regardless of the PB_MDataByteEn value. • The Parity/ECC field in the data array at the indexed location and doubleword offset is written with a hardware generated value. • The LRU bits in the WS array are updated to make the indexed way most-recently-used.
Index Store ECC (OPCODE: 3) ErrCtl.WST = 1	INDEX	• The Parity/ECC field in the data array at the indexed location and doubleword offset is written from L23DataLo[7:0]. • The LRU bits in the WS array are updated to make the indexed way most-recently-used.
HIT Inv (OPCODE: 4)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, it is invalidated and the dirty bit is cleared. • If any arrays are written, the appropriate parity fields are updated by hardware.
HIT WB Inv (OPCODE: 5)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, and it is dirty, the line is written back to main memory. It is then invalidated and the dirty bit is cleared. • If the address hits in L2, and it is clean, it is invalidated. • If any arrays are written, the appropriate parity fields are updated by hardware.
HIT WB (OPCODE: 6)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, and it is dirty, the line is written back to main memory and the dirty bit is cleared. • If the address hits in L2, and it is clean, nothing happens. • If any arrays are written, the appropriate parity fields are updated by hardware.
Fetch and Lock (OPCODE: 7)	ADDRESS	• If the address is not contained in L2, the line is refilled. The refilled line is 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 line is locked and the operation retires. The LRU bits or the dirty bits are not affected.

4.5.15.3 Sync in L2

A Sync operation can be used to guarantee ordering of transactions. The L2 ensures that all transactions preceding a Sync request will be ordered in front of transactions received after the Sync request. Within the L2 only requests are ordered, not responses, i.e., there is no guarantee of the ordering between a read response vs. the Sync.

One example of the use of a Sync involves cache operations. Normally, the L2 does not guarantee the ordering between a cache operation, such as a Hit-Writeback-Invalidate, vs. an subsequent uncached request. If the software wants to ensure that any writes on the system interface due to the Hit-Writeback-Invalidate will be ordered in front of a subsequent uncached write, then a Sync must be issued between the cache operation and uncached write. Note that in order for a core to externalize a Sync request, $Config7._{ES}$ bit must be set before the sync instruction.

The L2 issues a response to a Sync after all 3 of the following have completed:

- All previous requests have cleared the L2 pipeline
- The L2 has issued all requests to the system interface that are required by previous transactions, such as uncached requests, cache operations, cache misses, evictions, or previous Syncs.

- If the downstream system can take a sync OCP transaction (*L2_SyncTxEn=1*), it will externalize the sync transaction to the system once the above criteria has been satisfied. When the Sync response is received from the system interface, the L2 will return a Sync response to the processor interface.

4.5.15.4 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.

4.5.16 L2 Cache Error Management

This section describes parity and bus error support for the L2 cache.

4.5.16.1 Parity Support

If Parity support is selected at build time, and this support is enabled via software by setting the $ErrCtl_{PE}$ bit in the Error Control register (CP0 register 26, Select 0), then the tag and the data arrays are protected with single-error correction logic.

The Way Select RAM is protected with single-error detection logic. Correctable errors are not reported to the processor, but uncorrectable errors are reported to the processor. If Parity support is either not selected at build time or disabled, then no errors are detected on any of the cache arrays.

To perform a single detection the parity bits are placed at 2^n locations among the data bits. The bits at different locations are then grouped together. The grouping is done by analyzing the binary weights of the particular location.

For example, to protect 8 data bits, 4 parity bits are needed which will be placed as below:

Table 4.13 Parity Bit Distribution

Bit Location	12	11	10	9	8	7	6	5	4	3	2	1
Parity and data bits	d7	d6	d5	d4	p3	d3	d2	d1	p2	d0	p1	p0

Note that Bit location 0 does not exist.

The binary weight of bit location 3 is 2^0 and 2^1, which is derived from its binary value 0011b. Therefore, bit location 3 falls in group g0 and g1. Similarly, Bit location 11 falls into groups g0, g1 and g3.

Parity bit p0 will belong to g0 and its value will is generated such that g0 will have an even parity. Similarly all other parity bits are generated such that their respective group ends up in even parity.

This sharing of binary weights across groups enables the L2 to determine precisely which data or parity bit was in error. That is achieved by recreating the parity bits from the data read from the memory and XORing it with the parity bits read from the memory. The XORed value, or the syndrome, points to the bit in error. Once this error is detected the L2 corrects it. A value of zero on the syndrome indicates that there was no error in the parity and data bits.

To achieve double bit error detection an even parity is generated across the parity and data bits, which is termed as the total parity bit. The total parity bit will be flipped in case of a single bit error, whereas for a double bit error it will remain the same. The syndrome along with the total parity bit is then used to detect a double bit error.

The WSRAM's dirty bits are protected, whereas the LRU bits are not. For each dirty bit there is one more bit added called the dirty parity bit. The value of the dirty parity bit enforces even parity protection.

4.5.16.2 Tag, Data, and WS Array Format

Logical Tag Array Format

The width of the tag in an 8 way 128 MB cache is 18 bits per way. The data array format is as shown in Figure 4.14.

Table 4.14 Logical Tag Array Format for a 8 Way 128 MB Cache

Bit position	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Content	TP	L	V	d17	d16	d15	d14	d13	d12	d11	p4	d10	d9	d8	d7	d6	d5	d4	p3	d3	d2	d1	p2	d0	p1	p0

Where, d0-d17 : Tag
V : Valid bit
L : Lock bit
p0-p4 : parity bits
TP : Total parity bit

For larger caches, the width of the tag reduces. In that case, the upper data bits are ignored from the calculation as appropriate.

Logical Data Array Format

The data array format is as shown in Figure 4.15.

Table 4.15 Logical Data Array Format

Bit position	72	71..65	64	63:33	32	31:17	16	15..9	8	7..5	4	3	2	1
Content	TP	[63:57]	p6	[56:26]	p5	[25:11]	p4	[10:4]	p3	[3:1]	p2	[0]	p1	p0

4.5.16.3 Cache Parity Error Handling

The three types of memory arrays in the L2 have an option for parity. If selected, this option provides single bit correction and double bit detection of the tag rams and data rams.

- The Tag RAM coverage is for each way.
- The Data RAM coverage is for each way and each double-word in each way.
- The Way Select RAM has parity for each dirty bit. A correctable bit failure is corrected and no notification of this event is present at the L2 pins.

4.5.16.4 Multiple Uncorrectable Errors

This error is reported when more than one uncorrectable error is being reported on the same L2 clock cycle. Since double-bit Tag RAM errors, double-bit Data RAM error, and parity bit errors in the Way Select RAM are each reported in different L2 pipeline stages, this assertion indicates that different requests have encountered uncorrectable requests. In other words, if a single request suffers all three uncorrectable errors, the error will be reported three times.

4.5.16.5 Bus Error Handling

Bus errors are never originated by the L2. However, bus errors may be received from the system on an OCP read from the L2 to the system. The error is indicated when the read-data is returned back to the L2. The L2 propagates the bus error when returning data to the processor or CM2.

If a bus error is received on a 64-byte burst read to the system, the L2 signals the bus error for the processor read that originated the request. If the L2 receives a subsequent read to the same 64-byte cache line before all the data has been received from memory for the previous request, the new request also receives a bus error response.

In general, a bus error reported in a system response due to a processor/CM request is considered to be reporting the entire cache line as having a bus error. However, if the original request is satisfied before the L2 detects the system bus error, then the response to the processor/CM will not have a bus error.

There is no capability for signalling bus errors on writes.

4.6 The CACHE Instruction

The L1 instruction, L1 data, and L2 caches in the P6600 core each support the CACHE instruction, which allows users to manipulate the contents of the Data and Tag arrays, including the locking of individual cache lines. The behavior of the CACHE instruction is identical for both the L1 instruction and data caches.

4.6.1 Decoding the Type of Cache Operation

The type of cache operation performed is encoded using a combination of the 5-bit *op* field of the CACHE instruction, and selected bits from the *ErrCtl* register (CP0 Register 26, Select 0). In addition to performing operations on the caches themselves, there are other CACHE operations that are performed on internal memories such as the way selection RAM and the Dirty Bit RAM. The *ErrCtl* bits determine the type internal memory where the CACHE operation will be performed.

The selected bits of the ErrCtl register used to determine the type of CACHE operation are as follows:

- Bit 29, *WST*: If this bit is set, execution of a `cache IndexLoadTag` or `cache IndexStoreTag` instruction reads or writes the cache's internal *way-selection RAM* instead of the cache tags.
- Bit 21, *DYT*: Setting this bit allows `cache` load/store data operations to work on the "dirty array" associated with the L1 data cache.

4.6.2 CACHE Instruction Opcodes

Refer to the implementation-specific CACHE instruction at the back of this manual for a list of CACHE instruction opcodes.

4.6.3 Way Selection RAM Encoding

The CACHE Index Load Tag and Index Store Tag instructions can be used to read and write the Way Select (WS) RAM by setting the *WST* bit in the *ErrCtl* register. Note that when the *WST* bit is zero, the CACHE index instruction accesses the cache Tag array.

Not all values of the WS field are valid for defining the order in which the ways are selected. This is only an issue, however, if the WS RAM is written after the initialization (invalidation) of the Tag array. Valid WS field encodings for way selection order is shown in Table 4.16.

Table 4.16 Way Selection Encoding, 4 Ways

Selection Order[1]	WS[5:0]	Selection Order	WS[5:0]
0123	000000	2013	100010
0132	000001	2031	110010
0213	000010	2103	100110
0231	010010	2130	101110
0312	010001	2301	111010
0321	010011	2310	111110
1023	000100	3012	011001
1032	000101	3021	011011
1203	100100	3102	011101

Table 4.16 Way Selection Encoding, 4 Ways *(continued)*

Selection Order[1]	WS[5:0]	Selection Order	WS[5:0]
1230	101100	3120	111101
1302	001101	3201	111011
1320	101101	3210	111111

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

Chapter 5

Exceptions and Interrupts

The P6600 core receives exceptions from a number of sources, including arithmetic overflows, misses in the translation lookaside buffer (TLB), I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended and the processor enters kernel mode, disables interrupts, loads the *Exception Program Counter* (*EPC*) register with the location where execution can restart after the exception has been serviced, and forces execution of a software exception handler located at a specific address.

The software exception handler saves 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 it can be restored when the exception has been serviced.

Exceptions may be precise or imprecise. Precise exceptions are those for which the *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot (as indicated by the *BD* bit in the *Cause* register), the address of the branch instruction immediately preceding the delay slot. Imprecise exceptions, on the other hand, are those for which no return address can be identified. Bus error exceptions and CP2 exceptions are examples of imprecise exceptions.

This chapter contains the following sections:

5.1 Exception Conditions

When an 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.

When the exception condition is detected on an instruction fetch, the CPU aborts that instruction and all instructions that follow. When the instruction graduates, the exception flag causes it to write various CP0 registers with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clear the exception bits of earlier pipeline stages.

For most types of exceptions, this implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, 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. An instruction taking an exception may itself be aborted by an instruction further down the pipeline that takes an exception in a later cycle.

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

5.2 TLB Read Inhibit and Execute Inhibit Exceptions

The P6600 core supports the following new types of exceptions listed below:

- TLB Execute-Inhibit
- TLB Read-Inhibit

The *TLB Execute Inhibit* exception (TLBXI) is taken when there is a TLB hit during an instruction fetch, the XI bit of the entry is set, the Valid (V) bit is set, and the $PageGrain_{EIC}$ bit is set. If the $PageGrain_{EIC}$ bit is cleared, a *TLBL* exception is taken. This type of exception is used by the operating system to prevent execute accesses to a particular page. Refer to Section 5.8.13 "TLB Execute-Inhibit Exception (TLBXI)" for more information.

The *TLB Read Inhibit* exception (TLBRI) is taken when there is a TLB hit during a read operation, the RI bit of the entry is set, the Valid (V) bit is set, and the $PageGrain_{EIC}$ bit is set. If the $PageGrain_{EIC}$ bit is cleared, a *TLBL* exception is taken. This type of exception is used by the operating system to prevent read accesses from a particular page. Refer to Section 5.8.14 "TLB Read-Inhibit Exception (TLBRI)" for more information.

5.3 FTLB Parity Exception

An *FTLB Parity* exception is taken whenever a parity error is detected on an FTLB read. The error can occur in either the FTLB Tag RAM or FTLB Data RAM. The FTLB parity exception is taken only when bit 31 of the CP0 *Error Control* register ($ErrCtl._{PE}$) is set. If this bit is cleared, FTLB parity errors are ignored. Refer to Section 5.8.15 "FTLB Parity Exception" for more information.

5.4 Exception Priority

Table 5.1 contains a list and a brief description of all exception conditions, The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is taken.

Table 5.1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_Reset signal.

Table 5.1 Priority of Exceptions *(continued)*

Exception	Description
DSS	EJTAG Debug Single Step. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external *EJ_DINT* input, or by setting the *EjtagBrk* bit in the *ECR* register.
DDBLImpr/DDBSImpr	Debug Data Break Load/Store. Imprecise.
NMI	Asserting edge of *SI_NMI* signal.
FTLBPAR	FTLB instruction fetch parity error.
Machine Check	TLB write that conflicts with an existing entry.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
Debug Instruction Breakpoint	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
XTLBL - Instruction	Fetch XTLB miss. Fetch XTLB hit to page with V=0
TLBL - Instruction	Fetch TLB miss. Fetch TLB hit to page with V=0
TLBXI	TLB Execute Inhibit. Occurs when there is an execute access from a page table whose XI bit is set.
I-cache Error	Parity error on I-cache instruction fetch.
IBE	From Instruction Fetch Unit (IFU) instruction cache ops. Indicates a bus error on an instruction fetch.
D-cache Error	Data cache parity error. Imprecise.
L2-cache Error	L2 cache parity error. Imprecise.
DBE	Load or store bus error. Imprecise.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys (Execution exception)	Execution of SYSCALL instruction. Note that all of the execution exceptions have the same priority.
Bp (Execution exception)	Execution of BREAK instruction. Note that all of the execution exceptions have the same priority.
CpU (Execution exception)	Execution of a coprocessor instruction for a coprocessor that is not enabled. Note that all of the execution exceptions have the same priority.
CEU (Execution exception)	Execution of a CorExtend instruction modifying local state when CorExtend is not enabled. Note that all of the execution exceptions have the same priority.
RI (Execution exception)	Execution of a Reserved Instruction. Note that all of the execution exceptions have the same priority.
FPE (Execution exception)	Floating Point exception. Note that all of the execution exceptions have the same priority.
C2E (Execution exception)	Coprocessor 2 unusable exception. Note that all of the execution exceptions have the same priority.
ISI (Execution exception)	Implementation specific Coprocessor 2 exception. Note that all of the execution exceptions have the same priority.
Ov (Execution exception)	Execution of an arithmetic instruction that overflowed. Note that all of the execution exceptions have the same priority.
Tr (Execution exception)	Execution of a trap (when trap condition is true). Note that all of the execution exceptions have the same priority.

Table 5.1 Priority of Exceptions *(continued)*

Exception	Description
DDBL / DDBS	EJTAG Data Address Break (address only).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.
AdES	Store address alignment error. Store to protected address.
XTLBL	Load XTLB miss. Load XTLB hit to page with V = 0
TLBL	Load TLB miss. Load TLB hit to page with V = 0
DFTLBPAR	FTLB data load/store parity error.
XTLBS	Store XTLB miss. Store XTLB hit to page with V = 0.
TLBS	Store TLB miss. Store TLB hit to page with V = 0.
TLBRI	TLB Read Inhibit. Occurs when there is an attempt to access a page table whose RI bit is set.
TLB Mod	Store to TLB page with D = 0.

5.5 Exception Vector Locations

The location of the exception vector in the P6600 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. Refer to the EVA chapter at the end of this manual for more information.

The *SI_EVAReset* pin 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 *SI_EVAReset* pin is deasserted at reset, the P6600 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 P6600 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 *SI_EVAReset* pin is asserted at reset, the P6600 core comes up in the EVA configuration (default size for *xkseg0* space = 3 GB). Refer to the *EVA Application Note* for more information.

The function of the *Config5.*$_K$ bit and the *SI_UseExceptionBase* pin is shown in Table 5.2.

Table 5.2 *SI_UseExceptionBase* Pin and CONFIG5.K Encoding

CONFIG5.K Bit	*SI_UseExceptionBase* Pin	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 Use only *SI_ExceptionBase[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: *SI_ExceptionBase[31:12]* = 2'b10, *SI_ExceptionBase[29:12]* pins, 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 *SI_ExceptionBase[31:12]* pins. Refer to the EVA chapter for more information.	The *SI_ExceptionBase[31:12]* pins are used directly to derive the BEV location. The *SI_UseExceptionBase* pin is ignored.

Another degree of flexibility in the selection of the vector base address, for use when $Status_{BEV}$ equals 1, is provided via a set of input pins, *SI_UseExceptionBase*, *SI_ExceptionBase*[31:12], and *SI_ExceptionBaseMask*[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 5.3. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In the P6600 core, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0. Table 5.3 shows the vector base address when the core is in legacy setting and the *SI_UseExceptionBase* pin is 0.

Table 5.4 shows the vector base addresses when the core is in legacy setting and the *SI_UseExceptionBase* equals 1. As can be seen in Table 5.4, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where $Status_{BEV}$ = 0 are not affected.

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_UseExceptionBase = 0

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	`0xFFFF_FFFF_BFC0.0000`	
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=0)	`0xFFFF_FFFF_BFC0.0480`	
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr* `[31:7] || 7'b0000000`	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	`0xFFFF_FFFF_FF20.0200`	

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_UseExceptionBase = 0 *(continued)*

Exception	**$Status_{BEV}$** 0	1
Cache Error	$EBase_{63..30}$ || 1 || $EBase_{28..12}$ || `0x000` Note that $EBase_{31..30}$ have the fixed value of `2b'10`	`0xFFFF_FFFF_BFC0.0300`
Other	$EBase_{63..12}$ || `0x000` Note that $EBase_{31..30}$ have the fixed value of `2'b10 when WG = 0.`	`0xFFFF_FFFF_BFC0.0200`
'||' denotes bit string concatenation		

In legacy mode, when the *SI_UseExceptionBase* pin is 1, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 5.4.

Table 5.4 Exception Vector Base Addresses — Legacy Mode, SI_UseExceptionBase = 1

Exception	**$Status_{BEV}$** 0	1
Reset, NMI	`0xFFFF_FFFF || 0b10 ||` *SI_ExceptionBase* `[29:12] || 0x000`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=0)	`0xFFFF_FFFF || 0b10 ||`*SI_ExceptionBase*`[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)	`0x0xFFFF_FFFF_FF20.0200`	
Cache Error	$EBase_{63..30}$ || 1 || $EBase_{28..12}$ || `0x000` Note that $EBase_{31..30}$ have the fixed value `2'b10 when WG = 0. Exception vector resides in kseg1.`	`0xFFFF_FFFF` || `0b101` || *SI_ExceptionBase*`[28:12]` || `0x300` `Exception vector resides in kseg1.`
Other	$EBase_{63..12}$ || `0x000` Note that $EBase_{31..30}$ have the fixed value `2'b10 when WG = 0. Exception vector resides in kseg0/kseg1.`	`0xFFFF_FFFF` || `0b10` || *SI_ExceptionBase*`[29:12]` || `0x200` `Exception vector resides in kseg0/kseg1.`
'||' denotes bit string concatenation		

Table 5.5 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 5.26 (on page 322) shows the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1.

Table 5.6 combines these three tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, it is assumed that *IntCtl*$_{VS}$ = 0.

Table 5.5 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	`0x000`
XTLB Refill	`0x080`
General Exception	`0x180`
Interrupt, *Cause*$_{IV}$ = 1	`0x200`
Reset, NMI	None (uses reset base address)

Table 5.6 Exception Vectors

Exception	Config5$_K$	SI_UseExceptionBase	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	EJTAG ProbEn	Vector (IntCtl$_{VS}$ = 0)
Reset, NMI	0	0	x	x	x	x	`0xFFFF_FFFF_BFC0.0000`
Reset, NMI	0	1	x	x	x	x	`0xFFFF_FFFF` ‖ `2'b10` ‖ *SI_ExceptionBase* `[29:12]` ‖ `0x000`
Reset, NMI	1	x	x	x	x	x	`0xFFFF_FFFF` ‖ *SI_ExceptionBase* `[31:12]` ‖ `0x000`
EJTAG Debug	0	0	x	x	x	0	`0x0xFFFF_FFFF_BFC0.0480 (if` *DCR.RDVec*`=0)` *DebugVectorAddr* `[31:7]` ‖ `2b0000000 (if` *DCR.RDVec*`=1)`

EJTAG Debug	0	1	x	x	x	0	`0xFFFF_FFFF` ‖ `2'b10` ‖ *SI_ExceptionBase* `[29:12]` ‖ `0x480 (`
							`if` *DCR.RDVec*`=0)`
							DebugVectorAddr `[31:7]` ‖ `2b0000000`
							`(if` *DCR.RDVec*`=1)`

EJTAG Debug	1	x	x	x	x	0	`0xFFFF_FFFF` ‖ *SI_ExceptionBase* `[31:12]` ‖ `0x480 (if` *DCR.RDVec*`=0)` *DebugVectorAddr* `[31:7]` ‖ `2b0000000 (if` *DCR.RDVec*`=1)`
EJTAG Debug	x	x	x	x	x	1	`0x0xFFFF_FFFF_FF20.0200`
TLB Refill	x	x	0	0	x	x	*EBase* `[63:12]` ‖ `0x000 (EBase.WG = 1)` `2'b10` ‖ *EBase* `[29:12]` ‖ `0x000 (EBase.WG = 0)`
XTLB Refill	x	x	0	0	x	x	`0xFFFF_FFFF_8000_0080`
TLB Refill	x	x	0	1	x	x	*EBase* `[63:12]` ‖ `0x180 (EBase.WG = 1)` `2'b10` ‖ *EBase* `[29:12]` ‖ `0x180 (EBase.WG = 0)`
XTLB Refill	x	x	0	1	x	x	`0xFFFF_FFFF_8000_0180`
TLB Refill	0	0	1	0	x	x	`0x0xFFFF_FFFF_BFC0.0200`
XTLB Refill	x	x	1	0	x	x	`0x0xFFFF_FFFF_BFC0.0280`
TLB Refill	0	1	1	0	x	x	`0xFFFF_FFFF` ‖ `2'b10` ‖ *SI_ExceptionBase* `[29:12]` ‖ `0x200`
TLB Refill	1	x	1	0	x	x	`0xFFFF_FFFF` ‖ *SI_ExceptionBase* `[31:12]` ‖ `0x200`
TLB Refill	0	0	1	1	x	x	`0xFFFF_FFFF_BFC0.0380`

Table 5.6 Exception Vectors *(continued)*

Exception	$Config5_K$	SI_UseExceptionBase	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	EJTAG ProbEn	Vector ($IntCtl_{VS}$ = 0)
XTLB Refill	x	x	1	1	x	x	0xFFFF_FFFF_BFC0.0380
TLB Refill	0	1	1	1	x	x	0xFFFF_FFFF ‖ 2’b10 ‖ *SI_ExceptionBase*[29:12] ‖ 0x380
TLB Refill	1	x	1	1	x	x	0xFFFF_FFFF ‖ *SI_ExceptionBase*[31:12] || 0x380
Cache Error	0	x	0	x	x	x	*EBase*[63:30] ‖ 1’b1 ‖ *EBase*[28:12] ‖ 0x100 (EBase.WG = 1) *EBase*[31:30] ‖ 1’b1 ‖ *EBase*[28:12] ‖ 0x100 (EBase.WG = 0)
Cache Error	1	x	0	x	x	x	0xFFFF_FFFF_BFC0.0100 ($Config5_{CV}$ = 0)
Cache Error	1	x	0	x	x	x	0xFFFF_FFFF || EBase[31:12] || 0x100 ($Config5_{CV}$ = 1)
Cache Error	0	0	1	x	x	x	0x0xFFFF_FFFF_BFC0.0300
Cache Error	0	1	1	x	x	x	0xFFFF_FFFF || 2’b101 ‖ *SI_ExceptionBase*[28:12] ‖ 0x300
Cache Error	1	x	1	x	x	x	0xFFFF_FFFF || *SI_ExceptionBase*[31:12] ‖ 0x300
Interrupt	x	x	0	0	0	x	0xFFFF_FFFF || *EBase*[31:12] ‖ 0x180 (EBase.WG = 0) *EBase*[63:12] ‖ 0x180 (EBase.WG = 1)
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 ‖ *SI_ExceptionBase*[29:12] ‖ 0x380
Interrupt	1	x	1	0	0	x	*SI_ExceptionBase*[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	*SI_ExceptionBase*[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 ‖ *SI_ExceptionBase*[29:12] ‖ 0x380
All others	1	x	1	x	x	x	*SI_ExceptionBase*[31:12] ‖ 0x380

‘x’ denotes don’t care,
‘‖’ denotes bit string concatenation

5.6 General Exception Processing

With the exception of Reset, NMI, cache error, and EJTAG Debug exceptions, which have their own special processing as described below, 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 will be restarted, and the *BD* bit is set appropriately in the *Cause* register. The value loaded into the *EPC* register is dependent on whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 5.7 shows the value stored in each of the CP0 PC registers, including *EPC*.

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

Table 5.7 Value Stored in EPC, ErrorEPC, or DEPC on Exception

In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	Address of the instruction
Yes	Address of the branch or jump instruction (PC-4)
No	Upper 31 bits of the address of the instruction, combined with the ISA Mode bit
Yes	Upper 31 bits of the branch or jump instruction, combined with the ISA Mode bit

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

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

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

Operation:

```
/* If StatusEXL is 1, all exceptions go through the general exception vector */
/* and neither the EPC nor CauseBD are modified */
if StatusEXL = 1 then
   vectorOffset ← 0x180
else
      restartPC ← PC
      branchAdjust ← 4       /* Possible adjustment for delay slot */
   endif
   if InstructionInBranchDelaySlot then
      EPC ← restartPC - branchAdjust/* PC of branch/jump */
      CauseBD ← 1
   else
      EPC ← restartPC                /* PC of instruction */
      CauseBD ← 0
   endif
```

```
    /* Compute vector offsets as a function of the type of exception */
    if ExceptionType = TLBRefill then
        vectorOffset ← 0x000
    if ExceptionType = XTLBRefill then
        vectorOffset ← 0x080
    elseif (ExceptionType = Interrupt) then
        if (Cause_IV = 0) then
            vectorOffset ← 0x180
        else
            if (Status_BEV = 1) or (IntCtl_VS = 0) then
                vectorOffset ← 0x200
            else
                if Config3_VEIC = 1 then
                    VecNum ← Cause_RIPL
                else
                    VecNum ← VIntPriorityEncoder()
                endif
                vectorOffset ← 0x200 + (VecNum × (IntCtl_VS || 0b00000))
            endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
        endif /* if (Cause_IV = 0) then */
    endif /* elseif (ExceptionType = Interrupt) then */
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
Status_EXL ← 1

/* Calculate the vector base address */
if Status_BEV = 1 then
    vectorBase ← 0xFFFF.FFFF.BFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of EBase_31..30 forces the base to be in kseg0 or kseg1 */
        vectorBase ← 0xFFFF_FFFF || EBase_31..12 || 0x000
    else
        vectorBase ← 0xFFFF.FFFF.8000.0000
    endif
endif

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

5.7 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 and the *DBD* bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- 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 to 1.
- 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. Debug software need not look at the *DBD* bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

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

No other CP0 registers or fields are changed due to the debug exception, and thus no additional state is saved.

Operation:

```
if InstructionInBranchDelaySlot then
    DEPC ← PC-4
    Debug_DBD ← 1
else
    DEPC ← PC
    Debug_DBD ← 0
endif
Debug_D* bits at at [5:0] ← DebugExceptionType
Debug_Halt ← HaltStatusAtDebugException
Debug_Doze ← DozeStatusAtDebugException
Debug_DM ← 1
if EJTAGControlRegister_ProbTrap = 1 then
    PC ← 0xFFFF_FFFF_FF20_0200
else
    if DebugControlRegister_RDVec = 1 then
        if CacheErr then
            PC ← 2#101 || DebugVectorAddr_28..7 || 2#0000000
        else
            PC ← 2#10 || DebugVectorAddr_29..7 || 2#0000000
    else
        if SI_UseExceptionBase
            if CacheErr then
                PC ← 0xFFFF.FFFF ||2#101 || SI_ExceptionBase[28:12] || 0x000
            else
                PC ← 0xFFFF.FFFF ||2#10 || SI_ExceptionBase[29:12] || 0x000
        else
            PC ← 0xFFFF_FFFF_BFC0_0480
endif
```

The location of the debug exception vector is determined by the *ProbTrap* bit in the *EJTAG Control* register (*ECR*) and the *RDVec* bit in the *Debug Control* register (*DCR*), as shown in Table 5.8.

Table 5.8 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	RDVec bit in DCR Register	Debug Exception Vector Address
0	0	0xBFC0 0480
0	1	$\text{DebugVectorAddr}_{31..7}$ || 0000000
1	0	0xFF20 0200 in dmseg

Table 5.8 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	RDVec bit in DCR Register	Debug Exception Vector Address
1	1	

The value in the optional drseg register *DebugVectorAddr* (offset 0x00020) is used as the debug exception vector when the *ECR ProbTrap* bit is 0 and when enabled through the optional *RDVec* control bit in the *Debug Control Register* (*DCR*). Bit 0 of *DebugVectorAddr* determines the ISA mode used to execute the handler. Figure 5.1 shows the format of the *DebugVectorAddr* register; Table 5.9 describes the *DebugVectorAddr* register fields.

Figure 5.1 DebugVectorAddr Register Format

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

Table 5.9 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
1	31	Ignored on write; returns one on read.	R	1
DebugVectorOffset	29:7	Programmable Debug Exception Vector Offset	R/W	Preset to 0x7F8009
IM	0	ISA mode to be used for exception handler	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.

If the TAP is not implemented, the debug exception vector location is as if *ProbTrap*=0.

5.8 Exception Descriptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 5.1.

5.8.1 Reset Exception (Reset)

A reset exception occurs when the *SI_Reset* signal is asserted to the processor. This exception is not maskable. When a Reset exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset exception, the state of the processor is not defined, with the following exceptions:

- The *Wired* register is initialized to zero.
- The *Config* register is initialized with its boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.

- The *I*, *R*, and *W* fields of the *WatchLo* register are initialized to 0.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xFFFF_FFFF_BFC0_0000 (P6600) or other address depending on the product type.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

```
Wired ← 0
Config ← ConfigurationState
Status_BEV ← 1
Status_SR ← 0
Status_NMI ← 0
Status_ERL ← 1
WatchLo_I ← 0
WatchLo_R ← 0
WatchLo_W ← 0
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

5.8.2 Debug Single Step Exception (DSS)

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non-jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the *SSt* bit in the *Debug* register, and are always disabled for the first one/two instructions after a DERET.

The *DEPC* register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the *DEPC* register will not point to the instruction which has just been single stepped, but rather the following instruction. The *DBD* bit in the *Debug* register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and *DEPC* will point to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with *DEPC* pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

5.8.3 Debug Interrupt Exception (DINT)

A debug interrupt exception is either caused by the EjtagBrk bit in the *EJTAG Control* register (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The *DEPC* register is set to the instruction where execution should continue after the debug handler is through. The *DBD* bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

5.8.4 Non-Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the *SI_NMI* signal is asserted to the processor. *SI_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xFFFF_FFFF_BFC0_0000 (P6600) or other address depending on the product type.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

```
Status_BEV ← 1
Status_SR ← 0
Status_NMI ← 1
Status_ERL ← 1
if InstructionInBranchDelaySlot then
   ErrorEPC ← PC - 4
else
   ErrorEPC ← PC
endif
PC ← 0xFFFF_FFFF_BFC0_0000
```

5.8.5 Machine Check Exception

A machine check exception occurs when the processor detects an internal inconsistency. The following conditions cause a machine check exception:

- A TLBWI instruction to the FTLB and the index and VPN2 are not consistent and the EHINV bit is not set. See Section 3.12 of the MMU chapter.
- A TLBWI instruction to the FTLB and the PageMask register does not correspond to the FTLB page size setting in bits 12:8 of the *Config4* register ($Config4_{FTLB\ Page\ Size}$)
- A TLBP instruction and a duplicate/overlap is detected across the FTLB/VTLB.
- Any TLB lookup and a duplicate/overlap is detected across the FTLB/VTLB.

The machine check exception can be either precise or imprecise depending on the type of error.

The machine check exception is imprecise on:

- A Load/Store Unit (LSU) or Instruction Fetch Unit (IFU) lookup matching duplicate entries

The machine check exception is precise on:

- TLBP matching duplicate entries.
- TLBWI to the FTLB with the page size != the FTLB page size.
- TLBWI to the FTLB with EHINV=0 and the FTLB set implied by the VPN not the same as the set implied by the index.

Cause Register ExcCode Value:

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.6 Interrupt Exception (Int)

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register and the interrupt input is asserted. See 5.11 "Interrupts" on page 316 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 5.10 Register States an Interrupt Exception

Register State	Value
CauseIP	Indicates the interrupts that are pending.

Entry Vector Used:

See 5.11.2 "Generation of Exception Vector Offsets for Vectored Interrupts" on page 322 for the entry vector used, depending on the interrupt mode the processor is operating in.

5.8.7 Debug Instruction Break Exception (DIB)

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and DBD bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:

DIB

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.8.8 Watch Exception — Instruction Fetch or Data Access (WATCH)

The Watch facility provides a software debugging vehicle by initiating a watch exception when an instruction or data reference matches the address information stored in the *WatchHi* and *WatchLo* registers. A Watch exception is taken immediately if the *EXL* and *ERL* bits of the *Status* register are both zero and the *DM* bit of the *Debug* register is also zero. If any of those bits is a one at the time that a watch exception would normally be taken, then the *WP* bit in the *Cause* register is set, and the exception is deferred until all three bits are zero. Software may use the *WP* bit in the *Cause* register to determine if the *EPC* register points at the instruction that caused the watch exception, or if the exception actually occurred while in kernel mode.

The Watch exception can occur on either an instruction fetch or a data access. Watch exceptions that occur on an instruction fetch have a higher priority than watch exceptions that occur on a data access.

Register ExcCode Value:

WATCH

Additional State Saved:

Table 5.11 Register States on Watch Exception

Register State	Value
$Cause_{WP}$	Indicates that the watch exception was deferred until after $Status_{EXL}$, $Status_{ERL}$, and $Debug_{DM}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution.
$WatchHi_{I,R,W}$	Set for the watch channel that matched, and indicates which type of match there was.

Entry Vector Used:

General exception vector (offset 0x180)

5.8.9 Address Error Exception — Instruction Fetch/Data Access (AdEL/AdES)

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction that is not aligned on a word boundary
- LL, LLE, SC, and SCE instructions with misaligned addresses
- Any load instruction with a misaligned address and cacheable coherency attribute of uncached
- Any store instruction with a misaligned address and cacheable coherency attribute of uncached
- Any load/store instructions with misaligned address to a region defined as a non-speculative region by the MAAR register
- Reference the kernel address space from User mode

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both *EPC* and *BadVAddr* point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

Cause Register ExcCode Value:

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

Additional State Saved:

Table 5.12 CP0 Register States on Address Exception Error

Register State	Value
BadVAddr	Failing address
$Context_{VPN2}$	UNPREDICTABLE
$EntryHi_{VPN2}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.8.10 TLB Refill Exception — Instruction Fetch or Data Access (TLBL/TLBS)

During an instruction fetch or data access, a TLB refill exception occurs when no TLB entry matches a reference to a mapped address space and the *EXL* bit is 0 in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off. In that case, a TLB Invalid exception occurs.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 5.13 CP0 Register States on TLB Refill Exception

Register State	Value
BadVAddr	Failing address.
Context	The *BadVPN2* field contains $VA_{31:13}$ of the failing address.
EntryHi	The *VPN2* field contains $VA_{31:13}$ of the failing address; the *ASID* field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

TLB refill vector (offset 0x000) if $Status_{EXL}$ = 0 at the time of exception;

General exception vector (offset 0x180) if $Status_{EXL}$ = 1 at the time of exception

5.8.11 TLB Refill and XTLB Refill Exceptions — Instruction Fetch or Data Access (TLBL/TLBS)

A TLB Refill or XTLB Refill exception occurs in a TLB-based MMU when no TLB entry matches a reference to a mapped address space and the EXLbit is zero in the CP0 Status register. Note that this is distinct from the case in which an entry matches but has the valid bit off, in which case a TLB Invalid exception occurs. Refill exceptions have distinct exception vector offsets: 0x000 for a 32-bit TLB Refill and 0x080 for a 64-bit extended TLB ("XTLB") refill. The XTLB refill handler is used whenever a reference is made to an enabled 64-bit address space.

Cause Register ExcCode Value

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

See Table 9.56 on page 238

Additional State Saved:

Table 5.14 CP0 Register States on TLB Refill Exception

Register State	Value
Context	If Config3.CTXTC bit is set, then the bits of the Context register corresponding to the set bits of the VirtualIndex field of the ContextConfig register are loaded with the bits (starting at bit 31) of the virtual address that missed. If Config3.CTXTC bit is clear, then the BadVPN2 field contains VA31:13 of the failing address
XContext	If Config3.CTXTC bit is set, thenthe bits of the BadVPN2 field corresponding to the set bits of the VirtualIndex field of the ContextConfig register are loaded with the high-order bits (starting at SEGBITS-1) of the virtual address that missed and the R field contains VA[63:62] of the failing address. If Config3.CTXTC bit is clear, then the XContext BadVPN2 field contains VA[SEGBITS-1:13], and the XContext R field contains VA[63:62] of the failing address.
EntryHi	The EntryHi VPN2 field contains VA[SEGBITS-1:13] of the failing address and the EntryHi R field contains VA[63:62] of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used

- TLB Refill vector (offset 0x000) if 64-bit addresses are not enabled and Status.EXL = 0 at the time of exception.
- XTLB Refill vector (offset 0x080) if 64-bit addresses are enabled and Status.EXL = 0 at the time of exception.
- General exception vector (offset 0x180) in either case if Status.EXL = 1 at the time of exception

5.8.12 TLB Invalid Exception — Instruction Fetch or Data Access (TLBINV)

During an instruction fetch or data access, a TLB invalid exception occurs in one of the following cases:

- No TLB entry matches a reference to a mapped address space; and the *EXL* bit is 1 in the *Status* register.
- A TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 5.15 CP0 Register States on TLB Invalid Exception

Register State	Value
BadVAddr	Failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.8.13 TLB Execute-Inhibit Exception (TLBXI)

A *TLB execute-inhibit* exception occurs when there is a execute access from a TLB entry whose XI bit is set. The *TLB execute-inhibit* exception type can only occur if execute-inhibit exceptions are enabled by setting bit 30 (XIE) in the *PageGrain* register.

In addition, the type of exception taken depends on the state of the $PageGrain_{IEC}$ bit. If the XI bit of the entry is set, and the $PageGrain_{IEC}$ bit is set, a TLBXI exception is taken. If the $PageGrain_{IEC}$ bit is cleared, a *TLBL* exception is taken.

Cause Register ExcCode Value:

if $PageGrain._{IEC}$ == 0 TLBL

if $PageGrain._{IEC}$ == 1 TLBXI

Additional State Saved:

Table 5.16 CP0 Register States on TLB Execute-Inhibit Exception

Register State	Value
BadVAddr	Failing address.
Context	If the *Config3*.$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that misssed. If the *Config3*.$_{CTXTC}$ bit is clear, then the *BadVPN2* field contains $VA_{31:13}$ of the failing address.
EntryHi	The *VPN2* field contains $VA_{31:13}$ of the failing address; the *ASID* field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.8.14 TLB Read-Inhibit Exception (TLBRI)

A *TLB read-inhibit* exception occurs when there is an attempt to read a TLB entry whose RI bit is set. The *TLB read-inhibit* exception type can only occur if read-inhibit exceptions are enabled by setting bit 31 (RIE) in the *PageGrain* register.

In addition, the type of exception taken depends on the state of the *PageGrain*$_{IEC}$ bit. If the RI bit of the entry is set, and the *PageGrain*$_{IEC}$ bit is set, a TLBRI exception is taken. If the *PageGrain*$_{IEC}$ bit is cleared, a *TLBL* exception is taken.

Cause Register ExcCode Value:

if *PageGrain*.$_{IEC}$ == 0 TLBL

if *PageGrain*.$_{IEC}$ == 1 TLBRI

Additional State Saved:

Table 5.17 CP0 Register States on TLB Read-Inhibit Exception

Register State	Value
BadVAddr	Failing address.
Context	If the *Config3*.$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that misssed. If the *Config3*.$_{CTXTC}$ bit is clear, then the *BadVPN2* field contains $VA_{31:13}$ of the failing address.
EntryHi	The *VPN2* field contains $VA_{31:13}$ of the failing address; the *ASID* field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.8.15 FTLB Parity Exception

An F*TLB parity* exception occures when a parity error is detected on an FTLB read operation. The error can occur in either the FTLB Tag RAM of the FTLB Data RAM. Note that FTLB parity errors can only occur when the bit 31 (PE) of the CP0 *Error Control* register (*ErrCtl*.$_{PE}$) is set, enabling system-wide parity errors.

When an FTLB parity error occurs, hardware sets bits 31:30 of the CP0 *Cache Error* register (*CacheErr*.$_{EREC}$) to a value of 2'b11 to indicate that the register contains information based on a TLB error. When the EREC field is set to 2'b11, bits 29:28 of the *Cache Error* register (*CacheErr*.$_{ED}$ and *CacheErr*.$_{ET}$) indicate if the error occurred in the FTLB data RAM or the FTLB tag RAM respectively.

Additional State Saved:

Table 5.18 CP0 Register States on an FTLB Parity Exception

Register State	Value
CacheErr	Error state. Defined in bits 31:28 of this register.
ErrorEPC	Restart PC
StatusERL	Set to 1

Entry Vector Used:

Cache Error vector (offset 0x100)

5.8.16 Cache Error Exception (ICache Error/DCache Error)

A cache error exception occurs when an instruction or data reference detects a cache tag or data error. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address. This exception can be imprecise and the *ErrorEPC* may not point to the instruction that saw the error. Additionally, because the caches on the cores within the P6600 core are coherent, cache errors detected on other cores could indicate data corruption for a process on this CPU. An error on another CPU will still cause a Cache Error exception, with the $CacheErr_{EE}$ indicating that the error occurred on another processor.

L2 cache errors are considered to be imprecise. An L2 cache error on a data load operation can potentially corrupt the target GPR.

Cause Register ExcCode Value

N/A

Additional State Saved

Table 5.19 CP0 Register States on Cache Error Exception

Register State	Value
CacheErr	Error state
ErrorEPC	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

5.8.17 Bus Error Exception — Instruction Fetch or Data Access (IBE)

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data read. Bus error exceptions cannot be generated on data writes. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Instruction errors are precise, while data bus errors can be imprecise. These errors are taken when the ERR code is returned on the *OC_SResp* input.

Cause Register ExcCode Value:

IBE: Error on an instruction reference

DBE: Error on a data reference

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.18 Debug Software Breakpoint Exception (DBp)

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and DBD bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:

DBp

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.8.19 Execution Exception — System Call (Sys)

The system call exception is one of the execution exceptions. All of these exceptions have the same priority. A system call exception occurs when a SYSCALL instruction is executed.

Cause Register ExcCode Value:

Sys

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.20 Execution Exception — Breakpoint (Bp)

The breakpoint exception is one of the execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value:

Bp

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.21 Execution Exception — Coprocessor Unusable (CpU)

The coprocessor unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

Cause Register ExcCode Value:

CpU

Additional State Saved:

Table 5.20 Register States on Coprocessor Unusable Exception

Register State	Value
$Cause_{CE}$	Unit number of the coprocessor being referenced

Entry Vector Used:

General exception vector (offset 0x180)

5.8.22 Execution Exception — Reserved Instruction (RI)

The reserved instruction exception is one of the execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

Cause Register ExcCode Value:

RI

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.23 Execution Exception — Floating Point Exception (FPE)

A floating point exception is initiated by the floating point coprocessor.

Cause Register ExcCode Value:

FPE

Additional State Saved:

Table 5.21 Register States on Floating Point Exception

Register State	Value
FCSR	Indicates the cause of the floating point exception

Entry Vector Used:

General exception vector (offset 0x180)

5.8.24 Execution Exception — Integer Overflow (Ov)

The integer overflow exception is one of the execution exceptions. All of these exceptions have the same priority. An integer overflow exception occurs when selected integer instructions result in a 2’s complement overflow.

Cause Register ExcCode Value:

Ov

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.25 Execution Exception — Trap (Tr)

The trap exception is one of the execution exceptions. All of these exceptions have the same priority. A trap exception occurs when a trap instruction results in a TRUE value.

Cause Register ExcCode Value:

Tr

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.8.26 Debug Data Break Exception (DDBL/DDBS)

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:

DDBL for a load instruction or *DDBS* for a store instruction

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

5.8.27 TLB Modified Exception (TLB Mod)

During a data access, a TLB modified exception occurs on a store reference to a mapped address if the following condition is true:

- The matching TLB entry is valid, but not dirty.

Cause Register ExcCode Value:

Mod

Additional State Saved:

Table 5.22 Register States on TLB Modified Exception

Register State	Value
BadVAddr	Failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.9 Synchronous and Synchronous Hypervisor Exceptions

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

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

Table 5.23 summarizes hypervisor conditions.

Table 5.23 Hypervisor Exception Conditions

Type	Root-mode Vector	Causes	Reference
Synchronous Hypervisor	General	Guest Privileged Sensitive Instruction	Section 5.9.1
Synchronous Hypervisor	General	Guest Software Field Change	Section 5.9.2
Synchronous Hypervisor	General	Guest Hardware Field Change	Section 5.9.3
Synchronous Hypervisor	General	Guest Reserved Instruction Redirect	Section 5.9.4
Synchronous Hypervisor	General	Hypercall	Section 5.9.5

5.9.1 Guest Privileged Sensitive Instruction Exception

A Guest Privileged Sensitive Instruction exception occurs when an attempt is made to use a Guest Privileged Sensitive Instruction from guest mode, where the instruction is either not permitted in guest mode or is not enabled in guest mode. The list of sensitive instructions follows:

- WAIT
- CACHE, CACHEE
  - when $GuestCtl0_{CG}$=0

- with anything other than ‘Address’ as Effective Address Operand Type, if $GuestCtl0_{CG}$=1. Specifically CACHE(E) instructions with code 0b000, 0b001, 0b010, 0b011 will cause a GPSI.

$GuestCtl0Ext_{CGI}$ is an optional qualifier of $GuestCtl0_{CG}$. If $GuestCtl0Ext_{CGI}$ =1 and $GuestCtl0_{CG}$=1 then CACHE(E) instructions of type Index Invalidate (code 0b000) are excluded from the CACHE(E) instructions that cause a GPSI.

- TLBWR, TLBWI, TLBR, TLBP, TLBINV, TLBINVF when $GuestCtl0_{AT}$!= 3.
  - TLBINV, TLBINVF are optional in the baseline architecture.
- Access to *PageGrain*, *Wired*, *SegCtl0*, *SegCtl1, SegCtl2*, *PWBase*, *PWField*, *PWSize, PWCtl* when $GuestCtl0_{AT}$!= 3 (Guest TLB resources disabled)
- Write access to any $Config_{0-7}$ register when $GuestCtl0_{CF}$=0
- Access to *Count* or *Compare* registers when $GuestCtl0_{GT}$=0
  - including indirect read from CC using RDHWR providing CC is present and enabled by guest *HWREna*.
- Access to CP0 registers, or other non-CP0 sources (CCRes, Sync_Step), using RDHWR when $GuestCtl0_{CP0}$=0 providing the registers are enabled for access by guest user or kernel.

  - Guest user access is enabled either by guest *HWREna* or $Status_{CU0}$.
  - Guest kernel always has access to registers specified by RDHWR, regardless of guest *HWREna* and $Status_{CU0}$.
  - Guest access to CC may also cause GPSI based on $GuestCtl0_{GT}$.

  Whether a guest RDHWR access to an implementation defined register causes a GPSI is implementation defined i.e., the access may cause a GPSI or not in an implementation dependent manner. Access to reserved registers with RDWR generates a Reserved Instruction exception in respective context.

  Guest GPSI applies to both guest user and kernel access, as $GuestCtl0_{CP0}$ applies to guest kernel access also.
- Write to *Count* register
- All Privileged Instruction, excluding selected Release 3 EVA instructions, when $GuestCtl0_{CP0}$=0

  The baseline architecture defines privileged instructions as the following: CACHE, DI, EI, MTC0, MFC0, ERET, DERET, RDPGPR, WRPGPR, WAIT, all Enhanced Virtual Addressing (EVA) related instructions (e.g., LBE, LBUE) (optional), and all TLB related instructions.

  All EVA instructions except CACHEE are excluded from causing a GPSI when $GuestCtl0_{CP0}$=0.

  Privileged instructions are defined in Volume II of the architecture. Instructions that are supported depend on the architecture release that an implementation is compliant with, and in some cases instructions are optional within a release.
- Access to any Guest CP0 registers that are active in guest context and always take Guest Privileged Sensitive Instruction Exception.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

GPSI (0, 0x00)

Additional State saved

BadInstr

BadInstrP

Entry Vector Used

General Exception Vector (offset 0x180).

5.9.2 Guest Software Field Change Exception

A Guest Software Field Change exception occurs when the value of certain CP0 register bitfields changes during guest-mode execution.

Change is caused by MTC0 execution, the instruction is copied to the root context *BadInstr* register (if the implementation is so equipped) and the exception is taken. The exception is used to allow the hypervisor to track changes to certain guest-context fields (e.g. $Status_{RP}$ or $Cause_{IV}$). This can be used to ensure the proper operation of the emulated guest virtual machine.

This exception can only be raised by a MTC0 instruction executed in guest mode. It is the responsibility of Root to increment EPC in order to return to the instruction following the MTC0. Note that the guest MTC0 is never executed, unless causing GSFC exception is disabled by $GuestCtl0Ext_{FCD}$, or selectively by $GuestCtl0_{SFC1/2}$. It is the responsibility of Root to modify the field on the behalf of Guest, providing guest access causes a GSFC.

If a field indicated below is meant to enable access to a resource, but the implementation does not support the resource, then a GSFC exception is not taken. As an example, if $Guest.Config1_{MD}$=0, i.e.,, MDMX Module is not supported, then a guest write to $Guest.Status_{MX}$ will not cause a GSFC exception.

Changes to the following CP0 register bit fields always trigger the exception.

- *Guest.Status* bits: CU[2:1], FR, MX, BEV, SR, NMI, UM/KSU, ERL, Impl (17:16)

  A change to UM/KSU can only cause a GSFC if $GuestCtl0_{MC}$=1. Whether guest access to $Status_{Impl}$ causes a GSFC is implementation-dependent.

  The occurrence of GSFC on guest write to $Status_{FR}$ is dependent on $Config5_{UFR}$ as described below.

- *Config5* : MSAEn. (Enable for MIPS SIMD Architecture module. Applicable only if MSA implemented.)
  : UFR. (User FR enable)
- *PageGrain:* ELPA.
- *Guest.Cause* bits: DC, IV
- *Guest.IntCtl* bits: VS
- *Root.PerfCnt* w/ $PerfCnt_{EC}$=2/3: Event, EventExt(Optional)

  PerfCnt does not exist in guest context. When $PerfCnt_{EC}$=2/3, however root context registers are accessible to Guest. GPSI on guest access is only taken only in this configuration.

Guest software may modify CU[2:1] often. To prevent frequent GSFC on these events, a set of enables, $GuestCtl0_{SFC2}$ and $GuestCtl0_{SFC1}$, have been provided.

Guest write of 0 to SR or NMI will raise this exception. Guest write of 1 to Guest $Status_{SR}$ or $Status_{NMI}$ is **UNPREDICTABLE** behavior as specified in the base architecture. It is optional for an implementation to cause this exception on a guest write of 1 to either the SR or NMI within the *Status* register. Guest $Status_{SR}$ or $Status_{NMI}$ are never set by hardware, nor will Root software write of 1 to either Guest $Status_{SR}$ or $Status_{NMI}$ cause an interrupt in Guest context.

Guest software modification of EXL will not cause a GSFC. This is because guest kernel will often write EXL=1 prior to setting KSU to user mode(b10), allowing processor to stay in kernel mode. ERET will clear EXL, affecting change to user mode. To avoid frequent GSFC on such events, guest kernel modification of EXL is not trapped on.

If Root *PerfCnt.EC=2 or 3*, then Guest can access shared Root *PerfCnt* without GPSI exception. However, any change to the Event or EventExt fields must be reported as a GSFC exception to Root.

Release 6 introduces an optional feature which allows user code to change the value of $Status_{FR}$. The presence of this feature in a Release 6 implementation is determined by the writeable state of $Config5_{UFR}$. If $Config5_{UFR}$=1, then a GSFC exception on guest write to $Status_{FR}$ is not generated.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

GSFC(1, 0x01)

Additional State saved

BadInstr

BadInstrP

Entry Vector Used

General Exception Vector (offset 0x180)

5.9.3 Guest Hardware Field Change Exception

A Guest Hardware Field Change Exception is caused by exception/interrupt processing or a hardware initiated field change. The exception is taken after Guest state has been updated and before the following instruction is executed.

A Guest Hardware Field Change exception is considered synchronous with respect to the Guest action that caused it. In terms of priority, it is only lower than any asynchronous Root exception. It is not prioritized with respect to Guest exceptions: Guest exceptions are first prioritized amongst themselves, and then the Guest exception may then subsequently cause a Hardware Field Change exception.

When $GuestCtl0Ext_{FCD}$ = 1, then no Guest Hardware Field Change exception is triggered. Hardware events that cause the described events must be allowed to modify state as in the baseline architecture.

When $GuestCtl0_{MC}$=1, changes to the following bit-fields trigger this exception.

- Guest *Status* bits: EXL.

A change in value in this field causes a Guest Hardware Field Change exception, regardless of whether there is an effective change in mode.

Since events (Reset, NMI, Cache Error) that set ERL are always processed by Root, hardware initiated field changes involving ERL will not result in this exception.

Guest $Status_{EXL}$ will be modified by hardware on a Guest exception. The Guest Hardware Field Change exception is taken prior to the actual Guest exception handler (when EXL is set) and after the Guest exception handler is completed (when ERET clears EXL) but prior to the first Guest instruction after the handler. The Guest Hardware Field Change exception handler must compare state between successive invocations to determine if state of the EXL bit has changed.

For the transition of EXL from 0 to 1, it is recommended that guest context be loaded with exception related data as if the guest exception handler were to be executed. Prior to execution of first instruction of guest handler, hardware must cause a GHFC trap to root. The only root state modified is Root $Status_{EXL}$(=1), $Cause_{ExcCode}$(=”Guest Exit”) and $GuestCtl0_{GExcCode}$(=”GHFC”). Hardware handling of transition of EXL from 1 to 0 should be similar. In this manner, the hardware overhead of setting appropriate context for guest and root is kept to a minimum.

The GHFC exception must be viewed atomically with respect to the guest exception that caused it. In a recommended implementation, the guest exception will cause guest context to be updated simultaneously along with root context for the GHFC exception. Guest entry on completion of GHFC exception will cause related guest exception to be taken.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

GHFC(9, 0x09)

Entry Vector Used

General Exception Vector (offset 0x180).

5.9.4 Guest Reserved Instruction Redirect

A Guest Reserved Instruction Redirect Exception occurs when $GuestCtl0_{RI}$=1 and a guest mode instruction would trigger a Reserved Instruction Exception. This exception is raised before the guest mode exception can be taken. The instruction is not executed, the exception is taken in Root mode and the Guest context is unchanged.

The Reserved Instruction Redirect (GRR) must be prioritized in the context of other guest-mode exceptions. For e.g., a Coprocessor Unusable exception due to guest context is ranked higher in priority than a Reserved Instruction exception. Thus a Reserved Instruction Redirect exception is not taken in this case. Another e.g., relates to the case where $Root.Status_{CU1}$=0, while Guest.Status.CU1=1. If the processor is in guest-mode and executes a reserved COP1 instruction, then the Coprocessor Unusable exception is a result of Root qualification. It would be ranked higher priority than a Reserved Instruction exception for the same guest-mode instruction.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

GRR (3, 0x03)

Additional State saved

BadInstr

BadInstrP

Entry Vector Used

General Exception Vector (offset 0x180).

5.9.5 Hypercall Exception

A Hypercall Exception occurs when a HYPCALL instruction is executed. This is a Privileged Instruction and thus can only be executed in kernel mode (root-kernel or guest-kernel mode) or debug mode. It is specifically meant to cause a guest-exit.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

Hyp (2, 0x02)

Additional State saved

BadInstr

BadInstrP

Entry Vector Used

General Exception Vector (offset 0x180).

5.10 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- General exceptions
- TLB miss exceptions
- Reset and NMI exceptions
- Debug exceptions

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

Figure 5.2 General Exception Handler (HW)

Exceptions other than Reset, NMI, or first-level TLB miss. Note: Interrupts can be masked by IE or IMs, and Watch is masked if EXL = 1.

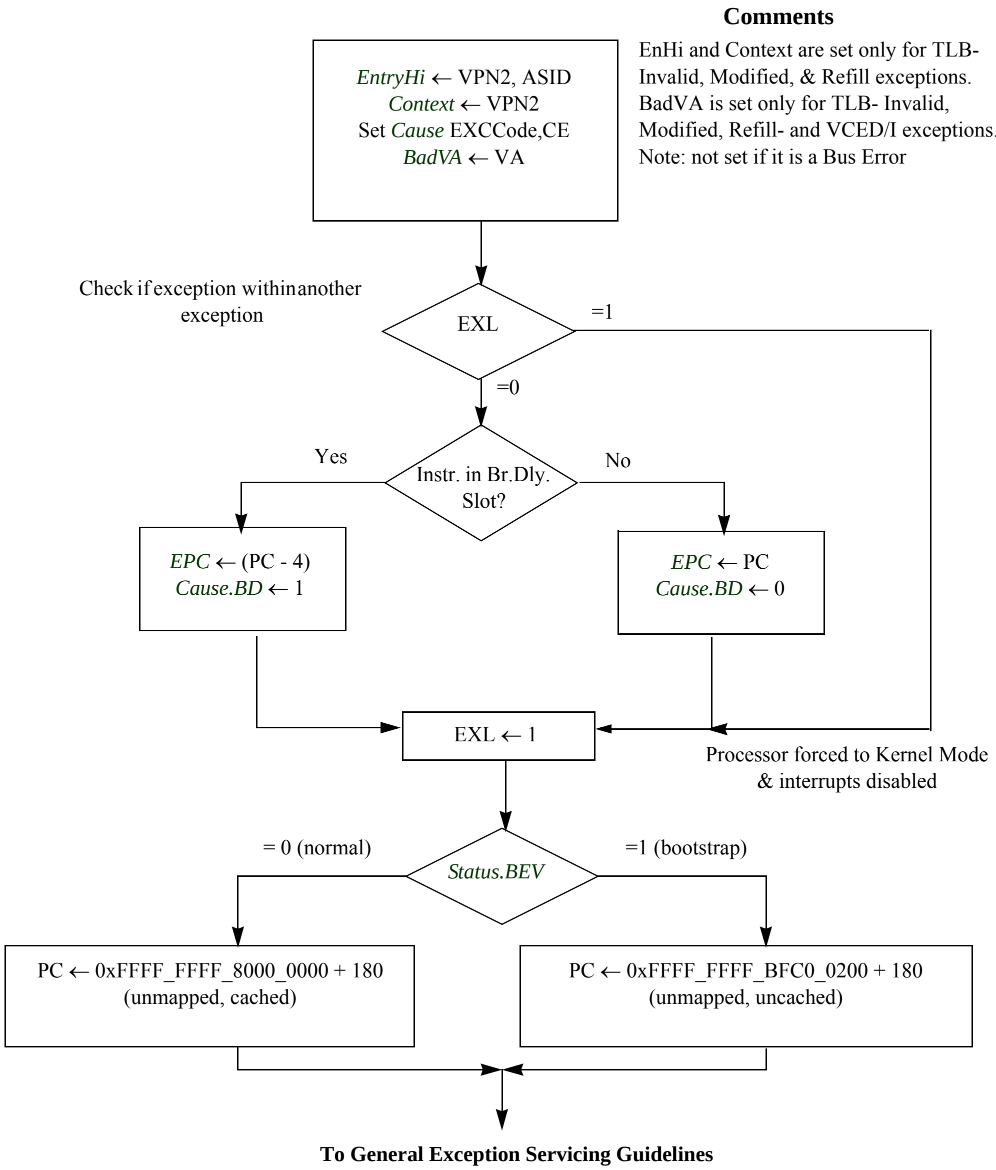


Figure 5.3 General Exception Servicing Guidelines (SW)

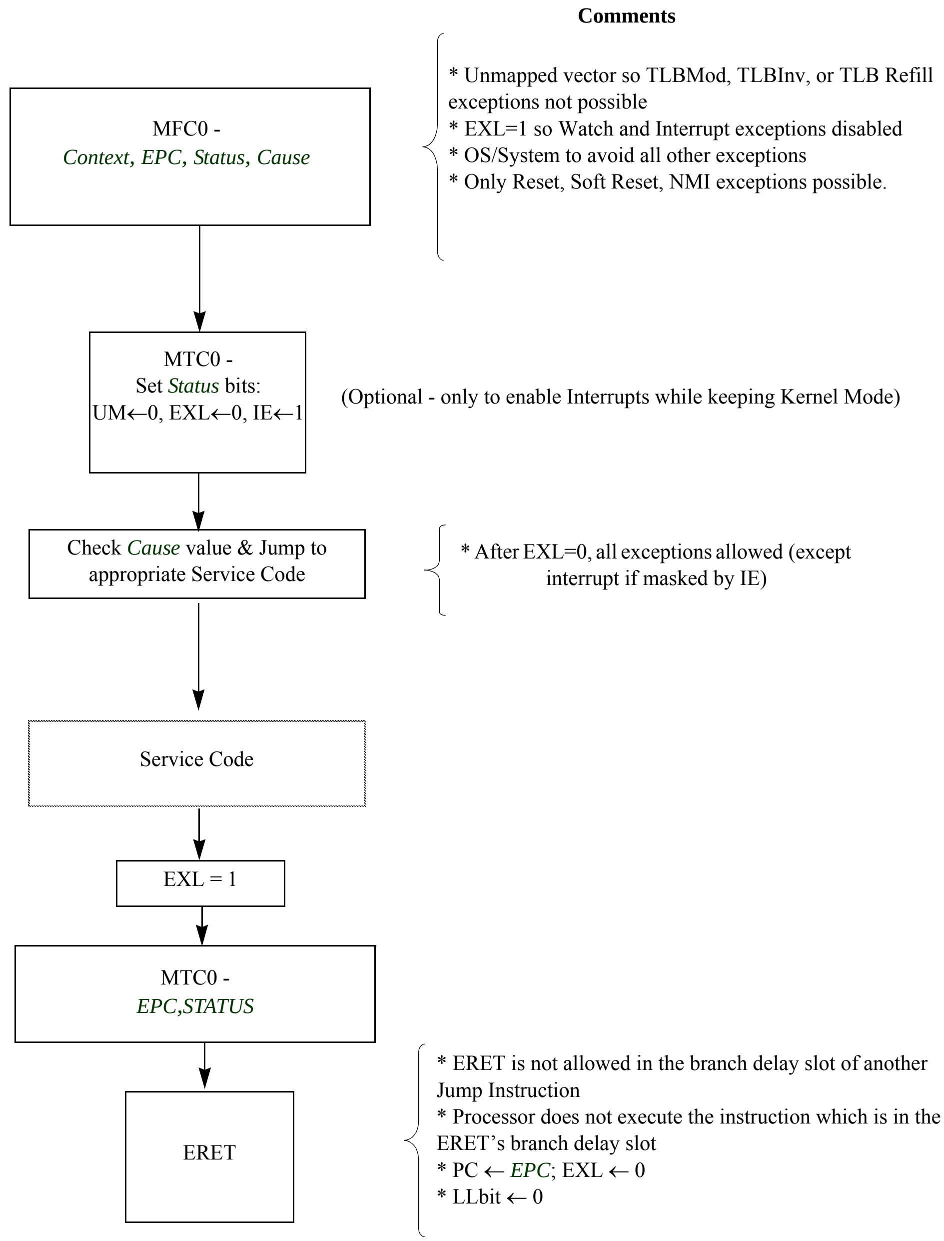


Figure 5.4 TLB Miss Exception Handler (HW)

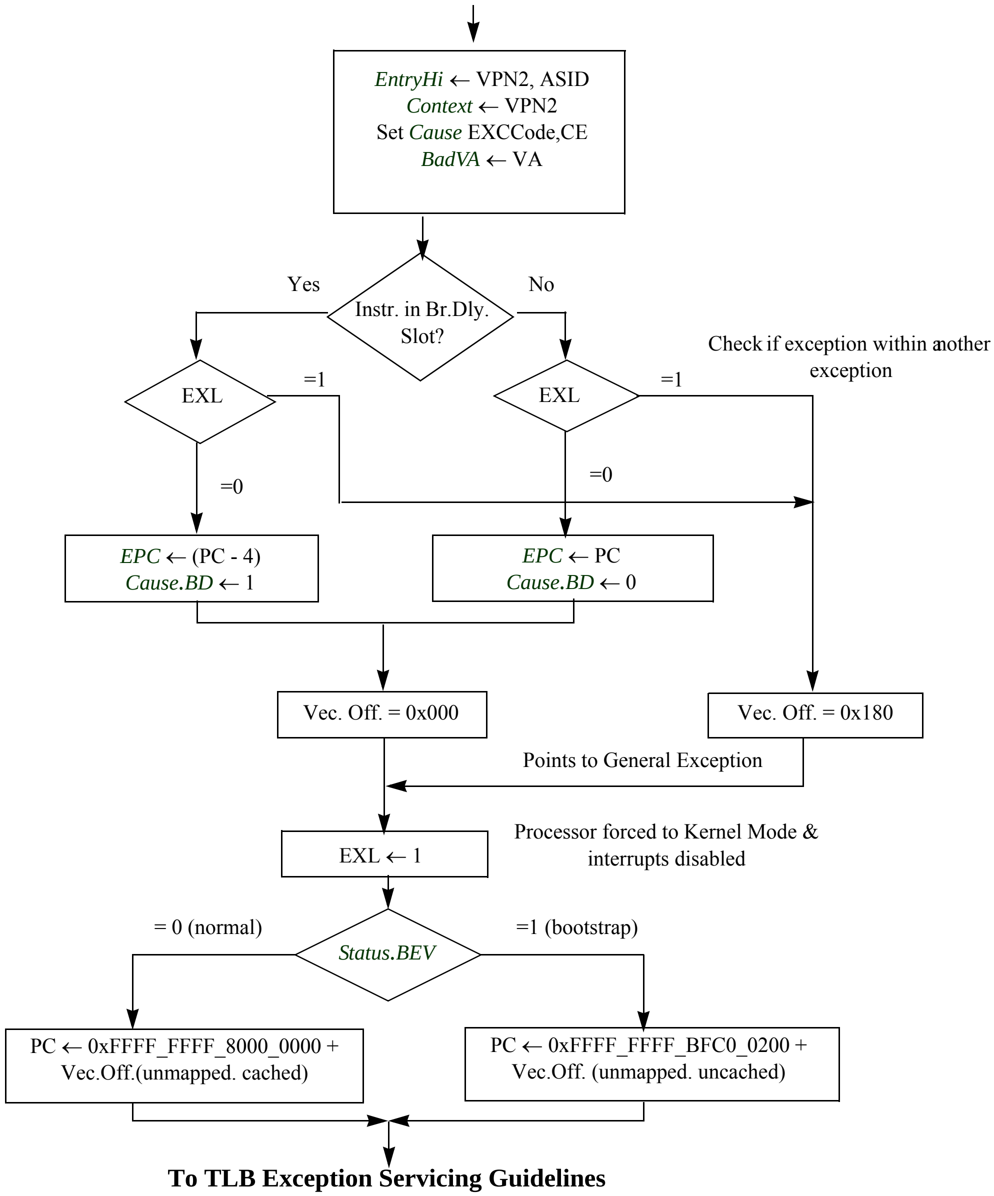


Figure 5.5 TLB Exception Servicing Guidelines (SW)

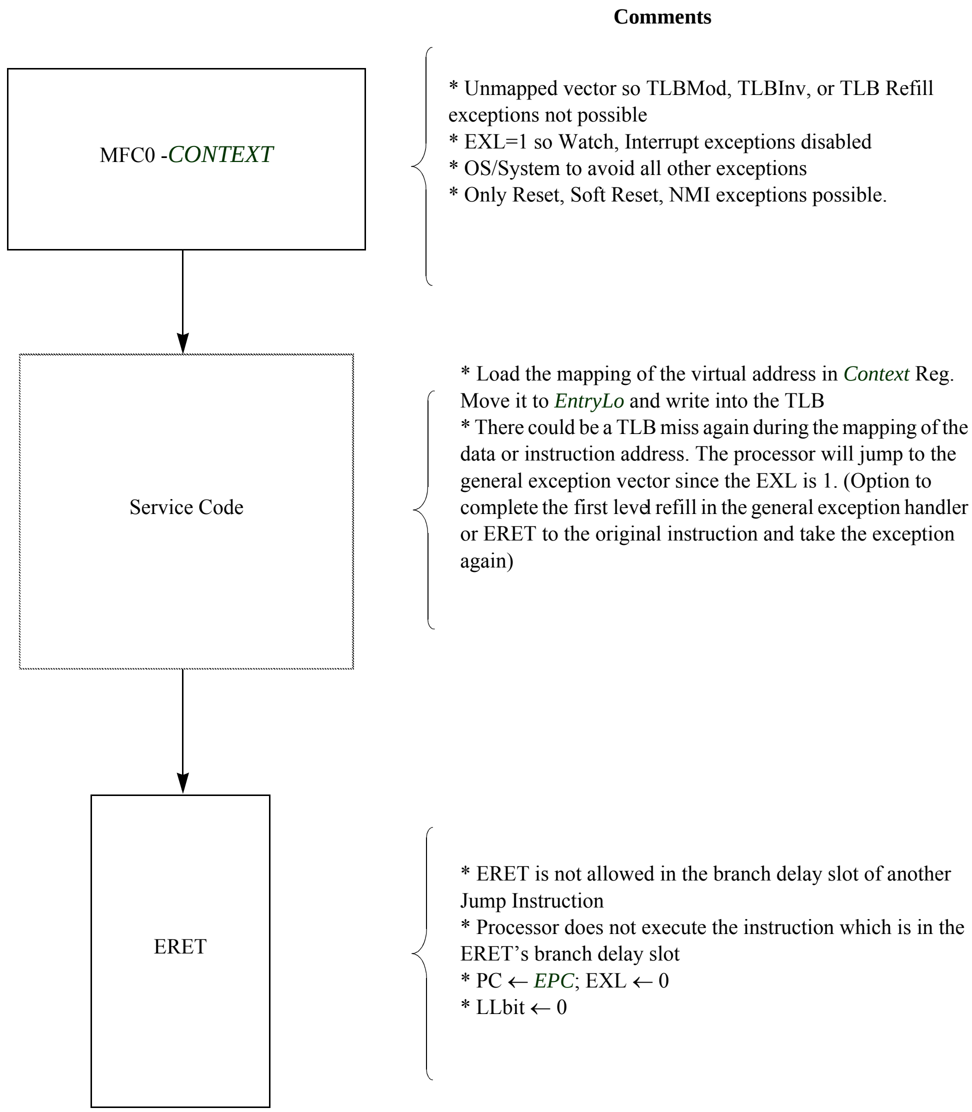
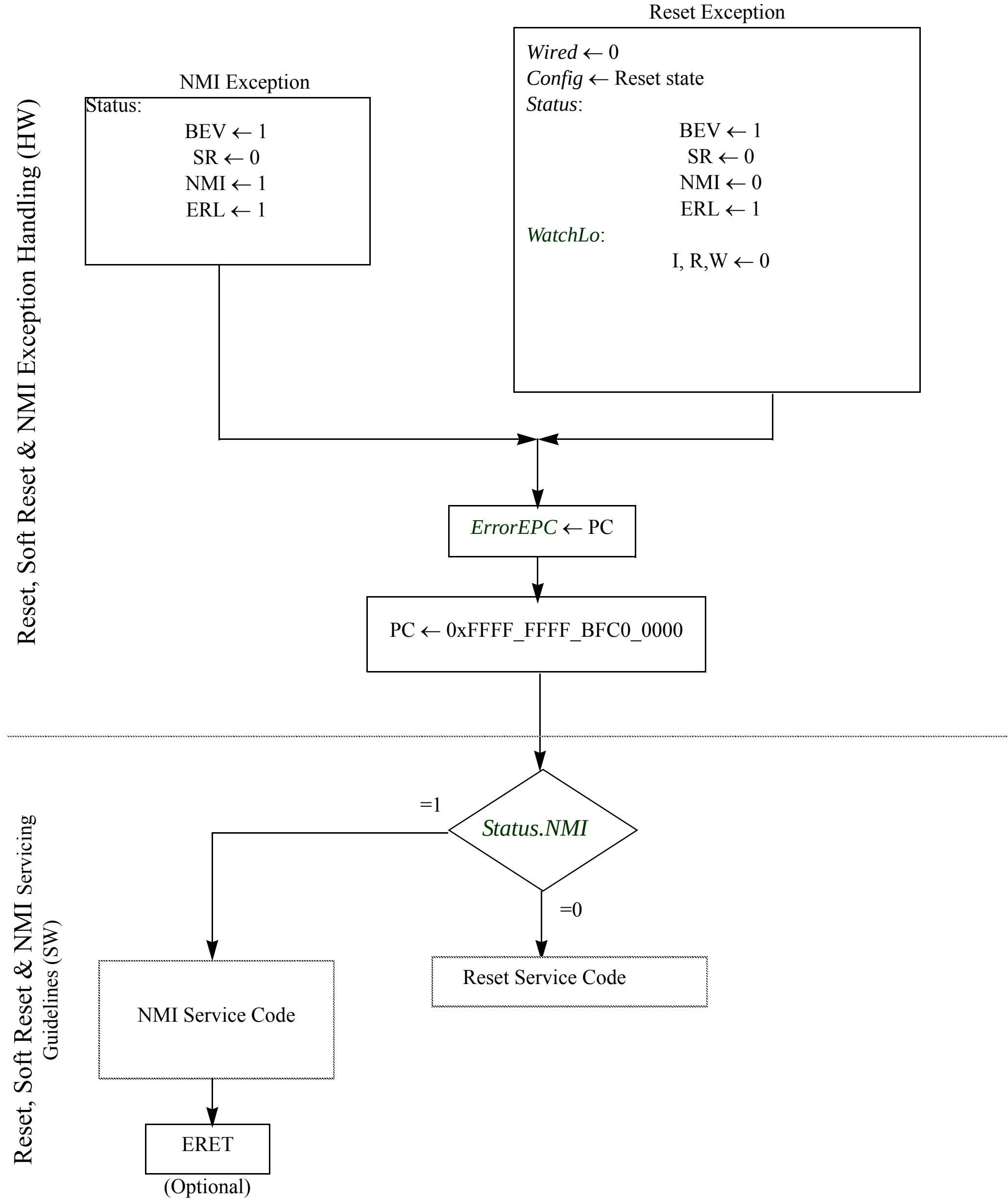


Figure 5.6 Reset and NMI Exception Handling and Servicing Guidelines



5.11 Interrupts

Release 6 of the MIPS64 architecture, implemented by the P6600 core, includes support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

Additionally, internal performance counters have been added to the P6600 core. These counters can be configured to count various events within the CPU. When the MSB of the counter is set, it can trigger a performance counter interrupt. This interrupt, like the timer interrupt, is an output from the core that can be brought back into the cores interrupt pins in a system-dependent manner.

The Fast Debug Channel feature in EJTAG provides a low overhead means for sending data between CPU software and the EJTAG probe. It includes a pair of FIFOs for transmit and receive data. Software can define FIFO thresholds for generating an interrupt. The fast debug channel interrupt is also routed similarly to the timer and performance counter interrupts. The interrupt status is made available on an output pin and can be brought back into the cores interrupt pins.

5.11.1 Interrupt Modes

The P6600 core includes support for three interrupt modes:

- Interrupt Compatibility mode, in which the behavior of the P6600 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. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. Although this mode is architecturally optional, it is always present on the P6600 core, so the *VInt* bit will always read as a 1.
- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the P6600 core, the *VEIC* bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

Following reset, the P6600 core defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 5.24 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 5.24 Interrupt Modes

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

5.11.1.1 Interrupt Compatibility Mode

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

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

Here is a typical software 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 */
```

5.11.1.2 Vectored Interrupt Mode

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

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

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 $IntCtl_{IPTI/IPCI/IPFDCI}$ 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 $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1, and if interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 5.25.

Table 5.25 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
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
	Software	SW1	IP1 and IM1	1
Lowest Priority		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 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. Such a routine might look as follows:

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

```
    mfc0   k0, C0_EPC          /* Get restart address */
    sw     k0, EPCSave         /* Save in memory */
    mfc0   k0, C0_Status       /* Get Status value */
    sw     k0, StatusSave      /* Save in memory */
    li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                               /*   this must include at least the IM bit */
                               /*   for the current interrupt, and may include */
                               /*   others */
    and    k0, k0, k1              /* Clear bits in copy of Status */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                   /* Clear KSU, ERL, EXL bits in k0 */
    mtc0   k0, C0_Status           /* Modify mask, switch to kernel mode, */
                                   /*   re-enable interrupts */

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

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

    di                         /* Disable interrupts - may not be required */
    lw     k0, StatusSave      /* Get saved Status (including EXL set) */
    lw     k1, EPCSave         /*   and EPC */
    mtc0   k0, C0_Status       /* Restore the original value */
    mtc0   k1, C0_EPC          /*  and EPC */
    ehb                        /* Clear hazard */
    eret                       /* Dismiss the interrupt */
```

5.11.1.3 External Interrupt Controller Mode

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

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

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

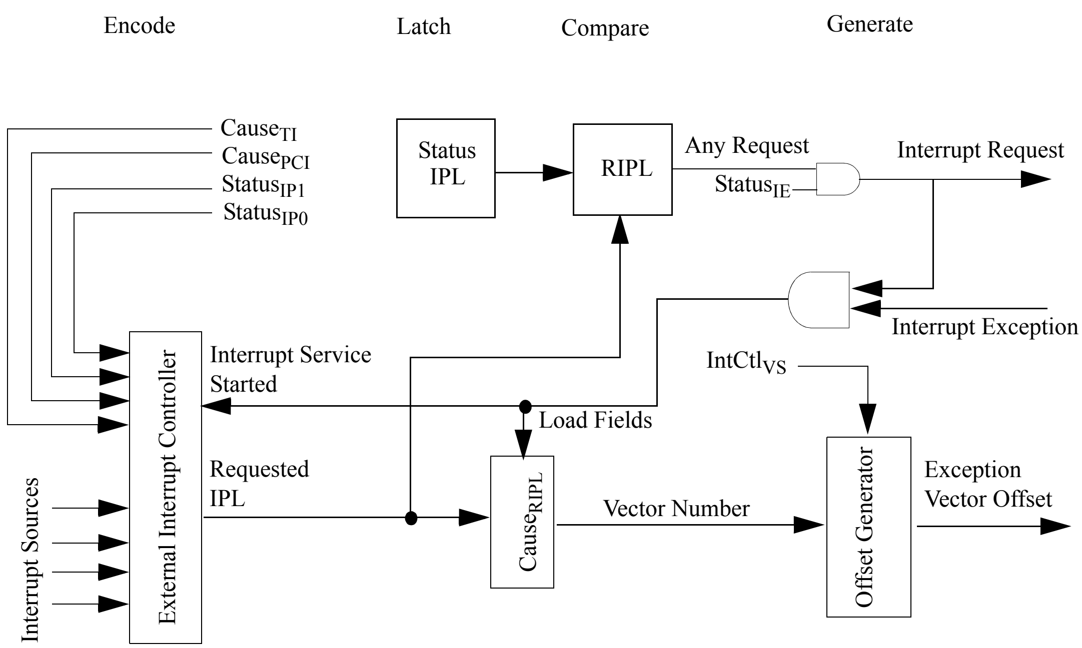
In EIC mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$) and the timer, performance counter, and fast debug channel interrupt requests ($Cause_{TI/PCI/FDCI}$) to the external interrupt controller, which prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hardwired logic block, or it can be configurable by control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. The values 1..63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. A value of 0 indicates that no interrupt requests are pending. The interrupt controller inputs this value on the 6 hardware interrupt lines, which are treated as an encoded value in EIC mode.

$Status_{IPL}$ (which overlays $Status_{IM7..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (a value of zero indicates that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with $Status_{IPL}$ to determine if the requested interrupt has a higher priority than the current IPL. If RIPL is strictly greater than $Status_{IPL}$, and interrupts are enabled ($Status_{IE}$ = 1, $Status_{EXL}$ = 0, and $Status_{ERL}$ = 0), an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $Cause_{RIPL}$ (which overlays $Cause_{IP7..IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. The interrupt exception uses the value of $Cause_{RIPL}$ as the vector number. Because $Cause_{RIPL}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing.

The operation of EIC interrupt mode is shown in Figure 5.7.

Figure 5.7 Interrupt Generation for External Interrupt Controller Interrupt Mode



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

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

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

```
mfc0   k1, C0_Cause         /* Read Cause to get RIPL value */
mfc0   k0, C0_EPC           /* Get restart address */
srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
sw     k0, EPCSave          /* Save in memory */
mfc0   k0, C0_Status        /* Get Status value */
sw     k0, StatusSave       /* Save in memory */
ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                              /* Clear KSU, ERL, EXL bits in k0 */
mtc0   k0, C0_Status          /* Modify IPL, switch to kernel mode, */
                              /*   re-enable interrupts */

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

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

5.11.2 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. Table 5.26 shows the exception vector offset for a representative subset of the vector numbers and values of the $IntCtl_{VS}$ field.

Table 5.26 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 ← 0x200 + (vectorNumber × (IntCtlVS ‖ 0b00000))
```

5.11.3 Global Interrupt Controller

The Global Interrupt Controller (GIC) handles the routing and masking of local interrupts, such as the timer, performance counter, fast debug channel interrupts, inter-processor interrupts, and external interrupts. This block can be configured to support various numbers of external interrupts and to support any of the CPU interrupt modes.

An interactive GUI is available to simplify the setup of desired event-routing through the GIC. The tool outputs a C-language function covering all required programming registers of the GIC.

Chapter 6

Coherence Manager

The coherence manager (CM2) in the P6600 Multiprocessing System is used to maintain coherency between the L1 caches of each core, and the shared L2 cache within the CM2. The CM2 also contains the Global Interrupt Controller (GIC), and Cluster Power Controller (CPC) and manages the interface of those components to the cores and the IOCU. The CM2 adds support for virtualization and L2 prefetching. Some of the new features are listed in Section 6.1, "CM2 Features".

The P6600 Global Control Registers address space (GCR) contains control/status registers for the entire P6600 Multiprocessing System cluster (see Section 6.4 "Global Control Block"), as well as the individual P6600 cores (see Section 6.5 "Core-Local and Core-Other Control Blocks") in the cluster.

The GCR address space has a total size of 32 KBytes, which is divided into 8 KByte blocks as described in Section 6.2 "Coherence Manager Address Map". The location of the GCR block in the system address map is controlled by the *GCR_BASE* register.

Physically, the registers are located within the GCR block of the Coherence Manager (CM2) and are accessed by the P6600 cores using 32-bit aligned uncached load/store instructions, or by I/O devices via the I/O Coherence Unit (IOCU), using read/write instructions.

This chapter contains the following sections:

6.1 CM2 Features

The P6600 coherence manager contains the following features:

- 128-bit data width between the CM2 and Cores, the CM2 and IOCU, IOCU to memory subsystem and CM2 to memory.
- When configured with 128-bit data the IOCU can handle requests of up to 256 bytes in length (previously was restricted to 128 bytes).
- The L2 Prefetcher that can dramatically improve performance for workloads with linear access patterns, such as memcopy.
- 40-bit address through the CM2 and IOCU.
- The CM2 PDtrace formats are extended to support 40-bit addresses.

- Virtualization support has been added to the General Interrupt Controller (GIC)
- Virtualization support via new IOMMU component included in IOCU.
- New performance counter events/qualifiers to measure L2 prefetcher effectiveness.
- New IOMMU functionality is embedded in the IOCU. An IOMMU standalone component is also available.
- Register access to multiple IOMMU's supported.
- CM Trace has a new field that indicates internal source of CPU request (instruction fetch, data load, prefetch instruction, hardware table walker).
- When Virtualization is enabled, the Guest ID is driven with the request on the main memory OCP port and the IOCU's Memory Mapped IO OCP Port.

6.2 Coherence Manager Address Map

Table 6.1 shows the address map of the four, 8-KB GCR blocks relative to the *GCR_BASE* as defined in the *GCR Base Register*. Each of these blocks of registers are described in the following sections.

Table 6.1 P6600 Control Space Address Map (Relative to GCR_BASE[39:15])

Address Range	Size (bytes)	Description
0x00_0000 - 0x00_1FFF	8 KB	**Global Control Block.** Contains registers pertaining to the global system functionality. All cores can access this block of registers.
0x00_2000 - 0x00_3FFF	8 KB	**Core-Local Control Block** (aliased for each P6600 core). Contains registers pertaining to the P6600 core issuing the request. Each core has its own copy of registers within this block.
0x00_4000 - 0x00_5FFF	8 KB	**Core-Other Control Block** (aliased for each P6600 core). This block of addresses gives each Core a window into another cores Core-Local Control Block. Before accessing this space, the *Core-Other_Addressing Register* in the Local Control Block must be set with the CORENum of the target Core.
0x00_6000 - 0x00_7FFF	8 KB	**Global Debug Block.** Contains global registers useful in debugging the P6600 MPS.

6.2.1 Block Offsets Relative to the Base Address

The block offsets for each of the four blocks listed in Table 6.1 above are relative to a GCR 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 field of the *GCR Base* register located at offset address 0x00_0000 in the Global Control Block. The MIPS default location for the GCR_BASE address is 0x00_1FBF_8. To determine the physical address of each block using the MIPS default, this value would be added to the GCR block offset to derive the absolute physical address as shown in Table 6.2.

Table 6.2 Absolute Address of GCR Register Blocks Using the MIPS Default

MIPS Default Base		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x00_1FBF_8	+	0x0000 - 0x1FFF	=	0x00_1FBF_ 8000 - 0x00_1FBF_9FFF	8 KB	Global Control Block.

Table 6.2 Absolute Address of GCR Register Blocks Using the MIPS Default *(continued)*

MIPS Default Base		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x00_1FBF_8	+	0x2000 - 0x3FFF	=	0x00_1FBF_ A000 - 0x00_1FBF_BFFF	8 KB	Core-Local Control Block
0x00_1FBF_8	+	0x4000 - 0x5FFF	=	0x00_1FBF_ C000 - 0x00_1FBF_DFFF	8 KB	Core-Other Control Block
0x00_1FBF_8	+	0x6000 - 0x7FFF	=	0x00_1FBF_ E000 - 0x00_1FBF_FFFF	8 KB	Global Debug Block

6.2.2 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 6.2 above. To determine the physical address of each register, the MIPS default base address is added to the corresponding GCR block offset plus the actual register offset to derive the absolute physical address as shown in Table 6.3. Note that this example shows only a few selected registers of the Global Control Block.

Table 6.3 Absolute Address of Individual Global Control Block Registers

MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address	Global Control Register
0x00_1FBF_8	+	0x0000	+	0x0000	=	0x00_1FBF_ 8000	CM2 Configuration.
0x00_1FBF_8	+	0x0000	+	0x0008	=	0x00_1FBF_ 8008	GCR Base.
0x00_1FBF_8	+	0x0000	+	0x0010	=	0x00_1FBF_ 8010	CM2 Control.
0x00_1FBF_8	+	0x0000	+	0x0018	=	0x00_1FBF_ 8018	CM2 Control2.
0x00_1FBF_8	+	0x0000	+	0x0020	=	0x00_1FBF_ 8020	CM2 Access Privilege.
....		....		....		....	....
0x00_1FBF_8	+	0x0000	+	0x0228	=	0x00_1FBF_ 8228	Attribute-Only Region 3 Mask.

The registers within the Core-Local blocks would be accessed in a similar manner as shown in Table 6.4.

Table 6.4 Absolute Address of Individual Core-Local Block Registers

MIPS Default Base		Core-Local Block Offset		Core-Local Register Offset		Absolute Physical Address	Global Control Register
0x00_1FBF_8	+	0x2000	+	0x0000	=	0x00_1FBF_ A000	Reserved.
0x00_1FBF_8	+	0x2000	+	0x0008	=	0x00_1FBF_ A008	Core-Local Coherence Control.
0x00_1FBF_8	+	0x2000	+	0x0010	=	0x00_1FBF_ A010	Core-Local Configuration.
0x00_1FBF_8	+	0x2000	+	0x0018	=	0x00_1FBF_ A018	Core-Other Addressing.
0x00_1FBF_8	+	0x2000	+	0x0020	=	0x00_1FBF_ A020	Core-Local Reset Exception Base.
0x00_1FBF_8	+	0x2000	+	0x0028	=	0x00_1FBF_ A028	Core-Local Identification.

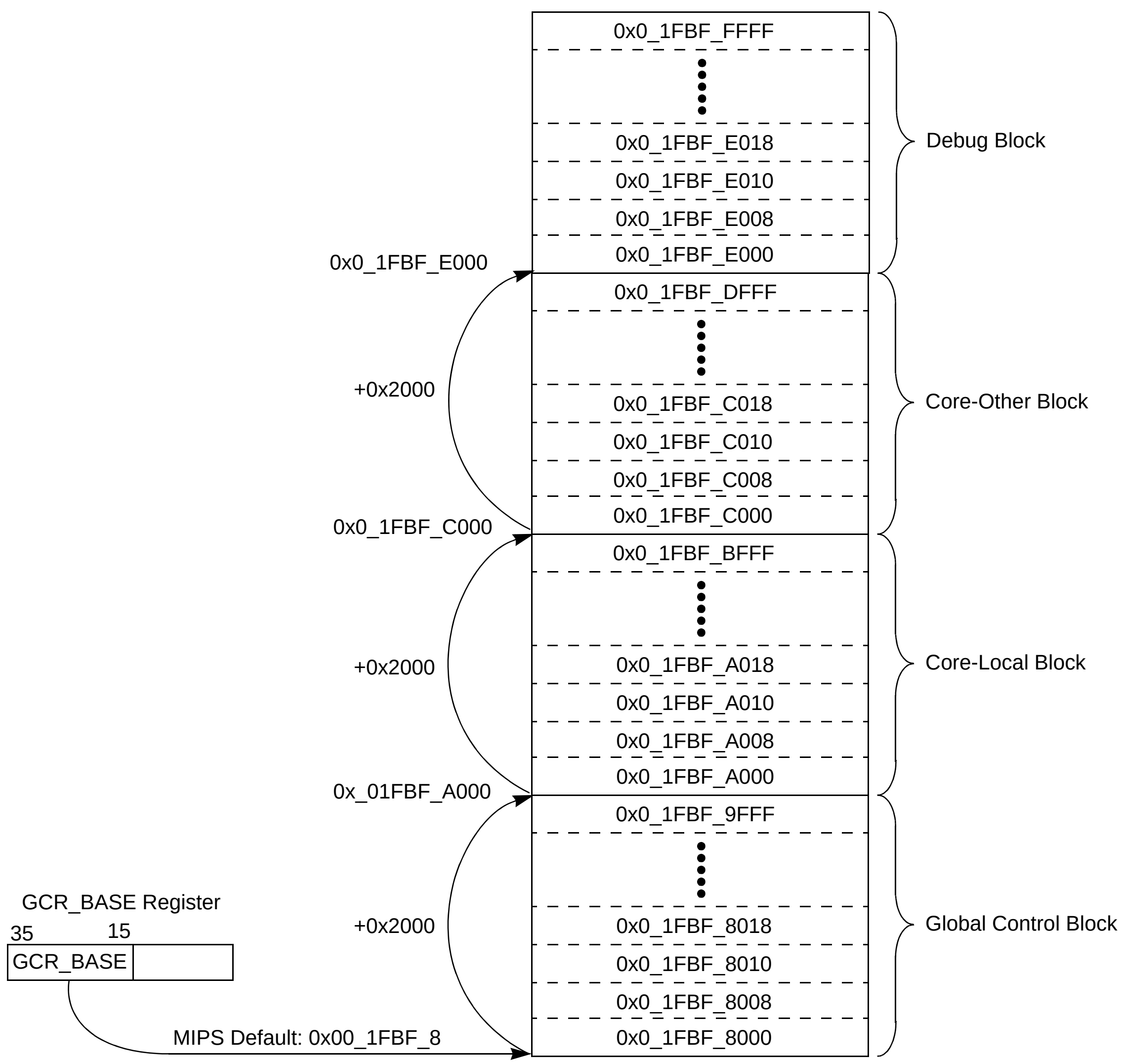
Table 6.4 Absolute Address of Individual Core-Local Block Registers*(continued)*

MIPS Default Base		Core-Local Block Offset		Core-Local Register Offset		Absolute Physical Address	Global Control Register
0x00_1FBF_8	+	0x2000	+	0x0030	=	0x00_1FBF_ A030	Core-Local Reset Exception Extended Base.
0x1FBF_8	+	0x2000	+	0x0040	=	0x00_1FBF_ A040	TCID 0 Priority.

The Core-Other block would be accessed in the same manner, just with a different (Core-Other) block offset (0x4000).

This concept is described in Figure 6.1 below. For simplicity, the MIPS default value is used for the GCR base address.

Figure 6.1 CM2 Register Addressing Scheme Using the MIPS Default in GCR_BASE



6.3 CM2 Programming

This section provides programming examples based on the capability of the CM2 register set. Some topics described are:

- Section 6.3.1, "40-bit Physical Address Support"
- Section 6.3.2, "L2 Cache Prefetcher"
- Section 6.3.3, "Verifying Overall System Configuration"
- Section 6.3.4, "Requestor Access to GCR Registers"
- Section 6.3.5, "CM2 Interface Ports"
- Section 6.3.6, "Setting the CM2 Register Block Base Address"
- Section 6.3.7, "Address Regions"
- Section 6.3.8, "Address Map Programming Example"
- Section 6.3.9, "Core-Local GCRs"
- Section 6.3.10, "Core-Other GCRs"
- Section 6.3.11, "Accessing Another Cores CM2 GCR Registers"
- Section 6.3.12, "Coherency Domains"
- Section 6.3.13, "L2-Only SYNC Operation"
- Section 6.3.14, "Handling of Addresses Not Mapped to a Defined Region"
- Section 6.3.15, "Setting the Cache Coherency Attributes for Default Memory Transfers"
- Section 6.3.16, "In-Flight L1 and L2 Cache Operations"
- Section 6.3.17, "MIPS System Trace"
- Section 6.3.18, "Error Processing"
- Section 6.3.19, "Custom GCR Implementation"
- Section 6.3.20, "Attribute-Only Regions"

6.3.1 40-bit Physical Address Support

The P6600 Multiprocessing System (MPS) supports a 40-bit physical address (PA). The 40-bit address allows for seamless integration with other IP with similar addressing capability.

All ‘base address’ registers in the CM2 register space have been extended to include a second register used to store bits 32 through 39 of the 40-bit address. Table 6.5 lists those new CM2 registers that have been added to support the 40-bit address. Note that all register addresses are relative to the Global Control Block offset.

Table 6.5 Registers Used to Support the 40-bit Physical Address

Register Address	Name
0x000C	GCR Base Upper Register (*GCR_BASE_UPPER*). This register works in conjunction with the (*GCR_BASE*) register at 0x0008 to store upper address bits 35:32. Note that the GCR_BASE extends to only 36 bits instead of 40 bits.
0x0054	Global CM2 Error Address Upper Register (*GCR_ERROR_ADDR_UPPER*). This register works in conjunction with the (*GCR_ERROR_ADDR*) register at 0x0050 to store upper address bits 39:32.
0x0064	GCR Custom Base Upper Register (*GCR_CUSTOM_BASE_UPPER*). This register works in conjunction with the (*GCR_CUSTOM_BASE*) register at 0x0060 to store upper address bits 39:32.
0x0074	Global L2 only Sync Upper Register (*GCR_L2_ONLY_SYNC_BASE_UPPER*). This register works in conjunction with the (*GCR_L2_ONLY_SYNC_BASE*) register at 0x0070 to store upper address bits 39:32.
0x0084	Global Interrupt Controller Base Address Upper Register (*GCR_GIC_BASE_UPPER*). This register works in conjunction with the (*GCR_GIC_BASE*) register at 0x0080 to store upper address bits 39:32.
0x008C	Cluster Power Controller Base Address Upper Register (*GCR_CPC_BASE_UPPER*). This register works in conjunction with the (*GCR_CPC_BASE*) register at 0x0088 to store upper address bits 39:32.
0x0094	CM2 Region0 Base Address Upper Register (*GCR_REG0_BASE_UPPER*). This register works in conjunction with the (*GCR_REG0_BASE*) register at 0x0090 to store upper address bits 39:32.
0x009C	CM2 Region0 Address Mask Upper Register (*GCR_REG0_MASK_UPPER*). This register works in conjunction with the (*GCR_REG0_MASK*) register at 0x0098 to store upper address bits 39:32.
0x00A4	CM2 Region1 Base Address Upper Register (*GCR_REG1_BASE_UPPER*). This register works in conjunction with the (*GCR_REG1_BASE*) register at 0x00A0 to store upper address bits 39:32.
0x00AC	CM2 Region1 Address Mask Upper Register (*GCR_REG1_MASK_UPPER*). This register works in conjunction with the (*GCR_REG1_MASK*) register at 0x00A8 to store upper address bits 39:32.
0x00B4	CM2 Region2 Base Address Upper Register (*GCR_REG2_BASE_UPPER*). This register works in conjunction with the (*GCR_REG2_BASE*) register at 0x00B0 to store upper address bits 39:32.
0x00BC	CM2 Region2 Address Mask Upper Register (*GCR_REG2_MASK_UPPER*). This register works in conjunction with the (*GCR_REG2_MASK*) register at 0x00B8 to store upper address bits 39:32.
0x00C4	CM2 Region3 Base Address Upper Register (*GCR_REG3_BASE_UPPER*). This register works in conjunction with the (*GCR_REG3_BASE*) register at 0x00C0 to store upper address bits 39:32.
0x00CC	CM2 Region3 Address Mask Upper Register (*GCR_REG3_MASK_UPPER*). This register works in conjunction with the (*GCR_REG3_MASK*) register at 0x00C8 to store upper address bits 39:32.
0x0194	CM Attribute-Only Region0 Base Address Upper Register (*GCR_REG0_ATTR_BASE_UPPER*). This register works in conjunction with the (*GCR_REG0_ATTR_BASE*) register at 0x0190 to store upper address bits 39:32.
0x019C	CM Attribute-Only Region0 Address Mask Upper Register (*GCR_REG0_ATTR_MASK_UPPER*). This register works in conjunction with the (*GCR_REG0_ATTR_MASK*) register at 0x0198 to store upper address bits 39:32.
0x01A4	CM Attribute-Only Region1 Base Address Upper Register (*GCR_REG1_ATTR_BASE_UPPER*). This register works in conjunction with the (*GCR_REG1_ATTR_BASE*) register at 0x01A0 to store upper address bits 39:32.
0x01AC	CM Attribute-Only Region1 Address Mask Upper Register (*GCR_REG1_ATTR_MASK_UPPER*). This register works in conjunction with the (*GCR_REG1_ATTR_MASK*) register at 0x01A8 to store upper address bits 39:32.

Table 6.5 Registers Used to Support the 40-bit Physical Address *(continued)*

Register Address	Name
0x0214	CM Attribute-Only Region2 Base Address Upper Register (*GCR_REG2_ATTR_BASE_UPPER*). This register works in conjunction with the (*GCR_REG2_ATTR_BASE*) register at 0x0210 to store upper address bits 39:32.
0x021C	CM Attribute-Only Region2 Address Mask Upper Register (*GCR_REG2_ATTR_MASK_UPPER*). This register works in conjunction with the (*GCR_REG2_ATTR_MASK*) register at 0x0218 to store upper address bits 39:32.
0x0224	CM Attribute-Only Region3 Base Address Upper Register (*GCR_REG3_ATTR_BASE_UPPER*). This register works in conjunction with the (*GCR_REG3_ATTR_BASE*) register at 0x0220 to store upper address bits 39:32.
0x022C	CM Attribute-Only Region3 Address Mask Upper Register (*GCR_REG3_ATTR_MASK_UPPER*). This register works in conjunction with the (*GCR_REG3_ATTR_MASK*) register at 0x0228 to store upper address bits 39:32.

6.3.2 L2 Cache Prefetcher

The coherence manager in the P6600 MPS contains an L2 prefetcher used to enhance L2 performance. The L2 prefetcher contains the following features.

- Improves memcopy/memset performance
- Recognizes streams with strides of +/-1 and prefetches ahead
- Increases size of prefetch window until requests for that stream hits in L2
- Up to 16 streams can be tracked simultaneously
- Tracks data fetches. GCR bit turns on prefetching for Instructions fetches
- Prefetches will be throttled when CM2.5 resources run low
- L2 prefetcher does not prefetch beyond an O/S page

The L2 prefetcher monitors requests from the cores and IOCU's and detect strides +/- 1 that miss in L2. It then issues a prefetch read for subsequent cachelines and regulates the amount of prefetching based on hit/miss and strides of requests in same stream.

The L2 prefetcher contains a series of prefetch trackers. Each prefetch tracker tracks a particular request stream based on the address. The output of each prefetch tracker is input to an arbiter which selects the prefetch request to forward. Each prefetch tracker maintains its own prefetch window, which is defined as the area between the last demanded address and the prefetch limit (the point after which the prefetcher cannot access).

The L2 prefetcher is controlled using the following two registers. Refer to the Shared register section for more information.

Table 6.6 Registers Used to Support L2 Prefetcher

Register Address	Name
0x0300	L2 Prefetcher control register. (*GCR_L2_PFT_CONTROL*). Provides L2 prefetch control.
0x0308	L2 Prefetcher control register 2. (*GCR_L2_PFT_CONTROL*_B). Provides additional L2 prefetch control.

6.3.3 Verifying Overall System Configuration

At build-time, the developer selects the number of cores in the system, the number of I/O coherency units (IOCU's), and the number of address regions. When the device is built, these values are hard-wired into the *Global Configuration* register at offset address 0x0000. Reading this register provides the following information:

- Bits 7:0 — Number of cores in the system (up to 6)
- Bits 11:8 — Number of IOCU's (1)
- Bits 19:16 — Number of address regions

6.3.4 Requestor Access to GCR Registers

The CM2 allows up to seven requestor's in a system. A requestor can be either a core or an IOCU. The P6600 core allows up to 7 requestors in a multiprocessing system; six cores and one IOCU.

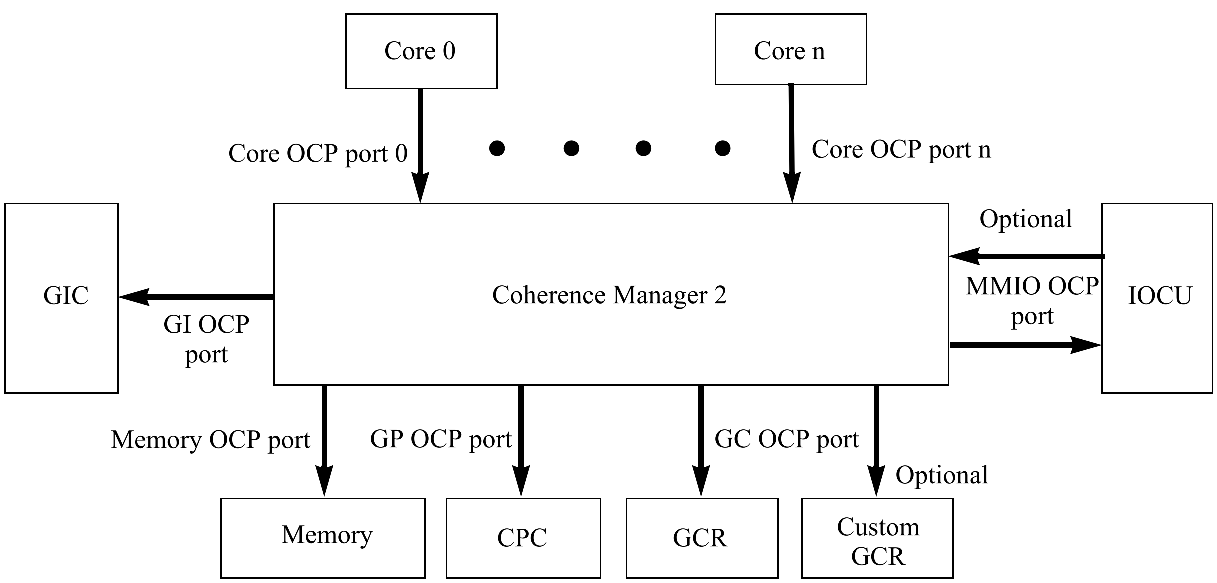
The requestor's may not have unrestricted access to the CM2 registers. During boot time, software determines which requestor's are provided access to the CM2 registers by programming the *CM2_ACCESS_EN* field of the *Global CSR Access Privilege* register located at offset 0x0020. Each bit in this field corresponds to a specific requestor.

The MIPS default for this field is 0xFF, meaning that all requestor's in the system have access to the CM2 register set. To disable access to the registers for a particular requestor, software need only clear the corresponding bit of this field to zero and all write requests to the CM2 registers by that requestor will be ignored.

6.3.5 CM2 Interface Ports

The CM2 contains numerous ports that allow the various system peripherals to communicate with the CM2. The ports connected to the CM2 are shown in Figure 6.2. The P6600 Multiprocessing System can have up to 6 cores.

Figure 6.2 Interface Ports of the CM2



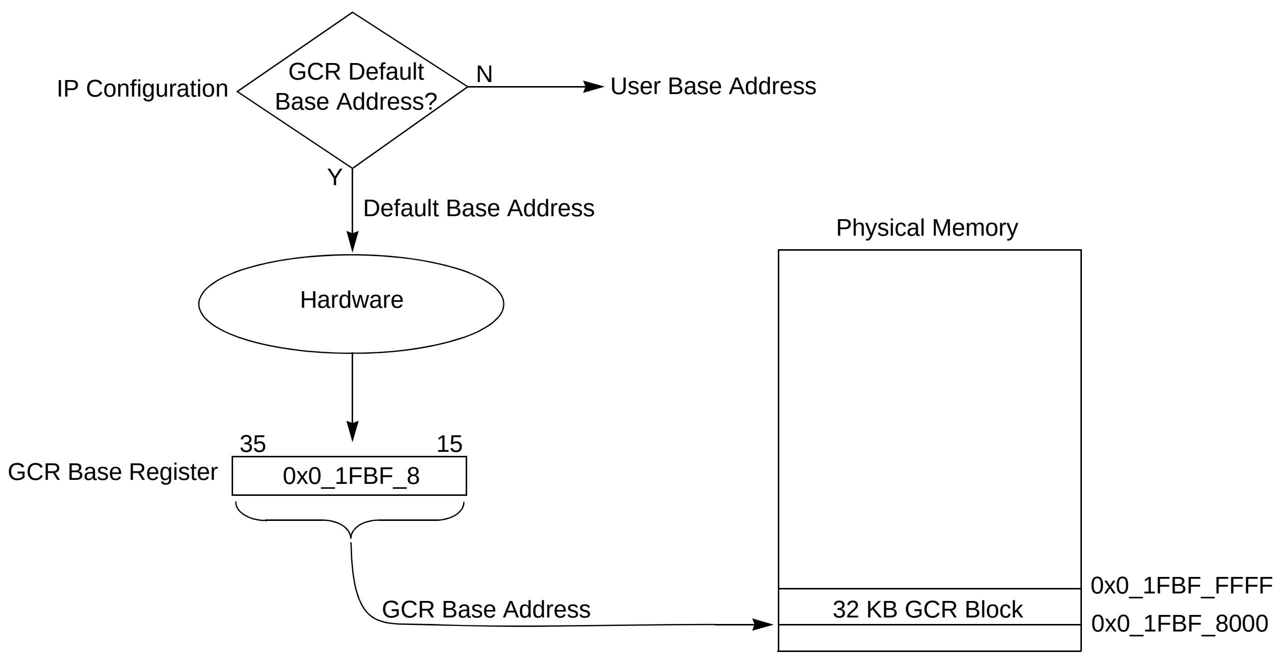
6.3.6 Setting the CM2 Register Block Base Address

As shown in Table 6.1 above, the CM2 register map contains four contiguous 8K blocks and can be located anywhere within physical memory. During IP configuration, the user can select the option to use the MIPS default base address of 0x0_1FBF_8, or they can select any 32 KB location in memory to locate the CM2 registers.

This decision determines how the 17-bit GCR_BASE field is programmed. If the MIPS default base address option is selected, a value of 0x0_1FBF_8 is loaded into this field. If the user selects their own base address, then that address is programmed into the GCR_BASE field. Refer to Section 6.4.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information. In addition to the value in the GCR_BASE field, the user can also select whether this field is R/W or RO during IP configuration.

The following example shows the assignment of the CM2 GCR registers in memory using the MIPS default address. Note that the physical address is shown in this diagram. During actual programming, the programmer may use the virtual address associated with a physical address of 0x0_1FBF_8 to address the GCR block. The virtual address is provided prior to address translation and will be different from the resulting physical address. Refer to Chapter 3 of this manual for more information on virtual to physical address translation.

Figure 6.3 Mapping the CM2 Registers in Physical Memory Using the MIPS Default Value



6.3.7 Address Regions

The CM2 divides the address space into two types of regions:

- Fixed-size regions
- Variable-size regions

6.3.7.1 Fixed-Size Regions

Fixed-size regions are those that have a fixed size in memory. These include:

- GCR Base; contains the global, core-local, core-other, and debug register blocks, fixed at 32 KB.
- GIC (global interrupt controller) address space, fixed at 128 KB
- CPC (cluster power controller) address space, fixed at 32 KB
- Custom GCR address space, fixed at 64 KB

The 32 KB GCR Base region is further divided into four 8 KB blocks as described in Table 6.1. Refer to Section 6.3.6, "Setting the CM2 Register Block Base Address" for more information on setting the base address in memory for the CM2 register block.

The GIC region is fixed at 128 KB. Refer to Section 6.4.3.1, "Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)" for more information on programming the base address for the GIC interface.

The CPC region is fixed at 32 KB. Refer to Section 6.4.3.3, "Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)" for more information on programming the base address for the CPC interface.

The Custom GCR region is fixed at 64 KB. Refer to Section 6.4.2.13, "GCR Custom Base Register (GCR_CUSTOM_BASE Offset 0x0060)" for more information on programming the base address for the Custom GCR interface.

6.3.7.2 Variable-Size Regions

The P6600 multiprocessing system may provide four programmable variable size address regions for mapping the IOCU's and memory. The number of regions is determined at IP configuration time. If an IOCU is not present, then the regions registers are not used. The number of regions implemented is determined as follows.

Table 6.7 Setting the Number of Regions

ADDR_REGIONS Field	Number of Regions	Region Assignments
0x0	0	None (typically used when there is no IOCU).
0x4	4	4 standard regions.
0x6	6	4 standard regions and 2 attribute-only regions.
0x8	8	4 standard regions and 4 attribute-only regions.

For more information, refer to the ADDR_REGIONS field in bits 19:16 of the Section 6.4.2.1, "Global Config Register (GCR_CONFIG Offset 0x0000)". For more information on the attribute-only regions, refer to Section 6.3.20.

Each region is controlled by a corresponding base and mask register as described below. These registers are used to determine not only the location and size of the memory space, but also whether this space is mapped to an IOCU or to memory. In addition, the cache coherency attributes (CCA) for each region can be defined as described in Section 6.3.7.6, "Setting the Cache Coherency Attributes for Region Memory Transfers".

In a MIPS core, mapped addresses are processed by the memory management unit (MMU) and the cache coherency attributes for a given memory page are determined. In this case, the CCA corresponds to both the L1 and L2 caches. In some situations it may be advantageous to have the CCA of the L2 different from that of the L1 cache. In this case, software can use the *CCA_Override_Value* field of each *Region Address Mask* register to set the CCA for the L2 cache. This changes the attributes of the cache from what was originally assigned by the core.

The CM2 provides four base address and four address mask registers for controlling variable-size address regions 0 through 3. These regions control how some transactions are routed by the CM2. The possible routing options for requests that map to these variable-size regions are:

- To/From Memory via the CM2's system memory OCP port
- To/From the IOCU's via the CM2's MMIO OCP port for Memory-Mapped I/O (in hardware I/O coherent systems only)

Refer to Section 6.4.3.5, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" and Section 6.4.3.7, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information on these registers.

6.3.7.3 Address Region Priorities

The priority for the region decode is as follows:

1. GCR (highest priority)
2. Custom GCR
3. CPC
4. GIC
5. IOCU
6. Programmed MMIO regions
7. Programmed memory regions
8. *CM2_DEFAULT_TARGET* (lowest priority)

The above priority allows for large memory regions to be defined with small IOCU regions carved out. Note that these regions can overlap as described in Section 6.3.7.8, "Overlapping Regions".

6.3.7.4 Defining the Base Address Location and Size for Each Region

The address map is programmable through a set of registers located in the GCR as summarized below. Up to 8 variable-size programmable regions can be implemented. When an IOCU is present (i.e., hardware I/O Coherence is implemented), these regions determine if requests are routed to memory or to the IOCU via the CM2's MMIO port. The regions can also be used with or without an IOCU for the CCA Override feature as described in Section 6.3.15 "Setting the Cache Coherency Attributes for Default Memory Transfers".

- The *GCR Base Register* defines the address base of the GCR region. The GCR region has a fixed size of 32 KB (see Table 6.20), hence no corresponding Mask register is required. Note that this region must reside on a 32 KB boundary.
- The *Cluster Power Controller Base Address Register* defines the address base of the CPC address region. This CPC region may be disabled via the *CPC_EN* bit in that register. When enabled, the CPC address region has a fixed size of 32 KB (see Table 6.38), hence no corresponding Mask register is required. Note that this region must reside on a 32KB boundary.

- The *Global Interrupt Controller Base Address Register* defines the address base of the GIC address region. This GIC region may be disabled via the *GIC_EN* bit in that register. When enabled, the GIC address region has a fixed size of 128 KB (see Table 6.36), hence no corresponding Mask register is required. Note that this region must reside on a 128 KB boundary.

- The *CM2 Region [0-3] Base Address Registers* define the address base for each of the four programmable regions. The regions have a programmable base address and a programmable size that is selected via the corresponding Mask register.

- The *CM2 Region [0-3] Address Mask Registers* define the size for each of the four programmable regions. These registers work in conjunction with the corresponding *CM2 Region [0-3] Base Address Registers* to configure a given region.

- The *Custom GCR Base Register* defines the address base of the Custom GCR region. This region defines the location of registers that are implemented by the user. This region may be disabled via the *GGU_EN* bit in the *Custom GCR Base Register*. When enabled, the Custom GCR region has a fixed size of 64 KB (see Table 6.31), hence no corresponding Mask register is required. Note that this region must reside on a 64 KB boundary.

As described above, the base of each region is defined in the corresponding CM2 *Region [0,1,2,3] Address Base Register* (see Table 6.40), and the size of the region is defined in the corresponding *CM2 Region [0,1,2,3] Address Mask Register* (see Table 6.42). Because a base/mask scheme is used, the base must be located on a boundary of its size. A region can be sized from 64K to the entire 32-bit address space.

Table 6.8 Setting the Base Address for the CM2 Peripheral Devices

Block	Register Name	Offset Address	Field Name	Bits	Description
GCR	GCR_BASE	0x0008	GCR_BASE_ADDR	35:15	Sets the base address of the GCR registers. This field has a fixed size of 32 KB.
Custom GCR	GCR_CUSTOM_BASE	0x0060	CUSTOM_ BASE	39:16	Sets the base address of the Customer GCR registers. This field has a fixed size of 64 KB.
GIC	GCR_GIC_BASE	0x0080	GIC_BASE_ADDR	39:17	Sets the base address of the GIC. This field has a fixed size of 128 KB.
CPC	GCR_CPC_BASE	0x0088	CPC_BASE_ADDR	39:15	Sets the base address of the CPC. This field has a fixed size of 32 KB.
Region 0	GCR_REG0_BASE	0x0090	REGION0_BASE_ADDR	39:16	Sets the base address of region 0 in memory. Minimum size is 64 KB.
	GCR_REG0_MASK	0x0098	REGION0_BASE_MASK	39:16	Sets the size of region 0 in memory.
Region 1	GCR_REG1_BASE	0x00A0	REGION1_BASE_ADDR	39:16	Sets the base address of region 1 in memory. Minimum size is 64 KB.
	GCR_REG1_MASK	0x00A8	REGION1_BASE_MASK	39:16	Sets the size of region 1 in memory.
Region 2	GCR_REG2_BASE	0x00B0	REGION2_BASE_ADDR	39:16	Sets the base address of region 2 in memory. Minimum size is 64 KB.
	GCR_REG2_MASK	0x00B8	REGION2_BASE_MASK	39:16	Sets the size of region 2 in memory.
Region 3	GCR_REG3_BASE	0x00C0	REGION3_BASE_ADDR	39:16	Sets the base address of region 3 in memory. Minimum size is 64 KB.
	GCR_REG3_MASK	0x00C8	REGION3_BASE_MASK	39:16	Sets the size of region 3 in memory.

As described above, some of the blocks are a fixed size, hence there is no corresponding Mask register. Since the GCR, GIC, and CPC blocks each contain a dedicated Base Address register, the Region 0 - 3 registers are used to access the memory and IOCU peripherals.

6.3.7.5 Defining the Target Device

Each *CM2 Region Address Mask* register contains a field that determines how the CM2 routes requests whose address matches the corresponding region. As defined in the *CM2_REGION_TARGET* field, the transaction may be routed to memory or to an I/O device via the CM2's MMIO port and IOCU. A region may be disabled by setting the *CM2_REGION_TARGET* in the corresponding *CM2 Region Address Mask* register to 0.

The *CM2_DEFAULT_TARGET* field in the *GCR Base Register* determines how to route the requests that don't match any of the defined regions. Refer to Section 6.3.14, "Handling of Addresses Not Mapped to a Defined Region" for more information.

6.3.7.6 Setting the Cache Coherency Attributes for Region Memory Transfers

As described in Section 6.3.6 "Setting the CM2 Register Block Base Address", the P6600 core provides a CCA override capability that allows the CCA's for the L2 cache to be different from those of the L1 data cache.

This capability can be achieved via the CCA override feature in the CM2 Region Address Map Registers listed in Table 6.8. Software can establish up to 4 address map regions by programming the *CM2 Region Base Register 0-3* and *CM2 Region Mask Register 0-3*.

Programming the CCA

Each region has the *CCA_Override_Enable* and *CCA_Override_Value* fields which can be used to set the CCA for transactions on the system memory OCP port. If the *CCA_Override_Enable* field is set to 1 for a given region and the corresponding *CM2_TARGET* field in bits 1:0 is set to memory (0x1), then transactions that map to that region and proceed to the system memory port will have a CCA value set to the corresponding *CCA_Override_Value* for that region. This field also determines the CCA value driven to system memory.

Any valid CCA value can be programmed into *CCA_Override_Value*, but because the L2 does not process coherent CCA's, a value of CWB (5) or CWBE (4) is automatically changed to WB (3) by the CM2 before being driven on the system memory OCP port. The encoding of the *CCA_Override_Value* field is identical to that shown in Table 6.9.

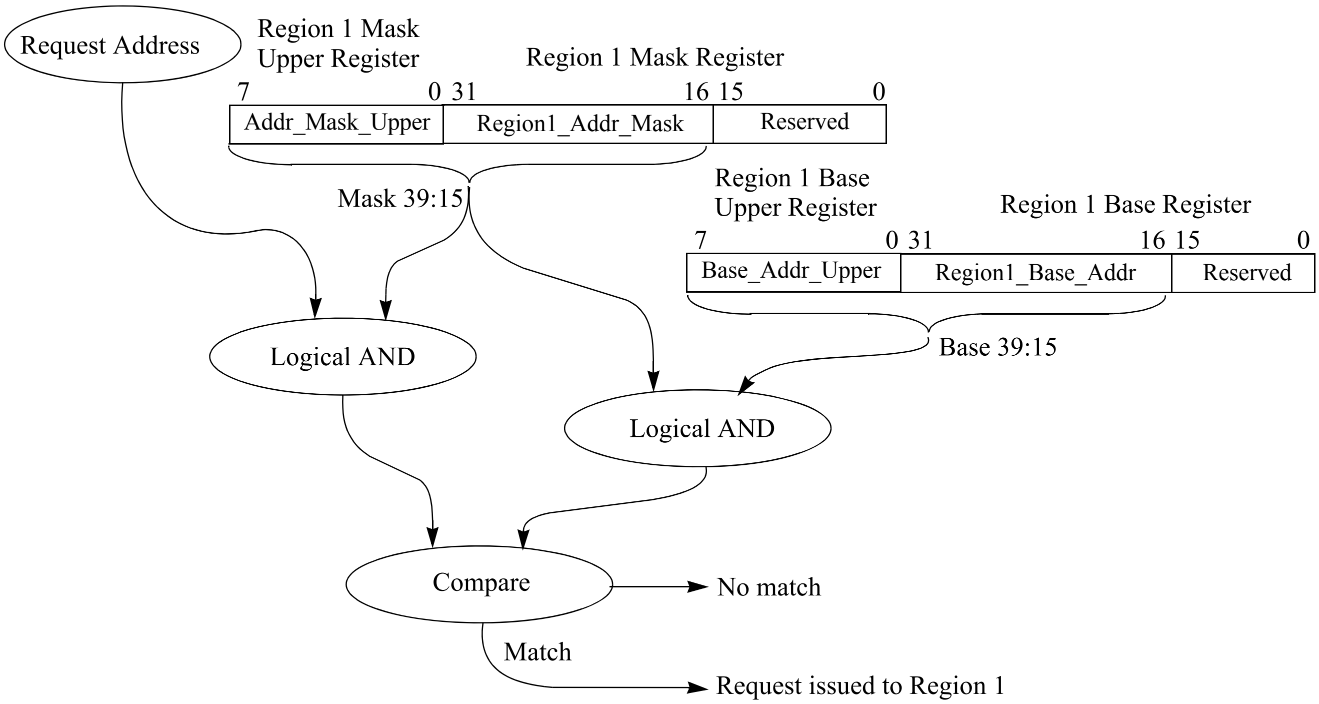
6.3.7.7 Issue Request Protocol and Region Masking

The CM2 contains four region mask registers used to set the size of a given region. These mask registers work in conjunction with their corresponding base address registers as shown in Table 6.8. The requesting address is logically ANDed with the value in the selected *Region Address Mask* register. At the same time, the value in the corresponding *REGION_BASE_ADDR* field is compared to the value in the *Region Address Mask* register. If both outputs match, the request is routed to this region.

When performing a comparison on a 40-bit address, the requesting address in the *CM2_REGION1_BASE_ADDR* and *CM2_REGION1_BASE_ADDR_UPPER* registers are compared to the value in the *CM2_REGION1_ADDR_MASK* and *CM2_REGION1_ADDR_MASK_UPPER* registers. If there is a match, the requesting address is routed to region 1. This concept is shown in Figure 6.4.

The only allowed values in this register are contiguous sets of leading 0x1's. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFFF0 is allowed, but the value 0xFFEF is not allowed).

Figure 6.4 Mapping a Request to Region 1 Using the Region 1 Base and Mask Registers

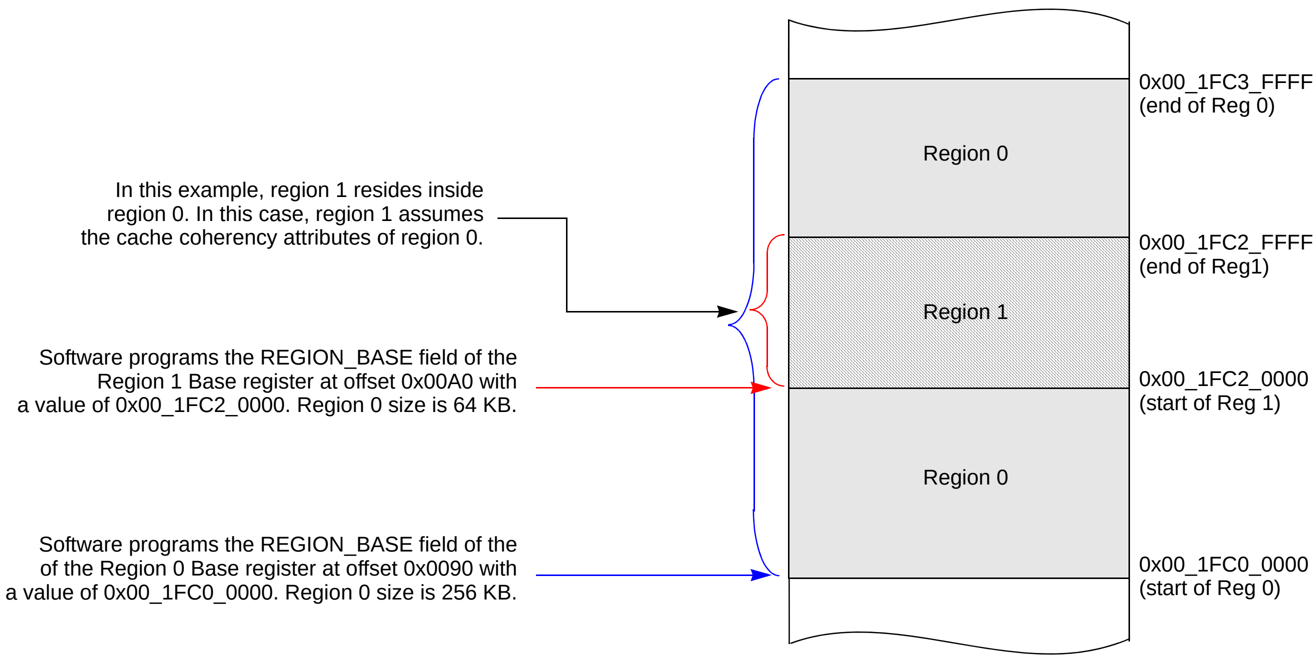


6.3.7.8 Overlapping Regions

Since overlapping regions are supported, it is possible that an address maps to more than one region. In this case, the CCA override enable and value are used from the lowest numbered region mapped to memory. For example, if an address matches both *CM2 Region Base/Mask Register 0* and *CM2 Region Base/Mask Register 1*, and both regions 0 and 1 are mapped to Memory (*CM2_REGION_TARGET* is set to 1 in both *CM2 Region Mask Register 0* and *1*), then the values of *CCA_Override_Enable* and *CCA_Override_value* in CM2 Region Mask Register 0 is used to determine the CCA value driven on the system memory OCP Port.

This concept is shown in Figure 6.5. In this example, region 1 is a 64 KB space located inside the larger 256 KB region 0.

Figure 6.5 Example of Overlapping Regions

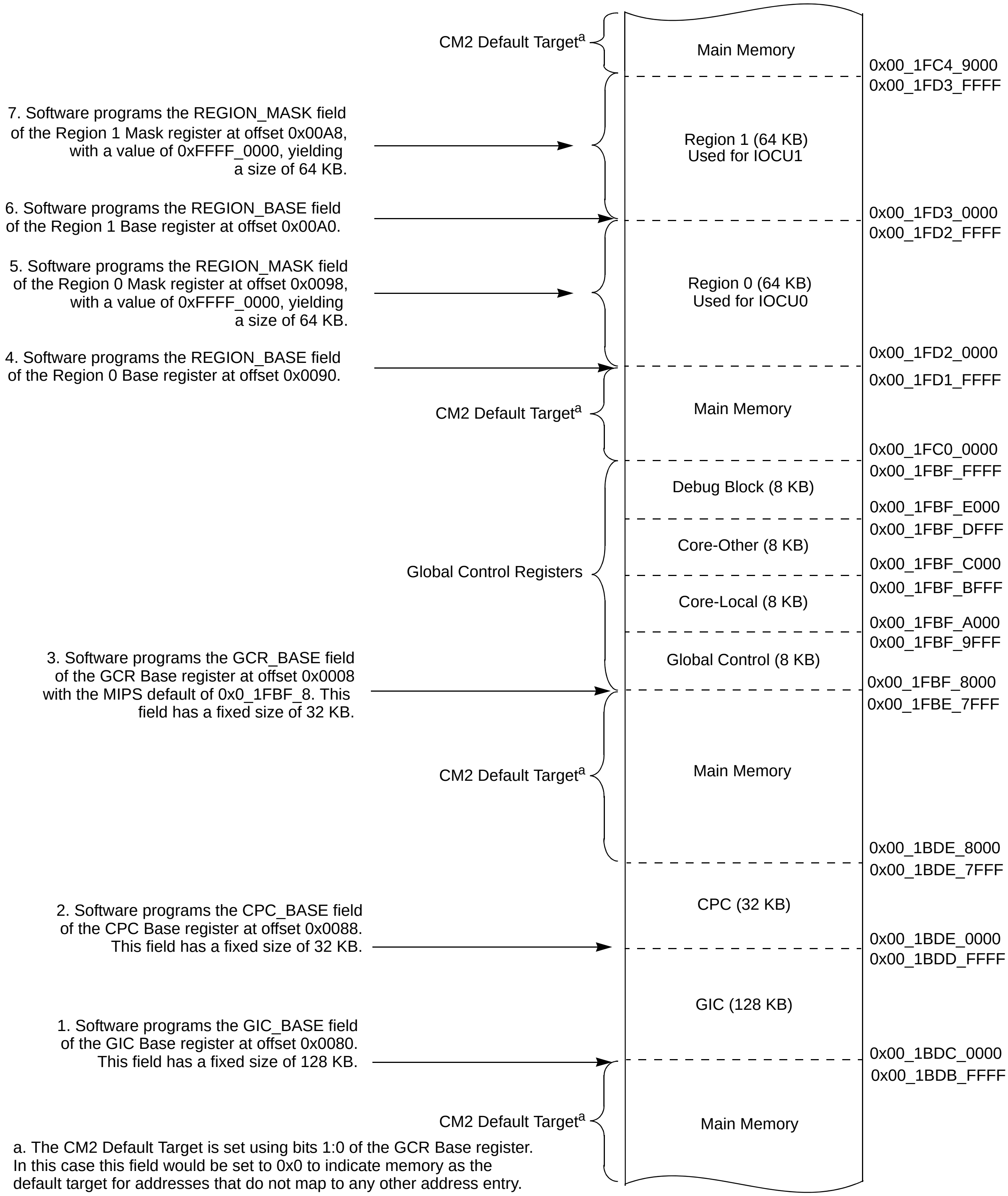


When overriding a CCA value, only the CCA driven to the system memory OCP is affected. Otherwise, the functionality of the transaction within the CM2 is based on the original CCA. When the CM2 is programmed to override the CCAs for an address region, all accesses to that region including speculative reads and write-backs (explicit or implicit) from the L1 are overridden. Transactions that are never mapped to regions, such as Legacy Syncs, CohCompletionSyncs or L2/L3 CacheOps are unaffected by the CCA override functionality.

6.3.8 Address Map Programming Example

This subsection provides an example of memory mapping for all of the aforementioned regions at different locations using the MIPS default base address. The memory map for this example is shown in Figure 6.6.

Figure 6.6 Address Map Programming Example



The following programming sequence is used to configure the memory map as shown in Figure 6.6 above.

1. Software programs the *GIC_BASE* field of the *GIC Base* register located at offset 0x0080 with a value of 0x1BDC. This sets the base address of the GIC registers. This block has a fixed size of 128 KB. Refer to bits 31:17 in Section 6.4.3.1, "Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)" for more information. Note that this block must reside on a 128 KB boundary.

2. Software programs the *CPC_BASE* field of the *CPC Base* register located at offset 0x0088 with a value of 0x1BDE_0. This sets the base address of the CPC registers. This block has a fixed size of 32 KB. Refer to bits 31:15 in Section 6.4.3.3, "Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)" for more information. Note that this block must reside on a 32 KB boundary.

3. Software programs the *GCR_BASE* field of the *GCR Base* register located at offset 0x0008 with a value of 0x1FBF_8. This sets the base address of the 32 KB block of GCR registers. This block is divided into four 8 KB subblocks that contain the Global, Core-Local, Core-Other, and Debug register blocks. Note that if the MIPS default address of 0x1FBF_8 is selected for the base address of the GCR registers during IP configuration, this field becomes read-only. In this case, hardware writes the default value of 0x1FBF_8 to this field. Refer to bits 31:15 in Section 6.4.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information.

4. Software programs the *REGION_BASE_ADDR* field of the *CM2 Region 0 Base* register located at offset 0x0090 with a value of 0x1FD2. This sets the base address of region 0 to 0x1FD2_0000. Refer to bits 31:16 in Section 6.4.3.5, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" for more information.

5. Software programs the *REGION_ADDR_MASK* field of the *CM2 Region 0 Address Mask* register located at offset 0x0098 with a value of 0xFFFF_0000. This sets the size of region 0 to 64 KB. Refer to bits 31:16 in Section 6.4.3.7, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information. Other values for this field could be 0xFFFE (128 KB), 0xFFFC (256 KB), etc.

6. Software programs the *REGION_BASE_ADDR* field of the *CM2 Region 1 Base* register located at offset 0x00A0 with a value of 0x1FD3. This sets the base address of region 1 to 0x1FD3_0000. Refer to bits 31:16 in Section 6.4.3.5, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" for more information.

7. Software programs the *REGION_ADDR_MASK* field of the *CM2 Region 1 Address Mask* register located at offset 0x00A8 with a value of 0xFFFF_0000. This sets the size of region 1 to 64 KB. Refer to bits 31:16 in Section 6.4.3.7, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information. Other values for this field could be 0xFFFE (128 KB), 0xFFFC (256 KB), etc.

8. Software programs the *CM2_DEFAULT_TARGET* field of the *GCR Base* register with a value of 2’b00, indicating that memory is the target device for addresses that do not map to any of the address blocks shown in Figure 6.6. Refer to bits 1:0 in Section 6.4.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information.

9. Software programs the *CM2_TARGET* field of the *CM2 Region 0 Address Mask* register located at offset 0x0098 with a value of 2’b10. This maps region 0 to IOCU0.

10. Software programs the *CM2_TARGET* field of the *CM2 Region 1 Address Mask* register located at offset 0x00A8 with a value of 2’b11. This maps region 1 to IOCU1.

6.3.9 Core-Local GCRs

The Core-Local GCR block contains the configuration and status registers for a given core. Each core has its own copy of Core-Local registers. A core can access its own Core-Local block to determine the programmable parameters for that core. Parameters include base address assignments for cache coherency attributes, reset exception base, boot exception vector mask, etc.

6.3.10 Core-Other GCRs

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

6.3.11 Accessing Another Cores CM2 GCR Registers

As shown in Table 6.1, the CM2 provides two blocks of registers.

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

Each core contains a copy of these registers. The Core-Local address space contains the GCR registers for that core. The Core-Other address space allows a core to access the GCR registers for another cores Core-Local GCR block.

As described in Section 6.3.6, these registers can be located anywhere in physical memory if this option is selected during IP configuration. If this option is not selected, the location of these registers are located at the MIPS default address of 0x00_1FBF_8000. Refer to Section 6.2 "Coherence Manager Address Map" and related subsection for more information on use of the MIPS default memory location.

The Core-Local block represents registers corresponding to that core. If a core wishes to modify the contents of its own set of CM2 GCR registers, it writes to the Core-Local block located at the address range shown in Table 6.1. If a core wishes to program the GCR registers of another core, it selects the core number and writes this value into the Core-Other Addressing register in its own Core-Local block at offset address 0x0018. The actual register in the other core to be written would use the corresponding offset in the Core-Other block shown in Table 6.1.

In a multiprocessor system, it is common for one core to boot up first, then have that core boot the other cores in the system. In the following example, assume core 0 is booted up first. Then core 0 is used to program the GCR registers in core 1. This example examines how core 0 would program the boot exception vector location for core 1. Note that this example uses the MIPS default addressing scheme. The programming sequence would be as follows:

1. Core 0 writes a value of 0x0001 to the *CORENUM* field (bits 31:16) of the *Core-Other Addressing* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018 in Table 6.3). This indicates that the register to be programmed corresponds to core 1. Refer to Section 6.5.2.3, "Core-Other Addressing Register" for more information.

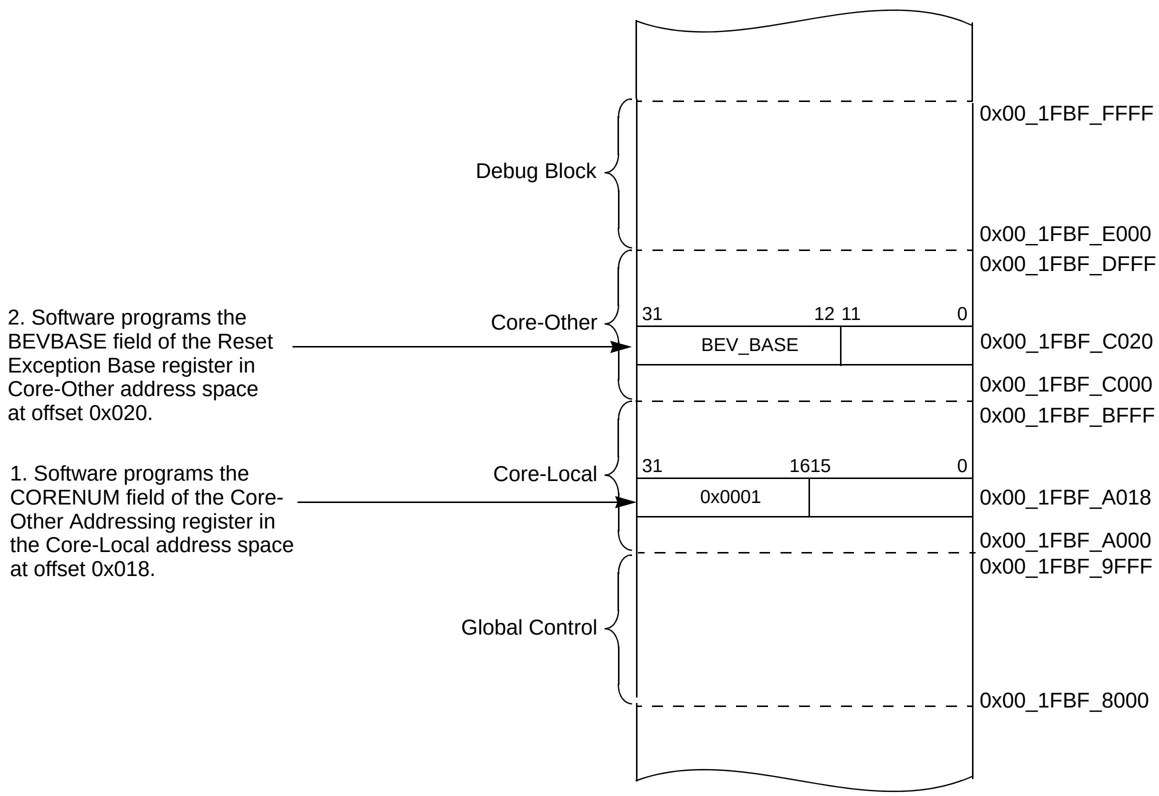
2. Core 0 writes the appropriate value into the *BEVEXCBase* field (bits 31:12) of the *Reset Exception Base* register located in the Core-Other block at offset 0x0020 (physical address of 0x00_1FBF_C020 in Table 6.4). Because core 0 is setting the BEV base value for core 1, as opposed to its own core, the write is done to the Core-Other address block. Refer to Section 6.5.2.4, "Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)" for more information.

Note that in addition to the *CORENUM* field in the *Core-Other Addressing* register used to indicate the number of the destination core as described in #1 above, a core can determine its own core number by reading the *CORENUM* field in its own *Core-Local Identification* register located at offset 0x0028 in Core-Local address space. Refer to Section 6.5.2.5, "Core Local Identification Register (GCR_Cx_ID Offset 0x0028)" for more information.

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

Since there is only one Core-Other block in Table 6.1, this means that when one core wants to access any of the other cores in the system, the register to be accessed always resides in the Core-Other block, regardless of the number of cores in the system. The state of the CORENUM field in the *Core-Other Addressing* register in that cores own Core-Local space determines which core the data will be written to. This concept is shown in Figure 6.7.

Figure 6.7 Core 0 Accessing the BEV_BASE GCR of Core 1



6.3.12 Coherency Domains

The CM2 provides the *COH_DOMAIN_EN* field in *Core-Local Coherence Control* register at offset 0x0008 for managing the coherency aspects of each requestor in the system. There is one register per core. A requestor can be either a core or an IOCU.

In the 8-bit *COH_DOMAIN_EN* field, each bit corresponds to one requestor. Setting a given bit in the *COH_DOMAIN_EN* field for the GCR local register corresponding to a given core puts that core into coherent mode. If the same bit in the *COH_DOMAIN_EN* is 0 for the GCR local register corresponding to a given core, then that core is not in coherence mode and will never issue a coherent request.

For example, if bit 1 of this field is set, then interventions from core 1 to core 0 are enabled and can occur. Note that 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.

Also note that if bit 1 of the *COH_DOMAIN_EN* field is set for the GCR local register corresponding to core 0, then software should also set bit 0 of the *COH_DOMAIN_EN* field for the GCR local register corresponding to core 1.

There is no need to program *COH_DOMAIN_EN* for the GCR local register corresponding to IOCUs.

Section 7.1.2, "Operating Level Transitions" in Chapter 7 of this manual provides examples of how this field is used to transition between coherency domains.

Figure 6.8 Encoding of COH_DOMAIN_EN Field — 2 or 4 Core Package

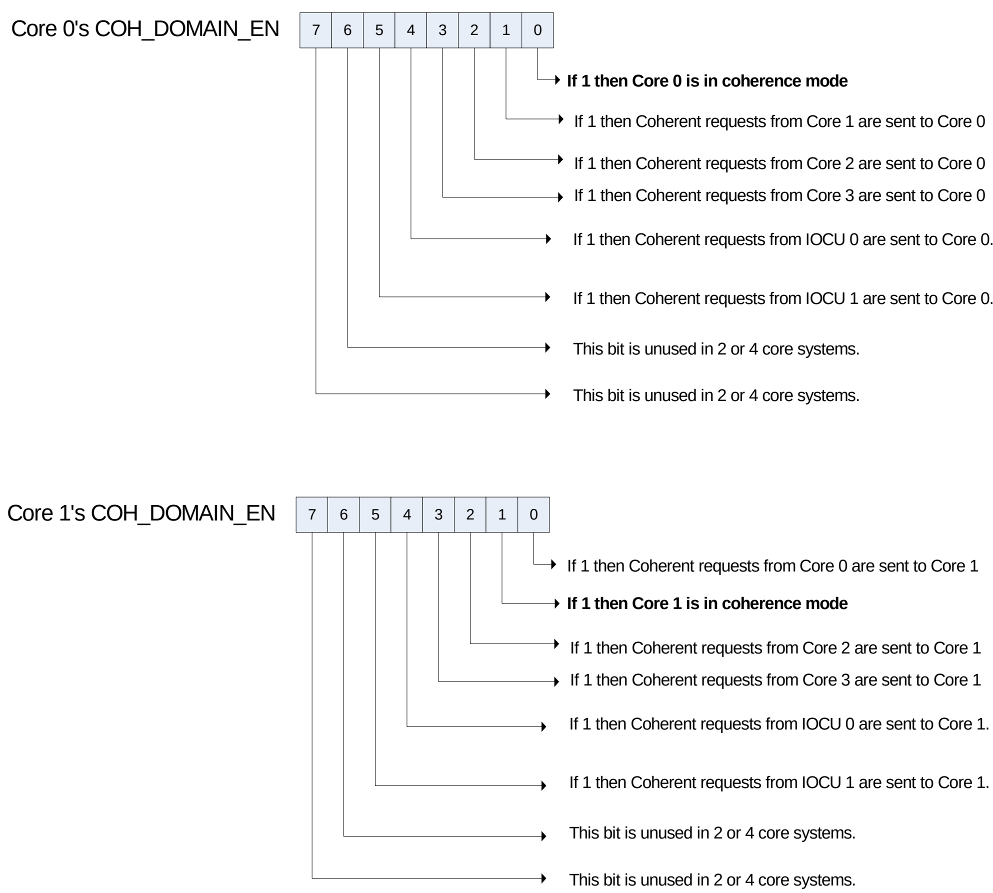
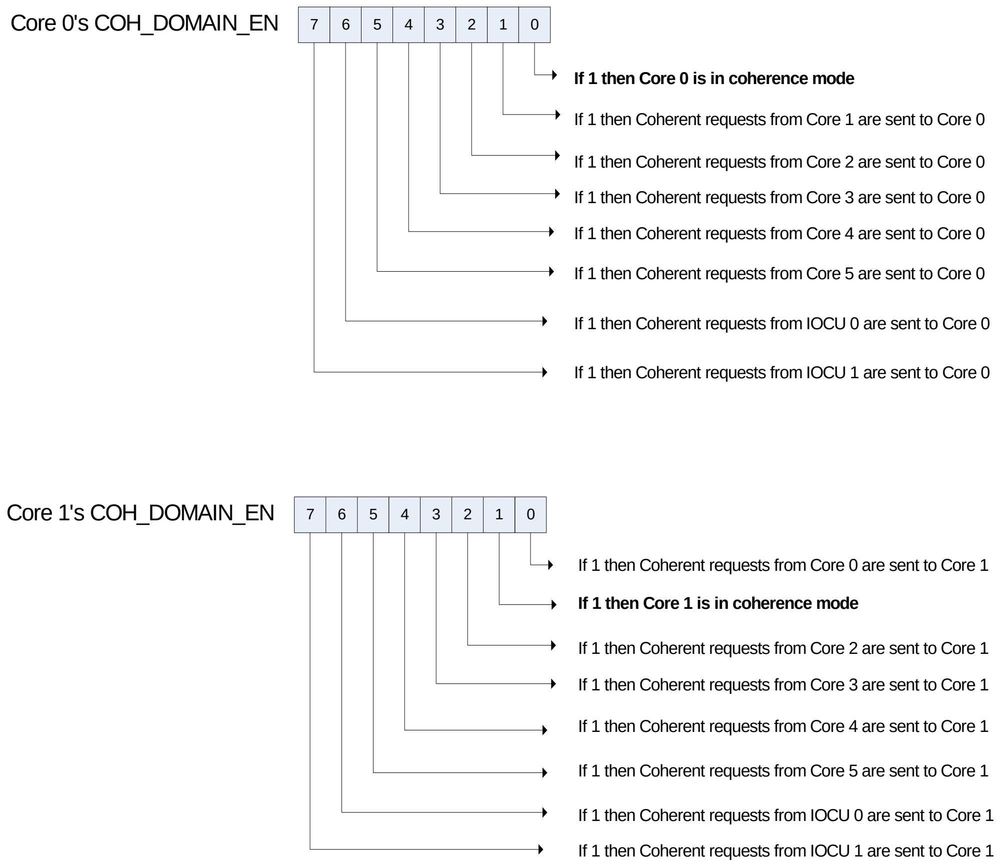


Figure 6.9 Encoding of COH_DOMAIN_EN Field — 6 Core Package



6.3.13 L2-Only SYNC Operation

In previous generation MIPS processors, the execution of a SYNC instruction would cause the entire core pipeline to stall until all read/write requests were completed. This included the L2 pipeline. After all instructions had been completed, a signal was sent to the L2 cache to continue. This caused a sometimes unnecessary stalling of the L2 cache.

The P6600 core provides a way to perform a SYNC operation on only the L2 cache. The core defines a fixed 4 KB address space for performing L2 only SYNC operations. The base address for the location of this fixed 4 KB segment is programmed using bits 31:12 of the *L2-Only Sync Base* register located at offset 0x0070.

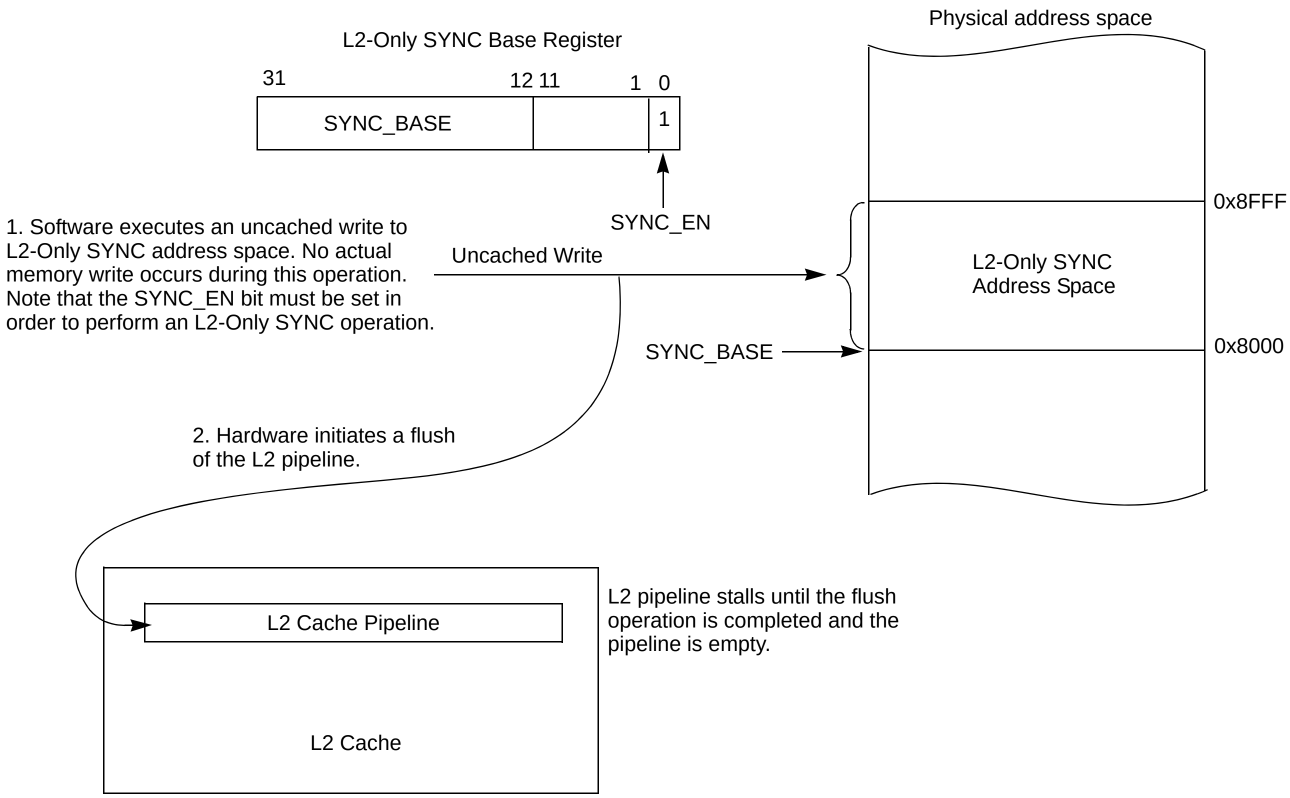
Bit 0 of the *L2-Only Sync Base* register enabled the L2-only SYNC function. If this bit is set, the CM2 treats an uncached write to anywhere within the 4 KB block as an L2-only SYNC. This operation does not write anything to memory, but rather just initiates the L2-only SYNC.

The L2-only SYNC provides a way for the software to ensure that subsequent uncached loads and stores from a core will not pass previous L2 cache operations, such as L2 cacheops.

Note that the L2-Only SYNC is not required, but it can be useful for optimizing performance. Since the L2-Only SYNC operation does not synchronize to the L1 caches, care should be taken to ensure correct system functionality.

As an example of how this operation works, assume the 4 KB block is located at offset address 0x8000 as shown in Figure 6.10.

Figure 6.10 Example of an L2-Only SYNC Operation



6.3.14 Handling of Addresses Not Mapped to a Defined Region

The CM2 handles transactions between the core and several devices as described in Figure 6.2.

For addresses that do not map to any of the defined address regions, these transactions can be mapped to either memory or one of the IOCU's as determined by the *CM2_DEFAULT_TARGET* field in bits 1:0 of the *GCR Base* register located at offset 0x0008. The default state of this field is determined by the value of the *SI_CM_Default_Target[1:0]* pins at reset, but can be changed by software at any point. Refer to Section 6.4.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information on the CM2_TARGET field.

Because programmable regions of the address map are disabled at reset, the value of *SI_CM_Default_Target[1:0]* determines whether the initial boot code upon power-up is fetched from the L2/Memory port or the MMIO port. For systems without an IOCU, *SI_CM_Default_Target[1:0]* should be set to 0 (memory) so that all non-coherent requests are routed to memory.

6.3.15 Setting the Cache Coherency Attributes for Default Memory Transfers

In previous generation MIPS processors, the cache coherency attributes (CCA) for the L1 and L2 caches were configured as one, and the CCA for the L2 cache could not be different from the CCA for the L1 data cache. The P6600 core provides a CCA override capability that allows the CCA's for the L2 cache to be different from those of the L1 data cache. For example, it may be useful to treat a line as cached in the L1, but uncached in the L2.

The default region determined by the *GCR Base Address* register described in Section 6.3.6 above contains a mechanism for modifying the cache coherency attributes of the base region relative to that of the L1 cache. The attributes are programmed using the *CCA_Override_Enable* (bit 4) and *CCA_Override_Value* (bits 7:5) fields in the *CM2 GCR Base Address Register*. Addresses that do not map to any other region are mapped to the default region.

Any valid CCA value can be programmed into *CCA_Override_Value*, but because the L2 does not process coherent CCAs, a value of CWB (0x5) or CWBE (0x4) is automatically changed to WB (0x3) by the CM2 before being driven on the system memory OCP port.

The various coherency options are shown in Table 6.9. Note that the CCA overrides shown below only affect the L2 cache and not the L1 cache.

Table 6.9 Cache Coherency Attributes

Encoding	Name	Descriptions
0x0	WT	Write through.
0x1	—	Reserved.
0x2	UC	Uncached.
0x3	WB	Writeback, cacheable, non-coherent.
0x4	CWBE	Coherent writeback exclusive. Since the CM2 does not process coherent CCA's, this encoding automatically maps to WB (0x3).
0x5	CWB	Coherent writeback. Since the CM2 does not process coherent CCA's, this encoding automatically maps to WB (0x3).
0x6	—	Reserved.
0x7	UCA	Uncached accelerated.

The *CCA_Override_Enable* (bit 4) must be set in order for the *CCA_Override_Value* field to have meaning.

When overriding a CCA value, the CCA used within the L2 cache and driven to the system memory OCP interface is affected. Otherwise, the functionality of the transaction within the CM2 is based on the original CCA. Transactions that are not routed to the system memory OCP port, such as accesses to GCRs, GIC, CPC, or MMIO are also unaffected by the CCA Override.

6.3.16 In-Flight L1 and L2 Cache Operations

A core has the ability to issue a steady stream of cache operations and can potentially saturate the CM2 resources. To mitigate the possibility of this happening, the CM2 provides a mechanism to limit the number of successive cache transactions by a particular core. This limits a single core from issuing cache operations in rapid succession. The CM2 provides limits for both the L1 cache and the L2 cache via the *Global CM2 Control2* register located at offset address 0x0018. The default limit for successive L2 cache operations is four, meaning that a given core can execute a maximum of four cache operations (bits 19:16). For the L1 cache the limit is six cache operations (bits 3:0).

Setting a value of 0x0 in either of these fields disables this limitation. In this case the CM2 will not limit the number of successive cache operations that can be issued by a single core.

6.3.17 MIPS System Trace

The MIPS System trace is a new feature to the P6600 Multiprocessing System and allows the SoC designer to place signals from their non-probe SoC logic directly into the trace funnel for PDTrace to capture. The logic and registers that controls System Trace are handled by the CM2. For more information, refer to Section 3.6.2 in Chapter 3 of the *P6600 Multiprocessing System Hardware User's Manual* for more information on MIPS System Trace.

6.3.18 Error Processing

The CM2 detects, reports, and handles several types of errors that may be caused by errant software or hardware soft or hard errors. Table 6.10 lists the errors detected by the CM2. The first 7 errors are invalid requests to the GCR, GIC, or MMIO. There are two errors for invalid intervention responses due to inconsistent L1 cache states. And thereare 3 errors due to L2 RAM parity errors.

When an error is detected, information that may be useful in debugging the error is captured in the *Global CM2 Error Cause Register* and *Global CM2 Error Address Register*. Refer to Section 6.4.2.9, "Global CM2 Error Cause Register (GCR_ERROR_CAUSE Offset 0x0048)" and Section 6.4.2.10, "Global CM2 Error Address Register (GCR_ERROR_ADDR Offset 0x0050)" for more information.

If these registers already have valid error information and a second error isdetected, the error type of the second error is captured in the *CM2 Error Multiple Register*. However, an L2 ram correctable error is overwritten by a 2nd error that is not a second L2 ram correctable error. Refer to Section 6.4.2.12, "Global CM2 Error Multiple Register (GCR_ERROR_MULT Offset 0x0058)" for more information. Note that for the second error, only the error type is captured, not the associated error address.

When the *Global CM2 Error Cause Register* is loaded, an interrupt may be generated if the corresponding bit for that type of error is set in the *Global CM2 Error Mask Register* (see Table 6.26). If the error was generated by a request that requires a response and the corresponding *Global CM2 Error Mask Register* bit is 0, then the CM2 issues an ERROR response. However, if the corresponding *Global CM2 Error Mask Register* bit is 1, then the CM2 issues a normal response and an interrupt will be generated instead.

Table 6.10 CM2 Error Types

CM2_ERROR_TYPE	Error Name	Description	Action
0	-	Reserved	-
1	*GC_WR_ERR*	Non-Coherent Write of length > 1 to GCR or GIC	Drop Write Signal Interrupt if *CM_ERROR_MASK[1]* = 1
2	*GC_RD_ERR*	Non_Coherent Read of length > 1 to GCR or GIC	No GCR access Return SResp = ERROR if *CM_ERROR_MASK[2]* = 0 Signal Interrupt if CM2_ERROR_MASK[2] = 1
3	*COH_WR_ERR*	Coherent Writeback, Cacheop, or CohWriteInvalidate to GIC, GCR, MMIO	Intervention occurs Signal Interrupt if *CM_ERROR_MASK[3]* = 1
4	*COH_RD_ERR*	Coherent Read to GIC, GCR, MMIO	Intervention occurs After intervention, return SResp = ERROR to the original requestor if *CM_ERROR_MASK[4]* = 0 Signal Interrupt if *CM_ERROR_MASK[4]* = 1

Table 6.10 CM2 Error Types *(continued)*

CM2_ERROR_ TYPE	Error Name	Description	Action
5	*MMIO_WR_ERR*	Write to MMIO from the IOCU (only occurs if *CM_DISABLE_MMIO_LIMIT* = 0)	Drop Write Signal Interrupt if *CM_ERROR_MASK[5]* = 1
6	*MMIO_RD_ERR*	Write to MMIO from the IOCU (only occurs if *CM_DISABLE_MMIO_LIMIT* = 0)	Return SResp = ERROR if *CM_ERROR_MASK[6]* = 0 Signal Interrupt if *CM_ERROR_MASK[6]* = 1
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	If multiple M or E responses then data from core with lowest port ID is used. Signal Interrupt if *CM_ERROR_MASK[17]* = 1
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	If multiple M or E responses then data from core with lowest port ID is used. Return SResp = ERROR if *CM_ERROR_MASK[18]* = 0 Signal Interrupt if *CM_ERROR_MASK[18]* = 1
24	*L2_RD_UNCORR*	Request requires a response and: an uncorrectable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[24]* = 1
25	*L2_WR_UNCORR*	Request does not require a response and: an uncorrectable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[25]* = 1
26	*L2_CORR*	A correctable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[26]* = 1

When an error occurs, hardware updates the read-only CM2_ERROR_TYPE field in bits 31:27 of the Global Config register with one of the values listed in Table 6.10 above. Refer to Section 6.4.2.1 "Global Config Register (GCR_CONFIG Offset 0x0000)" for more information. When this field is written, hardware also updates the 27-bit ERROR_INFO field that provides additional information about the error. The organization of this field varies depending on the value in the CM2_ERROR_TYPE field.

6.3.18.1 Error Codes 1 - 15

If the decimal value in the CM2_ERROR_TYPE field is between 1 and 15, the ERROR_INFO field in the *Global CM2 Error Cause* register is organized as shown in Table 6.11.

Table 6.11 State of ERROR_INFO Field for Error Types 1 through 15

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 Table 6.12)
6:3	Source TagID
2:0	Source Port

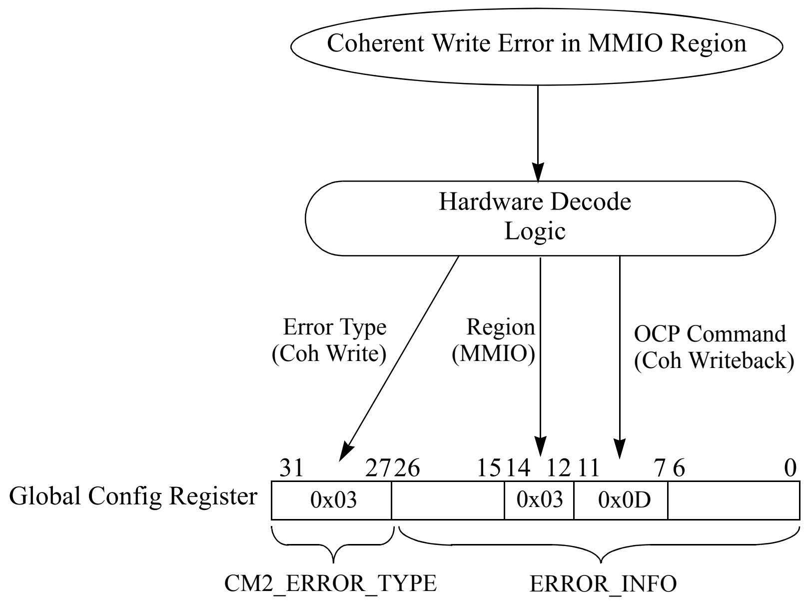
As shown in the above table, the *OCP MCmd* field in bits 11:7 is further encoded as shown in Table 6.12 below.

Table 6.12 MCmd (Bits 11:7) Encoding for CM2_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

Consider the example where a coherent write error occurs to the MMIO region during a coherent writeback operation. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 6.11 Example of a Coherent Write Error to MMIO



6.3.18.2 Error Codes 16 - 23

If the decimal value in the CM2_ERROR_TYPE field is between 16 and 23, the ERROR_INFO field in the *Global Config* register is organized as shown in Table 6.13.

Table 6.13 State of ERROR_INFO Field for Error Types 16 through 23

Bit	Meaning
26:21	Reserved
20:19	Coherent state from core 3 (see Table 6.14)
18	Intervention SResp from core 3 (see Table 6.15)
17:16	Coherent state from core 2 (see Table 6.14)
15	Intervention SResp from core 2 (see Table 6.15)
14:13	Coherent state from core 1 (see Table 6.14)
12	Intervention SResp from core 1 (see Table 6.15)
11:10	Coherent state from core 0 (see Table 6.14)
9	Intervention SResp from core 0 (see Table 6.15)
8	Request was from a Store Conditional
7:3	OCP MCmd (see Table 6.12)
2:0	Source port

Note that for each of the coherent state errors in Table 6.13 (bits 20:19, 17:16, 14:13, and 11:10), the encoding for these fields is shown in Table 6.14.

Table 6.14 Coherent State Values for Error Types 16 through 23

Encoding	Meaning
0	Invalid
1	Shared
2	Modified
3	Exclusive

For each of the Intervention SResp errors in Table 6.13 (bits 18, 15, 12, and 9), the encoding for these bits is shown in Table 6.15.

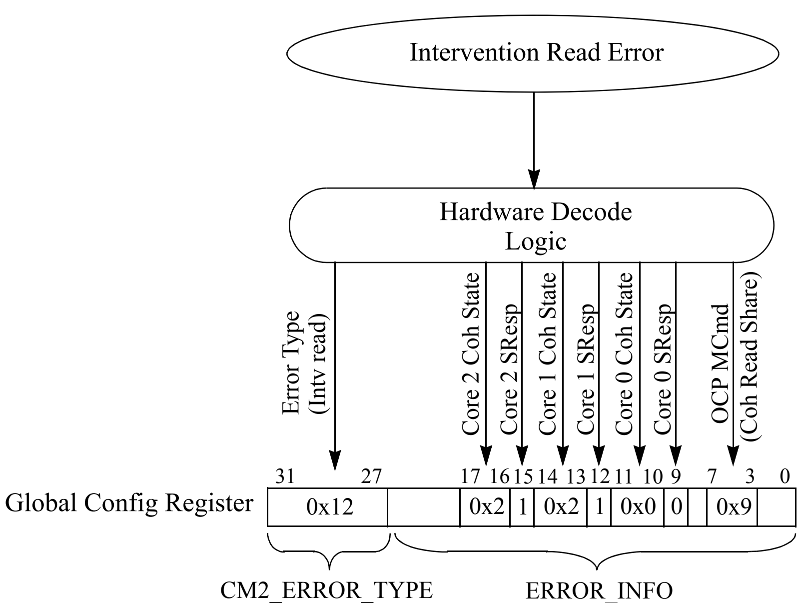
Table 6.15 Intervention SResp Values for Error Type 16 to 23

Encoding	Meaning
0	OK
1	Data (DVA)

Bits 7:3 of the ERROR_INFO field are encoded the same as those shown in Table 6.12.

Consider the example where a core issues a coherent read, and both cores 1 and 2 respond with modified data. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 6.12 Example of a Intervention Read Error to MMIO



6.3.18.3 Error Codes 24 - 26

If the decimal value in the *CM2_ERROR_TYPE* field is between 24 and 26, the *ERROR_INFO* field in the *Global Config* register is organized as shown in Table 6.16.

Table 6.16 State of ERROR_INFO Field for Error Types 24 to 26

Bit	Meaning
26:24	Reserved (zero)
23	Multiple Uncorrectable
22:18	Instruction[4:0] associated with the error see Table 6.17
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

For each of the errors types 24 - 26 listed in Table 6.10, the instruction associated with the error is encoded into bits 22:18 of the ERROR_INFO field as shown in Table 6.16. The encoding for these bits is shown in Table 6.17 below.

Table 6.17 Instructions for Error Type 24 to 26

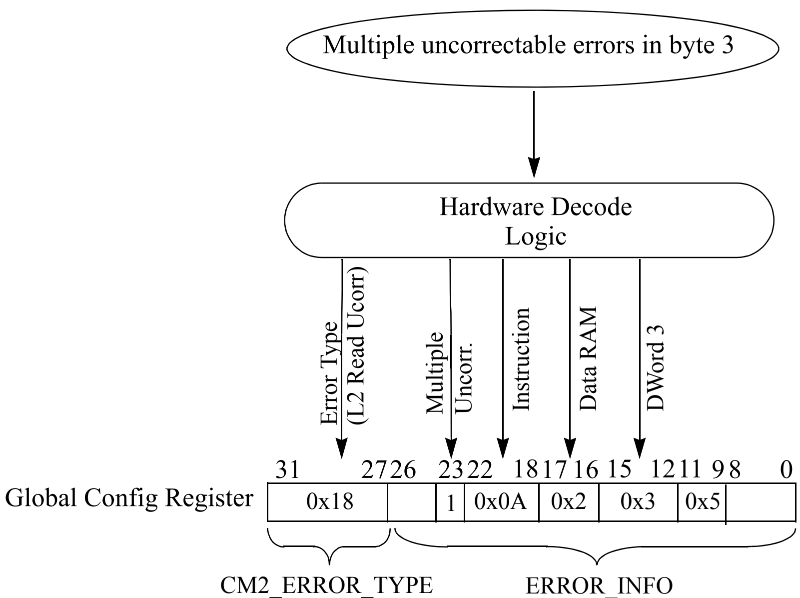
Bit	Meaning
0x00	L2_NOP
0x01	L2_ERR_CORR
0x02	L2_TAG_INV
0x03	L2_WS_CLEAN
0x04	L2_RD_MDYFY_WR
0x05	L2_WS_MRU
0x06	L2_EVICT_LN2
0x08	L2_EVICT
0x09	L2_REFL
0x0A	L2_RD
0x0B	L2_WR
0x0C	L2_EVICT_MRU
0x0D	L2_SYNC
0x0E	L2_REFL_ERR
0x10	L2_INDX_WB_INV
0x11	L2_INDX_LD_TAG

Table 6.17 Instructions for Error Type 24 to 26 *(continued)*

Bit	Meaning
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

Consider the example of multiple uncorrectable errors in DWord 3, way 5 of the Data RAM during an *L2 Read* instruction. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 6.13 Multiple Uncorrectable Errors to Byte 3 of the Data RAM During an L2 Hit Writeback Instruction



6.3.19 Custom GCR Implementation

The CM2 provides the ability for the user to implement a 64 KB block of custom registers that can be used to control system level functions. These registers are defined by the user and then instantiated into the design. The CM2 provides two global registers to handle the implementation of customer registers: the *Global Custom Base* register at offset 0x0060, and the *Global Custom Status* register located at offset 0x0068.

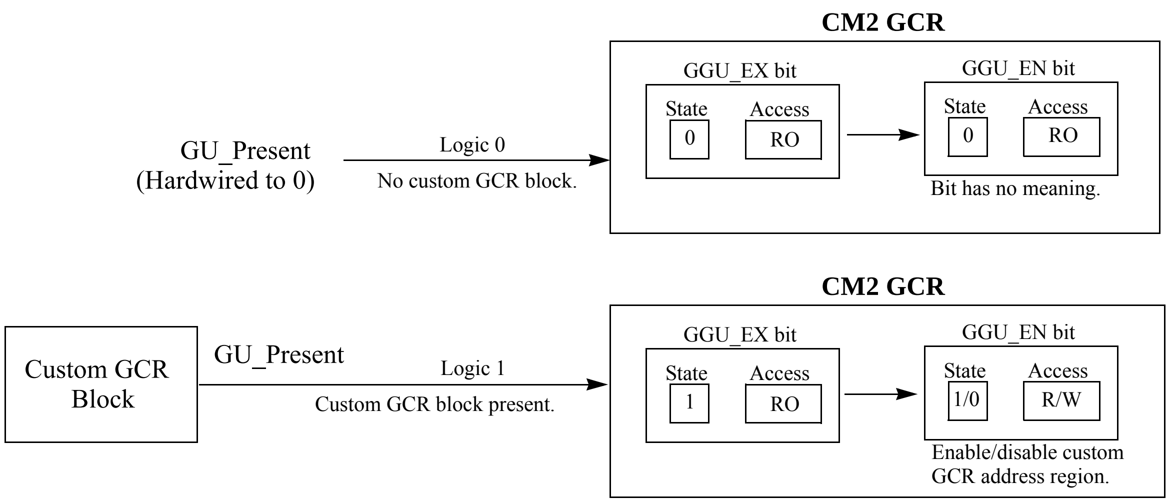
The existence of a custom GCR implementation in the system is selected during IP Configuration. If this option is selected, custom GCR hardware must drive the internal *GU_Present* pin to the CM2. The state of this pin is loaded into the GGU_EX bit in the *Global Custom Status* register. This bit indicates that a custom GCR block is connected to the CM2. Note that *GU_Present* is an internal signal that is an output of the Custom GCR and is connected to the CM2 logic.

If a custom block is implemented, the starting address in memory of the 64 KB block is determined using the 16-bit CUSTOM_BASE field in the *Global Custom Base* register. Note that unlike the configuration of the CM2 Global control registers described in Section 6.3.6, the CUSTOM_BASE field does not have a default base address and this field is undefined at reset. Therefore, it is software's responsibility to program the base address into this field during boot time if a custom GCR block is implemented.

In addition, the selected address region where the registers will reside must be enabled by setting the GGU_EN bit in the *Global Custom Base* register. Note that the accessibility of this bit by software depends on the state of the *GGU_EX* bit described above. If *GGU_EX* is cleared (zero), indicating that no custom GCR is connected to the CM2, then the *GGU_EN* bit becomes RO and is not accessible by software. If this bit is set, indicating that a custom GCR is connected to the CM2, then the *GGU_EN* bit becomes R/W and is accessible by software.

This concept is described in Figure 6.14 below.

Figure 6.14 Relationship Between the CM_Present Signal and the GGU_EX and GGU_EN Bits at Reset



Note that, depending on the user's implementation, the custom GCR may handle 64-bit reads/writes (unlike the normal GCR which only handles 32-bit accesses). For more information on this feature, contact MIPS Customer Support.

6.3.20 Attribute-Only Regions

The CM2 provides four standard variable-size regions as described in Section 6.3.7, "Address Regions", as well as four additional attribute-only regions. The attribute only regions allows the cache coherency attributes for that region to be modified, but they cannot be used to select between memory and I/O as the target.

In a situation where all of the standard variable size regions have been allocated, the attribute-only regions can be used to override the cache coherency attributes for that memory region. For example, all four attribute-only regions can be mapped to a single IOCU.

The CM2 uses four sets of base/mask registers to manage up to four attribute-only regions. The Base registers described in Section 6.4.5.1, "CM2 Attribute-Only Region [0 - 3] Base Address Registers (GCR_REGn_ATTR_BASE Offsets 0x0190, 0x01A0, 0x0210, 0x0220)" contain the base address in memory for each region. The Mask registers described in Section 6.4.5.3, "CM Attribute-Only Region[0 - 3] Address Mask

Registers (GCR_REGn_ATTR_MASK Offsets 0x0198, 0x1A8, 0x218, 0x228)" contain the size of the region and the CCA override information.

These registers are shown starting at offset address 0x0190 in Table 6.18 below:

6.4 Global Control Block

6.4.1 Global Control Block Address Map

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

Table 6.18 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x0000	Global Config Register (*GCR_CONFIG*)	R	Indicates the number of Processor cores, number of interrupts, number of IOCUs, etc.
0x0008	GCR Base Register (*GCR_BASE*)	R/W	Base of the control register space.
0x000C	GCR Base Upper Register (*GCR_BASE_UPPER*)	R/W	Upper bits of the base of the control register space.
0x0010	Global CM2 Control Register (*GCR_CONTROL*)	R/W	Control bits for the Coherence Manager
0x0018	Global CM2 Control2 Register (*GCR_CONTROL2*)	R/W	More Control bits for the Coherence Manager
0x0020	Global CSR Access Privilege Register (*GCR_ACCESS*)	R/W	Controls which Cores can modify the GCR Registers
0x0030	GCR Revision Register (*GCR_REV*)	R	RevisionID of the GCR hardware
0x0040	Global CM2 Error Mask Register (*GCR_ERROR_MASK*)	R/W	Controls what Errors are reported as Interrupts
0x0048	Global CM2 Error Cause Register (*GCR_ERROR_CAUSE*)	R/W	Captures info when an Error occurs within the CM2
0x0050	Global CM2 Error Address Register (*GCR_ERROR_ADDR*)	R/W	Captures address which caused the CM2 error.
0x0054	Global CM2 Error Address Upper Register (*GCR_ERROR_ADDR_UPPER*)	R/W	Captures the upper bits of the address (above bit 32) which caused the CM2 error.
0x0058	Global CM2 Error Multiple Register (*GCR_ERROR_MULT*)	R/W	Captures information for subsequent CM2 errors.
0x0060	GCR Custom Base Register (*GCR_CUSTOM_BASE*)	R/W	Base address of the custom user-defined 64KB control register space.
0x0064	GCR Custom Base Upper Register (*GCR_CUSTOM_BASE_UPPER*)	R/W	Upper bits of the base address of the custom user-defined 64KB control register space.

Table 6.18 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x0068	GCR Custom Status Register (*GCR_CUSTOM_STATUS*)	R/W	Existence and status of the custom user-defined GCR
0x0070	Global L2 only Sync Register (*GCR_L2_ONLY_SYNC_BASE*)	R/W	Base address of the L2 only Sync 4KB address space
0x0074	Global L2 only Sync Upper Register (*GCR_L2_ONLY_SYNC_BASE_UPPER*)	R/W	Upper bits of the base address of the L2 only Sync 4KB address space.
0x0080	Global Interrupt Controller Base Address Register (*GCR_GIC_BASE*)	R/W	GIC Base Address
0x0084	Global Interrupt Controller Base Address Upper Register (*GCR_GIC_BASE_UPPER*)	R/W	GIC Upper base address. Stores address bits 39:32.
0x0088	Cluster Power Controller Base Address Register (*GCR_CPC_BASE*)	R/W	CPC base address
0x008C	Cluster Power Controller Base Address Upper Register (*GCR_CPC_BASE_UPPER*)	R/W	CPC base address. Stores address bits 39:32.
0x0090	CM2 Region0 Base Address Register (*GCR_REG0_BASE*)	R/W	Address Region0 Base Address This register is present onlywhen the IOCU is present.
0x0094	CM2 Region0 Base Address Upper Register (*GCR_REG0_BASE_UPPER*)	R/W	Address Region0 Base Address. Stores address bits 39:32 of region 0 address. This register is present only when the IOCU is present.
0x0098	CM2 Region0 Address Mask Register (*GCR_REG0_MASK*)	R/W	Address Region0 Size and Destination This register is present onlywhen the IOCU is present.
0x009C	CM2 Region0 Address Mask Upper Register (*GCR_REG0_MASK_UPPER*)	R/W	Address Region0 Size and Destination. Stores address mask bits 39:32 of region 0. This register is present onlywhen the IOCU is present
0x00A0	CM2 Region1 Base Address Register (*GCR_REG1_BASE*)	R/W	Address Region1 Base Address This register is present onlywhen the IOCU is present
0x00A4	CM2 Region1 Base Address Upper Register (*GCR_REG1_BASE_UPPER*)	R/W	Address Region1 Base Address. Stores address bits 39:32 of region 1. This register is present only when the IOCU is present.
0x00A8	CM2 Region1 Address Mask Register (*GCR_REG1_MASK*)	R/W	Address Region1 Size and Destination This register is present onlywhen the IOCU is present.
0x00AC	CM2 Region1 Address Mask Upper Register (*GCR_REG1_MASK_UPPER*)	R/W	Address Region1 Size and Destination. Stores address mask bits 39:32 of region 1. This register is present onlywhen the IOCU is present
0x00B0	CM2 Region2 Base Address Register (*GCR_REG2_BASE*)	R/W	Address Region2 Base Address This register is present onlywhen the IOCU is present
0x00B4	CM2 Region2 Base Address Upper Register (*GCR_REG2_BASE_UPPER*)	R/W	Address Region1 Base Address. Stores address bits 39:32 of region 2. This register is present only when the IOCU is present.

Table 6.18 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x00B8	CM2 Region2 Address Mask Register (*GCR_REG2_MASK*)	R/W	Address Region2 Size and Destination This register is present onlywhen the IOCU is present
0x00BC	CM2 Region2 Address Mask Upper Register (*GCR_REG2_MASK_UPPER*)	R/W	Address Region2 Size and Destination. Stores address mask bits 39:32. This register is present only when the IOCU is present
0x00C0	CM2 Region3 Base Address Register (*GCR_REG3_BASE*)	R/W	Address Region3 Base Address This register is present onlywhen the IOCU is present
0x00C4	CM2 Region3 Base Address Upper Register (*GCR_REG3_BASE_UPPER*)	R/W	Address Region1 Base Address. Stores address bits 39:32 of region 3. This register is present only when the IOCU is present.
0x00C8	CM2 Region3 Address Mask Register (*GCR_REG3_MASK*)	R/W	Address Region3 Size and Destination This register is present onlywhen the IOCU is present
0x00CC	CM2 Region3 Address Mask Upper Register (*GCR_REG3_MASK_UPPER*)	R/W	Address Region3 Size and Destination. Stores address mask bits 39:32 of region 3. This register is present onlywhen the IOCU is present
0x00D0	Global Interrupt Controller Status Register (*GCR_GIC_STATUS*)	R	Existence and status of GIC
0x00E0	Cache Revision Register (*GCR_CACHE_REV*)	R	Revision of cache attached to the coherent Cluster.
0x00F0	Cluster Power Controller Status Register (*GCR_CPC_STATUS*)	R	Existence and status of CPC.
0x0100	IOCU Base Address Register (GCR_IOC_BASE)	R/W	Address Base for IOMMU registers contained within the IOCUs.
0x0104	IOCU Base Address Upper Register (GCR_IOC_BASE_UPPER)	R/W	Upper portion of address base for IOMMU registers contained within the IOCUs.
0x0108	IOMMU Status Register (GCR_IOMMU_STATUS)	R	Existence of IOMMU inside IOCU.
0x0190	CM Attribute-Only Region0 Base Address Register (*GCR_REG0_ATTR_BASE*)	R/W	Attribute-only region 0 base address.
0x0194	CM Attribute-Only Region0 Base Address Upper Register (*GCR_REG0_ATTR_BASE_UPPER*)	R/W	Attribute-only region 0 upper base address. Stores bits 39:32 of the address.
0x0198	CM Attribute-Only Region0 Address Mask Register (*GCR_REG0_ATTR_MASK*)	R/W	Attribute-only region 0 mask bits.
0x019C	CM Attribute-Only Region0 Address Mask Upper Register (*GCR_REG0_ATTR_MASK_UPPER*)	R/W	Attribute-only region 0 upper mask bits. Stores bits 39:32 of the address mask.
0x01A0	CM Attribute-Only Region1 Base Address Register (*GCR_REG0_ATTR_BASE*)	R/W	Attribute-only region 1 base address.
0x01A4	CM Attribute-Only Region1 Base Address Upper Register (*GCR_REG1_ATTR_BASE_UPPER*)	R/W	Attribute-only region 1upper base address. Stores bits 39:32 of the address.
0x01A8	CM Attribute-Only Region1 Address Mask Register (*GCR_REG1_ATTR_MASK*)	R/W	Attribute-only region 1 mask bits.

Table 6.18 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x01AC	CM Attribute-Only Region1 Address Mask Upper Register (*GCR_REG1_ATTR_MASK_UPPER*)	R/W	Attribute-only region 1 upper mask bits. Stores bits 39:32 of the address mask.
0x0200	IOCU Revision Register (*GCR_IOCU1_REV*)	R	Revision of IOCU
0x0210	CM Attribute-Only Region2 Base Address Register (*GCR_REG2_ATTR_BASE*)	R/W	Attribute-only region 2 base address.
0x0214	CM Attribute-Only Region2 Base Address Upper Register (*GCR_REG2_ATTR_BASE_UPPER*)	R/W	Attribute-only region 2 upper base address. Stores bits 39:32 of the address.
0x0218	CM Attribute-Only Region2 Address Mask Register (*GCR_REG2_ATTR_MASK*)	R/W	Attribute-only region 2 mask bits.
0x021C	CM Attribute-Only Region2 Address Mask Upper Register (*GCR_REG2_ATTR_MASK_UPPER*)	R/W	Attribute-only region 2 upper mask bits. Stores bits 39:32 of the address mask.
0x0220	CM Attribute-Only Region3 Base Address Register (*GCR_REG3_ATTR_BASE*)	R/W	Attribute-only region 3 base address.
0x0224	CM Attribute-Only Region3 Base Address Upper Register (*GCR_REG3_ATTR_BASE_UPPER*)	R/W	Attribute-only region 3 upper base address. Stores bits 39:32 of the address.
0x0228	CM Attribute-Only Region3 Address Mask Register (*GCR_REG3_MASK*)	R/W	Attribute-only region 3 mask bits.
0x022C	CM Attribute-Only Region3 Address Mask Upper Register (*GCR_REG3_ATTR_MASK_UPPER*)	R/W	Attribute-only region 3 upper mask bits. Stores bits 39:32 of the address mask.
0x0240	L2 RAM Configuration register. (*GCR_L2_RAM_CONFIG*)	R/W	L2 RAM configuration parameters.
0x0300	L2 Prefetch control register. (*GCR_L2_PFT_CONTROL*)	R/W	L2 prefetch control.
0x0308	L2 Prefetch 2nd control register. (*GCR_L2_PFT_CONTROL_B*)	R/W	L2 prefetch 2nd control register.
All Others	Reserved.	-	For Future Extensions

6.4.2 CM2 Configuration Registers

This section describes the CM2 configuration registers, including control, error and mask, revision, and custom-GCR registers.

6.4.2.1 Global Config Register (GCR_CONFIG Offset 0x0000)

This register provides information on the overall system configuration. These fields are read-only and their reset state is determined at IP configuration time. Refer to Section 6.3.7, "Address Regions" for more information on how the address regions are used.

Figure 6.15 Global Configuration Register Format

31 … 20	19 … 16	15 … 12	11 … 8	7 … 0
R	ADDR_REGIONS	R	NUMIOCU	PCORES

Table 6.19 Global Config Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>RESERVED</td><td>31:20</td><td>Reserved, Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
<tr><td>ADDR_REGIONS</td><td>19:16</td><td>Number of address regions. Total number of CM2 Address Regions. Note: only 0, 4, 6, or 8 address regions are currently supported. All other encoded values not listed below are reserved.
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>0 Address Regions - no IOCU</td></tr>
<tr><td>0x4</td><td>4 Address Regions - standard</td></tr>
<tr><td>0x6</td><td>6 Address Regions - 4 standard + 2 Attribute Only</td></tr>
<tr><td>0x8</td><td>8 Address Regions - 4 standard + 4 Attribute Only</td></tr>
</table></td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>RESERVED</td><td>15:12</td><td>Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
<tr><td>NUMIOCU</td><td>11:8</td><td>Total number of IOCUs in the system. Note:only1 IOCU is currently supported.

0x0: Reserved
0x1: 1 IOCU
0x2 - 0xF: Reserved</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>PCORES</td><td>7:0</td><td>Total number of P6600 cores in the system not including the IOCUs. All values not shown are reserved.

0x00: 1 core
0x01: 2 cores
0x02: 3 cores
0x03: 4 cores
0x04: 5 cores
0x05: 6 cores
0x06 - 0xFF: Reserved</td><td>R</td><td>IP Configuration Value</td></tr>
</table>

6.4.2.2 GCR Base Register (GCR_BASE Offset 0x0008)

Within the physical address space, the location of the GCR is set by the *GCR_BASE* register. The MIPS default power-up value produces the physical address 0x00_1FBF_8000. A different default value may be specified at IP configuration time.

Refer to Section 6.3.6, "Setting the CM2 Register Block Base Address" and Section 6.3.15, "Setting the Cache Coherency Attributes for Default Memory Transfers" for more information on how this register is used.

Figure 6.16 GCR Base Register Format

31 … 15	14 … 8	7 … 5	4	3 2	1 … 0
GCR_BASE	R	CCA	CCAEN	R	CM2_TARGET

Table 6.20 GCR Base Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>GCR_BASE</td><td>31:15</td><td>This field works in conjunction with the GCR_BASE_UPPER register below to set the base address of the 32KB GCR block of the P6600 MPS.
This register has a fixed value after reset if configured as Read-Only (an IP Configuration Option).</td><td>R or R/W (IP Configuration)</td><td>IP Configuration Value
MIPS Default:
0x00_1FBF_8</td></tr>
<tr><td>RESERVED</td><td>14:8</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CCA</td><td>7:5</td><td>CCA default override value. Used in conjunction with CCAEN to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See CCAEN field.
<table>
<tr><th>Encoding</th><th>Name</th><th>Description</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>Writeback, cacheable, noncoherent</td></tr>
<tr><td>0x4</td><td>CWBE</td><td>Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td><td>Mapped to WB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>CCAEN</td><td>4</td><td>If CCA_DEFAULT_OVERRIDE_ENABLE is set to 1 and CM2_DEFAULT_TARGET is set to Memory, then transactions with addresses that do not map to any region will have a CCA value set to CCA_DEFAULT_OVERRIDE_VALUE when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>RESERVED</td><td>3:2</td><td>Read as 0x0. Must be written with a value of 0x0.</td><td>-</td><td>0x0</td></tr>
</table>

Table 6.20 GCR Base Register Descriptions *(continued)*

Name	Bits	Description	Read/ Write	Reset State
CM2_DEFAULT_ TARGET	1:0	Determines the target device for addresses which do not match any address map entry. 00: Memory 01: Reserved 10: IOCU 11: Reserved Only used for hardware I/O-Coherent systems.	R/W	Value of signal *SI_CM_Default_ Target[1:0]*

6.4.2.3 GCR Base Upper Register (GCR_BASE_UPPER Offset 0x000C)

Within the physical address space, the location of the GCR is set by the *GCR_BASE* register. This register works in conjunction with the GCR Base Register described above to provide a complete 36-bit base address.

Figure 6.17 GCR Base Upper Register Format

31 … 5	4 … 0
R	GCR_BASE_UPPER

Table 6.21 GCR Base Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
R	31:5	Reads as 0x0. Must be written with a value of 0x0.	R	0
GCR_BASE_UPPER	4:0	This field works in conjunction with the GCR_BASE register above to set the base address of the 32KB GCR block of the P6600 MPS. This register has a fixed value after reset if configured as Read-Only (an IP Configuration Option).	R or R/W (IP Configuration)	IP Configuration Value MIPS Default: 0x00

6.4.2.4 Global CM2 Control Register (GCR_CONTROL Offset 0x0010)

Figure 6.18 Global CM2 Control Register Format

31 … 17	16
R	SYNCCTL

15 … 8	7 … 6	5	4	3	2	1	0
R	U	SYNCDIS	IVU_EN	SHST_EN	PARK_EN	MMIO_LIMIT_DIS	SPEC_READ_EN

Table 6.22 Global CM2 Control Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:17	Read as 0x0. Must be written with a value of 0x0.	-	0x0
SYNCCTL	16	Determines SYNC behavior when a SYNC level 0x0 is executed by a core. *SyncCtl* = 1 means Sync0 generates a memory sync *SyncCtl* = 0 means Sync0 generates an intervention sync	RW	0x0
RESERVED	15:8	Read as 0x0. Must be written with a value of 0x0.	R	0x0
UNUSED	7:6	These bits are currently unused. When writing to this register, software should assign a value of 2’b00 to this field.	R/W	0x0

Table 6.22 Global CM2 Control Register Descriptions *(continued)*

Name	Bits	Description	Read/ Write	Reset State
SYNCDIS	5	SYNC transmit disable. Set to 1 to disable the propagation of SYNC transactions on the system memory port. This has the same effect as deasserting *SI_SyncTxEn*. Setting to 0 makes the propagation of SYNC transactions on the system memory port dependent solely on the state of *SI_SyncTxEn*. Refer to the pin descriptions chapter in the P6600 Hardware User’s Manual for more information on this pin.	RW	0x0
IVU_EN	4	Stall until interventions are completed. Set to 1 to stall serialization when a core’s clock is stopping or is being powered down by the CPC until all previous interventions are complete. Set to 0 for no stalling of serialization when a core is going offline.	RW	0x0
SHST_EN	3	Force coherent read data to shared state in L1 data cache. If set to 1 then Coherent Read Data is always installed in the Level 1 cache of the requesting P6600 core in the SHARED state. If set to 0 then Coherent Read Data may be installed in the Level 1 cache in the SHARED state (if the data coexists in other Level 1 caches) or EXCLUSIVE (if the data does not coexist in other Level 1 caches).	RW	0x0
PARK_EN	2	I/O port parking enable. If set to 1 and the *SI<iocu>_CMP_IOC_ParkEn* signal is 1, then I/O Port Parking is enabled for the corresponding IOCU. I/O Port parking is a mechanism where the CM2 only serializes requests from the IOCU for some period of time. If set to 0 or *SI<iocu>_CMP_IOC_ParkEn* signal is 0, then the I/O Port Parking is disabled for the corresponding IOCU. This bit has no effect in systems without an IOCU (i.e., they are not hardware I/O coherent).	RW	0x0

Table 6.22 Global CM2 Control Register Descriptions *(continued)*

Name	Bits	Description	Read/ Write	Reset State
MMIO_LIMIT_DIS	1	Limit requests to memory-mapped I/O. If set to 0, the CM2 avoids deadlock in systems with hardware I/O coherence by limiting requests issued to Memory-Mapped I/O. An MMIO request will be selected for serialization only if the previous request and write data (if applicable) has been accepted by the IOCU. If set to 1, MMIO requests are not limited and therefore deadlock may occur in systems with hardware I/O coherence unless avoided by some other mechanism. This bit has no effect in systems without an IOCU (i.e., they are not hardware I/O coherent) because there are no MMIO ports and therefore the limit does not apply.	RW	0x0
SPEC_READ_EN	0	Speculative coherent read enable. If set to 1, the CM2 may speculatively read memory for a coherent read before the intervention for that read has completed. Performance is improved by reading memory in parallel with the intervention. If set to 0, the CM2 will never issue speculative reads to memory.	R/W	0x1

6.4.2.5 Global CM2 Control2 Register (GCR_CONTROL2 Offset 0x0018)

This register sets limits on how many consecutive cache operations are allowed to the L1 and L2 caches. Refer to Section 6.3.16, "In-Flight L1 and L2 Cache Operations" for more information on how this register is used.

Figure 6.19 Global CM2 Control2 Register Format

31 — 20	19 — 16	15 — 4	3 — 0
R	L2_CACEOP_LIMIT	R	L1_CACEOP_LIMIT

Table 6.23 Global CM2 Control2 Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:20	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0x0

Table 6.23 Global CM2 Control2 Register *(continued)*

Name	Bits	Description	Read/ Write	Reset State
L2_CACHEOP_LIMIT	19:16	L2 CacheOp transaction limit. The total number of L2 CacheOp transactions allowed by the CM2 serialization arbiter to be simultaneously in-flight. An L2 CacheOp is defined as any transaction with MAddrSpace = 0b001 or 0b010. In this context, an L2 CacheOp transaction is considered in-flight when it is selected for serialization by the CM2 until the request is issued on the CM2's system memory OCP Port. Setting a value of 0x0 disables the limit (i.e., the CM2 serialization arbiter will not explicitly limit the number of in-flight L12 CacheOps). Setting a value of 0x1 allows only a single in-flight L2 CacheOp. Setting a value of 0x2 allows two in-flight L2 CacheOps, etc. The purpose of this limit is to avoid the case where one or more cores substantially impact the performance of other cores by issuing a rapid succession of L2 CacheOps.	R/W	0x4
RESERVED	15:4	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0x0
L1_CACHEOP_LIMIT	3:0	L1 CacheOp transaction limit. The total number of L1 CacheOp transactions allowed by the CM2 serialization arbiter to be simultaneously in-flight. A L1 CacheOp is defined as a transaction with MAddrSpace = 0b011 or 0b1xx. In this context, a transaction is considered in-flight when it is selected for serialization by the CM2 until its intervention response is processed by the CM2 (if the cacheOp did not receive a DVA intervention response) or until all intervention data has been received (if the cacheOp received a DVA intervention response). Setting a value of 0x0 disables the limit (i.e., the CM2 serialization arbiter will not explicitly limit the number of in-flight L1 CacheOps). Setting a value of 0x1 allows only a single in-flight L1 CacheOp. Setting a value of 0x2 allows two in-flight L1 CacheOps, etc... The purpose of this limit is to avoid the case where one or more cores substantially impact the performance of other cores by issuing a rapid succession of L1 CacheOps that receive an intervention response of DVA.	R/W	0x6

6.4.2.6 Global CSR Access Privilege Register (GCR_ACCESS Offset 0x0020)

A request can be initiated by either a core or an IOCU. The CM2 allows for a maximum of seven requestors. However, these requestors do not have unrestricted access to the CM2 register set and must be granted permission by software via this register. Refer to Section 6.3.4, "Requestor Access to GCR Registers" for more information on how this register is used.

Figure 6.20 Global CSR Access Privilege Register Format

31 … 8	7 … 0
R	CM2_ACCESS_EN

Table 6.24 Global CSR Access Privilege Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
CM2_ACCESS_EN	7:0	Requester access to global control registers. 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.	R/W	0xFF

6.4.2.7 CM2 Revision Register (GCR_REV Offset 0x0030)

Figure 6.21 GCR Revision Register Format

31 … 16	15 … 8	7 … 0
R	MAJOR_REV	MINOR_REV

Table 6.25 GCR Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000

Table 6.25 GCR Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
MAJOR_REV	15:8	CM2 Major revision number. This field reflects the major revision of the GCR block. A major revision might reflect the changes from one product generation to another. This value changes based on the processor revision. Refer to the errata sheet of the P6600 core for the exact value of this field.	R	Preset
MINOR_REV	7:0	CM2 Minor revision number. This field reflects the minor revision of the GCR block. A minor revision might reflect the changes from one release to another. This value changes based on the processor revision. Refer to the errata sheet of the P6600 core for the exact value of this field.	R	Preset

6.4.2.8 Global CM2 Error Mask Register (GCR_ERROR_MASK Offset 0x0040)

This register is used in conjunction with the *Global CM2 Error Cause* and *Global CM2 Error Address* registers to determine the type of error and the address which caused the error. Refer to Section 6.3.18, "Error Processing" for more information on how this register is used.

Figure 6.22 Global CM2 Error Mask Register Format

31 … 0
CM2_ERROR_MASK

Table 6.26 Global CM2 Error Mask Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_MASK	31:0	CM2 Error Mask field. 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-Type Errors completes with OK response to avoid double reporting of the error. The Error Types that can be captured are implementation-specific.	R/W	0x000A_002A (write errors cause interrupts; read errors provide error response)

6.4.2.9 Global CM2 Error Cause Register (GCR_ERROR_CAUSE Offset 0x0048)

This register is used in conjunction with the *Global CM2 Error Mask* and *Global CM2 Error Address* registers to determine the type of error and the address which caused the error. Refer to Section 6.3.18, "Error Processing" for more information on how this register is used.

Figure 6.23 Global CM2 Error Cause Register Format

31 27	26 0
CM2_ERROR_TYPE	ERROR_INFO

Table 6.27 Global CM2 Error Cause Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_TYPE	31:27	Indicates type of error detected. When *CM2_ERROR_TYPE* is zero, no errors have been detected. When *CM2_ERROR_TYPE* is non-zero, another error will not be reloaded until a power-on reset or this field is written to 0.	R/W	0
ERROR_INFO	26:0	Information about the error. If *CM2_ERROR_TYPE* = 1 through 15, see Table 6.11 if *CM2_ERROR_TYPE* = 16 through 23, see Table 6.13 if *CM2_ERROR_TYPE* = 24 through 26, see Table 6.16	R/W	Undefined

6.4.2.10 Global CM2 Error Address Register (GCR_ERROR_ADDR Offset 0x0050)

This register is used in conjunction with the *Global CM2 Error Cause* and *Global CM2 Error Mask* registers to determine the type of error and the address which caused the error. Refer to Section 6.3.18, "Error Processing" for more information on how this register is used.

Figure 6.24 Global CM2 Error Address Register Format

31 0
CM2_ERROR_ADDR

Table 6.28 Global CM2 Error Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_ADDR	31:0	This register works in conjunction with the CM2 Error Upper Address register below to request the address which caused the error. Loaded when the *Global Error Cause Register* is loaded. Bits 2:0 should always be 0.	R/W	Undefined

6.4.2.11 Global CM2 Error Address Upper Register (GCR_ERROR_ADDR_UPPER Offset 0x0054)

This register works in conjunction with the Global CM2 Error Address register above to provide a complete 40-bit address.

Figure 6.25 Global CM2 Error Address Upper Register Format

31 … 8	7 … 0
Reserved	CM2_ERROR_ADDR_UPPER

Table 6.29 Global CM2 Error Address Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_00
CM2_ERROR_ADDR_UPPER	7:0	This register works in conjunction with the CM2 Error Address register above to request the address which caused the error. Loaded when the *Global Error Cause Register* is loaded. Bits 2:0 should always be 0.	R/W	Undefined

6.4.2.12 Global CM2 Error Multiple Register (GCR_ERROR_MULT Offset 0x0058)

The *Global CM2 Error Cause*, *Global CM2 Error Address*, and *Global CM2 Error Mask* registers described above provide information on the type of error, and the address which caused the error. In addition to this information, the P6600 core also provides a way to determine the type of error should an secondary error occur. However, for the secondary error, only the type of error is logged, not the associated address. This register is used to log the type of secondary error. Refer to Section 6.3.18, "Error Processing" for more information on how this register is used.

Figure 6.26 Global CM2 Error Multiple Register Format

31 … 5	4 … 0
R	ERROR_2ND

Table 6.30 Global CM2 Error Multiple Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:5	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_000
CM2_ERROR_2ND	4:0	Type of second error. Loaded when the *Global CM2 Error Cause Register* has valid error information and a second error is detected.	R/W	5'b0

6.4.2.13 GCR Custom Base Register (GCR_CUSTOM_BASE Offset 0x0060)

This register allows for the implementation of custom registers that are designed by the customer and instantiated into the design at build time. Refer to Section 6.3.19, "Custom GCR Implementation" for more information on how this register is used.

Figure 6.27 GCR Custom Base Register Format

31–16	15–1	0
CUSTOM_BASE	R	GGU_EN

Table 6.31 GCR Custom Base Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CUSTOM_BASE	31:16	This field works in conjunction with the GCR Custom Base Upper register to set the base address of the 64KB GCR custom user-defined block of the P6600 Multiprocessing System.	R/W	Undefined
RESERVED	15:1	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000
GGU_EN	0	If this bit is set, the address region for the Custom GCR is enabled. This bit cannot be set to 1 if *GGU_EX* = 0, indicating that a custom GCR is not attached to the CM.	R/W (if *GGU_EX* = 1) R (if *GGU_EX* = 0)	0

6.4.2.14 GCR Custom Base Upper Register (GCR_CUSTOM_BASE_UPPER Offset 0x0064)

This register works in conjunction with the GCR Custom Base Address register above to provide a complete 40-bit address.

Figure 6.28 GCR Custom Base Register Upper Format

31–8	7–0
Reserved	CUSTOM_BASE_UPPER

Table 6.32 GCR Custom Base Register Upper Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_00
CUSTOM_BASE_UPPER	7:0	This field works in conjunction with the GCR Custom Base register above to set the upper base address of the 64KB GCR custom user-defined block.	R/W	Undefined

6.4.2.15 GCR Custom Status Register (GCR_CUSTOM_STATUS Offset 0x0068)

Refer to Section 6.3.19, "Custom GCR Implementation" for more information on how this register is used.

Figure 6.29 Global Custom Status Register Format

31–1	0
R	GGU_EX

Table 6.33 GCR Custom Status Register Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:1	Reads as 0x0. Must be written with a value of 0x0.	R	0x0
GGU_EX	0	If this bit is set, the Custom GCR is connected to the CM2. The state of this bit is set based on whether or not this block is implemented at build time as determined by the state of the *GU_Present* signal. If a Custom GCR block is not present, the *GU_Present* pin is driven to 0. If there is a custom GCR block present, then the user must drive GU_Present = 1 inside their custom GCR module.	R	Build time option

6.4.2.16 L2-Only Sync Base Register (GCR_L2_ONLY_SYNC_BASE Offset 0x0070)

The P6600 core provides a mechanism to execute a SYNC operation to only the L2 cache, without affecting the core. Refer to Section 6.3.13, "L2-Only SYNC Operation" for more information on how this register is used.

Figure 6.30 L2-Only Sync Base Register Format

31 — 12	11 — 1	0
SYNC_BASE	R	SYNC_EN

Table 6.34 L2-Only Sync Base Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
SYNC_BASE	31:12	L2-only SYNC base address. This field works in conjunction with the L2-Only Sync Base Upper register below to set the base address of the 4KB GCR L2 only Sync of the P6600 MPS.	R/W	Undefined
RESERVED	11:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
SYNC_EN	0	L2-only SYNC enable. If this bit is set, the CM2 treats an uncached write request as an L2 only Sync. If set to 0, the CM2 treats the uncached write as a regular uncached request.	R/W	0x0

6.4.2.17 L2-Only Sync Base Upper Register (GCR_L2_ONLY_SYNC_BASE_UPPER Offset 0x0064)

This register works in conjunction with the GCR L2 Only Sync Base Address register above to provide a complete address.

Figure 6.31 GCR L2 Only Sync Base Upper Register Format

31 ... 8	7 ... 0
Reserved	SYNC_BASE_UPPER

Table 6.35 GCR L2 Only Sync Base Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_00
SYNC_BASE_UPPER	7:0	This field works in conjunction with the L2-Only Sync Base register above to set the upper base address of the 64KB L2 only sync 4 KByte address space.	R/W	Undefined

6.4.3 CM2 Region Address Map Registers

6.4.3.1 Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)

Figure 6.32 Global Interrupt Controller Base Address Register Format

31 ... 17	16 ... 1	0
GIC_BASE_ADDR	R	GIC_EN

Table 6.36 Global Interrupt Controller Base Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
GIC_BASE_ADDR	31:17	Global Interrupt Controller Base Address. This field works in conjunction with the Global Interrupt Controller Base Upper Address register below to set the base address of the 128KB Global Interrupt Controller.	R/W	Undefined
RESERVED	16:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
GIC_EN	0	Global Interrupt Controller Enable. If this bit is set, the address region for the GIC is enabled. This bit can not be set to 1 if *GIC_EX* = 0, indicating that a GIC is not attached to the CM2.	R/W (if *GIC_EX* = 1) R (if *GIC_EX* = 0)	0

6.4.3.2 GIC Base Address Upper Register (GCR_GIC_BASE_UPPER Offset 0x0084)

This register works in conjunction with the GCR GIC Base Address register above to provide a complete 40-bit address.

Figure 6.33 GCR GIC Base Upper Register Format

31 8	7 0
Reserved	GIC_BASE_UPPER

Table 6.37 GCR GIC Base Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_00
GIC_BASE_UPPER	7:0	This field works in conjunction with the Global Interrupt Controller Base Address register above to set the upper base address of the GIC base address space.	R/W	Undefined

6.4.3.3 Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)

Figure 6.34 Cluster Power Controller Base Address Register Format

31 15	14 1	0
CPC_BASE_ADDR	R	CPC_EN

Table 6.38 Cluster Power Controller Base Address Register

Name	Bits	Description	Read/ Write	Reset State
CPC_BASE_ADDR	31:15	This field works in conjunction with the Cluster Power Controller Base Upper Address register below to set the 40-bit base address of the 32K Cluster Power Controller.	R/W	Undefined
RESERVED	14:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CPC_EN	0	If this bit is set, the address region for the CPC is enabled. This bit can not be set if 1 *CPC_EX* = 0, indicating that a CPC is not attached to the CM2.	R/W (if *CPC_EX* = 1) R (if *CPC_EX* = 0)	0

6.4.3.4 GIC CPC Address Upper Register (GCR_CPC_BASE_UPPER Offset 0x0084)

This register works in conjunction with the GCR CPC Base Address register above to provide a complete 40-bit address.

Figure 6.35 GCR CPC Base Upper Register Format

31 ... 8	7 ... 0
Reserved	CPC_BASE_UPPER

Table 6.39 GCR CPC Base Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_00
CPC_BASE_UPPER	7:0	This field works in conjunction with the Cluster Power Controller Base Address register above to set the upper base address of the 40-bit CPC base address space.	R/W	Undefined

6.4.3.5 CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)

Some or all of these registers may be removed during IP configuration. When an IOCU is present, there may be 4 CM2 Address Mask Registers implemented. When no IOCU is present, there may be 0 or 4 CM2 Address Mask Registers. When a register is not present, it is defined as Reserved and Read-Only of 0.

Figure 6.36 CM2 Region [0 - 3] Base Address Register Format

31 ... 16	15 14 ... 0
CM2_REGION_BASE_ADDR	R

Table 6.40 CM2 Region [0 - 3] Base Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_REGION_BASE_ADDR	31:16	CM2 region base address. This field works in conjunction with the CM2 Region Base Address Upper register below to set the base physical address of the memory region.	R/W	Undefined
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0

6.4.3.6 CM2 Region [0 - 3] Base Upper Address Register (GCR_REGn_BASE_UPPER Offsets 0x0094, 0x00A4, 0x00B4, 0x00C4)

These registers work in conjunction with their associated CM2 Region 0-3 base address registers above to form a complete 40-bit address.

Figure 6.37 CM2 Region [0 - 3] Base Address Upper Register Format

31 ... 8	7 ... 0
Reserved	REGION_BASE_UPPER

Table 6.41 CM2 Region [0 - 3] Base Address Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
REGION_BASE_UPPER	7:0	CM2 region base address. This field works in conjunction with the CM2 Region Base Address Upper register below to set the 40-bit Region address.	R/W	Undefined

6.4.3.7 CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)

Some or all of these registers may be removed during IP configuration. When an IOCU is present, there may be 4 CM2 Address Mask Registers implemented. When no IOCU is present, there may be 0 or 4 CM2 Address Mask Registers. When a register is not present, it is defined as Reserved and Read-Only of 0.

Figure 6.38 CM2 Region [0-3] Address Mask Register Format

31 15	14 8	7 5	4	3	2	1 0
CM2_REGION_ADDR_MASK	R	CCA_Override _Value	CCA_Override _Enable	R	DROP_L2	CM2_TARGET

Table 6.42 CM2 Region [0 - 3] Address Mask Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_REGION_ADDR_MASK	31:16	This field works in conjunction with the CM2 Region Mask Upper Address register below to set the size of the CM2 Region. This field is used along with its equivalent *CM2 Region Base Address Register*. The request address is logically ANDed with the value of this register. The value of the associated *Base Address Register* is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM2 region. The only allowed values in this register are contiguous sets of leading 0x1's. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFFF0 is allowed, but the value 0xFFEF is not allowed).	R/W	Undefined
RESERVED	15:8	Reads as 0x0. Must be written with a value of 0x0.	R	0

Table 6.42 CM2 Region [0 - 3] Address Mask Register Descriptions *(continued)*

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>CCA_Override_Value</td><td>7:5</td><td>Used with CCA_Override_Enable to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See CCA_Override_Enable field.
<table>
<tr><th>Encoding</th><th>Name</th><th>CCA</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>WriteBack cacheable, non-coherent,</td></tr>
<tr><td>0x4</td><td>CWBE</td><td rowspan="2">Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>CCA_Override_Enable</td><td>4</td><td>If CCA_Override_Enable is set and the CM2_TARGET field is set to Memory (0x1), then transactions with addresses that map to this region will have a CCA value set to CCA_Override_Value when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>3</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>DROP_L2</td><td>2</td><td>Drop L2 CacheOp write.
If this bit is set, the CM2 drops the L2 CacheOp write after it has been serialized.
If this bit is cleared, the L2 CacheOp writes behave like a regular L2 CacheOp request.</td><td>R/W</td><td>0</td></tr>
<tr><td>CM2_TARGET</td><td>1:0</td><td>Maps this region to the specified device. The IOCU can only be mapped to regions 0 - 3, while memory can be mapped to all regions.

00: Disabled
01: Memory
10: IOCU
11: Reserved</td><td>R/W</td><td>0</td></tr>
</table>

6.4.3.8 CM2 Region [0 - 3] Address Mask Upper Address Register (GCR_REGn_Mask_UPPER Offsets 0x009C, 0x00AC, 0x00BC, 0x00CC)

These registers work in conjunction with their associated CM2 Region 0-3 address mask registers above to forma complete 40-bit address.

Figure 6.39 CM2 Region [0 - 3] Address Mask Upper Register Format

31 ... 8	7 ... 0
Reserved	REGION_ADDR_MASK_UPPER

Table 6.43 CM2 Region [0 - 3] Address Mask Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
REGION_ADDR_MASK_ UPPER	7:0	This field works in conjunction with the CM2 Region Mask Address register above to set the upper portion of the mask bits to define address region size beyond 4GBytes. This field is used along with its equivalent CM2 Region Base Address Upper Register. The request address is logically ANDed with the value of this register. The value of the associated Base Address Upper Register is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM2 region. The only allowed values in this register are contiguous sets of leading 0x1's. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFC is allowed, but the value 0xFE is not allowed).	R/W	Undefined

6.4.4 CM2 Status and Revision Registers

This section contains the status registers for the GIC and CPC, and the revision information for the L2 cache.

6.4.4.1 Global Interrupt Controller Status Register (GCR_GIC_STATUS Offset 0x00D0)

Figure 6.40 Global Interrupt Controller Status Register Format

31 … 1	0
R	GIC_EX

Table 6.44 Global Interrupt Controller Status Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
GIC_EX	0	GIC to CM2 connection. If this bit is set, the GIC is connected to the CM2.	R	1

6.4.4.2 Cache Revision Register (GCR_CACHE_REV Offset 0x00E0)

Figure 6.41 Cache Revision Register Format

31 16	15 8	7 0
R	MAJOR_REV	MINOR_REV

Table 6.45 Cache Revision Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
MAJOR_REV	15:8	This field reflects the major revision of the Cache block inside the CM2.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the Cache block inside the CM2.	R	Preset

6.4.4.3 Cluster Power Controller Status Register (GCR_CPC_STATUS Offset 0x00F0)

Figure 6.42 Cluster Power Controller Status Register Format

31 1	0
R	CPC_EX

Table 6.46 Cluster Power Controller Status Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CPC_EX	0	This bit is always 1 in the P6600 core as the CPC is always connected to the CM2.	R	1

6.4.4.4 IOCU Base Address Register (GCR_IOC_BASE Offset 0x0100)

The IOCU Base Address register enables accesses to the IOMMU within each IOCU. This register only exists if at least one IOCU in the system contains an IOMMU.

The 32KB IOCU Address Region covers the IOCU attached to the CM2. The lowest 4K sub-region addresses the IOMMU registers inside the IOCU. The other 7 4KB sub-regions are not currently used. Reads to these 7 sub-regions or an IOMMU that does not exist returns 0's. Writes to those regions are dropped silently.

Figure 6.43 IOCU Base Address Register Format

31 15	14 1	0
IOC_BASE_ADDR	R	IOC_REG_EN

Table 6.47 IOCU Base Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
IOC_BASE_ADDR	31:15	This field works in conjunction with the IOCU Base Upper Address register below to set the IOCU base address. This value contains the base address of the 32K IOC Address Region. This region is broken into eight 4K subregions, each of which addresses a particular IOCU. Only the first region is used in the P6600 core.	R/W	Undefined
RESERVED	14:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
IOC_REG_EN	0	If this bit is set, the address region for the IOMMU within the IOCU is enabled.	R/W	0

6.4.4.5 IOCU Base Address Upper Register (GCR_IOC_BASE_UPPER Offset 0x0104)

The IOCU Base Address register enables accesses to the IOMMU within each IOCU.

Figure 6.44 IOCU Base Address Upper Register Format

31 8	7 0
R	IOC_BASE_UPPER

Table 6.48 IOCU Base Address Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
IOC_BASE_UPPER	7:0	This field works in conjunction with the IOCU Base Address register above to set the IOCU base address.	R/W	Undefined

6.4.4.6 IOMMU Status Register (GCR_IOMMU_STATUS Offset 0x0108)

This register provides information about the existence of an IOMMU in the IO Coherence Unit (IOCU). The existence of an IOMMU for each IOCU is determined at IP configuration time.

Figure 6.45 IOMMU Status Register Format

31 1	0
R	IOMMU0

Table 6.49 IOMMU Status Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
IOMMU0	0	If this bit is set, IOCU #0 contains an IOMMU.	R	IP Config

6.4.4.7 IOCU Revision Register (GCR_IOCU1_REV Offset 0x0200)

This register gives the existence and revision information for an IOCU.

Figure 6.46 IOCU Revision Register Format

31 … 16	15 … 8	7 … 0
R	MAJOR_REV	MINOR_REV

Table 6.50 IOCU Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
MAJOR_REV	15:8	This field reflects the major revision of the IOCU attached to the CM2. A major revision might reflect the changes from one product generation to another. The value of 0x0 means that no IOCU is attached.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the IOCU attached to the CM2. A minor revision might reflect the changes from one release to another.	R	Preset

6.4.5 CM2 Attribute-Only Region Address Map Registers

This section contains the base address and address mask registers for CM2 attribute-only regions 0 through 3. These register have the same functionality as the normal region registers, except they can not be used to map to MMIO vs. memory.

6.4.5.1 CM2 Attribute-Only Region [0 - 3] Base Address Registers (GCR_REGn_ATTR_BASE Offsets 0x0190, 0x01A0, 0x0210, 0x0220)

Some or all of these registers may be removed during IP configuration. These registers are similar to the CM2 Region Address Register except the attribute-only regions can not be used to determine if a request is routed to memory or the IOCU.

Figure 6.47 CM2 Attribute-Only Region [0 - 3] Register Format

31 … 16	15 … 0
CM2_REGION_BASE_ADDR	R

Table 6.51 CM2 Attribute-Only Region [0 - 3] Base Address Register Format

Name	Bits	Description	Read/ Write	Reset State
CM2_REGION_BASE_ADDR	31:16	This field sets the base physical address of the memory region.	R/W	Undefined
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0

6.4.5.2 CM2 Attribute-Only Region [0 - 3] Base Upper Address Register (GCR_REGn_ATTR_BASE_UPPER Offsets 0x0194, 0x01A4, 0x0214, 0x0224)

These registers work in conjunction with their associated CM2 Attribute-Only Region 0-3 base address registers above to form a complete 40-bit address.

Figure 6.48 CM2 Attribute-Only Region [0 - 3] Base Address Upper Register Format

31 … 8	7 … 0
Reserved	ATTR_REGION_BASE_UPPER

Table 6.52 CM2 Attribute-Only Region [0 - 3] Base Address Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
ATTR_REGION_BASE_UPPER	7:0	CM2 region base address. This field sets the base physical address bits 39:32.	R/W	Undefined

6.4.5.3 CM Attribute-Only Region[0 - 3] Address Mask Registers (GCR_REGn_ATTR_MASK Offsets 0x0198, 0x1A8, 0x218, 0x228)

These registers may be removed during IP Configuration. These registers are similar to the CM Region Address Mask registers except they may not be used to route requests to memory or the IOCU.

Figure 6.49 CM2 Attribute Only Region [0-3] Address Mask Register Format

31 … 15	14 … 8	7 … 5	4	3	2	1 … 0
CM2_REGION_ADDR_MASK	R	CCA_Override_Value	CCA_Override_EN	R	DROP_L2	R

Table 6.53 CM Attribute-Only Region [0 - 3] Address Mask Register Descriptions

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CM2_REGION_ADDR_MASK</td><td>31:16</td><td>This field is used to set the size of the CM Region.
This field is used along with its equivalent CM Region Base Address Register.
The request address is logically ANDed with the value of this register. The value of the associated Base Address Register is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM region.
The only allowed values in this register are contiguous sets of leading 0x1’s. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xfff0 is allowed, but the value 0xffef is not allowed).</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>RESERVED</td><td>15:8</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CCA_Override_Value</td><td>7:5</td><td>Used with CCA_Override_Enable to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See CCA_Override_Enable field.
<table>
<tr><th>Encoding</th><th>Name</th><th>CCA</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>WriteBack cacheable, non-coherent</td></tr>
<tr><td>0x4</td><td>CWBE</td><td rowspan="2">Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>CCA_Override_Enable</td><td>4</td><td>If set CCA_Override_Enable is set to 1 and CM_TARGET is set to Memory, then transactions with addresses that map to this region will have a CCA value set to CCA_Override_Value when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>RESERVED</td><td>3</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>DROP_L2</td><td>2</td><td>Set to 1 for the CM to drop L2 CacheOp writes after it has been serialized.
If set to 0, the L2 CacheOp writes behaves like a regular L2 CacheOp request.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>RESERVED</td><td>1:0</td><td>Reads as 0x0. Must be written with a value of 0x0.
Since the attribute-only registers can not be used to map to MMIO vs. memory, this field is not needed and is reserved.</td><td>R/W</td><td>0x0</td></tr>
</table>

6.4.5.4 CM2 Atrribute-Only Region [0 - 3] Address Mask Upper Address Register (GCR_REGn_Attr_Mask_Upper, Offsets 0x019C, 0x01AC, 0x021C, 0x022C)

These registers work in conjunction with their associated CM2 attribute-only Region 0-3 address mask registers above to form a complete 40-bit address.

Figure 6.50 CM2 Attribute-Only Region [0 - 3] Address Mask Upper Register Format

31 ... 8	7 ... 0
Reserved	REGION_ADDR_MASK_UPPER

Table 6.54 CM2 Attribute-Only Region [0 - 3] Address Mask Upper Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
REGION_ADDR_MASK_ UPPER	7:0	This field is used to set the upper portion of the mask bits which define the size beyond 4GBytes for attribute only memory region. This field is used along with its equivalent CM2 Attribute-Only Region Base Address Upper Register. The request address is logically ANDed with the value of this register. The value of the associated Base Address Upper Register is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM2 region. The only allowed values in this register are contiguous sets of leading 0x1's. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFC is allowed, but the value 0xFE is not allowed).	R/W	Undefined

6.4.5.5 L2 RAM Configuration Register (GCR_L2_RAM_CONFIG, Offset 0x0240)

These registers manage the L2 prefetch control mechanism in the P6600 MPS.

Figure 6.51 L2 RAM Configuration Register Format

31	30 ... 17	16	15 ... 13	12	11 ... 10	9 ... 8	7 ... 6	5 ... 4	3 ... 2	1 ... 0
PRESENT	R	L2_RAM_ COMPAT	R	L2_PIPE	R	L2_TAGRAM_ STALLS	R	L2_WSRAM_ STALLS	R	L2_DATA RAM_STALLS

Table 6.55 L2 RAM Configuration Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
PRESENT	31	This bit is always set in the P6600 CM2 to indicate that the L2 RAM Configuration register is present.	R	1
Reserved	30:17	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
L2_RAM_COMPAT	16	This bit is set to indicate that the L2 is configured in RAM compatibility mode. This selection is made during IP configuration.	R	IP Config
Reserved	15:13	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
L2_PIPE	12	Setting this bit indicates that the L2 is configured to use pipeline RAMs. This selection is made during IP configuration.	R	IP Config
Reserved	11:10	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
L2_TAGRAM_STALLS	9:8	Number of stall cycles for L2 Tag RAM. Determined by the L2_TagStall pins.	R	Set by hardware pins
Reserved	7:6	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
L2_WSRAM_STALLS	5:4	Number of stall cycles for L2 WS RAM. Determined by the L2_WSStall pins.	R	Set by hardware pins
Reserved	3:2	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
L2_DATARAM_STALLS	1:0	Number of stall cycles for L2 Data RAMS. Determined by the L2_DataStall pins.	R	Set by hardware pins

6.4.5.6 L2 Prefetch Control Register (GCR_L2_PFT_CONTROL, Offset 0x0300)

These registers manage the L2 prefetch control mechanism in the P6600 MPS.

Figure 6.52 L2 Prefetch Control Register Format

31 … 12	11 … 9	8	7 … 0
PAGE_MASK	R	PFTEN	NFPT

Table 6.56 L2 Prefetch Control Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
PAGE_MASK	31:12	This field is a mask that indicates the minimum operating system page size. Address bits larger than 31 default to a bit mask of 1. The default value can change as follows depending on the page size. As the page size increases, less mask bits are required. 4K page size: 0xF_FFFF 8K page size: 0xF_FFFE 16K page size: 0xF_FFFC	R/W	0xF_FFFF
Reserved	11:9	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
PFTEN	8	Prefetch enable. This bit should be set by software only if the number of prefetch units in the NPFT field is greater than zero.	R/W	1
NPFT	7:0	Number of prefetch units. Note that if this field contains a value greater than 0, the PFTEN bit must be set in order for prefetching to occur.	RO	IP Config

6.4.5.7 L2 Prefetch Control Register 2 (GCR_L2_PFR_CONTROL_B, Offset 0x0300)

These registers work in conjunction with L2 Prefetch Control register 2 to manage the L2 prefetch control mechanism in the P6600 MPS.

Figure 6.53 L2 Prefetch Control 2 Register Format

31 … 9	8	7 … 0
Reserved	CEN	PORT_ID

Table 6.57 L2 Prefetch Control Register 2 Descriptions

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:9	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
CEN	8	Code prefetch enable.	R/W	0
PORT_ID	7:0	Enable port ID for L2 prefetching. Each bit in this field corresponds to a CM2 port ID. Each bit of this field is encoded as follows: 0: Requests from the corresponding CM2 port are not monitored for L2 prefetching. 1: Requests from the corresponding CM2 port are monitored for L2 prefetching.	R/W	0xFF

6.5 Core-Local and Core-Other Control Blocks

6.5.1 Core-Local and Core-Other Control Blocks Address Map

A set of these registers exists for each core in the P6600 MPS. These registers can also be accessed from other cores by first writing the *Core Other Addressing Register* (in the Core-Local Control Block) with the proper core number and then accessing these registers using the Core Other Register block.

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCR address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

Table 6.58 Core Local and Core Other Block Register Map (Relative to Core-Local/Core-Other CB Offset)

Register Offset	Name	Type	Description
0x0000	Reserved	-	Reserved
0x0008	Core Local Coherence Control Register (*GCR_CL_COHERENCE GCR_CO_COHERENCE*)	R/W	Controls which coherent intervention transactions apply to the local core.
0x0010	Core Local Config Register (*GCR_CL_CONFIG GCR_CO_CONFIG*)	R	Contains configuration parameters for the Core-Local address space.
0x0018	Core Other Addressing Register (*GCR_CL_OTHER GCR_CO_OTHER*)	R/W	Used to access the registers of another core.
0x0020	Core Local Reset Exception Base Register (*GCR_CL_RESET_BASE GCR_CO_RESET_BASE*)	R/W	Sets the Reset Exception Base for the local core.
0x0028	Core Local Identification Register (*GCR_CL_ID GCR_CO_ID*)	R	Indicates the ID number of the local core.
0x0030	Core Local Reset Exception Extended Base *(GCR_CL_RESET_EXT_BASE GCR_CO_RESET_EXT_BASE)*	R/W	Extends the capabilities of the Core Local Reset Exception Base Register.
0x0040	Core Local TCID_0_PRIORITY Register *(GCR_CL_TCID_0_PRIORITY GCR_CO_TCID_0_PRIORITY)*	R/W	TCID 0 Priority value (2 bits) if IOCU_TYPE=0 in GCR_Cx_CONFIG.
All Others	RESERVED	-	Reserved for future expansion.

6.5.2 Core-Local and Core-Other Control Block Registers

6.5.2.1 Core Local Coherence Control Register (GCR_Cx_COHERENCE Offset 0x0008)

This register allows each core to respond to intervention requests from only a subset of the coherent masters within the P6600 Multiprocessing System (MPS). Software can control entry and exit from the coherence domain by setting the *COH_DOMAIN_EN* bit in this register for:

- Initialization during (asynchronous) boot
- Power control for shutting down and bringing up a core

Table 6.59 Core Local Coherence Control Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:8	Reads as 0. Writes ignored. Must be written with a value of 0x0.	W	0x0
COH_DOMAIN_EN	7:0	Each bit in this field represents a coherent requester within the MPS. Setting a bit within this field will enable interventions to this Core from that requester. The requestor bit which 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. Refer to Section 6.3.12, "Coherency Domains" for more information on the encoding of this field.	R/W	0x0

6.5.2.2 Core Local Config Register

Figure 6.54 Core Local Config Register Format

31 ... 12	11 ... 10	9 ... 0
R	IOCU_TYPE	PVPE

Table 6.60 Core Local Config Register (GCR_Cx_CONFIG Offset 0x0010)

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>RESERVED</td><td>31:12</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
<tr><td>IOCU_TYPE</td><td>11:10</td><td><table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>This is a P6600 core and not an IOCU[1]. Only the P6600 core can access priority values in the GCR_Cx_TCID_n_PRIORITY registers.</td></tr>
<tr><td>0x1</td><td>This is a non-caching IOCU (no intervention port). The IOCU does not access the GCR_Cx_TCID_n_PRIORITY registers.</td></tr>
<tr><td>0x2</td><td>This is a caching IOCU (not currently implemented by MIPS).</td></tr>
<tr><td>0x3</td><td>Reserved</td></tr>
</table>
1. Note that the first encoding is redundant information for convenience. It is possible for the system to determine if a core is an IOCU or not by reading the Global Config register.</td><td>R</td><td>IP Configurable Value</td></tr>
<tr><td>PVPE</td><td>9:0</td><td>Number of VPE’s in the system. Note that in the P6600 core, the term VPE is analogous to a core since there is one VPE per core.

0x000: 1 VPE
0x001 - 0x3FF: Reserved</td><td>R</td><td>0x000</td></tr>
</table>

6.5.2.3 Core-Other Addressing Register

This register must be written with the correct core number before accessing the Core-Other address segment.

Figure 6.55 Core Local Config Register Format

31 ... 16	15 ... 0
CORENUM	R

Table 6.61 Core-Other Addressing Register (GCR_Cx_OTHER Offset 0x0018)

Name	Bits	Description	Read/ Write	Reset State
CORENUM	31:16	Core number of the register set to be accessed in the Core-Other address space.	R/W	0x0
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.-	R	-

6.5.2.4 Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)

This register is used to drive the *SI_ExceptionBase[31:12]* input to the local core. The value is used for placing the exception vectors within the virtual address map during core boot-up time (e.g., when COP0 $Status_{BEV}$ = 1). The value in this register is reset only on Cold Reset (not Warm Reset).

Figure 6.56 Core Local Reset Exception Base Register Format

31 — 12	11 — 0
BEVEXCBASE	R

Table 6.62 Core Local Reset Exception Base Register

Name	Bits	Description	Read/ Write	Cold Reset State
BEVEXCBase	31:12	Bits [31:12] of the virtual address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$=1).	R/W	IP Configuration Value. MIPS Default Value is 0xBFC00
RESERVED	11:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	-

For Core 0, the user can configure the reset location at IP configuration.

Core 0 can write the register to force any of the other cores to use a different reset vector. This register write is done before releasing the other core from reset.

This allows a subset of the processor cores to boot one operating system while another subset of the processor cores boot a different operating system.

6.5.2.5 Core Local Identification Register (GCR_Cx_ID Offset 0x0028)

The aliased memory scheme is normally invisible to software when accessing GCR registers within the Core-Local control block. What actually happens is that an offset is used to make a subset of the GCR registers appear in the Core-Local addressing window.

This register reports the core number that is used as the addressing offset for the Core-Local control block.

Figure 6.57 Core Local Identification Register Format

31 — 0
CORENUM

Table 6.63 Core Local Identification Register

Name	Bits	Description	Read/ Write	Reset State
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.	R	-

6.5.2.6 Core Local Reset Exception Extended Base Register (GCR_Cx_RESET_EXT_BASE Offset 0x0030)

This register is an extension to the Core-Local Reset Exception Base Register (see Section 6.5.2.4 "Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)"). It also is used to drive the *SI_ExceptionBase* input to the local core. The value is used for placing the exception vectors within the virtual address map during core boot-up time (e.g., when COP0 $Status_{BEV}$=1). The value in this register is reset only on Cold Reset (not Warm Reset).

Figure 6.58 Core Local Exception Extended Base Register Format

31	30	29 28	27 20	19 8	7 1	0
EVAReset	UEB	R	BEVExceptionBaseMask	R	BEVExceptionBasePA	PRESENT

Table 6.64 Core Local Reset Exception Extended Base Register

Name	Bits	Description	Read/ Write	Cold Reset State
EVAReset	31	Assertion of this bit indicates to the core to come up in the EVA configuration at reset. This bit is originally set based on the state of the *EVA_Reset* pin during reset.	R/W	IP Configuration Value. MIPS Default Value is 0
UseExceptionBase	30	UseExceptionBase address. This bit reflects the state of the SI_UseExceptionBase pin at reset. In the legacy configuration, if the *SI_UseExceptionBase* pin is not asserted, then the BEV location defaults to 0xBFC0_0000. If the *SI_UseExceptionBase* pin is asserted, address bits *SI_ExceptionBase[31:30]* are forced to a value of 2'b10 to force the BEV location into the KSEG0/KSEG1 space. Refer to Section 3.7.2 in Chapter 3 for more information. This pin is only used in the legacy configuration. There is one *SI_UseExceptionBase* pin per core.	R/W	IP Configuration Value. MIPS Default Value is 1
RESERVED	29:28	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	-
BEVExceptionBaseMask	27:20	This field is used to determine the size of the boot exception vector overlay region from 1 MB to 256 MB in powers of two. This field reflects the state of the *SI_ExceptionBaseMask[27:20]* pins at reset. This field is used to mask bits [27:20] of the virtual address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$ = 1). These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBaseMask* pins per core. Refer to Section 3.7.2 in Chapter 3 for more information.	R/W	IP Configuration Value. MIPS Default Value is 0x00
RESERVED	19:8	Reads as 0x0. Must be written with a value of 0x0.	R	-

Table 6.64 Core Local Reset Exception Extended Base Register *(continued)*

Name	Bits	Description	Read/ Write	Cold Reset State
BEVExceptionBasePA	7:1	BEV exception base physical address. This field contains the upper bits of the physical address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$ = 1).and reflects the state of the *SI_ExceptionBasePA[31:29]* pins at reset. The size of the overlay region defined by *SI_ExceptionBaseMask[27:20]* is remapped to a location in physical address space pointed to by the *SI_ExceptionBasePA[31:29]* pins. 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 *SI_ExceptionBasePA* pins per core. Note that the bits of this field correspond to upper address bits 35:29. Refer to Section 3.7.2 in Chapter 3 for more information.	R/W	IP Configuration Value. MIPS Default Value is 0x00.
PRESENT	0	Reads as 0x1. Writes are ignored	R	1

6.5.2.7 Core Local TCID Registers (GCR_Cx_TCID_PRIORITYOffset 0x0040)

In the P6600 core, there is one thread context per core. Hence only one TCID register is required.

Figure 6.59 Core Local TCID Register Format

31 ... 2	1 ... 0
Reserved	TCID_PRIORITY

Table 6.65 Core Local TCID Register Description

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:2	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_000
TCID_PRIORITY	1:0	TCID priority. This 2-bit value contains the thread context priority level and is encoded as follows: 00: Lowest priority .... 11: Highest priority	R	0x0

6.6 Global Debug Control Block

6.6.1 Global Debug Control Block Address Map

This block holds registers which are used for debugging the CM2 and software which uses the coherence features supplied by the CM2. The registers associated with PDTrace are reset upon assertion of the TAP controller reset. The other registers in this block are reset when the CM2 is reset. TAP reset occurs when *PB_EJ_TRST_N* is asserted or the Test-Logic-Reset TAP state is entered.

Table 6.66 Global Debug Block Register Map (Relative to Global Debug Block Offset)

Register Offset	Name	Type	Reset Source	Description
0x0008	PDTrace TCBControlB Register *(GCR_DB_TCBCONTROLB)*	R/W	TAP	Controls how the TCB deals with the trace information. This register only exists if the CM2 is configured with PDTrace.
0x0010	CM2 PDTrace TCBControlD Register (*GCR_DB_TCBCONTROLD*)	R/W	TAP	Controls CM2 PDTrace. This register only exists if the CM2 is configured with PDTrace.
0x0020	PDTrace TCBControlE Register *(GCR_DB_TCBCONTROLE)*	R/W	TAP	Controls how the TCB deals with trace information. This register only exists if the CM2 is configured with PDTrace.
0x0028	PDTrace TCB Config Register *(GCR_DB_TCBConfig)*	R/W	TAP	Contains trace control block configuration information such as probe width, on-trace memory size, and trace clock ratios.
0x0040	PDTrace TCBSYS Register *(GCR_DB_TCBSYS)*	R/W	TAP	Controls how external logic uses the System Trace interface. Bit 31 is a PRESENT bit and bits [30:0] are completely user defined. The output of this register is available on the TC_Sys_UserCtl pins. This register only exists if the CM2 is configured with PDTrace.
0x0100	CM2 Performance Counter Control Register (*GCR_DB_PC_CTL*)	R/W	CM2	Controls starting/stopping of Performance Counters.
0x0108	PDTrace Trace Word Read Pointer Register *(GCR_DB_TCBRDP)*	R/W	TAP	Pointer into the On-Chip Trace Buffer memory for reads from *GCR_DB_TCBTW_LO* and *GCR_DB_TCBTW_HI* registers. This register only exists if the CM2 is configured with PDTrace.
0x0110	PDTrace Trace Word Write Pointer Register (*GCR_DB_TCBWRP)*	R/W	TAP	Pointer into the On-Chip Trace Buffer memory for the next TraceWord write from *GCR_DB_TCBTW_LO* and *GCR_DB_TCBTW_HI* registers. This register only exists if the CM2 is configured with PDTrace.

Table 6.66 Global Debug Block Register Map (Relative to Global Debug Block Offset)*(continued)*

Register Offset	Name	Type	Reset Source	Description
0x0118	PDTrace Trace Word Start Pointer Register (*GCR_DB_TCBSTP)*	R/W	TAP	Pointer into On-Chip Trace Buffer that is used to determine when all entries in the trace buffer have been filled. This register only exists if the CM2 is configured with PDTrace.
0x0120	CM2 Performance Counter Overflow Status Register (*GCR_DB_PC_OV*)	R/W	CM2	Indicates which performance counters have overflowed.
0x0130	CM2 Performance Counter Event Select Register (*GCR_DB_PC_EVENT*)	R/W	CM2	Selects event type of each performance counter.
0x0180	CM2 Performance Cycle Counter Register (*GCR_DB_PC_CYCLE*)	R/W	CM2	Counts cycles.
0x0190	CM2 Performance Counter 0 Qualifier Register (*GCR_DB_PC_QUAL0*)	R/W	CM2	Performance counter 0 event qualifiers.
0x0198	CM2 Performance Counter 0 Register (*GCR_DB_PC_CNT0*)	R/W	CM2	Performance Counter 0 value.
0x01A0	CM2 Performance Counter 1 Qualifier Register (*GCR_DB_PC_QUAL1*)	R/W	CM2	Performance counter 1 event qualifiers.
0x01A8	CM2 Performance Counter 1 Register (*GCR_DB_PC_CNT1*)	R/W	CM2	Performance Counter 1 value.
0x0200	PDTrace Trace Word Lo Register *(GCR_DB_TCBTW_LO)*	R/W	TAP	Access point to read TraceWords from the On-Chip Trace Buffer memory, Least Significant 32-bits.
0x0208	PDTrace Trace Word Hi Register *(GCR_DB_TCBTW_HI)*	R/W	TAP	Access point to read TraceWords from the On-Chip Trace Buffer memory, Most Significant 32-bits.
All Others	RESERVED			

6.6.2 Global Debug Control Block Registers

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCR address space return 0x0 and writes to those locations should be silently dropped without generating any exceptions.

6.6.2.1 CM2 PDTrace TCB ControlB Register (GCR_DB_TCBCONTROLB Offset 0x0008)

The TCB includes a control register, *GCR_DB_TCBCONTROLB* (0x11). This register configures interfaces to the trace buffer. This register only exists if the CM2 is configured with PDTrace.

The format of the *GCR_DB_TCBCONTROLB* register is shown below, and the fields are described in Table 6.67.

Figure 6.60 PDTrace TCB ControlB Register Format

31	30 28	27 26	25 20	19	18	17	16	15	14	13 12	11	10 8	7	6 2	1	0
WE	R	TWSrcWidth	R	STCE	TRPAD	R	RM	TR	BF	TM	R	CR	Cal	R	OfC	EN

Table 6.67 PDTrace TCB ControlB Register

Fields		Description	Read / Write	Reset State
Name	**Bits**			
WE	31	Write Enable. Only when set to 1 will the other bits of this register be written. This bit will always read 0.	R	0
Reserved	30:28	Reserved. Must be written as zero; returns zero on read.	R	0
TWSrcWidth	27:26	Used to indicate the number of bits used in the source field of the Trace Word. The value for the CM2 is always 2’b10, indicating a four bit source field width.	R	2’b10
Reserved	25:20	This field is used by EJTAG to access other PDTtrace registers. Although the field is R/W via core accesses, this field has no function for core accesses.	R/W	0
STCE	19	System Trace capture enable. When asserted, the System Trace port of the Funnel is enabled to capture System Trace stream data. When not asserted, System Trace stream data is not captured regardless of *TC_Sys_Valid[1:0]* input pin state.	R/W	0
TRPAD	18	Trace RAM access disable bit. When set, core reads and writes to the on-chip trace RAM using GCR accesses are inhibited. If TRPAD is set, memory-mapped writes to the GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI registers have no effect, and memory-mapped reads from GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI do not access the Trace RAM and 0 is returned. Also, when TRPAD is set, then memory-mapped writes to the following registers are inhibited: *TCBTW* *TCBRDP* *TCBWRP* *TCBSTP*	R/W	0
Reserved	17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	16	Read on-chip trace memory. When this bit is set, the read address-pointer of the on-chip memory in register *TCBRDP* is set to the value held in *TCBSTP*. Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer in register *TCBRDP* after each read. When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros. Once set to 1, writing 1 again will have no effect. The bit is reset by setting the TR bit or by reading the last Trace word in *TCBTW*.	R/W	0

Table 6.67 PDTrace TCB ControlB Register *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TR</td><td>15</td><td>Trace memory reset.
When written to one, the address pointers for the on-chip trace memory TCBSTP, TCBRDP and TCBWRP are reset to zero. Also the RM and BF bits are reset to 0.
This bit is automatically reset back to 0, when the reset specified above is completed.</td><td>R/W1</td><td>0</td></tr>
<tr><td>BF</td><td>14</td><td>Buffer Full indicator that the TCB uses to communicate to external software that the on-chip trace memory is full. This bit is cleared when writing a 1 to the TR bit.
This bit has no function if on-chip memory is not implemented.</td><td>R</td><td>0</td></tr>
<tr><td>TM</td><td>13:12</td><td>Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace™ IF to start or stop trace.
<table><tr><th>TM</th><th>Trace Mode</th></tr><tr><td>00</td><td>Trace-To</td></tr><tr><td>01</td><td>Trace-From</td></tr><tr><td>10</td><td>Reserved</td></tr><tr><td>11</td><td>Reserved</td></tr></table>In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around, overwriting older Trace Words, as long as there is trace data coming from the core.
In Trace-From mode, the on-chip trace memory is filled from the point that the core starts tracing until the on-chip trace memory is full (when the write pointer address is the same as the start pointer address). If a TCBTRIGx trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode.
These bits have no function if on-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11</td><td>Read as Zero. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CR</td><td>10:8</td><td>Off-chip Clock Ratio. Writing this field, sets the ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 6.68.
Note: As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per core clock rising edge.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>3’b100</td></tr>
</table>

Table 6.67 PDTrace TCB ControlB Register *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Cal</td><td>7</td><td>Calibrate off-chip trace interface.
If set, the off-chip trace pins will produce the following pattern in consecutive trace clock cycles. If more than 4 data pins exist, the pattern is replicated for each set of 4 pins. The pattern repeats from top to bottom until the Cal bit is de-asserted.
<table>
<caption>This pattern is replicated for every 4 bits of TR_DATA pins.</caption>
<tr><th colspan="4">Calibrations pattern</th></tr>
<tr><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>
Note: The clock source of the TCB and PIB must be running.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>6:2</td><td>Read as Zero. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>OfC</td><td>1</td><td>If set to 1, trace is sent to off-chip memory using TR_DATA pins.
If not set, trace info is sent to on-chip memory.
This bit is read only if one of these options exists.</td><td>R/W</td><td>Preset</td></tr>
<tr><td>EN</td><td>0</td><td>Funnel Trace Enable. When this bit is set, the trace funnels accepts trace information from the CM2, cores, and/or system trace and writes the information to off-chip or on-chip memory.
When this bit is cleared, the trace funnel drops all new trace information from the those sources. The trace information already accepted by the trace funnel is sent to the off-chip or on-chip memory, but new trace information is dropped and not written out.</td><td>R/W</td><td>0</td></tr>
</table>

Table 6.68 Clock Ratio Encoding of the CR Field

Encoding of CR Field	Trace Clock:Core Clock Ratio
3’b000	1:20
3’b001	1:16
3’b010	1:12
3’b011	1:10
3’b100	1:2
3’b101	1:4
3’b110	1:6
3’b111	1:8

6.6.2.2 CM2 PDTrace TCB ControlD Register (GCR_DB_TCBCONTROLD Offset 0x0010)

Figure 6.61 PDTrace TCB ControlD Register Format

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 12	11 8	7	6	5	4 3	2	1	0
R	P6_Ctl	P5_Ctl	P4_Ctl	P3_Ctl	P2_Ctl	P1_Ctl	P0_Ctl	R	TWSrcVal	WB	STEn	IO	TLev	AE	GCE	CME

Table 6.69 CM2 PDTrace TCB ControlD Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:30	Reserved.	R/W	0x0
P6_Ctl	29:28	Provides specific control over tracing transactions on Port 6 of the CM. (the IOCU on 6 core configurations). **Encoding** / **Description** 00 — Tracing Enabled, no Address Tracing 01 — Tracing Enabled with Address Tracing 10 — Reserved 11 — Tracing Disabled	R/W	0x0
P5_Ctl	27:26	Provides specific control over tracing transactions on Port 5 of the CM2 (core 5). See encoding for *P6_Ctl*.	R/W	0x0
P4_Ctl	25:24	Provides specific control over tracing transactions on Port 4 of the CM2 (core 4 on 6 core configurations or the IOCU on 4 core or less configurations). See encoding for *P6_Ctl*.	R/W	0x0
P3_Ctl	23:22	Provides specific control over tracing transactions on Port 3 of the CM2 (core 3). See encoding for *P6_Ctl*.	R/W	0x0

Table 6.69 CM2 PDTrace TCB ControlD Register Descriptions *(continued)*

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>P2_Ctl</td><td>21:20</td><td>Provides specific control over tracing transactions on Port 2 of the CM2 (core 2). See encoding for P6_Ctl.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>P1_Ctl</td><td>19:18</td><td>Provides specific control over tracing transactions on Port 1 of the CM2 (core 1). See encoding for P6_Ctl.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>P0_Ctl</td><td>17:16</td><td>Provides specific control over tracing transactions on Port 0 of the CM2 (core 0). See encoding for P6_Ctl.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>RESERVED</td><td>15:12</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0x0</td></tr>
<tr><td>TwSrcVal</td><td>11:8</td><td>The source ID inserted into the Trace Word by the CM.
NOTE: When disabling trace by setting Global_CM_En to 0, the value in TWSrcVal continues to be used until all trace messages have been flushed from the CM. Therefore, when writing to this register to disabled, the correct value must still be written into the TWSrcVal field.</td><td>R/W</td><td>0xF</td></tr>
<tr><td>WB</td><td>7</td><td>When this bit is set, Coherent Writeback requests are traced. If this bit is not set, all Coherent Writeback requests are suppressed from the CM2 PDTrace Stream.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>ST_En</td><td>6</td><td>System Trace Enable. Driven to the CM2 output pin TC_Sys_Enable. External logic can use this output to control generation of the System Trace stream.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>IO</td><td>5</td><td>Inhibit Overflow on the CM2 PDTrace FIFO full condition. When set to 0, the CM2 will drop a new PDTrace message if the internal PDTrace FIFOs are full.
When set to 1, the CM2 will not drop PDTrace messages, but may stall transactions within the CM2 when the internal PDTrace FIFOs are full.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>TLev</td><td>4:3</td><td>This defines the current trace level being used by CM2 PDtrace:<table><tr><th>Encoding</th><th>Description</th></tr><tr><td>00</td><td>No Timing Information</td></tr><tr><td>01</td><td>Include Stall Times, Causes</td></tr><tr><td>10</td><td>Reserved</td></tr><tr><td>11</td><td>Reserved</td></tr></table></td><td>R/W</td><td>0x0</td></tr>
<tr><td>AE</td><td>2</td><td>When set to 1, address tracing is always enabled for the CM. When set to 0, address tracing may be enabled on a per-port basis through the P<x>_Ctl bits.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>Global_CM_En</td><td>1</td><td>Setting this bit to 1 enables tracing from the CM2 as long as the CM_EN bit is also enabled.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>CM_EN</td><td>0</td><td>This is the master trace enable for the CM. When zero, tracing from the CM2 is always disabled. When set to one, tracing is enabled from whenever the other enabling functions are also true.</td><td>R/W</td><td>0x0</td></tr>
</table>

This register only exists if the CM2 is configured with PDTrace.

6.6.2.3 CM2 PDTrace TCB ControlE Register (GCR_DB_TCBCONTROLE Offset 0x0020)

Figure 6.62 PDTrace TCB ControlE Register Format

31 … 9	8	7	1 … 0	0
R	Tridle	WB	R	PeC

Table 6.70 TCBCONTROLE Register

Name	Bits	Description	Read / Write	Reset State
0	31:26	Reserved for future use. Must be written as zero; returns zero on read.	0	0
UPR	25	Indicates that for 128 bit load/ stores (MSA, if tracing of 128 bit MSA ld/st is not implemented (see bit TraceControl3.MSA) and bonded 2x64) only the lower 64 bits are traced.	R	1
0	24:9	Reserved. Must be written as zeros; returns zeros on reads.	R	0
TrIdle	8	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware. TrIdle is set when the system traces on all cores, and the CM2, have disabled PDTrace and the trace funnel has written all outstanding trace information to the off-chip or on-chip memory.	R	1
0	7:1	Reserved for future use; Must be written as zero; returns zero on read. (Hint to architect, Reserved for future expansion of performance counter trace events).	0	0
PeC	0	Performance Control Tracing is not implemented.	R	0

This register only exists if the CM2 is configured with PDTrace.

6.6.2.4 CM2 PDTrace TCB Config Register (GCR_DB_TCBConfig Offset 0x0028)

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 6.63 PDTrace TCB Config Register Format

31	30 … 21	20 … 17	16 … 14	13 … 11	10 … 9	8 … 6	5	4	3 … 0
CF1	R	SZ	CRMax	CRMin	PW	R	OnT	OfT	REV

Table 6.71 TCBCONFIG Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CF1	31	This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* does not exist, and this bit reads zero.	R	0
Reserved	30:21	Read as Zero. Writes ignored. Must be written with a value of 0x0.	R	0

Table 6.71 TCBCONFIG Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
SZ	20:17	On-chip trace memory size. This field holds the encoded size of the on-chip trace memory. The size in bytes is given by $2^{(SZ+8)}$. i.e., the lowest value is 256 bytes, and the highest is 8 MB. This bit is reserved if on-chip memory is not implemented.	R	Preset
CRMax	16:14	Off-chip Maximum Clock Ratio. This field indicates the maximum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 6.68. This bit is reserved if off-chip trace option is not implemented.	R	Preset
CRMin	13:11	Off-chip Minimum Clock Ratio. This field indicates the minimum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 6.68. This bit is reserved if off-chip trace option is not implemented.	R	Preset
PW	10:9	Probe Width: Number of bits available on the off-chip trace interface *TR_DATA* pins. The number of TR_DATA pins is encoded, as shown in the table. 00: 4 bits 01: 8 bits 10: 16 bits 11: Reserved This field is preset based on input signals to the TCB and the actual capability of the TCB. This bit is reserved if the off-chip trace option is not implemented.	R	Preset
Reserved	8:6	Read as Zero. Must be written with a value of 0x0.	R	0
OnT	5	When set, this bit indicates that on-chip trace memory is present. This bit is preset based on the selected option when the TCB is implemented.	R	Preset
OfT	4	When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module (*TC_PibPresent* asserted).	R	Preset
REV	3:0	Revision of TCB. Indicates the revision of the PDTrace Specification. This field is set to a value of 0x4 to indicate PDTrace revision 8.0 in the P6600 core.	R	0x4

This register only exists if the CM2 is configured with PDTrace.

6.6.2.5 CM2 Performance Counter Control Register (GCR_DB_PC_CTL Offset 0x0100)

Figure 6.64 CM2 Performance Counter Control Register Format

31	30	29	28 — 10
R	Perf-Int_En	Perf_OvF_Stop	R

9	8	7	6	5	4	3 0
P1_Reset	P1_CountOn	P1_Reset	P1_CountOn	Cycl_Cnt_Reset	Cycl_Cnt_CountOn	Perf_Num_Cnt

Table 6.72 CM2 Performance Counter Control Register

Name	Bits	Description	Read/ Write	Reset State
Reserved	31	Read as Zero. Must be written with a value of 0x0.	R	0x0
Perf_Int_En	30	Enable Interrupt on counter overflow. If set to 1, a CM2 performance counter interrupt is generated when any enabled CM2 performance counter overflows.	R/W	0x0
Perf_Ovf_Stop	29	Stop Counting on overflow. If set to 1, all CM2 Performance counters stop counting when any enabled CM2 performance counter overflows i.e., the counter has reached 0xFFFF_FFFF.	R/W	0x0
Reserved	28:10	Read as Zero. Must be written with a value of 0x0.	R	0x0
P1_Reset	9	If set to 1, CM2 Perfomance Counter 1 and *P1_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *P1_Reset* is always read as 0.	R/W	0x0
P1_CountOn	8	Start Counting. If this bit is set to 1 then CM2 Performance Counter 1 and the *P1_Overflow* bit starts counting the specified event. If this bit is set to 0 then CM2 Performance Counter 1 is disabled. This bit is automatically set to 0 if any counter overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0
P0_Reset	7	If set to 1, CM2 Perfomance Counter 0 and *P0_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *P0_Reset* is always read as 0.	R/W	0x0
P0_CountOn	6	Start/Stop Counting. If this bit is set to 1 then CM2 Performance Counter 0 starts counting the specified event. If this bit is set to 0 then CM2 Performance Counter 0 is disabled. This bit is automatically set to 0 if any counter overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0
Cycl_Cnt_Reset	5	If set to 1, the *CM2 Cycle Counter Register* and the *Cycl_Cnt_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *Cycl_Cnt_Reset* is always read as 0.	R/W	0x0
Cycl_Cnt_CountOn	4	Start/Stop the Cycle Counter. If this bit is set to 1 then CM2 Cycle Counter starts counting. If this bit is set to 0 then CM2 Cycle Counter is disabled. This bit is automatically set to 0 if any Counter Overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0
Perf_Num_Cnt	3:0	The number of performance counters implemented (not including the cycle counter). The CM2 has 2 performance counters.	R	0x2

6.6.2.6 CM2 PDTrace TCB Trace Word Read Pointer Register (*GCR_DB_TCBRDP* Offset 0x0108)

The *TCBRDP* register is an address pointer to on-chip trace memory. It points to the TW read when reading the *TCBTW* register. When writing the $TCBCONTROLB_{RM}$ bit to 1, this pointer is reset to the current value of *TCBSTP*.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBRDP* register is shown below and the fields are described in Table 6.73. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 6.65 TCBRDP Register Format

31 … n+1	n … 0
Data	Address

Table 6.73 TCBRDP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written with zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

6.6.2.7 CM2 PDTrace TCB Trace Word Write Pointer Register (GCR_DB_TCBWRP Offset 0x0110)

The *TCBWRP* register is an address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBWRP* register is shown below and the fields are described in Table 6.74. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

Figure 6.66 TCBWRP Register Format

31 … n+1	n … 0
Data	Address

Table 6.74 TCBWRP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

6.6.2.8 CM2 PDTrace TCB Trace Word Start Pointer Register (GCR_DB_TCBSTP Offset 0x0118)

The *TCBSTP* register is the start pointer register. This pointer is used to determine when all entries in the trace buffer have been filled (when *TCBWRP* has the same value as *TCBSTP*). This pointer is reset to zero when the

TCBCONTROLB$_{TR}$ bit is written to 1. If a continuous trace to on-chip memory wraps around the on-chip memory, *TSBSTP* will have the same value as *TCBWRP*.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBSTP* register is shown below and the fields are described in Table 6.75. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

Figure 6.67 TCBSTP Register Format

31 ... n+1	n ... 0
Data	Address

Table 6.75 TCBSTP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

6.6.2.9 CM2 PDTrace TCB System Trace User Control Register (GCR_DB_TCBSYS Offset 0x0040)

The *TCBSYS* register contents are driven to the *TC_Sys_UserCtl[31:0]* output signals. This register is also mapped to offset 0x0040 in the Global Debug Block of the CM GCRs. Thus, any change to this register will be reflected in these output signals. The format of the *TCBSYS* register is shown below, and the fields are described in Table 6.76.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 6.68 TCBSYS Register Format

31	30 ... 0
STA	UsrCtl

Table 6.76 TCBSYS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
STA	31	System Trace Available. Set to 1 if the System Trace Interface is present. Otherwise it is set to 0.	R	Preset
UsrCtl	30:0	User-defined Control.	R/W	0

6.6.2.10 CM2 Performance Counter Overflow Status Register (GCR_DB_PC_OV Offset 0x120)

Figure 6.69 Performance Counter Overflow Status Register Format

31 ... 3	2	1	0
R	P1_OF	P0_OF	Cycl_Cnt_OF

Table 6.77 Performance Counter Overflow Status Register

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
Reserved	31:3	Reserved. Must be written zero, reads back zero.	R	0x0
P1_OF	2	If this bit is set to 1, *CM2 Performance Counter 1* has overflowed i.e., the counter has reached 0xFFFF_FFFF.	R Write 1 to clear	0x0
P0_OF	1	If this bit is set to 1, *CM2 Performance Counter 0* has overflowed i.e., the counter has reached 0xFFFF_FFFF.	R Write 1 to clear	0x0
Cycl_Cnt_OF	0	If this bit is set to 1, the *CM2 Cycle Counter Register* has overflowed.	R Write 1 to clear	0x0

6.6.2.11 CM2 Performance Counter Event Select Register (GCR_DB_PC_EVENT Offset 0x130)

Figure 6.70 CM2 Performance Counter Event Select Register Format

31 ... 16	15 ... 8	7 ... 0
R	P1_Event	P0_Event

Table 6.78 CM2 Performance Counter Event Select Register

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:16	Reserved. Must be written zero, reads back zero.	R	0x0
P1_Event	15:8	Event Selection for CM2 Performance Counter 1. Event numbers are defined in Table 14.1.	R/W	0x0
P0_Event	7:0	Event Selection for CM2 Performance Counter 0. Event numbers are defined in Table 14.1.	R/W	0x0

6.6.2.12 CM2 Cycle Counter Register

The CM2 Cycle Count Register is a 32-bit register that keeps count of CM2 clock cycles. It is controlled through the *Cycl_Cnt_CountOn* and *Cycl_Cnt_Reset* bits in the CM2 Performance Counter Control Register. An overflow of the cycle counter is indicated by a 1 in the *Cycl_Cnt_Overflow* bit in the CM2 Performance Counter Overflow Status Register.

Figure 6.71 CM2 Cycle Count Register Format

31 ... 0
Cycle_Cnt

Table 6.79 CM2 Cycle Counter Register (GCR_DB_PC_CYCLE Offset 0x180)

Name	Bits	Description	Read/ Write	Reset State
Cycle_Cnt	31:0	32-bit count of CM2 clock cycles.	R/W	0x0

6.6.2.13 CM2 Performance Counter n Qualifier Field Register (GCR_DB_PC_QUALn Offset 0x190, 0x1a0)

Figure 6.72 Performance Counter n Qualifier Field Register Format

31 ... 0
Pn_Qualifier

Table 6.80 CM2 Performance Counter n Qualifier Field Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Pn_Qualifier	31:0	CM2 Performance Counter n Event Qualifier. The qualifier corresponds to the event configured through the *Performance Counter 0 Event Select Register*.	R/W	0x0

6.6.2.14 CM2 Performance Counter n Register (GCR_DB_PC_CNTn Offset 0x198, 0x1A8)

Figure 6.73 Performance Counter n Register Format

31 ... 0
Pn_Count

Table 6.81 CM2 Performance Counter n Register

Name	Bits	Description	Read/ Write	Reset State
Pn_Count	31:0	32-bit Performance Counter. The event counted is specified in the *CM2 Performance Counter Event Select Register* and by the corresponding *Qualifier Register*.	R/W	0x0

6.6.2.15 CM2 PDTrace TCB Trace Word LO Register (*GCR_DB_TCBTW_LO* Offset 0x0200)

Reads to this register access the contents of the On-Chip Trace Buffer entry (least significant 32-bits) which is referenced by the *GCR_DB_TCBRDP* register. Writes to this register modify the On-Chip Trace Buffer entry (least significant 32-bits) which is referenced by the *GCR_DB_TCBWRP* register.

A side effect of reading the *TCBTW_LO* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero. A side effect of writing the *TCBTW_LO* register is that the *TCBWRP* register increments to the next TW in the on-chip trace memory. If *TCBWRP* is at the max size of the on-chip trace memory, the increment wraps back to address zero. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 6.74 TCBTW_LO Register Format

31	0
Data	

Table 6.82 TCBTW_LO Register Field Descriptions

Names	Bits	Description	Read / Write	Reset State
Data	31:0	Trace Word, least significant 32-bits.	R/W	0

6.6.2.16 CM2 PDTrace TCB Trace Word HI Register (*GCR_DB_TCBTW_HI* Offset 0x0208)

Reads to this register access the contents of the On-Chip Trace Buffer entry (most significant 32-bits) which is referenced by the *GCR_DB_TCBRDP* register. Writes to this register modify the On-Chip Trace Buffer entry (most significant 32-bits) which is referenced by the *GCR_DB_TCBWRP* register.

To read or write a 64-bit trace word from the Trace Buffer, the *GCR_DB_TCBTW_HI* register must be accessed first before the *GCR_DB_TCBTW_LO* register. The access of the *GCR_DB_TCBTW_LO* register causes the appropriate pointer register to be incremented. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 6.75 TCBTW_HI Register Format

31	0
Data	

Table 6.83 TCBTW_HI Register Field Descriptions

Names	Bits	Description	Read / Write	Reset State
Data	31:0	Trace Word, most significant 32-bits.	R/W	0

Chapter 7

Power Management and the Cluster Power Controller

This chapter describes the Cluster Power Controller (CPC) included in the P6600 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.

The chapter contains the following sections:

7.1 Introduction to the Cluster Power Controller

The Cluster Power Controller (CPC) works in conjunction with the power management features of the individual P6600 cores to provide a comprehensive power management scheme.

The main purpose of the Cluster Power Controller (CPC) is to manage static leakage and dynamic power consumption based on system-level power states assigned to the individual components of the P6600 Multiprocessing System. As such, the CPC acts as a programmable platform peripheral, accessible through cluster CPU software and SOC-level hardware protocols.

The CPC is an integral part of the coherent cluster and is designed to boostrap, reset, tree root clock-gate and power-gate cluster CPUs and the Coherence Manager. Implementors may or may not chose 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:

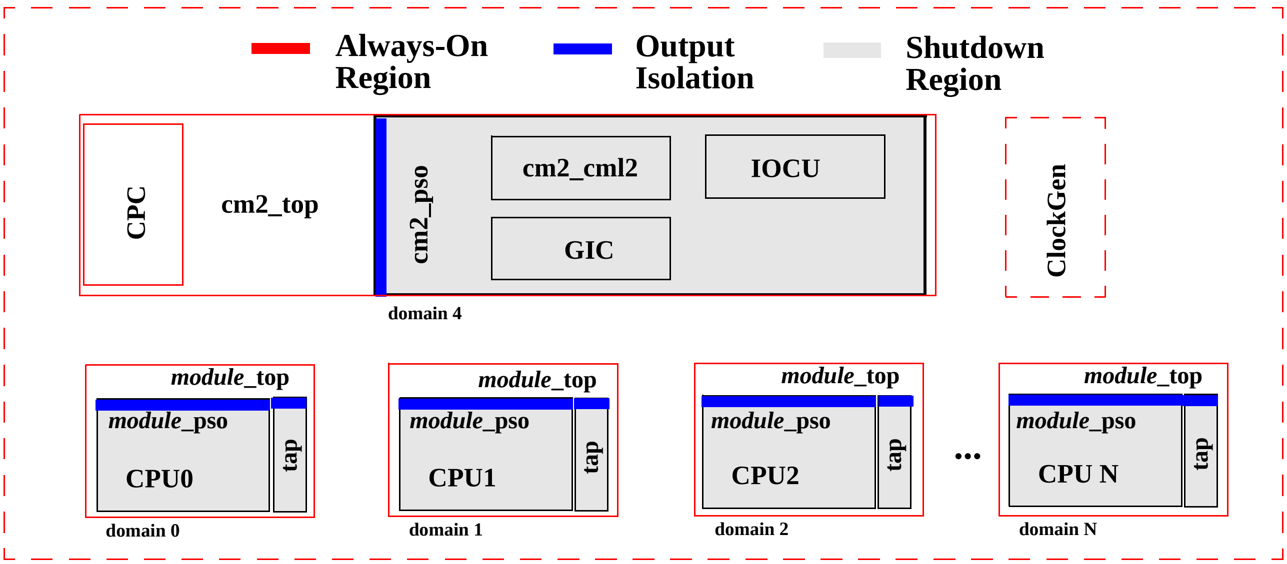
- **Power gating of selected CPUs and/or the CM.** Supported by industry-standard physical design flows, supply voltage of individual power domains can be switched on-chip. Currently, the Common Power Format (CPF) and Unified Power Format (UPF) are provided for a seamless front to back-end design flow. Besides CPF/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 in order 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 boostrap 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 EJTAG debug probe access to cores by detecting the connection of a probe and enabling cores to respond to debug interrupt requests.

7.1.1 Power Domains of the P6600 Multiprocessing System

Figure 7.1 P6600 Multiprocessing System Power Domains



To individually power gate each core, independently controlled power domains are introduced to the P6600 core. 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. 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 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 sub-domains.

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

Level 2 cache is part of the CM2. However, power management of the L2 cache is not handled by the CPC. The CMP cluster implementation ensures that power-down of cores and Coherence Manager does not affect L2 status.

7.1.2 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.

7.1.2.1 Coherent to Non-Coherent Mode Transition

To leave the coherent domain and operate independently or prepare for shutdown, the following sequence should be followed:

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 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.

7.1.2.2 Non-Coherent to Coherent Mode Transition

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

- Caches must be initialized first (since last reset)
- 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.
- The GCR local coherence control register is programmed to add the core to the coherent domain.
- Switch to coherent Cache Coherence Attribute (CCA).
- Regular coherent programs can now start on this core.

7.1.2.3 Non-Coherent to Power Down Mode Transition

A core which is not member of a coherent domain is powered down. NOTE: When an EJTAG probe is detected, the CPC will prevent power down to preserve the connectivity of the TAP scan chain. A power-down command will instead cause the core to enter clock off mode.

- The GIC might be programmed to re-route interrupts away from this core.
- The CPC must be programmed to enter power-down mode.
- 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.

- The CPC initiates the clock and power shutdown micro-sequence.

7.1.2.4 Non-Coherent to Clock Off Mode Transition

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

- Programming a CPC ClkOff command will disable the clock tree root for this core.
- 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.
- The GIC might be programmed to re-route interrupts for this core to others.

7.1.2.5 Clock Off to Power Down Mode Transition

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

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

7.1.2.6 Clock Off to Non-Coherent Mode Transition

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

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

7.1.2.7 PowerDown to Non-Coherent Mode Transition

A core is powered up and becomes operational.

- 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.
- 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.
- The CPC will schedule 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.
- The GIC might be reprogrammed to perform interrupt routing to this core.

7.2 CPC Register Programming

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

7.2.1 Requestor Access to CPC Registers

The CPC allows up to eight requestor's in a system. A requestor can be either a core or an IOCU. The P6600 core allows up to 7 requestors in a multiprocessing system; six cores and one IOCU.

The requestor's may not have unrestricted access to the CPC registers. During boot time, software determines which requestor's are provided access to the CPC registers by programming the 8-bit *CPC_ACCESS_EN* field of the *Global CPC Access Privilege* register located at offset 0x000. Each bit in this field corresponds to a specific requestor.

The MIPS default for this field is 0xFF, meaning that all requestor's in the system have access to the CPC register set. To disable access to the registers for a particular requestor, software need only clear the corresponding bit of this field to zero and all write requests to the CPC registers by that requestor will be ignored.

7.2.2 Global Sequence Delay Count

The Sequence Delay register (*CPC_SEQDEL_REG*) located at offset 0x0008 in the CPC Global Control Block, contains a 10-bit field that describes the number of clock cycles each domain micro-sequencer will take to advance. It describes a set of worst-case timing of the physical implementation and is used to ensure electrical and bus protocol integrity. Typically, the *CPC_SEQDEL_REG* contents would be defined at IP configuration time. However, runtime write capability allows fine tuning to optimize sequencer timing. Domain sequencing begins once the RAILDELAY field has counted down to zero. Refer to Section 7.2.3, "Rail Delay" for more information.

The 10-bit MICROSTEP field is encoded as follows:

Table 7.1 Encoding of MICROSTEP Field

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

Note that the physical implementation might not allow power sequence micro steps to advance with full cluster speed. At cluster cold start, the counter divides cluster frequency by a hard coded IP configuration value to derive a micro step width.

7.2.3 Rail Delay

The Rail Delay register (*CPC_RAIL_REG*) located at offset 0x010 in the CPC Global Register Block contains a 10-bit counter field (*RAILDELAY*) used to schedule delayed start of power domain sequencing after the *RailEnable* signal has been activated by the CPC. This allows the CPC to compensate for slew rates at the gated rail, since hardware interlocks such as *SI_VddOk* are either unavailable or don't reflect to complete power up time of a domain.

The 10-bit counter value delays the power-up sequence per domain after the *SI_RailStable* and *VddOK* signals become active. The power-up micro-sequence starts after RAILDELAY has been loaded into the internal counter and a count-down to zero has concluded.

After completion of the domain power-up micro-sequence, the DomainReady signal is raised and can be used for domain daisy-chaining.

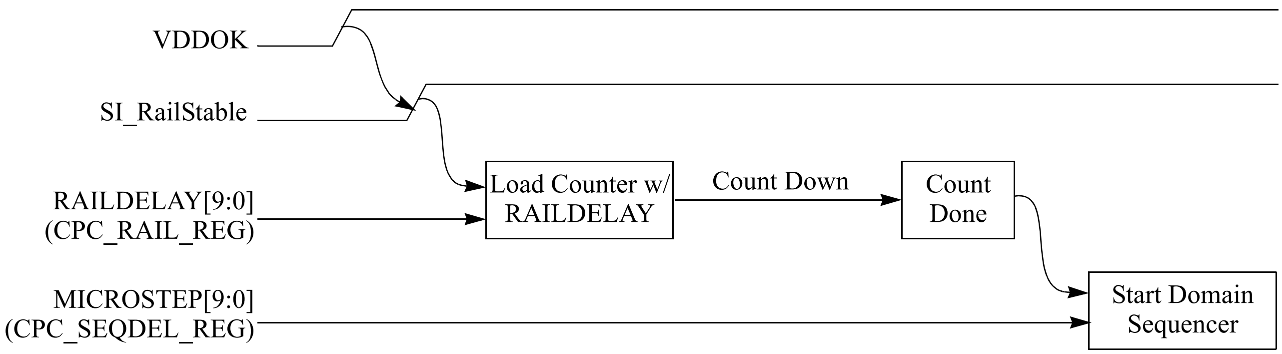
At IP configuration time, the contents of the *CPC_RAIL_REG* register are preset. However, for fine tuning, the register can be written at run time.

The 10-bit RAILDELAY field is encoded as follows:

Table 7.2 Encoding of RAILDELAY Field

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

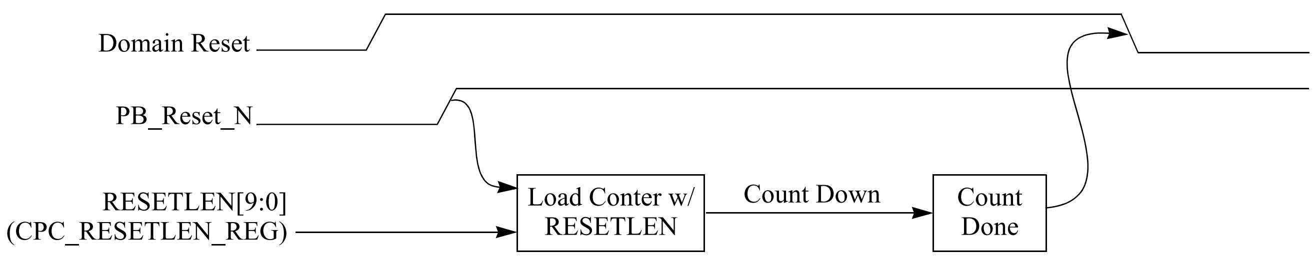
Figure 7.2 Relationship Between RAILDELAY and MICROSTEP During Power-Up Sequence



7.2.4 Reset Delay

Within the power-up micro-sequence, reset is applied. Typically, reset is active until the domain responds by asserting the internal *PB_Reset_N* signal. However, the *CPC_RESETLEN_REG* allows reset to be extended beyond the assertion of *PB_ResetN*. The down-counter starts after the sequencer has detected the assertion of *PB_Reset_N*. Domains without a *PB_ResetN* signal could tie this input low or connect it to an inverted reset signal.

Figure 7.3 Extending the Reset Sequence Beyond the Assertion of the Reset Signal



7.2.5 Executing a Power Sequence

The power sequence for the CPC block support the following commands:

- ClockOff: This command causes the domain to cycle into clock-off mode.It disables the clock to this power domain.
- PwrDown: This command uses the setup values in the CPC_STAT_CONF_REG register.
- PwrUp: This command uses the setup values in the CPC_STAT_CONF_REG register.
- Reset: When this command is issued, the domain is reset if it is in non-coherent mode.

A command can be executed in the local core by writing and encoded value to bits 3:0 of the Command register (CPC_CL_CMD_REG) of the Core-Local block located at offset address 0x000. To write a command to another core, bits 3:0 of the Command reigster (CPC_CO_CMD_REG) in the Core-Other block is used.

7.2.6 Accessing Another Core

To access another core, the number of the core to be accessed is programmed into bits 23:16 of the Core-Other Addressing register (CPC_CL_OTHER_REG) located at offset 0x010 of the Core-Local block. This field selects the core number of the register set to be accessed in Core-Other address space. Refer to Section 7.3.4.2, "Core-Other Addressing Register" for more information.

7.3 Cluster Power Controller Address Map

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

In Table 7.3, 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 CM2. Refer to Chapter 8, *CM2 Global Control Registers* for more information on this register.

Table 7.3 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 P6600 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 P6600 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.

7.3.1 Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in Table 7.3 above 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_CPC_BASE field of the *GCR CPC Base* register located at offset address 0x0088 in the Global Control Block of the CM2 registers. Note that this Global Control Block is different from the one listed in Table 7.3 above. Refer to the *GCR_CPC_BASE Register in* Chapter 8, *CM2 Global Control Registers* for more information on this register.

To determine the physical address of each block listed in Table 7.3, 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 7.4. Note that an example base address of 0x1BDE_0 is used for these calculations.

Table 7.4 Example Physical Address Calculation of the CPC Register Blocks

Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x00_1BDE_0	+	0x0000 - 0x1FFF	=	0x00_1BDE_ 0000 - 0x1BDE_1FFF	8 KB	CPC Global Control Block.

Table 7.4 Example Physical Address Calculation of the CPC Register Blocks *(continued)*

Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x00_1BDE_0	+	0x2000 - 0x3FFF	=	0x00_1BDE_ 2000 - 0x00_1BDE_3FFF	8 KB	CPC Core-Local Control Block.
0x00_1BDE_0	+	0x4000 - 0x5FFF	=	0x00_1BDE_ 4000 - 0x00_1BDE_5FFF	8 KB	CPC Core-Other Control Block.

7.3.2 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 7.4 above. 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 Table 7.5. In this table an example base address of 0x1BDE_0 is used.

Table 7.5 Absolute Address of Individual CPC Global Control Block Registers

MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address	Global Control Register
0x00_1BDE_0	+	0x0000	+	0x0000	=	0x00_1BDE_0	CPC Access Privilege.
0x00_1BDE_0	+	0x0000	+	0x0008	=	0x00_1BDE_0	CPC Global Sequence Delay.
0x00_1BDE_0	+	0x0000	+	0x0010	=	0x00_1BDE_0	CPC Rail Delay.
0x00_1BDE_0	+	0x0000	+	0x0018	=	0x00_1BDE_0	CPC Reset Length.
0x00_1BDE_0	+	0x0000	+	0x0020	=	0x00_1BDE_0	CPC Revision.

Table 7.6 shows the absolute physical addresses for the CPC Core-Local block. In this table an example base address of 0x1BDE_0 is used.

Table 7.6 Absolute Address of Individual CPC Core-Local Block Registers

MIPS Default Base		Core-Local Register Block Offset		Core-Local Register Offset		Absolute Physical Address	Core-Local Register
0x00_1BDE_0	+	0x2000	+	0x0000	=	0x00_1BDE_2000	CPC Core-Local Command.
0x00_1BDE_0	+	0x2000	+	0x0008	=	0x00_1BDE_2008	CPC Core-Local Status and Configuration.
0x00_1BDE_0	+	0x2000	+	0x0010	=	0x00_1BDE_2010	CPC Core-Other Addressing.

Table 7.6 shows the absolute physical addresses for the CPC Core-Other block. In this table an example base address of 0x1BDE_0 is used.

Table 7.7 Absolute Address of Individual CPC Core-Other Block Registers

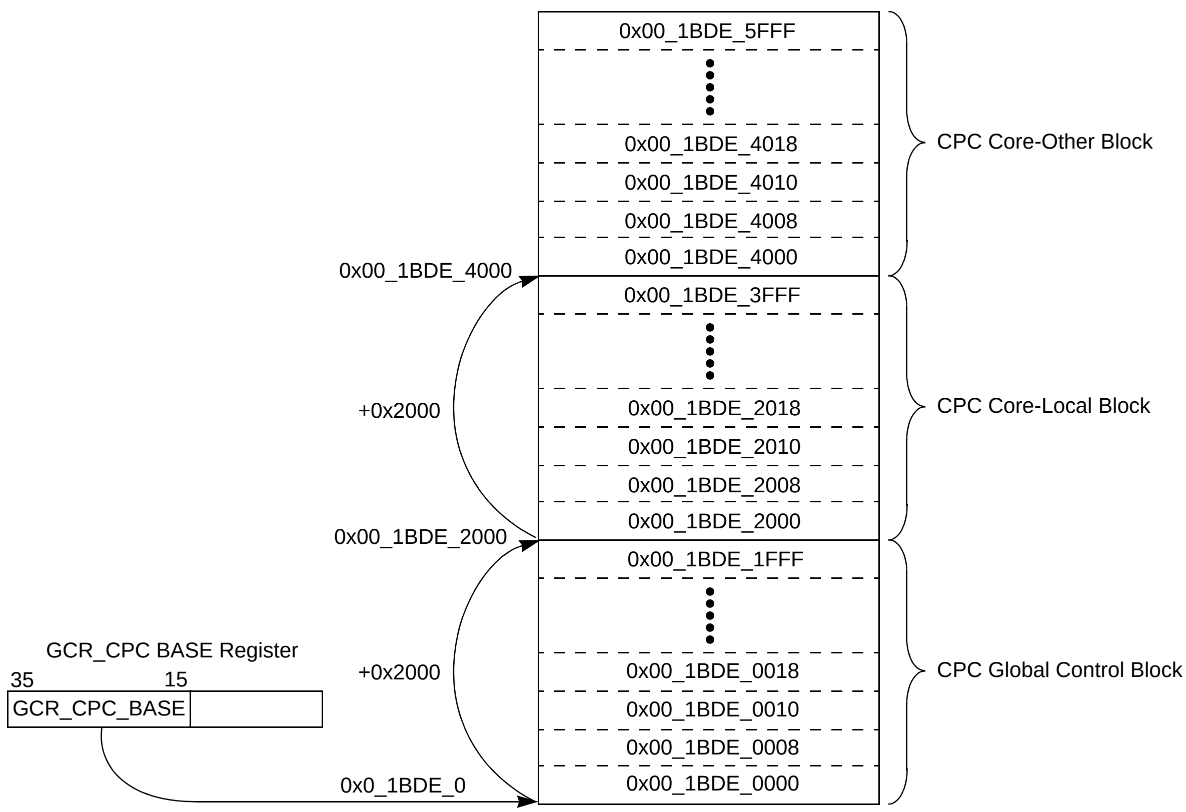
MIPS Default Base		Core-Other Register Block Offset		Core-Other Register Offset		Absolute Physical Address	Core-Other Register
0x00_1BDE_0	+	0x4000	+	0x0000	=	0x00_1BDE_4000	CPC Core-Other Command.

Table 7.7 Absolute Address of Individual CPC Core-Other Block Registers*(continued)*

MIPS Default Base		Core-Other Register Block Offset		Core-Other Register Offset		Absolute Physical Address	Core-Other Register
0x00_1BDE_0	+	0x4000	+	0x0008	=	0x00_1BDE_4008	CPC Core-Other Status and Configuration.
0x00_1BDE_0	+	0x4000	+	0x0010	=	0x00_1BDE_4010	CPC Core-Other Addressing.

This concept is described in Figure 7.4 below. In this figure an example base address of 0x1BDE_0 is used.

Figure 7.4 CPC Register Addressing Scheme Using an Example Base Address of 0x1BDE_0



7.3.3 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 7.8 Global Control Block Register Map (Relative to Global Control Block offset)

Register Offset in Block	Name	Type	Description
0x000	CPC Global CSR 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.
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.	-	For Future Extensions

7.3.3.1 Global CSR Access Privilege Register

Table 7.9 CPC Global CSR Access Privilege Register (CPC_ACCESS_REG Offset 0x000)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CM_ACCESS_EN	7:0	Each bit in this field represents a power domain CPU. If the bit is set, that requester is able to write to the CPC registers (this includes all registers within the Global, Core-Local and Core-Other blocks. If the bit is clear, any write request from that requestor to the CPC registers (Global, Core-Local, Core-Other) will be dropped.	R/W	0xff

The Access privilege register configures the CPU access rights towards CPC programming registers. Its function is defined equally to the GCR Access Privilege Register.

7.3.3.2 Global Sequence Delay Counter

The *CPC_SEQDEL_REG* describes globally the number of clock cycles each domain micro-sequencer will take to advance. It describes a set of worst-case timing of the physical implementation and is used to ensure electrical and bus protocol integrity. Mainly, buffer tree delays on *SI_Isolate* and/or *SI_RailEnable* can be used to set proper micro sequencer delay values.

Typically, the *CPC_SEQDEL_REG* contents would be defined at IP configuration time. However, runtime write capability allows fine tuning to optimize sequencer timing.

Table 7.10 Global Sequence Delay Counter Register (CPC_SEQDEL_REG, Offset 0x008)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
MICROSTEP	9:0	This field reflects the delay in clock cycles, taken by each power domain micro-sequencer to advance between atomic micro steps. Cycles/Step = MICROSTEP[9:0] value + 1; 0 => 1cycle, 1 => 2cycles... Physical implementation might not allow power sequence micro steps to advance with full cluster speed. At cluster cold start, the counter divides cluster frequency by a hardcoded IP configuration value to derive a micro step width.	R/W	IP Configuration Value

7.3.3.3 Global Rail Delay Counter

The *CPC_RAIL_REG* represents a 10-bit counter register to schedule delayed start of domain operation after the *RailEnable* signal has been activated by the CPC. This allows to compensate for slew rates at the gated rail, since hardware interlocks such as *SI_VddOk* are either unavailable or don't reflect to complete power up time of a domain.

At IP configuration time, the contents of *CPC_RAIL_REG* is preset. However, for fine tuning, the register can be written at run time.

Table 7.11 Global Rail Delay Counter Register (CPC_RAIL_REG, Offset 0x010)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
RAILDELAY	9:0	10-bit counter value to delay power-up sequence per domain after *RailStable* and *VddOK* signals became active. The power-up micro-sequence starts after RAILDELAY has been loaded into the internal counter and a counted down to zero has concluded. After completion of the domain power-up micro-sequence, the *DomainReady* signal is raised and can be used for domain daisy-chaining.	R/W	IP Configuration Value

7.3.3.4 Global Reset Width Counter

Within the power-up micro-sequence, reset is applied. Typically, reset is active until the domain responds with *PB_Reset_N* feedback. However, the *CPC_RESETLEN_REG* allows reset to be extended beyond the *ResetN* feedback, or in case the reset feedback is unavailable. Counting down will start after the sequencer has received the *PB_Reset_N* feedback. Domains without *PB_ResetN* feedback could tie this input low or connect it to an inverted reset signal.

Table 7.12 Global Reset Width Counter Register (CPC_RESETLEN_REG, Offset 0x018)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
RESETLEN	9:0	10-bit counter value to extend reset duration beyond *PB_Reset_N* feedback. The domain behavior after reset is determined by the domain local setup register.	R/W	IP Configuration Value

7.3.3.5 Revision Register

Table 7.13 Revision Register (CPC_Revision_REG, Offset 0x020)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	
MAJOR_REV	15:8	This field reflects the major revision of the CPC block. A major revision might reflect the changes from one product generation to another.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the CPC block. A minor revision might reflect the changes from one release to another.	R	Preset

7.3.4 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 P6600 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 listed in Table 7.14.

Table 7.14 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.

Table 7.14 Core-Local Block Register Map *(continued)*

Register Offset in Block	Name	Type	Description
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 CM2	Used to access local registers of another core.
0x018 0x0F8	CPC Local RESERVED registers	-	For Future Extensions

The register offsets shown are relative to the offsets listed in Table 7.15.

Table 7.15 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

CPC Local register are used to set power-down conditions. After setup of conditions, the micro-sequencer can be activated through the command register. The execution of the micro-sequencer can be observed via the status register. Reading the status and configuration register retrieves the last executed command and status flags to reflect on recent commands given.

7.3.4.1 Command Register

Table 7.16 Local Command Register (CPC_CL[CO]_CMD_REG, Offset 0x000)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>RESERVED</td><td>31:4</td><td>Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CMD</td><td>3:0</td><td>Requests a new power sequence execution for this domain. Read value is the last executed command.
<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td>4’d1</td><td>ClockOff
This command causes the domain to cycle into clock-off mode.It disables the clock to this power domain. Only successful if SI_CoherenceEnable and other protocol interlocks are observed. If not, the command remains inactive until the protocol barriers subside. After that, the command is executed.
Depending on the current sequencer state, the command either causes power-up of a domain, or a domain leaves active duty to become inactive. A power-up leads to sequencer state U2, which will require the execution of a subsequent Reset or PwrUp command to make this domain operational.</td></tr>
<tr><td>4’d2</td><td>PwrDown
this domain using setup values in CPC_STAT_CONF_REG. Only successful if SI_CoherenceEnable inactive and all protocol interlocks are observed. If not, the command remains inactive until the protocol barriers subside. Then, the command is executed.</td></tr>
<tr><td>4’d3</td><td>PwrUp
this domain using setup values in CPC_STAT_CONF_REG. Usable only for Core-Others access. It is the software equivalent to SI_PwrUp hardware signal</td></tr>
<tr><td>4’d4</td><td>Reset
This domain is reset if in non-coherent mode. After the domain has been reset, the domain becomes operational and the CMD field reads as PwrUp cmd.</td></tr>
<tr><td>Others</td><td>Reserved</td></tr>
</table>
</td><td>R/W
Not available in CM domain</td><td>0</td></tr>
</table>

Table 7.17 Local Status and Configuration Register (CPC_CL[CO]_STAT_CONF_REG, Offset 0x008)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>RESERVED</td><td>[31:24]</td><td>Reserved.</td><td>R</td><td>0</td></tr>
<tr><td>PWRUP_EVENT</td><td>23</td><td>The SI_PowerUp pin had been activated and caused the sequencer to cycle into power up state. The event also caused the sequencer to place a PwrUp command into the CMD field. Writing a 0 into the PWRUP_EVENT field will clear this bit.</td><td>R/W0</td><td>0</td></tr>
<tr><td>SEQ_STATE</td><td>[22:19]</td><td>Current domain sequencer state. State description:
<table>
<tr><th>Code</th><th>State</th></tr>
<tr><td>4’h0</td><td>D0 - PwrDwn</td></tr>
<tr><td>4’h1</td><td>U0 - VddOK</td></tr>
<tr><td>4’h2</td><td>U1 - UpDelay</td></tr>
<tr><td>4’h3</td><td>U2 - UClkOff</td></tr>
<tr><td>4’h4</td><td>U3 - Reset</td></tr>
<tr><td>4’h5</td><td>U4 - ResetDly</td></tr>
<tr><td>4’h6</td><td>U5 - nonCoherent execution</td></tr>
<tr><td>4’h7</td><td>U6 - Coherent execution</td></tr>
<tr><td>4’h8</td><td>D1 - Isolate</td></tr>
<tr><td>4’h9</td><td>D3 - ClrBus</td></tr>
<tr><td>4’ha</td><td>D2 - DClkOff</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>RESERVED</td><td>18</td><td>Reserved.</td><td>R</td><td>-</td></tr>
<tr><td>CLKGAT_IMPL</td><td>17</td><td>If set, this domain is implemented with clock tree root gating.
If cleared, the CPC will still execute power-down/clock-off sequences if commanded; however, no physical clock gating is performed.</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>PWRDN_IMPL</td><td>16</td><td>If set, this domain is implemented as power-gated.
If cleared, the CPC will still execute power-down sequences if commanded; however, no physical power switching is performed.</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>EJTAG_PROBE</td><td>15</td><td>An EJTAG probe connection event has been seen. The domain powers up if required and observes a reset sequence. Thereafter the core transitions into clock-off mode. After a probe has been seen once, the power domain will not assume power-off mode until this bit is written to zero or the CPC experiences a cold reset.</td><td>R/W0</td><td>0</td></tr>
<tr><td>Reserved</td><td>14:11</td><td>Reserved.</td><td>R</td><td>0</td></tr>
<tr><td>Reserved</td><td>10</td><td>Reserved.</td><td>R/W</td><td>1</td></tr>
</table>

Table 7.17 Local Status and Configuration Register (CPC_CL[CO]_STAT_CONF_REG, Offset 0x008)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PWUP_POLICY</td><td>[9:8]</td><td>Each CPC domain sequencer is hardwired through the SI_ColdPwrUp signal to either power up, remain power-gated, go into clock-off mode, or become operational. To influence the cold start behavior of the domain, three distinct policies can be wired for this domain:
<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td>2’b00</td><td>This CPU remains powered down after a system cold start. A later PwrUp or Reset command, or SI_PwrUp signal assertion will make this domain operational.</td></tr>
<tr><td>2’b01</td><td>Go into Clock-Off mode. Disables domain clock after power-up sequence. Core will wake up through a CPC PwrUp or Reset command or a SI_PwrUp signal assertion. In this Clock-Off mode, the core will not be initialized and its boundary isolation will be maintained.</td></tr>
<tr><td>2’b10</td><td>Power up this domain after system cold start. The CPU will be reset and become operational based on its boot vector contents.</td></tr>
<tr><td>2’b11</td><td>Reserved</td></tr>
</table>
Within a processor cluster, CPU zero would power-up, while peer CPU 1-3 remain unpowered until released through a PwrUp commands. The PWUP_POLICY field reflects the hardwired SI_ColdPwrUp bus.</td><td>R</td><td>Hardwired IP Configuration Value

CM domain is hard coded to powerUp if any CPU domain is powered up initially.</td></tr>
<tr><td>RESERVED</td><td>[7:5]</td><td>Reads zero. Writes ignored</td><td>R</td><td>0</td></tr>
<tr><td>IO_TRFFC_EN</td><td>[4]</td><td>Enable CM for stand alone IOCU traffic. Setting this bit changes the low power state of the CM power domain from PwrDwn to ClkOff. The CM_IOPwrUp signal can be used byan external device to enable the CM to perform IOCU data transfers without CPU activities.
Deselecting IO_TRFFC_EN will power down the CM if all CPUs are powered down. In this case, CM_IOPwrUp signal activity is not observed by the CPC.
A powered down CM domain will clear all preset CM/IOCU control registers. Powering up due to CPU power-up will send the CM/IOCU through a reset sequence, together with the CPU.</td><td>R/O for CPUs, read zero

R/W for CM</td><td>0</td></tr>
<tr><td>CMD</td><td>3:0</td><td>Reflects most recent placed sequencer command. See definition in CPC_CMD_REG Table 7.3.4.1. The sequencer will overwrite the field after a Reset command, or SI_PwrUp signal caused power up of the domain. The command reads then as PwrUp.</td><td>R</td><td>0</td></tr>
</table>

7.3.4.2 Core-Other Addressing Register

This register must be written with the correct CoreNum value before accessing the Core-Other address segment. This register is not available within the CM local domain. Read access to the CM *CPC_OTHER_REG* will yield zero. Writes are ignored.

Table 7.18 Core-Other Addressing Register (CPC_CL[CO]_OTHER_REG Offset 0x010)

Register Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
RESERVED	31:19	Reads as 0. Writes ignored. Must be written with a value of 0x000.	R	0
CORENUM	18:16	CoreNum of the register set to be accessed in the Core-Other address space.	R/W	0x0
RESERVED	15:0	Reads as 0. Writes ignored. Must be written with a value of 0x0000.	R	0

7.4 Cluster Power Controller Commands

The CPC provides a set of commands to establish a desired power domain state. CPC commands are:

- **ClockOff** - a power domain is brought into ClockOff state as programmed into the *CPC_CMD_REG* Table 7.3.4.1. If the domain was powered down before, the power-on sequence is applied according to *CPC_STAT_CONF_REG* settings. If the domain was active before and was in non-coherent operation, the domain is brought into ClockOff state D2. A domain in ClockOff state can be sent into operation using the PwrUp command. A ClockOff command given to a domain in coherent operation will remain inactive until the CPU has left the coherent mode of operation. Sending a ClkOff command to the CPC before a previous command completed will cause the CPC domain target to be redirected towards ClockOff. However, the previous steady state can be observed temporarily before the newly programmed state is reached.

- **PwrDwn** - a power domain is powered down into state D0. *CPC_STAT_CONF_REG* and *CPC_CMD_REG* settings determine the sequence observed by the CPC. Note, both register settings are observed dynamically. The sequencer will preempt an in flight command at the next steady state to execute the newly given command.

- **PwrUp** - the execution of this command depends on the previous domain power state. If the domain is powered down to state D0, a PwrUp command will enable power for the domain and bring the domain into operational state U5. However, if *SI_CoherenceEnable* is active, the domain will advance into state U6 - coherent operation. Please note, that a set of software initialization needs to complete to safely bring a non-coherent core into coherent state. If the previous power domain state was ‘ClkOff’, a PwrUp command will raise the domain state to either non-coherent or coherent operation, dependent on the GCR coherence status settings. This will be domain state U5 and U6 respectively.

  When bringing a domain up after a PwrDwn command is executed, the Reset command is generally preferable to PwrUp. If the domain did not reach state D0 or was prevented from entering D0 because an EJTAG probe was connected, the CPC may identify that a reset is not required for PwrUp and will simply restart the clocks. This may be fine, but also may cause some problems. One common example where a reset is required is if the core enters an infinite loop after requesting PwrDwn.

  A PwrUp command given to an active domain in non-coherent or coherent operation U5/U6 has no effect.

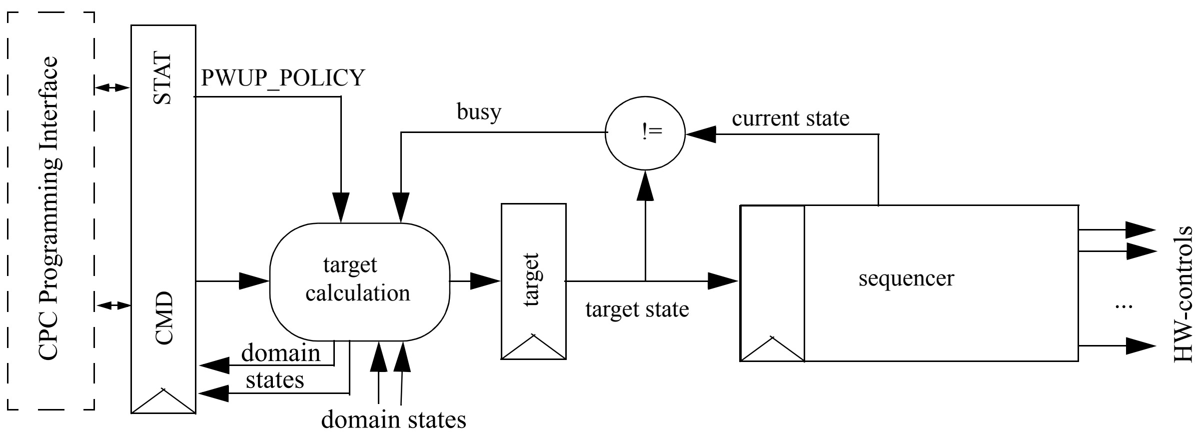
If a PwrUp command is given to the CPC while a previous command is still in flight, the command is placed in the CPC command register and is executed at the earliest possible state, i.e., when the sequencer has reached a non-transitional state.

The hardware *SI_PwrUp* signal activated for this domain will always bring the core into power-up mode with enabled clocks. The *PWUP_POLICY* settings of *CPC_STAT_CONF_REG* have no effect on hardware wake-ups. Also, the hardware wake-up has priority over software commands.The *PWRUP_EVENT* bit of *CPC_STAT_CONF_REG* is set after a hardware power-up has been executed.

- **Reset** - this command allows a domain in non-coherent operation (state U5) to be reset. It also can be sent to a domain in power-down or clock-off mode. The domain will then become active, and a reset sequence is executed which leads to an operational steady state of the domain (U5 or U6, dependent on GCR programming).

Figure 7.5 details the CPC domain command execution. A command given to a CPC power domain will be translated into a domain target state, and the domain sequencer will progress towards this target. A new command is accepted as soon as a suitable state transition is found within the traversed states. Domain sequencer states translate directly to hardware control signals for reset and power gating, as depicted in Figure 7.5.

Figure 7.5 CPC Command Execution



7.5 P6600 Core Power Management Options

In addition to the Cluster Power Controller described in the previous sections, MIPS Technologies provides a mechanism for reducing power in the P6600 core depending on the work load. The conditions under which the P6600 core is placed in power-down mode are determined by the SOC.

The information in the following sections should be used only when all cores in the system are shut down. The processor and cache states need not be saved for each core shut down as long as their is one core operation. However, once the last core is to be shutdown by the SOC, the following procedure can be used to save the processor state.

There are two basic options for power management in the P6600 core.

1. Clock gating: Used to stop the clocks and put the core into sleep mode. Refer to Section 7.6, "P6600 Core Clock Gating" for more information. In this mode the VDD levels are maintained and power is preserved, so no data is lost.

2. Power gating: Used to shut down power to selected parts of the P6600 core. In this mode certain elements of the core, such as registers, caches, TLB, etc. are saved, allowing for a more efficient power-up process. Refer to Section 7.7, "P6600 Core Power Gating" for more information.

7.6 P6600 Core Clock Gating

Clock gating provides a way for the P6600 core to shut down the core clock under certain conditions. The mechanism used to suspend and then resume the core clock depends on the power management options selected during the core configuration process. These options include;

- Enabling of 'top level clock gating'
- Enabling of 'fine grain clock gating'

7.6.1 Designs Implementing Top Level Clock Gating

Top level clock gating is provided as an option during the core configuration process. For designs implementing top level clock gating, the P6600 core can be placed into sleep mode using the WAIT instuction.

When the WAIT instruction is executed during normal operation, the P6600 core completes all outstanding operations, then freezes the pipeline and asserts the SI_SLEEP signal, indicating to external logic that the P6600 core has entered sleep mode.

If top level clock gating is enabled, the processor turns off the internal clock to most of the P6600 core automatically once SI_SLEEP is asserted. The clock is maintained only for a small amount of logic that waits for an interrupt intended to bring the processor out of sleep mode. In addition to the interrupt logic, the following signals also remain active in sleep mode;

- SI_INT[5:0]
- SI_NMI
- SI_RESET
- EJ_DEBUGM

Once the clocks are suspended, the entire contents of the processor, including registers, caches, and TLB, are saved. Once the ‘wake’ interrupt is received, the processor restarts its internal clock and can resume normal operation within a few clock cycles. The ‘wake’ interrupt can be any enabled interrupt, NMI, or debug interrupt. This is the fastest and most efficient mechanism to transition the P6600 core in and out of sleep mode.

Note that the SI_RESET signal can also be used to exit sleep mode. However, assertion of SI_RESET causes all internal data to be lost and the registers to revert back to their default values.

7.6.1.1 Reduction of VDD During Sleep Mode

The information described above deals with clock gating only. In this example, during the time that the clocks are powered down, VDD remains at normal power levels. To obtain the maximum power savings during sleep mode, external logic can reduce the core VDD voltage once the P6600 core has asserted SI_SLEEP. This additional step can greatly reduce leakage and consequently power consumption during sleep mode. The minimum VDD voltage that can be used, and still allow the P6600 core to retain state, is process dependent.

The reduction of VDD can only be controlled by external means. The P6600 core does not provide a mechanism to reduce VDD internally during sleep mode. Note that if this option is implemented, it will take longer to restart the processor since the VDD must be ramped up to appropriate level before asserting the wake interrupt.

Refer to Section 7.7 “P6600 Core Power Gating” for more information.

7.6.1.2 Restart Latency Trade-Offs

Once the decision is made to enter sleep mode, some number of clocks are required to place the P6600 core into sleep mode, and bring the core out of sleep mode. In most designs, once sleep mode is entered, the core must remain in sleep mode for at least 100 clock cycles. Otherwise, the trade-off in time and power savings becomes negligible.

7.6.2 Designs Not Implementing Top Level Clock Gating

If top level clock gating was not enabled during the core configuration process, instruction-controlled power management can still be used.

From an instruction standpoint, the WAIT instruction and SI_SLEEP signal can still be used to place the P6600 core into sleep mode. However, since top level clock gating is disabled, it is incumbent upon external logic to suspend the input clock to the processor. If the input clock is suspended, it is suspended to the entire P6600 core. As a result, the processor has no way to detect a ‘wake’ interrupt. Therefore, the assertion of SI_RESET is the only way to restart the P6600 core. Note that if this method is used, all data will be lost and the registers will revert back to their default values.

7.6.3 Designs Implementing Fine Grain Clock Gating

Fine grain clock gating allows the P6600 core to shut down the clocks to individual blocks of logic within the chip. When the ‘fine grain clock gating’ option is selected during build time, separate clock domains are assigned to the various register blocks within the P6600 core. In the P6600 core, there is one write enable that is used to write all registers at once. If fine grain clock gating is enabled, the clock can be enabled only to the register block that is being accessed. The write enable for the other blocks is still driven, but no clock is supplied to those blocks not being accessed.

The implementation of fine grain clock gating requires the logic required to implement multiple clock trees within the P6600 core. Therefore, it works best in ASIC implementations where any number of clock domains can be assigned. It is less useful in FPGA implementations where the number of clock trees may be limited.

7.7 P6600 Core Power Gating

In addition to clock gating, power gating can be used to gain additional power savings. The saving and restoring of processor state can be used when the power savings provided by clock gating alone are not enough. In clock gating, the state of the processor need not be saved externally because even though the clocks are suspended, the power is still applied to the P6600 core, allowing the processor state to be saved internally.

In power gating, some or all of the power to the P6600 core can be shut down. This causes all data within the corresponding power domain(s) to be lost once the voltage falls below the retention value as defined by the process vendor. As a result, careful consideration must be taken to save some or all of the processor states before the power is shut down. Some of the logic blocks that can be saved prior to suspending the processor are:

- Registers (GPR, CP0, CP1, and/or CP2)
- Caches (instruction and/or data)
- Translation Lookaside Buffer (TLB)
- Scratch Pad RAM (Instruction and/or Data)

There are two methods that can be used to implement a suspend/resume mechanism in a P6600 core. These concepts are described in the following subsections.

- Hardware Suspend/Resume
- Software Suspend/Resume

7.7.1 Hardware Suspend/Resume

The hardware suspend/resume mechanism in the P6600 core allows the state of the caches, scratch pad RAM, and TLB to be transferred to memory via hardware using the suspend/resume (BIST) sideband signals that are defined during chip configuration. This process of moving data to and from the P6600 core is much faster than a pure software implementation. This process is covered in more detail in the P6600 *Hardware User's Manual*.

7.7.2 Software Suspend/Resume

For systems that have not implemented any hardware suspend/resume mechanism as described in the previous section, a software mechanism can be used to save state and power down the P6600 core. This section describes the tasks that should be performed during the suspend and resume processes.

7.7.2.1 Overview of Suspend/Resume Process

The recommended way of implementing a system suspend/resume in software is having a function that will perform a seamless suspend/resume operation. This means that to the rest of the software it looks like the function was entered and exited like any normal function, while in reality this function self-terminates in the middle of its execution by turning off the power the core, then resumes from where it left off shortly after power is restored.

At a high level, the assembly language skeleton should look like this:

/* Entry point to suspend/resume function, including the function prologue. */

```
suspend_resume:
...
...

/* Here we start the suspend sequence */

suspend:
...
...
...



/* At the end of the suspend sequence we turn off power to the core. The suspend sequence should never reach the
power_is_off label*/

power_is_off:

/* This is the starting point of the resume sequence. We will get here shortly after a warm reset.*/

resume:

...
...
...

/* At the end of the resume sequence we have the function epilogue, which includes a return to the calling function.*/

...
...
...

        jr      $31
        nop
```

As one can observe this function is clearly divided into two parts:

- The first part is the function entry (prologue) and the suspend sequence all the way down to the power shutdown. The suspend sequence includes the state saving and other supporting actions which are described in more details in the other sections.
- The second part is the resume sequence followed by the function exit (epilogue) and return to caller. The resume sequence includes state restoring and other actions which are described in more details in other sections.

If we look at the sequence of events on a time line it will look like this:

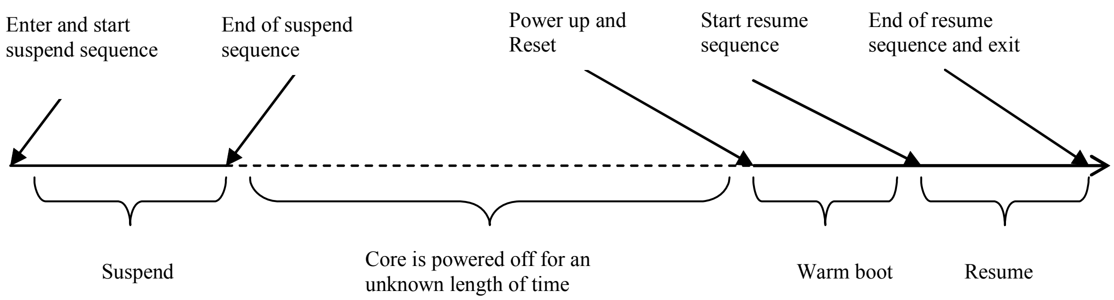


Figure 7.6 Suspend/Resume Sequence Time Line

7.7.3 Suspend Process

During a software suspend process, the following tasks are recommended. Each of these tasks is described in the following subsections.

- Save General Purpose Registers (GPR)
- Save some or all CP0 registers
- Flush the L1 data cache dirty lines and L2 cache dirty lines (if applicable)
- Save the return address
- Copy memory power down sequence into cache before switching memory to low-power mode (if applicable)
- Move memory to low-power mode (if applicable)
- Shut down power to the P6600 core

The GPR and CP0 registers are moved to the memory stack prior so that they can be easily retrieved when power is restored to the P6600 core. In this example, the registers would be moved to the stack and placed at the following memory offset addresses shown in Figure 7.7.

Offset	Memory Stack
0x74	Wired
0x70	Context
0x6C	Pagemask
0x68	Ebase
0x64	Config3
0x60	Config2
0x5C	Config1
0x58	Config0
0x54	Status
0x50	GPR31
0x4C	GPR30
0x48	GPR29
0x44	GPR28
0x40	GPR27
0x3C	GPR26
0x38	GPR23
0x34	GPR22
0x30	GPR21
0x2C	GPR20
0x28	GPR19
0x24	GPR18
0x20	GPR17
0x1C	GPR16
0x18	GPR7
0x14	GPR6
0x10	GPR5
0x0C	GPR4
0x08	GPR3
0x04	GPR2
0x00	GPR1

Figure 7.7 GPR and CP0 Register Locations in the Memory Stack

7.7.3.1 Save GPR Registers

MIPS recommends saving those GPR registers shown in the code example below. Note that the register numbers corresponding to the scratch registers are not saved. This includes GPR8 - GPR15, GPR24, and GPR25. For each GPR, a store word (*sw*) instruction is used to move the contents of the GPR register to memory.

```
sw      $1      0x00(sp)
sw      $2      0x04(sp)
sw      $3      0x08(sp)
```

```
sw      $4      0x0C(sp)
sw      $5      0x10(sp)
sw      $6      0x14(sp)
sw      $7      0x18(sp)
sw      $16     0x1C(sp)
sw      $17     0x20(sp)
sw      $18     0x24(sp)
sw      $19     0x28(sp)
sw      $20     0x2C(sp)
sw      $21     0x30(sp)
sw      $22     0x34(sp)
sw      $23     0x38(sp)
sw      $26     0x3C(sp)
sw      $27     0x40(sp)
sw      $28     0x44(sp)
sw      $29     0x48(sp)
sw      $30     0x4C(sp)
sw      $31     0x50(sp)
```

7.7.3.2 Save CP0 Registers

In the MIPS architecture the CP0 registers cannot be moved directly to memory. Therefore, they must first be moved to a GPR register. In this example the registers are moved to the k0 scratch pad register, then from the k0 register to memory at the location shown in the corresponding *sw* instruction. Note that the offset addresses for each *sw* instruction correspond to those shown in Figure 7.7.

As shown in the code snippet below, only a partial set of CP0 registers are saved. This is only an example. In some cases additional registers may need to be saved depending on the implementation.

```
mfco    k0,     CP0_STATUS      /*Move from coprocessor 0, CP0_STATUS to k0*/
sw      k0,     0x54(sp)        /*Store word k0 to offset 0x54 in memory*/
mfco    k0,     CP0_CONFIG0     /*Move from coprocessor 0, CP0_CONFIG0 to k0*/
sw      k0,     0x58(sp)        /*Store word k0 to offset 0x58 in memory*/
mfco    k0,     CP0_CONFIG1     /*Move from coprocessor 0, CP0_CONFIG1 to k0*/
sw      k0,     0x5C(sp)        /*Store word k0 to offset 0x5C in memory*/
mfco    k0,     CP0_CONFIG2     /*Move from coprocessor 0, CP0_CONFIG2 to k0*/
sw      k0,     0x60(sp)        /*Store word k0 to offset 0x60 in memory*/
mfco    k0,     CP0_CONFIG3     /*Move from coprocessor 0, CP0_CONFIG3 to k0*/
sw      k0,     0x64(sp)        /*Store word k0 to offset 0x64 in memory*/
mfco    k0,     CP0_EBASE       /*Move from coprocessor 0, CP0_EBASE to k0*/
sw      k0,     0x68(sp)        /*Store word k0 to offset 0x68 in memory*/
mfco    k0,     CP0_PAGEMASK    /*Move from coprocessor 0, CP0_PAGEMASK to k0*/
sw      k0,     0x6C(sp)        /*Store word k0 to offset 0x6C in memory*/
mfco    k0,     CP0_CONTEXT     /*Move from coprocessor 0, CP0_CONTEXT to k0*/
sw      k0,     0x70(sp)        /*Store word k0 to offset 0x70 in memory*/
mfco    k0,     CP0_WIRED       /*Move from coprocessor 0, CP0_WIRED to k0*/
sw      k0,     0x74(sp)        /*Store word k0 to offset 0x74 in memory*/
```

7.7.3.3 Flush Dirty Lines in L1 Data Cache

The following routine can be used to flush the dirty lines in a 32 Kbyte, 4-way set associative data cache with a 32-byte line size in preparation for shut-down. In this routine software examines each cache line and performs an invalidate on all non-dirty lines, and a writeback-invalidate on all dirty lines. A similar routine must be applied for L2 dirty lines in systems implementing a level 2 cache.

```
#define INDEX_BASE 0x80000000 // We use KSEG0 address as the base address for cache index access
#define WAY_SIZE 0x2000          // size of one way in a 4-way set associative 32K cache (8K)
#define WAYOFFSET 13             // offset of bits which determine the cache way to access
#define ASSOC   4                // associativity (4 ways)
#define LINE_SIZE 32             // size of each cache line
#define IDX_WB_INV_DC 0x01       // code of index write-back invalidate D-cache operation

/* This macro performs the same cache op on 32 consecutive lines. */

#define cache32_unroll32(base,op)               \

        __asm__ __volatile__(                              \
                ".set push                         \n"       \
                ".set noreorder                    \n"       \
                ".set mips3                        \n"       \
                "cache %1, 0x000(%0); cache %1, 0x020(%0)\n"  \
                "cache %1, 0x040(%0); cache %1, 0x060(%0)\n"  \

                "cache %1, 0x080(%0); cache %1, 0x0a0(%0)\n"  \
                "cache %1, 0x0c0(%0); cache %1, 0x0e0(%0)\n"  \
                "cache %1, 0x100(%0); cache %1, 0x120(%0)\n"  \
                "cache %1, 0x140(%0); cache %1, 0x160(%0)\n"  \|
                "cache %1, 0x180(%0); cache %1, 0x1a0(%0)\n"  \
                "cache %1, 0x1c0(%0); cache %1, 0x1e0(%0)\n"  \
                "cache %1, 0x200(%0); cache %1, 0x220(%0)\n"  \
                "cache %1, 0x240(%0); cache %1, 0x260(%0)\n"  \
                "cache %1, 0x280(%0); cache %1, 0x2a0(%0)\n"  \
                "cache %1, 0x2c0(%0); cache %1, 0x2e0(%0)\n"  \
                "cache %1, 0x300(%0); cache %1, 0x320(%0)\n"  \
                "cache %1, 0x340(%0); cache %1, 0x360(%0)\n"  \
                "cache %1, 0x380(%0); cache %1, 0x3a0(%0)\n"  \
                "cache %1, 0x3c0(%0); cache %1, 0x3e0(%0)\n"  \

                ".set pop                          \n"       \
                :                                            \
                : "r" (base),                                \
                  "i" (op));

/* This function scans a 4-way set associative 32K bytes data cache with 32-byte line size and performs an index
write-back invalidate cache operation on each of the cache lines.*/

static  void flush_32k_4way_32byteline_dcache(void)

{                                                                          \
        unsigned long start = INDEX_BASE;
        unsigned long end = start + WAY_SIZE;
        unsigned long ws_inc = 1UL << WAYOFFSET;
        unsigned long ws_end = ASSOC << WAYOFFSET;
        unsigned long ws, addr;
```

```
/* For every way (ws = the bits in the address which dertmine the cache way to access). */
for (ws = 0; ws < ws_end; ws += ws_inc)
        /* In each way go from start to end address. */
        for (addr = start; addr < end; addr += LINE_SIZE * 32)
                /* Each time we perform the cache op on 32 lines. The address is a
                  combination of the cache line offset in side the way (addr) and the way bits (ws).*/
                cache32_unroll32(addr|ws, IDX_WB_INV_DC);
```

7.7.3.4 Save the Resume Address

This routine takes the starting address of the resume sequence and saves it somewhere on the board, external to the P6600 core. Later, after power up and reset, the warm boot sequence retrieves that address and jumps to it. This initiates execution of the resume process.

7.7.3.5 Copy Memory Power Down Sequence Into Cache

This piece of code loads the remaining instructions of the suspend sequence into the instruction cache. This is done since the memory (e.g. DRAM) is about to be put in low power mode and thus become inaccessible to the core. It is important that all instruction fetches hit in the instruction cache because if they miss the core won't be able to fetch them from memory.

```
*/

        .set noreorder

/* load the start address and end address of the remaining instructions */

        la      $8, mem_to_low_power
        la      $9, post_suspend        /*after power is removed*/

/* Now fill the cache line by line starting from the start address and incrementing the address by a line size in each
iteration until we get beyond the en address.*/

fill_icache:

        cache   0x14, 0($8)
        addiu   $8, $8, 32
        bltu    $8, $9, fill_icache
        nop

mem_to_low_power:
```

7.7.3.6 Move Memory to Low Power Mode

```
/* Here we have a sequence of instructions that will move the memory to low power mode. These instructions used to
perform this function are SOC specific depending on the particular way the memory is implemented and addressed.*/

...
...
...

/* The following label comes after the end of the suspend sequence. We should never get here because we are sup-
posed to loose power earlier.*/
```

post_suspend:

7.7.3.7 Shut Down Power to the P6600 Core

Once all of the above tasks have been performed, power to the P6600 core can be suspended by reducing VDD to 0V. This task is performed by the SOC and is implementation-dependent.

7.7.4 Resume Process

During the software resume process, the following tasks are recommended. The tasks are handled in the opposite order in which they were executed during the suspend operation.

- System Wake-up
- Power-Up VDD to the P6600 core and Assert Power-On Reset
- Warm/Cold Boot Detection
- Exit memory low-power mode
- Initialize caches and TLB
- Jump to resume address
- Restore CP0 registers
- Restore GPR registers

7.7.4.1 System Wake-Up

In a typical system the power management (PM) module stays active after the system enters suspend mode. This component will consume very little power but will keep monitoring external signals that may trigger the system to resume normal operation. Once a trigger is detected, the PM block will wake up various system components, one of these being the P6600 core. Since power to the core was shut down earlier, the core must be powered up and brought to its Reset state.

7.7.4.2 Power-Up VDD to the P6600 Core and Assert Power-On Reset

Once the system logic detects a resume condition, the system power management block must raise the VDD levels of the P6600 core to their normal operating levels and allow the voltage to stabilize. Once the voltages are stabilized, assert the power-on reset pin to the P6600 core.

7.7.4.3 Warm/Cold Boot Detection

When a processor core goes to its reset state it starts executing instructions from its Reset vector address. We call the initial sequence of instructions "boot" and it typically starts executing off of "boot ROM" memory. At this point the system must distinguish between two boot modes: cold boot and warm boot.

- A cold boot is typically performed when the entire system is powered up and has to initialize all of its hardware components. In this scenario there is typically no (or little) memory of the system's state prior to boot (although some systems will save configuration information in non-volatile memory). After the initial boot the operating system has to go through its own complete boot sequence which takes a relatively long time.

- A warm boot is typically performed to resume a system that was previously suspended for power saving. In this case much of the system state prior to boot is available and can be restored (for example, it was saved into a memory component which did not loose power or otherwise in non-volatile memory). The warm boot sequence is typically short as users expect instant response (from a user point of view the system is available even when it was suspended for power saving). A warm boot does not require the operating system to perform its full boot sequence. For the most part the OS will continue from where it left off.

In the case of a warm boot, the boot software sequence starts from the same place (the Reset vector address) whether it is a cold boot or warm boot condition. However, shortly thereafter it detect its mode whether it is a cold or warm boot. If the system resumes from suspend mode, the boot software will detect this and decide to perform a warm boot. The indication that the system is coming back from suspend mode may be available in the PM block or in some piece of memory. This mechanism is implementation dependent.

Once a decision is made to perform a warm boot and not a cold boot, the warm boot sequence will perform a basic initialization and then jump to the resume address in the suspend/resume function. The resume address will be available in an implementation dependent location where it was saved by the suspendsequence. Then, as discussed earlier, the function will restore some system state and return to its caller as if nothing ever happened. The caller may have no indication that the system was suspended for a while.

Examples of basic core initialization that must be carried out regardless of the boot mode are caches and TLB initialization. Many users will opt not to save and restore their cache and/or TLB states. Note that the P6600 core caches and TLB wake-up in a random state and must be initialized before data can be written to them.

7.7.4.4 Exit Memory Low-Power Mode

This is an optional system-dependent function. If the external memory devices were placed in low-power mode during the suspend process, the memory must exit its low-power mode before the instructions stored to the stack during the suspend process can be fetched by the P6600 core.

7.7.4.5 Initialize Caches and TLB

The initialize caches and TLB routines are always performed when reset is asserted to the P6600 core. This is done to bring the caches to an initial state. This routine would be exactly the same as the one used in the boot example that accompanies the delivery of each P6600 core. Refer to the boot example associated with the P6600 core package.

7.7.4.6 Jump to Resume Address

At this point the boot process is done with general initialization process initiated by the assertion of reset and is ready to start the actual resume sequence. It retrieves the starting address of the resume sequence that was saved earlier (as part of the suspend sequence) and jumps to it, thereby initiating execution of the resume sequence.

7.7.4.7 Restore CP0 Registers

In the MIPS architecture the CP0 registers cannot be moved directly from memory. Therefore, they must first be moved to a GPR register. In this example the registers are moved to the k0 scratch pad register, then from the k0 register to memory at the location shown in the corresponding *lw* instruction. Note that the offset addresses for each *lw* instruction correspond to those shown in Figure 7.7.

```
lw      k0,     0x74(sp)            /*Load word k0 from offset 0x74 in memory*/
mtco    k0,     CP0_WIRED           /*Move to coprocessor 0, CP0_WIRED from k0*/
lw      k0,     0x70(sp)            /*Load word k0 from offset 0x70 in memory*/
mtco    k0,     CP0_CONTEXT         /*Move to coprocessor 0, CP0_CONTEXT from k0*/
lw      k0,     0x6C(sp)            /*Load word k0 from offset 0x6C in memory*/
mtco    k0,     CP0_PAGEMASK        /*Move to coprocessor 0, CP0_PAGEMASK from k0*/
```

```
lw      k0,     0x68(sp)        /*Load word k0 from offset 0x68 in memory*/
mtco    k0,     CP0_EBASE       /*Move to coprocessor 0, CP0_EBASE from k0*/
lw      k0,     0x64(sp)        /*Load word k0 from offset 0x64 in memory*/
mfco    k0,     CP0_CONFIG3     /*Move to coprocessor 0, CP0_CONFIG3 from k0*/
lw      k0,     0x60(sp)        /*Load word k0 from offset 0x60 in memory*/
mtco    k0,     CP0_CONFIG2     /*Move to coprocessor 0, CP0_CONFIG2 from k0*/
lw      k0,     0x5C(sp)        /*Load word k0 from offset 0x5C in memory*/
mtco    k0,     CP0_CONFIG1     /*Move to coprocessor 0, CP0_CONFIG1 from k0*/
lw      k0,     0x58(sp)        /*Load word k0 from offset 0x58 in memory*/
mtco    k0,     CP0_CONFIG0     /*Move to coprocessor 0, CP0_CONFIG0 from k0*/
lw      k0,     0x54(sp)        /*Load word k0 from offset 0x54 in memory*/
mtco    k0,     CP0_STATUS      /*Move to coprocessor 0, CP0_STATUS from k0*/
```

7.7.4.8 Restore GPR Registers

MIPS recommends loading those GPR registers shown in the code example below. Note that the register numbers corresponding to the scratch pad registers are not loaded. This includes GPR8 - GPR15, GPR24, and GPR25. For each GPR, a load word (*lw*) instruction is used to move the contents of the corresponding memory location into the GPR.

```
lw      $31     0x50(sp)|
lw      $30     0x4C(sp)
lw      $29     0x48(sp)
lw      $28     0x44(sp)
lw      $27     0x40(sp)
lw      $26     0x3C(sp)
lw      $23     0x38(sp)
lw      $22     0x34(sp)
lw      $21     0x30(sp)
lw      $20     0x2C(sp)
lw      $19     0x28(sp)
lw      $18     0x24(sp)
lw      $17     0x20(sp)
lw      $16     0x1C(sp)
lw      $7      0x18(sp)
lw      $6      0x14(sp)
lw      $5      0x10(sp)
lw      $4      0x0C(sp)
lw      $3      0x08(sp)
lw      $2      0x04(sp)
lw      $1      0x00(sp)
```

Chapter 8

Global Interrupt Controller

This chapter describes the optional Global Interrupt Controller (GIC) included in the P6600 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 GIC handles the distribution of interrupts between and among the CPU's in the cluster. The GIC has the ability to route interrupts to each core independently. The GIC processes incoming external interrupts and provides maximum flexibility in the type of level, polarity, and edge-triggering mechanism. For example, each individual interrupt can be level-triggered (high or low), single edge triggered (rising or falling edge), or dual edge triggered. The GIC routes the interrupt to the appropriate core and associated interrupt pin in the manner that the core expects based on the programming of the GIC registers.

The P6600 Multiprocessing System incorporates Virtualization into the interrupt control system, allowing separate interrupt controllers for guest and root processes. Refer to the chapter in Virtualization in this manual for more information. In the P6600 MPS, the GIC is responsible for routing the interrupt sources to either the root or guest interrupt interface. These changes are only applicable for the External Interrupt Controller (EIC) mode of the GIC. In non-EIC mode, the GIC operates as before by routing all interrupts on to a single interrupt interface for processing inside the GIC. Note that shadow register sets are not present in the P6600 core.

The chapter contains the following sections:

8.1 General 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.
- Steers any interrupt source to any core interrupt input (Interrupt pin, NMI).

- Allows any core to interrupt any other core.
- 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 cores in the system.
- Able to integrate interrupt messages from peripherals such as PCI-Express.
- Hardware assist features are configurable be software at run-time.
- Provides interval and watchdog timers.

8.2 GIC Address Map Overview

The P6600 Multiprocessing System can contain up to six cores. To avoid the large address space needed for core-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 core number of the requester is supplied for each load/store command. The core number is used as an index to reference the appropriate subset of the instantiated control registers. By using the core number information, the hardware writes/reads the correct subset of the control registers pertaining to that core. Software does not need to explicitly calculate the register index for the core in question; it is done entirely by hardware.

In the P6600 Multiprocessing System, any core can access the registers of any other core by using the *Core-Other* address spaces. Software must write the *Core-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" core (as specified by the core number information).
- A second window for an "Other" core that allows a core to access the register set belonging to another core. The "Other" core is specified by first writing the *Core-Other Addressing Register* in the "local" core address space.

An additional section called 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 address map of the GIC is shown in Table 8.1.

Table 8.1 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
Core-Local Section Offset	0x08000	Offset relative to *GCR_GIC_Base* + using core number as Index	16 KB	Kernel
Core-Other Section Offset	0x0C000	Offset relative to *GCR_GIC_Base* + using *Core-Other Addressing Register* as Index	16 KB	Kernel
User-Mode Visible Section Offset	0x10000	Offset relative to *GCR_GIC_Base*	64 KB	User

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

- A *Shared* section in which the external interrupt sources are registered, masked, and assigned to a particular core and interrupt pin. This section is used by all cores in the system.
- A *Core-Local* section in which interrupts local to a core are registered, masked, and assigned to a particular interrupt pin. If External Interrupt Controller Mode (EIC) mode is used for a particular core, the EIC encoder is instantiated here.
- A *Core-Other* section in which the local core can access the Core-Local section of another core by which the interrupt can be registered, masked, and assigned to a particular interrupt pin of the other core. One core can setup the GIC for all cores in the system using this section.
- A *User Mode Visible* section that 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*, *Core-Local*, and *Core-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. Refer to Section 8.7 "GIC User-Mode Visible Section" for more information.

8.2.1 GIC Base Address

The GIC base address is a 17-bit value that is programmed into the GCR_CPC_BASE field of the *GCR CPC Base* register located at offset address 0x0088 in the Global Control Block of the CM2 registers. Refer to the *GCR_CPC_BASE Register in* Chapter 8, *CM2 Global Control Registers* for more information on this register.

8.2.2 Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in Table 8.1 above are relative to a GIC base address described above and can be located anywhere in physical memory. To determine the physical address of each block listed in Table 8.2, the base address written to the *GCR_GIC_BASE Register* this value would be added to the GIC block offset ranges to derive the absolute physical address as shown in Table 8.2. Note that an example base address of 0x1BDC_0 is used for these calculations.

Table 8.2 Example Physical Address Calculation of the GIC Register Blocks

Example Base Address PA[39:15]		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x00_1BDC_0	+	0x0000 - 0x7FFF	=	0x00_1BDC_ 0000 - 0x00_1BDC_7FFF	32 KB	GIC Shared Control Block
0x00_1BDC_0	+	0x8000 - 0xBFFF	=	0x00_1BDC_ 8000 - 0x00_1BDC_BFFF	16 KB	GIC Core-Local Control Block
0x00_1BDC_0	+	0xC000 - 0xFFFF	=	0x00_1BDC_ C000 - 0x00_1BDC_FFFF	16 KB	GIC Core-Other Control Block
0x00_1BDC_0	+	0x10000 - 0x1FFFF	=	0x00_1BDD_ 0000 - 0x00_1BDD_FFFF	64 KB	User-Mode Visible Block

8.2.3 Register Offsets Relative to the Block Offsets

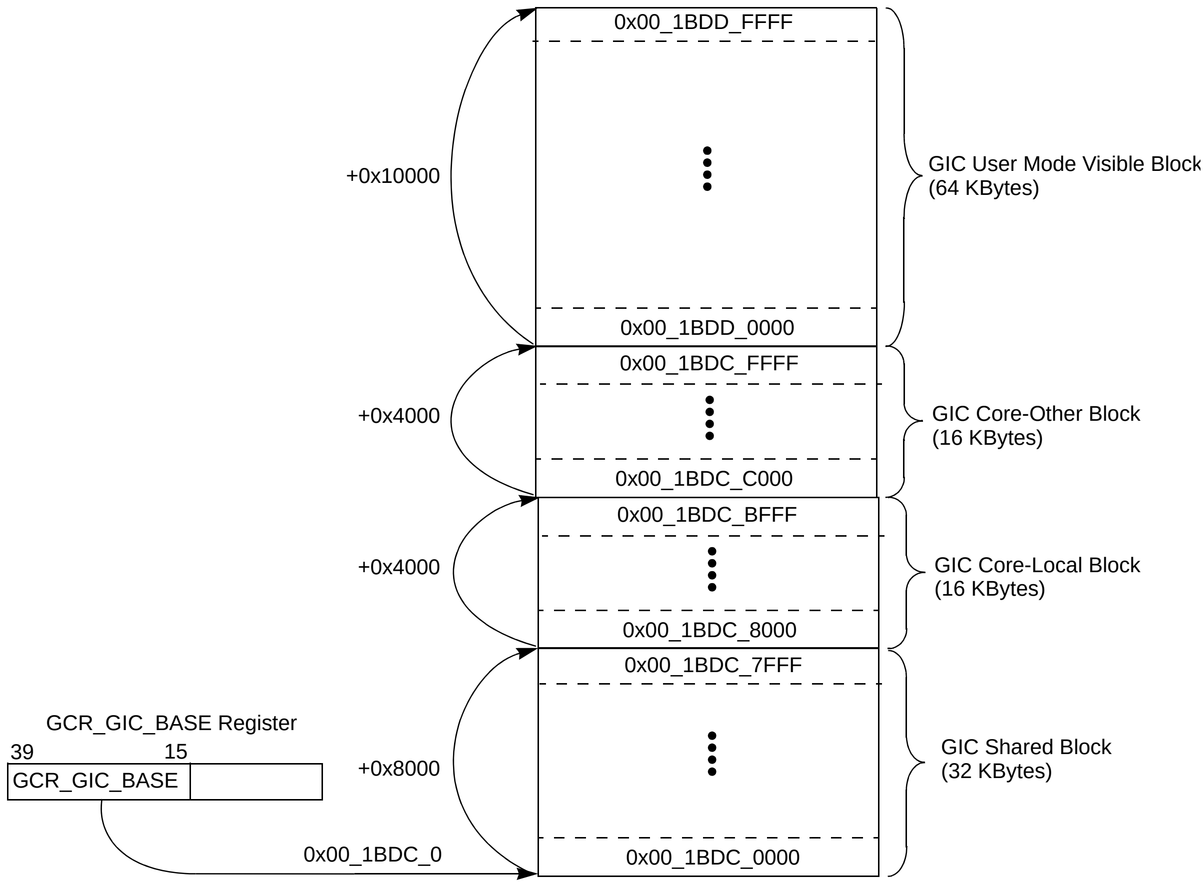
In addition to the block offsets, the register offsets provided in each register description of this chapter are relative to the block offsets shown in Table 8.1 above. To determine the physical address of each register, the base address programmed into the *GCR_GIC_BASE* register is added to the corresponding GIC block offset described above, plus the actual register offset to derive the absolute physical address as shown in Table 8.3.This table shows the physical address for the first few registers of the GIC Shared block. In this table an example base address of 0x00_1BDC_0 is used.

Table 8.3 Absolute Address of Individual GIC Shared Block Registers

MIPS Default Base PA[39:15]		Global Register Block Offset		Global Register Offset		Absolute Physical Address (40-bit)	Global Control Register
0x00_1BDC_0	+	0x0000	+	0x0000	=	0x00_1BDC_0000	GIC Config
0x00_1BDC_0	+	0x0000	+	0x0010	=	0x00_1BDC_0010	GIC CounterLo
0x00_1BDC_0	+	0x0000	+	0x0014	=	0x00_1BDC_0014	GIC CounterHi
0x00_1BDC_0	+	0x0000	+	0x0020	=	0x00_1BDC_0020	GIC Revision
0x00_1BDC_0	+	0x0000	+	0x0100	=	0x00_1BDC_0100	CPC Interrupt Polarity 0
...	+	...	+	...	=	...	...

This concept is described in Figure 8.1 below. In this figure an example base address of 0x00_1BDE_0 is used.

Figure 8.1 GIC Register Addressing Scheme Using an Example Base Address of 0x00_1BDC_0



8.3 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

8.3.1 Setting the GIC Base Address and Enabling the GIC

As described in Section 8.2.1 "GIC Base Address", 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 executing outside of the boot process.

To enable the GIC the GIC_EN bit must be set in this same register.

8.3.2 Enabling Virtualization Mode

The P6600 GIC provides Virtualization support as indicated by a logic 1 in the GIC_CONFIG_{VZP} bit. The GIC can be programmed by software to operate in either virtualized (GIC_CONFIG_{VZE} = 1) or non-virtualized (GIC_CONFIG_{VZE} = 0) modes.

In the GIC non-virtualized mode, the following rules apply:

- Any registers, or any fields in the Shared and Core-Local sections that have been added for virtualization should be considered reserved and read-only.
- Any Core-Local state is maintained in the fully populated root context.
- The GIC interface to guest context in core (Guest Interrupt Bus) is always inactive (always 0) in either EIC or non-EIC modes.
- If the core is enabled for virtualization, all guest accesses must be ignored (loads return 0s, stores are dropped).

Refer to Section 8.4, "Virtualization Support" for more information on virtualization.

8.3.3 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 would control 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. Refer to Section 8.5.3.1 "Global Config Register (GIC_SH_CONFIG — Offset 0x0000)" for more information.

Each of the interrupt sources can be of either positive (asserted high) or negative (asserted low) polarity. Similarly, any of these sources can be either level-sensitive, single-edge-sensitive, or dual-edge-sensitive. Through the polarity control registers (*GIC_SH_POLx_y*), the trigger type control registers (*GIC_SH_TRIGx_y*) and dual edge control registers (*GIC_SH_DUALx_y*), all of the sources are normalized to positive, level-sensitive signals. This is the interrupt type supported by the CPU interrupt inputs.

For single-edged signaling, the *Polarity* register denotes which edge is used for setting the interrupt register and which edge is ignored. For double-edged signaling, both the rising and falling edges are used to set the interrupt register. These three registers work in conjunction with one another to define the characteristics of each specific interrupt in the system. Each bit of each register corresponds to an interrupt. So for a given bit, the corresponding interrupt characteristics would be defined as shown in Table 8.4. The 'n' in the table entries denotes that it can be any bit of a given register, but must be the same bit of each register.

Table 8.4 Selecting Interrupt Polarity, Edge Sensitivity, and Triggering

Polarity (GIC_SH_POL[n])	Trigger (GIC_SH_TRIG[n])	Single/Dual Edge (GIC_SH_DUAL[n])	Description
0	0	x	Interrupt is level sensitive and active low. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
1	0	x	Interrupt is level sensitive and active high. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
0	1	0	Interrupt is single edge triggered on the falling edge of the signal.
1	1	0	Interrupt is single edge triggered on the rising edge of the signal.
x	1	1	Interrupt is dual edge triggered. In this case the contents of the GIC_SH_POL have no meaning because interrupts occur on both the rising and falling edges of the signal.

8.3.3.1 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. Refer to Section 8.5.3.8 "Global Interrupt Trigger Type Registers (GIC_SH_TRIGx_y — See Table 8.26 for Mapping)", for more information on how to assign this parameter.

8.3.3.2 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.

Refer to Section 8.5.3.9 "Global Interrupt Dual Edge Registers (GIC_SH_DUALx_y — See Table 8.28 for Mapping)" for more information on how to assign this parameters.

8.3.3.3 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.

Refer to Section 8.5.3.7 "Global Interrupt Polarity Registers (GIC_SH_POLx_y — See Table 8.24 for Mapping)"for more information on how to assign this parameter.

8.3.4 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_CORE — maps the interrupt to a processor.
- Global Interrupt Map to Pin Register, GIC_SH_MAP_PIN — maps interrupt to a specific signal on a processor.

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 Table 8.5.

Table 8.5 Mapping of External Interrupts

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
0	0x2000	GIC_SH_MAP0_CORE31:0	248	0x3F00	GIC_SH_MAP248_CORE31:0
	0x0500	GIC_SH_MAP0_PIN		0x08E0	GIC_SH_MAP248_PIN
1	0x2020	GIC_SH_MAP1_CORE31:0	249	0x3F20	GIC_SH_MAP249_CORE31:0
	0x0504	GIC_SH_MAP1_PIN		0x08E4	GIC_SH_MAP249_PIN
2	0x2040	GIC_SH_MAP2_CORE31:0	250	0x3F40	GIC_SH_MAP250_CORE31:0
	0x0508	GIC_SH_MAP2_PIN		0x08E8	GIC_SH_MAP250_PIN

Table 8.5 Mapping of External Interrupts *(continued)*

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
3	0x2060	GIC_SH_MAP3_CORE31:0	251	0x3F60	GIC_SH_MAP251_CORE31:0
	0x050C	GIC_SH_MAP3_PIN		0x08EC	GIC_SH_MAP251_PIN
4	0x2080	GIC_SH_MAP4_CORE31:0	252	0x3F80	GIC_SH_MAP252_CORE31:0
	0x0510	GIC_SH_MAP4_PIN		0x08F0	GIC_SH_MAP252_PIN
5	0x20A0	GIC_SH_MAP5_CORE31:0	253	0x3FA0	GIC_SH_MAP253_CORE31:0
	0x0514	GIC_SH_MAP5_PIN		0x08F4	GIC_SH_MAP253_PIN
6	0x20C0	GIC_SH_MAP6_CORE31:0	254	0x3FC0	GIC_SH_MAP254_CORE31:0
	0x0518	GIC_SH_MAP6_PIN		0x08F8	GIC_SH_MAP254_PIN
7	0x20E0	GIC_SH_MAP7_CORE31:0	255	0x3FE0	GIC_SH_MAP255_CORE31:0
	0x051C	GIC_SH_MAP7_PIN		0x08FC	GIC_SH_MAP255_PIN
8 - 247	0x2100 - 0x3EE0	GIC_SH_MAP8_CORE31:0 GIC_SH_MAP247_CORE31:0			
	0x0520 - 0x08DC	GIC_SH_MAP8_PIN - GIC_SH_MAP247_PIN			

8.3.4.1 Mapping an Interrupt Source to a Processor

There is one shared "Global Interrupt Map to Core Register", GIC_SH_MAP_CORE 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. Refer to Section 8.5.3.16 "Global Interrupt Map to Core Registers (GIC_SH_MAPn_CORE31:0) — See Table 8.5 for Mapping)" for more information.

8.3.4.2 Mapping and Interrupt Source to a Specific Processor 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 processor. There are two bits that control the type of signals that can be assigned to the interrupt source. Refer to Section 8.5.3.15 "Global Interrupt Map to Pin Registers (GIC_SH_MAPx_y)" for more information.

- If set, the MAP_TO_PIN bit maps 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 maps the external interrupt source to the NMI bit in the CP0 Status register. This in essence causes the processor to soft boot using the boot exception vector as the start of the interrupt routine.

8.3.5 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 4 shared registers groups for Enabling, Disabling and Polling of interrupts.

- Enabling an interrupt using the "GIC Set Mask Registers", GIC_SH_SMASK
- Disabling an interrupt using the "GIC Reset Mask Registers", GIC_SH_RMASK
- Determining the Enable/Disable state of an interrupt state using "GIC Mask Register", GIC_SH_MASK
- Polling the interrupt active state using the "GIC Pending Register", GIC_PEND_MASK

Like the trigger registers, each interrupt source is represented by one bit in each register group. Each register in a group is 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 Section 8.5.3.1 "Global Config Register (GIC_SH_CONFIG — Offset 0x0000)" for more information.

8.3.5.1 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. Refer to Section 8.5.3.12 "Global Interrupt Set Mask Registers (GIC_SH_SMASKx_y — See Table 8.33 for Mapping)" for more information.

8.3.5.2 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. Refer to Section 8.5.3.11 "Global Interrupt Reset Mask Registers (GIC_SH_RMASKx_y — See Table 8.31 for Mapping)" for more information.

8.3.5.3 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. Refer to Section 8.5.3.13 "Global Interrupt Mask Registers (GIC_SH_MASKx_y — See Table 8.35 for Mapping)" for more information.

8.3.5.4 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. Refer to Section 8.5.3.14 "Global Interrupt Pending Registers (GIC_SH_PENDx_y — See Table 8.37 for Mapping)" for more information.

8.3.5.5 Programming Example

Incoming interrupts are registered in the *Global Interrupt Pending* registers (*GIC_SH_PENDx_y*). This is the register that software needs to probe to discern the source of the interrupt. The *Global Interrupt Mask* registers (*GIC_SH_MASKx_y*) allow software to temporarily disable any particular interrupt source.

There are separate set (*GIC_SH_SMASKx_y*) and reset (*GIC_SH_RMASKx_y*) mask registers to set/clear individual interrupts to avoid any read-modify-write hazards within the system (multiple cores reading/writing the mask register simultaneously). This mechanism is shown in Figure 8.2 for interrupts 31:0. For interrupts 64:32, a different set of registers is used. Similar for interrupts 95:64, and so on through interrupts 255:224.

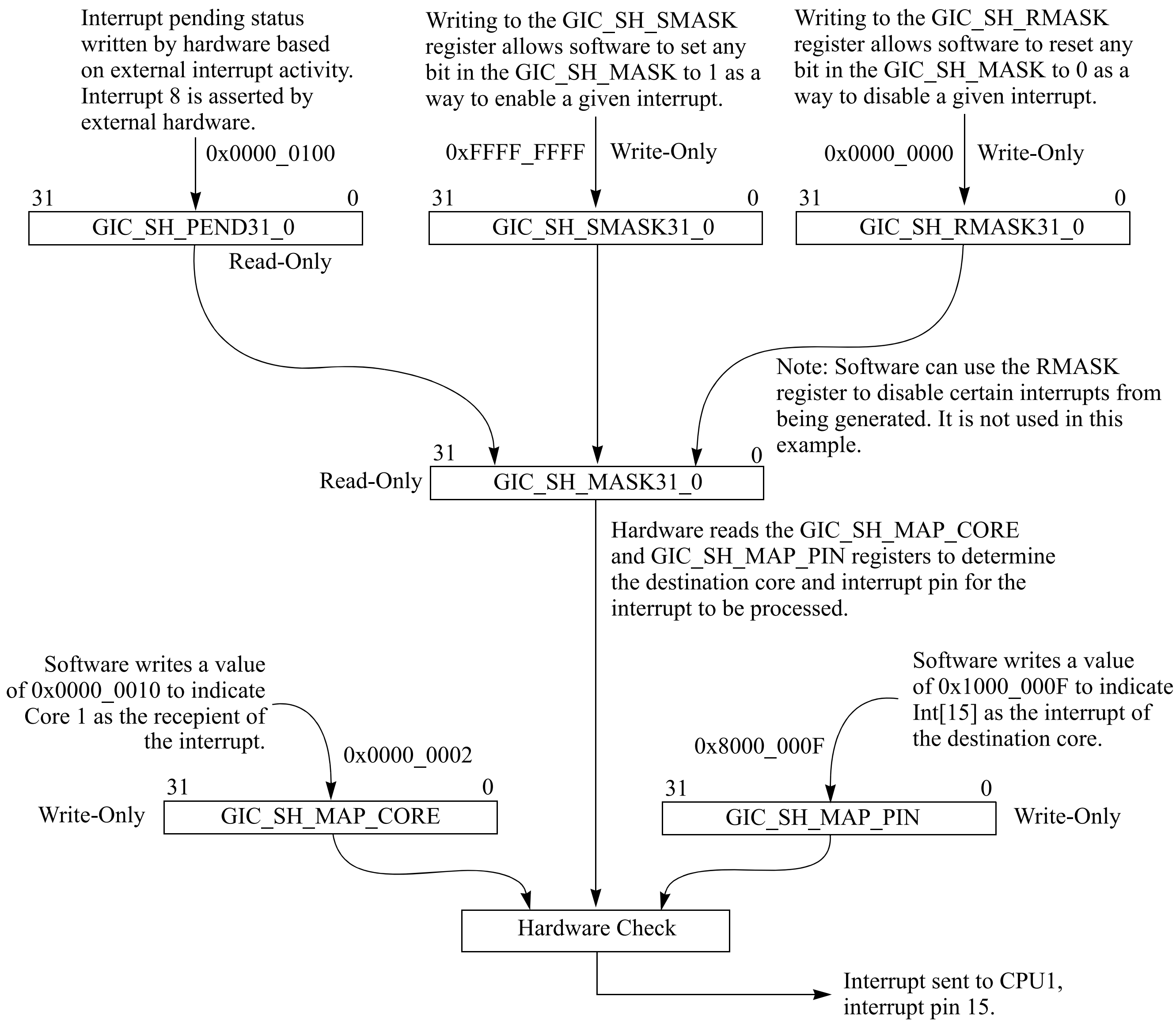
When an interrupt occurs, the corresponding bit in the *GIC_SH_PEND* register is set by hardware. If the corresponding interrupt enable bit in the *GIC_SH_MASK* bit is set, the GIC delivers the interrupt to the appropriate core. The hardware does this by using the *GIC_SH_MAP_CORE* register to send the interrupt to the appropriate core and the *GIC_SH_MAP_PIN* register to set the interrupt pins for that core.

In the following example:

- External interrupt 8 is asserted
- All bits of the *GIC_SH_SMASK* register are set, enabling all 32 interrupts.
- The receiving core is #1, and the receiving interrupt is #15.

This example is shown in Figure 8.2 below.

Figure 8.2 Masking and Mapping of Interrupts in the GIC



8.3.6 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.

8.3.6.1 WEDGE Register Programming Example

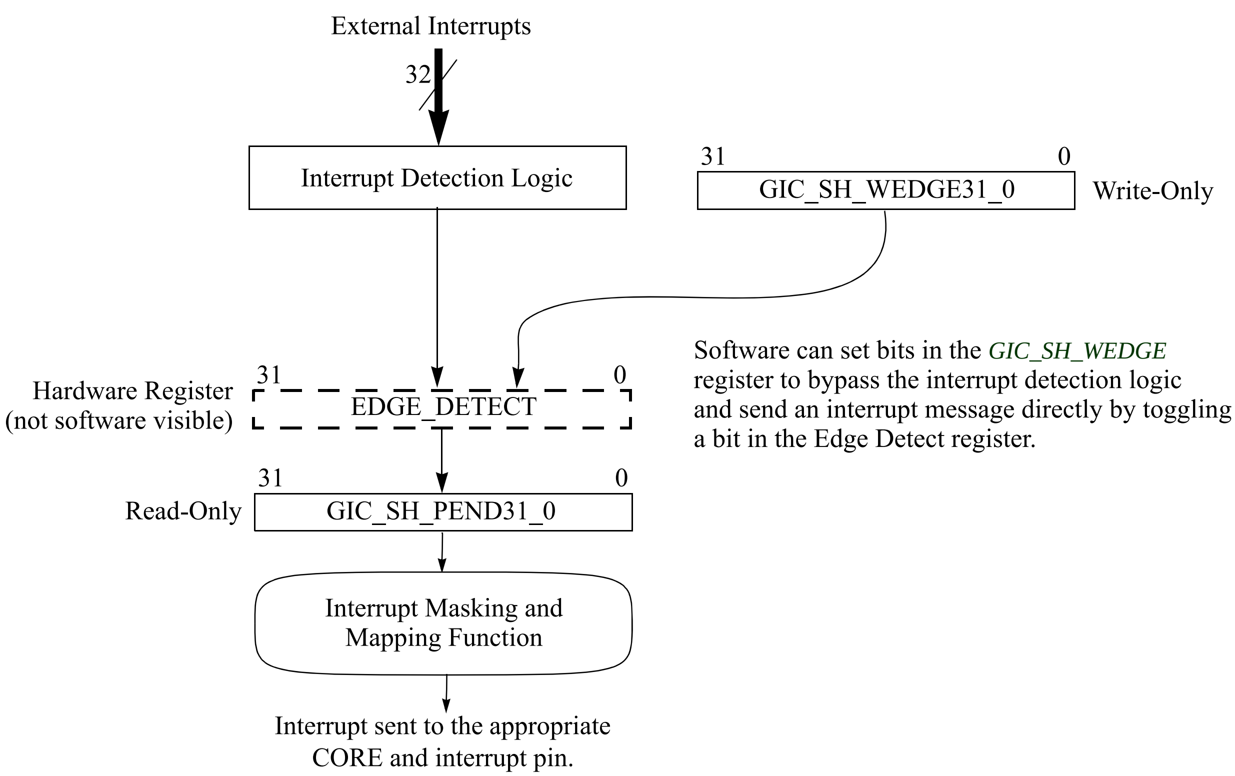
Setting a bit in the *Write Edge* register is treated equivalently to having the edge detection logic see an active edge. Because the programming of the Write Edge register has a direct effect on the state of the internal Edge Detect register, the *Write Edge* register can be used to bypass the edge detection logic. Thus, it does not matter whether the corresponding interrupt is configured to be rising, falling, or dual edge sensitive.

When core 0 wants to interrupt core 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 core (core1 in this example) using the GIC_SH_MAPi_CORE register).

For example, assume core 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 core 1, not core 0. The GIC routing logic then routes interrupt 40 onto the appropriate core 1 interrupt pins.

Figure 8.3 shows how the *Write Edge* register can be used to bypass the interrupt detection logic and assert interrupt directly. Setting a bit in the *Write Edge* register in turn sets the corresponding bit in the internal Edge Detect register, forcing an interrupt to be generated and allowing for inter-processor interrupts within the GIC.

Figure 8.3 Sending Inter-Processor Interrupts in the GIC



8.3.6.2 Inter-Processor Interrupt Code Example

Here is an example on how to set up interrupt sources 32 through 39 for inter-processor interrupts. First here is a table of what the #defines are set to.

Table 8.6 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	Base address of the Global Configuration Register
GCR_GIC_BASE	0x0080	Offset int the GCR of the GIC base Address
GIC_SH_MAP0_CORE31_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 CORE 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 Core Registers (GIC_SH_MAP0_CORE31_0).


li a0, 1           // set bit 0 processor 0

// Map Source 32 processor 0

sw a0,GIC_SH_MAP0_CORE31_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_CORE31_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_CORE31_0+(GIC_SH_MAP_SPACER * 34)(a1)
sll a0, a0, 1 // set bit 3 for processor 3 or for CORE3

// Source 35 to processor 3

sw a0,GIC_SH_MAP0_CORE31_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_CORE31_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_CORE31_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_CORE31_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_CORE31_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.

8.3.6.3 Example of Sending an Inter-Processor Interrupt

The following is a C coding example of sending an inter-processor interrupt. First the #defines:

#define	Value	Description
GIC_SH_WEDGE	*((volatile unsigned int*) (0xbbdc0280))	Address of the GIC_WEDGE_REGISTER.
FIRST_IPI	32	Source number for the first IPI.

void set_ipi(int cpu_num) {

// Add the enable bit, the first IPI number and the cpu number
// and write it to the GIC_SH_WEDGE register

GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num ;

8.3.6.4 Example of Clearing an Inter-Processor Interrupt

Once received, the interrupt routine should do whatever action is intended for the interrupt and clear the interrupt by writing the interrupt number to the GIC_SH_WEDGE register before executing the ERET instruction. NOTE: only the interrupt number is set before the write so the R/W bit is cleared, indicating that the interrupt is to be cleared.

```
li      k0, (GIC_SH_WEDGE | GIC_BASE_ADDR)
mfc0    k1, C0_EBASE              // Get cp0 EBase
ext     k1, k1, 0, 10       // Extract CPUNum
addiu   k1, 0x20          // Offset to base of IPI interrupts.
sw      k1, 0(k0)         // Clear this IPI.
```

8.3.7 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_CORE_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_CORE_WD_COUNT that can be used as liveliness signal for a processor.

8.3.7.1 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 MPS so all processors 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 (link to register reference of GIC_SH_CONFIG). 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 in Section 8.5.3.1, "Global Config Register (GIC_SH_CONFIG — Offset 0x0000)" 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 in Section 8.5.3.2, "GIC CounterLo (GIC_SH_CounterLo — Offset 0x0010)". Used in conjunction with the *GIC_SH_CounterHi* register. Sets the lower 32-bits of the starting count value.
- *GIC_SH_CounterHi* register in Section 8.5.3.3, "GIC CounterHi (GIC_SH_CounterHi — Offset 0x0014)". Used in conjunction with the *GIC_SH_CounterLo* register. Sets the upper 32-bits of the starting count value.

Compare Registers

The compare registers, GIC_COREi_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_COREi_CompareLo/Hi register. The compare registers are defined as follows:

- *GIC_COREi_CompareLo* register in Section 8.6.4.4, "Compare Low Register (GCI_COREi_ComparLo — Offset 0x00A0)". Used in conjunction with the *GIC_COREi_CompareHi* register to set the count value at which an internal interrupt is generated.
- *GIC_COREi_CompareHi* register in Section 8.6.4.5, "Core-Local CompareHi Register (GCI_COREi_ComparHi — Offset 0x00A4)". Used in conjunction with the *GIC_COREi_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:

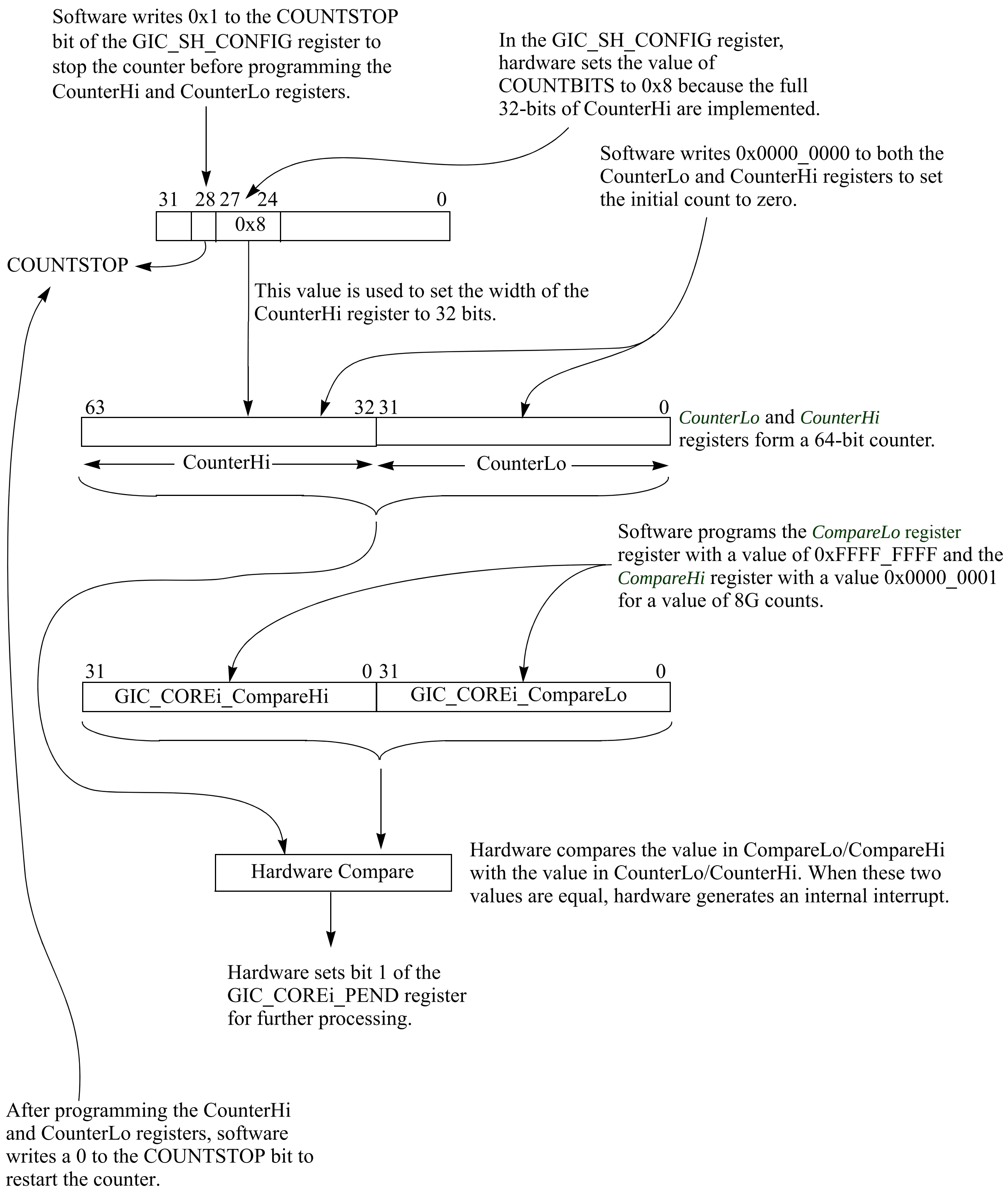
```
32 + 8 x 4 = 64 bits
```

In the GIC design, the COUNTBITS field is fixed at a value of 0x8, indicating a total counter size of 64-bits.

Counter Based Interrupt Example

In the example shown in Figure 8.4, the width of the counter is 64-bits, and the CompareLo/Hi value is 0x1_FFFF_FFFF which corresponds to 8G clock cycles. When this count is reached, hardware generates an internal interrupt.

Figure 8.4 Example of GIC Internal Counter-Based Interrupt Generation



8.3.7.2 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_COREi_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 stop counting when the processor is executing a wait instruction or is in a low power stats controlled by the Cluster Power Controller. Setting this bit to 1 causes 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.

Table 8.7 GIC Watchdog Timer Modes

Encoding	Mode	Behavior
0x2	One Trip	An interrupt is asserted and the timer stops.
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 MPS are reset. This mode provides a way to distinguish between a Software hang and a Hardware Hang. Usually the Watchdog Timer Interrupt is routed to NMI. This causes 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 is 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_CORE_WD_COUNT register with the value in the GIC_COREi_WD_INITIAL register. Refer to Section 8.6.4.1, "Watchdog Timer Config Register (GCI_COREi_WD_CONFIG0 — Offset 0x0090)" 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 Section 8.6.4.3, "Watchdog Timer Initial Count Register (GIC_COREi_WD_INITIAL — Offset 0x0098)" for more information.

Watchdog Timer Count Register

The "Watchdog Timer Count Register", GIC_CORE_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 Section 8.6.4.2, "Watchdog Timer Count Register (GIC_COREi_WD_COUNT — Offset 0x0094)" for more information.

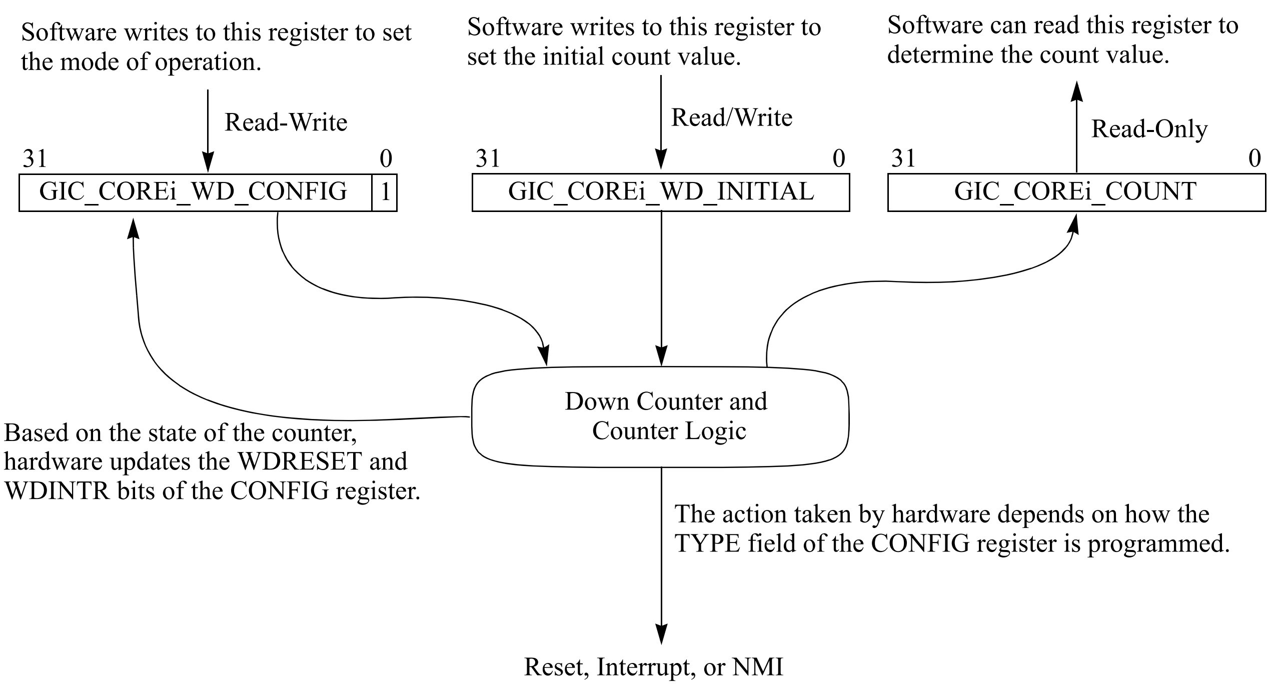
Configuring the Watchdog Timer

Software can configure the WatchDog timer with a starting count value by programming the *WatchDog Timer Initial Count* register (*GIC_COREi_WD_INITIAL*) located at offset address 0x0098. Refer to Section 8.6.4.3 "Watchdog Timer Initial Count Register (GIC_COREi_WD_INITIAL — Offset 0x0098)" for more information.

Software can read the state of the count at any time by reading the *WatchDog Timer Count* register (*GIC_COREi_WD_COUNT*) located at offset address 0x0094. Refer to Section 8.6.4.2 "Watchdog Timer Count Register (GIC_COREi_WD_COUNT — Offset 0x0094)" for more information.

Figure 8.5 shows the timer counter configuration process.

Figure 8.5 Local Watchdog Timer Interrupt Count Configuration



Watchdog Timer Masking and Mapping

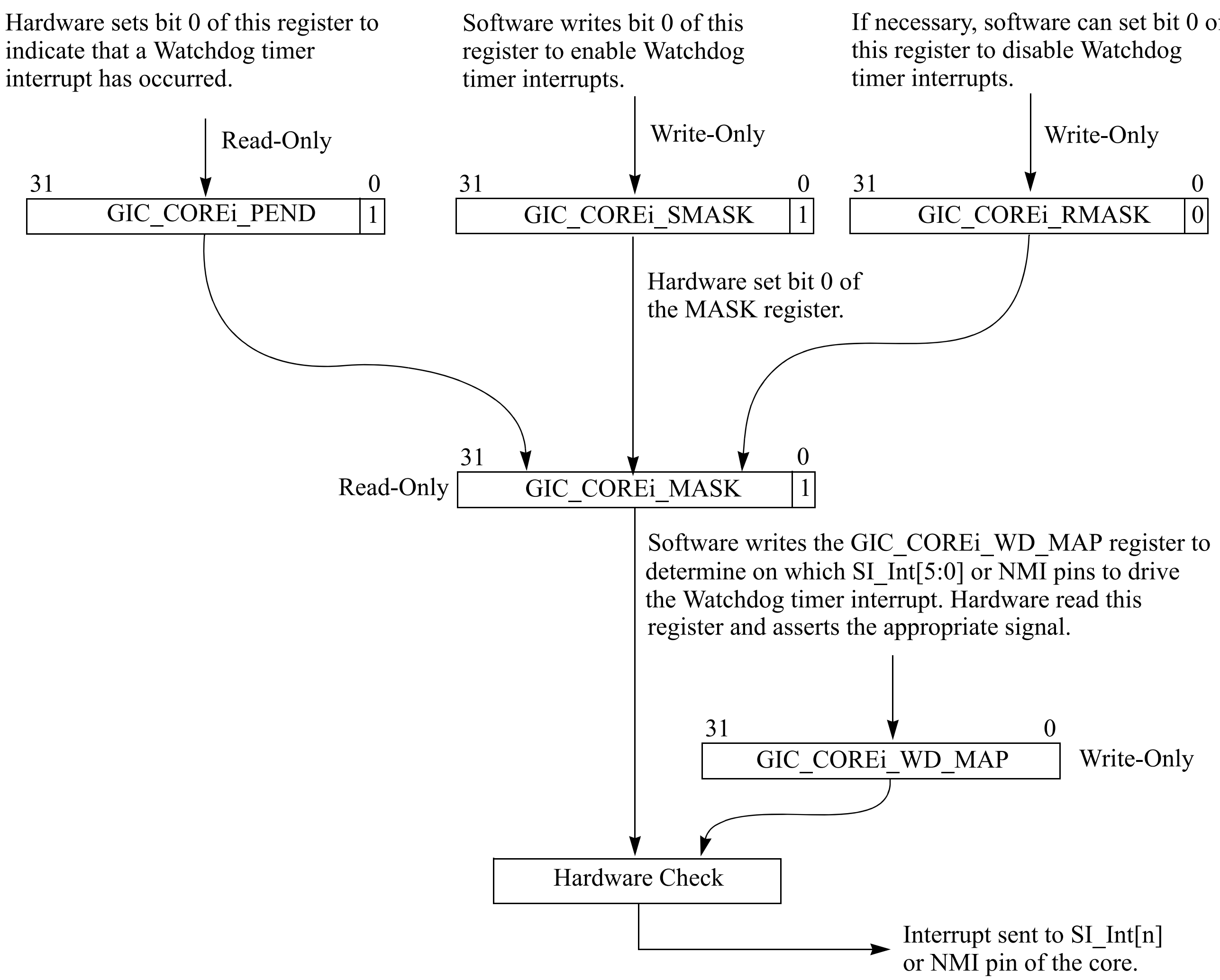
Figure 8.5 above shows the process used to configure the Watchdog timer. Once a Watchdog timer interrupt is generated (output of Figure 8.5), hardware sets bit 0 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 0 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the Watchdog timer interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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]*, or *NMI* pins the interrupt is driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. For example, if software programs this field with a value of 0x2, then the Watchdog timer interrupt is driven into *SI_Int[2]*. In non-EIC mode, only encodings 0 - 5 are valid.

In EIC mode, the core 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 33. This encoded value is then driven onto *SI_Int[5:0]*.

Figure 8.6 Watchdog Timer Interrupt Masking and Mapping in the GIC



Watchdog Timer and Debug Mode

Under certain conditions, software may want to suspend Watchdog timer operation while the P6600 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 P6600 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.

8.3.8 Local Interrupt Routing

8.3.8.1 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 core 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_COREi_CTL (link to register reference of GIC_COREi_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_COREi_CTL* register determines the routing of the following interrupts. In the P6600 GIC design, these bits are hard-wired to 1. Note that Software Interrupts from the core 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_COREi_CTL* register.

Table 8.8 GIC_COREi_CTL Register Fields

Bit Field Name	Behavior if cleared
FDC_ROUTABLE	The CPU Fast Debug Channel Interrupt is hard wired to one of the SI_Int pins as described by the CPU's COP0 IntCtlI.PFDCI register field.
SWINT_ROUTABLE	The CPU SW Interrupts are routed back to the CPU directly.
PERFCOUNT_ROUTABLE	The CPU Performance Counter Interrupt is hard wired to one of SI_Int pins as described by the CPU's COP0 IntCtl.IPPCI register field.
TIMER_ROUTABLE	The CPU Timer Interrupt is hard wired to one of the SI_Int pins, as described by the CPU's COP0 IntCtl.IPTI register field

8.3.8.2 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 (link to register reference of Local WatchDog Timer/Compare/CPU Timer/PerfCount/SWInt0-1 Map to Pin Registers) for each local interrupt source that further maps the local interrupt to a specific input on the processor. There are two bits, MAP_TO_PIN and MAP_TO_NMI that control the type of input that is assigned to the interrupt source. Only one of these bits can be set at any one time.

- If set, the MAP_TO_PIN bit maps 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 bit maps the local interrupt source to the NMI bit in the CP0 Status register. This in essence causes the processor to soft boot using the boot exception vector as the start of the interrupt routine.

Each of these interrupt types is described in the following subsections. Table 8.9 lists the registers and associated bits that would be programmed to facilitate each type of interrupt listed above.

Table 8.9 Local Interrupt Masking and Mapping Register Usage Per Interrupt Type

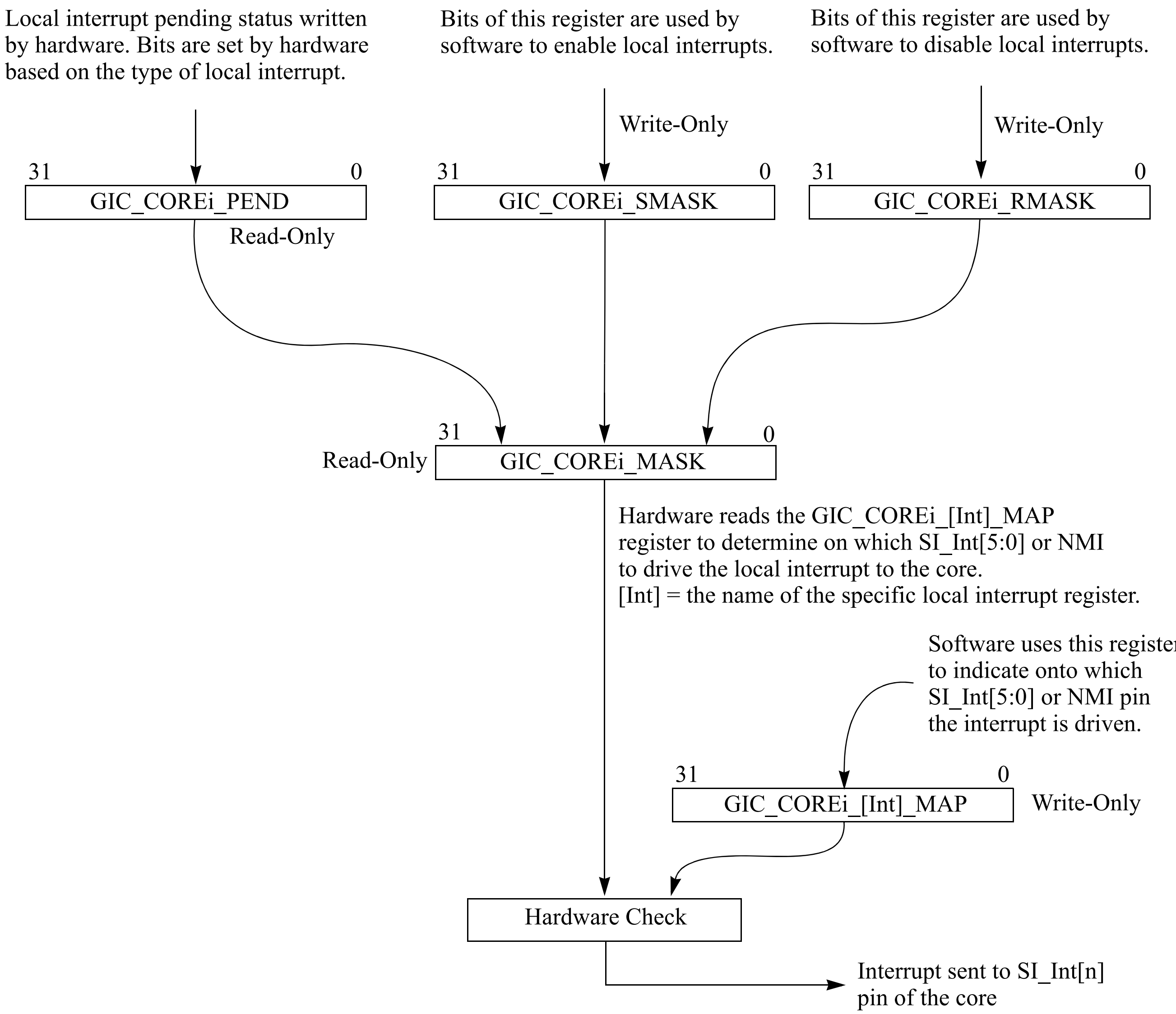
Interrupt	Register Name	Offset	Bits Used	Function
WatchDog	GIC_COREi_PEND	0x0004	0	Set by hardware on a local WatchDog timer interrupt.
	GIC_COREi_MASK	0x0008	0	Set by hardware based on the state of bit 0 of the SMASK and RMASK registers. Used to determine whether the interrupt is processed or ignored.
	GIC_COREi_RMASK	0x000C	0	Used by software to disable WatchDog timer interrupts.
	GIC_COREi_SMASK	0x0010	0	Used by software to enable WatchDog timer interrupts.
	GIC_COREi_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 P6600 core.
Count and Compare	GIC_COREi_PEND	0x0004	1	Set by hardware on a local Count/Compare interrupt.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	1	Used by software to disable Count/Compare interrupts.
	GIC_COREi_SMASK	0x0010	1	Used by software to enable Count/Compare interrupts.
	GIC_COREi_ 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 P6600 core.
Timer	GIC_COREi_PEND	0x0004	2	Set by hardware on a local timer interrupt.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	2	Used by software to disable timer interrupts.
	GIC_COREi_SMASK	0x0010	2	Used by software to enable timer interrupts.
	GIC_COREi_ 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 P6600 core.
Performance Counter	GIC_COREi_PEND	0x0004	3	Set by hardware on a performance counter interrupt.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	3	Used by software to disable performance counter interrupts.
	GIC_COREi_SMASK	0x0010	3	Used by software to enable performance counter interrupts.
	GIC_COREi_ 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 P6600 core.
Software Interrupt 0	GIC_COREi_PEND	0x0004	4	Set by hardware on a software interrupt 0 occurrence.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	4	Used by software to disable software interrupt 0 interrupts.
	GIC_COREi_SMASK	0x0010	4	Used by software to enable software interrupt 0 interrupts.
	GIC_COREi_ 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 P6600 core.

Table 8.9 Local Interrupt Masking and Mapping Register Usage Per Interrupt Type *(continued)*

Interrupt	Register Name	Offset	Bits Used	Function
Software Interrupt 1	GIC_COREi_PEND	0x0004	5	Set by hardware on a software interrupt 1 occurrence.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	5	Used by software to disable software interrupt 1 interrupts.
	GIC_COREi_SMASK	0x0010	5	Used by software to enable software interrupt 1 interrupts.
	GIC_COREi_ 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 P6600 core.
Fast Debug Channel	GIC_COREi_PEND	0x0004	6	Set by hardware on a Fast Debug Channel (FDC) interrupt.
	GIC_COREi_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 is processed or ignored.
	GIC_COREi_RMASK	0x000C	6	Used by software to disable FDC interrupts.
	GIC_COREi_SMASK	0x0010	6	Used by software to enable FDC interrupts.
	GIC_COREi_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 P6600core.

The general overview of the local interrupt pending, masking, and mapping process is shown in Figure 8.7.

Figure 8.7 Local Interrupt Masking and Mapping in the GIC



Each of the registers listed in Figure 8.7 above can be found in the following sections:

- Section 8.6.3.2 "Local Interrupt Pending Register (GIC_COREi_PEND — Offset 0x0004)"
- Section 8.6.3.3 "Local Interrupt Mask Register (GCI_COREi_MASK — Offset 0x0008)"
- Section 8.6.3.4 "Local Interrupt Reset Mask Register (GCI_COREi_RMASK — Offset 0x000C)"
- Section 8.6.3.5 "Local Interrupt Set Mask Register (GCI_COREi_SMASK — Offset 0x0010)"
- Section 8.6.3.6 "Local Map to Pin Registers (Offset 0x0040 - 0x0058 — See Table 8.48 for Mapping)"

8.3.8.3 Watchdog Timer Interrupts

For more information, refer to Section 8.3.7.2, "GIC Watchdog Timer".

8.3.8.4 Count and Compare Interrupts

A count and compare interrupt occurs when the contents of the of *GIC_COREi_CompareLo* and *GIC_COREi_CompareHi* registers match the contents of *GIC_SH_CounterLo* and *GIC_SH_CounterHi*, the Count/Compare interrupt is triggered.

When a count and compare interrupt is generated, hardware sets bit 1of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 1 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the count and compare interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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] or NMI* pins the interrupt is driven onto. In vectored interrupt mode, bits 5:0 of this register are used to select one of 6 core interrupts. In this mode, only encodings 0 - 5 are valid. In EIC mode, the core 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]*.

8.3.8.5 Timer Interrupts

When a timer interrupt is generated, hardware sets bit 2 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 2 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the timer interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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 determine 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] or NMI* pins the interrupt is driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts.

8.3.8.6 Performance Counter Interrupts

When a timer interrupt is generated, hardware sets bit 3 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 3 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the performance counter interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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 determine 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] or NMI* pins the interrupt is driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts.

8.3.8.7 Software Interrupts

Each core provides two software interrupts; 0 and 1. Software interrupts originate from the CPU and are only used by the GIC in EIC mode. In non-EIC mode they are routed internally within the CPU.

When software interrupt 0 is generated, hardware sets bit 4 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 4 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the software interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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 determine 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] or NMI* pins the interrupt is driven onto. In EIC mode, the core 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 is driven onto.

8.3.8.8 Fast Debug Channel Interrupts

When a Fast Debug Channel (FDC) interrupt is generated, hardware sets bit 6 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 6 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the fast debug channel interrupt has been masked. The *GIC_COREi_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_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_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 determine 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] or NMI* pins the interrupt is driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the P6600 core encodes this field to support up to 63 interrupts.

8.3.9 EIC Mode Setting

EIC mode is controlled through software by setting the EIC_MODE bit in the Local interrupt Control Register, GIC_COREi_CTL. Setting this bit enables EIC mode. This bit defaults to 0, vectored interrupt mode. Refer to Section 8.6.3.1 "Local Interrupt Control Register (GCI_COREi_CTL — Offset 0x0000)" for more information.

8.3.10 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_COREi_SMASK
- Disabling an interrupt using the "GIC Local Reset Mask Registers", GIC_COREi_RMASK
- Determining the Enable/Disable state of an interrupt state using "GIC Local Interrupt Mask Register", GIC_COREi_MASK
- Polling the interrupt active state using the "GIC Local Interrupt Pending Register", GIC_COREi_PEND

8.3.10.1 Enabling External Interrupts

The "GIC Local Set Mask Register", GIC_COREi_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. Refer to Section 8.6.3.5 "Local Interrupt Set Mask Register (GCI_COREi_SMASK — Offset 0x0010)" for more information.

Table 8.10 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

8.3.10.2 Disabling External Interrupts

The "GIC Local Reset Mask Register", GIC_COREi_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. Refer to Section 8.6.3.4 "Local Interrupt Reset Mask Register (GCI_COREi_RMASK — Offset 0x000C)" for more information.

Table 8.11 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

Table 8.11 Disabling External Interrupts

Field Name	Interrupt Controlled
WD_RESET_MASK	Watchdog

8.3.10.3 Determining the Enabled or Disabled Interrupt state

The "GIC Local Mask Register", GIC_COREi_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. Refer to Section 8.6.3.3 "Local Interrupt Mask Register (GCI_COREi_MASK — Offset 0x0008)" for more information

Table 8.12 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

8.3.10.4 Polling for an Active Interrupt

The "GIC Pending Register", GIC_COREi_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. Refer to Section 8.6.3.2 "Local Interrupt Pending Register (GIC_COREi_PEND — Offset 0x0004)" for more information

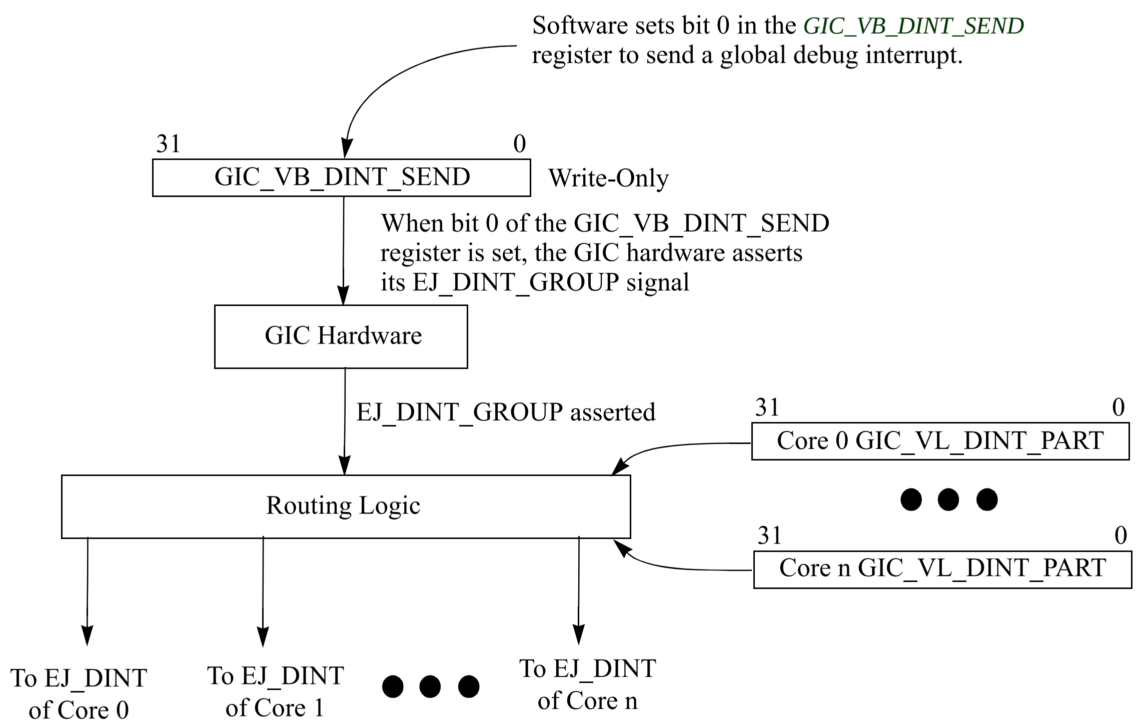
Table 8.13 Polling for an Active Interrupt

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

8.3.11 Debug Interrupt Generation

The GIC of the P6600 Multiprocessing System allows software to globally assert a debug interrupt to all cores in the system. When the *Send_DINT* bit of the *DINT Send to Group* register (GIC_VB_DINT_SEND) in Section 8.5.3.17, "DINT Send to Group Register (GIC_VB_DINT_SEND Offset 0x6000)" is set, the *EJ_DINT_GROUP* signal of the GIC is asserted. Based on thestate of this signal and the core-Local GIC_VL_DINT_PART registers, hardware asserts the EJ_DINT signal of each core in the system. This concept is shown in Figure 8.8.

Figure 8.8 Global EJTAG Debug Interrupt Generation in the GIC



8.4 Virtualization Support

As mentioned above, the P6600 MPS supports virtualization and the concept of guest and root modes. The following list shows some of the changes made to the GIC to support Virtualization.

The main changes to the GIC to support virtualization are summarized below and this functionality is only applicable to the EIC mode of the GIC.

- Incorporates logic required to route the guest external interrupts to the core. Each external interrupt source is assigned a GuestID for this purpose. The hypervisor is expected to program these fields prior to initializing interrupts in the system.
- A qualification mechanism has been added for the root and guest access of the GIC registers. This makes sure only the registers associated with the intended guest context is being accessed.
- Count-Compare (CC) timer interrupts are supported by root and guest contexts.
- WatchDog (WD) timer interrupts are supported by root or guest contexts, but never simultaneously.
- Additional interrupt interface added per-core to send the interrupts targeted to the guest context.
- Interrupts targeted to the root context are sent on the existing interrupt interface. This root interrupt interface contains a 4-bit bus that identifies the guest virtual machine to which the interrupt is targeted.
- An input port added for the GIC to provide the core's resident GuestID. This resident GuestID gets used GIC logic to route the target guest interrupts to the core and also to qualify the guest accesses to GIC registers in Core-Local section.
- New register fields added to control the GIC operating in virtualized or non-virtualized mode.
- Addition of duplicate registers and interface pins to support routing of guest context's local interrupts through the GIC. These are for guest context's count/compare, timer, performance counter and software interrupts.
- Add support for generating NMI interrupts from guest interrupts sources under the control of root.

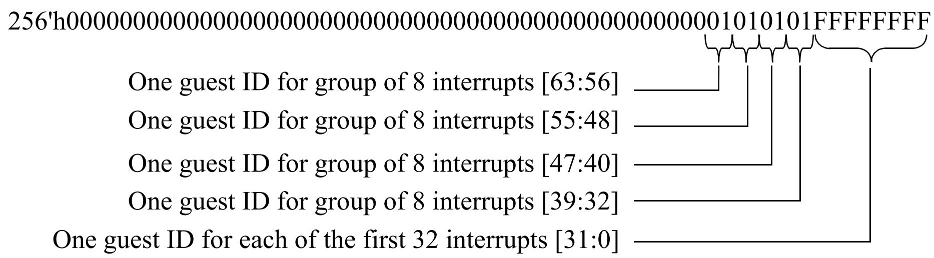
8.4.1 Routing of Guest External Source Interrupts

Each external interrupt source, or a logical group of external interrupt sources, is assigned a GuestID. This GuestID may be a maximum of 8-bits, but is set to 4 through a build time configuration parameter for initial set of cores. The per external interrupt source GuestID has been added as a new field to the shared section Global Interrupt Map to Pin registers.

The developer may choose to assign one GuestID to each external interrupt source. Alternatively, since the number of interrupt sources may be large (up to 256 interrupts), an implementation may choose to group external interrupt sources by GuestID, or provide an intermediate configuration such that some number of sources are each assigned a GuestID, while the remaining are grouped, and each group is assigned a GuestID. An example intermediate solution is one where the 1st 32 interrupt sources are individually assigned GuestIDs, while the remaining sources are divided up into groups of 8, each group with a GuestID.

To facilitate the configuration of GuestID grouping, a 256 bits wide vector is provided which needs to be set at build time as per the required GuestID grouping scheme. This vector is 256 bits wide, which is the maximum number of external interrupt sources supported by the P6600 GIC. However, only the relevant lower indexed bits takes effect when the GIC is configured for less than 256 external interrupts.

Each bit in this vector represents whether or not a physical GuestID register exists ('1' in the bit) or not ('0' in the bit) for that bits corresponding external interrupt source. In the case where a physical GuestID register does not exist for an external interrupt source, that external interrupt source uses the GuestID value from whatever the next lower indexed external interrupt source which has a physical GuestID register. For example, in a 64-interrupt system where the 1st 32 interrupt sources are individually assigned GuestIDs and the remaining sources are divided up into groups of 8, the 256-bit GuestID grouping vector would be configured with the value shown below:

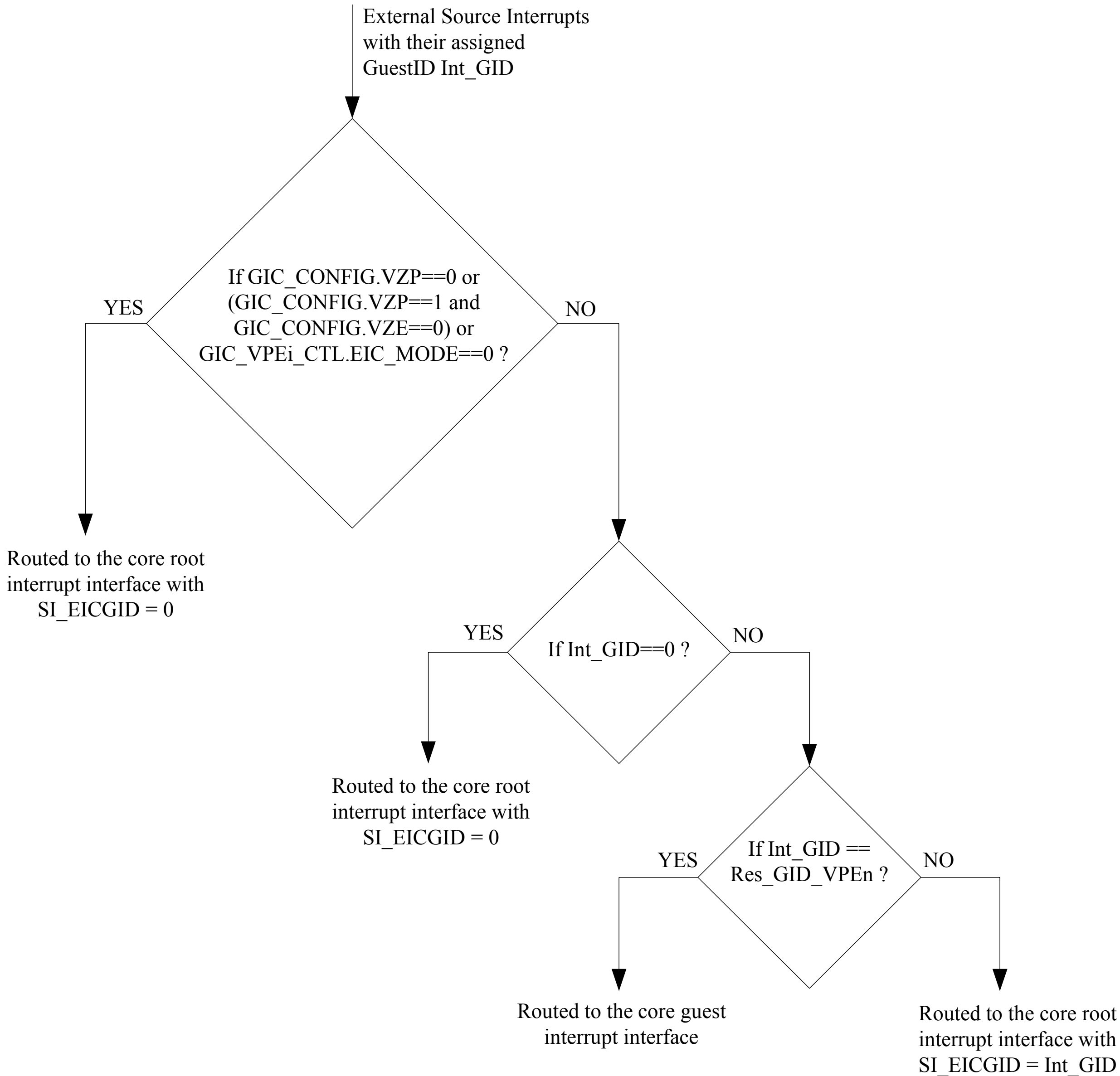


Software can determine the build time configured GuestID grouping scheme by reading this 256-bit GuestID grouping vector via the registers described in Section 8.5.3.6, "ID Group Configuration Registers (GIC_SH_GID_CONFIG, Offsets 0x0080 - 0x009C)".

By convention, a GuestID of 0 specifies root, while a non-zero GuestID specifies a guest. In addition, each Core-Local section in the GIC is aware of the GuestID resident in the physical core. These resident GuestIDs is brought into the GIC via the SI*_GID input ports and this is equal to the core cores *Guest*$_{ID}$ register field.

The routing of external source interrupts to either of cores root or guest interrupt busses is illustrated in Figure 8.9 below.

Figure 8.9 External Source Interrupt Routing to the Cores Root or Guest Interrupt Bus



8.4.2 Qualification of Root or Guest Software Access to GIC registers

In general, only the root software (hypervisor) requires access to the GIC configuration registers. Such configuration registers include, but not limited to, are for the specification of each interrupt's type (e.g., polarity, edge/level etc), Core assignment, interrupt routing etc. However, the guest software may require access to a subset of GIC registers for reading interrupt pending information, masking and clearing interrupts etc. Since a subset of GIC registers are shared by multiple guests and root, any guest-specific reads/writes must be qualified to avoid effecting the interrupts that are not associated with the intended guest.

The below listed shared section registers need to be directly accessed by guest. In the list below uses n_m nomenclature,

where n_m = 31+32xi_32xi, and i = 0 to 7.

- GIC_SH_WEDGE - to cause Inter-Core interrupts and clear EDGE registered external interrupts.
- GIC_SH_PENDn_m - to determine which external interrupts are pending.
- GIC_SH_MASKn_m - to determine which external interrupts are masked.
- GIC_SH_SMASKn_m - to set mask bits for external interrupts.
- GIC_SH_RMASKn_m - to clear mask bits for external interrupts.
- GIC_SH_TRIGn_m - to allow guest to set EDGE for causing IPI to other cores.
- GIC_SH_POLn_m - there is currently no identified reason for guest access to this register, but it is safe to do so.
- GIC_SH_DUALn_m - there is currently no identified reason for guest access to this register, but it is safe to do so.

Apart from the WEDGE register, all of the above listed registers contains one bit per external interrupt source. Guest access to each of these per external interrupt source bits are qualified with a per-external interrupt source valid vector. On guest writes to the WEDGE register, the encoded interrupt number value gets decoded out to drive the per-external interrupt source logic. Guest writes to the WEDGE register are qualified by gating this driving of per external interrupt source logic with the same per external interrupt source valid vector.

The guest context replicated Core-Local section registers may need to be directly accessed by guest software. Those registers are listed below.

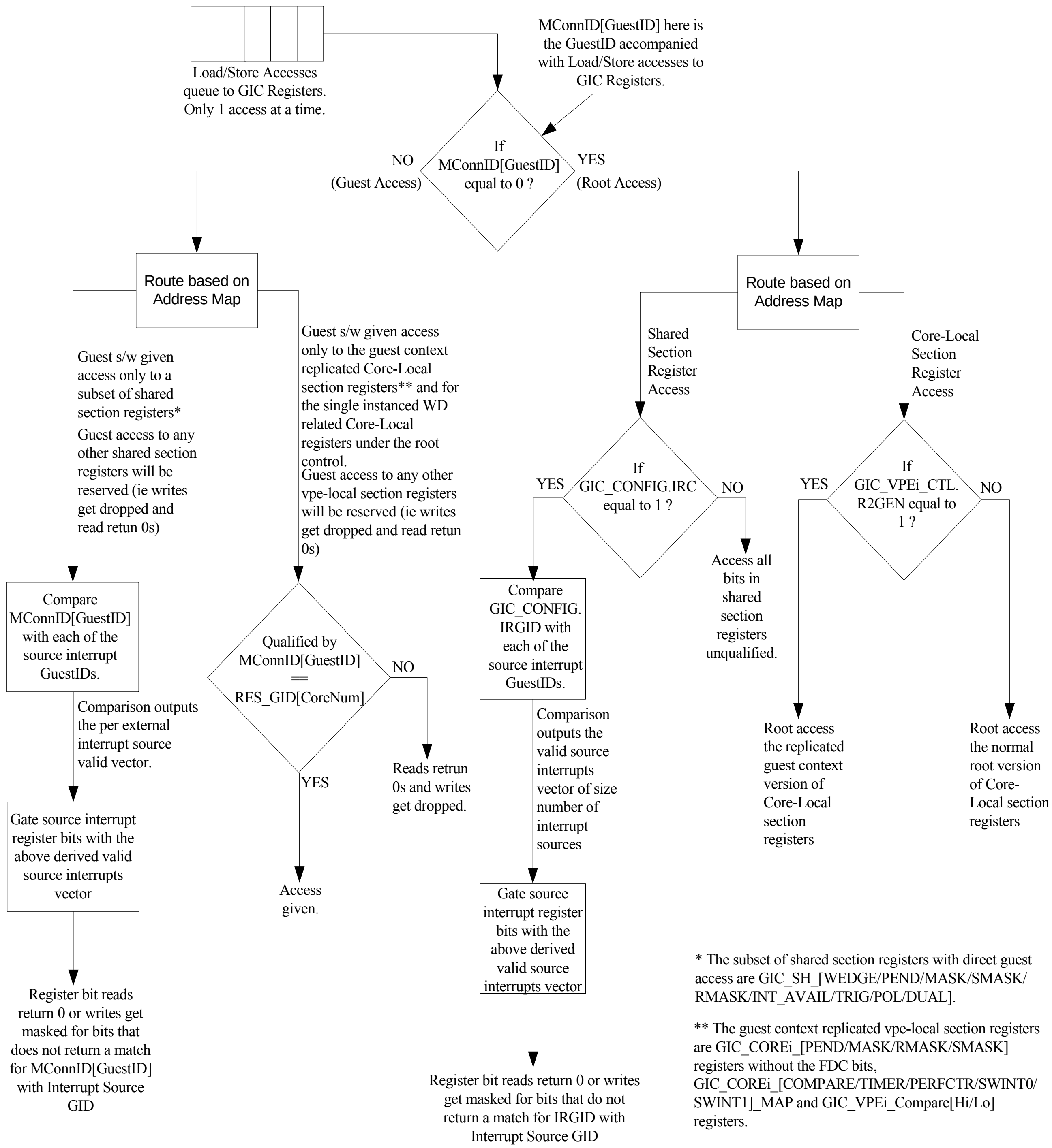
- GIC_COREi_PEND - for guest software to determine which local guest interrupts are pending.
- GIC_COREi_MASK - for guest software to determine which local guest interrupts are masked.
- GIC_COREi_SMASK - for guest software to set mask bits for local guest interrupts.
- GIC_COREi_RMASK - for guest software to clear mask bits for local guest interrupt.
- GIC_COREi_CompareLo/Hi - This allows the guest software to directly set its compare value after sampling its offsetted counter value.

where i = 0 to 5, the max number of configured cores.

8.4.3 Guest Accesses to Core-Local Registers

The guest accesses to the above listed core-local registers need to be qualified within the GIC to protect against unwanted guest accesses. This is done by comparing the guest load/store associated OCP MConnID[GuestID] with the target cores resident GuestID. The target CORE number for this is derived by using the MReqInfo[VPENum] port of OCP bus for these register accesses.

Figure 8.10 Root or Guest Access Flow into the GIC Registers



* The subset of shared section registers with direct guest access are GIC_SH_[WEDGE/PEND/MASK/SMASK/RMASK/INT_AVAIL/TRIG/POL/DUAL].

** The guest context replicated vpe-local section registers are GIC_COREi_[PEND/MASK/RMASK/SMASK] registers without the FDC bits, GIC_COREi_[COMPARE/TIMER/PERFCTR/SWINT0/SWINT1]_MAP and GIC_VPEi_Compare[Hi/Lo] registers.

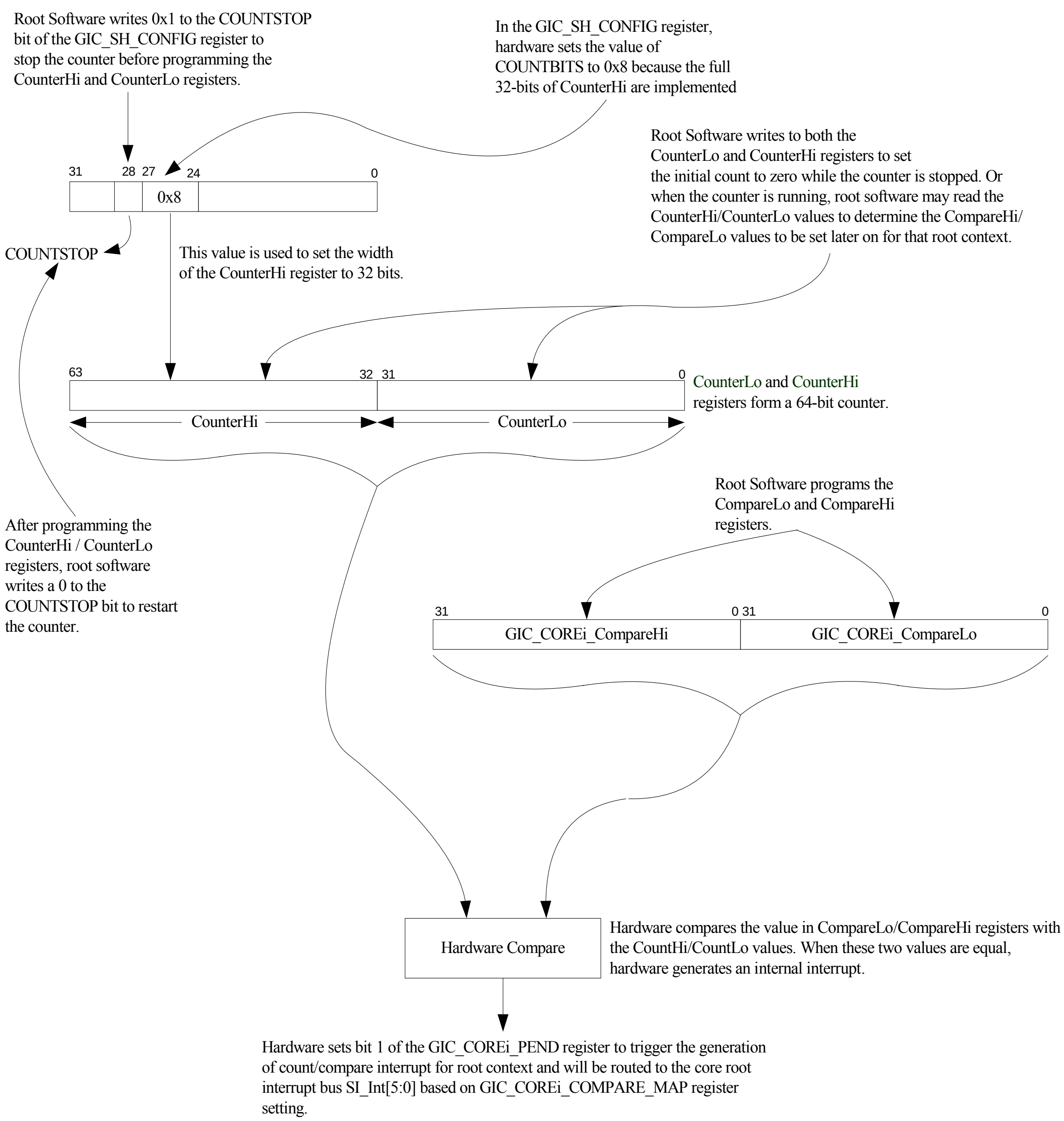
8.4.4 Count-Compare (CC) Timer Interrupts

The Count-Compare (CC) timer interrupts can be generated independently for both root and guest contexts. They are routed to their relevant root or guest interrupt bus of the core. The Root and Guest processing is described in the following subsections.

8.4.4.1 Root Mode Count-Compare Timer Interrupts

The root context use of the Count-Compare (CC) timer interrupts remain the same as in the existing GIC by using its existing relevant registers. This CC timer interrupt generation flow for root context is illustrated in Figure 8.11.

Figure 8.11 Root Context Count-Compare Timer Interrupt Generation Flow



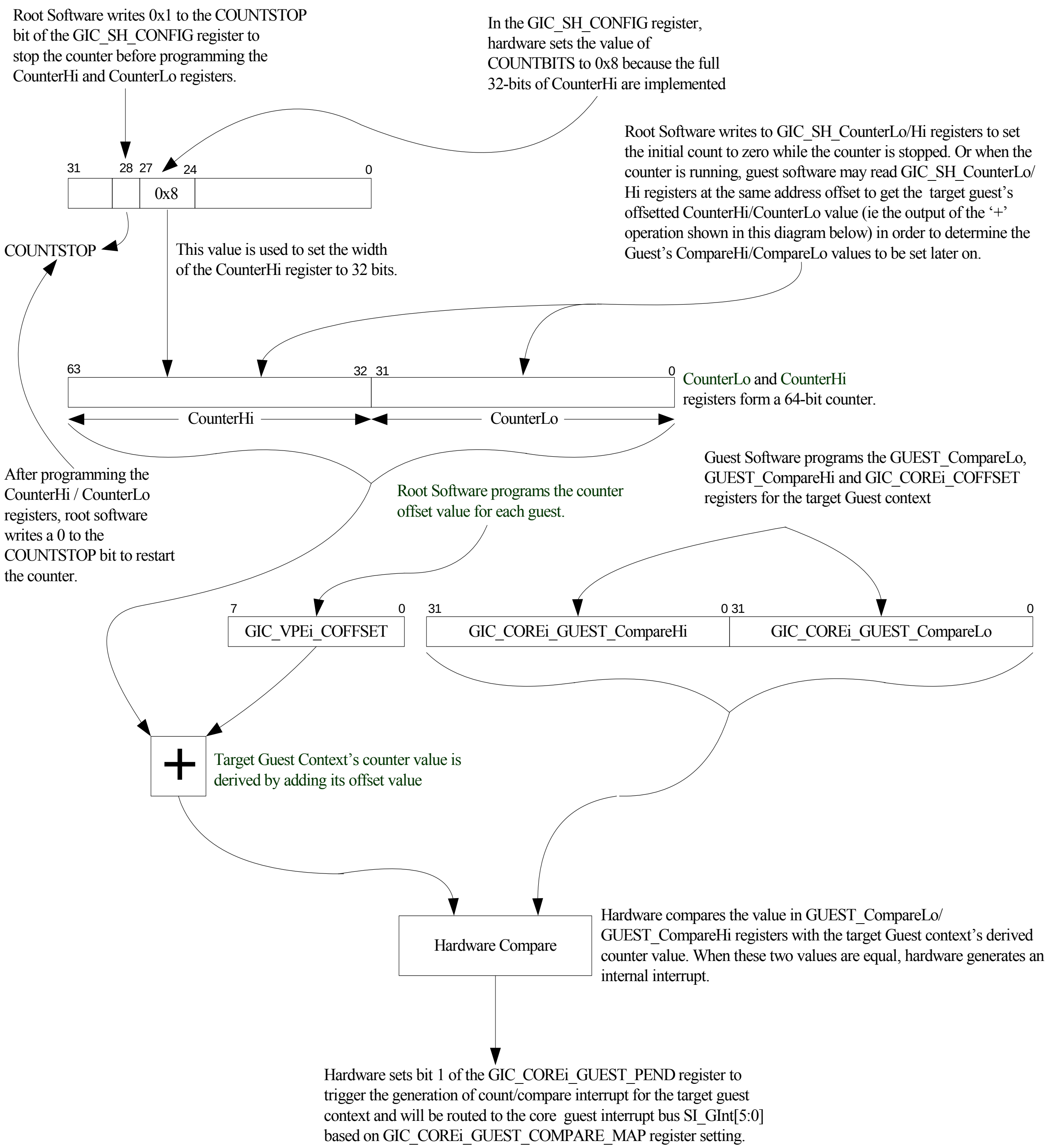
8.4.4.2 Guest Mode Count-Compare Timer Interrupts

For guest context use of the Count-Compare (CC) timer interrupts, the global counter value that is common to root and all guests cannot be used. Therefore, a counter which is offset by an n-bit (set to 8 by default) value is used for each guest context. To specify this guest counter offset value, a GIC_COREi_COFFSET register is added to each Core-Local section and the root is expected to program this offset value register. In addition, the compare value registers are replicated for the guest context and these are added as GIC_COREi_CompareLo/Hi registers to each Core-Local section. This allows guest and root contexts in each core to set compare independently.

To facilitate this guest context interrupt routing, the Count-Compare register bits are replicated for guest context registers GIC_COREi_[PEND/MASK/SMASK/RMASK] and also the GIC_COREi_COMPARE_MAP map-to-pin register replicated for guest context.

This CC timer interrupt generation flow for guest context is illustrated in Figure 8.12.

Figure 8.12 Guest Context Count-Compare Timer Interrupt Generation Flow



As illustrated in Figure 8.12, the guest software would need to sample the target guest's offset counter value in order to set the Compare Hi/Lo values for the target guest. The guest software reads the offset counter value via the GIC_SH_CounterLo/Hi registers using the same existing address offsets 0x0010, 0x0014 in the Shared register section. These guest reads would return the appropriate counter offset value where the offset is obtained from Core-Local's *GIC_COREi_COFFSET* register and the core is determined by the CoreNum value associated with the load access. Note the guest software is not allowed to write to *GIC_SH_CounterLo/Hi* registers and also cannot disable the counter by writing to the $GIC\_SH\_CONFIG_{COUNTSTOP}$ field.

8.4.5 Watchdog (WD) Timer Interrupts

In the GIC, a single WatchDog timer is present for the root context. The root may allow the guest to utilize this single WatchDog timer by setting the newly added control bit GEN in the *GIC_COREi_WD_CONFIG* register. In virtualized mode ($GIC\_SH\_CONFIG_{VZP}$ = 1 & $GIC\_SH\_CONFIG_{VZE}$ = 1) if the root software sets GEN = 1, then the guest software is allowed to access the WatchDog timer related registers *GIC_COREi_WD_[MAP/CONFIG/COUNT/INIIAL]*. However, in non-virtualised mode ($GIC\_SH\_CONFIG_{VZP}$ = 1 & $GIC\_SH\_CONFIG_{VZE}$ = 0), this GEN control bit is a don't care and is not used to qualify any GIC register accesses.

Even when guest is allowed access to WatchDog timer with GEN = 1, there are further restrictions for guest accesses of certain WatchDog timer related register fields. These further restrictions are listed below,

- Guest has limited access to *GIC_COREi_WD_CONFIG* register:
  - The *WDRESET*, *WAIT* and *DEBUG* fields are read-only 0 for guest.
  - The guest can only set the *TYPE* field with values 0x0 and 0x2 and not the value of 0x1. Thus when guest writes this 3-bit field, the LSB is dropped and for guest reads, the LSB returns 0.
- Guest has limited access to *GIC_COREi_WD_MAP* register.
  - The guest writes to *MAP_TO_NMI* field is further gated by $GIC\_SH\_CONFIG_{GNMI}$ field.

When guest is allowed access to WatchDog timer, the guest may handle the generated WatchDog interrupts without root intervention. To facilitate this, the WatchDog related bits are replicated in *GIC_COREi_[PEND/MASK/RMASK/SMASK]* registers for guest context and guest software is given direct access to them.

The diagrams in following sub sections illustrate the flow for generating RIPL and NMI interrupts from WatchDog timer for root and guest contexts.

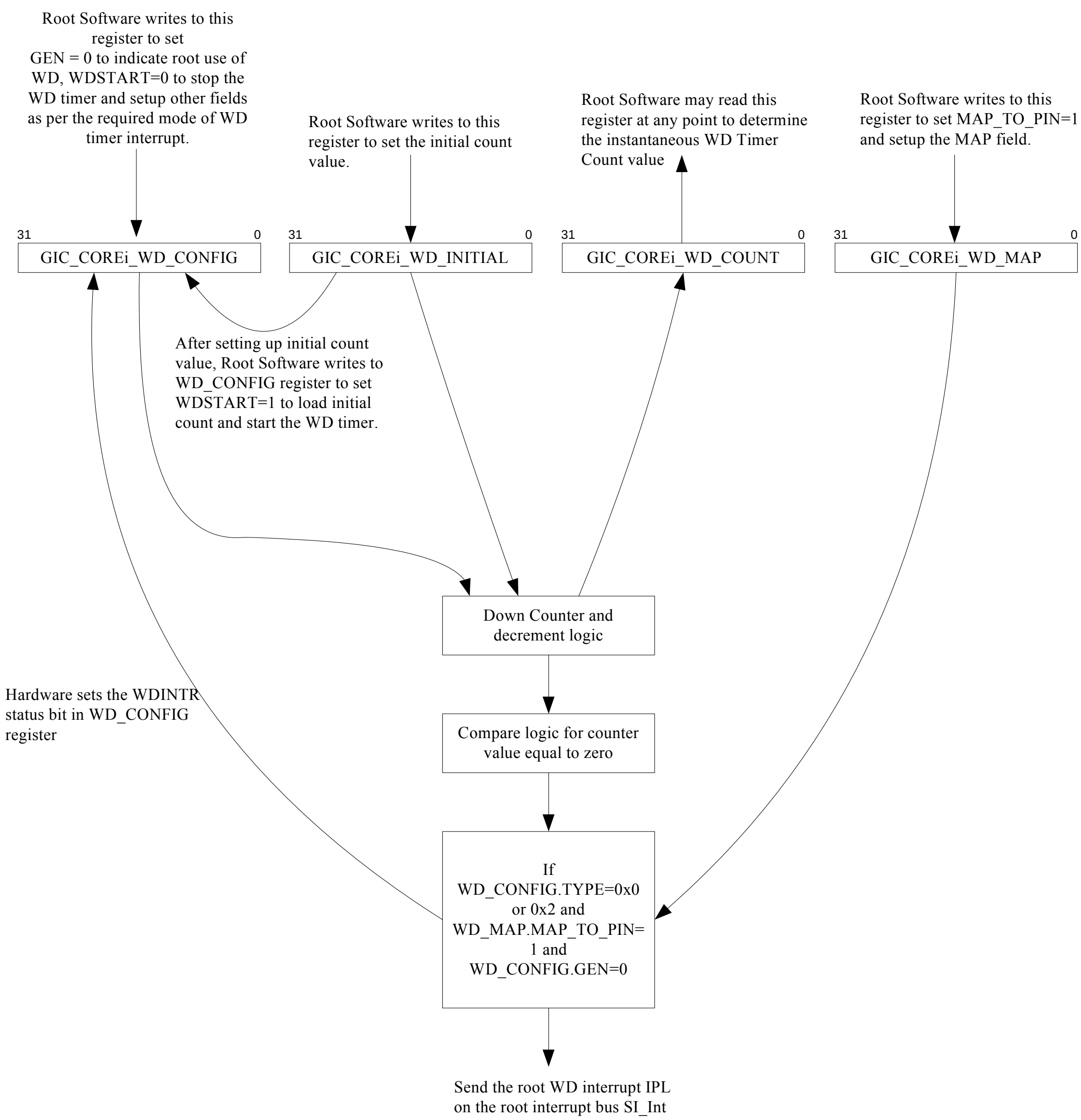
8.4.6 WatchDog Timer RIPL and NMI Generation

The following subsections discus the WatchDog timer generation for the RIPL and NMI interrupts for both root and guest mode.

8.4.6.1 Root Context WatchDog Timer RIPL Generation

Figure 8.13 shows the root context watch dog timer RIPL interrupt generation flow.

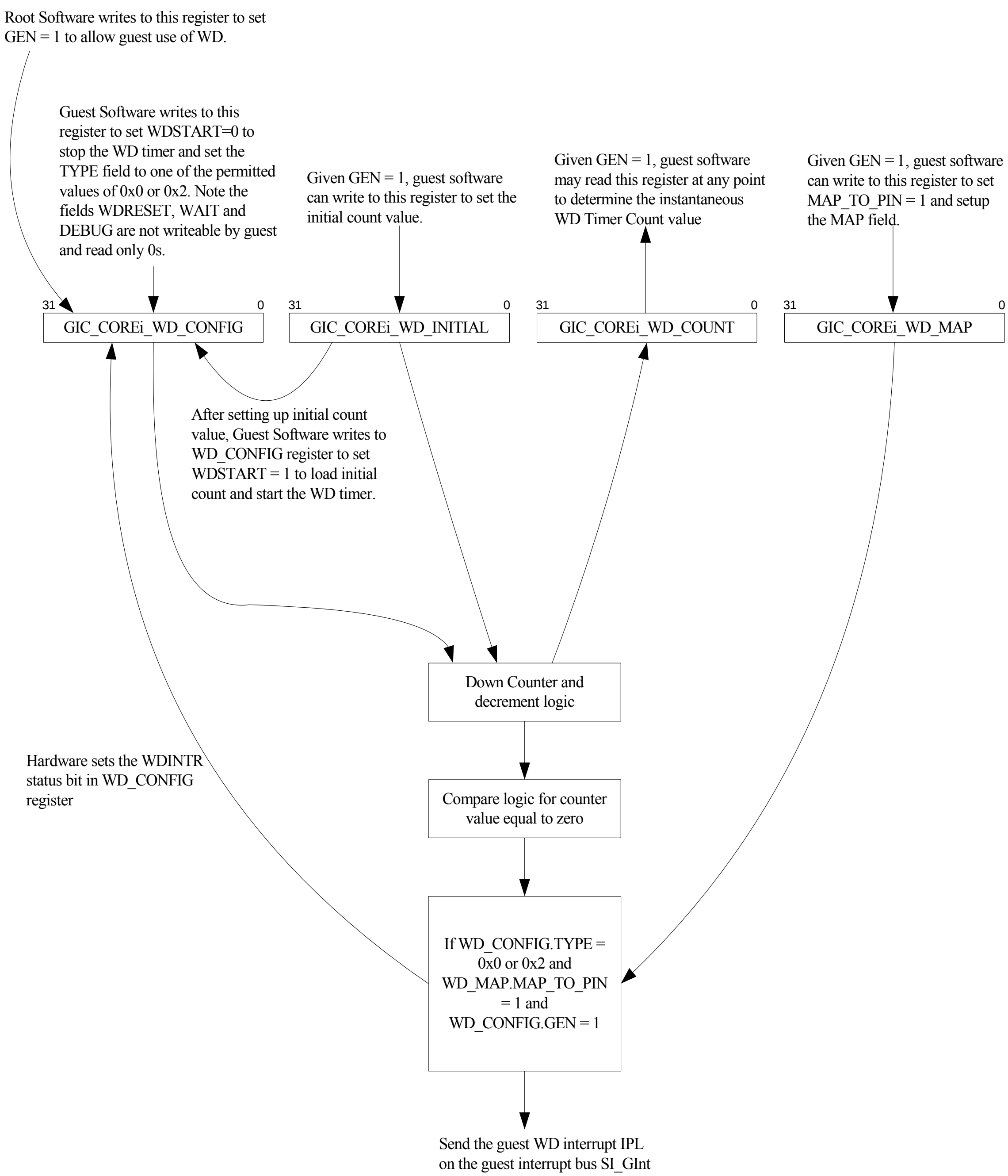
Figure 8.13 Root Context WatchDog Timer RIPL Generation Flow



8.4.6.2 Guest Context WatchDog Timer RIPL Generation

Figure 8.14 shows the guest context watch dog timer RIPL interrupt generation flow.

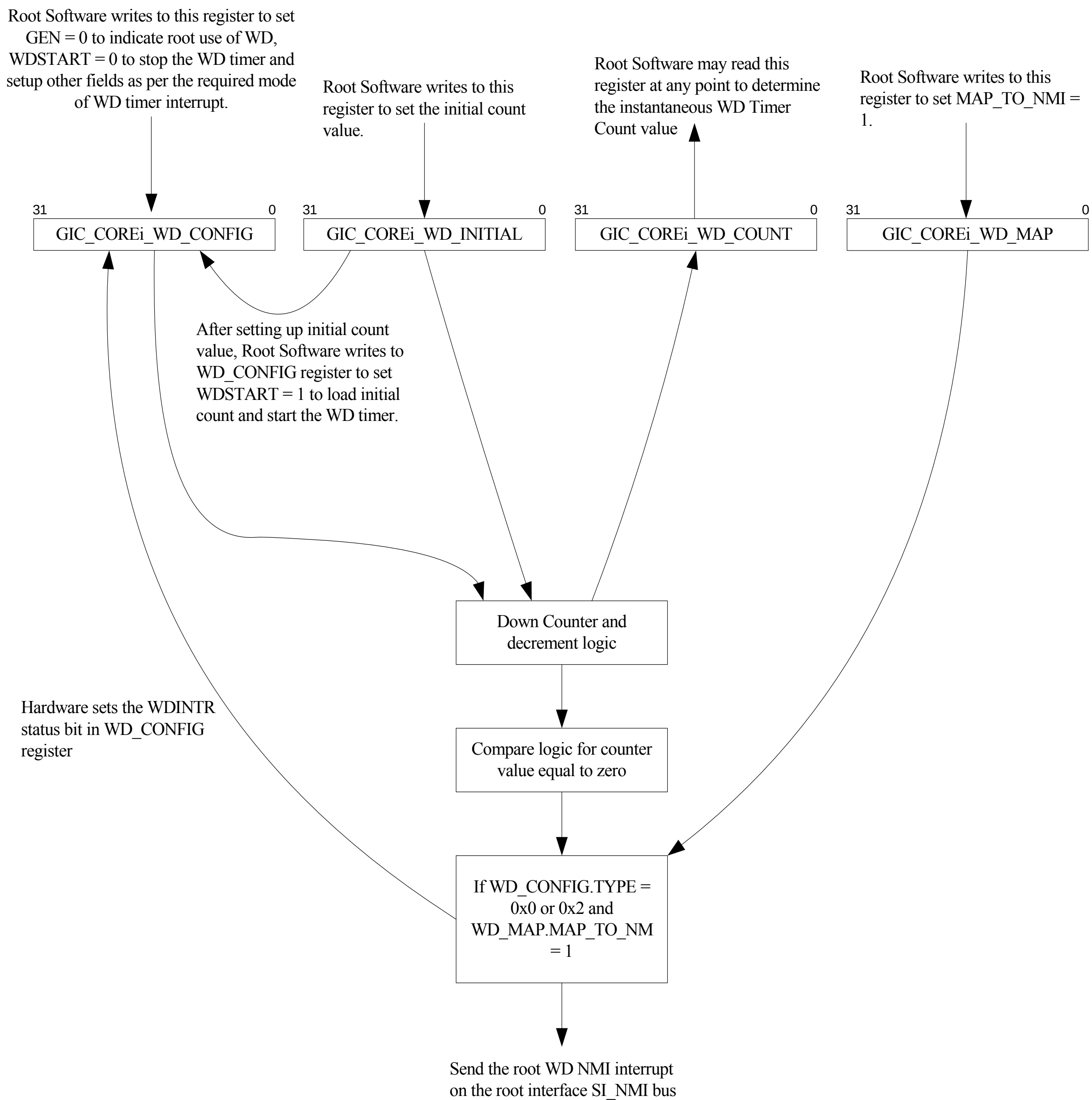
Figure 8.14 Guest Context WatchDog Timer RIPL Generation Flow



8.4.6.3 Root Context WatchDog Timer NMI Interrupt Generation

Figure 8.15 shows the root context watch dog timer NMI interrupt generation flow.

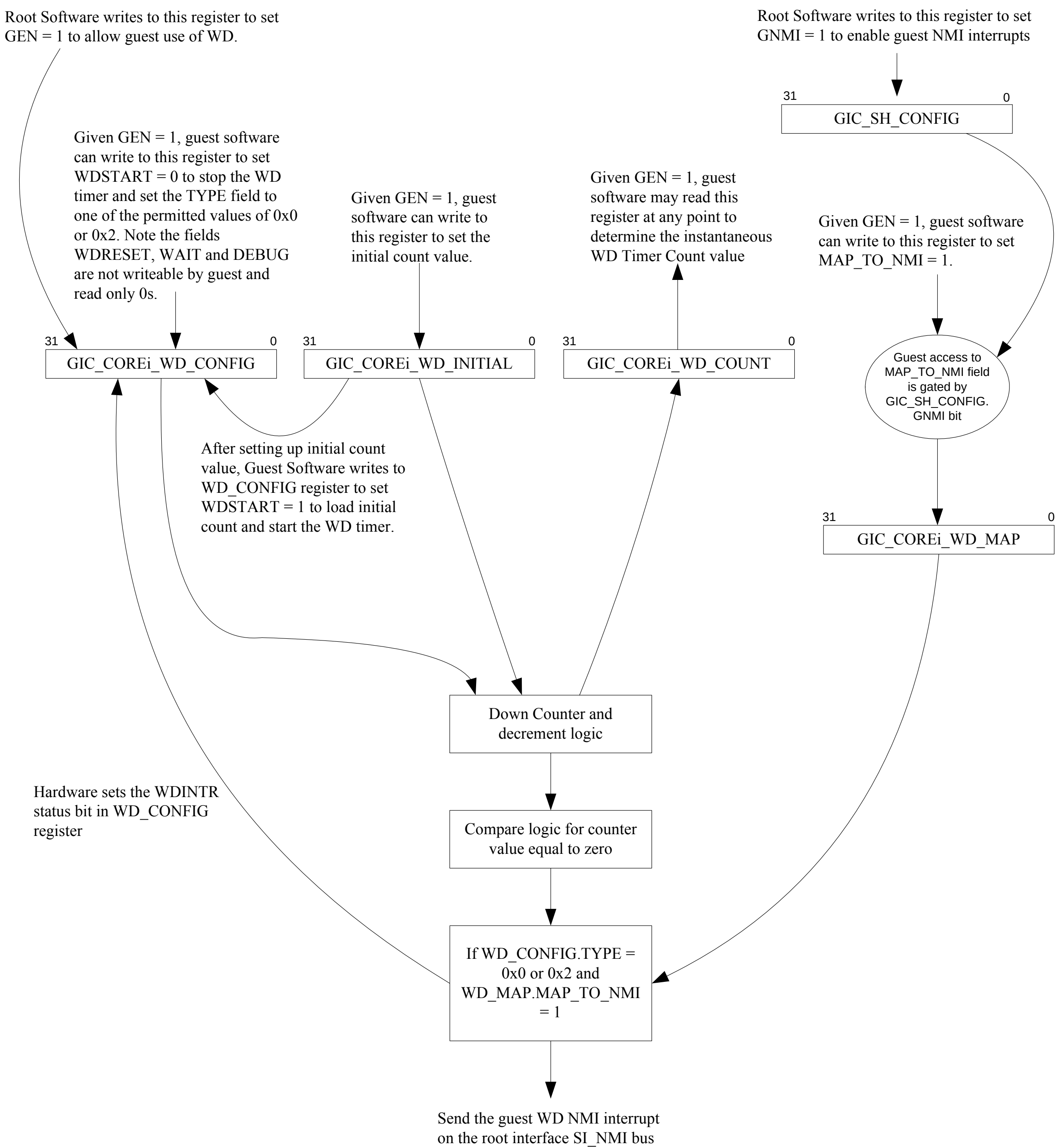
Figure 8.15 Root Context WatchDog Timer NMI Generation Flow



8.4.6.4 Guest Context WatchDog Timer NMI Interrupt Generation

Figure 8.16 shows the guest context watch dog timer NMI interrupt generation flow.

Figure 8.16 Guest Context WatchDog Timer NMI Interrupt Generation Flow



8.5 Shared Register Set

This section describes the various registers in the Shared register set.

8.5.1 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:

Table 8.14 CP0 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 returns a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	
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 returns 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.

8.5.2 Shared Section Register Map

The register map of the shared section is shown in Table 8.15. These registers are accessible by any core. For the base address of this block, see Table 8.1.

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+Register_Offset
```

Table 8.15 Shared Section Register Map

Register Offset	Name	Type	Description
0x0000	GIC Config Register (GIC_SH_CONFIG)	R	Indicates the number of interrupts, number of cores, etc.
0x0010	GIC CounterLo (GIC_SH_CounterLo)	R/W	Shared Global Counter.
0x0014	GIC CounterHi (GIC_SH_CounterHi)	R/W	
0x0020	GIC Revision Register (GIC_RevisionID)	R	RevisionID of the GIC hardware.
0x0024	GIC Interrrupt[31:0] Availability Register (GIC_SH_INT_AVAIL31_0)	R	Indicates the availability of interrupts 0 - 31.
0x0028	GIC Interrrupt[63:32] Availability Register (GIC_SH_INT_AVAIL63_32)	R	Indicates the availability of interrupts 32 - 63.
0x002C	GIC Interrrupt[95:64] Availability Register (GIC_SH_INT_AVAIL95_64)	R	Indicates the availability of interrupts 95 - 64.
0x0030	GIC Interrrupt[127:96] Availability Register (GIC_SH_INT_AVAIL127_96)	R	Indicates the availability of interrupts 96 - 127.
0x0034	GIC Interrrupt[159:128] Availability Register (GIC_SH_INT_AVAIL159_128)	R	Indicates the availability of interrupts 128 - 159.
0x0038	GIC Interrrupt[191:160] Availability Register (GIC_SH_INT_AVAIL191_160)	R	Indicates the availability of interrupts 160 - 191.
0x003C	GIC Interrrupt[223:192] Availability Register (GIC_SH_INT_AVAIL223_192)	R	Indicates the availability of interrupts 192 - 223.
0x0040	GIC Interrrupt[255:224] Availability Register (GIC_SH_INT_AVAIL255_224)	R	Indicates the availability of interrupts 224 - 255.
0x0080	GIC Guest ID Group Configuration Register (GIC_SH_GID_Config31_0)	R	Indicates the availability existence of a physical GuestID register for external interrupts 0 - 31.
0x0084	GIC Guest ID Group Configuration Register (GIC_SH_GID_Config63_32)	R	Indicates the availability existence of a physical GuestID register for external interrupts 32 - 63.
0x0088	GIC Guest ID Group Configuration Register (GIC_SH_GID_Config95_64)	R	Indicates the availability existence of a physical GuestID register for external interrupts 95 - 64.

Table 8.15 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x008C	GIC Guest ID Group Configuration Register (`GIC_SH_GID_Config127_96`)	R	Indicates the availability existence of a physical GuestID register for external interrupts 96 - 127.
0x0090	GIC Guest ID Group Configuration Register (`GIC_SH_GID_Config159_128`)	R	Indicates the availability existence of a physical GuestID register for external interrupts 128 - 159.
0x0094	GIC Guest ID Group Configuration Register (`GIC_SH_GID_Config191_160`)	R	Indicates the availability existence of a physical GuestID register for external interrupts 160 - 191.
0x0098	GIC Guest ID Group Configuration Register (`GIC_SH_GID_Config223_192`)	R	Indicates the availability existence of a physical GuestID register for external interrupts 192 - 223.
0x009C	GIC Guest ID Group Configuration Register (`GIC_SH_GID_Config255_224`)	R	Indicates the availability existence of a physical GuestID register for external interrupts 224 - 255.
0x0100	Global Interrupt Polarity Register0 (`GIC_SH_POL31_0`)	R/W	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.
0x0104	Global Interrupt Polarity Register1 (`GIC_SH_POL63_32`)	R/W	
0x0108	Global Interrupt Polarity Register2 (`GIC_SH_POL95_64`)	R/W	
0x010c	Global Interrupt Polarity Register3 (`GIC_SH_POL127_96`)	R/W	
0x0110	Global Interrupt Polarity Register4 (`GIC_SH_POL159_128`)	R/W	
0x0114	Global Interrupt Polarity Register5 (`GIC_SH_POL191_160`)	R/W	
0x0118	Global Interrupt Polarity Register6 6(`GIC_SH_POL223_192`)	R/W	
0x011c	Global Interrupt Polarity Register7 (`GIC_SH_POL255_224`)	R/W	

Table 8.15 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0180	Global Interrupt Trigger Type Register0 (GIC_SH_TRIG31_0)	R/W	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.
0x0184	Global Interrupt Trigger Type Register1 (GIC_SH_TRIG63_32)	R/W	
0x0188	Global Interrupt Trigger Type Register2 (GIC_SH_TRIG95_64)	R/W	
0x018c	Global Interrupt Trigger Type Register3 (GIC_SH_TRIG127_96)	R/W	
0x0190	Global Interrupt Trigger Type Register4 (GIC_SH_TRIG159_128)	R/W	
0x0194	Global Interrupt Trigger Type Register5 (GIC_SH_TRIG191_160)	R/W	
0x0198	Global Interrupt Trigger Type Register6 (GIC_SH_TRIG223_192)	R/W	
0x019c	Global Interrupt Trigger Type Register7 (GIC_SH_TRIG255_224)	R/W	
0x0200	Global Interrupt Dual Edge Register (GIC_SH_DUAL31_0)	R/W	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.
0x0204	Global Interrupt Dual Edge Register (GIC_SH_DUAL63_32)	R/W	
0x0208	Global Interrupt Dual Edge Register (GIC_SH_DUAL95_64)	R/W	
0x020c	Global Interrupt Dual Edge Register (GIC_SH_DUAL127_96)	R/W	
0x0210	Global Interrupt Dual Edge Register (GIC_SH_DUAL159_128)	R/W	
0x0214	Global Interrupt Dual Edge Register (GIC_SH_DUAL191_160)	R/W	
0x0218	Global Interrupt Dual Edge Register (GIC_SH_DUAL223_192)	R/W	
0x021c	Global Interrupt Dual Edge Register (GIC_SH_DUAL255_224)	R/W	
0x0280	Global Interrupt Write Edge Register (GIC_SH_WEDGE)	W	Used for Interrupt Messages. Writes to this register atomically set or clear a specified bit in the *Edge Detect Register*.

Table 8.15 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0300	Global Interrupt Reset Mask Register (GIC_SH_RMASK31_0)	W	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.
0x0304	Global Interrupt Reset Mask Register (GIC_SH_RMASK63_32)	W	
0x0308	Global Interrupt Reset Mask Register (GIC_SH_RMASK95_64)	W	
0x030c	Global Interrupt Reset Mask Register (GIC_SH_RMASK127_96)	W	
0x0310	Global Interrupt Reset Mask Register (GIC_SH_RMASK159_128)	W	
0x0314	Global Interrupt Reset Mask Register (GIC_SH_RMASK191_160)	W	
0x0318	Global Interrupt Reset Mask Register (GIC_SH_RMASK223_192)	W	
0x031c	Global Interrupt Reset Mask Register (GIC_SH_RMASK255_224)	W	
0x0380	Global Interrupt Set Mask Register (GIC_SH_SMASK31_00)	W	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.
0x0384	Global Interrupt Set Mask Register (GIC_SH_SMASK63_32)	W	
0x0388	Global Interrupt Set Mask Register (GIC_SH_SMASK95_64)	W	
0x038c	Global Interrupt Set Mask Register (GIC_SH_SMASK127_96)	W	
0x0390	Global Interrupt Set Mask Register (GIC_SH_SMASK159_128)	W	
0x0394	Global Interrupt Set Mask Register (GIC_SH_SMASK191_160)	W	
0x0398	Global Interrupt Set Mask Register (GIC_SH_SMASK223_192)	W	
0x039c	Global Interrupt Set Mask Register (GIC_SH_SMASK255_224)	W	

Table 8.15 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0400	Global Interrupt Mask Register (GIC_SH_MASK31_00)	R	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.
0x0404	Global Interrupt Mask Register (GIC_SH_MASK63_32)	R	
0x0408	Global Interrupt Mask Register (GIC_SH_MASK95_64)	R	
0x040c	Global Interrupt Mask Register (GIC_SH_MASK127_96)	R	
0x0410	Global Interrupt Mask Register (GIC_SH_MASK159_128)	R	
0x0414	Global Interrupt Mask Register (GIC_SH_MASK191_160)	R	
0x0418	Global Interrupt Mask Register (GIC_SH_MASK223_192)	R	
0x041c	Global Interrupt Mask Register (GIC_SH_MASK255_224)	R	
0x0480	Global Interrupt Pending Register (GIC_SH_PEND31_00)	R	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.
0x0484	Global Interrupt Pending Register (GIC_SH_PEND63_32)	R	
0x0488	Global Interrupt Pending Register (GIC_SH_PEND95_64)	R	
0x048c	Global Interrupt Pending Register (GIC_SH_PEND127_96)	R	
0x0490	Global Interrupt Pending Register (GIC_SH_PEND159_128)	R	
0x0494	Global Interrupt Pending Register (GIC_SH_PEND191_160)	R	
0x0498	Global Interrupt Pending Register (GIC_SH_PEND223_192)	R	
0x049c	Global Interrupt Pending Register (GIC_SH_PEND255_224)	R	
0x0500	Global Interrupt Map Src0 to Pin Register (GIC_SH_MAP0_PIN)	R/W	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.
0x0504	Global Interrupt Map Src1 to Pin Register (GIC_SH_MAP1_PIN)	R/W	
0x0508	Global Interrupt Map Src2 to Pin Register (GIC_SH_MAP2_PIN)	R/W	
...	...	R/W	
0x08fc	Global Interrupt Map Src255 to Pin Register (GIC_SH_MAP255_PIN)	R/W	

Table 8.15 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x2000	Global Interrupt Map Src0 to Core Register (GIC_SH_MAP0_CORE31_0)	R/W	Assigns this interrupt source to a particular core. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources and the number of cores.
0x2020	Global Interrupt Map Src1 to Core Register (GIC_SH_MAP1_CORE31_0)	R/W	
0x2040	Global Interrupt Map Src2 to Core Register (GIC_SH_MAP2_CORE31_0)	R/W	
.....	....	R/W	
0x3fe0	Global Interrupt Map Src255 to Core Register (GIC_SH_MAP255_CORE31_0)	R/W	
0x6000	DINT Send to Group Register (GIC_VB_DINT_SEND)	R/W	Sends the DebugInterrupt to the specified core.
All other offsets	Reserved for future extensions		Reserved for future extensions.

8.5.3 Shared Section Register Descriptions

The physical address for the Shared Section registers is calculated as follows:

```
GIC_BaseAddress + SharedSection_BaseAddress + RegisterOffset
```

8.5.3.1 Global Config Register (GIC_SH_CONFIG — Offset 0x0000)

Figure 8.17 Global Config Register Format

31	30	29	28	27 ... 24	23 ... 16	15 ... 8	7	6 ... 0
VZP	VZE	IRC	COUNT STOP	COUNTBITS	NUMINTERRUPTS	IRGID	0	PVPES

Table 8.16 GIC Config Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
VZP	31	This bit is set to 1 to indicate that the P6600 GIC supports virtualization.	R	1
VZE	30	Controls the GIC mode of operation. 1: VZ enabled. GIC operates in virtualized mode. 0: VZ disabled. GIC operates in non-virtualized mode.	R/W	0
IRC	29	Interrupt Read Control. Allows root software visibility into root and guest-specific interrupts. 0: Root accesses all register bits unqualified. 1: Root accesses only those register bits that are specific to GIC_VZ_CONFIG.IRGID. This may be root (IRGID = 0), or guest (IRGID = nZ).	R/W	0

Table 8.16 GIC Config Register Bit Descriptions *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COUNTSTOP	28	Setting this bit stops *GIC_CounterHi* and *GIC_CounterLo*. Used to freeze the shared counters when cores go into power-down or debug modes.	R/W	0
COUNTBITS	27:24	Number of Implemented Bits in *GIC_CounterHi*. Total Number of Counter Bits = 32 + COUNTBITS*4, E.g.: 0x0: 32bits, *GIC_CounterHi* not implemented 0x1: 36bits, *GIC_CounterHi* width = 4 bits 0x2: 40bits, *GIC_CounterHi* width = 8 bits ... 0x7: 60bits, *GIC_CounterHi* width = 28 bits 0x8: 64bits, *GIC_CounterHi* width = 32 bits 0x9-0xF: Reserved	R	0x8
NUMINTERRUPTS	23:16	Number of External Interrupt Sources. 0x0: 8 External interrupt sources 0x1: 16 External interrupt sources 0x2: 24 External interrupt sources 0x3: 32 External interrupt sources 0x4: 40 External interrupt sources .... 0x1E: 248 External interrupt sources 0x1F: 256 External interrupt sources Value is fixed by customer at IP configuration time.	R	IP Configuration Value
IRGID	15:8	Interrupt Read Guest ID. Specified GuestID for root read of the shared section registers. Field width matches that of GuestCtl0.GID.	R/W	0
PVPES	6:0	Total number of cores in the system. Note that in the P6600 core, there is one VPE per core. 0: 1 VPE (1 core)	R	IP Configuration Value

8.5.3.2 GIC CounterLo (GIC_SH_CounterLo — Offset 0x0010)

Figure 8.18 GIC CounterLo Register Format

31 ... 0
GIC_SH_CounterLo

Table 8.17 GIC CounterLo Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_CounterLo	31:0	Lower Half of an up-counter. When the counter reaches its maximum value, the counter rolls over to a value of 0x0. The counter is running at an implementation-specific frequency which is fixed, that is, not changing dynamically due to power management. It is recommended that this frequency be as close as possible to the highest clock frequency of the CPU subsystem. This counter is disabled by writing the *COUNTSTOP* bit in the *GIC_SH_CONFIG* register. This counter should only be written when $GIC\_SH\_CONFIG_{COUNTSTOP}$ = 1; otherwise, the registers results after the write are unpredictable.	R/W	0

8.5.3.3 GIC CounterHi (GIC_SH_CounterHi — Offset 0x0014)

Figure 8.19 GIC CounterHi Register Format

31 ... 0
GIC_SH_CounterHi

Table 8.18 GIC CounterHi Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_CounterHi	31:0	Upper Half of an up-counter. When the counter reaches its maximum value, the counter rolls over to a value of 0x0. The counter is running at an implementation-specific frequency which is fixed, that is, not changing dynamically due to power management. It is recommended that this frequency be as close as possible to the highest clock frequency of the CPU subsystem. This counter is disabled by writing the *COUNTSTOP* bit in the *GIC_SH_CONFIG* register. This counter should only be written when $GIC\_SH\_CONFIG_{COUNTSTOP}$ = 1; otherwise, the register results after the write are unpredictable. Unimplemented bits ignore writes and return 0 when read.	R/W	0

8.5.3.4 GIC Revision Register (GIC_RevisionID — Offset 0x0020)

Figure 8.20 GIC Revision Register Format

31 ... 16	15 ... 8	7 ... 0
0	MAJOR_REV	MINOR_REV

Table 8.19 GIC Revision Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
0	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
MAJOR_REV	15:8	This field reflects the major revision of the GIC block. A major revision might reflect the changes from one product generation to another.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the GIC block. A minor revision might reflect the changes from one release to another.	R	Preset

8.5.3.5 Interrupt Availability Registers (GIC_SH_INT_AVAIL — Offsets 0x0024 - 0x0040)

The *GIC_SH_INT_AVAIL* registers indicate which external interrupt sources are available to a guest based on the GuestIDs assigned to external interrupt sources. If guest software is to program interrupts by writing to *GIC_SH_WEDGE* and *GIC_SH_MAPi_PIN* registers, it must first read these *GIC_SH_INT_AVAIL* registers to determine whether it owns the external interrupt source for which it intends to program for interrupts.

The list of guest interrupt availability registers is shown in Table 8.20.

Table 8.20 Guest Interrupt Availability Register Mapping

Offset	Acronym	Register Name
0x0024	GIC_SH_INT_AVAIL31_0	Guest interrupt availability for external interrupts 31:0
0x0028	GIC_SH_INT_AVAIL63_32	Guest interrupt availability for external interrupts 63:32
0x002C	GIC_SH_INT_AVAIL95_64	Guest interrupt availability for external interrupts 95:64
0x0030	GIC_SH_INT_AVAIL127_96	Guest interrupt availability for external interrupts 127:96
0x0034	GIC_SH_INT_AVAIL159_128	Guest interrupt availability for external interrupts 159:128
0x0038	GIC_SH_INT_AVAIL191_160	Guest interrupt availability for external interrupts 191:160
0x003C	GIC_SH_INT_AVAIL223_192	Guest interrupt availability for external interrupts 223:191
0x0040	GIC_SH_INT_AVAIL255_224	Guest interrupt availability for external interrupts 255:192

Figure 8.21 Interrupt Availability Register Format

31 ... 0
GIC_SH_INT_AVAILx_y[1]

1. This format applies to all GIC_SH_INT_AVAIL registers. The x_y indicates the bit range based on Table 8.20 above. For example; x_y = 31:0

Table 8.21 Guest Interrupt Availability Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_INT_AVAILx_y	31:0	Each bit in this register indicates if that corresponding external interrupt source is available for Guest software. 0: The interrupt source is not available to guest software. 1: The interrupt source is available to guest software.	R	0x0

8.5.3.6 ID Group Configuration Registers (GIC_SH_GID_CONFIG, Offsets 0x0080 - 0x009C)

The *GIC_SH_GID_CONFIG* registers provides the information for physical existence of the $GIC\_SH\_MAPi\_PIN_{GID}$ register field for a corresponding indexed external interrupt source.

The list of ID configuration registers is shown in Table 8.22.

Table 8.22 ID Group Configuration Register Mapping

Offset	Acronym	Register Name
0x0080	GIC_SH_GID_CONFIG31_0	Guest ID group configuration register for external interrupts 31:0
0x0084	GIC_SH_GID_CONFIG63_32	Guest ID group configuration register for external interrupts 63:32
0x0088	GIC_SH_GID_CONFIG95_64	Guest ID group configuration register for external interrupts 95:64
0x008C	GIC_SH_GID_CONFIG127_96	Guest ID group configuration register for external interrupts 127:96
0x0090	GIC_SH_GID_CONFIG159_128	Guest ID group configuration register for external interrupts 159:128
0x0094	GIC_SH_GID_CONFIG191_160	Guest ID group configuration register for external interrupts 191:160
0x0098	GIC_SH_GID_CONFIG223_192	Guest ID group configuration register for external interrupts 223:191
0x009C	GIC_SH_GID_CONFIG255_224	Guest ID group configuration register for external interrupts 255:192

Figure 8.22 ID Group Configuration Register Format

31 ... 0
GIC_SH_GID_CONFIGx_y[1]

1. This format applies to all GIC_SH_GID_CONFIG registers. The x_y indicates the bit range based on Table 8.20 above. For example; x_y = 31:0

Table 8.23 ID Group Configuration Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_GID_CONFIGx_y	31:0	Each bit in these registers provides the information for physical existence of the GIC_SH_MAPi_PIN.GID register field for a corresponding indexed external interrupt source. The physical existence of the GIC_SH_MAPi_PIN.GID register field is configured through a build time configuration parameter. This field is encoded as follows: 1: Physical GIC_SH_MAPi_PIN.GID register field exists for corresponding indexed external interrupt source. 0: Physical GIC_SH_MAPi_PIN.GID register field does not exist for the corresponding indexed external interrupt source.	R	0x0

8.5.3.7 Global Interrupt Polarity Registers (GIC_SH_POLx_y — See Table 8.24 for Mapping)

There are eight Global Interrupt Polarity registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Trigger Type (GIC_SH_TRIGn)* and *Global Interrupt Dual Edge (GIC_SH_DUALn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

They are located at the following eight offsets.

Table 8.24 Global Interrupt Polarity Register Mapping

Offset	Acronym	Register Name
0x0100	GIC_SH_POL31_0	Polarity selection for interrupt pins 31:0
0x0104	GIC_SH_POL63_32	Polarity selection for interrupt pins 63:32
0x0108	GIC_SH_POL95_64	Polarity selection for interrupt pins 95:64
0x010C	GIC_SH_POL127_96	Polarity selection for interrupt pins 127:96
0x0110	GIC_SH_POL159_128	Polarity selection for interrupt pins 159:128
0x0114	GIC_SH_POL191_160	Polarity selection for interrupt pins 191:160
0x0118	GIC_SH_POL223_192	Polarity selection for interrupt pins 223:191
0x011C	GIC_SH_POL255_224	Polarity selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_POL63_32 indicates that this register handles the polarity for interrupts 63:32.

Figure 8.23 GIC Interrupt Polarity Register Format

31 ... 0
GIC_SH_POLx_y

Table 8.25 Global Interrupt Polarity Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_POLx_y	31:0	Each bit in this register represents an interrupt source. The state of the bit indicates the polarity of the interrupt. If the interrupt type (as denoted by *Global Interrupt Trigger Type* and *Global Interrupt Dual Edge* registers) is Level triggered, then each bit of this register is encoded as follows: 0: Active Low 1: Active High If the interrupt is single-edge triggered, each bit of this register is encoded as follows: 0: Falling edge denotes interrupt source has toggled 1: Rising edge denotes interrupt source has toggled If the interrupt type is Dual-edge, this register is not used.	R/W	0

8.5.3.8 Global Interrupt Trigger Type Registers (GIC_SH_TRIGx_y — See Table 8.26 for Mapping)

There are eight Global Interrupt Trigger Type registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Polarity (GIC_SH_POLn)* and *Global Interrupt Dual Edge (GIC_SH_DUALn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

They are located at the following eight offsets.

Table 8.26 Global Interrupt Trigger Type Register Mapping

Offset	Acronym	Register Name
0x0180	GIC_SH_TRIG31_0	Interrupt trigger selection for interrupt pins 31:0
0x0184	GIC_SH_TRIG63_32	Interrupt trigger selection for interrupt pins 63:32
0x0188	GIC_SH_TRIG95_64	Interrupt trigger selection for interrupt pins 95:64
0x018C	GIC_SH_TRIG127_96	Interrupt trigger selection for interrupt pins 127:96
0x0190	GIC_SH_TRIG159_128	Interrupt trigger selection for interrupt pins 159:128
0x0194	GIC_SH_TRIG191_160	Interrupt trigger selection for interrupt pins 191:160
0x0198	GIC_SH_TRIG223_192	Interrupt trigger selection for interrupt pins 223:191
0x019C	GIC_SH_TRIG255_224	Interrupt trigger selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_TRIG63_32 indicates that this register handles the trigger level for interrupts 63:32.

Figure 8.24 GIC Interrupt Trigger Type Register Format

31	0
GIC_SH_TRIGx_y	

Table 8.27 Global Interrupt Trigger Type Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_TRIGx_y	31:0	Each bit in this register represents an interrupt source. The state of the bit indicates the nature of the interrupt signaling. 0: Level 1: Edge (Single edge or dual-edge signaling denoted by *Global Interrupt Dual Edge Register*)	R/W	0

8.5.3.9 Global Interrupt Dual Edge Registers (GIC_SH_DUALx_y — See Table 8.28 for Mapping)

There are eight Global Interrupt Dual Edge registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Polarity (GIC_SH_POLn)* and *Global Interrupt Trigger Type (GIC_SH_TRIGn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

They are located at the following eight offsets.

Table 8.28 Global Interrupt Dual Edge Register Mapping

Offset	Acronym	Register Name
0x0200	GIC_SH_DUAL31_0	Interrupt single/dual edge selection for interrupt pins 31:0
0x0204	GIC_SH_DUAL63_32	Interrupt single/dual edge selection for interrupt pins 63:32
0x0208	GIC_SH_DUAL95_64	Interrupt single/dual edge selection for interrupt pins 95:64
0x020C	GIC_SH_DUAL127_96	Interrupt single/dual edge selection for interrupt pins 127:96
0x0210	GIC_SH_DUAL159_128	Interrupt single/dual edge selection for interrupt pins 159:128
0x0214	GIC_SH_DUAL191_160	Interrupt single/dual edge selection for interrupt pins 191:160
0x0218	GIC_SH_DUAL223_192	Interrupt single/dual edge selection for interrupt pins 223:191
0x021C	GIC_SH_DUAL255_224	Interrupt single/dual edge selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_DUAL63_32 indicates that this register handles the edge triggering for interrupts 63:32.

Figure 8.25 GIC Interrupt Dual Edge Register Format

31	0
GIC_SH_DUALx_y	

Table 8.29 Global Dual Edge Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_DUALx_y	31:0	Each bit in this register represents an interrupt source. This register is only meaningful is the equivalent bit in the *Global Interrupt Trigger Type* register is set to 0x1, indicating edge-triggering, in which case each bit of this register is encoded as follows: 0: Single-edge 1: Dual-edge	R/W	0

8.5.3.10 Global Interrupt Write Edge Register (GIC_SH_WEDGE Offset 0x0280)

This register is used to support interrupt messages. A write to this register automatically sets or clears one bit in the *Edge Detect Register*. Setting a bit in this register is equivalent to having the edge detection logic see an active edge. This bypasses the edge detection logic and thus it does not matter whether the corresponding interrupt is configured to be rising, falling, or dual edge sensitive. However, the behavior is undefined unless the equivalent bit in the *Global Interrupt Trigger Type* register is set to 0x1 indicating edge signaling.

Figure 8.26 GIC Interrupt Write Edge Register Format

31	30 ... 0
RW	INTERRUPT

Table 8.30 Global Interrupt Write Edge Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RW	31	Controls whether this write is setting or clearing a bit in the *Edge Detect Register*. If this bit is set, the selected bit in the register is set. If this bit is cleared, the selected bit in the register is cleared.	W	Undefined
Interrupt	30:0	This field is the encoded value of the interrupt that is being cleared or set. For example, a value of 0xB means interrupt 11 (decimal).	W	Undefined

8.5.3.11 Global Interrupt Reset Mask Registers (GIC_SH_RMASKx_y — See Table 8.31 for Mapping)

There are eight Global Interrupt Reset Mask registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)* registers to enable and disable individual interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

These registers are located at the following eight offsets.

Table 8.31 Global Interrupt Reset Mask Register Mapping

Offset	Acronym	Register Name
0x0300	GIC_SH_RMASK31_0	Interrupt reset mask for interrupt pins 31:0
0x0304	GIC_SH_RMASK63_32	Interrupt reset mask for interrupt pins 63:32

Table 8.31 Global Interrupt Reset Mask Register Mapping

Offset	Acronym	Register Name
0x0308	GIC_SH_RMASK95_64	Interrupt reset mask for interrupt pins 95:64
0x030C	GIC_SH_RMASK127_96	Interrupt reset mask for interrupt pins 127:96
0x0310	GIC_SH_RMASK159_128	Interrupt reset mask for interrupt pins 159:128
0x0314	GIC_SH_RMASK191_160	Interrupt reset mask for interrupt pins 191:160
0x0318	GIC_SH_RMASK223_192	Interrupt reset mask for interrupt pins 223:191
0x031C	GIC_SH_RMASK255_224	Interrupt reset mask for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_RMASK63_32 indicates that this register handles the reset mask for interrupts 63:32.

Figure 8.27 GIC Interrupt Reset Mask Register Format

31 ... 0
GIC_SH_RMASKx_y

Table 8.32 Global Interrupt Reset Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_RMASKx_y	31:0	Each bit in this register represents an interrupt source. Writing this register with a 0x1 in any bit position(s) causes only the corresponding bit/interrupt(s) in the *Global Interrupt Mask Register* to be reset (value->0). This is used by software to temporarily disable interrupts.	W	Undefined

8.5.3.12 Global Interrupt Set Mask Registers (GIC_SH_SMASKx_y — See Table 8.33 for Mapping)

There are eight Global Interrupt Set Mask registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Reset Mask (GIC_SH_RMASKn)* registers to enable and disable individual interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

These registers are located at the following eight offsets.

Table 8.33 Global Interrupt Set Mask Register Mapping

Offset	Acronym	Register Name
0x0380	GIC_SH_SMASK31_0	Interrupt set mask for interrupt pins 31:0
0x0384	GIC_SH_SMASK63_32	Interrupt set mask for interrupt pins 63:32
0x0388	GIC_SH_SMASK95_64	Interrupt set mask for interrupt pins 95:64
0x038C	GIC_SH_SMASK127_96	Interrupt set mask for interrupt pins 127:96
0x0390	GIC_SH_SMASK159_128	Interrupt set mask for interrupt pins 159:128
0x0394	GIC_SH_SMASK191_160	Interrupt set mask for interrupt pins 191:160
0x0398	GIC_SH_SMASK223_192	Interrupt set mask for interrupt pins 223:191

Table 8.33 Global Interrupt Set Mask Register Mapping *(continued)*

Offset	Acronym	Register Name
0x039C	GIC_SH_SMASK255_224	Interrupt set mask for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_SMASK63_32 indicates that this register handles the set mask for interrupts 63:32.

Figure 8.28 GIC Interrupt Set Mask Register Format

31 ... 0
GIC_SH_SMASKx_y

Table 8.34 Global Set Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_SMASKx_y	31:0	Each bit in this register represents an interrupt source. Writing this register with a 0x1 in any bit position(s) causes only the corresponding bit/interrupt(s) in the *Global Interrupt Mask Register* to be set (value->0x1). This is used by software to enable interrupts.	W	Undefined

8.5.3.13 Global Interrupt Mask Registers (GIC_SH_MASKx_y — See Table 8.35 for Mapping)

There are eight Global Interrupt Reset Mask registers to cover all 256 possible system interrupts. These read-only registers are used to indicate when an external interrupt occurs. An individual interrupt bit is set when an interrupt occurs and the corresponding Global Interrupt Set Mask bit is set, thereby enabling the interrupt. Refer to Section 8.5.3.12, "Global Interrupt Set Mask Registers (GIC_SH_SMASKx_y — See Table 8.33 for Mapping)" for more information.

These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)* and *Global Interrupt Reset Mask (GIC_SH_RMASKn)* registers to manage and process interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

These registers are located at the following eight offsets.

Table 8.35 Global Interrupt Mask Register Mapping

Offset	Acronym	Register Name
0x0400	GIC_SH_MASK31_0	Interrupt status for interrupt pins 31:0
0x0404	GIC_SH_MASK63_32	Interrupt status for interrupt pins 63:32
0x0408	GIC_SH_MASK95_64	Interrupt status for interrupt pins 95:64
0x040C	GIC_SH_MASK127_96	Interrupt status for interrupt pins 127:96
0x0410	GIC_SH_MASK159_128	Interrupt status for interrupt pins 159:128
0x0414	GIC_SH_MASK191_160	Interrupt status for interrupt pins 191:160
0x0418	GIC_SH_MASK223_192	Interrupt status for interrupt pins 223:191
0x041C	GIC_SH_MASK255_224	Interrupt status for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_MASK63_32 indicates that this register handles the masking for interrupts 63:32.

Figure 8.29 GIC Interrupt Mask Register Format

31	0
GIC_SH_MASKx_y	

Table 8.36 Global Interrupt Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_MASKx_y	31:0	Each bit in this register represents an interrupt source. Reports which of the external interrupt sources are enabled. Used by software to determine which interrupt sources are currently enabled.	R	0x00000000

8.5.3.14 Global Interrupt Pending Registers (GIC_SH_PENDx_y — See Table 8.37 for Mapping)

There are eight Global Interrupt Pending registers to cover the pending status of all 256 possible system interrupts. These read-only registers are set by hardware when an external interrupt is pending.

These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)*, *Global Interrupt Reset Mask (GIC_SH_RMASKn)*, and *Global Interrupt Mask (GIC_SH_MASKn)* registers to manage and process interrupts. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

These registers are located at the following eight offsets.

Table 8.37 Global Interrupt Pending Register Mapping

Offset	Acronym	Register Name
0x0480	GIC_SH_PEND31_0	Interrupt pending status for interrupt pins 31:0
0x0484	GIC_SH_PEND63_32	Interrupt pending status for interrupt pins 63:32
0x0488	GIC_SH_PEND95_64	Interrupt pending status for interrupt pins 95:64
0x048C	GIC_SH_PEND127_96	Interrupt pending status for interrupt pins 127:96
0x0490	GIC_SH_PEND159_128	Interrupt pending status for interrupt pins 159:128
0x0494	GIC_SH_PEND191_160	Interrupt pending status for interrupt pins 191:160
0x0498	GIC_SH_PEND223_192	Interrupt pending status for interrupt pins 223:191
0x049C	GIC_SH_PEND255_224	Interrupt pending status for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_PEND63_32 indicates that this register handles the interrupt pending status for interrupts 63:32.

Figure 8.30 GIC Interrupt Pending Register Format

31	0
GIC_SH_PENDx_y	

Table 8.38 Global Interrupt Pending Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_PENDx_y	31:0	There are eight Interrupt Pending register that are used to indicate the pending status of all 256 possible interrupts in the system Each bit indicates which of the external interrupt sources are asserted/pending before masking. Used by software to find the external source that caused the CPU interrupt.	R	Undefined

8.5.3.15 Global Interrupt Map to Pin Registers (GIC_SH_MAPx_y)

There are up to 256 Global Interrupt Map-to-Pin registers in the GIC to cover the mapping of all 256 possible system interrupts. This corresponds to one register per external interrupt signal. The number of registers instantiated at build time depends on the number of external system interrupts. These are write-only registers. Software is not expected to change these registers frequently. Software is expected to keep a back-up copy of these registers in memory so that Read-Modify-Write hazards are avoided.

Each interrupt pin can be mapped to one of three signal types: *SI_Int[5:0]* or *SI_NMI*. Bits 31:30 of this register are used to indicate to which signal type the interrupt is mapped. Only one of these bits can be set at any given time. Bits 5:0 indicate the actual mapping for each external interrupt pin. For example, if bit 31 of this register is set, the external interrupt is routed to the *SI_Int[5:0]* pins of the appropriate core .

For the register offset addresses corresponding to each register, refer Table 8.5, "Mapping of External Interrupts"

Figure 8.31 GIC Interrupt Map to Pin Register Format

31	30	29 — 16	15 — 8	7 — 6	5 — 0
MAP_TO_PIN	MAP_TO_NMI	R	GID	R	MAP

Table 8.39 Global Interrupt Map to Pin Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
MAP_TO_PIN	31	If this bit is set, this interrupt source is mapped to a core interrupt pin (specified by the *MAP* field below). Only one of the *MAP_TO_PIN* or *MAP_TO_NMI* bits can be set at any one time.	RW	0x1
MAP_TO_NMI	30	If this bit is set, this interrupt source is mapped to NMI. Only one of the *MAP_TO_PIN* or *MAP_TO_NMI,* or *MAP_TO_YQ* bits can be set at any one time.	RW	0
Reserved	29:16	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0

Table 8.39 Global Interrupt Map to Pin Register Bit Descriptions *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
GID	15:8	This field contains the Guest ID of the guest context to which this interrupt is targeted. The Hypervisor is expected to program this field prior to initializing interrupts in the system. This field is set to zero if this interrupt is to be assigned on the root interrupt bus of the core. To optimize for area, a group of external interrupt sources may share a common GID field value and thus the GID register field may not have a physical existence for the higher indexed external interruptsources of the group. The physical existence of this register field is controlled via a build time parameter. In the case where a physical GID register field does not exist for an external interrupt source, that external interrupt source uses the GID field value from whatever the next lower indexed external interrupt source which has a physical GID register field. Software can determine the physical existence of this register field by reading the GuestID Group Config Registers. Any writes to a physically non-existing GID field is discarded and thus does not alter the group's GID. Any reads from a physically non-existing GID field returns the group's GID value.	R/W	4
MAP	5:0	When the *MAP_TO_PIN* bit is set, this field contains the encoded value of the core interrupts signals *Int[62:0]*. In EIC mode, this represents one less than the EIC interrupt level (e.g. a value of 0x20 represents interrupt level 21). For non-EIC mode, the value represents the CPU interrupt to be asserted (e.g. a value of 0x03 represents interrupt 3), and only values of 0 to 5 are legal. When virtualization is supported in EIC mode, the root assigned interrupts should be programmed with a higher RIPL than the guest assigned interrupts. This condition is only applicable to root and guest assigned interrupts which are programmed to route to the same core. (This description needs to be added on top of the existing description for the MAP field.)	RW	0

8.5.3.16 Global Interrupt Map to Core Registers (GIC_SH_MAPn_CORE31:0) — See Table 8.5 for Mapping)

There are up to 512 Global Interrupt Map-to-Core registers in the GIC to cover the mapping of all 256 possible system interrupts. This corresponds to two registers per external interrupt signal. However, the high-order register is not used in the P6600 core as described in Section 8.5.3.16, "Global Interrupt Map to Core Registers (GIC_SH_MAPn_CORE31:0) — See Table 8.5 for Mapping)".

The number of registers instantiated at build time depends on the number of external system interrupts. These are write-only registers. Software is not expected to change these registers frequently. Software is expected to keep a back-up copy of these registers in memory so that Read-Modify-Write hazards are avoided.

For the register offset addresses corresponding to each register, refer Table 8.5, "Mapping of External Interrupts"

Figure 8.32 GIC Interrupt Map to Core31:0 Register Format

31 … 0
GIC_SH_MAPi_COREn

Table 8.40 Global Interrupt Map to Core31:0 Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_MAPi_COREn	31:0	Setting any bit in this register causes the interrupt source to be routed to the corresponding core. For all GIC_SH_MAPi_CORE registers, only one bit may be set at a time. That is, an interrupt source is routed to one and only one core.	W	0

8.5.3.17 DINT Send to Group Register (GIC_VB_DINT_SEND Offset 0x6000)

This register allows software to assert the EJ_DINT_GROUP signal directly. Refer to Section 8.3.11 "Debug Interrupt Generation" for more information.

Figure 8.33 DINT Send to Group Register Format

31 … 1	0
R	SEND_DINT

Table 8.41 DINT Send to Group Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	[31:1]	Read as Zero. Writes ignored.	-	0x0
SEND_DINT	[0]	If this register field is written with a value of 0x1, the *EJ_DINT_GROUP* signal is asserted in a one-shot manner.	W	0x0

See Chapter 14, "Multi-CPU Debug" on page 735 for more information about how this register is used.

8.6 GIC Core-Local and Core-Other Register Set

8.6.1 Core-Local and Core-Other Register Maps

The Core-Local and Core-Other interrupt register maps are described in Table 8.42 below. For the base addresses of these blocks, see Table 8.1. Each core in the P6600 core contains a set of these registers.

The physical address for the registers within the Core-Local section are calculated as follows:

```
Core-Local_Register_Physical_Address = GIC_BaseAddress + Core-Local_BaseOffset +
Register Offset
```

Similarly, for the Core-Other section:

```
Core-Other_Register_Physical_Address = GIC_BaseAddress + Core-Other_BaseOffset +
Register Offset
```

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 returns 0x0, and writes to those locations is silently dropped without generating any exceptions.

Table 8.42 Core-Local and Core-Other Register Maps

Register Offset	Name	Type	Description
0x0000	Local Interrupt Control Register (`GIC_COREi_CTL`)	R/W	Enable EIC Mode.
0x0004	Local Interrupt Pending Register (`GIC_COREi_PEND`)	R	Status of the local interrupts before masking. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x0008	Local Mask Register (`GIC_COREi_MASK`)	R	Mask bits, if set, enables the corresponding interrupts in the interrupt vector. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x000c	Local Reset Mask Register (`GIC_COREi_RMASK`)	W	Setting a bit in this register causes the corresponding bits in the *GIC_COREi_MASK* register to be cleared atomically with respect to other bits. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.

Table 8.42 Core-Local and Core-Other Register Maps *(continued)*

Register Offset	Name	Type	Description
0x0010	Local Set Mask Register (`GIC_COREi_SMASK`)	W	Setting a bit in this register causes the corresponding bits in the *GIC_COREi_MASK* register to be set atomically with respect to other bits. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x0040	Local WatchDog Map-to-Pin Register (`GIC_COREi_WD_MAP`)	R/W	This register is used to route the local WatchDog interrupt to the desired core pin.
0x0044	Local GIC Counter/Compare Map-to-Pin Register (`GIC_COREi_COMPARE_MAP`)	R/W	This register is used to route the local GIC Compare/Count Interrupt to the desired core pin. This is an optional register instantiated at IP configuration time. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x0048	Local CPU Timer Map-to-Pin Register (`GIC_COREi_TIMER_MAP`)	R/W	This register is used to route the local CPU Timer interrupt to the desired core pin. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x004c	Local CPU Fast Debug Channel Map-to-Pin Register (`GIC_COREi_FDC_MAP`)	R/W	This register is used to route the local CPU Fast Debug Channel interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0050	Local Perf Counter Map-to-Pin Register (`GIC_COREi_PERFCTR_MAP`)	R/W	This register is used to route the local Performance Counter interrupt to the desired core pin. This is an optional register instantiated at IP configuration time. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.

Table 8.42 Core-Local and Core-Other Register Maps *(continued)*

Register Offset	Name	Type	Description
0x0054	Local SWInt0 Map-to-Pin Register (GIC_COREi_SWInt0_MAP)	R/W	This register is used to route the local SWInt0 interrupt to the desired core pin. This is an optional register instantiated at IP configuration time. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x0058	Local SWInt1 Map-to-Pin Register (GIC_COREi_SWInt1_MAP)	R/W	This register is used to route the local SWInt1 interrupt to the desired core pin. This is an optional register instantiated at IP configuration time. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x0080	Core-Other Addressing Register (GIC_COREi_OTHER_ADDR)	R/W	Sets the *VPENum* of the register that is accessed through the Core-Other address space.
0x0088	Core-Local Identification Register (GIC_COREi_IDENT)	R	Indicates the Core number of the local Core.
0x0090	Programmable/Watchdog Timer0 Config Register (GIC_COREi_WD_CONFIG0)	R/W	Local Programmable or Watchdog Timer0 related registers. See register description for more details.
0x0094	Programmable/Watchdog Timer0 Count Register (GIC_COREi_WD_COUNT0)	R	
0x0098	Programmable/Watchdog Timer0 Initial Count Register (GIC_COREi_WD_INITIAL0)	R/W	
0x00A0	CompareLo Register (GIC_COREi_CompareLo)	R/W	Compare Register. See register description for more details. Note that for each offset address, there are two copies of each register. One copy is for the root and the other copy is for the guest. Refer to Section 8.6.2, "Guest and Root Register Accesses" for more information.
0x00A4	CompareHi Register (GIC_COREi_CompareHi)	R	
0x0200	Core-Local Counter Offset Register (GIC_COREi_COFFSET)	R/W	Stores the counter offset.
0x3000	Core-Local DINT Group Participate Register (GIC_VL_DINT_PART GIC_VO_DINT_PART)	R/W	Controls whether this core pays attention to the *DebugInt_GroupRequest* register.
0x3080	Core-Local DebugBreak Group Register (GIC_VL_BRK_GROUP GIC_VO_BRK_GROUP)	R/W	Allows multiple Core to simultaneously enter Debug Mode.
All Other Offsets	RESERVED		Reserved for Future Extensions.

8.6.2 Guest and Root Register Accesses

As shown in the above table, the P6600 core supports both Root and Guest registers. When virtualization is enabled in the P6600 core, there are two copies of these registers at the same address offset. One copy is for the root and the other copy is for the guest. The root software can accesses the root copy and also the guest copy by setting the R2GEN field to ‘1’ in GIC_COREi_CTL register. Refer to Section 8.6.3.1, "Local Interrupt Control Register (GCI_COREi_CTL — Offset 0x0000)" for more information. The guest software cannot access the root copy and only the qualified guests may accesses the guest copy of this register.

8.6.3 Core-Local and Core-Other Section Register Description

The following subsections describes the registers of the Core-Local and Core-Other sections.

8.6.3.1 Local Interrupt Control Register (GCI_COREi_CTL — Offset 0x0000)

Figure 8.34 Local Interrupt Control Register Format

31 … 7	6	5	4 … 1	0
R	GNMI	R2GEN	R	EIC_MODE

Table 8.43 Local Interrupt Control Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
RESERVED	31:7	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
GNMI	6	This allows the root control over guest NMI. Applies to core-local guest NMI sources. The Guest NMI enable is encoded as follows: 0: Guest NMI disabled 1: Guest NMI enabled	R/W	0
R2GEN	5	This bit enables root R/W to duplicate guest registers at same address. 0: Root accesses root copy at address. 1: Root accesses guest copy at address.	R/W	0
RESERVED	4:1	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
EIC_MODE	0	Writing a 1 to this bit sets the local interrupt controller to EIC (External Interrupt Controller) mode.	R/W	0

8.6.3.2 Local Interrupt Pending Register (GIC_COREi_PEND — Offset 0x0004)

This register stores the local interrupt pending information before masking.

Figure 8.35 Local Interrupt Pending Register Format

31 ... 7	6	5	4	3	2	1	0
R	FDC _PEND	SWINT1 _PEND	SWINT0 _PEND	PERFCOUNT _PEND	TIMER _PEND	COMPARE _PEND	WD _PEND

Table 8.44 Local Interrupt Pending Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:7	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
FDC_PEND	6	Indicates the status of the local Fast Debug Channel interrupt prior to masking.	R	Undefined
SWINT1_PEND	5	Indicates the status of the local software interrupt 1 prior to masking.	R	Undefined
SWINT0_PEND	4	Indicates the status of the local software interrupt 0 prior to masking.	R	Undefined
PERFCOUNT_PEND	3	Indicates the status of the local Performance Counter interrupt prior to masking.	R	Undefined
TIMER_PEND	2	Indicates the status of the local CPU Timer interrupt prior to masking.	R	Undefined
COMPARE_PEND	1	Indicates the status of the local Count/Compare interrupt prior to masking.	R	Undefined
WD_PEND	0	Indicates the status of the local WatchDog interrupt prior to masking.	R	Undefined

8.6.3.3 Local Interrupt Mask Register (GCI_COREi_MASK — Offset 0x0008)

This is a read-only register. Refer to Section 8.3.3, "Configuring Interrupt Sources" for more information.

Figure 8.36 Local Interrupt Mask Register Format

31 ... 7	6	5	4	3	2	1	0
R	FDC_ MASK	SWINT1_ MASK	SWINT0_ MASK	PERFCOUNT_ MASK	TIMER_ MASK	COMPARE_ MASK	WQ_MASK

Table 8.45 Local Interrupt Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:7	Read as 0x0	R	0x0000_00
FDC_MASK	6	If this bit is set, the local Fast Debug Channel interrupt is enabled.	R	1
SWINT1_MASK	5	If this bit is set, the local software interrupt 1 is enabled.	R	1

Table 8.45 Local Interrupt Mask Register Bit Descriptions *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
SWINT0_MASK	4	If this bit is set, the local software interrupt 0 is enabled.	R	1
PERFCNT_MASK	3	If this bit is set, the local Performance Counter Interrupt is enabled.	R	1
TIMER_MASK	2	If this bit is set, the local CPU Timer Interrupt is enabled.	R	1
COMPARE_MASK	1	If this bit is set, the local Count/Compare Interrupt is enabled.	R	1
WQ_MASK	0	If this bit is set, the local WatchDog Interrupt is enabled.	R	1

8.6.3.4 Local Interrupt Reset Mask Register (GCI_COREi_RMASK — Offset 0x000C)

Figure 8.37 Local Interrupt Reset Mask Register Format

31 … 7	6	5	4	3	2	1	0
R	FDC_ RMASK	SWINT1_ RMASK	SWINT0_ RMASK	PERFCOUNT_ RMASK	TIMER_ RMASK	COMPARE_ RMASK	WQ_RMASK

Table 8.46 Local Interrupt Reset Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:7	Writes ignored. Must be written with a value of 0x0.		Undefined
FDC_RMASK	6	Writing a 0x1 to this bit disables the local Fast Debug Channel interrupt	W	Undefined
SWINT1_RMASK	5	Writing a 0x1 to this bit disables the local software interrupt (SWInt1).	W	Undefined
SWINT0_RMASK	4	Writing a 0x1 to this bit disables the local software interrupt (SWInt0).	W	Undefined
PERFCNT_RMASK	3	Writing a 0x1 to this bit disables the local Performance Counter Interrupt.	W	Undefined
TIMER_RMASK	2	Writing a 0x1 to this bit disables the local Timer Interrupt.	W	Undefined
COMPARE_RMASK	1	Writing a 0x1 to this bit disables the local Count/Compare Interrupt.	W	Undefined
WQ_RMASK	0	Writing a 0x1 to this bit disables the local WatchDog Timer Interrupt.	W	Undefined

8.6.3.5 Local Interrupt Set Mask Register (GCI_COREi_SMASK — Offset 0x0010)

This is a write-only register. For more information, refer to Section 8.3.3, "Configuring Interrupt Sources".

Figure 8.38 Local Interrupt Set Mask Register Format

31 7	6	5	4	3	2	1	0
R	FDC SMASK	SWINT1_ SMASK	SWINT0_ SMASK	PERFCOUNT_ SMASK	TIMER_ SMASK	COMPARE_ SMASK	WQ_SMASK

Table 8.47 Local Interrupt Set Mask Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:7	Writes ignored. Must be written with a value of 0x0.		Undefined
FDC_SMASK	6	Writing a 0x1 to this bit sets the local Fast Debug Channel Interrupt	W	Undefined
SWINT1_SMASK	5	Writing a 0x1 to this bit sets the local SWInt1 interrupt mask.	W	Undefined
SWINT0_SMASK	4	Writing a 0x1 to this bit sets the local SWInt0 interrupt mask.	W	Undefined
PERFCNT_SMASK	3	Writing a 0x1 to this bit sets the local performance counter interrupt mask.	W	Undefined
TIMER_SMASK	2	Writing a 0x1 to this bit sets the local Timer Interrupt mask.	W	Undefined
COMPARE_SMASK	1	Writing a 0x1 to this bit sets the local GIC Count/Compare Interrupt mask.	W	Undefined
WQ_SMASK	0	Writing a 0x1 to this bit sets the local WatchDog Timer Interrupt mask.	W	Undefined

8.6.3.6 Local Map to Pin Registers (Offset 0x0040 - 0x0058 — See Table 8.48 for Mapping)

This section includes the local map to pin registers described in Table 8.48. The bit assignments for each of these registers is identical. There is one register per instantiated core. The 'i' indicates a number between 1 and 6 6depending on the number of cores in the system.

Table 8.48 Local Map-to-Pin Register Mapping

Offset	Acronym	Register Name
0x0040	GIC_COREi_WD_MAP	Local Watchdog Map-to-Pin register.
0x0044	GIC_COREi_COMPARE_MAP	Local Counter/Compare Map-to-Pin register.
0x0048	GIC_COREi_TIMER_MAP	Local Timer Map-to-Pin register.
0x004C	GIC_COREi_FDC_MAP	Local Fast Debug Channel Map-to-Pin register.
0x0050	GIC_COREi_PERFCTR_MAP	Local Performance Counter Map-to-Pin register.
0x0054	GIC_COREi_SWInt0_MAP	Local Software Interrupt 0 Map-to-Pin register.
0x0058	GIC_COREi_SWInt1_MAP	Local Software Interrupt 1 Map-to-Pin register.

Figure 8.39 Local Map-to-Pin Register Format

31	30	29 — 6	5 — 0
MAP_TO_PIN	MAP_TO_NMI	R	MAP

Table 8.49 Local Map to Pin Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
MAP_TO_PIN	31	If this bit is set, this interrupt source is mapped to a core interrupt pin (specified by the *MAP* field below). Only one of the *MAP_TO_PIN* or *MAP_TO_NMI* bits can be set at any one time.	R/W	0x1 for Timer, PerfCount and SWIntx; 0x0 for WatchDog
MAP_TO_NMI	30	If this bit is set, this interrupt source is mapped to a core NMI interrupt pin of the root interface. Note the the root controls the generation of NMI interrupts from guest interrupt sources and thus the software access to this bit is gated by GIC_SH_CONFIG.GNMI field setting. Only one of the *MAP_TO_PIN or MAP_TO_NMI* bits can be set at any one time.	R/W	0x1 for WatchDog; 0x0 for Others
R	29:6	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
MAP	5:0	When the *MAP_TO_PIN* bit is set, this field contains the encoded value of guest interrupts signals *SI_Int[5:0]* (for root), and *SI_GInt[5:0]* (for guest). In EIC mode, this represents one less than the EIC interrupt level (e.g. a value of 0x20 represents interrupt level 21). For non-EIC mode, the value represents the CPU interrupt to be asserted (e.g. a value of 0x03 represents interrupt 3), and only values of 0 to 5 are legal. Also in non-EIC mode, the guest software is not allowed write accesses to this field.	W	0x5 for Timer, PerfCount, and Fast Debug Channel, 0x0 for all others

8.6.3.7 Core-Other Addressing Register (GCI_COREi_OTHER_ADDR — Offset 0x0080)

This register must be written with the correct value before accessing the Core-Other address section.

Figure 8.40 Core-Other Addressing Register Format

31 — 16	15 — 0
R	VPENUM

Table 8.50 Core-Other Addressing Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0

Table 8.50 Core-Other Addressing Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
VPENUM	15:0	Number of the register set to be accessed in the Core-Other address space. Note that in the P6600 core, there is one VPE per core, hence a VPE and a core are the same thing.	R/W	0

8.6.3.8 Core-Local Identification Register (GCI_COREi_IDENT — Offset 0x0088)

The aliased memory scheme is normally invisible to software when accessing GIC registers within the Core-Local Control Block. What actually happens is that an offset is used to make a subset of the GIC registers appear in the Core-Local addressing Window.

This register reports the Core number that is used as the addressing offset for the Core-Local Control Block.

Figure 8.41 Core-Local Addressing Register Format

31 ... 0
CORENUM

Table 8.51 Core-Local Identification Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
CORE*NUM*	31:0	This number is used as an index to the registers within the GIC when accessing the Core-local control block for this core. Note that in the P6600 core, there is one VPE per core, hence a VPE and a core are the same thing.	R	-

8.6.4 Local Timer Register Descriptions

8.6.4.1 Watchdog Timer Config Register (GCI_COREi_WD_CONFIG0 — Offset 0x0090)

For more information on the usage of this register, refer to Section 8.3.7.2, "GIC Watchdog Timer".

Figure 8.42 Watchdog Timer Config Register Format

31 … 9	8	7	6	5	4	3 … 1	0
R	GEN	WDRESET	WDINTR	WAIT	DEBUG	TYPE	WDSTART

Table 8.52 Watchdog Timer Config Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:9	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
GEN	8	Guest Enable for WatchDog timer use. Only the Root has access to this bit and it allows the root to control the guest software access to WatchDog timer related registers (GIC_COREi_WD_[MAP/CONFIG/COUNT). This bit is encoded as follows: 0 : Guest software not allowed access to WatchDog timer related registers. 1 : Guest software allowed access to WatchDog timer related registers.	R/WC	0
WDRESET	7	Status bit which indicates that a Watchdog was responsible for resetting the P6600 MPS. A write of 0x1 to this bit of this register automatically clears this bit. This bit needs to survive a watchdog triggered reset.	R/WC	0
WDINTR	6	Status bit which indicates that a Watchdog was responsible for generating this interrupt. A write of 0x1 to this bit automatically clears the bit. Typically this interrupt is routed to the *NMI* interrupt input of the core, but could be routed to another interrupt as well.	R/WC	Undefined
WAIT	5	Stop countdown if the core is in an implementation-defined low power mode (including the mode which is entered on a WAIT instruction). 0x0 - Stop countdown if core is in low power mode. 0x1 - Low power mode has no effect on countdown.	R/W	0
DEBUG	4	Stop countdown if the core is in debug mode. 0x0 - Stop countdown if core is in Debug Mode (CP0 $DEBUG_{DM}$ bit is set). 0x1 - Debug Mode has no effect on countdown.	R/W	0

Table 8.52 Watchdog Timer Config Register Bit Descriptions *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
TYPE	3:1	Interrupt type. There are *three* ways to setup the watchdog timer which are encoded into this field: 0x0: WD One Trip Mode. Once the counter decrements to 0x0, it causes an interrupt, typically and NMI, and then stops. 0x1: WD Second Countdown Mode. Once the counter decrements to 0x0, the initial value is reloaded and the countdown continues. If on the second trip, the counter reaches 0x0, the *SI_Reset* signal is asserted to all cores in the system. 3. Programmable Interrupt Timer (PIT) Mode. This asserts an interrupt, reloads, and keeps going.	R/W	0
WD_START	0	Watchdog timer start/stop. Setting this bit starts the Watchdog timer, while clearing the bit stops the timer. 0 - Stop the Watchdog timer 1 - Reload the initial count and start the Watchdog timer.	R/W	0

8.6.4.2 Watchdog Timer Count Register (GIC_COREi_WD_COUNT — Offset 0x0094)

For more information on the usage of this register, refer to Section 8.3.8.3, "Watchdog Timer Interrupts".

Figure 8.43 Watchdog Timer Count Register Format

31 ... 0
COUNT

Table 8.53 Watchdog Timer Count Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
COUNT	31:0	This read-only register indicates the state of the decrementing counter. The width of the counter is 32 bits.	R	Undefined

8.6.4.3 Watchdog Timer Initial Count Register (GIC_COREi_WD_INITIAL — Offset 0x0098)

For more information on the usage of this register, refer to Section 8.3.8.3, "Watchdog Timer Interrupts".

Figure 8.44 Watchdog Timer Initial Count Register Format

31 — 0
INIT

Table 8.54 Watchdog Timer Initial Count Register

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
INIT	31:0	Initial value to be loaded into the Watchdog counter. Needs to be done with the counter disabled; otherwise, the results are UNPREDICTABLE.	R/W	Undefined

8.6.4.4 Compare Low Register (GCI_COREi_ComparLo — Offset 0x00A0)

For more information on the usage of this register, refer to Section 8.3.8.3, "Watchdog Timer Interrupts".

Figure 8.45 CompareLo Register Format

31 — 0
COMPARELO

Table 8.55 CompareLo Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COMPARELO	31:0	When the contents of *GIC_COREi_CompareLo* and *GIC_COREi_CompareHi* registers match the contents of *GIC_SH_CounterLo* and *GIC_SH_CounterHi*, the *CORE*i_Compare interrupt is triggered. This registered interrupt can only be deasserted by writing either the *GIC_COREi_CompareLo* or *GIC_COREi_CompareHi* registers.	R/W	0xFFFF_FFFF

8.6.4.5 Core-Local CompareHi Register (GCI_COREi_ComparHi — Offset 0x00A4)

For more information on the usage of this register, refer to Section 8.3.8.3, "Watchdog Timer Interrupts".

Figure 8.46 Local CompareHi Register Format

31 — 0
COMPAREHI

Table 8.56 Core-Local CompareHi Register

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COMPAREHI	31:0	See description for GIC_COREi_CompareLo. The width of this register matches the width of GIC_SH_COUNTER.	R/W	All instantiated bits = 0x1

8.6.4.6 Local Counter Offset Register (GCI_COREi_COFFSET — Offset 0x0200)

Indicates the counter offset. The value in the Hi andLo Counter registers must be offset by the value in the COFFSET field.

Figure 8.47 Local Counter Offset Register Format

31 8	7 0
Reserved	COFFSET

Table 8.57 Local Counter Offset Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00
COFFSET	7:0	Counter Offset. Guest read of GIC_SH_CounterHi/Lo must be offset by this value.	R/W	0x00

8.6.4.7 Core-Local DINT Group Participate Register (GIC_Vx_DINT_PART — Offset 0x3000)

When bit 0 of this register is set, the local core monitors the state of the DINT_Send_to_Group register in the Shared register set, as well as the EJ_DINT_IN pin for debug activity. Refer to Section 8.3.11, "Debug Interrupt Generation" for more information.

Figure 8.48 Core-Local EIC DINT Group Participate Register Format

31 1	0
R	DINT_GP

Table 8.58 Core-Local DINT Group Participate Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0

Table 8.58 Core-Local DINT Group Participate Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
DINT_GP	0	If this bit is set, the local core pays attention to the *DINT_Send_to_Group* register as well as the external *EJ_DINT_IN* signal pin. For this case, when the *Send_DINT* bit within the *DINT _Send_to_Group* register is asserted (or the external *EJ_DINT_IN* signal is asserted), the *EJ_DINT* or *EJ_DINT_1* signal of the local core is asserted. If this bit is clear, the local core is not affected by the *DINT_Send_to_Group* register nor the external *EJ_DINT_IN* pin signal.	R/W	0x1

See Chapter 14, “Multi-CPU Debug” on page 735 for more information about how this register is used.

8.6.4.8 Core-Local DebugBreak Group Register (GIC_Cx_BRK_GROUP — Offset 0x3080)

When the local core enters Debug Mode (denoted by the local *EJTAG_TAP.DebugM* bit being asserted), this register defines which other cores in the system subsequently also receives a Debug Interrupt. This allows multiple cores to be synchronized to a single software debugger by entering debug mode somewhat simultaneously.

Figure 8.49 Core-Local EIC DINT Group Participate Register Format

31 … 0
JOIN_DB

Table 8.59 Core-Local DebugBreak Group Register Bit Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
JOIN_DB	31:0	Each bit in this register represents a core in the system. If the bit is set, the corresponding core has its *EJ_DINT* or *EJ_DINT_1* signal asserted when the local core enters Debug Mode. If the bit is clear, the corresponding core is not affected when the core enters Debug Mode. The bit which represents the local core cannot be used to disable Debug Mode for the local core. For example, if the local core is represented by bit *i*, clearing bit *i* does NOT disable Debug Mode for the local core.	R/W	All zeros

See Chapter 14, “Multi-CPU Debug” on page 735 for more information about how this register is used.

8.7 GIC User-Mode Visible Section

The Shared, Core-local, and Core-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 64KB address space is allocated so that it may be mapped to user-mode virtual address space. Within this address space are aliases for GIC registers that are read so often that it makes sense to make them available to user-mode programs without requiring a system call. The aliases for these registers are read-only. Currently, the only 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
```

Table 8.60 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.

Chapter 9

I/O Memory Management Unit

The I/O Memory Management Unit (IOMMU) consists of a software-visible Translation Lookaside Buffer (TLB) with a memory-mapped register-based data and command interface to access the TLB and configure the IOMMU. The IOMMU serves exactly the same purpose as the CPU MMU in the context of I/O devices. In a standard SOC implementation, the CPU MMU is a requirement for virtual memory support. The IOMMU, on the other hand, is optional as devices can also be initialized by kernel-mode device drivers.

In some applications, such as those that employ a GPU (Graphic Processing Unit), the graphics application may operate in its own virtual address space, thus requiring an MMU to translate to physical addresses.

9.1 IOMMU Overview

The following subsections describe an overveiw of the IOMMU.

9.1.1 IOMMU and Virtualization

The P6600 Multiprocessing System implements the MIPS Virtualization Module, which requires a second level of translation due to the introduction of the additional level of privilege called Root. Guest addresses must also be translated through the Root's MMU to gain access to system physical memory.

In the P6600 Multiprocessing System, the hypervisor-managed IOMMU would be programmed with guest physical to root physical address mappings, typically with large pages to minimize the number of guest exits to root. Use of large pages allows root to program once for the entire guest address space. Subsequently, guest can program any device with root intervention only required to arbitrate access to the shared resource.

9.1.2 IOMMU Address Translation

The P6600 core supports a software-managed IOMMU that is programmed by the hypervisor. The hypervisor initializes the IOMMU with mappings for guest and/or root addresses. This configuration is expected to be sufficient for most applications, but more importantly is the minimum requirement for virtualization. It is assumed that the memory mapping requirements for a guest in this scenario are static - no capability exists to service a translation miss in the IOMMU and then restart a device request.

9.1.3 Overview of MIPS IOMMU Software Interface

The P6600 IOMMU provides the following feature set:

1. Native 32-bit addressing support (MIPS32 Module).

2. Hypervisor programmable CSRs (Control and Status Registers) to access the IOMMU TLB and Device Table.

3. Hypervisor programmable CSRs (Control and Status Registers) to configure the IOMMU.

4. Hypervisor privileged commands for IOMMU TLB and Device Table management.

5. Error monitoring, logging and interrupt signaling capability.

6. Optional support for a I/O Page Table to translate device originated guest physical or root virtual addresses to system physical memory.

9.1.4 IOMMU Programming Model

In the P6600 Multiprocessing System, the hypervisor actively manages the IOMMU as a shared resource since multiple active guests are supported and are context-switching in and out of guest mode, since the IOMMU TLB capacity may be limited for the multi-guest workload.

For the case where the IOMMU needs to be managed in a demand-based manner, the guest OS may execute a hypercall prior to device access in order to initialize the IOMMU with the appropriate mappings. In general, the hypercall need not be executed for every device access. The guest OS may request the hypervisor to program the IOMMU for a large range of guest physical addresses with large pages instead of prior to every device access.

9.2 IOMMU Virtual Memory Management

The Virtualization Module in the P6600 Multiprocessing System translates from Guest Virtual Address (GVA) to a Guest Physical Address (GPA) to a Root Physical Address (RPA) through a two step process. The RPA represents system physical memory. The GVA to GPA translation is done by the guest OS Page Table, while the GPA to RPA translation is done by the Page Table managed by hypervisor for the guest.

9.2.1 IOMMU Address Translation

A device may be programmed by Root or Guest privileged software. The latter is possible only if Hypervisor maps guest access through the CPU root MMU. The IOMMU explicitly distinguishes between root and guest device addresses through a combination of the Device Table and Segmentation Control Hypervisor programmed state.

The Segmentation Control registers are used to define the address spaces. The Device Table is used to provide root-level control for the various steps of address translation in the IOMMU.

9.2.1.1 IOMMU Guest Address Translation

The IOMMU assumes that a guest programmed device always sources a Guest Physical Address (GPA) to the IOMMU. A guest programmed device can never bypass the IOMMU TLB. Any device request looks up the Device Table to determine the GuestID associated with the device.

Subsequently, the IOMMU TLB must be accessed to obtain the corresponding RPA allocated to that guest for the GPA. Segmentation Control only applies to RVA and not GPA as guest addresses are always mapped through the IOMMU TLB.

9.2.1.2 IOMMU Root Address Translation

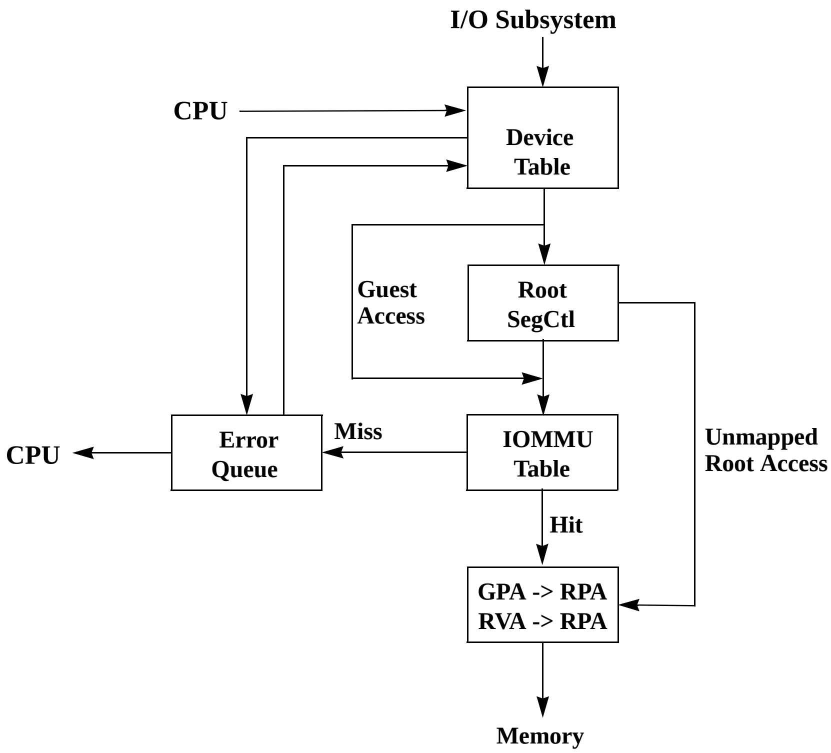
Root programmed device addresses require at most one step of address translation, to translate Root Virtual Address (RVA) to the Root Physical Address (RPA), though it is possible for hypervisor-programmed devices to bypass the IOMMU TLB using an RVA that decodes to an unmapped address segment of the Root Segmentation Control.

Any device request must be checked by a lookup of the Device Table. If the hypervisor has programmed the device with the RVA that decodes to a mapped segment of Root Segmentation Control, then an additional translation step through the IOMMU TLB is required to convert the RVA to RPA. If the Hypervisor has programmed the device with an RVA that decodes to an unmapped segment of Root Segmentation Control, then the IOMMU TLB is bypassed.

9.2.2 IOMMU Block-Level Address Translation Flow

Figure 3.1 shows the IOMMU block-level flow for address translation. The Device Table, Error Queue and IOMMU TLB. The registers used to control Segmentation are described in Section 9.3.6.6 through Section 9.3.6.8 below. Segmentation Control is further defined in the Enhanced Virtual Address (EVA) section in Chapter 3 of this manual.

Figure 9.1 IOMMU Address Translation



9.3 IOMMU Software Interface

The software interface supports commands for writing, reading, probing and invalidating the IOMMU TLB. In addition, the interface also allows for writing and reading the internal Device Table. The Device Table is required for all IOMMU configurations.

All TLB commands are mapped to a common Command register. The commands are encoded in the data of a store to the Command register. This format is followed as the TLB commands require the setup of multiple data registers prior to execution of a command. All Device Table accesses are loads (read) or stores (writes) executed with Device Table address are described in Section 9.3.1, "Device Table".

9.3.1 Device Table

The Hypervisor-managed Device Table supplies device-specific information required to enable IOMMU processing for a guest or root programmed DMA device.

In the IOMMU, a device request first causes a lookup of the Device Table which provides a GuestID. If the GuestID is non-zero, then the device address and GuestID are used to lookup the IOMMU TLB. If the GuestID field of the Device Table entry is zero, then Segmentation Control must be used to determine whether the root-programmed device address is mapped or unmapped. If mapped, then device address and GuestID are used to lookup the IOMMU TLB. If unmapped, then the IOMMU TLB is bypassed.

A load or store to the Device Table requires that the index be initialized before execution of the load or store to read or write the Device Table, respectively. In the case the width of the Device Table entry exceeds the width of the load or the store data, then a field in the Index will be used to index a 32-bit aligned word of the entry.

Table 9.1 Device Table Entry Format

Bit Position	Acronym	Field Name	Description	R/W	Reset State
31:15	R	Reserved	Reserved field. Write as zero, returns zero when read.	R	Undefined
14	P	Prefetch	If the P bit is set, a read request may cause hardware to prefetch data from the address stream into the cache, providing the request is allowed to allocate. The allocate permission is determined by the transaction itself, or the AR field. Otherwise, prefetching is disallowed.	R/W	Undefined
13	AW	Allocatate Write	A write transaction may allocate the data of the specified size to the cache.	R/W	Undefined
12	AR	Allocate Read	A read transaction may allocate the data of the specified size to the cache.	R/W	Undefined
11	SE	Sticky Error	The Sticky Error (SE) bit is set by hardware when ERT = 1 and an error for the device is encountered. When software writes to this bit, it is cleared by hardware.	R	Undefined
10	ERT	Error Tagging	If the ERT bit is set, any transaction related to a device results in an error, and all subsequent transactions must be serviced as if they had errors also. Error Tagging ends when the device table entry is rewritten to re-initialize the device. This may be a read-modify- write without change in content, i.e., a dummy write. The likely application of ERT is for devices that do not support error recovery.	R/W	Undefined
9	ERD	Error Reporting Disabled	Error reporting is disabled for the device. Software may set the bit to prevent reporting any further errors from the device, specifically within an interrupt handler that was invoked for an error from that device. This bit is encoded as follows: 0: Error reporting is enabled 1: Error reporting is disabled	R/W	Undefined
8	V	Valid	The entry is valid. Software must initialize and mark as valid or invalid all device table entries which are logically accessible before use. An invalid Device Table entry causes an error. A DeviceID out-of-range of the table also causes an error.	R/W	Undefined

Table 9.1 Device Table Entry Format *(continued)*

Bit Position	Acronym	Field Name	Description	R/W	Reset State
7:0	GID	GuestID	Guest associated with device. GuestID is used to lookup the TLB. The GuestID determines whether device access is guest or root owned.	R/W	Undefined

9.3.2 TLB Commands

The IOMMU supports the following TLB commands.

1. **Write**. Both Index and Random writes of the TLB are supported. *EntryLo0*, *EntryLo1*, *EntryHi* and *PageMask* data registers must be written before the write command itself is executed. The indexed write always supports $EntryHi_{EHINV}$ for invalidation on a per-entry basis. The Index must also be initialized prior to any indexed TLB write. In addition, software must initialize the *Wired* register to indicate the range of TLB entries that are considered wired and thus cannot be written to by a random TLB write. The *Wired* register is typically initialized once by software. Refer to Chapter 2 of this manual for more information on the *Wired* register.

   If the value of Index exceeds the number of TLB entries on a TLBWI, then the write is dropped, and an error may be logged providing error logging is enabled. The error encoding table is described below.

2. **Read**. *EntryLo0*, *EntryLo1*, *EntryHi* and *PageMask* data registers are loaded with the contents of a TLB entry at Index on execution of a read command. A read of the *EntryLo0*, *EntryLo1*, *EntryHi* and *PageMask* registers return the data to General Purpose Registers (GPRs).

3. **Probe**. This command determines whether there is an entry that matches the contents of the *EntryHi* and *PageMask* registers. These registers must be written before the probe command is executed. If there is a match, Index register is written with matching index. Otherwise the probe-fail bit is set in Index. Read of the Index subsequently returns data to the GPRs.

4. **Invalidate**. Execution of the invalidate command invalidates any entry that matches $EntryHi_{GuestID}$. The definition of the IOMMU TLB Invalidate command differs from the core TLB Invalidate in that the core command invalidates any entry for a guest process (ASID specific), while the IOMMU invalidates any entry for a guest.

5. **Invalidate Flush**. Execution of the invalidate flush command invalidates all entries in the IOMMU TLB. The definition of the IOMMU TLB Invalidate Flush command differs from the core TLB Invalidate Flush command in that the core invalidates any entry for a guest, while the IOMMU invalidate command invalidates all IOMMU TLB entries.

There is no command to invalidate by DeviceID. This is because the Hypervisor TLB mappings for a guest are globally applicable to the guest across all devices. If a device switches guest ownership, then it must refer to another guest's mappings. If a device retains guest ownership, but is reprogrammed for the guest, then the device guest physical address must also be reprogrammed, but it will refer to the same or new guest mappings.

9.3.3 Device Table Commands

The IOMMU supports the following Device Table commands.

1. **Write**. Execution of a store with Device Table address causes a write of the store's data to the Device Table at entry specified by the *Index* register. This register must be initialized before execution of the store. If the value of the Index field exceeds the number of Device Table entries on a write to the Device Table, then the write is dropped and an error may be logged providing error logging is enabled.

2. **Read**. Execution of a load with Device Table address causes a read of an entry indexed by the *Index* register. This register must be initialized before execution of the store. Contents of the Device Table entry are returned to the appropriate GPR specified by the load.

9.3.4 TLB Command Format

To execute a command, software must write the Command data register with a legal value defined in Table 9.2 below. Each of the TLB related commands has a counterpart of the same name in the baseline instruction set.

Figure 9.2 TLB Command Register Format

31 … 5	4 … 0
0	CMD

Table 9.2 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:5	Ignored on write; returns zero on read.	R	0
CMD	4:0	Command field. This field is encoded as follows: 00000: TLBWI — TLB Write Indexed 00001: TLBWR — TLB Write Random 00010: TLBR — TLB Read 00011: TLBP — TLB Probe 00100: TLBINV — TLB Invalidate 00101: TLBINVF — TLB Invalidate Flush 00110 - 11111: Reserved	R/W	Undefined

9.3.5 TLB Command to CP0 Register Relationship

When a TLB command is executed, the following IOMMU registers are updated as shown in Table 9.3.

Table 9.3 TLB Command to IOMMU Register Relationship

TLB Command	Preceeding IOMMU Register Write	Following IOMMU Register Read	Number of Data Accesses
TLBWI	EntryLo0, EntryLo1, EntryHi, PageMask, Index	None	5
TLBWR	EntryLo0, EntryLo1, EntryHi, PageMask	None	4
TLBR	Index	EntryLo0, EntryLo1, EntryHi, PageMask	3
TLBP	EntryHi	Index	2
TLBINV	EntryHi	None	1
TLBINVF	None	None	0

9.3.6 IOMMU Register Interface

As shown in the above table, the following IOMMU registers are used during the execution of TLB commands. The IOMMU registers are accessed using an offset that is relative to the IOCU base addresss located in the IOCU. The *IOCU Base Address* register stores the base address of the IOMMU registers when the device is in 32-bit address mode, and also stores the lower 32-bits of address when the device is in 40-bit address mode. When XPA is enabled, the *IOCU Base Address Upper* register is used to store the upper bits of the base address. Refer to Chapter 11, Section 11.4.4.4 for more information on the *IOCU Base Address* register, and Section 11.4.4.5 for more information on the *IOCU Base Address Upper* register.

Table 9.4 lists the control and status registers in the IOMMU. Note that these registers have the same names as their CP0 counterparts and perform basically the same functions, but they are contained within the IOMMU and are accessed using the offset addresses shown below.

Table 9.4 IOMMU Control and Status Registers

IOMMU Register Acronym	Full Name	Offset Address
ENTRYLO0	EntryLo0	0x000
ENTRYLO1	EntryLo1	0x008
ENTRYHI	EntryHi	0x010
INDEX	Index	0x018
WIRED	Wired	0x020
PAGEMASK	PageMask	0x028
TCFG	TLB Configuration	0x48
GCFG	Global Configuration	0x50
ESR0	Error Status Register 0	0x58
ESR1	Error Status Register 1	0x60
COMMAND	Command	0x68
DVT	Device Table	0x70

9.3.6.1 IOMMU EntryLo0 and EntryLo1 (Offsets 0x000, 0x004, 0x008, 0x00C)

The IOMMU *EntryLo0/EntryLo1* registers are similar in format to their CP0 counterparts with a few exceptions. The IOMMU supports the Read Inhibit (RI) function (bit 31), but does not support the Execute Inhibit (XI) function (bit 30). In the IOMMU *EntryLo0/EntryLo1* registers this bit is reserved.

These registers have been expanded to 64-bits in the P6600 Multiprocessing System. The full PFN is located at the following bits:

- PFN[35:12] stored in *EntryLo0/EntryLo1* bits 29:6
- PFN[39:36] stored in *EntryLo0/EntryLo1* bits 35:32

Figure 9.3 IOMMU EntryLo0 and EntryLo1 Register Format

63 ... 36	35 ... 32
U	PFNX

31	30	29 ... 6	5 ... 3	2	1	0
RI	0	PFN	C	D	V	G

Table 9.5 Field Descriptions for the IOMMU EntryLo0 and EntryLo1 Registers

Name	Bit(s)	Description	Read/ Write	Reset State
U	63:36	The upper 28 bits of the PFNX are not used. They cannot be written by software and will return 0 on reads.	R	Undefined
PFNX	35:32	Page Frame Number Extension. This field is concatenated with the PFN field to form the full page frame number corresponding to the physical address, thereby providing up to 40 bits of physical address. Note that the IOMMU does not support 1 KB pages.	R/W	Undefined
RI	31	Read Inhibit. If this bit is set in a TLB entry, any attempt to read data on the virtual page causes either a TLB Invalid or a TLBRI exception, even if the V (Valid) bit is set. The RI bit is writable only if the RIE bit of the *PageGrain* register is set. For more information, refer to the PageGrain register in Chapter 2 of this manual. If the RIE bit of the *PageGrain* register is not set, the RI bit of *Entry 0* and *Entry 1* are set to zero on any write to the register, regardless of the value written.	R/W	Undefined
0	30	Reserved. Must be written as zero. Reads are undefined.	R	0
PFN	29:6	The "Physical Frame Number" represents the physical frame number. Bits 35:12 of the physical address are stored in bits 29:6 of this field. Bits 39:36 of the physical address are stored in the PFNX field in bits 35:32 of this register. This value is appended to the upper bits of the PFN to create the extended address.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 9.6.	R/W	Undefined
D	2	The "Dirty" flag. 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 zero, 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.	R/W	Undefined
V	1	The "Valid" flag. Indicates that the TLB entry, and thus the virtual page mapping, are valid. If this bit is a set, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a *TLB Invalid* exception. This bit can be used to make just one of a pair of pages valid.	R/W	Undefined
G	0	The "Global" bit. On a TLB write, the logical AND of the G bits in both the *Entry 0* and *Entry 1* registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both *Entry 0* and *Entry 1* reflect the state of the TLB G bit.	R/W	Undefined

Table 9.6 Cache Coherency Attributes Encoding of the C Field

C[5:3]	Name	Cache Coherency Attribute
0	—	Reserved
1	—	Reserved
2	UC	Uncached, non-coherent
3	WB	Cacheable, non-coherent, write-back, write allocate
4	CWBE	Cacheable, coherent, write-back, write-allocate, read misses request Exclusive
5	CWB	Cacheable, coherent, write-back, write-allocate, read misses request Shared
6	—	Reserved
7	UCA	Uncached Accelerated, non-coherent

9.3.6.2 IOMMU EntryHi Register (Offsets 0x010 and 0x014)

Like the *EntryLo0/EntryLo1* registers, the IOMMU *EntryHi* register is also expanded to 64-bits. Bits 39:32 comprise the VPNU field and are used to store bits 39:32 of the virtual address. Bits 31:13 of the address are stored in bits 31:13 of the register.

Bit 10 of the *EntryHi* register (EHINV) is used to allow a TLB index write command to also invalidate an entry if the bit is set prior to the write.

Bits 7:0 of this register comprise the *GuestID* field. Each guest's entry must be made unique by tagging with a GuestID. Devices belonging to a guest (or root) can share a common entry. In other words, a guest's mappings are globalized across all devices owned by the guest.

Figure 9.4 EntryHi Register Format

63–40	39–32
0	VPNU

31–13	12–11	10	9–8	7–0
VPN2	0	EHINV	0	GID

Table 9.7 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:40	Fill bits. Write as zero. Ignored on reads.	R	0
VPNU	39:32	Upper 8 bits of the virtual page number in 40-bit address mode.	R/W	Undefined

Table 9.7 Field Descriptions for EntryHi Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
VPN2	31:13	*EntryHi*$_{VPN2}$ is the virtual address to be matched on a **TLBP**. This field consists of $VA_{31:13}$ of the virtual address (virtual page number / 2). It is also the virtual address to be written into the TLB on a **TLBWI** and **TLBWR,** and the destination of the virtual address on a **TLBR**. On a TLB-related exception, the VPN2 field is automatically set to the virtual address that was being translated when the exception occurred. This field is written by software before a **TLBP** or **TLBWI** and written by hardware in all other cases.	R/W	Undefined
0	12:11	Reserved. Write as zero. Ignored on reads.	R	0
EHINV	10	TLBWI invalidate enable. When this bit is set, the TLBWI instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. When this bit is set, the *PageMask* and *EntryLo0/EntryLo1* registers do not need to be valid. Only the *Index* register is required to be valid. This bit is ignored on a **TLBWR** instruction.	R/W	Undefined
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
GID	7:0	GuestID. Each guest's entry must be made unique by tagging it with a GuestID. Devices belonging to a guest (or root) can share a common entry. In other words, a guest's mappings are globalized across all devices owned by the guest.	R/W	Undefined

9.3.6.3 IOMMU Index Register (Offset 0x018)

The IOMMU *Index* register is required to index into the TLB on an indexed write or read. *Index* is also used to index the Device Table. This register is used as the TLB index when reading or writing the TLB with **TLBR/TLBWI/TLBINV/TLBINVF** respectively. It is also set by a TLB probe (**TLBP**) instruction to return the location of an address match in the TLB.

During execution of a **TLBR** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be read. Hardware then uses this information to perform the read operation.

During execution of a **TLBWI**, **TLBINV**, or **TLBINVF** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be written or invalidated. Hardware then uses this information to perform the respective write or invalidate operation.

Prior to executing a **TLBP** instruction, the VPN to be searched should have been written to the VPN2 field in the *EntryHi* register. During the **TLBP** instruction, hardware searches the TLB array for a match to the VPN stored in the *EntryHi* register. If a match is found, hardware writes the index into the *Index* field of this register.

The *P* bit of this register is set by hardware to indicate that a match was not found. If this bit is not set, software can then read the corresponding index from this register.

In the P6600 IOMMU, the VTLB is 64 dual entries, and the Index field is 6 bits wide. This is shown in Figure 9.5 below.

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the IOMMU *Index* register.

Figure 9.5 IOMMU Index Register Format

31	30 … 6	5 … 0
P	0	Index (TLB only)

Table 9.8 Field Descriptions for Index Register

Name	Bit(s)	Description	Read/ Write	Reset State
P	31	Probe Failure. This bit is automatically set when a **`TLBP`** search of the TLB fails to find a matching entry.	R	Undefined
0	30:6	Must be written as zero; returns zero on reads.	0	0
Index	5:0	An index into the TLB used for **`TLBR, TLBWI, TLBINV`** and **`TLBINVF`** instructions. This field is set by the **`TLBP`** instruction when it finds a matching entry. The maximum number in this field is 64 entries, or 0x3F.	R/W	Undefined

9.3.6.4 IOMMU Wired Register (Offset 0x020)

The *Wired* register in the IOMMU is a read/write register that specifies the boundary between the wired and random entries in the TLB. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

Note that wired entries in the TLB 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.

The operation of the processor is undefined if a value greater than or equal to the number of VTLB entries is written to the *Wired* register. *Wired* can be set to a non-zero value to prevent the random replacement of up to 63 TLB pages.

Figure 9.6 Wired Register Format

31 … 6	5 … 0
0	Wired

Table 9.9 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:6	Ignored on write; returns zero on read.	R	0

Table 9.9 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
Wired	5:0	Defines the number of wired dual entries in the TLB. A value of 0 in this field indicates that no VTLB entries are hard wired. This field is encoded as follows: 0x00: 0 TLB entries are hardwired 0x01: 1 TLB entry is hardwired 0x02: 2 TLB entries are hardwired ...... 0x3F: 63 TLB entries are hardwired	R/W	0

9.3.6.5 IOMMU PageMask Register (Offset 0x028)

The *PageMask* register in the IOMMU is required to define the page size of a TLB entry. *PageMask* is used by TLB write, read and probe commands.

It is recommended that the IOMMU support page sizes no smaller than 1 MB. In general, the Hypervisor uses large pages to map guest physical addresses. It is however left to the specific implementation of the IOMMU to determine what the smallest page size is. Software can determine which sizes are implemented by first writing the encoding for a page-size to *PageMask* and then reading back. If the read returns zeroes, then the page-size is not implemented.

Figure 9.7 PageMask Register Format

63 … 33	32 … 13	12 … 0
0	Mask	0

Table 9.10 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	63:33	Ignored on write; returns zero on read.	R	0
Mask	32:13	The mask field is a bit mask in which a logic “1” indicates that the corresponding bit of the virtual address should not participate in the TLB match. Note that only a restricted range of *PageMask* values are legal (i.e., with "1"s filling the $PageMask_{Mask}$ field from low bits upward, two at a time). Maximum page size is 4 GB. The legal values for this field are shown in Table 9.11 below.	R/W	Undefined
0	12:0	Ignored on write; returns zero on read.	R	0

Table 9.11 PageMask Register Values

PageMask Register Value	Size of Each Output Page
0x0000_0000_0000.6000	16 Kbytes
0x0000_0000_0001.E000	64 Kbytes
0x0000_0000_0007.E000	256 Kbytes
0x0000_0000_001F.E000	1 Mbyte

Table 9.11 PageMask Register Values

PageMask Register Value	Size of Each Output Page
0x0000_0000_007F.E000	4 Mbytes
0x0000_0000_01FF.E000	16 Mbytes
0x0000_0000_07FF.E000	64 Mbytes
0x0000_0000_1FFF.E000	256 Mbytes
0x0000_0000_7FFF.E000	1 Gbytes
0x0000_0001_FFFF.E000	4 Gbytes

9.3.6.6 IOMMU Segmentation Control 0 Register (Offset 0x030)

The *SegCtl0* register in the IOMMU works in conjunction with the *SegCtl1* and *SegCtl2* registers to allow for configuration of the I/O memory segmentation system when the P6600 core is in EVA mode. If the device is in the normal 64-bit mode, these registers are not used.

Figure 9.8 shows the format of the *SegCtl0* Register.

Figure 9.8 IOMMU SegCtl0 Register Format

31 25	24 23	22 20	19	18 16	15 9	8 7	6 4	3	2 0
CFG1_PA	0	CFG1_AM	CFG1_EU	CFG1_C	CFG 0_PA	0	CFG0_AM	CFG0_EU	CFG0_C

Table 9.12 IOMMU SegCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CFG1_PA	31:25	Physical address bits 31:29 for segment 1. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion.	R/W	Configuration Dependent
0	24:23	Reserved.	RO	0
CFG1_AM	22:20	Configuration 1 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG1_EU	19	Error condition behavior. Configuration segment 1 becomes unmapped and uncached when $Status_{ERL}$ = 1.	R/W	Configuration Dependent
CFG1_C	18:16	Cache coherency attribute for segment 1. The encoding of the CFG1_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 9.6.	R/W	Configuration Dependent
CFG0_PA	15:9	Physical address bits 31:29 for segment 0. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion.	R/W	Configuration Dependent
0	8:7	Reserved.	RO	0
CFG0_AM	6:4	Configuration 0 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG0_EU	3	Error condition behavior.	R/W	Configuration Dependent

Table 9.12 IOMMU SegCtl0 Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
CFG0_C	2:0	Cache coherency attribute for segment 0. The encoding of the CFG0_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 9.6.	R/W	Configuration Dependent

9.3.6.7 IOMMU Segmentation Control 1 Register (Offset 0x038)

The *SegCtl1* register works in conjunction with the *SegCtl0* and *SegCtl2* registers to allow for configuration of the memory segmentation system when the P6600 core is in 32-bit EVA mode. If the device is in the normal 64-bit mode, these registers are not used..

Segmentation Control allows address-specific behaviors defined by the Privileged Resource Architecture to be modified or disabled. Figure 9.9 shows the format of the *SegCtl1* Register.

Figure 9.9 IOMMU SegCtl1 Register Format

31–25	24–23	22–20	19	18–16	15–9	8–7	6–4	3	2–0
CFG3_PA	0	CFG3_AM	CFG3_EU	CFG3_C	CFG2_PA	0	CFG2_AM	CFG2_EU	CFG2_C

Table 9.13 IOMMU SegCtl1 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
CFG3_PA	31:25	Physical address bits 31:29 for segment 3. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion.	R/W	Configuration Dependent
0	24:23	Reserved. Must be written as zeros; returns zeros on reads.	RO	0
CFG3_AM	22:20	Configuration 3 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG3_EU	19	Error condition behavior.	R/W	Configuration Dependent
CFG3_C	18:16	Cache coherency attribute for segment 3, for use when unmapped.	R/W	Configuration Dependent
CFG2_PA	15:9	Physical address bits 31:29 for segment 2. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion.	R/W	Configuration Dependent
0	8:7	Reserved. Must be written as zeros; returns zeros on reads.	RO	0
CFG2_AM	6:4	Configuration 2 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG2_EU	3	Error condition behavior.	R/W	Configuration Dependent
CFG2_C	2:0	Cache coherency attribute for segment 2, for use when unmapped.	R/W	Configuration Dependent

9.3.6.8 IOMMU Segmentation Control 2 Register (Offset 0x040)

The *SegCtl2* register works in conjunction with the *SegCtl0* and *SegCtl1* registers to allow for configuration of the memory segmentation system when the P6600 core is in 32-bit EVA mode. If the device is in the normal 64-bit mode, these registers are not used.

Segmentation Control allows address-specific behaviors defined by the Privileged Resource Architecture to be modified or disabled. Figure 9.10 shows the format of the *SegCtl2* Register.

Figure 9.10 IOMMU SegCtl2 Register Format

31 25	24 23	22 20	19	18 16	15 9	8 7	6 4	3	2 0
CFG5_PA	0	CFG5_AM	CFG5_EU	CFG5_C	CFG4_PA	0	CFG4_AM	CFG4_EU	CFG4_C

Table 9.14 IOMMU SegCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CFG5_PA	31:25	Physical address bits 31:29 for segment 5. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion. Note that for this field, bit 25 is ignored since CFG5 is mapped to a 1 GByte boundary.	R/W	Configuration Dependent
0	24:23	Reserved.	RO	
CFG5_AM	22:20	Configuration 5 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG5_EU	19	Error condition behavior.	R/W	Configuration Dependent
CFG5_C	18:16	Cache coherency attribute for segment 5. The encoding of the CFG5_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 9.6.	R/W	Configuration Dependent
CFG4_PA	15:9	Physical address bits 31:29 for segment 4. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion. Note that for this field, bit 9 is ignored since CFG4 is mapped to a 1 GByte boundary.	R/W	Configuration Dependent
0	8:7	Reserved.	RO	
CFG4_AM	6:4	Configuration 4 access control mode. See Table 9.15 for encoding.	R/W	Configuration Dependent
CFG4_EU	3	Error condition behavior.	R/W	Configuration Dependent
CFG4_C	2:0	Cache coherency attribute for segment 4. The encoding of the CFG4_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 9.6.	R/W	Configuration Dependent

Table 9.15 describes the access control modes specifiable in the CFG_{AM} fields.

Table 9.15 Segment Configuration Access Control Modes

Mode		Action when referenced from Operating Mode			Description
		User mode	Supervisor mode	Kernel mode	
UK	000	Address Error	Address Error	Unmapped	Kernel-only unmapped region e.g. kseg0, kseg1
MK	001	Address Error	Address Error	Mapped	Kernel-only mapped region e.g. kseg3
MSK	010	Address Error	Mapped	Mapped	Supervisor and kernel mapped region e.g. ksseg, sseg
MUSK	011	Mapped	Mapped	Mapped	User, supervisor and kernel mapped region e.g. useg, kuseg, suseg
MUSUK	100	Mapped	Mapped	Unmapped	Used to implement a fully-mapped flat address space in user and supervisor modes, with unmapped regions which appear in kernel mode.
USK	101	Address Error	Unmapped	Unmapped	Supervisor and kernel unmapped region e.g. sseg in a fixed mapping TLB.
-	110	Undefined	Undefined	Undefined	Reserved
UUSK	111	Unmapped	Unmapped	Unmapped	Unrestricted unmapped region

9.3.6.9 IOMMU TLB Configuration Register (Offset 0x048)

The TLB Configuration register (TCFG) determines the number of entries in the VTLB.

Figure 9.11 TLB Configuration Register Format

31 … 10	9 … 1	0
0	VSIZE	TT

Table 9.16 Field Descriptions for TLB Configuration Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:10	Ignored on write; returns zero on read.	R	0
VSIZE	9:1	In the IOMMU, the TLB size is fixed at 64 entries	R/W	0x3F
TT	0	Indicates the TLB type supported. In the P6600 IOMMU, this bit is always 0 to indicate that the VTLB is supported.	R/W	0

9.3.6.10 IOMMU Global Configuration Register (Offset 0x050)

The IOMMU Global Configuration register (GCFG) is used to configure the global functionality of the IOMMU.

Figure 9.12 IOMMU Global Configuration Register Format

31 ... 22	21 ... 6	6	5	4	3	2	1	0
DVNUM	0	IDLE	VZEN	ELPA	LPA	IE	PW	TT

Table 9.17 Field Descriptions for IOMMU Global Configuration Register

Name	Bit(s)	Description	Read/ Write	Reset State
DVNUM	31:22	The DVNUM field indicates the number of entries in the Device Table. In the IOMMU, the maximum number of Device Table entries is 64.	R/W	0x3F
0	21:6	Ignored on write; returns zero on read.	R	0
IDLE	6	This bit is set hardware to indicate that the IOMMU has no traffic and no outstanding response.	R	1
VZEN	5	Virtualization enabled. This bit is set to indicate that IOMMU virtualization is enabled. This bit is independent of the virtualization enable bit in the CP0 registers used to enable or disable virtualization in the core. 0: Virtualization disabled 1: Virtualization enabled	R/W	0
ELPA	4	Enable Large Physical Address. Setting this bit enables support for large physical addresses and is encoded as follows: 0: Large physical address support is disabled. 1: Large physical address support is enabled. If this bit is set, the following changes occur: The PFNX field of the *EntryLo0/EntryLo1* registers is writeable and concatenated with the PFN field to form the full page frame number.	R/W	0
LPA	3	Large Physical Address support implemented. Hardware sets this bit to indicate that large physical address support is implemented. This bit is encoded as follows: 0: Large physical address support is not implemented. 1: Large physical address support is implemented. If this bit is set, the PFNX field of the *EntryLo0/EntryLo1* registers are expanded to 64-bits.	R	1
IE	2	Interrupt Enable. When this bit is set, errors cause interrupts. Otherwise interrupts are disabled. This enable applies to all errors, regardless of device origin. The IOMMU continues to capture errors even if interrupts are disabled.	R/W	0
PW	1	PageWalker implemented. This bit is always 0 to indicate that hardware page walker support is not implemented in the IOMMU.	R	0
EN	0	IOMMU enable. This bit enables the IOMMU and is encoded as follows: 0: IOMMU is disabled and all device requests bypass the IOMMU. 1: IOMMU is enabled and all device addresses are translated by the IOMMU.	R/W	0

9.3.6.11 IOMMU Error Status Register 0 (Offset 0x050)

The IOMMU Error Status registers provide information about the oldest error detected in the IOMMU for which an interrupt has been signaled to the CPU. Error status is reported through two registers, Error Status 0 (ESR0) and Error Status 1 (ESR1).

Error Status 0 stores the oldest error. Software cannot read any other errors in the queue. The number of entries in the error queue is implementation defined but should not exceed 16.

On detection of error, the IOMMU returns an error response to the I/O subsystem, providing the error is due to a read or non-posted write. A posted write will never deliver an error response to the I/O subsystem. While ESR0 remains valid, the read pointer of the error queue cannot be advanced until the software handler clears the valid bit (V) of ESR0.

In the interrupt handler, software may disable error reporting for the device by writing to the device's Device Table entry. The action of clearing the Device entry should clear any errors from the Error Queue (except for the head) belonging to that device. Prior to clearing ESR0 valid, the handler should reprogram the device, and the IOMMU specifically for the device. Once the valid bit in ESR0 is clear, software may restart the device.

Table 9.18 IOMMU Status Register 0 Format

31 24	23 16	15 12	11	10 9	8 7	6	5 2	1	0
DID	SIZE	ECNT	ERM	0	ATYPE	0	ETYPE	OV	V

Table 9.19 IOMMU Status Register 0 Descriptions

Bit Position	Field	Description	R/W	Reset State
31:24	DID	Device ID for which error was reported.	R	Undefined
23:16	SIZE	Encoded size of DMA request in bytes. Where 8-bits is not sufficient, the value must be saturated to 28-1.	R	Undefined
15:12	ECNT	Number of errors in error queue, excluding the error at the head of the queue. The total number of errors is thus ECNT+1 if valid in ESR0 is 1, otherwise it is 0.	R	Undefined
11	ERM	Hardware sets this bit to indicate that the entry has been used to merge subsequent errors for a device. The criteria for merging is a match on Device-ID and Error-Type.	R	Undefined
10:9	0	Reserved. Written as zero. Reads are undefined.	R	0
8:7	ATYPE	Type of Device Address. Only relevant to TLB errors. Refer to Table 4.9. The Root Physical Address (RPA) is never logged as it is unmapped and thus would not cause TLB related errors.	R	Undefined
6	0	Reserved. Written as zero. Reads are undefined.	R	0
5:2	ETYPE	Type of Error related to the Device Request. Refer to Table 4.10. Device requests that bypass IOMMU TLB do not generate TLB errors.	R	Undefined
1	OV	Error output queue has overflowed. Errors are not written to queue when overflow bit is set. Overflow bit is cleared when the V bit of this register is cleared.	R	0
0	V	This bit is set by hardware to indicate that an error has been reported to core. The Error handler writes a 0 to clear this bit once it processes the source of the error. This allows hardware to read the next error from the error queue and signal the CPU with an interrupt. This bit is only set by hardware and cleared by software. If software attempts to set this bit, the write is ignored.	R/W0	0

Table 9.20 shows the encoding of the ATYPE field (bits 8:7) described above

Table 9.20 ESR0 Register ATYPE Field Encoding

Encoding	Description
00	Device address is a GPA. This is the case if a GuestID read from Device Table is non-zero for DeviceID of the request.
01	Device address is a mapped RVA. This is the case if GuestID read from Device Table is zero but the decode of SegCtl indicates the access is mapped.
10 - 11	Reserved

Table 9.21 shows the encoding of the ETYPE field (bits 5:2) described above. All encodings not shown are reserved.

Table 9.21 ESR0 Register ATYPE Field Encoding

Encoding	Description
IOMMU TLB Errors	
0000	Refill error. There is no TLB entry that matches the device request.
0001	Read-Inhibit error. Device makes read request but TLB entry RI = 1.
0010	Dirty error. Device makes write request but TLB entry D = 0.
0011	Invalid error. Device address matches TLB entry but TLB entry V = 0.
0100	TLB Page-crossing error. This error is logged if an access crosses a page boundary i.e., the access is not contained in a single page.
0101 - 0111	Reserved.
Device Table Access Errors	
1000	Device Table entry is invalid.
1001	Device request's DeviceID is out-of-range of Device Table.
1100 - 1011	Reserved.
Programming Errors	
1100	Index exceeds GCFG[DVNUM] on store to memory-mapped DVT.
1101	Index exceeds number of TLB entries on write to TLB.
1110 - 1111	Reserved.

9.3.6.12 IOMMU Error Status Register 1 (Offset 0x060)

The IOMMU Error Status registers provide information about the oldest error detected in the IOMMU for which an interrupt has been signaled to the CPU. Error status is reported through two registers, Error Status 0 (ESR0) and Error Status 1 (ESR1). Error Status 1 stores the address corresponding to the device which caused the error.

Table 9.22 IOMMU Status Register 1 Format

63–40	39–32	31–0
0	EADDRX	EADDR

Table 9.23 IOMMU Status Register 1 Descriptions

Bit Position	Field	Description	R/W	Reset State
63:40	0	Reserved. Written as zero. Reads are undefined.	R	0
39:32	EADDRX	An extension of EADDR to support up to 40-bits of physical address. The upper 32-bits of this register are used only when GCFG.LPA = 1, which is always the case in the P6600. If GCFG.ELPA = 0, indicating that Large Physical Address support is disabled, then this field is read as 0.	R	Undefined
31:0	EADDR	32-bit device address related to error. EADDR may be a GPA or RVA. The type of address is determined from the ESR0.ATYPE field. Unused bits must be zeroed prior to write.	R	Undefined

9.3.6.13 Command Register (Offset 0x068)

The Command register is described in Section 9.3.4, "TLB Command Format". A list of associated TLB commands if provided in Section 9.3.2, "TLB Commands".

9.3.6.14 Device Table Register (Offset 0x070)

The Command register is described in Section 9.3.1, "Device Table".

Chapter 10

Virtualization

The Virtualization Module defines a set of new instructions, registers, and machine states to the P6600 core to mange the efficient implementation of virtualized systems. The Virtualization Module is designed to enable full virtualization of operating systems. The Virtualization Module allows for the execution of guest Operating Systems in a fully virtualized environment.

10.1 Elements of Virtualization

The Virtualization Module defines the following elements which are related to virtualization:

- Guest Operating Mode
- Partial CP0 register set (or context) for Guest Mode use
- Registers for Guest Mode control
- Guest interrupt system
- Two-level address translation
- Detection of Virtualization Features

The Virtualization Module provides a separate Coprocessor 0 register set (or context) for guest mode operation, which is physically separate from, and a subset of the Root Coprocessor 0 context. The presence of the virtualization module is indicated by the CP0 Config3.VZ bit. Refer to Chapter 2 of this manual for more information.

10.2 Introduction to the Hypervisor

Virtualization is enabled by software. The key element is a control program known as a Virtual Machine Monitor (VMM) or ‘Hypervisor’. The Hypervisor is in full control of machine resources at all times. When an operating system (OS) kernel is run within a virtual machine (VM), it becomes a ‘guest’ of the Hypervisor. All operations performed by a guest must be explicitly permitted by the Hypervisor. To ensure that it remains in control, the Hypervisor always runs at a higher level of privilege than a guest operating system kernel. The hypervisor is responsible for managing access to sensitive resources, maintaining the expected behavior for each VM, and sharing resources between multiple VMs.

In a traditional operating system, the kernel (or ‘supervisor’) typically runs at a higher level of privilege than user applications. The kernel provides a protected virtual-memory environment for each user application, inter-process communications, and I/O device sharing. The hypervisor performs the same basic functions in a virtualized system - except that the Hypervisor’s clients are full operating systems rather than user applications.

The virtual machine execution environment created and managed by the Hypervisor consists of the full Instruction Set Architecture, including all Privileged Resource Architecture facilities, plus any device-specific or board-specific peripherals and associated registers. It appears to each guest operating system as if it is running on a real machine with full and exclusive control.

The Virtualization Module enables full virtualization, and is intended to allow VM scheduling to take place while meeting real-time requirements, and to minimize costs of context switching between VMs.

In virtualization, the guest operating system operates in unprivileged mode. All privileged operations attempted by the guest will trap back to the Hypervisor, which executes in the privileged mode. The Hypervisor emulates all guest privileged operations, keeps track of the guest view of privileged state, and ensures that the system behaves as expected by the guest. Full address translation allows an unmodified guest kernel to execute from its original location in memory, and allows the hypervisor to manage address translation to match the expectations of the guest kernel.

A Segmentation Control system is available for use by the Virtualization Module. This is a programmable memory segmentation system defined to support remapping (and therefore virtualization) of the existing fixed segment memory model.

10.3 Root and Guest Operating Modes

The virtualization module contains a operating modes for one **Root** and multiple **Guests**. The non-guest operating mode is known as **root mode**. The pre-existing kernel, user and supervisor operating modes can be referred to as **root-kernel**, **root-user** and **root-supervisor** respectively, to distinguish them from their guest-mode equivalents.

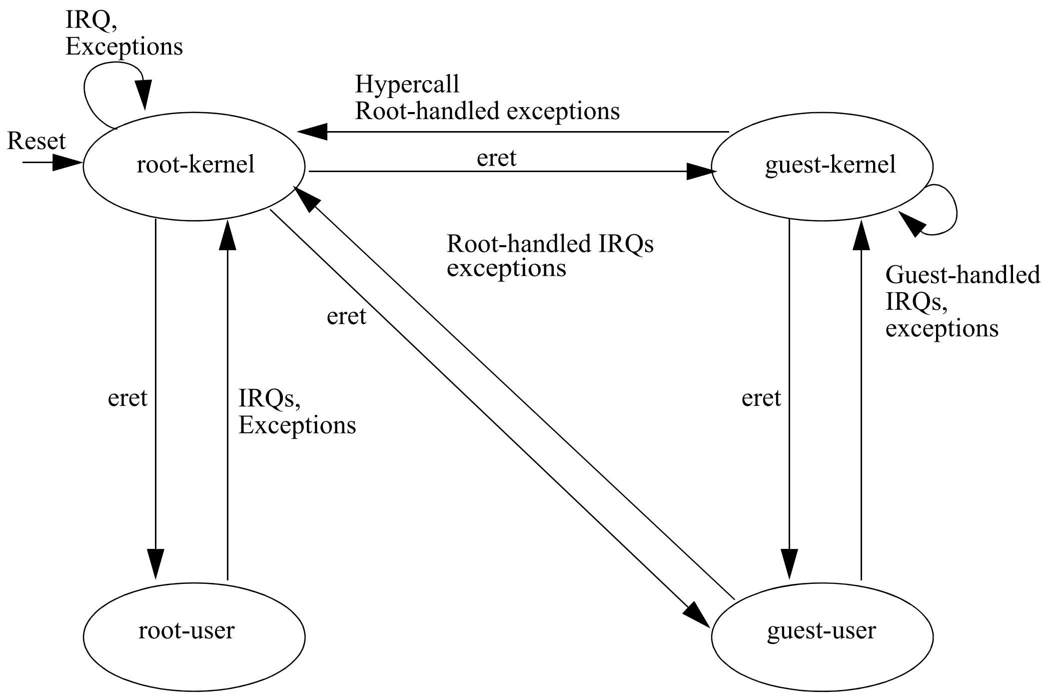
Guest mode consists of new operating modes guest-kernel, guest-user and guest-supervisor modes. The guest mode allows the separation between kernel, user and supervisor modes to be retained for a guest operating system running within a virtual machine. The guest-kernel mode can handle interrupts and exceptions, and manage virtual memory for guest-user mode processes.

The separation between root mode and the limited-privilege guest mode allows root mode software to be in full control of the machine at all times even when a guest is running. Backward compatibility is retained for existing software running in root mode.

The *GuestCtl0* register contains the GM (Guest Mode) bit. This bit is used along with root-mode exception and error status bits ($Status_{EXL}$, $Status_{ERL}$) and the Debug Mode bit ($Debug_{DM}$) to determine whether the processor is operating in guest mode or root mode.

Figure 10.1 shows the state transitions between operating modes.

Figure 10.1 State Transitions Between Operating Modes



10.3.1 Enabling Guest Mode Translations

The Virtualization Module in the P6600 core provides a separate CP0 register set and MMU for guest-mode execution. In guest mode when guest segmentation and translation are enabled ($GuestCtl0_{AT}$ = 3), two levels of address translation are performed as described above.

10.3.2 MMU Considerations

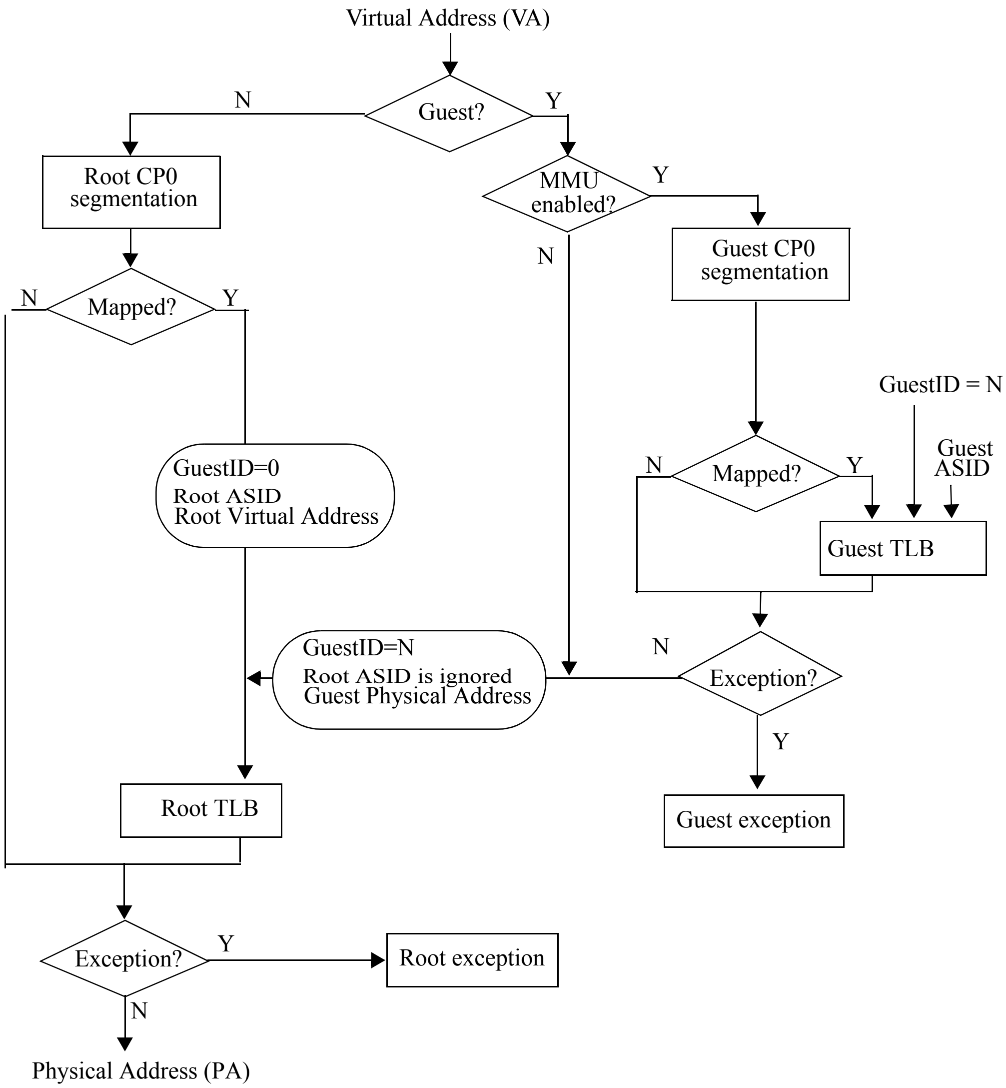
For the TLB-based guest MMU, MIPS recommends that the number of entries be equal to the number of entries in the root-context TLB used for Guest mappings. The page sizes used in the root-mode TLB must be carefully considered to allow sufficient control for root-mode software, while maximizing the number of guest-mode TLB entries which are mapped through each root-mode TLB entry. Larger root TLB pages will likely result in better performance.

Both the guest and root MMU’s can be active at the same time. MIPS recommends that the Root TLB maintain an adequate amount of reserved TLB entries for its own use to avoid cascading TLB evictions (thrashing).

Note that the TLBP/TLBGP differentiate between guest and root entries respectively. Software should use the results of TLBP/TLBGP to selectively read entries. The root TLBR instruction is used exclusively for logical Root TLB reads, while root TLBGR is used exclusively for logical Guest TLB reads.

Figure 10.2 shows the outline of address translation in the Virtualization Module.

Figure 10.2 Outline of Address Translation



Guest mode segmentation controls and the guest mode MMU have no effect on the root mode address space.

10.3.3 Guest ID

The 'GuestID' field (*GuestCtl1*$_{ID}$ or *GuestCtl1*$_{RID}$) represents a unique identifier for Root and all Guest Virtual Address spaces. Each Guest's address space is identified by a unique non-zero GuestID. The GuestID value zero is reserved for Root address space. The *GuestCtl1* CP0 register is unique in the Root register space and inaccessible in guest mode. GuestID is an optimization, designed to minimize TLB invalidation overhead on a virtual machine context switch and simplify Root access to Guest TLB entries.

The P6600 core implements a 16-bit Guest ID. This allows the Root TLB to distinguish between Root and Guest Entries, and flush either set of mappings in entirety with the TLBINVF instruction.

10.3.4 Address Translation Pseudocode

The pseudocode below describes the complete address translation process for the P6600 Virtualization Module. Segmentation, TLB lookups, hardware TLB refill and second-level address translation are invoked below. The process is described in top-down order - subsequent sections describe the subroutines called.

```
/* Inputs
 * vAddr  - Virtual Address
 * IorD   - Access type - INSTRUCTION or DATA
 * LorS   - Access type - LOAD or STORE
 * pLevel - Privilege level - USER, SUPER, KERNEL
 *
 * Outputs
 * pAddr  - physical address
 * CCA    - cache attribute (valid when mapped)
 *
 * Exceptions: See called functions
 * Called from guest or root context.
 */
subroutine AddressTranslation(vAddr, IorD, LorS, pLevel)

   // Initialization.
   // GuestID is only applicable if GuestCtl0_RAD=0. Otherwise GuestID
   // is ignored (not applicable) in process of address translation.
   GuestID ← ignored

   if (IsGuestMode()) then
      // This is a Guest Address translation
      // step 1: Guest Virtual -> Guest Physical Address translation
      if (GuestCtl0_RAD=0)
            GuestID ← GuestCtl1_ID
      endif
      (mapped, addr, CCA) ← AddressDecode(vAddr, pLevel)
      if (Config_MT=1 or Config_MT=4) then // TLB type MMU
         if (mapped) then
            asid ← Guest.EntryHi_ASID
            (addr, CCA) ← Guest.TLBLookup(asid, GuestID, addr, IorD, LorS)
         endif
      endif
      if (exception)
         Guest Exception
         // TLB exceptions may include Refill, Invalid, Execute-Inhibit for
         // Instruction, Refill, Invalid, Modified, Read-Inhibit for Data.
         // Guest segment map related exceptions may include Address Error
      endif

      // step 2: Guest Physical -> Root Physical Address translation
      // if GuestCtl0_RAD=0, then guest entry ASID is global in Root TLB.
      // H/W must set G=1 for guest entry for TLBWI and TLBWR.
      asid ← Root.EntryHi_ASID
      pAddr ← Root.TLBLookup(asid, GuestID, addr, IorD, LorS)
      if (exception)
         Root Exception
         // This is a Root exception initiated in guest context
         // This includes all TLB exceptions.
         // Segment map Address Error exception not included, as guest does not
         // lookup root segment map.
```

```
        endif

    else
        // This is a Root Address translation
        // Root Virtual -> Root Physical Address translation
        // If GuestCtl0_DRG=1,GuestCtl1_RID is non-zero,Root.Status_EXL,ERL=0,
        // and Debug_DM=0, then all root kernel data accesses are mapped and root
        // SegCtl is ignored.H/W must set G=1 as if the access were for guest.
        drg_valid ← (GuestCtl0_DRG=1 and Root.Status_KSU=00 and Root.Status_EXL=0 and
        Root.Status_ERL=0 and Debug_DM=0 and GuestCtl1_RID!=0 and !Instruction)
        if (drg_valid) then
            mapped ← 1
            addr ← vAddr
        else
            (mapped, addr, CCA) ← AddressDecode(vAddr, pLevel)
        endif
        if (!mapped) then
            pAddr ← addr
        else   if (GuestCtl0_RAD=0)
                    if (Instruction or (!drg_valid))
                            GuestID ← 0
                    else
                            GuestID ← GuestCtl1_RID
                    endif
                endif
            asid ← Root.EntryHi_ASID
            (pAddr, CCA) ← Root.TLBLookup(asid, GuestID, addr, IorD, LorS)
        endif
    endif
    if (exception)
        Root Exception
        // Includes all TLB and Segment related exceptions in Root context.
        // If drg_valid, and access is not by root-kernel,then an Address Error
        // exception is caused.
    endif

    return (pAddr,CCA)
end

subroutine AddressDecode(vAddr, pLevel) :
    # Determine whether address is mapped
    # - if unmapped, obtain physical address and cache attribute
    if (Config3_SC) then
        // optional Segmentation Control based address decode
        (mapped, addr, CCA) ← SegmentLookup(vAddr, pLevel)
    else
        (mapped, addr, CCA) ← LegacyDecode (pLevel)
    endif
    return (mapped, addr, CCA)
endsub
```

10.3.5 Address Translation for the Root and Guest Processes

In virtualization, there are two basic elements, a *Root* process, which stores the kernel or user software, and multiple *Guest* processes, which typically consists of user-level applications.

In the *Root* process, there is one level of address translation:

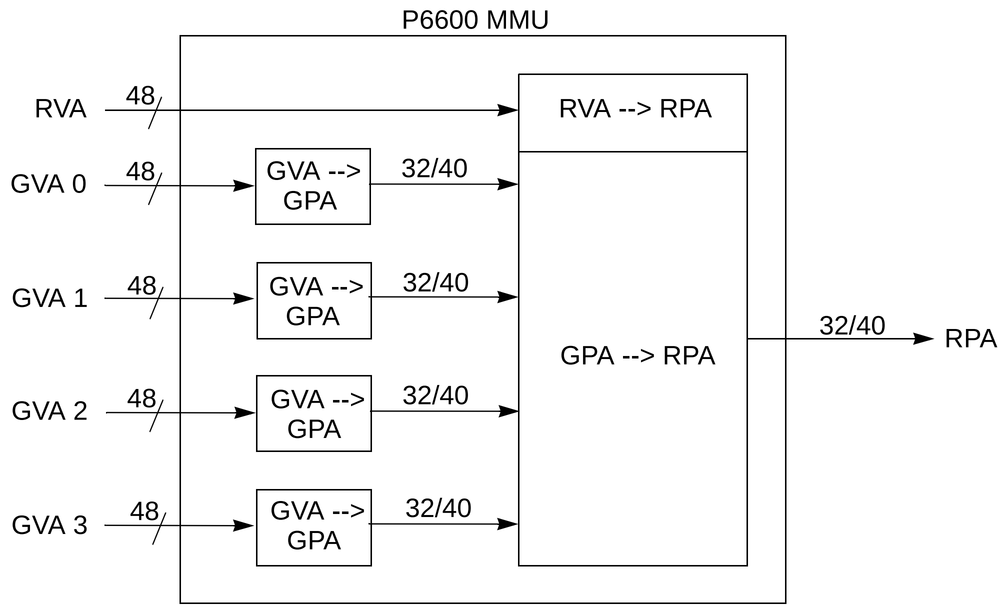
- 48-bit Root virtual address (RVA) --> 32- or 40-bit Root physical address (RPA)

In the *Guest* processes, there are two levels of address translation that occur in the following order:

- 48-bit Guest virtual address (GVA) --> 32- or 40-bit Guest physical address (GPA)
- 32- or 40-bit Guest physical address (GPA) --> 32- or 40-bit Root physical address (RPA)

Figure 10.3 shows an overview of the multi-level PA translation process.

Figure 10.3 Overview of Address Translation Process



10.3.6 Enabling Guest Mode Translations

The Virtualization Module in the P6600 core provides a separate CP0 register set and MMU for guest-mode execution. In guest mode when guest segmentation and translation are enabled ($GuestCtl0_{AT}$ = 3), two levels of address translation are performed as described above.

10.4 Software Detection of Virtualization

Software can determine if the Virtualization Module is implemented by checking the state of the VZ bit in the *Config3* CP0 register. If Virtualization is supported ($Config3_{VZ}$ = 1), and GuestID is supported, then explicit invalid TLB entry support (EHINV) is required in order for a Guest to be able to detect invalid entries in the Guest TLB.

Figure 10.4 Config3 Register Format

31	30	29	28	27	26	25	24	23	22 … 16
								VZ	0

Table 10.1 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
VZ	23	Virtualization Module implemented. This bit indicates whether the Virtualization Module is implemented. This bit is always 1 for the P6600 core. 0: Virtualization module not implemented 1: Virtualization module is implemented	R	1

10.5 CP0 Structure in Root and Guest Mode

In the P6600 core, Coprocessor 0 (CP0) contains system control registers and can be accessed only by privileged instructions. The presence of virtualization in the P6600 core means that a subset of the Coprocessor 0 register set are physically replicated for use by the Guest Operating System.

During guest mode execution, both the guest Coprocessor 0 and the root Coprocessor 0 are active. The presence of two simultaneously active Coprocessor 0 contexts is fundamental to the operation of the Virtualization Module. The presence of these two sets of Coprocessor 0 (CP0) registers allows for an immediate switch between guest and root modes without requiring a context switch to/from memory. Simultaneously accesses to the guest and root Coprocessor 0 registers allows guest-kernel privileged code accesses to execute with the minimum hypervisor intervention, and ensures that key root-mode machine systems such as timekeeping, address translation and external interrupt handling continue to operate without major changes during guest execution.

Table 10.2 describes the how the various CP0 register fields are used to enter or exit an operating mode.

Table 10.2 Guest, Root and Debug Modes

<table>
<tr><th colspan="5">Root</th><th colspan="3">Guest</th><th rowspan="2">Mode</th></tr>
<tr><th>Debug$_{DM}$</th><th>Status$_{ERL}$</th><th>Status$_{EXL}$</th><th>Status$_{KSU}$</th><th>GuestCtl0$_{GM}$</th><th>Status$_{ERL}$</th><th>Status$_{EXL}$</th><th>Status$_{KSU}$</th></tr>
<tr><td>1</td><td colspan="7">Don’t care</td><td>Debug</td></tr>
<tr><td rowspan="11">0</td><td>1</td><td colspan="6">Don’t care</td><td rowspan="3">Root-Kernel</td></tr>
<tr><td rowspan="10">0</td><td>1</td><td colspan="5">Don’t care</td></tr>
<tr><td rowspan="9">0</td><td>00</td><td rowspan="3">0</td><td colspan="3" rowspan="3">Don’t care</td></tr>
<tr><td>01</td><td>Root-Supervisor</td></tr>
<tr><td>10</td><td>Root-User</td></tr>
<tr><td rowspan="6">Don’t care</td><td rowspan="6">1</td><td>1</td><td colspan="2">Don’t care</td><td rowspan="3">Guest-Kernel</td></tr>
<tr><td rowspan="4">0</td><td>1</td><td>Don’t care</td></tr>
<tr><td rowspan="3">0</td><td>00</td></tr>
<tr><td>01</td><td>Guest-Supervisor</td></tr>
<tr><td>10</td><td>Guest-User</td></tr>
<tr><td colspan="2">Don’t care</td><td>11</td><td>UNPREDICTABLE</td></tr>
</table>

10.5.1 Root Mode Operation

Root mode operation uses one set of Coprocessor 0 registers and Guest mode operation the other. The software visible state is the contents of these registers and any state which is accessed via these registers, such as TLB entries and Segmentation Control configurations.

For a Hypervisor to save, restore or switch context from one guest to another, it is the entire software visible state which must be saved and restored, not solely the replicated registers themselves, but also the physical resources which are shared between Root and Guest, such as the GPRs, FPRs and Hi/Lo registers.

The following subroutine can be used to test whether processor is in root-mode.

```
subroutine IsRootMode() :
    if (
        (GuestCtl0GM=0) or
        ((GuestCtl0GM=1) and not ((Root.DebugDM=0) and
        (Root.StatusERL=0) and (Root.StatusEXL=0))
        ) then
        return(true)
    else
        return(false)
    endif
endsub
```

10.5.2 Guest Mode Operation

In guest mode, all guest operations are first tested against the guest CP0 context, and then against the root CP0 context. An ‘operation’ is any process which can trigger an exception. This includes address translation, instruction fetches, memory accesses for data, instruction validity checks, coprocessor accesses and breakpoints.

Guest mode software has no access to the root Coprocessor 0. Root mode software can access the guest Coprocessor 0, and if required can emulate guest-mode accesses to disabled or unimplemented features within guest Coprocessor 0. The guest Coprocessor 0 is partially populated - only a subset of the complete root Coprocessor 0 is implemented.

The recommended method of entering Guest mode is by executing an ERET instruction when $Root.GuestCtl0_{GM}$=1, $Root.Status_{EXL}$=1, $Root.Status_{ERL}$=0 and $Root.Debug_{DM}$=0.

Guest mode operation is determined as follows. This subroutine can be used to test whether processor is in guest-mode.

```
subroutine IsGuestMode() :
    if (GuestCtl0GM=1) and (Root.DebugDM=0) and
        (Root.StatusERL=0) and (Root.StatusEXL=0) then
        return(true)
    else
        return(false)
    endif
endsub
```

10.5.3 Debug Mode

For processors that implement EJTAG, the processor is operating in debug privileged execution mode (Debug Mode) when $Root.Debug_{DM}$=1. If the processor is running in Debug Mode, it has full access to all resources that are available to Root Kernel Mode operation.

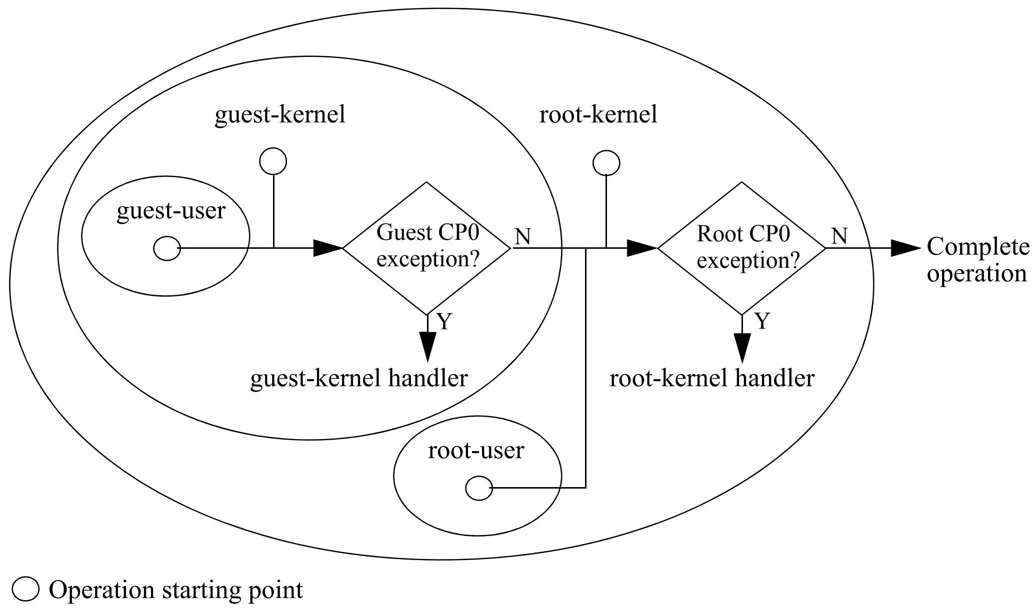
Debug Mode, Root Mode and Guest Mode are mutually exclusive. At any given time, the processor can only be in one of the three modes. Note that Debug mode operates in the Root context, while Guest mode operates in its own unique context.

10.6 Exception Handling in Root and Guest Mode

Exceptions are handled in the mode whose context triggered the exception. An exception triggered by the guest CP0 context will be handled in guest mode. An exception triggered by the root CP0 context is handled in root mode.

Figure 10.5 shows the how exceptions are handled in each of the operating modes (supervisor modes are omitted for clarity).

Figure 10.5 Exception Handling in Root and Guest Mode



In Figure 10.5, an operation executed in guest-user mode must travel through the root kernel to complete the operation.

The first layer to be crossed is the guest CP0 context (controlled by guest-kernel mode software). All exception and translation rules defined by the guest CP0 context are applied, and resulting exceptions are taken in guest mode by the guest kernel handler.

If the operation does not trigger a guest-context exception, the next layer to be crossed is the root CP0 context (controlled by root-kernel mode software). All exception and translation rules defined by the root CP0 context are applied, and resulting exceptions taken in root mode by the root kernel handler as shown.

For example, an access to Coprocessor 1 (the Floating Point Unit) must first be permitted by the guest context $Status_{CU1}$ bit, and then by the root context $Status_{CU1}$ bit.

Table 10.3 specifies the association of GuestID with TLB instructions. For supporting information, refer to Section 10.7.

Table 10.3 GuestID Use by TLB Instructions

TLB Operation	GuestID ($GuestCtl1_{ID}$/$GuestCtl1_{RID}$)
TLBGINV	$GuestCtl1_{RID}$
TLBGINVF	$GuestCtl1_{RID}$
TLBGP	$GuestCtl1_{RID}$
TLBGR	$GuestCtl1_{RID}$
TLBGWI	$GuestCtl1_{RID}$
TLBGWR	$GuestCtl1_{RID}$
TLBINV	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBINVF	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBP	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBR	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBWI	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$
TLBWR	if RootMode then $GuestCtl1_{RID}$ else $GuestCtl1_{ID}$

10.6.1 Root and Guest Shared TLB Operation

The P6600 core shares a common physical TLB amongst root and guest. The P6600 core contains a TLB structure that incorporates a VTLB (Variable page size TLB) and FTLB (Fixed page size TLB). As such, the VTLB must accommodate wired entries for both root and guest in a shared structure.

10.6.1.1 Root and Guest Access to the Shared TLB

In a shared TLB implementation, the root index increases from the bottom of the physical TLB while the guest index increases from the top of the physical TLB. This is to avoid overlap of root and guest wired entries. On the other hand, the root and guest indices to the FTLB grow from the bottom of the FTLB. Both guest and root TLB operations must interpret the TLB index accordingly.

10.6.1.2 Wired Register Management

The Root allocates the appropriate number of wired entries to itself, and then writes the guest *Config1* and *Config4* related fields to set the available VTLB entries for guest. Since the entries allocated for guest use also includes non wired entries shared by both root and guest, root software must be careful not to allocate all remaining non root-wired entries to the guest. This prevents the guest from populating all remaining non root-wired entries with its own guest-wired entries, leaving no entries for non root-wired entries.

Root software should not change guest MMU configuration while the guest is in operation, as is the case for any guest configuration that is read-only to guest but writeable by root.

10.6.1.3 CP0 Register Allocation

The Virtualization Module provides a partial set of CP0 registers for use by the guest. This is known as the *guest context*. When in guest mode, the behavior of the machine is controlled by the combination of the guest CP0 context and the root CP0 context. When in root mode, the behavior of the machine is controlled entirely by the root CP0 context.

The guest CP0 context consists of a base set plus optional features. Access to features within the guest CP0 context is controlled from root mode. The *Guest.Config0* through *Guest.Config5* registers determine which features are active during guest mode execution. The *GuestCtl0* register controls whether a guest access to a privileged feature triggers an exception.

10.6.1.4 CP0 Register Access

Guest CP0 registers can be accessed from root mode by using the root-only *MFGC0* and *MTGC0* instructions. Guest TLB contents can be accessed by using the root-only *TLBGP, TLBGR, TLBGWI* and *TLBGWR* instructions.

10.6.1.5 CP0 Register Initialization and Control

Root context software (hypervisor) is required to manage the initial state of writable Guest context registers. On power-up, the initial state defaults to the hardware reset state. On a Guest context save and restore, the hypervisor is required to preserve and re-initialize the Guest state. For virtual boot of a Guest, the hypervisor is required to initialize the Guest state equivalent to the hardware reset state. The Root may deconfigure one or more guest CP0 registers by writing to the guest configuration registers.

The Virtualization Module requires that scratch registers *KScratch1* and *KScratch2* are present in the root context. This ensures that hypervisor exception handlers have an adequate number of scratch registers to save and restore all general purpose registers in use by the guest.

10.6.2 New CP0 Registers

Coprocessor 0 registers have been added by the Virtualization Module to control the guest context. Table 10.4 describes CP0 registers introduced by the Virtualization Module. Refer to Chapter 2 of this manual for more information.

Table 10.4 CP0 Registers Introduced by the Virtualization Module

Register Number	Sel	Register Name	Description
12	6	*GuestCtl0*	Controls guest mode behavior.
10	4	*GuestCtl1*	Guest ID
10	5	*GuestCtl2*	Virtual Interrupts
11	4	*GuestCtl0Ext*	Extension to GuestCtl0
12	7	*GTOffset*	Offset for guest timer value

10.6.3 Guest CP0 Register Accesses Using Instructions

Guest CP0 registers can be accessed from root mode by using the root-only *MFGC0* and *MTGC0* instructions.

10.6.4 Guest CP0 Register Initialization and Control

Root context software (hypervisor) manages the initial state of writable Guest context registers. On power-up, the initial state defaults to the hardware reset state. On a Guest context save and restore, the hypervisor is required to preserve and re-initialize the Guest state. For virtual boot of a Guest, the hypervisor is required to initialize the Guest state equivalent to the hardware reset state. The Root may deconfigure one or more guest CP0 registers by writing to the guest configuration registers.

The Virtualization Module requires that scratch registers *KScratch1* and *KScratch2* are present in the root context. This ensures that hypervisor exception handlers have an adequate number of scratch registers to save and restore all general purpose registers in use by the guest.

10.6.5 CP0 Registers in the Guest Context

When a CP0 register is defined in the guest context, it is used to control guest execution. Fields in the *GuestCtl0* register can be used to cause Guest Privileged Sensitive Instruction exceptions when an access from guest mode is attempted. This allows hypervisor software to control the value of a register in the guestCP0 context (thus controlling guest-mode execution) while denying guest-kernel access to the register.

Attempting modification of certain fields in guest context CP0 registers triggers a Guest Software Field Change exception. In a similar manner, the Guest Hardware Field Change exception is triggered when a hardware initiated change to Guest CP0 registers occurs. These mechanisms are used to support Root recognition of Guest initiated changes to guest context CP0 registers.

Table 10.5 lists the CP0 registers that can be accessed by the Guest under the conditions shown.

Table 10.5 CP0 Registers in Guest CP0 Context

Register Number	Sel	Register Name	Available to Guest-Kernel software when	Guest Privileged Sensitive Instruction Exception when Root.GuestCtl0$_{CP0}$ = 0, or
0	0	*Index*	*Guest.Config*$_{MT}$ = 1 or *Guest.Config*$_{MT}$ = 4	*GuestCtl0Ext*$_{MG}$ = 1
1	0	*Random*		
2	0	*EntryLo0*		
3	0	*EntryLo1*		
4	0	*Context*		
4	1	*ContextConfig*	*Guest.Config3*$_{SM}$ = 1 or *Guest.Config3*$_{CTXTC}$ = 1	
4	2	*UserLocal*	*Guest.Config3*$_{ULRI}$ = 1	*GuestCtl0Ext*$_{OG}$ = 1
4	3	*XContextConfig*		

Table 10.5 CP0 Registers in Guest CP0 Context *(continued)*

Register Number	Sel	Register Name	Available to Guest-Kernel software when	Guest Privileged Sensitive Instruction Exception when Root.GuestCtl0$_{CP0}$ = 0, or
5	0	*PageMask*	$Guest.Config_{MT}$ = 1 or $Guest.Config_{MT}$ = 4	$GuestCtl0Ext_{MG}$ = 1
5	1	*PageGrain*		$GuestCtl0_{AT}$ = 1
5	2	*SegCtl0*	$Guest.Config3_{SC}$ = 1	
5	3	*SegCtl1*		
5	4	*SegCtl2*		
5	5	*PWBase*	$Guest.Config3_{PW}$ =1	
5	6	*PWField*		
5	7	*PWSize*		
6	0	*Wired*	$Guest.Config_{MT}$ = 1 or $Guest.Config_{MT}$ = 4	
6	6	*PWCtl*	$Guest.Config3_{PW}$ =1	
7	0	*HWREna*	$Guest.Config_{AR}$> = 1	$GuestCtl0Ext_{OG}$ = 1
8	0	*BadVAddr*	Always	$GuestCtl0Ext_{BG}$ = 1
8	1	*BadInstr*	$Guest.Config3_{BI}$ = 1	$GuestCtl0Ext_{BG}$ = 1
8	2	*BadInstrP*	$Guest.Config3_{BP}$ = 1	$GuestCtl0Ext_{BG}$ = 1
9	0	*Count*	*Always*	$GuestCtl0_{GT}$ = 0
10	0	*EntryHi*	$Guest.Config_{MT}$ = 1 or $Guest.Config_{MT}$ = 4	$GuestCtl0Ext_{MG}$ = 1
11	0	*Compare*	Always	$GuestCtl0_{GT}$ = 0
12	0	*Status*	Always	-
12	1	*IntCtl*	$Guest.Config_{AR}$> = 1	-
12	2	*SRSCtl*	$Guest.Config_{AR}$ >=1	Always
13	0	*Cause*	Always	-
14	0	*EPC*	Always	-
15	0	*PRid*	-	Always
15	1	*EBase*	$Guest.Config_{AR}$> = 1	-
15	2	*CDMMBase*	$Guest.Config3_{CDMM}$ = 1	Always
15	3	*CMGCRBase*	$Guest.Config3_{CMGCR}$ = 1	
16	0	*Config*	Always	On write access when $GuestCtl0_{CF}$ = 0.
16	1	*Config1*	$Guest.Config_{M}$ = 1	
16	2	*Config2*	$Guest.Config1_{M}$ = 1	
16	3	*Config3*	$Guest.Config2_{M}$ = 1	
16	4	*Config4*	$Guest.Config3_{M}$ = 1	
16	5	*Config5*	$Guest.Config4_{M}$ = 1	

Table 10.5 CP0 Registers in Guest CP0 Context *(continued)*

Register Number	Sel	Register Name	Available to Guest-Kernel software when	Guest Privileged Sensitive Instruction Exception when Root.GuestCtl0$_{CP0}$ = 0, or
17	0	*LLAddr*		*GuestCtl0Ext$_{OG}$* = 1
17	1	*MAAR*	*Guest.Config5$_{MRP}$* = 1	Always
17	2	*MAARI*	*Guest.Config5$_{MRP}$* = 1	Always
18	0	*WatchLo*	*Guest.Config1$_{WR}$* = 1	Conditional
19	0	*WatchHi*	*Guest.Config1$_{WR}$* = 1	
20	0	*XContext*		
23	0	*Debug*	*Guest.Config1$_{EP}$* = 1	Always
24	0	*DEPC*	*Guest.Config1$_{EP}$* = 1	
25	0-n	*PerfCnt*	*Guest.Config1$_{PC}$* = 1	Conditional, refer to Section 10.9.4
26	0	*ErrCtl*	-	Always
27	0	*CacheErr*		
28	0	*ITagLo*		
28	1	*IDataLo*		
28	2	*DTagLo*		
28	3	*DDataLo*		
28	4	*L2/3TagLo*		
28	5	*L2/3DataLo*		
29	0	*ITagHi*		
29	1	*IDataHi*		
29	5	*L2/3DataHi*		
30	0	*ErrorEPC*	Always	-
31	2	*KScratch1*	Always Defined by *Guest.Config4$_{KScrExist}$*	*GuestCtl0Ext$_{OG}$*=1
31	3	*KScratch2*		
31	4	*KScratch3*		
31	5	*KScratch4*		
31	6	*KScratch5*		
31	7	*KScratch6*		

10.6.6 Guest Config Register Fields

The *Guest.Config$_{0-5}$* registers control the behavior of architecture features during guest execution. The guest context is a subset of the root context. In addition, the guest context can only include features available in the root context. Root mode software can determine whether programmable features are available in the guest context by attempting to write values to *Guest.Config* fields.

Table 10.6 lists *Guest.Config* register fields which can be written from root mode

Table 10.6 Guest CP0 Read-only Config Fields Writable from Root Mode

Register	Field	Purpose
Config	MT	MMU Type
Config1	MMU Size - 1	Number of entries in (guest) MMU
Config1	PC	Performance Counter registers implemented
Config1	WR	Watch registers implemented
Config1	FP	FPU implemented
Config3	MSAP	MSA (MIPS SIMD Architecture) implemented
Config3	CTXTC	*ContextConfig* etc. implemented
Config3	LPA	40-bit PA is implemented

1. Root must be able to write guest MMU size related fields in *Config1* and *Config4* if a TLB is shared between root and guest.

10.6.7 Read-Only Guest Context Fields Writeable from Root

The Guest context CP0 registers include fields that are read only and dynamically set by hardware. Corresponding fields in the guest context can be written from root mode, but remain read-only to the guest.

Table 10.7 lists fields which are read-only to the guest and writable from root mode.

Table 10.7 Guest CP0 Read-only Fields Writable from Root Mode

Register	Field	Purpose
Index	P	Root restore of P in guest context.
Context	BadVPN2	Virtual Page Number from the address causing last exception.
BadVAddr	BadVAddr	Address causing last exception
Cause	BD	Last exception occurred in a delay slot
Cause	TI	Timer interrupt is pending
Cause	CE	Coprocessor number for coprocessor unusable exception
Cause	FDCI	Fast Debug Channel interrupt is pending
Cause	IP7..2	Non-EIC interrupt pending bits. Write to Cause[7:2] is *Optional* if GuestCtl2 implemented.
Cause	RIPL	EIC interrupt pending level. *Optional* if GuestCtl2 implemented.
Cause	ExcCode	Exception code, from last exception
EBase	CPUNum	CPU number in multi-core system
Status	SR	Soft Reset. Root write is *Optional*.[1]
Status	NMI	Non Maskable Interrupt. Root write is *Optional*. [1]
BadInstr	BadInstr	Faulting Instruction Word. *Optional* in base architecture.
BadInstrP	BadInstrP	Prior Branch Instruction. *Optional* in base architecture.
Wired	Limit	Allow root to set guest *Wired* Limit field. (Release 6)

1 Root writes of 1 to Guest.$Status_{SR}$ or Guest.$Status_{NMI}$ will not directly cause an interrupt in the guest. Root software may set EPC to the guest's reset vector and ERET back to the guest such that to the guest it appears as if an NMI or SR had occurred. This feature is useful for resetting a guest that might be hung or otherwise unresponsive.

10.7 New CP0 Instructions

The Virtualization Module introduces new instructions for root mode access to the guest CP0 context, and for a guest to make a call into root mode - a 'hypervisor call'.

Table 10.8 describes CP0 instructions introduced by the Virtualization Module.

Table 10.8 CP0 Instructions Introduced by the Virtualization Module

Instruction	Description
HYPCALL	Hypercall - call to root mode.
DMFGC0	Double Move from Guest CP0
DMTGC0	Double Move to Guest CP0
MFGC0	Move from Guest CP0
MTGC0	Move to Guest CP0
TLBGINV	Guest TLB Invalidate
TLBGINVF	Guest TLB Invalidate Flush
TLBGP	Probe Guest TLB
TLBGR	Read Guest TLB
TLBGWI	Write Guest TLB
TLBGWR	Write Random to Guest TLB

10.8 Virtualization Exceptions

Normal execution of instructions can be interrupted when an exception occurs. Such events can be generated as a by-product of instruction execution (e.g., an integer overflow caused by an add instruction or a TLB miss caused by a load instruction), by an illegal attempt to use a privileged instruction (e.g. MTC0 from user mode), or by an event not directly related to instruction execution (e.g., an external interrupt).

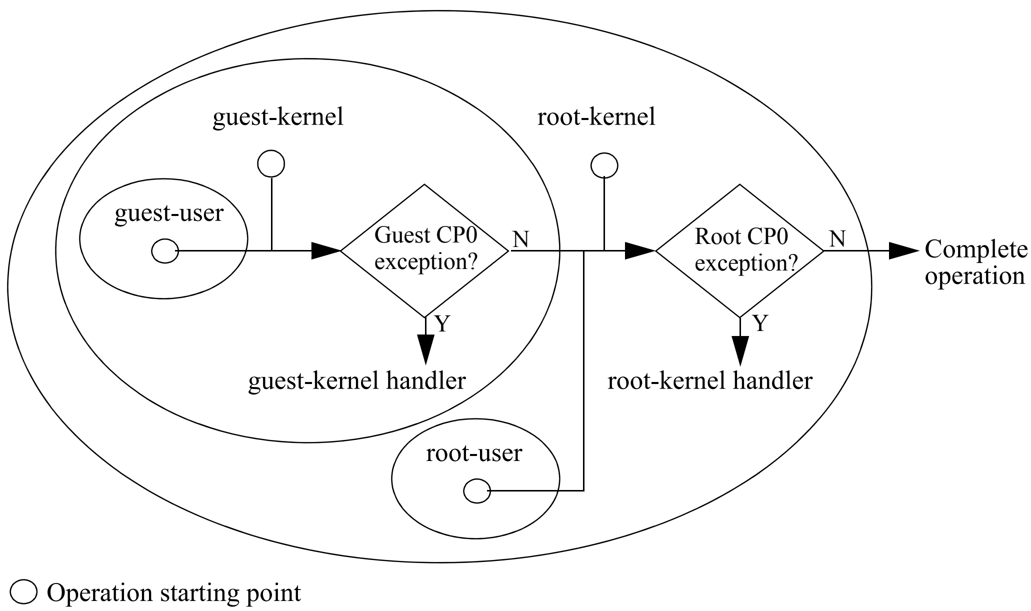
When an exception occurs, the processor stops processing instructions, saves sufficient state to resume the interrupted instruction stream, enters Exception or Error mode, and starts a software exception handler. The saved state and the address of the software exception handler are a function of both the type of exception, and the current state of the processor.

10.8.1 Overview of Exception Handling in Root and Guest Mode

Exceptions are handled in the mode whose context triggered the exception. An exception triggered by the guest CP0 context will be handled in guest mode. An exception triggered by the root CP0 context is handled in root mode.

Figure 10.6 shows the how exceptions are handled in each of the operating modes (supervisor modes are omitted for clarity).

Figure 10.6 Exception Handling in Root and Guest Mode



In Figure 10.6, an operation executed in guest-user mode must travel through the root kernel to complete the operation.

The first layer to be crossed is the guest CP0 context (controlled by guest-kernel mode software). All exception and translation rules defined by the guest CP0 context are applied, and resulting exceptions are taken in guest mode by the guest kernel handler.

If the operation does not trigger a guest-context exception, the next layer to be crossed is the root CP0 context (controlled by root-kernel mode software). All exception and translation rules defined by the root CP0 context are applied, and resulting exceptions taken in root mode by the root kernel handler as shown.

For example, an access to Coprocessor 1 (the Floating Point Unit) must first be permitted by the guest context *Status*$_{CU1}$ bit, and then by the root context *Status*$_{CU1}$ bit. However, access of guest to Coprocessor 0 is not qualified by root context *Status*$_{CU0}$ as Coprocessor 0 state is not shared with root.

10.8.2 Exceptions in Guest Mode

The Virtualization Module retains the exception-processing methodology of the base microMIPS architecture, and adds additional rules for processing of exception conditions detected during guest-mode execution.

The 'onion model' requires that every guest-mode operation be checked first against the guest CP0 context, and then against the root CP0 context. Exceptions resulting from the guest CP0 context can be handled entirely within guest mode without root-mode intervention. Exceptions resulting from the root-mode CP0 context (including *GuestCtl0* permissions) require a root mode (hypervisor) handler.

During guest mode execution, the mode in which an exception is taken is determined by the following:

- Guest-mode operations must first be permitted by guest-mode CP0 context and then by root mode CP0 context
  - This includes all operations for which exceptions can be generated - memory accesses, coprocessor accesses, breakpoints and so forth.
- Exceptions are always taken in the mode whose CP0 state triggered the exception
  - When architecture features in the guest context are present and enabled by the *Guest.Config* registers, exceptions triggered by those features are taken in guest mode.
  - Exceptions resulting from control bits set in the *Root.GuestCtl0* register, and exceptions resulting from address translation of guest memory accesses through the root-mode TLB are taken in root mode.

Asynchronous exceptions such as Reset, NMI, Memory Error, Cache Error are taken in root mode. External interrupts are received by the root CP0 context, and if enabled are taken in root mode. If an interrupt is not enabled in root mode and is bypassed to the guest CP0 context, and is enabled in the guest CP0 context, the interrupt is taken in guest mode.

When an exception is detected during guest mode execution, any required mode switch is performed after the exception is detected and before any machine state is saved. This allows machine state to be saved to either the root or guest contexts, and allows the exception to be handled in the proper mode. See also Section 10.8.3.

```
# Booleans, indicating source of exception:
#   root_async       - Asynchronous root context exception
#   root_sync        - Synchronous exception triggered by root context
#   guest_async      - Asynchronous exception triggered by guest context
#   guest_sync       - Synchronous exception triggered by guest context
#
# Exceptions directed to root context set Root.Status.ERL or Root.Status.EXL,
# meaning that the processor executes the handler in root mode.

# Ordering of exception conditions
if (root_async) then
    ctx ← Root
elsif (guest_async) then
    ctx ← Guest
elsif (guest_sync) then
    ctx ← Guest
```

```
elsif (root_sync) then
    ctx ← Root
else
    ctx ← null
endif

if (ctx) then
    # Defined by MIPS Privileged Resource Architecture
    ctx.GeneralExceptionProcessing()
endif
```

10.8.3 Faulting Address for Exceptions from Guest Mode

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions.

- Address error
- TLB Refill
- TLB Invalid
- TLB Modified
- TLB Execute Inhibit
- TLB Read Inhibit

10.8.4 Guest Initiated Root TLB Exception

When an exception is triggered as a result of a root TLB access during guest-mode execution, the handler will be executed in root mode, and exception state is stored into root CP0 registers. The registers affected are *GuestCtl0, Root.EPC*, *Root.BadVAddr*, *Root.EntryHi*, *Root.Cause* and $Root.Context_{BadVPN2}$.

The faulting address value stored into *Root.BadVAddr* and $Root.Context_{BadVPN2}$ is ideally the Guest Physical Address (GPA) presented to the root TLB by the guest context.

Whether the GPA can be provided is implementation dependent. If a GVA is mapped by the Guest MMU, yet the GPA is not available for write to root context, then $GuestCtl0_{GExcCode}$ must indicate this. In a specific e.g., guest TLB refill exception will always set GPA in $GuestCtl0_{GExcCode}$, while TLB modified/invalid/execute-inhibit/read-inhibit exceptions may set GVA due to implementation limitations.

The GPA presented to the root TLB is the result of translation through the guest context Segmentation Control if implemented, and through the guest TLB if in a mapped region of memory. The value stored in *Root.BadVAddr* and $Root.Context_{BadVPN2}$ is the Guest Physical Address being accessed by the guest.

This process ensures that after an exception, both *Root.BadVAddr* and $Root.Context_{BadVPN2}$ refer to a virtual address which is immediately usable by a root-mode handler, irrespective of whether the exception was triggered by root-mode or guest-mode execution.

10.8.5 Exception Priority

Table 10.9 lists all possible exceptions, and the relative priority of each, highest to lowest. The table also lists new exception conditions introduced by the Virtualization Module, and defines whether a switch to root mode is required before handling each exception.

Table 10.9 Priority of Exceptions

Exception	Description	Type	Taken in mode
Reset	The Cold Reset signal was asserted to the processor	Asynchronous Reset	Root
Soft Reset	The Reset signal was asserted to the processor		
Debug Single Step	An EJTAG Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.	Synchronous Debug	Root
Debug Interrupt	An EJTAG interrupt (EjtagBrk or DINT) was asserted.	Asynchronous Debug	Root
Imprecise Debug Data Break	An imprecise EJTAG data break condition was asserted.		
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor.	Asynchronous	Root
Machine Check	Root, or Root TLB related. This can only occur as part of a guest (second step) address translation, root address translation, and root TLB operation (write, probe) whether for guest or root TLB. It is recommended that the Machine-Check be synchronous. A TLB instruction must cause a synchronous Machine Check.	Asynchronous or Synchronous	Root
	An internal inconsistency was detected by the processor.		Root
	Guest TLB related. This can only occur as part of a guest address translation (first step), and guest TLB operation (write, probe). It is recommended that the Machine-Check be synchronous. A TLB instruction must cause a synchronous Machine Check.		Guest
Interrupt	A root enabled interrupt occurred.	Asynchronous	Root
Deferred Watch	A Root watch exception, deferred because EXL was one when the exception was detected, was asserted after EXL went to zero. A deferred root watch exception may occur in guest mode in which case it is prioritized higher than a simultaneous occuring guest interrupt.	Asynchronous	Root
Interrupt	A guest enabled interrupt occurred.	Asynchronous	Guest
Deferred Watch	A Guest watch exception, deferred because Guest EXL was one when the exception was detected, was asserted after EXL went to zero.	Asynchronous	Guest
Debug Instruction Break	An EJTAG instruction break condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses.	Synchronous Debug	Root

Table 10.9 Priority of Exceptions *(continued)*

Exception	Description	Type	Taken in mode
Watch - Instruction fetch	A root context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.	Synchronous	Root
	A guest-context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.		Guest
Address Error - Instruction fetch	A non-word-aligned address was loaded into PC.	Synchronous	Current
TLB Refill - Instruction fetch	A Guest TLB miss occurred on an instruction fetch	Synchronous	Guest
	A Root TLB miss occurred on an instruction fetch. This can occur due to a Root or Guest translation.		Root
TLB Invalid - Instruction fetch	The valid bit was zero in the guest context TLB entry mapping the address referenced by an instruction fetch.	Synchronous	Guest
	The valid bit was zero in the Root TLB entry mapping the address referenced by an instruction fetch. This can occur due to a Root or Guest translation.		Root
TLB Execute-inhibit	An instruction fetch matched a valid Guest TLB entry which had the XI bit set.	Synchronous	Guest
	An instruction fetch matched a valid Root TLB entry which had the XI bit set. This can occur due to a Root or Guest translation.		Root
Cache Error - Instruction fetch	A cache error occurred on an instruction fetch.	Synchronous or Asynchronous	Root
Bus Error - Instruction fetch	A bus error occurred on an instruction fetch.		
SDBBP	An EJTAG SDBBP instruction was executed.	Synchronous Debug	Root
Guest Reserved Instruction Redirect	A guest-mode instruction will trigger a Reserved Instruction Exception. When *$GuestCtl0_{RI}$*=1, this root-mode exception is raised before the guest-mode exception can be taken. Reserved Instruction Exception processing otherwise follow standard rules of prioritization within a given context - Reserved Instruction Redirect is taken as a side-effect of this processing.	Synchronous Hypervisor	Root

Table 10.9 Priority of Exceptions *(continued)*

Exception	Description	Type	Taken in mode
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources, or was illegal: Coprocessor Unusable,Reserved Instruction, MSA disabled. If exceptions occur on the same instruction, the Coprocessor Unusable, MSA disabled Exception take priority over the Reserved Instruction Exception.	Synchronous	Current
	Coprocessor unusable - guest. Access to a coprocessor was permitted by the *Guest.Status*$_{CU1-2}$ bits, but denied by *Root.Status*$_{CU1-2}$ bits. MSA disabled - guest. Access to the MSA unit was permitted by Guest.*Config5*$_{MSAEn}$, but denied by Root.*Config5*$_{MSAEn}$.		Root
Machine Check	Root TLB related. This can only occur as part of a Guest or Root address translation, or a TLBP/TLBWI/TLBGP/TLBGWI executed in root-mode.	Synchronous	Root
	Guest TLB related. This can only occur as part of a Guest address translation, or a TLBP/TLBWI executed in guest-mode		Guest
	An internal inconsistency was detected by the processor.		Root
Guest Privileged Sensitive Instruction Exception	An instruction executing in guest-mode could not be completed because it was denied access to the required resources by the *Root.GuestCtl0* register.	Synchronous Hypervisor	Root
Hypercall	A HYPCALL hypercall instruction was executed.	Synchronous Hypervisor	Root
Guest Software Field-Change	During guest execution, a software initiated change to certain CP0 register fields occured.	Synchronous Hypervisor	Root
Guest Hardware Field-Change	During guest execution, a hardware initiated set of *Status*$_{EXL/TS}$ occurred.	Synchronous Hypervisor	Root
Execution Exception	An instruction-based exception occurred: Integer overflow, trap, system call, breakpoint, floating point, coprocessor 2 exception.	Synchronous	Current
Precise Debug Data Break	A precise EJTAG data break on load/store (address match only) or a data break on store (address+data match) condition was asserted. Prioritized above data fetch exceptions to allow break on illegal data addresses.	Synchronous Debug	Root
Watch - Data access	A root context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses.	Synchronous	Root
	A guest context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses.		Guest
Address error - Data access	An unaligned address, or an address that was inaccessible in the current processor mode was referenced, by a load or store instruction	Synchronous	Current

Table 10.9 Priority of Exceptions *(continued)*

Exception	Description	Type	Taken in mode
TLB Refill - Data access	A guest TLB miss occurred on a data access	Synchronous	Guest
	A root TLB miss occurred on a data access. This can occur due to a Root or Guest translation.		Root
TLB Invalid - Data access	On a data access, a matching guest TLB entry was found, but the valid (V) bit was zero.	Synchronous	Guest
	On a data access, a matching root TLB entry was found, but the valid (V) bit was zero. This can occur due to a Root or Guest translation.		Root
TLB Read-Inhibit	On a data read access, a matching guest TLB entry was found, and the RI bit was set.	Synchronous	Guest
	On a data read access, a matching root TLB entry was found, and the RI bit was set. This can occur due to a Root or Guest translation.		Root
TLB Modified - Data access	The dirty bit was zero in the guest TLB entry mapping the address referenced by a store instruction	Synchronous	Guest
	The dirty bit was zero in the root TLB entry mapping the address referenced by a store instruction. This can occur due to a Root or Guest translation.		Root
Cache Error - Data access	A cache error occurred on a load or store data reference	Synchronous or Asynchronous	Root
Bus Error - Data access	A bus error occurred on a load or store data reference		
Precise Debug Data Break	A precise EJTAG data break on load (address+data match only) condition was asserted. Prioritized last because all aspects of the data fetch must complete in order to do data match.	Synchronous Debug	Root

The “Type” column of Table 10.9 describes the type of exception. Table 10.10 explains the characteristics of each exception type.

Table 10.10 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous Reset	Denotes a reset-type exception that occurs asynchronously to instruction execution. These exceptions always have the highest priority to guarantee that the processor can always be placed in a runnable state. These exceptions always require a switch to root mode.
Asynchronous Debug	Denotes an EJTAG debug exception that occurs asynchronously to instruction execution. These exceptions have very high priority with respect to other exceptions because of the desire to enter Debug Mode, even in the presence of other exceptions, both asynchronous and synchronous. These exceptions always require a switch to root mode.

Table 10.10 Exception Type Characteristics

Exception Type	Characteristics
Asynchronous	Denotes any other type of exception that occurs asynchronously to instruction execution. These exceptions are shown with higher priority than synchronous exceptions mainly for notational convenience. If one thinks of asynchronous exceptions as occurring between instructions, they are either the lowest priority relative to the previous instruction, or the highest priority relative to the next instruction. The ordering of the table above considers them in the second way. These exceptions always require a switch to root mode.
Synchronous Debug	Denotes an EJTAG debug exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions are prioritized above other synchronous exceptions to allow entry to Debug Mode, even in the presence of other exceptions. These exceptions always require a switch to root mode.
Synchronous Hypervisor	Denotes an exception that occurs as a result of guest-mode instruction execution which requires hypervisor intervention. It is reported precisely with respect to the instruction that caused the exception. These exceptions always require a switch to root mode.
Synchronous	Denotes any other exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions tend to be prioritized below other types of exceptions, but there is a relative priority of synchronous exceptions with each other. In some cases, these exceptions can be handled without switching modes.

10.8.6 Exception Vector Locations

Exception vector locations are as defined in the base architecture.

The vector location is determined from the values of *EBase*, $Status_{EXL}$, $Status_{BEV}$, $IntCtl_{VS}$ and $Config3_{VEIC}$ obtained from the context in which the exception will be handled.

The General Exception entry point is used for new hypervisor exceptions Guest Privileged Sensitive Instruction, Guest Reserved Instruction Redirect, Guest Software Field Change, Guest Hardware Field Change and Hypercall.

10.8.7 Synchronous and Synchronous Hypervisor Exceptions

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

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

Refer to the Exceptions chapter for more information on these exceptions.

10.8.8 Guest Exception Code in Root Context

In the case of a guest exception which causes a guest exit to root, hardware must supply the appropriate value for $Root.Cause_{ExcCode}$ and $GuestCtl0_{GExcCode}$, as described in the pseudo-code below.

```
if guest exception is (GPSI or GSFC or GHFC or HC or GRR or IMP) then
        Root.Cause_ExcCode ← "GE"
        Root.GuestCtl0_GExcCode ← "GPSI" or "GSFC" or "GHFC" or "HC" or "GRR" or "IMP"
elseif guest exception is (Root TLB-Refill or TLB-Invalid)
                Root.Cause_ExcCode ← "TLBS" or "TLBL"
                # loading of GPA for both TLB-Refill and TLB-Invalid is recommended.
                Root.GuestCtl0_GExcCode ← "GPA"
elseif guest exception is (Root TLB-Execute_Inhibit or TLB-Read_Inhibit)
        if (Root.PageGrain_IEC = 0) then
                Root.Cause_ExcCode ← "TLBL"
                Root.GuestCtl0_GExcCode ← "GPA" or GVA"
        elseif (TLB Execute-Inhibit)
                Root.Cause_ExcCode ← "TLBXI"
                Root.GuestCtl0_GExcCode ← "GVA" or "GPA"
        else
                Root.Cause_ExcCode ← "TLBRI"
                Root.GuestCtl0_GExcCode ← "GVA" or "GPA"
        endif
elseif guest exception is (TLB Modified)
                Root.Cause_ExcCode ← "MOD"
                Root.GuestCtl0_GExcCode ← "GVA" or "GPA"
else
        Root.Cause_ExcCode ←baseline "ExcCode"
        Root.GuestCtl0_GExcCode ← "UNDEFINED"
endif
```

10.9 Interrupts

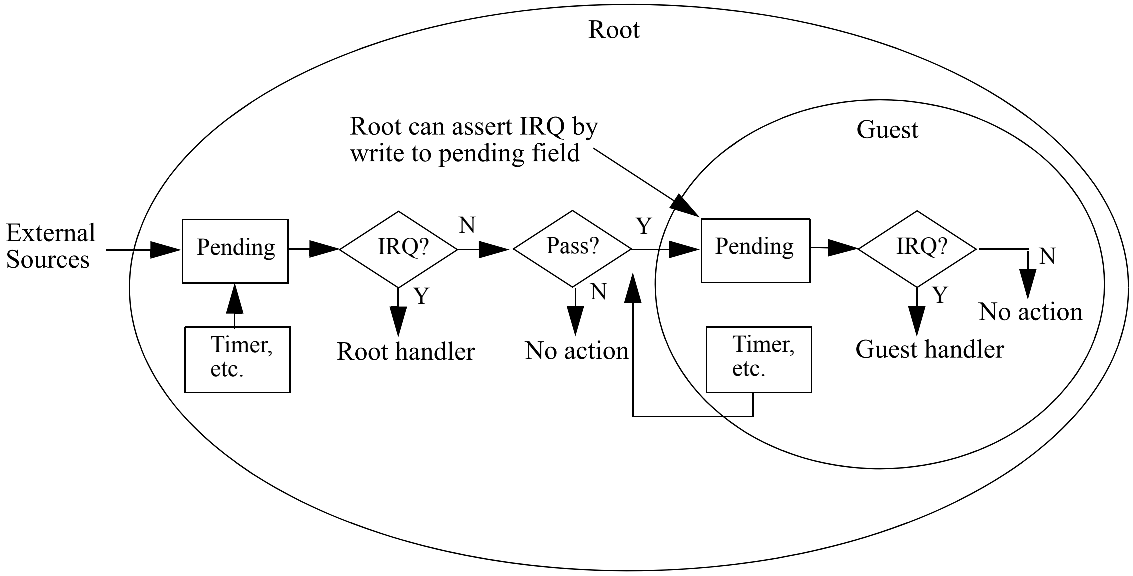
The Virtualization Module provides a virtualized interrupt system for the guest.

The root context interrupt system is always active, even during guest mode execution. An interrupt source enabled in the root context will always result in a root-mode interrupt. Guests cannot disable root mode interrupts.

Standard interrupt rules are used by both root and guest contexts to determine when an interrupt should be taken. An interrupt enabled in the root context is taken in root mode. An interrupt masked by root and enabled in the guest context is taken in guest mode. Root interrupts take priority over guest interrupts.

Figure 10.7 shows the how virtualized interrupts are managed in the P6600 core.

Figure 10.7 Interrupt Handling in the Virtualization Module I



The *Guest.Cause*$_{RIPL/IP}$ field is the source of guest interrupts. The behavior of this field is controlled from the root context. Two methods can be used to trigger guest interrupts - a root-mode write to the *Guest.Cause* register, or direct assignment of real interrupt signal to the guest interrupt system. Interrupt sources are combined such that both methods can be used.

Timers and related interrupts are available in both guest and root contexts.

The set of pending interrupts seen by the guest context is the combination (logical OR) of:

- External interrupts passed through from the root context, enabled by *GuestCtl0*$_{PIP}$ if implemented.
- Interrupts generated within the guest context (e.g., Timer interrupts, Software interrupts)
- Root asserted interrupts, set by software write to *GuestCtl2*$_{VIP}$ field in non-EIC mode, or hardware capture of a guest interrupt in *GuestCtl2*$_{GRIPL}$ in EIC mode.

Software should enable direct interrupt assignment only when root and guest agree on the interpretation of interrupt pending/enable fields in the *Status* and *Cause* registers. Direct assignment is appropriate if both Root and Guest use EIC mode, or if both use non-EIC mode. Root can track changes to the guest interrupt system status using the field-change exceptions which result from guest initiated changes to fields *Status*$_{BEV}$, *Cause*$_{IV}$ or *IntCtl*$_{VS}$.

Root must assign interrupts to Guest with caution. For example, in non-EIC mode, if an interrupt pin (HW[5:0]) is shared by multiple interrupt sources, then enabling direct guest visibility (in Guest *Cause*$_{IP[n]}$ via *GuestCtl0*$_{PIP[n]}$=1) will cause all the interrupt sources on that pin to be visible to the Guest, possibly removing Root intervention capability. If Root Software needs to guarantee Root intervention capability on an interrupt then that interrupt should not be directly visible to Guest.

In non-EIC mode, the guest timer interrupt is always applied to the interrupt source indicated by the *Guest.IntCtl*$_{IPTI}$ field and is not affected by the *GuestCtl0*$_{PIP}$ field. Similarly, Guest software interrupts are not affected by the *GuestCtl0*$_{PIP}$ field, and are always applied to the interrupt source indicated by *Guest.IntCtl*$_{IPPCI}$

A virtualization-based external interrupt delivery system, whether EIC or non-EIC provides the following capabilities:

1. Root assignment of External Interrupt.

   Hardware delivers interrupt to root context, with root-mode servicing of external interrupt.

2. Guest assignment of External Interrupt with Root Intervention.

   Hardware delivers interrupt to root context, with root-mode hand-off to guest by writing to $GuestCtl2_{vIP}$, followed by guest servicing of external interrupt.

   If root requires visibility into guest interrupts, then root should use this method to deliver interrupts to guest.

3. Guest assignment of External Interrupt without Root Intervention.

   Hardware delivers interrupt to guest context without root intervention, followed by guest servicing of external interrupt. The interrupt is not visible to root as root has made the choice to assign to guest.

A MIPS enabled virtualized external interrupt delivery system also provides support for Virtual Interrupts. Root can simulate a guest interrupt by writing 1 to $GuestCtl2_{vIP}$. It can subsequently clear the interrupt by writing 0 to $GuestCtl2_{vIP}$.

Virtual Interrupt capability can be used to support guest virtual drivers. Root will inject an interrupt into guest context. Guest will field the interrupt, and in so doing cause a trap to Root, either by device activity or protected memory access. Root may then clear the interrupt by writing to guest $Cause_{IP}$ set earlier.

10.9.1 External Interrupts

10.9.1.1 Non-EIC Interrupt Handling

This section provides a detailed description of non-EIC handling in a recommended implementation. The term HW is used to represent an external interrupt source. HW is alternatively referred to as IRQ in other sections of the Module. HW is a set of interrupt pins common to both root and guest context.

Whether an external interrupt is visible to guest context or root context is dependent on $GuestCtl0_{PIP}$ (Pending Interrupt Passthrough). If $GuestCtl0_{PIP[n]}$ =1, then HW[n] is visible to guest context through $Guest.Cause_{IP[n+2]}$, otherwise it is visible to root context through $Root.Cause_{IP[n+2]}$.

If $GuestCtl0_{PIP[n]}$=0, but Root needs to transfer the external interrupt to Guest, then it must write to a software visible register, $GuestCtl2_{vIP[n]}$ (Interrupt Pending, Virtual). This method is also used by Root to inject a virtual interrupt into guest context. It is also a convenient way for Root to save and restore interrupt state of a Guest, if an interrupt had been injected by Root, but needs to be preserved across context switches. In the absence of $GuestCtl2_{vIP}$, Root would need to derive the equivalent of vIP by reading $Guest.Cause_{IP}$ which may be problematic since other interrupts could also be present.

$GuestCtl2_{vIP}$, $Guest.Cause_{IP}$ and $Root.Cause_{IP}$ handling is described below in relation to $GuestCtl2_{vIP}$ and $GuestCtl0_{PIP}$. The application of $GuestCtl2_{HC}$ is discussed below.

$GuestCtl2_{vIP}$ Handling:

```
if (MTC0[GuestCtl2_vIP[n]]=1)
        GuestCtl2_vIP[n] ← 1
else if ((Deassertion of HW[n] and GuestCtl2_HC[n]) or (MTC0[GuestCtl2_vIP[n]]=0))
        GuestCtl2_vIP[n] ← 0
endif
```

$Guest.Cause_{IP}$ Handling:

```
Guest.Cause_IP[n+2] = ((HW[n] and GuestCtl0_PIP[n]) or GuestCtl2_vIP[n])
```

$Root.Cause_{IP}$ Handling:

```
Root.Cause_IP[n+2]
= (HW[n] and !(GuestCtl0_PIP[n] or (GuestCtl2_vIP[n] and GuestCtl2_HC[n])))
```

$GuestCtl2_{HC}$ is provided to control how $GuestCtl2_{vIP}$ is reset. If a bit of $GuestCtl2_{HC}$ is 1, then the deassertion of related external interrupt will always cause associated $GuestCtl2_{vIP}$ to be cleared. If a bit of $GuestCtl2_{HC}$ is 0 then the deassertion of HW[n] will not cause $GuestCtl2_{vIP}$ to be cleared. In this case, it is the responsibility of root software to clear by writing 0 to $GuestCtl2_{vIP\ [n]}$.

In summary, interrupt injection in guest context serves two purposes - root assignment of external interrupts and injection of virtual interrupts to Guest. $GuestCtl2_{HC}$ provides the means to root software to distinguish between the two. Root software can use this facility to transfer an external interrupt HW[n] for guest servicing. In this scenario, $GuestCtl2_{HC[n]}$=1 and the assertion of $GuestCtl2_{vIP\ [n]}$ will cause corresponding $Root.Cause_{IP[n+2]}$ to be cleared, thus transparently affecting the transfer. Otherwise, Root would have to disable interrupts for that specific source by clearing $Root.Status_{IM[n]}$. On the other hand, Root can use this capability to inject interrupts into Guest context for guest virtual device drivers, as an e.g.. In this case, $GuestCtl2_{HC[n]}$=0, the assumption is that there is no external interrupt tied to the injected interrupt, and thus assertion of $GuestCtl2_{vIP\ [n]}$ should not cause $Root.Cause_{IP[n+2]}$ to be cleared. $Guest.Cause_{IP[n+2]}$ is asserted in both cases described.

Virtual interrupt handling is an option that can be detected by the presence of *GuestCtl2*. Hardware clear capability is also an option, even if virtual interrupts are supported. This capability exists if the field is writeable or preset to 1.

10.9.1.2 EIC Interrupt Handling

In EIC mode, the external interrupt controller (EIC) is responsible for combining internal and external sources into a single interrupt-priority level, which appears in the $Cause_{RIPL}$ field.

When an implementation makes EIC mode available (as indicated by $Guest.Config3_{VEIC}$=1), two interrupt priority-level signals must be generated within the EIC - one for the root context (affecting $Root.Cause_{RIPL}$), and one for the guest context (affecting $Guest.Cause_{RIPL}$). The root and guest timer interrupt signals are combined in an implementation-dependent way with external inputs to produce the root and guest interrupt priority levels.

In addition to RIPL, the interrupt Vector (offset or number), and EICSS will also be sent on each of the root and guest interrupt buses. The Vector from the EIC is either utilized by hardware as is, or derived from the EIC input. A GuestID accompanies only the root bus, providing GuestID is supported in the implementation. This is because the EIC can also send an interrupt for guest on the root interrupt bus. Thus the GuestID for the root interrupt bus may be non-zero. The GuestID for a guest interrupt taken in root mode must be registered in $GuestCtl1_{EID}$. The guest associated with the guest bus is by default equal to $GuestCtl1_{ID}$.

In the architecture as defined, the type of vector a virtualized core can accept from the EIC is fixed - it is either a vector number or offset but never both. This is because currently there is no capability to distinguish between the two types, intentionally so. It is recommended that a typical virtualized EIC source a vector number to the core.

The EIC should assign interrupts to root and guest interrupt buses as per the following rules:

- Root interrupts must always be taken in root context and thus be presented on root interrupt bus by the EIC.

- If a guest interrupt requires root intervention, then it must be presented on the root interrupt bus by the EIC. And interrupt for a non-resident guest must always be sent on the root interrupt bus. An interrupt for the resident guest may also be sent on the root interrupt bus.

  A guest interrupt while the processor is in root mode can cause an interrupt immediately unless masked by $Root.Status_{IPL}$. Hardware should not stall the interrupt until the processor enters guest mode.

- Only an interrupt for a resident guest can be sent on the guest interrupt bus. If software programs the EIC to send an interrupt for a non-resident guest on the guest interrupt bus, then an implementation of the core is not required to respond to this interrupt. .

To allow the EIC to distinguish between resident and non-resident guests, the core must send $GuestCtl1_{ID}$ to the EIC. An implementation must account for the delay between when the $GuestCtl1_{ID}$ changes and when it is visible to the EIC to avoid a spurious interrupt for a non-resident guest from being sent on the guest interrupt bus.

The processor and EIC are required to implement a protocol to avoid the above mentioned race. On a guest context switch, root software must first write 0 to $GuestCtl1_{ID}$. This is equivalent to a STOP command for the EIC. EIC will recognize this as a stall and will not send interrupts to guest context by setting the requested interrupt priority level to 0 on the guest interrupt bus to the core. Root software can then save and restore guest context, followed by a write of new GuestID to $GuestCtl1_{ID}$. Once the write is complete, root software can enable guest mode operation. If an EIC implementation and root software follow this recommendation, then this prevents loss of an interrupt posted to the guest interrupt bus while root is switching guest context. An interrupt for the formerly active guest will now be posted on the root interrupt bus.

An EIC mode interrupt is generated in either guest or root context whenever hardware detects a change in RIPL on the respective interrupt buses from the EIC. It is possible for an EIC implementation to have active interrupts on both bus. In this case the root interrupt is always higher priority then the guest interrupt.

For the case of an interrupt in root context, two different interrupt vectors are used, one for root, the other for guest. Hardware is able to distinguish between the two by checking the GuestID on the root interrupt bus. The following pseudo-code describes how hardware generates the interrupt vector, depending on whether the EIC provides a vector offset (vectorOffset) or vector number (vectorNumber).

```
EIC_mode ← Config3.VEIC=1 && IntCtl.VS!=0 && Cause.IV=1 && Status.BEV=0
if EIC_mode
       if (EIC provides vectorNumber)
              if  (GuestID=0)
                     vectorOffset ← 0x200 + (EIC_vectorNumber x (IntCtl.VS || 0b00000))
              else //GuestID is non-zero
                     vectorOffset ←0x200
              endif
       else // EIC provides vectorOffset
              if  (GuestID=0)
                     // EIC provides an offset relative to 0x200
                     vectorOffset ←EIC_vectorOffset
              else //GuestID is non-zero
                     vectorOffset ←0x200
              endif
       endif
endif
```

If the interrupt is for guest, then the handler must compare $GuestCtl1_{EID}$ to $GuestCtl1_{ID}$. If they are not equal, then interrupt is for non-resident guest, and interrupt servicing may either continue in root or guest context. If interrupt servicing is to continue in guest context, then the handler must first save the resident guest architected state (CP0, GPRs etc) following by a restore of the new guest's context. The root ERET instruction causes a transfer to guest mode (when $GuestCtl0_{GM}$=1), followed by a guest interrupt providing $GuestCtl2_{GRIPL}$ is non-zero.

If $GuestCtl1_{EID}$ and $GuestCtl1_{ID}$ are equal, then save and restore is not needed. Interrupt servicing may either continue in root or guest context. If the interrupt is to be serviced in guest context, then the root ERET instruction causes a change to guest mode (when $GuestCtl0_{GM}$=1), following by a guest interrupt providing $GuestCtl2_{GRIPL}$ is non-zero.

As described above, for any change in $GuestCtl1_{ID}$, root software must first insert a STOP command on interface to EIC by writing 0 to $GuestCtl1_{ID}$. Once quiescent, root software may execute whatever software sequence it needs to. This is followed by a write of new GuestID to $GuestCtl1_{ID}$, then the root ERET instruction. There may be some arbitrary delay between write of GuestID and ERET instruction where EIC can respond with an interrupt on guest bus, but hardware will not trigger an interrupt because processor is in root mode.

A root interrupt must use $Root.SRSCtl_{EICSS}$. Otherwise, hardware forces use of $Root.SRSCtl_{ESS}$ if the interrupt on the root interrupt bus is for any guest.

The guest interrupt in the scenario where the interrupt is transferred from root context after having been received on the root interrupt bus is caused when the processor enters guest mode and hardware detects that $GuestCtl2_{GRIPL}$ is non-zero.

Once in guest mode, the guest interrupt handler completes with an ERET instruction. The guest will continue execution from its *EPC*, and not transfer back to root mode even if there was a change in guest context. If a return to root mode is required, then the HYPERCALL instruction must be used.

The root CP0 register, *GuestCtl2,* where the root interrupt bus Vector, EICSS and RIPL. Storage in root CP0 state is required because in a typical EIC-based implementation, an acknowlegement is returned to the EIC when the interrupt is triggered. If an interrupt for the guest is initially triggered in root context, then the use of these fields will not occur until the root ERET instruction is executed to effect a change to guest mode. In the meanwhile, another root interrupt can occur which can overwrite the fields on the bus. Saving the fields as root CP0 register allows for nesting of these fields, and thus supports nesting of interrupts.

Hardware optimizes the transfer of $GuestCtl2_{GRIPL}$ and $GuestCtl2_{EICSS}$ into guest CP0 context on guest entry. Hardware will write $GuestCtl2_{GRIPL}$ to $Guest.Cause_{RIPL}$, and $GuestCtl2_{EICSS}$ to $Guest.SRSCtl_{EICSS}$ providing $GuestCtl2_{GRIPL}$ is non-zero. Root software thus has the option of preventing hardware transfer by clearing $GuestCtl2_{GRIPL}$ before guest entry.

In the case where root injects an interrupt into guest context after the interrupt was received on the root interrupt bus, hardware must ensure that two acknowledgements are not returned to the EIC as this may cause a loss of an interrupt. In the case where an interrupt is received on the root interrupt bus, hardware must always send an acknowledgement on the root interrupt bus. But in the case where the interrupt was injected into guest context by root, hardware should not send an acknowledgement on the guest interrupt bus as the interrupt was not received on this bus. Hardware can determine this because $GuestCtl2_{GRIPL}$ would be a non-zero value for the case of root injection.

Access to COP1 FPR and COP2 may be protected setting $Root.Status_{CU[2:1]}$ appropriately. If access is disabled in root context, then it is also disabled in guest and will cause the appropriate exception (Coprocessor Unusable in root context). Hi/Lo registers are not protected by any means, and must be saved/restored if necessary.

10.9.2 Derivation of Guest.Cause$_{IP/RIPL}$

The interrupt pending value seen by the guest is calculated as shown below. The result value can be read by the guest (and the root) from the *Guest.Cause$_{RIPL / IP}$* field and is the value used to determine whether a guest interrupt will be taken. Note that the value returned from *Guest.Cause$_{RIPL / IP}$* on a read is generated from the value originally written by the root and from the status of directly assigned external interrupts. Hence the value written by the root may not be equal to the value read back.

```
# Returns:
# Non-EIC     IP7..0.
# EIC -       (RIPL << 2) + IP1..0

subroutine GuestInterruptPending() :

if ((Guest.Config3_VEIC = 1) and
    (Guest.IntCtl_VS != 0) and
    (Guest.Cause_IV = 1) and
    (Guest.Status_BEV = 0)) then
    # Guest in EIC mode
    # - GuestCtl0_PIP does not apply in EIC mode.
    # - EIC must include guest interrupt sources in the EICGuestLevel signal
    #   - This includes Guest's TI, IP1, IP0 and PCI if implemented.
        - FDCI is only visible in root context.
    # - GuestCtl2 required in EIC mode.
    if (EICGuestLevel > GuestCtl2_GRIPL)
        irq ← EICGuestLevel
    else
        irq ← GuestCtl2_GRIPL
        # h/w must clear if GuestCtl2_GRIPL is source of interrupt.
        GuestCtl2_GRIPL ← 0
    endif
    # Guest.Cause_IP[1:0] is incorporated in EIC.
    # State of Guest.Cause_IP[1:0] is however preserved.
    r  ←(irq << 2) OR Guest.Cause_IP[1:0]

else
    # Guest in non-EIC mode
    # - External interrupts factored in if guest passthrough enabled.
    # - Internal interrupts applied here, if implemented
    # - Includes support for guest interrupt injection by root.
    irq[7:2]  ← HW[5:0]
    if (GuestCtl0_PT=0)
        # All interrupts processed first by root.
        if (GuestCtl0_G2=1)
            # root software injects interrupts.
            r  ← GuestCtl2_vIP[5:0]
        else
            # if GuestCtl2_vIP is not supported, then root writes Guest.Cause.IP
            # to inject interrupt in guest context. H/W captures the write in a
            # shadow register called Root_HW_VIP.
            r  ← Root_HW_VIP[5:0]
        endif
    else
        # Guest interrupt passthrough supported.
        if (GuestCtl0_G2=1)
            r  ← Root.GuestCtl2_vIP[5:0] OR (irq[7:2] AND Root.GuestCtl0_PIP[5:0])
```

```
        else
            r    ← Root_HW_VIP[5:0] OR (irq[7:2] AND Root.GuestCtl0PIP[5:0])
        endif
    endif
    r          ← r << 2
    r          ← r OR (GuestTimerInterrupt << Guest.IntCtlIPTI)
    r          ← r OR (PCIEvent << Guest.IntCtlIPPCI)
    r          ← r OR Guest.CauseIP[1:0]

endif

return(r)
endsub
```

The value returned by GuestInterruptPending() will subsequently be qualified by Guest $\textit{Status}_{IM}$ in non-EIC mode or Guest $\textit{Status}_{IPL}$ in EIC mode, as per the base architecture.

Fields in Guest *Config* registers indicate which interrupt options are available to the guest.

10.9.3 Timer Interrupts

Root may inject a timer interrupt in guest context by setting Guest $\textit{Cause}_{TI}$ and indirectly Guest $\textit{Cause}_{IP[IPTI]}$. This may happen under the scenario where a guest has been switched out, but its virtual timer, maintained by root, is triggered. Root would set Guest $\textit{Cause}_{TI}$ before entering guest mode for the guest. Guest would take a timer interrupt, clear Guest *Compare*, which would then clear Guest $\textit{Cause}_{TI}$. As per baseline MIPS architecture, a write to *Compare* will clear $\textit{Cause}_{TI}$.

Root maintaining a virtual timer for a guest is recommended if there are multiple guests in operation. Otherwise, if there is only one guest, but the processor is in root mode, then a match on Guest *Count* and Guest *Compare* is allowed in an implementation to set Guest $\textit{Cause}_{TI}$ and Guest $\textit{Cause}_{IP[IPTI]}$. Once Root transitions to guest mode, then guest timer interrupt can be signaled in guest mode.

```
Root Injection of Guest TI:
        if (MTGC0[Guest.CauseTI]=1)
                Root.Guest.CauseTI ← 1
        else if ((MTC0[Guest.Compare]))
                Root.Guest.CauseTI ← 0
        endif
```

where Root.Guest.Cause_{TI} is a hardware shadow copy of Guest.Cause_{TI} that is set when Guest.Cause_{TI} is written by Root.

Guest.Cause$_{IP[IPTI]}$ = `Root.Guest.CauseTI or "Other External and Internal interrupts".`

where “Other External and Internal interrupts” is defined in Section 10.9.2.

10.9.4 Performance Counter Interrupts

The presence of performance counter registers in Guest context is indicated by Guest.Config1.PC. This bit is read-only to Guest, but writable by Root.

If Guest Config1.PC=0, the performance counters are unimplemented in the guest context and are treated as architecture reserved.

If Guest Config1.PC=1, the performance counters are virtually shared by root and guest contexts.

If virtually shared, the encodings of Root PerfCtrl.EC as 0 or 1 cause a GPSI exception to be raised on Guest access to a performance counter register. Root software may choose to configure performance counters for legal Guest access by encoding PerfCtrl.EC as 2 or 3. The EC field is not visible to the guest. It returns zero on guest read.

M bit in PerfCtrl is read-only in both root and guest context. It is 1 for PerfCtl 0-2 and 0 for PerfCtl 3.

PerfCtrl use of Status register K, S, U and EXL fields is taken from the current Root and Guest context.

Table 10.11 Performance Counter Interrupts

Guest. $Config1_{PC}$	Root. $PerfCnt_{EC[1:0]}$	Root mfgc0/mtgc0 Access Perf[n]	Guest mfc0/mtc0 Access Perf[n]
0	--	Write is dropped. Read returns 0.	If GstCtl0Ext.OG = 1 || GstCtl0.CP0 = 0 then GPSI, else writes are dropped and reads return 0
1	00 | 01	Allowed EC returns 0 on read	GPSI
1	10 | 11	Allowed EC returns 0 on read	If GstCtl0Ext.CP0 = 1 then GPSI, else access allowed. EC returns 0 on read.

10.10 Floating Point Unit (Coprocessor 1)

The guest and root contexts share the Floating Point Unit. The floating point unit is available to the guest context when $Guest.Config1_{FP}$ = 1.

During guest mode execution, access to the floating point unit is controlled by the $Status_{CU1}$ bits from both the root and guest contexts. The coprocessor enable bit $Guest.Status_{CU1}$ is checked first. If access is not granted, a coprocessor unusable exception is taken in guest mode.

The $Root.Status_{CU1}$ bit is checked next. If access is not granted by the $Root.Status_{CU1}$ bit, a coprocessor unusable exception is taken in root mode.

10.11 MSA (MIPS SIMD Architecture)

The guest and root contexts share the MSA module, if it is implemented. The MSA module is available to the guest context when $Guest.Config5_{MSAEn}$=1.

During guest mode execution, access to the MSA module is controlled by the $Config5_{MSAEn}$ bits from both the root and guest contexts. $Guest.Config5_{MSAEn}$ is checked first. If access is not granted, a MSA disabled exception is taken in guest mode.

The $Root.Config5_{MSAEn}$ bit is checked next. If access is not granted by $Root.Config5_{MSAEn}$, a MSA disabled exception is taken in root mode.

10.12 Guest Mode and Debug Features

The Virtualization Module provides full access to Debug facilities implemented through the EJTAG interface. When the processor is running in Debug privileged execution mode, it has full access to all resources that are available in the Root context.

As per Table 10.2, The Debug privileged execution mode exists in the root context. A processor supporting virtualization operates in two contexts, Root and Guest. Within Guest, there are three privileged execution modes; kernel, supervisor and user, and in Root context, there are four; kernel, supervisor, user and debug.

Table 10.12 lists debug features and their application to the Virtualization Module.

Table 10.12 Debug Features and Application to Virtualization Module

Feature	Description
Debug mode	Guest mode is mutually exclusive with Debug mode. When in Debug mode ($Debug_{DM}$=1), the processor is not in guest mode.
	When the processor is running in Debug mode, it has full access to all resources that are available to Root-Kernel mode operation.
Debug Segment (dseg)	When the processor is running in Debug mode, the memory map is determined by the root context. Memory mappings are unchanged from the EJTAG specification.
Access to guest CP0 context	Debug tools access general purpose registers (GPRs) and coprocessor registers by executing instructions in the processor pipeline. Access to the guest CP0 context must use the Virtualization Module instructions provided to transfer data between the root and guest contexts - MTGC0 and MFGC0. Accesses to the guest TLB must use the instructions provided to initiate guest TLB operations from the root context - TLBGP, TLBGR, TLBGWI, TLBGWR. These operations are used to transfer data between the guest TLB and the guest CP0 context. When accessing the guest TLB in debug mode, a two-step process is required - to transfer data to/from the guest CP0 context and guest TLB, and to transfer data to/from the root CP0 context and guest CP0 context.
Hardware Breakpoints	When implemented, hardware breakpoints are part of the root context. The root context remains active during guest mode execution, allowing hardware breakpoints to be used to debug guest software. Exceptions resulting from hardware breakpoints are of type Synchronous Debug or Asynchronous Debug. In both cases, the exceptions are handled in Debug mode.
Watch registers	Support for use of watchpoint from the Guest is optionally provided.

10.13 Watchpoint Debug Support

Root and Guest Watchpoint debug support is provided by Coprocessor 0 WatchHi and WatchLo register pairs. These registers are present in Root if Config1.WR=1 and in Guest if Guest.Config1.WR=1. Guest Config1 is read-only to guest but writable by root.

Guest Config1.WR=0, then watch registers are unimplemented in the guest context.

Guest Config1.WR=1, then watch registers are virtually shared between root and guest context.

If watch registers are virtually shared between root and guest, root software may choose to assign a subset or all watch registers to guest. This is configured through Root watchHi.WM field. WM field is for root context only. They are reserved and read as 0 for the Guest WatchHi register.

The P6600 does not support root watch GPA, a write of 1 to Root WatchHi.WM[1:0] will write 0 into this field. A write of 3 to Root WatchHi.WM[1:0] writes a value of 2 into this field.

The M bit in the WatchHi register is read-only in both root and guest context. It is 1 for Watch register pairs 0-2 and 0 for watch register pair 3.

Table 10.13 Watch Debug Control

Guest. $Config1_{WR}$	Root. $WatchHi_{WM[1:0]}$	Function	Root mfgc0/mtgc0 Access WatchHi[n]	Guest mfc0/mtc0 Access WatchHi[n]	Root Exception on Match	Guest Exception on Match
0	--	Root Watch RVA	Write is dropped. Read returns 0.	If GstCtl0Ext.OG = 1 || GstCtl0.CP0=0 then GPSI, else writes are dropped and reads return 0	Watch exception	No
1	00	Root Watch RVA	Allowed WM returns 0 on read	GPSI	Watch exception	No
1	10	Guest Watch GVA	Allowed WM returns 0 on read	Allowed WM returns 0 on read	No	Watch exception

Guest watch is enabled strictly in guest mode as defined by the equation:

($Root.GuestCtl0_{GM}$ = 1 and $Root.Status_{EXL}$ = 0 and $Root.Status_{ERL}$ = 0 and $Root.Debug_{DM}$ = 0)

There is no facility for Guest to watch addresses related to Root intervention events. That is, events occurring when the following equation is true:

($Root.GuestCtl0_{GM}$ = 1 and ($Root.Status_{EXL}$ = 1 or $Root.Status_{ERL}$ = 1 or $Root.Debug_{DM}$ = 1))

Chapter 11

Floating-Point Unit

This chapter describes the optional MIPS64® Floating-Point Unit (FPU) and contains the following sections:

- Section 11.1, "Features Overview"
- Section 11.2 "IEEE Standard 754"
- Section 11.3 "Enabling the Floating-Point Coprocessor"
- Section 11.4 "Enabling MSA"
- Section 11.5 "Architectural Overview"
- Section 11.6 "MIPS SIMD Architecture"
- Section 11.7 "Data Formats"
- Section 11.8 "Mapping of Scalar Floating-Point Registers to MSA Vector Registers"
- Section 11.9 "Floating-Point General Registers"
- Section 11.10 "Floating-Point Control Registers"
- Section 11.11 "MSA Control Registers"
- Section 11.12 "Floating Point and MSA Exceptions"
- Section 11.13 "Floating Point Instruction Overview"
- Section 11.14 "MSA Instruction Descriptions"
- Section 11.15 "Alphabetical Listing of Floating Point Instructions"
- Section 11.16 "Alphabetical Listing of MSA SIMD Instructions"

11.1 Features Overview

The P6600 core features an optional IEEE 754 compliant 3rd generation Floating Point Unit (FPU3) with SIMD.[1]

The FPU contains thirty-two, 128-bit vector registers shared between SIMD and FPU instructions. Single precision floating point instructions use the lower 32 bits of the 128 bit register. Double precision floating point instructions use the lower 64 bits of the 128 bit register. SIMD instructions use the entire 128 bit register interpreted as multiple vector elements; 16 x 8-bit, 8 x 16-bit, 4 x 32-bit, and 2 x 64 bit vector elements.

Some of the features of the P6600 core FPU include:

- Supports scalar FPU and MSA SIMD instructions.
- 32 128-bit vector registers. FPU instructions zero the upper 64 bits of the 128 bit MSA register.
- Supports FR = 1 mode only.

SIMD instructions enable:

- Efficient vector parallel arithmetic operations on integer, fixed-point and floating-point data.
- Operations on absolute value operands.
- Rounding and saturation options available.
- Full precision multiply and multiply-add.
- Conversions between integer, floating-point, and fixed-point data.
- Complete set of vector-level compare and branch instructions with no condition flag.
- Vector (1D) and array (2D) shuffle operations.
- Typed load and store instructions for endian-independent operation.

The FPU plus SIMD can be fully synthesized and operates at the same clock speed as the CPU. The IIU can issue up to two instructions per cycle to the FPU.

The FPU contains two execution pipelines for floating point and SIMD instruction execution. These pipelines operate in parallel with the integer core and do not stall when the integer pipeline stalls. This allows long-running FPU/SIMD operations such as divide or square root, to be partially masked by system stall and/or other integer unit instructions.

An out-of-order scheduler in the FPU issues instructions to the two execution units. The exception model is ‘precise’ at all times.

The FPU supports fused multiply-adds as defined by the IEEE Standard for Floating-Point Arithmetic 754TM-2008. The FPU is optimized for SIMD performance. Most FPU and SIMD instructions have one cycle throughput. All floating point denormalized input operands and results are fully supported in hardware.

11.2 IEEE Standard 754

The IEEE Standard 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, is referred to in this chapter as “IEEE Standard 754”. IEEE Standard 754 defines the following:

1. Requires separate MIPS license.

- Floating-point data types
- The basic arithmetic, comparison, and conversion operations
- A computational model

IEEE Standard 754 does not define specific processing resources nor does it define an instruction set.

11.3 Enabling the Floating-Point Coprocessor

Coprocessor 1 is enabled by setting the CU1 bit in the CP0 *Status* register. When this bit is cleared, Coprocessor 1 is disabled, and any attempt to execute a floating-point instruction causes a *Coprocessor Unusable* exception.

11.4 Enabling MSA

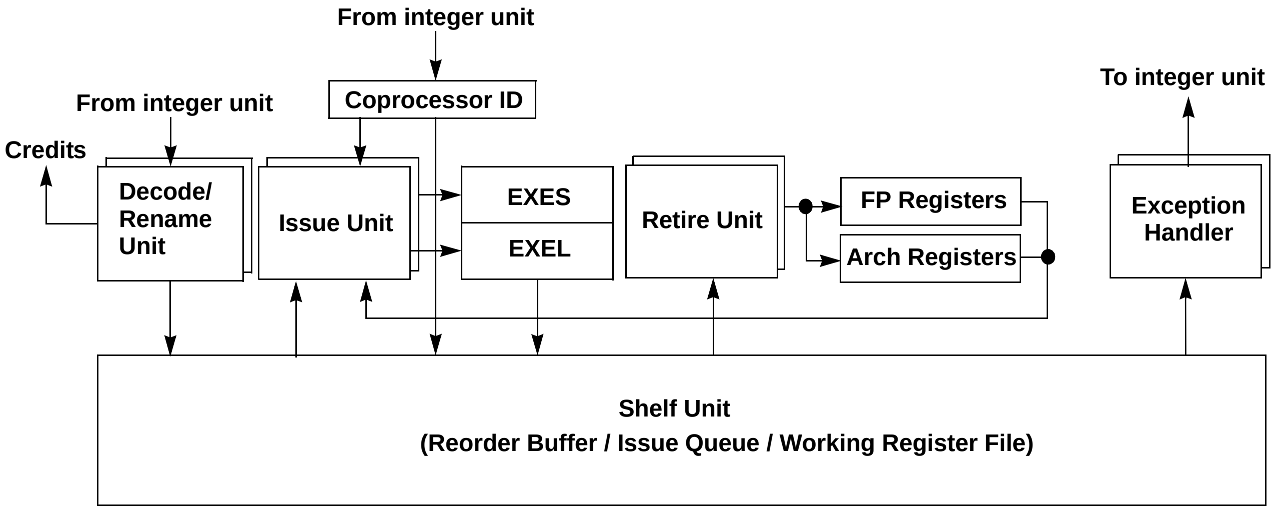
The presence of the MIPS SIMD architecture (MSA) implementation is indicated by the state of the Config3.MSAP bit (CP0 Register 16, Select 3, bit 28) at reset. The MSAP bit is fixed by the hardware implementation and is read-only for the software. Software can determine if MSA is implemented by checking if the MSAP bit is set. Any attempt to execute MSA instructions causes a Reserved Instruction Exception if the MSAP bit is not set. Note that this bit is always set in the P6600 core.

The Config5.MSAEn bit (CP0 Register 16, Select 5, bit 27) is used to enable access to the MSA instructions and the MSA vector registers. Executing a MSA instruction when MSAEn bit is not set causes a MSA Disabled Exception

11.5 Architectural Overview

Figure 11.1 shows a block diagram of the P6600 floating point unit.

Figure 11.1 Floating Point Unit Block Diagram



The blocks shown in Figure 11.1 are described in the following subsections.

11.5.1 Credits

The FPU uses a tagged interface to communicate with the integer core. Credits are sent to the core if resources are available. If the core has credits it can dispatch instructions to the FPU. The number and allocation of credits is a hardware function and is transparent to software.

The P6600 FPU allows up to 2 instructions to be dispatched per cycle. Each instruction is dispatched with a CID (Coprocessor ID) that is used to identify all subsequent interface transactions.

11.5.2 Coprocessor ID

The Coprocessor ID (CID) unit is responsible for mapping an incoming data for loads or move-to-FPU instructions with coprocessor ID to an entry in the shelf unit described in Section 11.5.9 "Shelf Unit".

There are two integer-to-floating point ports that contain the following features:

- A 128-bit SIMD load uses both 64-bit ports; both having the same coprocessor ID.
- An FP load hit/miss return uses a single 64-bit port, with one coprocessor ID.
- A bonded FP load hit/miss return uses both 64-bit ports, with different coprocessor IDs.
- A GPR register to FPU uses a single 64-bit port, with one coprocessor ID.

The Coprocessor ID block is responsible for determining if the transaction is GPR data or load data. A GPR transaction wakes up just that instruction and writes to only the lower 32 bits of the instruction shelf unit. A load transaction wakes up all load consumers and writes all 128 bits into the shelf unit.

11.5.3 Decode / Rename Unit

Dispatched instructions are decoded and registered in the Decode/Rename unit.

If the exception information can be determined at decode, then it is written to the shelves along with the rest of the decoded instruction. The Decode portion of the unit determines where the instruction executes, which sources are required, and which results are produced.

This Renamer starts source discovery for each dispatched instruction. Each of three source registers is compared across all shelf entries to see if that source is in the architecture register file or whether it should be read from a shelf. This rename step takes two clock cycles.

Each instruction has up to three operands and one result. Nominally, the sources are FPR registers, however the sources can also be mapped to a control register, and GPR data from the integer core. For each source, the operand parameters are compared against the result parameters of all in-flight instructions to determine whether that operand is produced by an in-flight instruction.

It is also possible that an instruction may be dependent upon an older instruction in the same dispatch clock cycle. Therefore, the rename unit also looks for dependencies across all concurrently dispatched instructions. If an older dependency is found, this dependency has higher priority than any matches found in the shelves.

11.5.4 Issue Unit

The Issue unit (ISU) determines the next instruction to issue to each of the two execution units; the short pipe (EXES) or the long pipe (EXEL). There is one ISU unit dedicated to each execution unit. The ISU is responsible for reading sources from either the architecture register file, the shelf entries, or from the bypass network.

The issue unit selects the oldest eligible instruction to issue, then looks up all instruction sources.

An instruction is eligible to issue if that instruction has all of the operands ready and all of the necessary execution resources available. If the instruction has immediate data, then the immediate was sign-replicated up to 11 bits and placed into the shelf during decode. In the Issue unit the immediate data is further sign-replicated up to the element size of the opcode and then replicated across all elements.

11.5.5 Execution Units

The P6600 FPU contains two execution units, one for short operations (EXES) and one for long operations (EXEL).

11.5.5.1 Short Operations

The short data path contains an integer add unit, logical unit, and div unit. The integer add unit and the logical unit each have 2-cycle latency outputs. One divide instruction can be issued to the div unit at a time. That divide will be worked on iteratively. Until the divide is done no other divide instructions can be issued. Two 64-bit data path modules are instantiated for 128 bit SIMD. Below is the diagram of how they are wired.

The short execution unit (EXES) executes the following instructions:

- All instructions that are sent back to the integer unit, including stores, move-from, and branches
- Control register moves (CTC1, CFC1, etc.)
- All integer add instructions

- All integer divide instructions
- Most 2-source logical operands.
- Floating point compares
  - min/max
  - fclass
  - abs, neg, mov
  - seleqz, selnez

Results from both execution pipelines are registered and written back to the shelf associated with the instruction. Additionally, exception information in the shelf is updated.

11.5.5.2 Long Operations

The long execution unit (EXEL) implements the following operations:

- Integer/fixed-point multiply
- FP adds, converts, multipliess, and divide-square roots
- All integer and fixed point multiply ops
- Logical operations with 3 sources

Results from the execution pipelines are registered and written back to the shelf associated with the instruction. Additionally, exception information in the shelf is updated.

11.5.6 Retire Unit

The retire units (RTU) commit data from the shelves to architectural state (ARF and FCSR) and deallocate the shelf entries. The P6600 core contains two Retire unit and therefore can retire two instructions per cycle. Retirement occurs in order. An instruction cannot retire until it is both graduated in the integer core and completed in the FPU.

The retire unit updates the architectural state and deallocates shelf entries. The architectural state update consists of:

- Write the instruction results from the shelf to the architectural registers.
- If the instruction is a CTC1, writes to the floating point control register.
- If the instruction is an arithmetic FP opcode, update the FP cause and flags fields in the floating point control registers.
- If the instruction is a CTCMSA, write to the MSA control registers.
- If the instruction an arithmetic MSA opcode, update the MSA cause and flags fields in the MSA control registers.

Retirement is strictly in-order. Retirement is implemented with a configurable number of identical retire units. If there are no hazards multiple instructions can retire per cycle, one from each retire unit.

Each Retire unit has a counter to point to the next shelf to retire in that unit. The count increments by the number of Retire units. With two Retire units one retires even shelves and the other retires odd shelves.

In order to make sure that there are no hazards, each retire unit broadcasts information about which shelf they are going to retire next to all other retire units. A retire unit stalls under any of the following conditions:

- There is an older instruction in the Retire unit that is not ready to retire
- There is a hazard with respect to an older instruction in another Retire unit

11.5.7 Architectural Register File

The FPU architectural register file supports five read ports. Three read ports are used by the long execution unit (EXEL) which supports three-source operations. Two read ports are used by the short exection unit (EXES).

11.5.8 Exception Handling

In the P6600 core exceptions are processed for every instruction as quickly as possible in order to speed up graduation and retirement in order to recycle resources for new instructions. The two EXCS modules can send exceptions from up to two instructions per cycle. One EXCS module manages exceptions across the even shelves and the other EXCS manages exceptions across the odd shelves. Exceptions from the FPU are tagged with the CID associated with the instruction.

11.5.9 Shelf Unit

The shelf is a unified re-order buffer, issue queue and working register file. The shelf unit is responsible for keeping track of the state of each instruction in the instruction stream, including selection, execution, and retirement of instructions.

11.6 MIPS SIMD Architecture

The MIPS® SIMD Architecture (MSA) module adds a set of more than 150 new instructions to the MIPS architecture that allow efficient parallel processing of vector operations. These instructions operate on 32 vector registers of 8-, 16-, 32-, and 64-bit integer, 16-and 32-bit fixed- point, or 32- and 64-bit floating-point data elements. In the P6600 core, MSA implements 128-bit wide vector registers shared with the 64-bit wide floating-point unit (FPU) registers.

The MSA provides increased system flexibility by incorporating a software-programmable solution for handling emerging codecs or other functions not covered by the dedicated hardware in the device. Rather than focusing on narrowly defined instructions that must have optimized code written manually in assembly language in order to be utilized, the MSA is designed to accelerate compute-intensive applications in conjunction with leveraging generic compiler support. Applications such as data mining, feature extraction in video, image and video processing, human-computer interaction, and others, have some built-in data parallelism that lends itself well to SIMD.

The SIMD instructions are easy to support within high-level languages such as C or OpenCL, enabling fast and simple development of new code, as well as leverage of existing code.

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™-2008. All standard operations are provided for 32-bit and 64-bit floating-point data. 16-bit floating-point storage format is supported through conversion instructions to/from 32-bit floating-point data.

11.6.1 MSA Vector Registers

The MSA operates on thirty-two 128-bit wide vector registers. If both MSA and the scalar floating-point unit (FPU) are present, the 128-bit MSA vector registers extend and share the 64-bit FPU registers.

MSA vector registers have four data formats: byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit). Corresponding to the associated data format, a vector register consists of a number of elements indexed from 0 to n, where the least significant bit of the 0^{th} element is the vector register bit 0 and the most significant bit of the n^{th} element is the vector register bit 127.

When both the FPU and the MSA are present, the floating-point registers are mapped on the corresponding MSA vector registers as the 0^{th} elements.

11.6.2 Layout of MSA Registers

Figure 11.2 through Figure 11.21 show the vector register layout for elements of all four data formats, where [n] refers to the n^{th} vector element and, MSB and LSB stand for the element's Most Significant and Least Significant Byte.

Figure 11.2 MSA Vector Register Byte Elements

127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
[15]	[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

Figure 11.3 MSA Vector Register Halfword Elements

127	112	111	96	95	80	79	64	63	48	47	32	31	16	15	0
[7]		[6]		[5]		[4]		[3]		[2]		[1]		[0]	
MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB

Figure 11.4 MSA Vector Register Word Elements

127			96	95			64	63			32	31			0
[3]				[2]				[1]				[0]			
MSB			LSB	MSB			LSB	MSB			LSB	MSB			LSB

Figure 11.5 MSA Vector Register Doubleword Elements

127							64	63							0
[1]								[0]							
MSB							LSB	MSB							LSB

MSA vectors are stored in memory starting from the 0th element at the lowest byte address. The byte order of each element follows the big- or little-endian convention of the system configuration.

11.6.3 MSA GNU Compiler Support

The GNU C Compiler (GCC) support for SIMD operations is based on a number of standard pattern names used for code generation. Ideally, the instruction set should implement as many of these operations as possible. In the process of MSA instruction selection and definition, supporting the standard GCC SIMD patterns was one of the most important objectives. Most of these patterns translate directly in single MSA instructions.

Another aspect related to efficient vector code compilation for SIMD architectures is the interoperability between the C language arrays (of scalar data types) and the native vector data types. To support seamless mixing of scalar and vector data types operations, the MSA provides a rich set of typed data transfer instructions.

11.6.3.1 MSA ABI

The O32 ABIs have been extended to allow efficient use of the vector registers and instructions defined by MSA. The MSA ABI extensions are compatible with the base ABIs in the sense that existing binaries run unchanged on systems supporting MSA. In other words, there are no incompatibilities between the base O32 ABI and the corresponding MSA extended ABI.

In particular, MSA ABI extensions;

- Do not change the base ABI data types layout / alignment
- Do not introduce new callee-saved (aka saved) registers
- Preserve the call-clobbered (aka temporary) or callee-saved (aka saved) status of the aliased floating-point registers.

However, vector data types are considered part of the MSA ABI by default and passed / returned by value without any MSA flags results in a compiler warning.

11.6.3.2 ABI Requirements

To be compatible with the MSA hardware, an ABI extension for MSA must support 32 64-bit floating point registers and a stack frame aligned to the size of the vector registers. The O32 FR1 ABI permits use of 64-bit floating point registers.

It is possible to adjust the stack alignment at run time using an existing compiler mechanism called dynamic stack realignment. Any ABI that does not meet the MSA stack alignment will therefore use dynamic stack re-alignment. For example, the 16-byte stack alignment of N32 and N64 ABIs is enough for MSA's 128-bit vector registers. However, the O32 ABI must perform dynamic stack re-alignment in this case.

11.6.3.3 Command Line Options and Function Attributes

Compiling for MSA (using the MSA defined instructions and vector registers) is enabled by the -mmsa command line option. A function compiled for MSA is referred to as a MSA function.

By default, the -mmsa option enables a faster calling convention for those functions passing vectors by value. This is achieved by using the vector registers for passing MSA vectors by value and returning MSA vector values.

A second MSA-related command line argument, -msimd-abi=none, can be used to disable the parameter passing/returning values in the vector registers. With -msimd-abi=none, all vector data types follow the calling conventions of the base ABI.

The use of vector types passed by value without the -mmsa option results in an ABI warning stating that a non-default ABI will be emitted. This warning can be disabled by explicitly passing the -msimd-abi=none option. It is illegal to use the -msimd-abi=msa option without -mmsa.

The functionality enabled by the command line option -mmsa can be disabled using -mno-msa. The SIMD ABI can be controlled by varying the value given to the -msimd-abi option. In particular, two SIMD ABIs are defined:

- none - Use the base calling convention
- msa - Use the MSA calling convention (default)

Equivalently, the same functionality could be enabled/disabled at the function level using __attribute__() as shown below.

- -mmsa __attribute__((msa))
- -mno-msa __attribute__((no_msa))
- -msimd-abi=none __attribute__((simd_abi_none))
- -msimd-abi=msa __attribute__((simd_abi_msa))

For convenience, pre-processor symbols are defined for each option as follows:

- -mmsa __MSA__
- -mno-msa __NO_MSA__
- -msimd-abi=none __SIMD_ABI_NONE___
- -msimd-abi=msa __SIMD_ABI_MSA__

11.6.3.4 Vector and Floating-Point Register Usage for -mmsa and -msimd-abi=msa

The MSA vector registers are temporary, and all live vector registers must be saved before calling a function. This ensures MSA functions can call any other function and compatibility with future MSA extensions.

The first 8 vector parameters are passed via vector registers w4 to w11 and vector results are returned via vector register w0. Floating-point registers are passed and returned as specified by the particular ABI.

For functions with variable arguments, no vector registers are used to pass vector parameters. This falls back to the original variable argument passing scheme from the particular ABI.

Note that compilers need to preserve the aliased callee-saved floating-point registers as specified by the O32 FR1, N32, and N64 ABIs: even f20, f22, ..., f30 for O32 FR1 and N32, and f24, f25, ..., f30, f31 for N64. For example, if the vector register w30 is used, the aliased floating point register f30 has to be preserved under all ABIs.

11.6.3.5 Inter-calling Between MSA and non-MSA Functions

A function that takes a MSA vector by value as a parameter or returns a MSA vector by value and is compiled with -mmsa can be called only by functions compiled with -mmsa.

Any function compiled with -msimd-abi=none can be called by non-MSA functions, i.e. a functions compiled under the base ABI with MSA disabled.

11.6.3.6 MSA GNU Options and Directives

The MSA is supported by the GNU toolchain starting with GAS (GNU Assembler) 2.22.51 and GCC 4.7.3. The command line options and assembly directives to enable/disable MSA are shown in Table 11.1.

The GCC options `-mfp64` and `-mhard-float` enforce the compatibility of the calling conventions of MSA and FPU, based on the fact that in the current release, MSA vector registers are shared with the 64-bit wide floating-point unit (FPU) registers.

Table 11.1 MSA GNU Options and Directives

	GAS		GCC	
	Enable	**Disable**	**Enable**	**Disable**
Command Line Options	`-mmsa`	`-mno-msa`	`-mmsa -mfp64 -mhard-float`	`-mno-msa`
Assembly Directives	`.set msa`	`.set nomsa`		

The GCC integer and floating-point vector data types with generic MSA operation support are listed in Table 11.2 and Table 11.3.

Table 11.2 GCC Integer Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of signed bytes	`typedef signed char **wi8_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of unsigned bytes	`typedef unsigned char **wu8_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of signed halfwords	`typedef short **wi16_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of unsigned halfwords	`typedef unsigned short **wu16_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of signed words	`typedef int **wi32_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of unsigned words	`typedef unsigned int **wu32_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of signed doublewords	`typedef long long **wi64_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of unsigned doublewords	`typedef unsigned long long **wu64_t** __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`

Table 11.3 GCC Floating-Point Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of single precision floating-point values	`typedef float wf32_t __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`
Vector of double precision floating-point values	`typedef double wf64_t __attribute__ ((vector_size(16))) __attribute__ ((aligned(16)));`

MSA instructions are available to the C/C++ programmer either by the inline assembly `__asm__` directive, by `msa_`*`mnemonic`*`()` intrinsics, or when using most of the C/C++ operators on vector data types. The list of supported vector C/C++ operators include: `+`, `-`, `*`, `/`, `%`, `^`, `|`, `&`, `<<`, `>>`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `~`.

For example, adding or comparing two single-precision floating-point vectors, as in:

```
wi32_t t;
wf32_t a, b, c;

a = b + c;
t = b < c;
```

compiles directly in MSA word floating-point add and compare instructions:

```
fadd.w $w3,$w0,$w1 # a is in $w3, b in $w0, c in $w1
fclt.w $w4,$w0,$w1 # t is in $w4
```

Regarding the vector parameter passing conventions, MSA registers are all caller-saved, i.e. temporary registers are not preserved between function calls. The first eight vector parameters are passed in vector registers W4 to W11. When compiled for the MSA, the stack pointer is always aligned to 16 bytes.

11.7 Data Formats

The FPU provides both floating-point and fixed-point data types, which are described below:

- The single- and double-precision floating-point data types are those specified by IEEE Standard 754.
- The signed integers provided by the CPU architecture.
- The fixed-point Q15 and Q31 types for MSA.

11.7.1 Floating-Point Formats

The FPU provides the following two floating-point formats:

- A 32-bit single-precision floating point (type S)
- A 64-bit double-precision floating point (type D)

The floating-point data types represent numeric values as well as the following special entities:

- Two infinities, $+\infty$ and $-\infty$

- Signaling non-numbers (SNaNs)
- Quiet non-numbers (QNaNs)
- Numbers of the form: $(-1)^s 2^E b_0.b_1 b_2..b_{p-1}$, where:
  - s = 0 or 1
  - E = any integer between E_min and E_max, inclusive
  - b_i = 0 or 1 (the high bit, b_0, is to the left of the binary point)
  - p is the signed-magnitude precision

The single and double floating-point data types are composed of three fields—sign, exponent, fraction—whose sizes are listed in Table 11.4.

Table 11.4 Parameters of Floating-Point Data Types

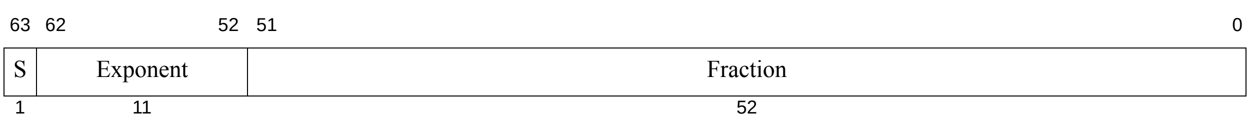
Parameter	Single	Double
Bits of mantissa precision, p	24	53
Maximum exponent, E_max	+127	+1023
Minimum exponent, E_min	-126	-1022
Exponent *bias*	+127	+1023
Bits in exponent field, *e*	8	11
Representation of b_0 integer bit	hidden	hidden
Bits in fraction field, *f*	23	52
Total format width in bits	32	64
Magnitude of largest representable number	3.4028234664e+38	1.7976931349e+308
Magnitude of smallest normalized representable number	1.1754943508e-38	2.2250738585e-308

Layouts of these three fields are shown in Figures 11.6 and 11.7 below. The fields are:

- 1-bit sign, *s*
- Biased exponent, $e = E + bias$
- Binary fraction, $f = .b_1 b_2..b_{p-1}$ (the *b*0 bit is *hidden*; it is not recorded)

Figure 11.6 Single-Precision Floating-Point Format (S)

Figure 11.7 Double-Precision Floating-Point Format (D)



Values are encoded in the specified format using the unbiased exponent, fraction, and sign values listed in Table 11.5. The high-order bit of the Fraction field, identified as b_1, is also important for NaNs.

Table 11.5 Value of Single or Double Floating-Point Data Type Encoding

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[1]	Typical Double Bit Pattern[1]
E_max + 1	≠ 0		1	SNaN	Signaling NaN ($FCSR_{NAN2008}$ = 0)	`0x7fffffff`	`0x7fffffff ffffffff`
			0	QNaN	Quiet NaN ($FCSR_{NAN2008}$ = 0)	`0x7fbfffff`	`0x7ff7ffff ffffffff`
E_max + 1	≠ 0		0	SNaN	Signaling NaN ($FCSR_{NAN2008}$ = 1)	`0x7fbfffff`	`0x7ff7ffff ffffffff`
			1	QNaN	Quiet NaN ($FCSR_{NAN2008}$ = 1)	`0x7fffffff`	`0x7fffffff ffffffff`
E_max +1	0	1		$-\infty$	Minus infinity	`0xff800000`	`0xfff00000 00000000`
		0		$+\infty$	Plus infinity	`0x7f800000`	`0x7ff00000 00000000`
E_max to *E_min*		1		$-(2^E)(1.f)$	Negative normalized number	`0x80800000` through `0xff7fffff`	`0x80100000 00000000` through `0xffefffff ffffffff`
		0		$+(2^E)(1.f)$	Positive normalized number	`0x00800000` through `0x7f7fffff`	`0x00100000 00000000` through `0x7fefffff ffffffff`
E_min -1	≠ 0	1		$-(2^{E\_min})(0.f)$	Negative denormalized number	`0x807fffff`	`0x800fffff ffffffff`
		0		$+(2^{E\_min})(0.f)$	Positive denormalized number	`0x007fffff`	`0x000fffff ffffffff`
E_min -1	0	1		- 0	Negative zero	`0x80000000`	`0x80000000 00000000`
		0		+ 0	Positive zero	`0x00000000`	`0x00000000 00000000`

1. The "Typical" nature of the bit patterns for the NaN and denormalized values reflects the fact that the sign might have either value (NaN) and that the fraction field might have any non-zero value (both). As such, the bit patterns shown are one value in a class of potential values that represent these special values.

11.7.1.1 Normalized and Denormalized Numbers

For single and double data types, each representable nonzero numerical value has just one encoding; numbers are kept in normalized form. The high-order bit of the p-bit mantissa, which lies to the left of the binary point, is "hidden," and not recorded in the *Fraction* field. The encoding rules permit the value of this bit to be determined by looking at the value of the exponent. When the unbiased exponent is in the range *E_min* to *E_max*, inclusive, the number is normalized and the hidden bit must be 1. If the numeric value cannot be normalized because the exponent would be less than *E_min*, then the representation is denormalized, the encoded number has an exponent of *E_min* – 1, and the hidden bit has the value 0. Plus and minus zero are special cases that are not regarded as denormalized values.

11.7.1.2 Reserved Operand Values—Infinity and NaN

A floating-point operation can signal IEEE exception conditions, such as those caused by uninitialized variables, violations of mathematical rules, or results that cannot be represented. If a program does not trap IEEE exception conditions, a computation that encounters any of these conditions proceeds without trapping but generates a result indicating that an exceptional condition arose during the computation. To permit this case, each floating-point format defines representations (listed in the table above) for plus infinity (+∞), minus infinity (–∞), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

11.7.1.3 Infinity and Beyond

Infinity represents a number with magnitude too large to be represented in the given format; it represents a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero operations and some cases of overflow as described in Section 11.12.2 “Exception Conditions”.

Once created as a default result, ∞ can become an operand in a subsequent operation. The infinities are interpreted such that -∞ < (every finite number) < +∞. Arithmetic with ∞ is the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such limits exist. In these cases, arithmetic on ∞ is regarded as exact, and exception conditions do not arise. The out-of-range indication represented by ∞ is propagated through subsequent computations. For some cases, there is no meaningful limiting case in real arithmetic for operands of ∞. These cases raise the Invalid Operation exception condition as described in Section 11.12.2.1 “Invalid Operation Exception”.

11.7.1.4 Signalling Non-Number (SNaN)

SNaN operands cause an Invalid Operation exception for arithmetic operations. SNaNs are useful values to put in uninitialized variables. An SNaN is never produced as a result value.

IEEE Standard 754 states that “Whether copying a signaling NaN without a change of format signals the Invalid Operation exception is the implementor’s option.” The MIPS architecture makes the formatted operand move instructions non-arithmetic; they do not signal IEEE 754 exceptions.

11.7.1.5 Quiet Non-Number (QNaN)

QNaNs provide retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires information contained in a QNaN to be preserved through arithmetic operations and floating-point format conversions.

QNaN operands do not cause arithmetic operations to signal an exception. When a floating-point result is to be delivered, a QNaN operand causes an arithmetic operation to supply a QNaN result. When possible, this QNaN result is one[2] of the operand QNaN values. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating-point result—specifically, comparisons. (For more information, see the detailed description of the floating-point compare instruction, C.cond.fmt.).

When certain invalid operations not involving QNaN operands are performed but do not trap (because the trap is not enabled), a new QNaN value is created. Table 11.6 shows the QNaN value generated when no input operand QNaN value can be copied. The values listed for the fixed-point formats are the values supplied to satisfy IEEE Standard

2. In case of one or more QNaN operands, a QNaN is propagated from one of the operands according to the following priority: 1: fs, 2: ft, 3: fr.

754 when a QNaN or infinite floating-point value is converted to fixed point. There is no other feature of the architecture that detects or makes use of these “integer QNaN” values.

Table 11.6 Value Supplied When a New Quiet NaN is Created

Format	QNaN value ($FCSR_{NAN2008}$ = 1)
Single floating point	`0x7FC0_0000`
Double floating point	`0x7FF8_0000_0000_0000`

Word fixed point	`0x7FFF_FFFF (value when converting any FP number too big to represent as a 32-bit positive integer)`
	`0x0000_0000 (value when converting any FP NaN)`
	`0x8000_0000 (value when converting any FP number too small to represent as a 32-bit negative integer)`

Longword fixed point	`0x7FFF_FFFF_FFFF_FFFF (value when converting any FP number too big to represent as a 64-bit positive integer)`
	`0x0000_0000 (value when converting any FP NaN)`
	`0x8000_0000 (value when converting any FP number too small to represent as a 64-bit negative integer)`

11.7.2 Signed Integer Formats

The FPU instruction set provides the following signed integer data types:

- A 32-bit Word fixed point (type W), shown in Figure 11.8.
- A 64-bit Longword fixed point (type L), shown in Figure 11.9.

The fixed-point values are held in 2’s complement format, which is used for signed integers in the CPU. Unsigned fixed-point data types are not provided by the architecture; application software can synthesize computations for unsigned integers from the existing instructions and data types.

Figure 11.8 Word Fixed-Point Format (W)

31	0
Integer	

Figure 11.9 Longword Fixed-Point Format (L)

63	0
Integer	

Only doing FPU, not FPU + MSA. FPU ISA supports 4 formats: S (32-bit single), D (32-bit single), W, L.

11.7.3 MSA Data Types

MSA instructions have 2- or 3-register, immediate, or element operands. One of the destination data format abbreviations shown in Table 11.7 is appended to the instruction name. Note that the data format abbreviation is the same regardless of the instruction's assumed data type. For example, all integer, fixed-point, and floating-point instructions operating on 32-bit elements use the same word (.W in Table 11.7) data format.

Table 11.7 Data Format Abbreviations

Data Format	Abbreviation
Byte, 8-bit	.B
Halfword16-bit	.H
Word, 32-bit	.W
Doubleword, 64-bit	.D
Vector	.V

11.7.4 MSA Vector Element Selection

MSA instructions select the n^{th} element in the vector register *ws* (*ws[n]* in assembly language) based on the data format *df*. Valid element index values for various data formats and vector register sizes are shown in Table 11.8.

Table 11.8 Valid Element Index Values

Data Format	Element Index
Byte	$n = 0, \ldots, 15$
Halfword	$n = 0, \ldots, 7$
Word	$n = 0, \ldots, 3$
Doubleword	$n = 0, 1$

11.7.5 Examples

Assume that vector registers W1 and W2 are initialized to the word values shown in Figure 11.10, Figure 11.11, and that general-purpose register R2 is initialized as shown in Figure 11.12.

Figure 11.10 Source Vector W1 Values

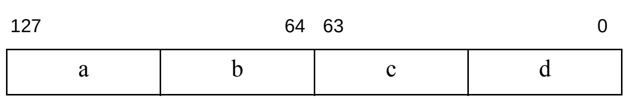


Figure 11.11 Source Vector W2 Values

127	64	63	0
A	B	C	D

Figure 11.12 Source GPR 2 Value

31 0
E

Regular MSA instructions operate element-by-element with identical source, target, and destination data types. Figure 11.13 through Figure 11.16 have the resulting values of destination vectors W4, W5, W6, and W7 after executing the following sequence of word additions and move instructions:

```
addv.w $w5,$w1,$w2
fill.w $w6,$2
addvi.w $w7,$w1,17
splati.w $w8,$w2[2]
```

Figure 11.13 Destination Vector W5 Value for ADDV.W Instruction

127	64	63	0
a + A	b + B	c + C	d + D

Figure 11.14 Destination Vector W6 Value for FILL.W Instruction

127	64	63	0
E	E	E	E

Figure 11.15 Destination Vector W7 Value for ADDVI.W Instruction

127	64	63	0
a + 17	b + 17	c + 17	d + 17

Figure 11.16 Destination Vector W8 Value for SPLAT.W Instruction

127	64	63	0
B	B	B	B

Other MSA instructions operate on adjacent odd/even source elements, generating results on data formats twice as wide. The signed doubleword dot product DOTP_S is such an instruction (see Figure 11.17):

```
dotp_s.d $w9,$w1,$w2
```

Note that the actual instruction specifies .D (doubleword) as the destination's data format. The data format of the source operands is inferred as being also signed and half the width, i.e. word, in this case.

Figure 11.17 Destination Vector W9 Value for DOTP_S Instruction

127 … 64	63 … 0
a * A + b * B	c * C + d * D

11.8 Mapping of Scalar Floating-Point Registers to MSA Vector Registers

The scalar floating-point unit (FPU) registers are mapped on the MSA vector registers. To facilitate register data sharing between scalar floating-point instructions and vector instructions, the FPU is required to use 64-bit floating-point registers operating in 64-bit mode.

More specifically, MSA instructions cannot be executed while the FPU (Coprocessor 1) is usable and operates in 32-bit mode. i.e. bit $Status_{CU1}$ (CP Register 12, Select 0, bit 29) is set. Note that $Status_{FR}$ (CP Register 12, Select 0, bit 26) is always set in the P6600 core.

When $Status_{FR}$ is set, the read and write operations for the FPU/MSA mapped floating-point registers are defined as follows:

- A read operation from the floating-point register *r,* where *r* = 0, …, 31, returns the value of the element with index 0 in the vector register *r*. The element's format is word for 32-bit (single precision floating-point) read or double for 64-bit (double precision floating-point) read.
- A 32-bit read operation from the high part of the floating-point register *r,* where *r* = 0, …, 31, returns the value of the word element with index 1 in the vector register *r*.
- A write operation of value *V* to the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the element with index 0 in the vector register *r* and writes 0 to all remaining elements. Figure 11-18 and Figure 11-19 show the vector register *r* after writing a 32-bit (single precision floating-point) and a 64-bit (double precision floating-point) value *V* to the floating-point register *r*.
- A 32-bit write operation of value *V* to the high part of the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the word element with index 1 in the vector register *r,* **preserves** word element 0, and writes 0 to all remaining elements. Figure 11-20 shows the vector register *r* after writing a 32-bit value *V* to the floating-point register *r*.

Changing the $Status_{FR}$ value renders all floating-point and vector registers **UNPREDICTABLE.**

Figure 11-18 FPU Word Write Effect on the MSA Vector Register ($Status_{FR}$ set)

127 … 96	95 … 64	63 … 32	31 … 0
0	0	0	Word value *V*

Figure 11-19 FPU Doubleword Write Effect on the MSA Vector Register ($Status_{FR}$ set)

127 … 64	63 … 0
0	Doubleword value *V*

Figure 11-20 FPU High Word Write Effect on the MSA Vector Register ($Status_{FR}$ set)

127 — 96	95 — 64	63 — 32	31 — 0
0	0	Word value *V*	**Unchanged**

11.9 Floating-Point General Registers

This section describes the organization and use of the Floating-Point general Registers (FPRs). The FPU is a 64-bit FPU. As such, the FR bit in the CP0 *Status* register is always 1. This selects the 64-bit register model, which defines thirty-two 64-bit registers with all formats supported in a register.

11.9.1 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the Floating-Point Register (FPR) that holds the value. Operands that are only 32 bits wide (*W* and *S* formats) use only half the space in an FPR.

Figures 11.21 and 11.22 show the FPR organization and the way that operand data is stored in them.

Figure 11.21 Single Floating-Point or Word Fixed-Point Operand in an FPR

	63 — 32	31 — 0
Reg 0	Undefined/Unused	Data Word

Figure 11.22 Double Floating-Point or Longword Fixed-Point Operand in an FPR

	63 — 0
Reg 0	Data Doubleword/Longword

11.9.2 Formats of Values Used in Floating Point Registers

Unlike the CPU, the FPU neither interprets the binary encoding of source operands nor produces a binary encoding of results for every operation. The value held in a floating-point operand register (FPR) has a format, or type, and it can be used only by instructions that operate on that format. The format of a value is either *uninterpreted*, *unknown*, or one of the valid numeric formats: *single* or *double* floating point, and *word* or *long* fixed point.

The value in an FPR is always set when a value is written to the register as follows:

- When a data transfer instruction writes binary data into an FPR (a load), the FPR receives a binary value that is *uninterpreted*.
- A computational or FP register move instruction that produces a result of type *fmt* puts a value of type *fmt* into the result register.

When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt*, and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted in a different format.

11.9.3 Binary Data Transfers (32-Bit and 64-Bit)

The data transfer instructions move words and doublewords between the FPU FPRs and the remainder of the system. The operations of the word and doubleword load and move-to instructions are shown in Figure 11.23 and Figure 11.24, respectively.

The store and move-from instructions operate in reverse, reading data from the location that the corresponding load or move-to instruction had written.

Figure 11.23 FPU Word Load and Move-to Operations

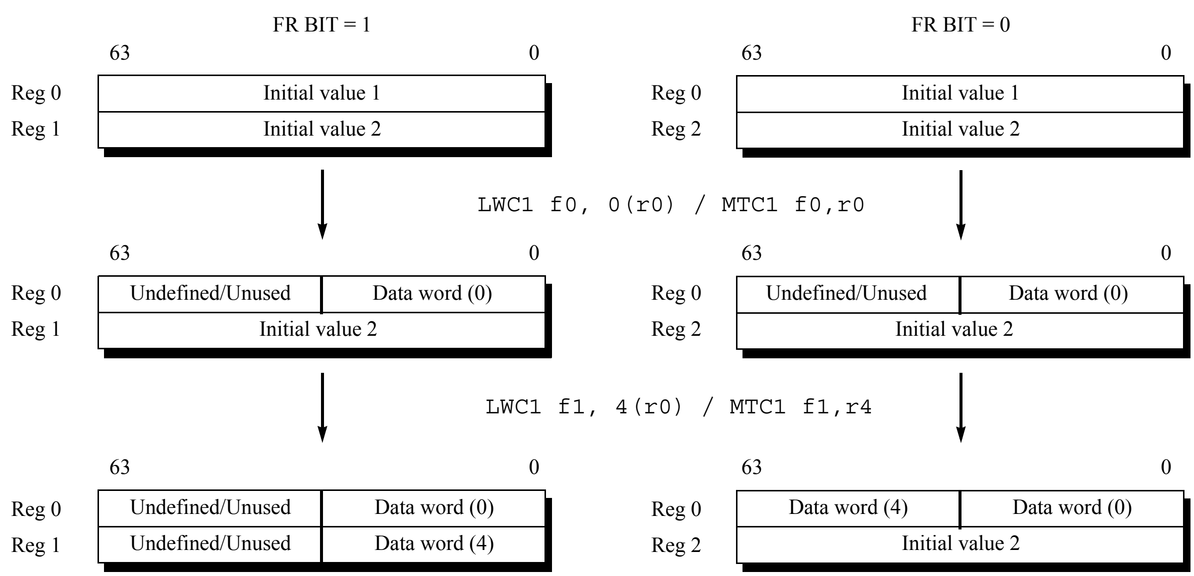
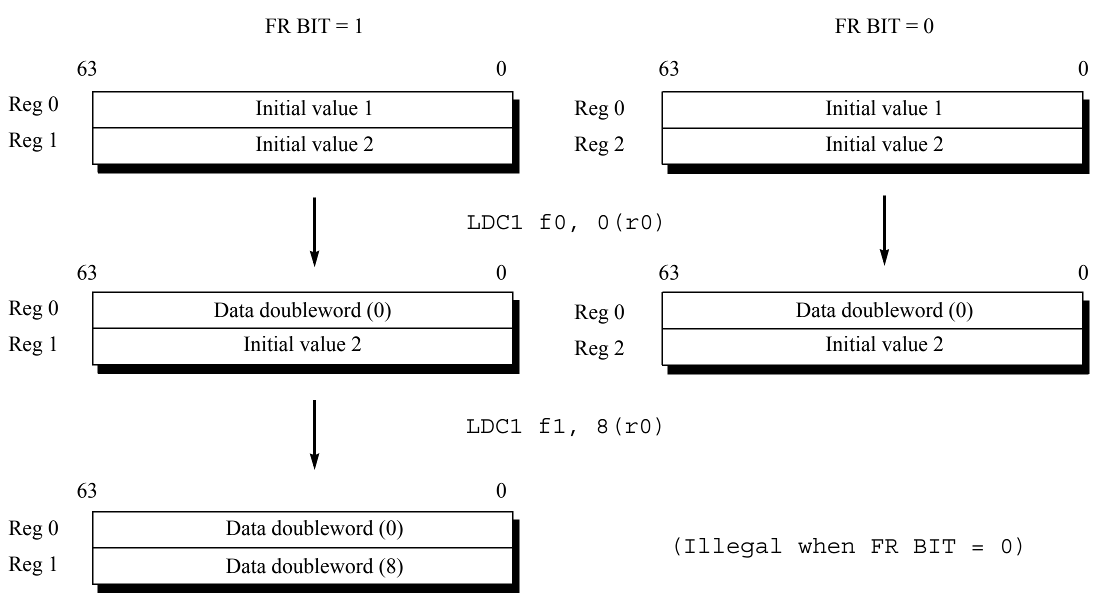


Figure 11.24 FPU Doubleword Load and Move-to Operations



11.10 Floating-Point Control Registers

The FPU Control Registers (FCRs) identify and control the FPU. The five FPU control registers are 32 bits wide: *FIR*, *FCCR*, *FEXR*, *FENR*, *FCSR*. Three of these registers, *FCCR*, *FEXR*, and *FENR*, select subsets of the floating-point Control/Status register, the *FCSR*. These registers are also denoted Coprocessor 1 (CP1) control registers.

CP1 control registers are summarized in Table 11.9 and are described individually in the following subsections of this chapter. Each register's description includes the read/write properties and the reset state of each field.

Table 11.9 Coprocessor 1 Register Summary

Register Number	Register Name	Function
0	FIR	Floating-Point Implementation register. Contains information that identifies the FPU.
1	UFR	User Floating-Point register mode control. The UFR register allows user mode to clear $Status_{FR}$ by executing a CTC1 to UFR with GPR[0] as input, and read $Status_{FR}$. by executing a CFC1 to UFR.
4	UNFR	User negated FP register mode control. The UNFR register allows user-mode to set StatusFR by executing a CTC1 to UNFR with GPR[0] as input. CTC1 to UNFR with any other input register is required to produce a Reserved Instruction Exception. User-mode software can determine presence of this feature from FIRUFRP.
25	FCCR	Floating-Point Condition Codes register.
26	FEXR	Floating-Point Exceptions register.
28	FENR	Floating-Point Enables register.
31	FCSR	Floating-Point Control and Status register.

Table 11.10 defines the notation used for the read/write properties of the register bit fields.

Table 11.10 Read/Write Properties

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	All bits in this field 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 returns a predictable value. This definition should not be confused with the formal definition of UNDEFINED behavior.	
R	This field 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 powerup. 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.

Table 11.10 Read/Write Properties

Read/Write Notation	Hardware Interpretation	Software Interpretation
0	Hardware does not update this field. Hardware can assume a zero value.	The value software writes to this field must be zero. Software writes of non-zero values to this field might 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.

11.10.1 Floating-Point Implementation Register (FIR, CP1 Control Register 0)

The Floating-Point Implementation Register (*FIR*) is a 32-bit read-only register that contains information identifying the capabilities of the FPU, the Floating-Point processor identification, and the revision level of the FPU. Figure 11.25 shows the format of the *FIR*; Table 11.11 describes the *FIR* bit fields.

Figure 11.25 FIR Format

31 30	29	28	27 24	23	22	21	20	19	18	17	16	15 8	7 0
0	FPRE	UFRP	0	Has2008	F64	L	W	3D	PS	D	S	ProcessorID	Revision

Table 11.11 FIR Register Bit Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:29	Reserved.	R	0
FPRE	29	User-mode access of *FRE* is supported. This bit is encoded as follows: 0: Support for emulation of $Status_{FR}$=0 handling on a 64-bit FPU with $Status_{FR}$=1 only is not available. 1: Support for emulation of $Status_{FR}$=0 handling on a 64-bit FPU with $Status_{FR}$=1 only is available. This bit is always '1' in the P6600 core. As such, the $Config5_{UFE}$ and $Config5_{FRE}$ bits are available, along with CFC1/CTC1, to allow user access to *FRE*. Note that this emulation facility is only available if an FPU is present ($Config1_{FP}$=1) and the FPU is 64-bit (FIR_{F64}=1). Note that in the P6600 FPU, the user can set $Status_{FR}$=0, but instead of implementing FR=0 mode, the core takes an exception for any instruction that would produce a different result between FR=0 and FR=1 mode.	R	1
UFRP	28	User mode FR switching. This bit is always 0 as User Mode FR switching is not supported in the P6600 core.	R	0
0	27:24	Reserved.	R	0

Table 11.11 FIR Register Bit Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
Has2008	23	Indicates that one or more IEEE-754-2008 features are implemented. This bit is always set in P6600 to indicate that the ABS2008 and NAN2008 bits within the FCSR register exist. For more information, refer to Section 11.10.4 "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R	1
F64	22	Indicates that this is a 64-bit FPU: • 0: Not a 64-bit FPU • 1: A 64-bit FPU. This bit is always 1 to indicate that this is a 64-bit FPU.	R	1
L	21	Indicates that the long fixed point (L) data type and instructions are implemented: • 0: Long type not implemented • 1: Long implemented This bit is always 1 to indicate that long fixed point data types are implemented.	R	1
W	20	Indicates that the word fixed point (W) data type and instructions are implemented: • 0: Word type not implemented • 1: Word implemented This bit is always 1 to indicate that word fixed point data types are implemented.	R	1
3D	19	Indicates if the MIPS-3D ASE is implemented. • 0: MIPS-3D not implemented • 1: MIPS-3D implemented This bit is always 0 in the P6600 core to indicate that the MIPS-3D ASE is not implemented.	R	0
PS	18	Indicates that the paired-single (PS) floating-point data type and instructions are implemented: • 0: PS floating-point not implemented • 1: PS floating-point implemented This bit is always 0 to indicate that paired-single floating-point data types are not implemented in the P6600 core.	R	0
D	17	Indicates that the double-precision (D) floating-point data type and instructions are implemented: • 0: D floating-point not implemented • 1: D floating-point implemented This bit is always 1 to indicate that double-precision floating-point data types are implemented.	R	1
S	16	Indicates that the single-precision (S) floating-point data type and instructions are implemented: • 0: S floating-point not implemented • 1: S floating-point implemented This bit is always 1 to indicate that single-precision floating-point data types are implemented.	R	1
Processor ID	15:8	Identifies the floating-point processor.	R	
Revision	7:0	Specifies the revision number of the FPU. This field allows software to distinguish between different revisions of the same floating-point processor type.	R	Hardwired

11.10.2 Floating-Point Exceptions Register (FEXR, CP1 Control Register 26)

The Floating-Point Exceptions Register (*FEXR*) is an alternative way to read and write the Cause and Flags fields that also appear in the *FCSR*. Figure 11.26 shows the format of the *FEXR*; Table 11.12 describes the *FEXR* bit fields.

Figure 11.26 FEXR Format

31–18	17	16	15	14	13	12	11–7	6	5	4	3	2	1–0
0	Cause						0	Flags					0
	E	V	Z	O	U	I		V	Z	O	U	I	

Table 11.12 FEXR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	31:18	These bits must be written as zeros; they return zeros on reads.	0	0
Cause	17:12	Cause bits. Refer to the description of this field in Section 11.10.4, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	11:7	These bits must be written as zeros; they return zeros on reads.	0	0
Flags	6:2	Flag bits. Refer to the description of this field in Section 11.10.4, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	1:0	These bits must be written as zeros; they return zeros on reads.	0	0

11.10.3 Floating-Point Enables Register (FENR, CP1 Control Register 28)

The Floating-Point Enables Register (*FENR*) is an alternative way to read and write the Enables, FS, and RM fields that also appear in the *FCSR*. Figure 11.27 shows the format of the *FENR*; Table 11.13 describes the *FENR* bit fields.

Figure 11.27 FENR Format

31–12	11	10	9	8	7	6–3	2	1–0
0	Enables					0	FS	RM
	V	Z	O	U	I			

Table 11.13 FENR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	31:12	These bits must be written as zeros; they return zeros on reads.	0	0
Enables	11:7	Enable bits. Refer to the description of this field in Section 11.10.4, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined

Table 11.13 FENR Bit Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
0	6:3	These bits must be written as zeros; they return zeros on reads.	0	0
FS	2	Flush to Zero bit. Refer to the description of this field in Section 11.10.4, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
RM	1:0	Rounding mode. Refer to the description of this field in Section 11.10.4, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined

11.10.4 Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)

The 32-bit Floating-Point Control and Status Register (*FCSR*) controls the operation of the FPU and shows the following status information:

- Selects the default rounding mode for FPU arithmetic operations
- Selectively enables traps of FPU exception conditions
- Controls some denormalized number handling options
- Reports any IEEE exceptions that arose during the most recently executed instruction
- Reports any IEEE exceptions that cumulatively arose in completed instructions
- Indicates the condition code result of FP compare instructions

Access to the *FCSR* is not privileged; it can be read or written by any program that has access to the FPU (via the coprocessor enables in the *Status* register). Figure 11.28 shows the format of the *FCSR*; Table 11.14 describes the *FCSR* bit fields.

Figure 11.28 FCSR Format

<table>
<tr><td>31 … 25</td><td>24</td><td>23 … 20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1 … 0</td></tr>
<tr><td>0</td><td>FS</td><td>0</td><td>ABS 2008</td><td>NAN 2008</td><td colspan="6">Cause</td><td colspan="5">Enables</td><td colspan="5">Flags</td><td>RM</td></tr>
<tr><td>0</td><td>U</td><td>0</td><td>1</td><td>1</td><td>E</td><td>V</td><td>Z</td><td>O</td><td>U</td><td>I</td><td>V</td><td>Z</td><td>O</td><td>U</td><td>I</td><td>V</td><td>Z</td><td>O</td><td>U</td><td>I</td><td></td></tr>
</table>

Table 11.14 FCSR Bit Field Descriptions

Fields				
Name	**Bit**	**Description**	**Read / Write**	**Reset State**
0	31:25	These bits must be written as zeros; they return zeros on reads.	0	0

Table 11.14 FCSR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bit**			
FS	24	Flush to Zero (FS). The FS bit controls the handling of denormalized operands and is encoded as follows: 0: IEEE-compliant mode. Input subnormal values and tiny non-zero results are not altered. 1: Regular embedded applications. When this bit is set, subnormal results are flushed to zero. In the P6600, every input subnormal value is replaced with zero of the same sign. Refer to Section 11.10.5 "Operation of the FS Bit" for more details on this bit.	R/W	Undefined
0	23:20	These bits must be written as zeros; they return zeros on reads.	0	0
ABS2008	19	ABS.fmt & NEG.fmt instructions compliant with IEEE Standard 754-2008. The IEEE 754-2008 standard requires that the ABS and NEG functions accept QNAN inputs without trapping. This bit is always set in the P6600 core to indicate support for the IEEE 754-2008 standard. 0: ABS & NEG trap for QNAN input 1: ABS & NEG accept QNAN input without trapping. IEEE 754-2008 behavior.	RO	1
NAN2008	18	Quiet and signaling NaN encodings recommended by the IEEE Standard 754-2008, i.e. a quiet NaN is encoded with the first bit of the fraction being 1 and a signaling NaN is encoded with the first bit of the fraction field being 0. In the P6600 core, this bit is always set to indicate support for the IEEE Standard 754-2008 encoding. 0: MIPS NaN encoding 1: IEEE 754-2008 NaN encoding	RO	1
Cause	17:12	Cause bits. These bits indicate the exception conditions that arise during execution of an FPU arithmetic instruction. A bit is set to 1 when the corresponding exception condition arises during the execution of an instruction; otherwise, it is cleared to 0. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 11.15 for the meaning of each cause bit.	R/W	Undefined
Enables	11:7	Enable bits. These bits control whether or not a trap is taken when an IEEE exception condition occurs for any of the five conditions. The trap occurs when both an enable bit and its corresponding cause bit are set either during an FPU arithmetic operation or by moving a value to the *FCSR* or one of its alternative representations. Note that Cause bit E (CauseE) has no corresponding enable bit; the MIPS architecture defines non-IEEE Unimplemented Operation exceptions as always enabled. Refer to Table 11.15 for the meaning of each enable bit.	R/W	Undefined

Table 11.14 FCSR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bit			
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for completed instructions since the flag was last reset by software. When an FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating-Point Exception (the enable bit was off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a Floating-Point Exception (the enable bit was on) do not update the Flags field. Hardware never resets this field; software must explicitly reset this field. Refer to Table 11.15 for the meaning of each flag bit.	R/W	Undefined
RM	1:0	Rounding mode. This field indicates the rounding mode used for most floating-point operations (some operations use a specific rounding mode). Refer to Table 11.16 for the encoding of this field.	R/W	Undefined

Table 11.15 Cause, Enable, and Flag Field Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation. This bit exists only in the Cause field.
V	Invalid Operation. The Invalid Operation Exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed. Under default exception handling, i.e. when the Invalid Operation Exception is not enabled, the default floating-point result is a quiet NaN (see Table 11.19).
Z	Divide by Zero. The Divide by Zero Exception is signaled if and only if an exact infinite result is defined for an operation on finite operands. Under default exception handling, i.e. when the Divide by Zero Exception is not enabled, the default result is an infinity correctly signed according to the operation (see Table 11.19).
O	Overflow. The Overflow Exception is signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. Under default exception handling, i.e. when the Overflow Exception is not enabled, the overflowed rounded result is delivered to the destination. In addition, the Inexact bit in the Cause field is set (see Table 11.19).

Table 11.15 Cause, Enable, and Flag Field Definitions

Bit Name	Bit Meaning
U	Underflow. If enabled, the Underflow Exception is signaled when a tiny non-zero result is detected after rounding regardless of whether the rounded result is exact or inexact. Under default exception handling, i.e. when the Underflow Exception is not enabled, the rounded result is delivered to the destination (see Table 11.19) and: • If the rounded result is inexact, the Inexact bit in the Cause field is set. • If the rounded result is exact, no bit in the Flags field is set. Such an underflow condition has no observable effect under default handling.
I	Inexact. Unless stated otherwise, if the rounded result of an operation is inexact -- that is, it differs from what would have been computed were both exponent range and precision unbounded -- then the Inexact Exception is be signaled. Under default exception handling, i.e. when the Inexact Exception is not enabled, the rounded result is delivered to the destination (see Table 11.19).

Table 11.16 Rounding Modes Definitions

RM Field Encoding	Meaning
0	Round to nearest / ties to even. Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even)
1	Round toward zero. Rounds the result to the value closest to but not greater in magnitude than the result.
2	Round towards positive / plus infinity. Rounds the result to the value closest to but not less than the result.
3	Round towards negative / minus infinity. Rounds the result to the value closest to but not greater than the result.

11.10.5 Operation of the FS Bit

Some floating point instructions might not handle subnormal input operands or compute tiny non-zero results. Such instructions may signal the Unimplemented Operation Exception and let the software emulation finalize the operation. If software emulation is not needed or desired, FS bit could be set to replace every tiny non-zero result and subnormal input operand with zero of the same sign.

The FS bit changes the behavior of the Unimplemented Operation Exception. All the other floating pointexceptions are signaled according to the new values of the operands or the results. In addition, when FS bit is set:

- Tiny non-zero results are detected before rounding1. Flushing of tiny non-zero results causes Inexact and Underflow Exceptions to be signaled for all instructions except the approximate reciprocals.
- Flushing of subnormal input operands in all instructions except comparisons causes Inexact Exception to be signaled.

- For floating-point comparisons, the Inexact Exception is not signaled when subnormal input operands are flushed.

11.11 MSA Control Registers

The control registers are used to record and manage the MSA state and resources. Two dedicated instructions are provided for this purpose: CFCMSA (Copy From Control MSA register) and CTCMSA (Copy To Control MSA register). The only information residing outside the MSA control registers is the implementation bit $Config3_{MSAP}$ and the enable bit $Config5_{MSAEn}$ discussed in Section 11.4 "Enabling MSA".

The P6600 core implements the following two MSA control registers.

- Section 11.11.1 "MSA Implementation Register (MSAIR, MSA Control Register 0)"
- Section 11.11.2 "MSA Control and Status Register (MSACSR, MSA Control Register 1)"

11.11.1 MSA Implementation Register (MSAIR, MSA Control Register 0)

The MSA Implementation Register (*MSAIR*) is a 32-bit read-only register that contains information specifying the identification of MSA. Figure 11.29 shows the format of the *MSAIR*; Table 11.17 describes the *MSAIR* fields.

The software can read the *MSAIR* using the CFCMSA (Copy From Control MSA register) instruction.

Figure 11.29 MSAIR Register Format

31 ... 17	16	15 ... 8	7 ... 0
0	WRP	ProcessorID	Revision

Table 11.17 MSAIR Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
0	31:17	Reserved for future use; reads as zero and must be written as zero.	R	0
WRP	16	Vector Registers Partitioning. Allows for multi-threaded implementations with fewer than 32 physical vector registers per hardware thread context. This bit is always 0 in the P6600 core since multi-threading is not supported.	R	0
ProcID	15:8	Processor ID number.	R	Preset
Rev	7:0	Revision number.	R	Preset

11.11.2 MSA Control and Status Register (MSACSR, MSA Control Register 1)

The MSA Control and Status Register (*MSACSR*) is a 32-bit read/write register that controls the operation of the MSA unit. Figure 11-30 shows the format of the *MSACSR*; Table 11.18 describes the *MSACSR* fields.

The software can read and write the *MSACSR* using the CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions.

The Floating Point Control and Status Register (*FCSR*, CP1 Control Register 31) and MSA Control and Status Register (*MSACSR*) are closely related in their purpose. However, each serves a different functional unit and can exist independently of the other.

Figure 11-30 MSACSR Register Format

31 25	24	23	22 21	20 19	18	17 12	11 7	6 2	1 0
0 00000000	FS	0	Impl	0	NX	Cause	Enables	Flags	RM
						E V Z O U I	V Z O U I	V Z O U I	

Table 11.18 MSACSR Register Field Descriptions

Fields: Name	Fields: Bits	Description	Read/ Write	Reset State
0	31:25	Reserved for future use; reads as zero and must be written as zero.	R0	0
FS	24	Flush to zero. If not implemented, reads as zero and writes are ignored. Every input subnormal value and tiny non-zero result is replaced with zero of the same sign. This bit is encoded as follows: 0: Input subnormal values and tiny non-zero results are not altered. Unimplemented Operation Exception may be signaled as needed. 1: Replace every input subnormal value and tiny non-zero result with zero of the same sign. No Unimplemented Operation Exception is signaled.	R/W	0
0	23	Reserved for future use; reads as zero and must be written as zero.	R0	0
Impl	22:21	Available to control implementation dependent features.	R/W	Undefined
0	20:19	Reserved for future use; reads as zero and must be written as zero.	R0	0

Table 11.18 MSACSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
NX	18	Non-trapping floating point exception mode. In normal exception mode, the destination register is not written and the floating point exceptions set the Cause bits and trap. In non-trapping exception mode, the operations which would normally signal floating point exceptions do not write the Cause bits and do not trap. All the destination register's elements are set either to the calculated results or, if the operation would normally signal an exception, to signaling NaN values with the least significant 6 bits recording the specific exception type detected for that element in the same format as the Cause field. The Flags bits are updated for all floating-point operation withan IEEE exception condition that does not result in a MSA floating point exception (i.e., the Enable bit is off). This bit is encoded as follows: 0: Normal exception mode 1: Non-trapping exception mode	R/W	0
Cause	17:12	Cause bits. These bits indicate the IEEE exception conditions that arise during the execution of all operations in a vector floating-point instruction. A bit is set to 1 if the corresponding exception condition arises during the execution of any operation in the vector floating-point instruction and is set to 0 otherwise. The exception conditions caused by the preceding vector floating-point instruction can be determined by reading the Cause field.For a definition of each bit in the Cause field, refer to Table 14.16, "Cause, Enable, and Flag Field Definitions".	R/W	Undefined
Enable	11:7	Enable bits. These bits control whether or not a exception is taken when an IEEE exception condition arises for any of the five conditions. The exception is taken when both an Enable bit and the corresponding Cause bit are set either during the execution of any operation in vector floating-point instruction or by moving a value to MSACSR or one of its alternative representations. Note that Cause bit E (Unimplemented Operation) has no corresponding Enable bit; the non-IEEE Unimplemented Operation Exception is defined by MIPS as always enabled. For a definition of each bit in the Enable field, refer to Table 14.16, "Cause, Enable, and Flag Field Definitions".	R/W	Undefined

Table 11.18 MSACSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for all operations in the vector floating-point instructions completed since the flag was last reset by software. When a floating-point operation raises an IEEE exception condition that does not result in a MSA floating point exception (i.e., the Enable bit is off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a floating point exception (i.e., the Enable bit is on) do not update the Flags bits.This field is never reset by hardware and must be explicitly reset by software. For a definition of each bit in the Flags field, refer to Table 14.16, "Cause, Enable, and Flag Field Definitions".	R/W	Undefined
RM	1:0	Rounding Mode. This field indicates the rounding mode used for most floating point operations (some operations use a specific rounding mode). For a definition of each bit in the RM field, refer to Table 14.17, "Rounding Modes Definitions".	R/W	0

11.12 Floating Point and MSA Exceptions

FPU exceptions are implemented in the MIPS FPU/MSA architecture with the Cause, Enables, and Flags fields of the *FCSR/MSACSR*. The flag bits implement IEEE exception status flags, and the cause and enable bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions. The Cause field has an additional exception bit, Unimplemented Operation, used to trap for software emulation assistance. If an exception type is enabled through the Enables field of the *FCSR/MSACSR*, then the FPU is operating in precise exception mode for this type of exception.

11.12.1 Precise Exception Mode

In precise exception mode, a trap occurs before the instruction that causes the trap or any following instruction can complete and write its results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The Cause field reports per-bit instruction exception conditions. The cause bits are written during each floating-point arithmetic operation to show any exception conditions that arise during the operation. A cause bit is set to 1 if its corresponding exception condition arises; otherwise, it is cleared to 0.

A floating-point trap is generated any time both a cause bit and its corresponding enable bit are set. This case occurs either during the execution of a floating-point operation or when moving a value into the *FCSR/MSACSR*. There is no enable bit for Unimplemented Operations; this exception always generates a trap.

In a trap handler, exception conditions that arise during any trapped floating-point operations are reported in the Cause field. Before returning from a floating-point interrupt or exception, or before setting cause bits with a move to the *FCSR*, software first must clear the enabled cause bits by executing a move to the *FCSR/MSACSR* to prevent the trap from being erroneously retaken.

If a floating-point operation sets only non-enabled cause bits, no trap occurs and the default result defined by IEEE Standard 754 is stored. When a floating-point operation does not trap, the program can monitor the exception conditions by reading the Cause field.

The Flags field is a cumulative report of IEEE exception conditions that arise as instructions complete; instructions that trap do not update the flag bits. The flag bits are set to 1 if the corresponding IEEE exception is raised, otherwise the bits are unchanged. There is no flag bit for the MIPS Unimplemented Operation exception. The flag bits are never cleared as a side effect of floating-point operations, but they can be set or cleared by moving a new value into the *FCSR*.

11.12.2 Exception Conditions

The subsections below describe the following five exception conditions defined by IEEE Standard 754:

- Section 11.12.2.1 “Invalid Operation Exception”
- Section 11.12.2.2 “Division By Zero Exception”
- Section 11.12.2.3 “Underflow Exception”
- Section 11.12.2.4 “Overflow Exception”
- Section 11.12.2.5 “Inexact Exception”
- Section 11.12.2.6 “Unimplemented Operation Exception”

11.12.2.1 Invalid Operation Exception

An Invalid Operation exception is signaled when one or both of the operands are invalid for the operation to be performed. When the exception condition occurs without a precise trap, the result is a quiet NaN.

The following operations are invalid:

- One or both operands are a signaling NaN (except for the non-arithmetic MOV.fmt, MOVT.fmt, MOVF.fmt, MOVN.fmt, and MOVZ.fmt instructions).
- Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$.
- Multiplication: $0 \times \infty$, with any signs.
- Division: 0/0 or ∞/∞, with any signs.
- Square root: An operand of less than 0 (-0 is a valid operand value).
- Conversion of a floating-point number to a fixed-point format when either an overflow or an operand value of infinity or NaN precludes a faithful representation in that format.
- Some comparison operations in which one or both of the operands is a QNaN value.

11.12.2.2 Division By Zero Exception

The divide operation signals a Division By Zero exception if the divisor is zero and the dividend is a finite nonzero number. When no precise trap occurs, the result is a correctly signed infinity. Divisions (0/0 and ∞/0) do not cause the Division By Zero exception. The result of (0/0) is an Invalid Operation exception. The result of (∞/0) is a correctly signed infinity.

11.12.2.3 Underflow Exception

Two related events contribute to underflow:

- Tininess: The creation of a tiny, nonzero result between $\pm 2^{E\_min}$ which, because it is tiny, might cause some other exception later such as overflow on division. IEEE Standard 754 allows choices in detecting tininess events. The MIPS architecture specifies that tininess be detected after rounding, when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E\_min}$.
- Loss of accuracy: The extraordinary loss of accuracy occurs during the approximation of such tiny numbers by denormalized numbers. IEEE Standard 754 allows choices in detecting loss of accuracy events. The MIPS architecture specifies that loss of accuracy be detected as inexact result, when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded.

The way that an underflow is signaled depends on whether or not underflow traps are enabled:

- When an underflow trap is not enabled, underflow is signaled only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or $\pm 2^{E\_min}$.
- When an underflow trap is enabled (through the *FCSR/MSACSR* Enables field), underflow is signaled when tininess is detected regardless of loss of accuracy.

11.12.2.4 Overflow Exception

An Overflow exception is signaled when the magnitude of a rounded floating-point result (if the exponent range is unbounded) is larger than the destination format's largest finite number.

When no precise trap occurs, the result is determined by the rounding mode and the sign of the intermediate result.

11.12.2.5 Inexact Exception

An Inexact exception is signaled when one of the following occurs:

- The rounded result of an operation is not exact.
- The rounded result of an operation overflows without an overflow trap.
- When a denormal operand is flushed to zero.

11.12.2.6 Unimplemented Operation Exception

The Unimplemented Operation exception is a MIPS-defined exception that provides software emulation support. This exception is not IEEE-compliant and is used to signal a need for software emulation of an instruction. Normally an IEEE arithmetic operation can cause only one exception condition; the only case in which two exceptions can occur at the same time are *Inexact With Overflow* and *Inexact With Underflow.*

The MIPS architecture is designed so that a combination of hardware and software can implement the architecture. Operations not fully supported in hardware cause an Unimplemented Operation exception, allowing software to perform the operation.

There is no enable bit for this condition; it always causes a trap (but the condition is effectively masked for all operations when FS=1). After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

An Unimplemented Operation exception is taken in the following situations:

- when denormalized operands or tiny results are encountered for instructions not supporting denormal numbers and where such are not handled by the FS bit.

11.12.3 Floating Point Exceptions

At the program's direction, an IEEE exception condition can either cause a trap or not cause a trap. IEEE Standard 754 specifies the result to be delivered in case no trap is taken. The FPU supplies these results whenever the exception condition does not result in a trap. The default action taken depends on the type of exception condition and, in the case of the Overflow and Underflow, the current rounding mode. Table 11.19 summarizes the default results.

11.12.3.1 MSA Non-Trapping Exceptions

MSA provides a non-trapping exception mode (bit NX) that enables determining which element in the MSA vector caused the floating point exception.

In normal operation mode, floating point exceptions are signaled if at least one vector element causes an exception enabled by the Enable bit-field. There is no precise indication in this case on which elements are at fault and the corresponding exception causes. The exception handling routine should set the non-trapping exception mode bit NX and re-execute the MSA floating point instruction. All elements which would normally signal an exception according to the Enable bit-field are set to signaling NaN values, where the least significant 6 bits have the same format as the Cause field (see Figure 11-31, Table 11.15) to record the specific exception or exceptions detected for that element. The other elements will be set to the calculated results based on their operands.

Figure 11-31 Output Format for Faulting Elements when NX is Set

...	6	5	4	3	2	1	0
Signaling NaN Bits		Cause					
		E	V	Z	O	U	I

When the non-trapping exception mode bit NX is set, no floating point exception will be taken, not even the always enabled Unimplemented Operation Exception. Note that by setting the NX bit, the *MSACSR* Enable bitfield is not changed and is still used to generate the appropriate default results. Regardless of the NX value, if a floating point exception is not enabled, i.e. the corresponding *MSACSR* Enable bit is 0, the floating point result is a default value as shown in Table 11.19.

11.12.3.2 Floating Point Exception Defaults

Table 11.19 shows each type of MSA floating point exception and the corresponding default value.

Table 11.19 Default Values for Floating Point Exceptions

Exception	Rounding Mode	Default Value, Disabled Exception	Default Value, Enabled Exception, and NX set
Invalid Operation		The default value is either the default quiet NaN (see Table 11.20), or one of the signaling NaN operands propagated as a quiet NaN.	The default signaling NaN (see Table 11.20) of the format shown in Figure 11-31 with Cause V bit set.
Divide by Zero		The default value is the properly signed infinity.	The default signaling NaN (see Table 11.20) of the format shown in Figure 11-31 with Cause Z bit set.
Underflow		The default value is the rounded result based on the rounding mode.	The default signaling NaN (see Table 11.20) of the format shown in Figure 11-31 with Cause U bit set.
Inexact		The default value is the rounded result based on the rounding mode. If caused by an overflow without the overflow exception enabled, the default value is the overflowed result.	The default signaling NaN (see Table 11.20) of the format shown in Figure 11-31 with Cause I bit set.
Overflow		The default value depends on the rounding mode, as shown below.	The default signaling NaN (see Table 11.20) of the format shown in Figure 11-31 with Cause O bit set.
	Round to nearest	An infinity with the sign of the overflow value.	
	Round toward zero	The format's largest finite number with the sign of the overflow value.	
	Round towards positive	For positive overflow values, positive infinity. For negative overflow values, the format's smallest negative finite number.	
	Round towards negative	For positive overflow values, the format's largest finite number. For negative overflow values, minus infinity.	

Table 11.20 Default NaN Encodings

Format	Quiet NaN	Signaling NaN
16-bit	`0x7E00`	`0x7CNN`[1]
32-bit	`0x7FC0 0000`	`0x7F80 00NN`
64-bit	`0x7FF8 0000 0000 0000`	`0x7FF0 0000 0000 00NN`

1. All signaling NaN values have the format shown in Figure 11-31. Byte `0xNN` has at least one bit set showing the reason for generating the signaling NaN value.

11.12.3.3 MSACSR Cause Register Update Pseudocode

The pseudocode below shows the process of updating the *MSACSR* Cause bits and setting the destination's value. This process is invoked element-by-element for all elements the instruction operates on. It is assumed *MSACSR* Cause bits are all cleared before executing the instruction. The *MSACSR* Flags bits are updated after all the elements have been processed and *MSACSR* Cause contains no enabled exceptions. If there are enabled exceptions in *MSACSR* Cause, a MSA floating-point exception will be signaled and the *MSACSR* flags are not updated. The pseudocode below describes the *MSACSR* Flags update and exception signaling condition.

For instructions with non floating-point results, the pseudocode the apply unchanged and both the format in Figure 11-31 and the default values from Table 11.19 are preserved for enabled exceptions when NX bit is set. For disabled exceptions, the default values are explicitly documented case-by-case in the instruction's description section.

***MSACSR$_{Cause}$* Update Pseudocode**

```
Input
    c: current element exception(s) E, V, Z, O, U, I bitfield
       (bit E is 0x20, O is 0x04, U is 0x02, and I is 0x01)
    d: default value to be used in case of a disabled exception
    e: signaling NaN value to be used in case of NX set, i.e. a non-trapping
       exception
    r: result value if the operation completed without an exception

Output
    v: value to be written to destination element
    Updated MSACSR_Cause

enable ← MSACSR_Enable | E /* Unimplemented (E) is always enabled */

/* Set Inexact (I) when Overflow (O) is not enabled (see Table 11.15) */
if (c & O) … 0 and (enable & O) = 0 then
    c ← c | I
endif

/* Clear Exact Underflow when Underflow (U) is not enabled (see Table 11.15) */
if (c & U) … 0 and (enable & U) = 0 and (c & I) = 0 then
    c ← c ^ U
endif

cause ← c & enable

if cause = 0 then
```

```
    /* No enabled exceptions, update the MSACSR Cause with all current exceptions */
    MSACSR_Cause ← MSACSR_Cause | c

    if c = 0 then
        /* Operation completed successfully, destination gets the result */
        v ← r
    else
        /* Current exceptions are not enabled, destination
            gets the default value for disabled exceptions case */
        v ← d
    endif
else
    /* Current exceptions are enabled */
    if MSACSR_NX = 0 then
        /* Exceptions will trap, update MSACSR Cause with all current exceptions,
            destination is not written */
        MSACSR_Cause ← MSACSR_Cause | c
    else
        /* No trap on exceptions, element not recorded in MSACSR Cause,
            destination gets the signaling NaN value for non-trapping exception */
        v ← ((e >> 6) << 6) | c
    endif
endif
```

$MSACSR_{Flags}$ Update and Exception Signaling Pseudocode

```
if (MSACSR_Cause & (MSACSR_Enable | E)) = 0 then /* Unimplemented (bit E 0x20)
                                                     is always enabled */
    /* No enabled exceptions, update the MSACSR Flags with all exceptions */
    MSACSR_Flags ← MSACSR_Flags | MSACSR_Cause
else
    /* Trap on the exceptions recorded in MSACSR Cause,
        MSACSR Flags are not updated */
    SignalException(MSAFPE, MSACSR_Cause)
```

11.13 Floating Point Instruction Overview

The functional groups into which the FPU instructions are divided are described in the following subsections:

- Section 11.13.1 "Data Transfer Instructions"
- Section 11.13.2 "Arithmetic Instructions"
- Section 11.13.3 "Conversion Instructions"
- Section 11.13.4 "Coprocessor 1 Branch Instructions"
- Section 11.13.5 "Miscellaneous Instructions"

11.13.1 Data Transfer Instructions

The FPU has two separate register sets: floating point coprocessor general registers (FPRs) and floating point coprocessor control registers (FCRs). The FPU has a load/store architecture; all computations are done on data held in coprocessor general registers. The control registers are used to control FPU operation. Data is transferred between registers and the rest of the system withdedicated load, store, and move instructions. The transferred data is treated as unformatted binary data; no format conversions are performed, and therefore no IEEE floating-point exceptions can occur.

Table 11.21 lists the supported transfer operations.

Table 11.21 FPU Data Transfer Instructions

Transfer Direction			Data Transferred
FPU general register	↔	Memory	Word/doubleword load/store
FPU general register	↔	CPU general register	Word/Doubleword move
FPU control register	↔	CPU general register	Word move

11.13.1.1 Data Alignment in Loads, Stores, and Moves

The P6600 core supports misaligned loads and stores as well as bonded loads and stores. Regardless of byte ordering (the endianness), the address of a word or doubleword is the smallest byte address in the object. For a big-endian machine, this is the most-significant byte; for a little-endian machine, this is the least-significant byte.

11.13.1.2 Addressing Used in Data Transfer Instructions

The FPU has loads and stores using the same register+offset addressing as that used by the CPU. Moreover, for the FPU only, there are load and store instructions using *register+register* addressing.

Tables 11.22 and 11.23 list the FPU data transfer instructions.

Table 11.22 FPU Loads and Stores Using Register+Offset Address Mode

Mnemonic	Instruction
LDC1	Load Doubleword to Floating Point
LWC1	Load Word to Floating Point
SDC1	Store Doubleword to Floating Point
SWC1	Store Word to Floating Point

Table 11.23 FPU Move To and From Instructions

Mnemonic	Instruction
CFC1	Move Control Word From Floating Point
CTC1	Move Control Word To Floating Point
MFC1	Move Word From Floating Point
MTC1	Move Word To Floating Point

11.13.2 Arithmetic Instructions

Arithmetic instructions operate on formatted data values. The results of most floating-point arithmetic operations meet IEEE Standard 754 for accuracy—a result is identical to an infinite-precision result that has been rounded to the specified format using the current rounding mode. The rounded result differs from the exact result by less than one Unit in the Least-significant Place (ULP).

Table 11.24 lists the FPU IEEE compliant arithmetic operations.

Table 11.24 FPU IEEE Arithmetic Operations

Mnemonic	Instruction
CLASS.fmt	Floating-Point Class Mask
CMP.cond.fmt	Floating-Point Conditional Compare
MADDF.fmt	Floating-Point Fused Multiply-Add
MAX.fmt	Floating-Point argument with Maximum Absolute Value
MAX_A.fmt	Floating-Point argument with Minimum Absolute Value
MIN.fmt	Floating-Point Maximum
MIN_A.fmt	Floating-Point Minimum
MSUBF.fmt	Floating-Point Fused Multiply-Subtract

There are four iterative FP instructions. Table 11.25 lists the FPU-approximate arithmetic operations.

Table 11.25 FPU-Approximate Arithmetic Operations

Mnemonic	Instruction
DIV.fmt	Floating-Point Divide
RECIP.fmt	Floating-Point Reciprocal Approximation
RSQRT.fmt	Floating-Point Reciprocal Square Root Approximation
SQRT.fmt	Floating-Point Square Root Approximation

The result of DIV, SQRT,RECIP are accurate as IEEE specification. The result of RSQRT differs from reciprocal square root by no more than one ULP.

11.13.3 Conversion Instructions

These instructions perform conversions between floating-point and fixed-point data types. Each instruction converts values from a number of operand formats to a particular result format. Some conversion instructions use the rounding mode specified in the Floating Control/Status register (*FCSR*), while others specify the rounding mode directly.

Table 11.26 and Table 11.27 list the FPU conversion instructions according to their rounding mode.

Table 11.26 FPU Conversion Operations Using the FCSR Rounding Mode

Mnemonic	Instruction
CVT.D.fmt	Floating-Point Convert to Double Floating Point
CVT.L.fmt	Floating-Point Convert to Long Fixed Point
CVT.S.fmt	Floating-Point Convert to Single Floating Point
CVT.W.fmt	Floating-Point Convert to Word Fixed Point
RINT.fmt	Scalar Floating-Point Round to Integral Floating Point Value

Table 11.27 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Instruction
CEIL.L.fmt	Floating-Point Ceiling to Long Fixed Point
CEIL.W.fmt	Floating-Point Ceiling to Word Fixed Point
FLOOR.L.fmt	Floating-Point Floor to Long Fixed Point
FLOOR.W.fmt	Floating-Point Floor to Word Fixed Point
ROUND.L.fmt	Floating-Point Round to Long Fixed Point
ROUND.W.fmt	Floating-Point Round to Word Fixed Point
TRUNC.L.fmt	Floating-Point Truncate to Long Fixed Point
TRUNC.W.fmt	Floating-Point Truncate to Word Fixed Point

11.13.4 Coprocessor 1 Branch Instructions

The P6600 MIPSR6 core contains two new branch instruction that branch on coprocessor 1 based on the state of the FPU and FPR register bits. These instructions are shown in Table 11.28 list the formatted operand-value move instructions.

Table 11.28 Coprocessor 1 Branch Instructions

Mnemonic	Instruction
BC1EQZ	Branch if Coprocessor 1 (FPU) register bit 1 is equal to zero.
BC1NQZ	Branch if Coprocessor 1 (FPU) register bit 1 is NOT equal to zero.

11.13.5 Miscellaneous Instructions

The MIPS64 architecture defines various miscellaneous instructions that conditionally move one CPU general register to another, based on an FPU condition code.

Table 11.29 lists these miscellaneous instructions.

Table 11.29 Miscellaneous Floating Point Select Instructions

Mnemonic	Instruction
SEL.fmt	Select Floating Point Values with FPR Condition
SELEQZ.fmt	Select Floating Point Values or Zero with FPR Condition
SELNEZ.fmt	Select Floating Point Values or Not Zero with FPR Condition

11.14 MSA Instruction Descriptions

The MSA implements simple, homogeneous instructions with explicit functionality. There are no mixed general purpose and vector register operations except for data movement. This simplifies the hardware implementation, and allows for faster and independent execution of scalar and vector instructions.

In the MSA, complex operations that can be implemented by a sequence of two or three existing instructions are not implemented as single instructions. This could increase the code size to some extent, but greatly benefits the execution speed. For example, MSA has no instructions for horizontal arithmetic operations between all elements in the same vector register because these are complex operations easily implemented with few additional element shuffle instructions.

Most MSA instructions operate vector-element-by-vector-element in a typical SIMD manner. Few instructions handle the operands as bit vectors, because the elements don't make sense (e.g., bitwise logical operations). For certain instructions, the source operand could be a scalar immediate value or a vector element selected by an immediate index. The scalar value is being replicated for all vector elements.

The MSA instruction set implements the following categories of instructions: arithmetic, bitwise, floating-point arithmetic, floating-point compare, floating-point conversions, fixed-point multiplication, branch and compare, load/store, element move, and element shuffle.

Each instruction category is briefly described in the following subsections.

11.14.1 Arithmetic Instructions

Arithmetic instructions (Table 11.30) include additions and subtractions combined with saturation and absolute value operations. There is also a dedicated saturation instruction for arbitrary clamping at any bit position. Average computing instructions are provided for full precision (i.e. no wrap-around on overflow) add and shift with or without rounding. Minimum and maximum value selection instructions work on signed, unsigned, and absolute values.

Addition, subtraction, minimum, and maximum instructions also take a small, 5-bit constant value to operate across all elements.

Multiply, multiply-add/sub, divide, and remainder (modulo) are defined with operands and results of the same size ranging from bytes to doublewords. A set of dot product instructions perform partitioned multiplication with reduction: essentially a multiply-add or sub on adjacent elements, with the full-precision result double the size (see the example Figure 11.17).

Bitwise instructions (Table 11.31) include logical (e.g., AND, OR, NOR, and XOR) operations and shifts. All operate on two vector registers or on a vector register and an immediate constant. More complex logical instructions do selective bit copy from two source vectors to the destination. Leading zero/one bit counting and population counting (all one bits) instructions are available as well.

Table 11.30 MSA Arithmetic Instructions

Mnemonic	Compiler Intrinsics	C Expression	Instruction Description

ADDV	`wi8_t  msa_addv(wi8_t, wi8_t )`
	`wi16_t msa_addv(wi16_t, wi16_t)`
	`wi32_t msa_addv(wi32_t, wi32_t)`
	`wi64_t msa_addv(wi64_t, wi64_t)`
		`a + b`	Add

ADDVI	`wi8_t  msa_addvi(wi8_t, unsigned char)`
	`wi16_t msa_addvi(wi16_t, unsigned char)`
	`wi32_t msa_addvi(wi32_t, unsigned char)`
	`wi64_t msa_addvi(wi64_t, unsigned char)`
		`a + b`	Add Immediate

ADD_A	`wi8_t msa_add_a(wi8_t, wi8_t)`
	`wi16_t msa_add_a(wi16_t, wi16_t)`
	`wi32_t msa_add_a(wi32_t, wi32_t)`
	`wi64_t msa_add_a(wi64_t, wi64_t)`
			Add Absolute Values

ADDS_A	`wi8_t msa_adds_a(wi8_t, wi8_t)`
	`wi16_t msa_adds_a(wi16_t, wi16_t)`
	`wi32_t msa_adds_a(wi32_t, wi32_t)`
	`wi64_t msa_adds_a(wi64_t, wi64_t)`
			Saturated Add Absolute Values

ADDS_S	`wi8_t msa_adds_s(wi8_t, wi8_t)`
	`wi16_t msa_adds_s(wi16_t, wi16_t)`
	`wi32_t msa_adds_s(wi32_t, wi32_t)`
	`wi64_t msa_adds_s(wi64_t, wi64_t)`
			Signed Saturated Add

ADDS_U	`wu8_t msa_adds_u(wu8_t, wu8_t)`
	`wu16_t msa_adds_u(wu16_t, wu16_t)`
	`wu32_t msa_adds_u(wu32_t, wu32_t)`
	`wu64_t msa_adds_u(wu64_t, wu64_t)`
			Unsigned Saturated Add

HADD_S	`wi16_t msa_hadd_s(wi8_t, wi8_t)`
	`wi32_t msa_hadd_s(wi16_t, wi16_t)`
	`wi64_t msa_hadd_s(wi32_t, wi32_t)`
			Signed Horizontal Add

HADD_U	`wu16_t msa_hadd_u(wu8_t, wu8_t)`
	`wu32_t msa_hadd_u(wu16_t, wu16_t)`
	`wu64_t msa_hadd_u(wu32_t, wu32_t)`
			Unsigned Horizontal Add

ASUB_S	`wi8_t msa_asub_s(wi8_t, wi8_t)`
	`wi16_t msa_asub_s(wi16_t, wi16_t)`
	`wi32_t msa_asub_s(wi32_t, wi32_t)`
	`wi64_t msa_asub_s(wi64_t, wi64_t)`
			Absolute Value of Signed Subtract

ASUB_U	`wu8_t msa_asub_u(wu8_t, wu8_t)`
	`wu16_t msa_asub_u(wu16_t, wu16_t)`
	`wu32_t msa_asub_u(wu32_t, wu32_t)`
	`wu64_t msa_asub_u(wu64_t, wu64_t)`
			Absolute Value of Unsigned Subtract

AVE_S	`wi8_t msa_ave_s(wi8_t, wi8_t)`
	`wi16_t msa_ave_s(wi16_t, wi16_t)`
	`wi32_t msa_ave_s(wi32_t, wi32_t)`
	`wi64_t msa_ave_s(wi64_t, wi64_t)`
		`(a + b) / 2`	Signed Average

Table 11.30 MSA Arithmetic Instructions *(continued)*

Mnemonic	Compiler Intrinsics	C Expression	Instruction Description

AVE_U	`wu8_t msa_ave_u(wu8_t, wu8_t)`
	`wu16_t msa_ave_u(wu16_t, wu16_t)`
	`wu32_t msa_ave_u(wu32_t, wu32_t)`
	`wu64_t msa_ave_u(wu64_t, wu64_t)`
		`(a + b) / 2`	Unsigned Average

AVER_S	`wi8_t msa_aver_s(wi8_t, wi8_t)`
	`wi16_t msa_aver_s(wi16_t, wi16_t)`
	`wi32_t msa_aver_s(wi32_t, wi32_t)`
	`wi64_t msa_aver_s(wi64_t, wi64_t)`
		`(a + b + 1) / 2`	Signed Average with Rounding

AVER_U	`wu8_t msa_aver_u(wu8_t, wu8_t)`
	`wu16_t msa_aver_u(wu16_t, wu16_t)`
	`wu32_t msa_aver_u(wu32_t, wu32_t)`
	`wu64_t msa_aver_u(wu64_t, wu64_t)`
		`(a + b + 1) / 2`	Unsigned Average with Rounding

DOTP_S	`wi16_t msa_dotp_s(wi8_t, wi8_t)`
	`wi32_t msa_dotp_s(wi16_t, wi16_t)`
	`wi64_t msa_dotp_s(wi32_t, wi32_t)`
			Signed Dot Product

DOTP_U	`wu16_t msa_dotp_u(wu8_t, wu8_t)`
	`wu32_t msa_dotp_u(wu16_t, wu16_t)`
	`wu64_t msa_dotp_u(wu32_t, wu32_t)`
			Unsigned Dot Product

DPADD_S	`wi16_t msa_dpadd_s(wi16_t, wi8_t, wi8_t)`
	`wi32_t msa_dpadd_s(wi32_t, wi16_t, wi16_t)`
	`wi64_t msa_dpadd_s(wi64_t, wi32_t, wi32_t)`
			Signed Dot Product Add

DPADD_U	`wu16_t msa_dpadd_u(wu16_t, wu8_t, wu8_t)`
	`wu32_t msa_dpadd_u(wu32_t, wu16_t, wu16_t)`
	`wu64_t msa_dpadd_u(wu64_t, wu32_t, wu32_t)`
			Unsigned Dot Product Add

DPSUB_S	`wi16_t msa_dpsub_s(wi16_t, wi8_t, wi8_t)`
	`wi32_t msa_dpsub_s(wi32_t, wi16_t, wi16_t)`
	`wi64_t msa_dpsub_s(wi64_t, wi32_t, wi32_t)`
			Signed Dot Product Sub-tract

DPSUB_U	`wi16_t msa_dpsub_u(wi16_t, wu8_t, wu8_t)`
	`wi32_t msa_dpsub_u(wi32_t, wu16_t, wu16_t)`
	`wi64_t msa_dpsub_u(wi64_t, wu32_t, wu32_t)`
			Unsigned Dot Product Sub-tract

DIV_S	`wi8_t msa_div_s(wi8_t, wi8_t)`
	`wi16_t msa_div_s(wi16_t, wi16_t)`
	`wi32_t msa_div_s(wi32_t, wi32_t)`
	`wi64_t msa_div_s(wi64_t, wi64_t)`
		`a / b`	Signed Divide

DIV_U	`wu8_t msa_div_u(wu8_t, wu8_t)`
	`wu16_t msa_div_u(wu16_t, wu16_t)`
	`wu32_t msa_div_u(wu32_t, wu32_t)`
	`wu64_t msa_div_u(wu64_t, wu64_t)`
		`a / b`	Unsigned Divide

MADDV	`wi8_t msa_maddv(wi8_t, wi8_t, wi8_t)`
	`wi16_t msa_maddv(wi16_t, wi16_t, wi16_t)`
	`wi32_t msa_maddv(wi32_t, wi32_t, wi32_t)`
	`wi64_t msa_maddv(wi64_t, wi64_t, wi64_t)`
		`a + b * c`	Multiply-Add

Table 11.30 MSA Arithmetic Instructions *(continued)*

Mnemonic	Compiler Intrinsics	C Expression	Instruction Description
MAX_A	wi8_t msa_max_a(wi8_t, wi8_t) wi16_t msa_max_a(wi16_t, wi16_t) wi32_t msa_max_a(wi32_t, wi32_t) wi64_t msa_max_a(wi64_t, wi64_t)		Maximum of Absolute Values
MIN_A	wi8_t msa_min_a(wi8_t, wi8_t) wi16_t msa_min_a(wi16_t, wi16_t) wi32_t msa_min_a(wi32_t, wi32_t) wi64_t msa_min_a(wi64_t, wi64_t)		Minimum of Absolute Values
MAX_S	wi8_t msa_max_s(wi8_t, wi8_t) wi16_t msa_max_s(wi16_t, wi16_t) wi32_t msa_max_s(wi32_t, wi32_t) wi64_t msa_max_s(wi64_t, wi64_t)	a > b ? a : b	Signed Maximum
MAXI_S	wi8_t msa_maxi_s(wi8_t, char) wi16_t msa_maxi_s(wi16_t, char) wi32_t msa_maxi_s(wi32_t, char) wi64_t msa_maxi_s(wi64_t, char)	a > b ? a : b	Signed Immediate Maximum
MAX_U	wi8_t msa_max_u(wi8_t, wi8_t) wi16_t msa_max_u(wi16_t, wi16_t) wi32_t msa_max_u(wi32_t, wi32_t) wi64_t msa_max_u(wi64_t, wi64_t)	a > b ? a : b	Unsigned Maximum
MAXI_U	wu8_t msa_maxi_u(wu8_t, unsigned char) wu16_t msa_maxi_u(wu16_t, unsigned char) wu32_t msa_maxi_u(wu32_t, unsigned char) wu64_t msa_maxi_u(wu64_t, unsigned char)	a > b ? a : b	Unsigned Immediate Maximum
MIN_S	wi8_t msa_min_s(wi8_t, wi8_t) wi16_t msa_min_s(wi16_t, wi16_t) wi32_t msa_min_s(wi32_t, wi32_t) wi64_t msa_min_s(wi64_t, wi64_t)	a < b ? a : b	Signed Maximum
MINI_S	wi8_t msa_mini_s(wi8_t, char) wi16_t msa_mini_s(wi16_t, char) wi32_t msa_mini_s(wi32_t, char) wi64_t msa_mini_s(wi64_t, char)	a < b ? a : b	Signed Immediate Maximum
MIN_U	wu8_t msa_min_u(wu8_t, wu8_t) wu16_t msa_min_u(wu16_t, wu16_t) wu32_t msa_min_u(wu32_t, wu32_t) wu64_t msa_min_u(wu64_t, wu64_t)	a < b ? a : b	Unsigned Maximum
MINI_U	wu8_t msa_mini_u(wu8_t, unsigned char) wu16_t msa_mini_u(wu16_t, unsigned char) wu32_t msa_mini_u(wu32_t, unsigned char) wu64_t msa_mini_u(wu64_t, unsigned char)	a < b ? a : b	Unsigned Immediate Maximum
MSUBV	wi8_t msa_msubv(wi8_t, wi8_t, wi8_t) wi16_t msa_msubv(wi16_t, wi16_t, wi16_t) wi32_t msa_msubv(wi32_t, wi32_t, wi32_t) wi64_t msa_msubv(wi64_t, wi64_t, wi64_t)	a - b * c	Multiply-Subtract

Table 11.30 MSA Arithmetic Instructions *(continued)*

Mnemonic	Compiler Intrinsics	C Expression	Instruction Description

MULV	`wi8_t msa_mulv(wi8_t, wi8_t)`
	`wi16_t msa_mulv(wi16_t, wi16_t)`
	`wi32_t msa_mulv(wi32_t, wi32_t)`
	`wi64_t msa_mulv(wi64_t, wi64_t)`
		`a * b`	Multiply

MOD_S	`wi8_t msa_mod_s(wi8_t, wi8_t)`
	`wi16_t msa_mod_s(wi16_t, wi16_t)`
	`wi32_t msa_mod_s(wi32_t, wi32_t)`
	`wi64_t msa_mod_s(wi64_t, wi64_t)`
		`a % b`	Signed Remainder (Modulo)

MOD_U	`wu8_t msa_mod_u(wu8_t, wu8_t)`
	`wu16_t msa_mod_u(wu16_t, wu16_t)`
	`wu32_t msa_mod_u(wu32_t, wu32_t)`
	`wu64_t msa_mod_u(wu64_t, wu64_t)`
		`a % b`	Unsigned Remainder (Modulo)

SAT_S	`wi8_t msa_sat_s(wi8_t, unsigned char)`
	`wi16_t msa_sat_s(wi16_t, unsigned char)`
	`wi32_t msa_sat_s(wi32_t, unsigned char)`
	`wi64_t msa_sat_s(wi64_t, unsigned char)`
			Signed Saturate

SAT_U	`wu8_t msa_sat_u(wu8_t, unsigned char)`
	`wu16_t msa_sat_u(wu16_t, unsigned char)`
	`wu32_t msa_sat_u(wu32_t, unsigned char)`
	`wu64_t msa_sat_u(wu64_t, unsigned char)`
			Unsigned Saturate

SUBS_S	`wi8_t msa_subs_s(wi8_t, wi8_t)`
	`wi16_t msa_subs_s(wi16_t, wi16_t)`
	`wi32_t msa_subs_s(wi32_t, wi32_t)`
	`wi64_t msa_subs_s(wi64_t, wi64_t)`
			Signed Saturated Subtract

SUBS_U	`wu8_t msa_subs_u(wu8_t, wu8_t)`
	`wu16_t msa_subs_u(wu16_t, wu16_t)`
	`wu32_t msa_subs_u(wu32_t, wu32_t)`
	`wu64_t msa_subs_u(wu64_t, wu64_t)`
			Unsigned Saturated Subtract

HSUB_S	`wi16_t msa_hsub_s(wi8_t, wi8_t)`
	`wi32_t msa_hsub_s(wi16_t, wi16_t)`
	`wi64_t msa_hsub_s(wi32_t, wi32_t)`
			Signed Horizontal Subtract

HSUB_U	`wi16_t msa_hsub_u(wu8_t, wu8_t)`
	`wi32_t msa_hsub_u(wu16_t, wu16_t)`
	`wi64_t msa_hsub_u(wu32_t, wu32_t)`
			Unsigned Horizontal Subtract

SUBSUU_S	`wi8_t msa_subsuu_s(wu8_t, wu8_t)`
	`wi16_t msa_subsuu_s(wu16_t, wu16_t)`
	`wi32_t msa_subsuu_s(wu32_t, wu32_t)`
	`wi64_t msa_subsuu_s(wu64_t, wu64_t)`
			Signed Saturated Unsigned Subtract (both arguments are unsigned, the result is signed)

SUBSUS_U	`wu8_t msa_subsus_u(wu8_t, wi8_t)`
	`wu16_t msa_subsus_u(wu16_t, wi16_t)`
	`wu32_t msa_subsus_u(wu32_t, wi32_t)`
	`wu64_t msa_subsus_u(wu64_t, wi64_t)`
			Unsigned Saturated Signed Subtract from Unsigned (the first argument is unsigned, the second is signed, and the result is unsigned)

Table 11.30 MSA Arithmetic Instructions *(continued)*

Mnemonic	Compiler Intrinsics	C Expression	Instruction Description

SUBV	`wi8_t msa_subv(wi8_t, wi8_t)`
	`wi16_t msa_subv(wi16_t, wi16_t)`
	`wi32_t msa_subv(wi32_t, wi32_t)`
	`wi64_t msa_subv(wi64_t, wi64_t)`
		`a - b`	Subtract

SUBVI	`wi8_t msa_subvi(wi8_t, unsigned char)`
	`wi16_t msa_subvi(wi16_t, unsigned char)`
	`wi32_t msa_subvi(wi32_t, unsigned char)`
	`wi64_t msa_subvi(wi64_t, unsigned char)`
		`a - b`	Subtract Immediate

Table 11.31 MSA Bitwise Instructions

Mnemonic	Compiler Intrinsics	Instruction Description
AND	`wu8_t msa_and(wu8_t, wu8_t)`	Logical And
ANDI	`wu8_t msa_andi(wu8_t, unsigned char)`	Logical And Immediate

BCLR	`wu8_t msa_bclr(wu8_t, wu8_t)`
	`wu16_t msa_bclr(wu16_t, wu16_t)`
	`wu32_t msa_bclr(wu32_t, wu32_t)`
	`wu64_t msa_bclr(wu64_t, wu64_t)`
		Bit Clear

BCLRI	`wu8_t msa_bclri(wu8_t, unsigned char)`
	`wu16_t msa_bclri(wu16_t, unsigned char)`
	`wu32_t msa_bclri(wu32_t, unsigned char)`
	`wu64_t msa_bclri(wu64_t, unsigned char)`
		Bit Clear Immediate

BINSL	`wu8_t msa_binsl(wu8_t, wu8_t, wu8_t)`
	`wu16_t msa_binsl(wu16_t, wu16_t, wu16_t)`
	`wu32_t msa_binsl(wu32_t, wu32_t, wu32_t)`
	`wu64_t msa_binsl(wu64_t, wu64_t, wu64_t)`
		Bit Insert Left

BINSLI	`wu8_t msa_binsli(wu8_t, wu8_t, unsigned char)`
	`wu16_t msa_binsli(wu16_t, wu16_t, unsigned char)`
	`wu32_t msa_binsli(wu32_t, wu32_t, unsigned char)`
	`wu64_t msa_binsli(wu64_t, wu64_t, unsigned char)`
		Bit Insert Left Immediate

BINSR	`wu8_t msa_binsr(wu8_t, wu8_t, wu8_t)`
	`wu16_t msa_binsr(wu16_t, wu16_t, wu16_t)`
	`wu32_t msa_binsr(wu32_t, wu32_t, wu32_t)`
	`wu64_t msa_binsr(wu64_t, wu64_t, wu64_t)`
		Bit Insert Right

BINSRI	`wu8_t msa_binsri(wu8_t, wu8_t, unsigned char)`
	`wu16_t msa_binsri(wu16_t, wu16_t, unsigned char)`
	`wu32_t msa_binsri(wu32_t, wu32_t, unsigned char)`
	`wu64_t msa_binsri(wu64_t, wu64_t, unsigned char)`
		Bit Insert Right Immediate

BMNZ	`wu8_t msa_bmnz(wu8_t, wu8_t, wu8_t)`	Bit Move If Not Zero
BMNZI	`wu8_t msa_bmnzi(wu8_t, wu8_t, unsigned char)`	Bit Move If Not Zero Immediate
BMZ	`wu8_tmsa_bmz(wu8_t, wu8_t, wu8_t)`	Bit Move If Zero
BMZI	`wu8_t msa_bmzi(wu8_t, wu8_t, unsigned char)`	Bit Move If Zero Immediate

Table 11.31 MSA Bitwise Instructions *(continued)*

Mnemonic	Compiler Intrinsics	Instruction Description

BNEG	`wu8_t msa_bneg(wu8_t, wu8_t)`
	`wu16_t msa_bneg(wu16_t, wu16_t)`
	`wu32_t msa_bneg(wu32_t, wu32_t)`
	`wu64_t msa_bneg(wu64_t, wu64_t)`
		Bit Negate

BNEGI	`wu8_t msa_bnegi(wu8_t, unsigned char)`
	`wu16_t msa_bnegi(wu16_t, unsigned char)`
	`wu32_t msa_bnegi(wu32_t, unsigned char)`
	`wu64_t msa_bnegi(wu64_t, unsigned char)`
		Bit Negate Immediate

BSEL	`wu8_t msa_bsel(wu8_t, wu8_t, wu8_t)`	Bit Select
BSELI	`wu8_t msa_bseli(wu8_t, wu8_t, unsigned char)`	Bit Select Immediate

BSET	`wu8_t msa_bset(wu8_t, wu8_t)`
	`wu16_t msa_bset(wu16_t, wu16_t)`
	`wu32_t msa_bset(wu32_t, wu32_t)`
	`wu64_t msa_bset(wu64_t, wu64_t)`
		Bit Set

BSETI	`wu8_t msa_bseti(wu8_t, unsigned char)`
	`wu16_t msa_bseti(wu16_t, unsigned char)`
	`wu32_t msa_bseti(wu32_t, unsigned char)`
	`wu64_t msa_bseti(wu64_t, unsigned char)`
		Bit Set Immediate

NLOC	`wi8_t msa_nloc(wi8_t)`
	`wi16_t msa_nloc(wi16_t)`
	`wi32_t msa_nloc(wi32_t)`
	`wi64_t msa_nloc(wi64_t)`
		Leading One Bits Count

NLZC	`wi8_t msa_nlzc(wi8_t)`
	`wi16_t msa_nlzc(wi16_t)`
	`wi32_t msa_nlzc(wi32_t)`
	`wi64_t msa_nlzc(wi64_t)`
		Leading Zero Bits Count

NOR	`wu8_t msa_nor(wu8_t, wu8_t)`	Logical Negated Or
NORI	`wu8_t msa_nori(wu8_t, unsigned char)`	Logical Negated Or Immediate

PCNT	`wi8_t msa_pcnt(wi8_t)`
	`wi16_t msa_pcnt(wi16_t)`
	`wi32_t msa_pcnt(wi32_t)`
	`wi64_t msa_pcnt(wi64_t)`
		Population (Bits Set to 1) Count

OR	`wu8_t msa_or(wu8_t, wu8_t)`	Logical Or
ORI	`wu8_t msa_ori(wu8_t, unsigned char)`	Logical Or Immediate
XOR	`wu8_t msa_xor(wu8_t, wu8_t)`	Logical Or
XORI	`wu8_t msa_xori(wu8_t, unsigned char)`	Logical Or Immediate

SLL	`wi8_t msa_sll(wi8_t, wi8_t)`
	`wi16_t msa_sll(wi16_t, wi16_t)`
	`wi32_t msa_sll(wi32_t, wi32_t)`
	`wi64_t msa_sll(wi64_t, wi64_t)`
		Shift Left

Table 11.31 MSA Bitwise Instructions *(continued)*

Mnemonic	Compiler Intrinsics	Instruction Description

SLLI	`wi8_t msa_slli(wi8_t, unsigned char)`
	`wi16_t msa_slli(wi16_t, unsigned char)`
	`wi32_t msa_slli(wi32_t, unsigned char)`
	`wi64_t msa_slli(wi64_t, unsigned char)`
		Shift Left Immediate

SRA	`wi8_t msa_sra(wi8_t, wi8_t)`
	`wi16_t msa_sra(wi16_t, wi16_t)`
	`wi32_t msa_sra(wi32_t, wi32_t)`
	`wi64_t msa_sra(wi64_t, wi64_t)`
		Shift Right Arithmetic

SRAI	`wi8_t msa_srai(wi8_t, unsigned char)`
	`wi16_t msa_srai(wi16_t, unsigned char)`
	`wi32_t msa_srai(wi32_t, unsigned char)`
	`wi64_t msa_srai(wi64_t, unsigned char)`
		Shift Right Arithmetic Immediate

SRAR	`wi8_t msa_srar(wi8_t, wi8_t)`
	`wi16_t msa_srar(wi16_t, wi16_t)`
	`wi32_t msa_srar(wi32_t, wi32_t)`
	`wi64_t msa_srar(wi64_t, wi64_t)`
		Shift Right Arithmetic with Rounding

SRARI	`wi8_t msa_srari(wi8_t, unsigned char)`
	`wi16_t msa_srari(wi16_t, unsigned char)`
	`wi32_t msa_srari(wi32_t, unsigned char)`
	`wi64_t msa_srari(wi64_t, unsigned char)`
		Shift Right Arithmetic with Rounding Immediate

SRL	`wi8_t msa_srl(wi8_t, wi8_t)`
	`wi16_t msa_srl(wi16_t, wi16_t)`
	`wi32_t msa_srl(wi32_t, wi32_t)`
	`wi64_t msa_srl(wi64_t, wi64_t)`
		Shift Right

SRLI	`wi8_t msa_srli(wi8_t, unsigned char)`
	`wi16_t msa_srli(wi16_t, unsigned char)`
	`wi32_t msa_srli(wi32_t, unsigned char)`
	`wi64_t msa_srli(wi64_t, unsigned char)`
		Shift Right Immediate

SRLR	`wi8_t msa_srlr(wi8_t, wi8_t)`
	`wi16_t msa_srlr(wi16_t, wi16_t)`
	`wi32_t msa_srlr(wi32_t, wi32_t)`
	`wi64_t msa_srlr(wi64_t, wi64_t)`
		Shift Right with Rounding

SRLRI	`wi8_t msa_srlri(wi8_t, unsigned char)`
	`wi16_t msa_srlri(wi16_t, unsigned char)`
	`wi32_t msa_srlri(wi32_t, unsigned char)`
	`wi64_t msa_srlri(wi64_t, unsigned char)`
		Shift Right with Rounding Immediate

11.14.2 MSA Floating-Point Instructions

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™-2008. The floating-point arithmetic operations implemented by dedicated instructions are: addition/subtract, multiply/divide, fused multiply add/sub, base 2 exponentiation and integer logarithm, max/min including for absolute values, and integer rounding (Table 11.32).

The floating-point compare instructions (Table 11.33) are similar with the integer comparisons: all set destination bits to zero (false) or one (true). The floating-point specific unordered relations are supported by dedicated quiet compare unordered instructions and a complete set of signaling compare instructions.

Format conversion instructions (Table 11.34) cover single (32-bit) to/from double-precision (64-bit) and single to/from 16-bit floating-point format. Integer and fixed-point conversions are also supported.

In the case of a floating-point exception, each faulting vector element is precisely identified without the need for software emulation for all vector elements.

Table 11.32 MSA Floating-Point Arithmetic Instructions

Mnemonic	Compiler Intrinsics	Instruction Description
FADD	`wf32_t msa_fadd(wf32_t, wf32_t)` `wf64_t msa_fadd(wf64_t, wf64_t)`	Floating-Point Addition
FDIV	`wf32_t msa_fdiv(wf32_t, wf32_t)` `wf64_t msa_fdiv(wf64_t, wf64_t)`	Floating-Point Division
FEXP2	`wf32_t msa_fexp2(wf32_t, wi32_t)` `wf64_t msa_fexp2(wf64_t, wi64_t)`	Floating-Point Base 2 Exponentiation
FLOG2	`wf32_t msa_flog2(wf32_t)` `wf64_t msa_flog2(wf64_t)`	Floating-Point Base 2 Logarithm
FMADD	`wf32_t msa_fmadd(wf32_t, wf32_t, wf32_t)` `wf64_t msa_fmadd(wf64_t, wf64_t, wf64_t)`	Floating-Point Fused Multiply-Add
FMSUB	`wf32_t msa_fmsub(wf32_t, wf32_t, wf32_t)` `wf64_t msa_fmsub(wf64_t, wf64_t, wf64_t)`	Floating-Point Fused Multiply-Subtract
FMAX	`wf32_t msa_fmax(wf32_t, wf32_t)` `wf64_t msa_fmax(wf64_t, wf64_t)`	Floating-Point Maximum
FMIN	`wf32_t msa_fmin(wf32_t, wf32_t)` `wf64_t msa_fmin(wf64_t, wf64_t)`	Floating-Point Minimum
FMAX_A	`wf32_t msa_fmax_a(wf32_t, wf32_t)` `wf64_t msa_fmax_a(wf64_t, wf64_t)`	Floating-Point Maximum of Absolute Values
FMIN_A	`wf32_t msa_fmin_a(wf32_t, wf32_t)` `wf64_t msa_fmin_a(wf64_t, wf64_t)`	Floating-Point Minimum of Absolute Values
FMUL	`wf32_t msa_fmul(wf32_t, wf32_t)` `wf64_t msa_fmul(wf64_t, wf64_t)`	Floating-Point Multiplication
FRCP	`wf32_t msa_frcp(wf32_t)` `wf64_t msa_frcp(wf64_t)`	Approximate Floating-Point Reciprocal
FRINT	`wf32_t msa_frint(wf32_t)` `wf64_t msa_frint(wf64_t)`	Floating-Point Round to Integer
FRSQRT	`wf32_t msa_frsqrt(wf32_t)` `wf64_t msa_frsqrt(wf64_t)`	Approximate Floating-Point Reciprocal of Square Root
FSQRT	`wf32_t msa_fsqrt(wf32_t)` `wf64_t msa_fsqrt(wf64_t)`	Floating-Point Square Root
FSUB	`wf32_t msa_fsub(wf32_t, wf32_t)` `wf64_t msa_fsub(wf64_t, wf64_t)`	Floating-Point Subtraction

Table 11.33 MSA Floating-Point Compare Instructions

Mnemonic	Compiler Intrinsics	Instruction Description
FCLASS	`wi32_t msa_fclass(wf32_t)` `wi64_t msa_fclass(wf64_t)`	Floating-Point Class Mask
FCAF	`wi32_t msa_fcaf(wf32_t, wf32_t)` `wi64_t msa_fcaf(wf64_t, wf64_t)`	Floating-Point Quiet Compare Always False
FCUN	`wi32_t msa_fcun(wf32_t, wf32_t)` `wi64_t msa_fcun(wf64_t, wf64_t)`	Floating-Point Quiet Compare Unordered
FCOR	`wi32_t msa_fcor(wf32_t, wf32_t)` `wi64_t msa_fcor(wf64_t, wf64_t)`	Floating-Point Quiet Compare Ordered
FCEQ	`wi32_t msa_fceq(wf32_t, wf32_t)` `wi64_t msa_fceq(wf64_t, wf64_t)`	Floating-Point Quiet Compare Equal
FCUNE	`wi32_t msa_fcune(wf32_t, wf32_t)` `wi64_t msa_fcune(wf64_t, wf64_t)`	Floating-Point Quiet Compare Unordered or Not Equal
FCUEQ	`wi32_t msa_fcueq(wf32_t, wf32_t)` `wi64_t msa_fcueq(wf64_t, wf64_t)`	Floating-Point Quiet Compare Unordered or Equal
FCNE	`wi32_t msa_fcne(wf32_t, wf32_t)` `wi64_t msa_fcne(wf64_t, wf64_t)`	Floating-Point Quiet Compare Not Equal
FCLT	`wi32_t msa_fclt(wf32_t, wf32_t)` `wi64_t msa_fclt(wf64_t, wf64_t)`	Floating-Point Quiet Compare Less Than
FCULT	`wi32_t msa_fcult(wf32_t, wf32_t)` `wi64_t msa_fcult(wf64_t, wf64_t)`	Floating-Point Quiet Compare Unordered or Less Than
FCLE	`wi32_t msa_fcle(wf32_t, wf32_t)` `wi64_t msa_fcle(wf64_t, wf64_t)`	Floating-Point Quiet Compare Less Than or Equal
FCULE	`wi32_t msa_fcule(wf32_t, wf32_t)` `wi64_t msa_fcule(wf64_t, wf64_t)`	Floating-Point Quiet Compare Unordered or Less Than or Equal
FSAF	`wi32_t msa_fsaf(wf32_t, wf32_t)` `wi64_t msa_fsaf(wf64_t, wf64_t)`	Floating-Point Signaling Compare Always False
FSUN	`wi32_t msa_fsun(wf32_t, wf32_t)` `wi64_t msa_fsun(wf64_t, wf64_t)`	Floating-Point Signaling Compare Unordered
FSOR	`wi32_t msa_fsor(wf32_t, wf32_t)` `wi64_t msa_fsor(wf64_t, wf64_t)`	Floating-Point Signaling Compare Ordered
FSEQ	`wi32_t msa_fseq(wf32_t, wf32_t)` `wi64_t msa_fseq(wf64_t, wf64_t)`	Floating-Point Signaling Compare Equal
FSUNE	`wi32_t msa_fsune(wf32_t, wf32_t)` `wi64_t msa_fsune(wf64_t, wf64_t)`	Floating-Point Signaling Compare Unordered or Not Equal
FSUEQ	`wi32_t msa_fsueq(wf32_t, wf32_t)` `wi64_t msa_fsueq(wf64_t, wf64_t)`	Floating-Point Signaling Compare Unordered or Equal
FSNE	`wi32_t msa_fsne(wf32_t, wf32_t)` `wi64_t msa_fsne(wf64_t, wf64_t)`	Floating-Point Signaling Compare Not Equal

Table 11.33 MSA Floating-Point Compare Instructions *(continued)*

Mnemonic	Compiler Intrinsics	Instruction Description
FSLT	`wi32_t msa_fslt(wf32_t, wf32_t)` `wi64_t msa_fslt(wf64_t, wf64_t)`	Floating-Point Signaling Compare Less Than
FSULT	`wi32_t msa_fsult(wf32_t, wf32_t)` `wi64_t msa_fsult(wf64_t, wf64_t)`	Floating-Point Signaling Compare Unordered or Less Than
FSLE	`wi32_t msa_fsle(wf32_t, wf32_t)` `wi64_t msa_fsle(wf64_t, wf64_t)`	Floating-Point Signaling Compare Less Than or Equal
FSULE	`wi32_t msa_fsule(wf32_t, wf32_t)` `wi64_t msa_fsule(wf64_t, wf64_t)`	Floating-Point Signaling Compare Unordered or Less Than or Equal

Table 11.34 MSA Floating-Point Conversion Instructions

Mnemonic	Compiler Intrinsics	Instruction Description
FEXDO	`wi16_t msa_fexdo(wf32_t, wf32_t)` `wf32_t msa_fexdo(wf64_t, wf64_t)`	Floating-Point Down-Convert Interchange Format
FEXUPL	`wf64_t msa_fexupl(wf32_t)` `wf32_t msa_fexupl(wi16_t)`	Left-Half Floating-Point Up-Convert Interchange Format
FEXUPR	`wf64_t msa_fexupr(wf32_t)` `wf32_t msa_fexupr(wi16_t)`	Right-Half Floating-Point Up-Convert Interchange Format
FFINT_S	`wf32_t msa_ffint_s(wi32_t)` `wf64_t msa_ffint_s(wi64_t)`	Floating-Point Convert from Signed Integer
FFINT_U	`wf32_t msa_ffint_u(wu32_t)` `wf64_t msa_ffint_u(wu64_t)`	Floating-Point Convert from Unsigned Integer
FFQL	`wf32_t msa_ffql(wi16_t)` `wf64_t msa_ffql(wi32_t)`	Left-Half Floating-Point Convert from Fixed-Point
FFQR	`wf32_t msa_ffqr(wi16_t)` `wf64_t msa_ffqr(wi32_t)`	Right-Half Floating-Point Convert from Fixed-Point
FTINT_S	`wi32_t msa_ftint_s(wf32_t)` `wi64_t msa_ftint_s(wf64_t)`	Floating-Point Round and Convert to Signed Integer
FTINT_U	`wu32_t msa_ftint_u(wf32_t)` `wu64_t msa_ftint_u(wf64_t)`	Floating-Point Round and Convert to Unsigned Integer
FTRUNC_S	`wi32_t msa_ftrunc_s(wf32_t)` `wi64_t msa_ftrunc_s(wf64_t)`	Floating-Point Truncate and Convert to Signed Integer
FTRUNC_U	`wu32_t msa_ftrunc_u(wf32_t)` `wu64_t msa_ftrunc_u(wf64_t)`	Floating-Point Truncate and Convert to Unsigned Integer
FTQ	`wi16_t msa_ftq(wf32_t, wf32_t)` `wi32_t msa_ftq(wf64_t, wf64_t)`	Floating-Point Round and Convert to Fixed-Point

11.14.3 Fixed-Point Multiplication Instructions

The fixed-point data formats are Q15 and Q31, i.e. one sign bit and 15 or 31 fractional bits, representing values in the (-1, 1) interval. While the fixed-point add/sub is the regular 2's complement add/sub with saturation, the multiplication operation requires scaling (left shift) with saturation.

The MSA has dedicated fixed-point multiplication instructions (Table 11.35) with optional rounding.

Table 11.35 MSA Fixed-Point Instructions

Mnemonic	Compiler Intrinsic	Instruction Description
MADD_Q	`wi16_t msa_madd_q(wi16_t, wi16_t, wi16_t)` `wi32_t msa_madd_q(wi32_t, wi32_t, wi32_t)`	Fixed-Point Multiply and Add
MADDR_Q	`wi16_t msa_maddr_q(wi16_t, wi16_t, wi16_t)` `wi32_t msa_maddr_q(wi32_t, wi32_t, wi32_t)`	Fixed-Point Multiply and Add with Rounding
MSUB_Q	`wi16_t msa_msub_q(wi16_t, wi16_t, wi16_t)` `wi32_t msa_msub_q(wi32_t, wi32_t, wi32_t)`	Fixed-Point Multiply and Subtract
MSUBR_Q	`wi16_t msa_msubr_q(wi16_t, wi16_t, wi16_t)` `wi32_t msa_msubr_q(wi32_t, wi32_t, wi32_t)`	Fixed-Point Multiply and Subtract with Rounding
MUL_Q	`wi16_t msa_mul_q(wi16_t, wi16_t)` `wi32_t msa_mul_q(wi32_t, wi32_t)`	Fixed-Point Multiply
MULR_Q	`wi16_t msa_mulr_q(wi16_t, wi16_t)` `wi32_t msa_mulr_q(wi32_t, wi32_t)`	Fixed-Point Multiply with Rounding

11.14.4 Branch and Compare Instructions

Branch and compare instructions (Table 11.36) are based on truth values: zero for false and non-zero for true. There are no dedicated condition flags.

The compare instructions set the destination element to the truth value of the compare operation for the corresponding source elements. All compare instructions accept a small, 5-bit constant as the second compare operand across all vector elements.

Both branch-on-false and branch-on-true condition instructions are provided, because the vector under test contains multiple truth values that cannot be negated by simply changing the compare operator. As such, there is a pair of branch-on-false (zero) instructions that test if at least one element is zero or if all elements are zero, and a pair of branch-on-true (not zero) instructions that test if all elements are not zero, or if at least one element is not zero.

Table 11.36 MSA Branch and Compare Instructions

Mnemonic	Compiler Intrinsic	Instruction Description
BNZ		Branch If Not Zero
BZ		Branch If Zero

Table 11.36 MSA Branch and Compare Instructions *(continued)*

Mnemonic	Compiler Intrinsic	Instruction Description

CEQ	`wi8_t msa_ceq(wi8_t, wi8_t)`
	`wi16_t msa_ceq(wi16_t, wi16_t)`
	`wi32_t msa_ceq(wi32_t, wi32_t)`
	`wi64_t msa_ceq(wi64_t, wi64_t)`
		Compare Equal

CEQI	`wi8_t msa_ceqi(wi8_t, char)`
	`wi16_t msa_ceqi(wi16_t, char)`
	`wi32_t msa_ceqi(wi32_t, char)`
	`wi64_t msa_ceqi(wi64_t, char)`
		Compare Equal Immediate

CLE_S	`wi8_t msa_cle_s(wi8_t, wi8_t)`
	`wi16_t msa_cle_s(wi16_t, wi16_t)`
	`wi32_t msa_cle_s(wi32_t, wi32_t)`
	`wi64_t msa_cle_s(wi64_t, wi64_t)`
		Compare Less-Than-or-Equal Signed and Unsigned

CLEI_S	`wi8_t msa_clei_s(wi8_t, char)`
	`wi16_t msa_clei_s(wi16_t, char)`
	`wi32_t msa_clei_s(wi32_t, char)`
	`wi64_t msa_clei_s(wi64_t, char)`
		Compare Less-Than-or-Equal Signed and Unsigned Immediate

CLE_U	`wi8_t msa_cle_u(wu8_t, wu8_t)`
	`wi16_t msa_cle_u(wu16_t, wu16_t)`
	`wi32_t msa_cle_u(wu32_t, wu32_t)`
	`wi64_t msa_cle_u(wu64_t, wu64_t)`
		Compare Less-Than-or-Equal Signed and Unsigned

CLEI_U	`wi8_t msa_clei_u(wu8_t, unsigned char)`
	`wi16_t msa_clei_u(wu16_t, unsigned char)`
	`wi32_t msa_clei_u(wu32_t, unsigned char)`
	`wi64_t msa_clei_u(wu64_t, unsigned char)`
		Compare Less-Than-or-Equal Signed and Unsigned Immediate

CLT_S	`wi8_t msa_clt_s(wi8_t, wi8_t)`
	`wi16_t msa_clt_s(wi16_t, wi16_t)`
	`wi32_t msa_clt_s(wi32_t, wi32_t)`
	`wi64_t msa_clt_s(wi64_t, wi64_t)`
		Compare Less-Than Signed and Unsigned

CLTI_S	`wi8_t msa_clti_s(wi8_t, char)`
	`wi16_t msa_clti_s(wi16_t, char)`
	`wi32_t msa_clti_s(wi32_t, char)`
	`wi64_t msa_clti_s(wi64_t, char)`
		Compare Less-Than Signed and Unsigned Immediate

CLT_U	`wi8_t msa_clt_u(wu8_t, wu8_t)`
	`wi16_t msa_clt_u(wu16_t, wu16_t)`
	`wi32_t msa_clt_u(wu32_t, wu32_t)`
	`wi64_t msa_clt_u(wu64_t, wu64_t)`
		Compare Less-Than Signed and Unsigned

CLTI_U	`wi8_t msa_clti_u(wu8_t, unsigned char)`
	`wi16_t msa_clti_u(wu16_t, unsigned char)`
	`wi32_t msa_clti_u(wu32_t, unsigned char)`
	`wi64_t msa_clti_u(wu64_t, unsigned char)`
		Compare Less-Than Signed and Unsigned Immediate

11.14.5 Load/Store and Element Move Instructions

The MSA is very flexible and consistent regarding data transfers between the vector registers and the general-purpose registers (GPRs) or memory. Data transfer instructions (Table 11.37) include vector memory load/store and element move instructions such as vector element data copy to GPR, all vector elements fill with GPR or immediate data, and insert GPR data to a specific element. The load/store instructions do not require 128-bit (16-byte) memory address alignment.

All data transfer instructions are typed, i.e., the data format is explicitly specified. This is particularly important for the vector load/store instructions, because it allows any halfword, word, or doubleword data to make the round-trip between GPRs, memory, and vector registers without any need for endian related byte swaps.

For example, a store halfword (source) vector register will write the eight halfword values to memory, which then can be loaded as halfwords one-by-one in GPRs, which then can be transferred one-by-one to another (destination) vector register. The source vector register from which the halfword values were initiated is identical to the destination vector register, regardless of the endian memory mode.

Table 11.37 MSA Load/Store and Move Instructions

Mnemonic	Compiler Intrinsics	Instruction Description
CFCMSA	`int msa_cfcmsa(unsigned char)`	Copy from MSA Control Register
CTCMSA	`void msa_ctcmsa(unsigned char, int)`	Copy to MSA Control Register

LD	`wi8_t msa_ld(wi8_t*, int)`
	`wi16_t msa_ld(wi16_t*, int)`
	`wi32_t msa_ld(wi32_t*, int)`
	`wi64_t msa_ld(wi64_t*, int)`
	`wf32_t msa_ld(wf32_t*, int)`
	`wf64_t msa_ld(wf64_t*, int)`
		Load Vector

LDI	`wi8_t msa_ldi(short)`
	`wi16_t msa_ldi(short)`
	`wi32_t msa_ldi(short)`
	`wi64_t msa_ldi(short)`
		Load Immediate

MOVE	`wi8_t msa_move(wi8_t)`
	`wi16_t msa_move(wi16_t)`
	`wi32_t msa_move(wi32_t)`
	`wi64_t msa_move(wi64_t)`
	`wf32_t msa_move(wf32_t)`
	`wf64_t msa_move(wf64_t)`
		Vector to Vector Move

SPLAT	`wi8_t msa_splat(wi8_t, int)`
	`wi16_t msa_splat(wi16_t, int)`
	`wi32_t msa_splat(wi32_t, int)`
	`wi64_t msa_splat(wi64_t, int)`
		Replicate Vector Element

SPLATI	`wi8_t msa_splati(wi8_t, unsigned char)`
	`wi16_t msa_splati(wi16_t, unsigned char)`
	`wi32_t msa_splati(wi32_t, unsigned char)`
	`wi64_t msa_splati(wi64_t, unsigned char)`
		Replicate Vector Element

FILL	`wi8_t msa_fill(int)`
	`wi16_t msa_fill(int)`
	`wi32_t msa_fill(int)`
	`wi64_t msa_fill(int)`
		Fill Vector from GPR

Table 11.37 MSA Load/Store and Move Instructions *(continued)*

Mnemonic	Compiler Intrinsics	Instruction Description

INSERT	`wi8_t msa_insert(wi8_t, unsigned char, int)`
	`wi16_t msa_insert(wi16_t, unsigned char, int)`
	`wi32_t msa_insert(wi32_t, unsigned char, int)`
	`wi64_t msa_insert(wi64_t, unsigned char, int)`
		Insert GPR and Vector element 0 to Vector Element

INSVE	`wi8_t msa_insve(wi8_t, unsigned char, wi8_t)`
	`wi16_t msa_insve(wi16_t, unsigned char, wi16_t)`
	`wi32_t msa_insve(wi32_t, unsigned char, wi32_t)`
	`wi64_t msa_insve(wi64_t, unsigned char, wi64_t)`
		Insert GPR and Vector element 0 to Vector Element

COPY_S	`int msa_copy_s(wi8_t, unsigned char)`
	`int msa_copy_s(wi16_t, unsigned char)`
	`int msa_copy_s(wi32_t, unsigned char)`
	`int msa_copy_s(wi64_t, unsigned char)`
		Copy element to GPR Signed and Unsigned

COPY_U	`int msa_copy_u(wi8_t, unsigned char)`
	`int msa_copy_u(wi16_t, unsigned char)`
	`int msa_copy_u(wi32_t, unsigned char)`
	`int msa_copy_u(wi64_t, unsigned char)`
		Copy element to GPR Signed and Unsigned

ST	`void msa_st(wi8_t*, int)`
	`void msa_st(wi16_t*, int)`
	`void msa_st(wi32_t*, int)`
	`void msa_st(wi64_t*, int)`
	`void msa_st(wf32_t*, int)`
	`void msa_st(wf64_t*, int)`
		Store Vector

11.14.6 Element Permute Instructions

Vector elements can be shuffled based on either a pre-defined pattern or an arbitrary mapping function. Pre-defined patterns are more efficient because no prior set-up is required. Mapping functions provide the most general shuffling, but could take an extra vector register to specify where each source element will be put in the destination vector.

The MSA has both generic mapping and pre-defined pattern-shuffle instructions (Table 11.38). Pre-defined pattern instructions interleave odd or even elements from two source vectors, or pack all odd or all even elements from two source vectors into the upper half and the lower half of a destination vector.

Note that the architecture independent GCC `__builtin_shuffle()` is intentionally semantically compatible with the MSA VSHF instruction.

A second class of predefined patterns are geometrical in nature: the two source vectors seen as byte arrays (of one line by eight columns, two lines by four columns, or four lines by two columns) are horizontally concatenated. The destination is a byte array selected by a sliding window of similar shape (array of one by eight, two by four, or four by two) over the concatenation of the source arrays.

Table 11.38 MSA Element Permute Instructions

Mnemonic	Compiler Intrinsics	Instruction Description

ILVEV	`wi8_t msa_ilvev(wi8_t, wi8_t)`
	`wi16_t msa_ilvev(wi16_t, wi16_t)`
	`wi32_t msa_ilvev(wi32_t, wi32_t)`
	`wi64_t msa_ilvev(wi64_t, wi64_t)`
		Interleave Even

ILVOD	`wi8_t msa_ilvod(wi8_t, wi8_t)`
	`wi16_t msa_ilvod(wi16_t, wi16_t)`
	`wi32_t msa_ilvod(wi32_t, wi32_t)`
	`wi64_t msa_ilvod(wi64_t, wi64_t)`
		Interleave Odd

ILVL	`wi8_t msa_ilvl(wi8_t, wi8_t)`
	`wi16_t msa_ilvl(wi16_t, wi16_t)`
	`wi32_t msa_ilvl(wi32_t, wi32_t)`
	`wi64_t msa_ilvl(wi64_t, wi64_t)`
		Interleave Left

ILVR	`wi8_t msa_ilvr(wi8_t, wi8_t)`
	`wi16_t msa_ilvr(wi16_t, wi16_t)`
	`wi32_t msa_ilvr(wi32_t, wi32_t)`
	`wi64_t msa_ilvr(wi64_t, wi64_t)`
		Interleave Right

PCKEV	`wi8_t msa_pckev(wi8_t, wi8_t)`
	`wi16_t msa_pckev(wi16_t, wi16_t)`
	`wi32_t msa_pckev(wi32_t, wi32_t)`
	`wi64_t msa_pckev(wi64_t, wi64_t)`
		Pack Even Elements

PCKOD	`wi8_t msa_pckod(wi8_t, wi8_t)`
	`wi16_t msa_pckod(wi16_t, wi16_t)`
	`wi32_t msa_pckod(wi32_t, wi32_t)`
	`wi64_t msa_pckod(wi64_t, wi64_t)`
		Pack Odd Elements

SHF	`wi8_t msa_shf(wi8_t, unsigned char)`
	`wi16_t msa_shf(wi16_t, unsigned char)`
	`wi32_t msa_shf(wi32_t, unsigned char)`
		Set Shuffle

SLD	`wi8_t msa_sld(wi8_t, wi8_t, int)`
	`wi16_t msa_sld(wi16_t, wi16_t, int)`
	`wi32_t msa_sld(wi32_t, wi32_t, int)`
	`wi64_t msa_sld(wi64_t, wi64_t, int)`
		Element Slide

SLDI	`wi8_t msa_sldi(wi8_t, wi8_t, unsigned char)`
	`wi16_t msa_sldi(wi16_t, wi16_t, unsigned char)`
	`wi32_t msa_sldi(wi32_t, wi32_t, unsigned char)`
	`wi64_t msa_sldi(wi64_t, wi64_t, unsigned char)`
		Element Slide Immediate

VSHF	`wi8_t msa_vshf(wi8_t, wi8_t, wi8_t)`
	`wi16_t msa_vshf(wi16_t, wi16_t, wi16_t)`
	`wi32_t msa_vshf(wi32_t, wi32_t, wi32_t)`
	`wi64_t msa_vshf(wi64_t, wi64_t, wi64_t)`
		Vector shuffle

11.15 Alphabetical Listing of Floating Point Instructions

Table 11.39 shows an alphabetical listing of the floating point unit instruction set, along with the associated instruction group. For the definition of each instruction, refer to Table 11.22 through Table 11.38 above.

Table 11.39 Alphabetical Listing of FPU Instructions

Instruction Name	Instruction Group
ABS.fmt	Move
ADD.fmt	Arithmetic
BC1EQZ	Conditional Branch
BC1NEZ	Conditional Branch
CLASS.fmt	Arithmetic
CMP.cond.fmt	Arithmetic
CEIL.L.fmt	Conversion
CEIL.W.fmt	Conversion
CFC1	Move
CTC1	Move
CVT.D.fmt	Conversion
CVT.L.fmt	Conversion
CVT.S.fmt	Conversion
CVT.W.fmt	Conversion
DIV.fmt	Arithmetic
FLOOR.L.fmt	Conversion
FLOOR.W.fmt	Conversion
LDC1	Load/Store
LWC1	Load/Store
MADDF.fmt	Fused Multiply-Add
MAX.fmt	Arithmetic
MAX_A.fmt	Arithmetic
MSUBF.fmt	Fused Multiply-Subtract
MFC1	Move
MFHC1	Move
MIN.fmt	Arithmetic
MIN_A.fmt	Arithmetic
MSUB.fmt	Multiply-Accumulate
MTC1	Move
MUL.fmt	Arithmetic
RECIP.fmt	Arithmetic
RINT.fmt	Conversion
ROUND.L.fmt	Conversion

Table 11.39 Alphabetical Listing of FPU Instructions *(continued)*

Instruction Name	Instruction Group
ROUND.W.fmt	Conversion
RSQRT.fmt	Arithmetic
SDC1	Load/Store
SELEQZ.fmt	Arithmetic
SELNEZ.fmt	Arithmetic
SQRT.fmt	Arithmetic
SUB.fmt	Arithmetic
SWC1	Load/Store
TRUNC.L.fmt	Conversion
TRUNC.W.fmt	Conversion

11.16 Alphabetical Listing of MSA SIMD Instructions

Table 11.40 shows an alphabetical listing of the MSA SIMD instruction set, along with the associated instruction group. For the definition of each instruction, refer to Table 11.30 through Table 11.38 above.

Table 11.40 Alphabetical Listing of MSA Instructions

Instruction Name	Instruction Group
ADD_A	Arithmetic
ADDS_A	Arithmetic
ADDS_S	Arithmetic
ADDS_U	Arithmetic
ADDV	Arithmetic
ADDVI	Arithmetic
AND	Bitwise
ANDI	Bitwise
ASUB_S	Arithmetic
ASUB_U	Arithmetic
AVE_S	Arithmetic
AVE_U	Arithmetic
AVER_S	Arithmetic
AVER_U	Arithmetic
BCLR	Bitwise
BCLRI	Bitwise
BINSL	Bitwise
BINSLI	Bitwise
BINSR	Bitwise
BINSRI	Bitwise

Table 11.40 Alphabetical Listing of MSA Instructions *(continued)*

Instruction Name	Instruction Group
BMZ	Bitwise
BMZI	Bitwise
BNEG	Bitwise
BNEGI	Bitwise
BNZ	Branch and Compare
BSEL	Bitwise
BSELI	Bitwise
BSET	Bitwise
BSETI	Bitwise
BZ	Branch and Compare
CEQ	Branch and Compare
CEQI	Branch and Compare
CFCMSA	Load / Store and Move
CLE_S	Branch and Compare
CLEI_S	Branch and Compare
CLE_U	Branch and Compare
CLEI_U	Branch and Compare
CLT_S	Branch and Compare
CLTI_S	Branch and Compare
CLT_U	Branch and Compare
CLTI_U	Branch and Compare
COPY_S	Load / Store and Move
COPY_U	Load / Store and Move
CTCMSA	Load / Store and Move
DIV_S	Arithmetic
DIV_U	Arithmetic
DOTP_S	Arithmetic
DOTP_U	Arithmetic
DPADD_S	Arithmetic
DPADD_U	Arithmetic
DPSUB_S	Arithmetic
DPSUB_U	Arithmetic
FADD	Floating Point Arithmetic
FCLASS	Floating Point Compare
FCAF	Floating Point Compare
FCUN	Floating Point Compare
FCOR	Floating Point Compare

Table 11.40 Alphabetical Listing of MSA Instructions *(continued)*

Instruction Name	Instruction Group
FCEQ	Floating Point Compare
FCUNE	Floating Point Compare
FCUEQ	Floating Point Compare
FCNE	Floating Point Compare
FCLT	Floating Point Compare
FCULT	Floating Point Compare
FCLE	Floating Point Compare
FCULE	Floating Point Compare
FDIV	Floating Point Arithmetic
FEXDO	Floaint Point Conversion
FEXP2	Floating Point Arithmetic
FEXUPL	Floaint Point Conversion
FEXUPR	Floaint Point Conversion
FFINT_S	Floaint Point Conversion
FFINT_U	Floaint Point Conversion
FFQL	Floaint Point Conversion
FFQR	Floaint Point Conversion
FILL	Load / Store and Move
FLOG2	Floating Point Arithmetic
FMADD	Floating Point Arithmetic
FMSUB	Floating Point Arithmetic
FMAX	Floating Point Arithmetic
FMIN	Floating Point Arithmetic
FMAX_A	Floating Point Arithmetic
FMIN_A	Floating Point Arithmetic
FMUL	Floating Point Arithmetic
FRCP	Floating Point Arithmetic
FRINT	Floating Point Arithmetic
FRSQRT	Floating Point Arithmetic
FSAF	Floating Point Compare
FSEQ	Floating Point Compare
FSLE	Floating Point Compare
FSLT	Floating Point Compare
FSNE	Floating Point Compare
FSOR	Floating Point Compare
FSUEQ	Floating Point Compare
FSUB	Floating Point Arithmetic

Table 11.40 Alphabetical Listing of MSA Instructions *(continued)*

Instruction Name	Instruction Group
FSULE	Floating Point Compare
FSULT	Floating Point Compare
FSUN	Floating Point Compare
FSUNE	Floating Point Compare
FTINT_S	Floaint Point Conversion
FTINT_U	Floaint Point Conversion
FTRUNC_S	Floaint Point Conversion
FTRUNC_U	Floaint Point Conversion
FTQ	Floaint Point Conversion
HADD_S	Arithmetic
HADD_U	Arithmetic
HSUB_S	Arithmetic
HSUB_U	Arithmetic
ILVEV	Element Permute
ILVOD	Element Permute
ILVL	Element Permute
ILVR	Element Permute
INSERT	Load / Store and Move
INSVE	Load / Store and Move
LD	Load / Store and Move
LDI	Load / Store and Move
MADD_Q	Fixed Point
MADDR_Q	Fixed Point
MADDV	Arithmetic
MAX_A	Arithmetic
MAX_S	Arithmetic
MAX_U	Arithmetic
MAXI_S	Arithmetic
MAXI_U	Arithmetic
MIN_A	Arithmetic
MIN_S	Arithmetic
MIN_U	Arithmetic
MINI_S	Arithmetic
MINI_U	Arithmetic
MOD_S	Arithmetic
MOD_U	Arithmetic
MOVE	Load / Store and Move

Table 11.40 Alphabetical Listing of MSA Instructions *(continued)*

Instruction Name	Instruction Group
MSUB_Q	Fixed Point
MSUBR_Q	Fixed Point
MSUBV	Arithmetic
MUL_Q	Fixed Point
MULR_Q	Fixed Point
MULV	Arithmetic
NLOC	Bitwise
NLZC	Bitwise
NOR	Bitwise
NORI	Bitwise
PCKEV	Element Permute
PCKOD	Element Permute
PCNT	Bitwise
OR	Bitwise
ORI	Bitwise
SAT_S	Arithmetic
SAT_U	Arithmetic
SHF	Element Permute
SLD	Element Permute
SLDI	Element Permute
SLL	Bitwise
SLLI	Bitwise
SPLAT	Load / Store and Move
SPLATI	Load / Store and Move
SRA	Bitwise
SRAI	Bitwise
SRAR	Bitwise
SRARI	Bitwise
SRL	Bitwise
SRLI	Bitwise
SRLR	Bitwise
SRLRI	Bitwise
ST	Load / Store and Move
SUB_S	Arithmetic
SUB_U	Arithmetic
SUBSUS_U	Arithmetic
SUBSUU_U	Arithmetic

Table 11.40 Alphabetical Listing of MSA Instructions *(continued)*

Instruction Name	Instruction Group
SUBV	Arithmetic
SUBVI	Arithmetic
VSHF	Element Permute

Chapter 12

Hardware and Software Initialization

A P6600 core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 12.1 "Hardware-Initialized Processor State"
- Section 12.2 "Software-Initialized Processor State"

12.1 Hardware-Initialized Processor State

The P6600 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.

12.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0:

- *Wired* - cleared to 0 on Reset
- $Status_{BEV}$ - set to 1 on Reset
- $Status_{TS}$ - cleared to 0 on Reset
- $Status_{NMI}$ - cleared to 0 on Reset
- $Status_{ERL}$ - set to 1 on Reset
- $Status_{RP}$ - cleared to 0 on Reset
- $CDMMBase_{EN}$ - cleared to 0 on Reset
- $WatchLo_{I,R,W}$ - cleared to 0 on Reset
- *Config* fields related to static inputs - set to input value by Reset
- $Config_{K0}$ - set to 010 (uncached) on Reset
- $Config_{KU}$ - set to 010 (uncached) on Reset

- *Config*$_{K23}$ - set to 010 (uncached) on Reset
- *Debug*$_{DM}$ - cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into Debug Mode, as described in Chapter 13, “EJTAG Debug Support”.
- *Debug*$_{LSNM}$ - cleared to 0 on Reset
- *Debug*$_{IBusEP}$ - cleared to 0 on Reset
- *Debug*$_{DBusEP}$ - cleared to 0 on Reset
- *Debug*$_{IEXI}$ - cleared to 0 on Reset
- *Debug*$_{SSt}$ - cleared to 0 on Reset

12.1.2 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.

12.1.3 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.

12.1.4 Static Configuration Inputs

All static configuration inputs (for example, those defining the bus mode and cache size) should only be changed during Reset.

12.1.5 Fetch Address

Upon Reset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0x0000_BFC0_0000 (PA 0x00_1FC0_0000). This address is in kseg1, which is unmapped and uncached, so that the TLB and caches do not require hardware initialization.

12.2 Software-Initialized Processor State

Software is required to initialize parts of the device, as described below.

12.2.1 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.

12.2.2 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.

12.2.3 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized before exiting the boot code. There are various exceptions which 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* should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing kseg0.
- *Count*: Should be set to a known value if timer interrupts are used.
- *Compare*: Should be set to a known value if timer interrupts 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 write-able, 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.

12.3 System Boot-up

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
```

Chapter 13

EJTAG Debug Support

The EJTAG block provides a system debug facility for the device. The EJTAG functions are not normally controlled by the end user, but rather are controlled by a debugger. This chapter is meant to be read in conjunction with the MIPS EJTAG Specification that was included as part of the release.

An EJTAG debug block is present in all cores available from MIPS Technologies, Inc. It contains support for things like hardware and software breakpoints, hardware single-step, and a JTAG based debug TAP for debug probe connection.

This chapter is used for debug of the P6600 core. For more information on the debugging of the Multiprocessing System, including the CM2 and CPC, refer to the next chapter entitled “Multi-CPU Debug”

This chapter contains the following sections:

13.1 Overview

The EJTAG debug logic in the P6600 core is compliant with EJTAG Specification 6.0 and includes:

1. Standard core debug features

2. Optional hardware breakpoints

3. Standard Test Access Port (TAP) for a dedicated connection to a debug host

4. Optional PDtrace capability for program counter/data address/data value trace to On-chip memory or to Trace probe

EJTAG debug resources are often controlled via high level debugger commands. The following is a brief overview of some EJTAG features.

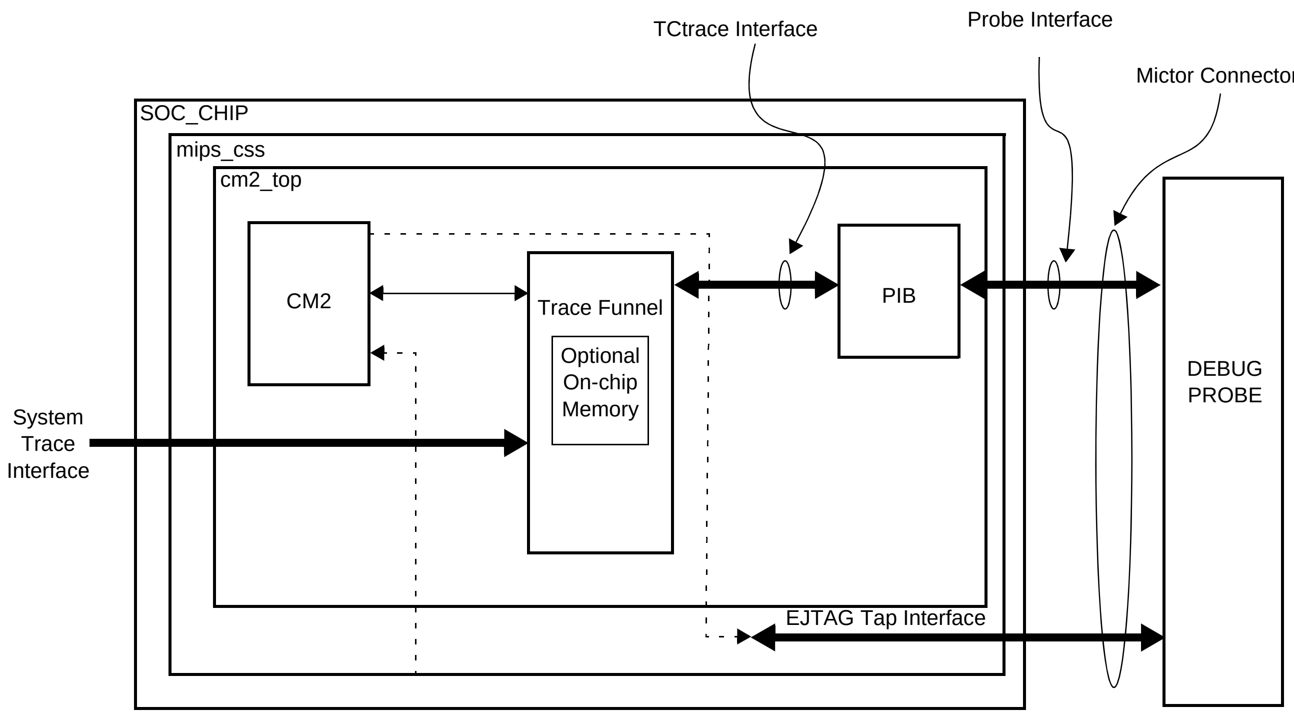
- PCSAMPLE: A feature allowing for non-intrusive reading of recently completed instruction addresses. The PCSAMPLE TAP instruction selects the TAP data register “PCSAMPLE” which contains an execution address and a flag indicating whether or not a new instruction has completed since the last read of the PCSAMPLE TAP data register.
- EJTAG TAP: The optional JTAG TAP associated with an EJTAG debug block used for communications with an EJTAG probe and debugger.
- ECR (EJTAG Control Register): This register is used mostly by probe developers and can only be accessed via a probe.
- DCR (Debug Control Register): This register is located in the drseg memory segment and can only be accessed in Debug mode.
- DINT (Debug Interrupt): an interrupt which causes a debug exception and entry into debug mode.
- DRSEG (Debug Register Segment): A memory overlay, present only while executing in debug mode, that allows access to registers controlling various EJTAG debug features.
- DMSEG (Debug Memory Segment): A memory overlay, present only while in debug mode and ECR.ProbEn is set, that an EJTAG probe emulates by satisfying processor accesses (fetches, loads, and stores.) The emulation is carried out via TAP data registers CONTROL, ADDRESS, and DATA.
- Single-Step: A debug setting that results in a debug exception after execution of a single12 non-debug mode instruction has completed.
- Hardware Breakpoint: A hardware resource capable of detecting execution or data access at virtual addresses.
- Software Breakpoint: The instruction “sdbbp” which causes a debug exception on execution. Debuggers will temporarily replace an instruction of your program with this instruction on setting a breakpoint in writeable memory.

13.2 Trace Funnel and Trace Types

The P6600 Multiprocessing System implements a trace funnel that is used to communicate with the debug probe via the probe interface block. The trace funnel can accept trace information from either the CM2, the core, or the MIPS system trace.

The trace funnel and its connections are shown in Figure 13.1. Refer to Section 13.2.1 "Trace Types" for more information on the types of traces shown.

Figure 13.1 Trace Connections in the MIPS Debug Architecture



13.2.1 Trace Types

The P6600 Multiprocessing System supports the following trace types:

1. CM2 Trace

2. System Trace

3. Core Trace

CM2 Trace — The CM2 has its own trace and also manages the trace funnel. The functionality of CM2 trace and the registers used to control it are described in the CM2 chapter. Refer to the Coherency Manager chapter of this manual for more information.

MIPS System Trace — The MIPS System trace is a new feature to the P6600 core and allows the SoC designer to place signals from their non-probe SoC logic directly into the trace funnel for PDTrace to capture. The logic and registers that controls System Trace are handled by the CM2. Refer to Section 3.6.2 of the P6600 Hardware User's Manual for more information on MIPS System Trace.

Core Trace — Core trace allows CPU signals to be traced and routed to the trace funnel for processing. The functionality of core trace and the registers used to control it are described throughout this chapter.

13.2.2 EJTAG TAP Interface

Every TAP register access (also referred to as a "scan") is a read-before-write operation. A TAP register access captures (reads) a register value from the target and then that value is serially shifted out to the tool as a new value is simultaneously shifted in. After all of the bits of the register have been shifted the input value is updated (written.)

There are two main paths through an EJTAG TAP state machine. One provides access to the single, 5-bit instruction register and the other provides access to the currently selected data register(s). Every TAP instruction access should result in the 5 bit binary value "00001" being read. Most EJTAG TAP instructions' sole purpose is to select which data register is accessed during a data scan. EJTAG TAP instructions not intended to select specific TAP data registers will select the BYPASS data register.

In a multi-device target system, the term "scan chain" is used to describe the serial (daisy-chained) set of TAPS which are read/written in a single scan.

13.2.3 EJTAGBOOT vs NORMALBOOT

The EJTAGBOOT TAP instruction modifies the reset value of the *ECR.ProbTrap*, *ECR.ProbEn*, and *ECR.EjtagBrk,* thereby changing device reset behavior. Subsequent warm resets result in a debug exception after release from reset. Any EJTAG TAP reset will clear the EJTAGBOOT indication as will sending a NORMALBOOT TAP instruction.

13.3 Detecting Debug Mode

The DM bit of the CP0 Debug register (CP0 Register 23, Select 0) indicates if the processor is operating in debug mode. If this bit is set, the processor is operating in debug mode. This bit is set on any debug exception and is cleared by executing a *DERET* instruction. Refer to Chapter 2, CP0 Registers, for more information on the *Debug* register.

This bit is available to both probe and non-probe related configurations and can be read at any time. The user does not need to be in Debug mode in order to read this bit. This bit, along with the associated fields in this register, can be used by software to determine the conditions under which Debug mode was entered.

13.4 Ways of Entering Debug Mode

There are five ways to enter Debug mode. Each of these ways can be entered from either software, or from a debug probe. All of these ways cause the *DM* bit in the *CP0 Debug* register to be set.

1. EJTAG Debug Single Step

2. EJTAG Debug Interrupt. Caused by the assertion of the external EJ_DINT input, or by setting the EJTAGBrk bit in the ECR register.

3. EJTAG debug hardware data breakpoint match

4. EJTAG debug hardware instruction breakpoint match

5. EJTAG Breakpoint (execution of SDBBP instruction)

13.4.1 EJTAG Debug Single Step

To enter Debug single step mode, the core must implement the single step mode. This can be determined by reading the *NoSST* bit (9) of the *CP0 Debug* register. If this bit is zero, the debug single step feature is implemented in the core. In the P6600 core, this bit is always zero to indicate that the single step feature is implemented by the core.

Single step mode can be enabled or disabled by writing to the *SST* bit (8) of the *CP0 Debug* register. If the *SST* bit is set, the single step function is available once the core enters debug mode using any of the ways listed above. For implementation that include a probe, the common way is to generate the EJTAG DINT signal, which causes a debug interrupt to the core. For non-probe implementations, software can set the EJTAGBRK bit. Both of these methods are described in the following subsection.

13.4.2 EJTAG Debug Interrupt

The EJTAG DINT signal is an implementation dependent feature. The *DINTsup* bit (24) in the *Implementation* register indicates whether the DINT signal is supported. This bit is written by the *EJ_DINTsup* signal at reset. This is a common way for probe or logic analyzer implementations to enter debug mode. Refer to Section 13.14.4.5 “Implementation Register” for more information.

Software can enter debug mode by setting the *EJTAGbrk* bit (12) or the *EJTAG Control* register. Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low power mode. This bit is cleared by hardware when the debug exception is taken. Refer to Section 13.14.4.6 “EJTAG Control Register” for more information.

13.4.3 EJTAG Hardware Data Breakpoint Match

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value. Refer to Section 13.8 “Hardware Breakpoints” for more information and a list of registers used to set up a data breakpoint.

13.4.4 EJTAG Hardware Instruction Breakpoint Match

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address used by the instruction fetch unit. Instruction breaks can also be made on the ASID value used by the MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions. Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a trigger is generated and a debug exception is optionally signalled. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

Refer to Section 13.8 “Hardware Breakpoints” for more information and a list of register used to set up an instruction breakpoint.

13.4.5 EJTAG Software Breakpoint

Software can execute a software debug breakpoint using the SDBBP instruction. When this instruction is executed, the debugger temporarily replaces the program instruction with the SDBBP instruction when setting a breakpoint in memory.

13.5 Exiting Debug Mode

As described above, there are five basic ways to enter debug mode. Once in debug mode, the mode can only be exited in one of three ways:

- Execution of a Debug Exception Return (DERET) instruction.
- Reset the core
- Power cycle the core

During normal operation, exceptions are taken by the core and processed. Once the exception processing is complete, software executes an Exception Return (ERET) instruction. When in debug mode, software executes a Debug Exception Return (DERET) instruction. This causes the core to exit debug mode and return to previous mode as determined by the programmer (normal, kernel, supervisor, etc.).

Note that for a DERET instruction to be executed, the core must be in a state where it is fetching instructions. If for any reason the instruction stream has been halted and cannot resume, then the DERET instruction cannot be executed. In this case, the only other options are resetting the core, or cycling the power to the P6600 core.

13.6 EJTAG and PDTrace Revisions

This chapter is intended to be used in conjunction with the EJTAG specification (MIPS document number MD00047) and the MIPS PDTrace specification (MIPS document number MD00439). These documents contain information for multiple types of MIPS cores, so the EJTAG and PDTrace versions of the core in question must be known in order to use these documents.

- ***EJTAG version with probe***: When using the MIPS Debug facility with a debug probe, the EJTAG version used in the P6600 core can be determined by reading the EJTAGver field in bits 31:29 of the *Implementation* register. This is a TAP controller register that is only accessible through an EJTAG probe. The P6600 core implements EJTAG revision 6.0. Refer to Section 13.14.4.5 "Implementation Register" for more information. Note that the probe can read either the *Implementation* register of the CP0 *Debug* register described below to determine the EJTAG revision number.
- ***EJTAG version without probe***: When using the MIPS Debug facility without a debug probe, the EJTAG version used in the P6600 core can be determined by reading the *EJTAGver* field in bits 17:15 of the CP0 *Debug* register located at CP0 register 23, select 0. The P6600 core implements EJTAG revision 6.0. Refer to Chapter 2 of this manual for more information on the CP0 *Debug* register. Note that the kernel can only read the CP0 *Debug* register to determine the EJTAG version and does not have access to the EJTAG *Implementation* register described above.
- ***PDTrace version with probe***: When using the MIPS Debug facility with a debug probe, the PDTrace version used in the P6600 core can be determined by reading the REV field in bits 3:0 of the *Trace Buffer Configuration* (TCBCONFIG) register located in the EJTAG TAP controller. Refer to the Section 13.14.10.7 "TCBCONFIG Register (Reg 0)" for more information on this register. The current revision is 3.0 as noted by the default value. Note that this register can only be read when an EJTAG probe is connected to the device.

- ***PDTrace version without probe***: When using the MIPS Debug facility without a debug probe, the PDTrace version used in the P6600 core can be determined by reading the REV field in bits 3:0 of the *Trace Buffer Configuration* (TCBCONFIG) register at offset 0x3028 in DRSEG.

  However, since a probe is not attached in this case, the core must be in Debug mode in order to read this register. Debug mode can be entered in any of the ways described in Section 13.4 "Ways of Entering Debug Mode". Refer to the Section 13.14.10.7 "TCBCONFIG Register (Reg 0)" for more information on this register.

It should be noted that the *Device Identification* register located in Section 13.14.4.4 on page 703 contains version and part number information. This register is only accessible when an EJTAG probe is attached, but does not provide EJTAG or PDTrace revision information. This register is used to by the manufacturer for their own device identification purposes and should not be used in an attempt to determine the EJTAG or PDTrace revisions.

13.7 Connection Options

The EJTAG debug port of the P6600 core can be accessed either via a TAP (five JTAG pins), or the EJTAG debug block through the CP0 Debug register, the DCR, and drseg space. If the TAP is used, no ROM monitor is required and there is no interference with the customers code. If there is no TAP, then the user must write their own ROM monitor.

There are two ways to connect to access the EJTAG debug facility:

- Software via the General Control Registers (GCR)
- Debug probe via the EJTAG Test Access Port (TAP)

The DCR (Debug Control Register) can be used to access the EJTAG debug port via software. This register is located in the drseg memory segment and can only be accessed in Debug mode. This register can be accessed by anyone that enters Debug mode and does not require that a probe be attached.

Access via software would mostly be performed during normal operation. As described in Section 13.4 "Ways of Entering Debug Mode" above, the CP0 Debug register (CP0 Register 23, Select 0) indicates whether or not the device is in Debug mode and the cause as to how it got there. Bit 30 of this register indicates if the core has entered Debug mode. If the core is not in Debug mode, the other bits have no meaning. If the core is in Debug mode, the other bits are used to provide additional information about how the device got into Debug mode. For example, setting a software breakpoint allows thc core to enter Debug mode.

The ECR (EJTAG Control Register) is used mostly by probe developers and can only be accessed via a probe. Refer to Section 13.14.4.6 "EJTAG Control Register" for more information.

13.8 Hardware Breakpoints

Hardware breakpoints provide for the comparison by hardware of executed instructions and data load/store transactions. It is possible to set instruction breakpoints on addresses even in ROM area. Data breakpoints can be set to cause a debug exception on a specific data transaction. Instruction and data hardware breakpoints are alike for many aspects, and are thus described in parallel in the following. The term hardware is not applied to breakpoint, unless required to distinguish it from software breakpoint.

There are two types of simple hardware breakpoints implemented in the P6600 core; Instruction breakpoints and Data breakpoints.

A core may be configured with the following breakpoint options:

- Four instruction breakpoints
- Two data breakpoints

13.8.1 Instruction Breakpoints

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address used by the instruction fetch unit. Instruction breaks can also be made on the ASID value used by the TLB-based MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions.

Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a trigger is generated and a debug exception is optionally signalled. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

13.8.2 Data Breakpoints

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), the ASID, accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a trigger is generated and a debug exception is optionally signalled. An internal bit in the data breakpoint registers is set to indicate that the match occurred.

13.8.3 Instruction Breakpoint Registers Overview

The register with implementation indication and status for instruction breakpoints in general is shown in Table 13.1.

Table 13.1 Overview of Status Register for Instruction Breakpoints

Register Mnemonic	Register Name and Description
IBS	Instruction Breakpoint Status

Up to four instruction breakpoints are available and are numbered 0 to 3 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 13.2.

Table 13.2 Overview of Registers for Each Instruction Breakpoint

Register Mnemonic	Register Name and Description
IBAn	Instruction Breakpoint Address n
IBMn	Instruction Breakpoint Address Mask n
IBASIDn	Instruction Breakpoint ASID n
IBCn	Instruction Breakpoint Control n

13.8.4 Data Breakpoint Registers Overview

The register with implementation indication and status for data breakpoints in general is shown in Table 13.3.

Table 13.3 Overview of Status Register for Data Breakpoints

Register Mnemonic	Register Name and Description
DBS	Data Breakpoint Status

Up to two data breakpoints are available and are numbered 0 and 1 for registers and breakpoints, and the number is indicated by *n*. The registers for each breakpoint are shown in Table 13.4.

Table 13.4 Overview of Registers for Each Data Breakpoint

Register Mnemonic	Register Name and Description
DBAn	Data Breakpoint Address *n*
DBMn	Data Breakpoint Address Mask *n*
DBASIDn	Data Breakpoint ASID *n*
DBCn	Data Breakpoint Control *n*
DBCSn	Data Breakpoint Control SIMD *n*
DBVn	Data Breakpoint Value *n*
DBVSn	Data Breakpoint Value SIMD *n*

13.8.5 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, as described in this section. Breakpoints only match for instructions executed in non-debug mode, never on instructions executed in debug mode.

The match of an enabled breakpoint always generates a trigger indication and can also generate a debug exception. The *BE* and/or *TE* bits in the *IBCn* or *DBCn* registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on an ASID value unless a TLB is present in the implementation.

13.8.5.1 Conditions for Matching Instruction Breakpoints

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on an instruction fetch. The breakpoint is not evaluated on instructions from a speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

A breakpoint match depends on the virtual address of the executed instruction (PC), which can be masked at the bit level, The match can also include an optional compare of the ASID value.

The registers for each instruction breakpoint contain the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

```
IB_match =
          ( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
          ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) &&
```

```
( (IBMn_ISAM | ~(ISAMode ^ IBAn_ISA))) )
```

The match indication for instruction breakpoints is always precise, i.e., indicated on the instruction causing the `IB_match` to be true.

13.8.5.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including coprocessor loads/stores and transactions causing an address error on data access. The breakpoint is not evaluated due to a PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

A breakpoint match depends on the transaction type (`TYPE`) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint contain the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

The overall match equation is the `DB_match`.

```
DB_match =
          ( ( ( TYPE == load ) && ! DBCn_NoLB ) ||
            ( ( TYPE == store ) && ! DBCn_NoSB ) ) &&
          DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

The match on the address part, `DB_addr_match`, depends on the virtual address of the transaction (`ADDR`), the `ASID` value, and the accessed bytes (`BYTELANE`) where `BYTELANE`[0] is 1 only if the byte at bits [7:0] on the bus is accessed, and `BYTELANE`[1] is 1 only if the byte at bits [15:8] is accessed, etc. The `DB_addr_match` is shown below.

```
DB_addr_match =
          ( ! DBCn_ASIDuse || ( ASID == DBASIDn_ASID ) ) &&
          ( <all 1's> == ( DBMn_DBM | ~ ( ADDR ^ DBAn_DBA ) ) ) &&
          ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of $DBCn_{\mathrm{BAI}}$ and `BYTELANE` is 8 bits. They are 8 bits to allow for data value matching on doubleword floating point loads and stores. For non-doubleword loads and stores, only the lower 4 bits will be used.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (`BYTELANE` as described above) accessed by the transaction, and the contents of breakpoint registers. The `DB_no_value_compare` is shown below.

```
DB_no_value_compare =
          ( <all 1's> == ( DBCn_BLM | DBCn_BAI | ~ BYTELANE ) )
```

The size of $DBCn_{\mathrm{BLM}}$, $DBCn_{\mathrm{BAI}}$ and `BYTELANE` is 8 bits.

In case a data value compare is required, `DB_no_value_compare` is false, then the data value from the data bus (`DATA`) is compared and masked with the registers for the data breakpoint. The endianess is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianess.

```
DB_value_match =
    ( ( DATA[7:0] == DBVn_DBV[7:0] ) || !BYTELANE[0] || DBCn_BLM[0] || DBCn_BAI[0] ) &&
    ( ( DATA[15:8] == DBVn_DBV[15:8] ) || !BYTELANE[1] || DBCn_BLM[1] || DBCn_BAI[1] ) &&
    ( ( DATA[23:16] == DBVn_DBV[23:16] ) || !BYTELANE[2] || DBCn_BLM[2] || DBCn_BAI[2] )&&
```

```
( ( DATA[31:24] == DBVn_DBV[31:24] ) || !BYTELANE[3] || DBCn_BLM[3] || DBCn_BAI[3] ) &&
( ( DATA[39:32] == DBVn_DBV[39:32] ) || !BYTELANE[4] || DBCn_BLM[4] || DBCn_BAI[4] ) &&
( ( DATA[47:40] == DBVn_DBV[47:40] ) || !BYTELANE[5] || DBCn_BLM[5] || DBCn_BAI[5] ) &&
( ( DATA[55:48] == DBVn_DBV[55:48] ) || !BYTELANE[6] || DBCn_BLM[6] || DBCn_BAI[6] ) &&
( ( DATA[63:56] == DBVn_DBV[63:56] ) || !BYTELANE[7] || DBCn_BLM[7] || DBCn_BAI[7] ))
```

The match for a data breakpoint without value compare is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true `DB_match` can thereby be indicated on the very same instruction causing the `DB_match` to be true. The match for data breakpoints with value compare is always imprecise.

13.8.5.3 Misaligned SIMD Load/Store Data Handling

Misaligned SIMD load/store data requires a pair of breakpoint register sets to support the breakpointing on misaligned 128-bit wide data. This example assumes that an even numbered register set, labelled n, and an adjacent odd register set labelled n+1 have been initialized for this purpose. It is assumed that the SIMD load/store is processed as two separate 128-bit aligned requests, such that the even register set applies to access with the lower address, while the odd register set applies to the access with the upper address. A single request is assumed to be 128-bit aligned, though the address is aligned to the bytes sourced for the load/store.

Software must take into account the endianness of the access while programming the pair. The pseudo-code and thus implementation itself need not differentiate based on endianness.

// Even Register Set Breakpoint Handling Pseudo-Code:

```
DB_match_even =
(!DBCnTCuse || ( TC == DBCnTC ) ) &&
( ( ( TYPE == load ) && ! DBCnNoLB ) || ( ( TYPE == store ) && ! DBCnNoSB ) ) &&
DB_addr_match_even && ( DB_no_value_compare_even || DB_value_match_even)

DB_addr_match_even =
( ! DBCnASIDuse || ( ASID == DBASIDnASID ) ) &&
( ! DBASIDnUGID || ( GuestID == DBASIDnGuestID ) ) &&
( ( DBMnDBM | ~ ( ADDR ^ DBAnDBA ) ) == ~0 ) &&
( ( ~ DBCnBAI & BYTELANE ) != 0 )

DB_no_value_compare_even =
( ( DBCnBLM | DBCnBAI | ~ BYTELANE ) == ~0 )

// Bytes 15:0 on data-bus are checked for match with even register set. Value
// match is extended with new DBCS and DBVS registers of even set.

DB_value_match_even =
DBCnIVM ^
( ( DATA[7:0] == DBVnDBV[7:0] ) || ! BYTELANE[0] || DBCnBLM[0] || DBCnBAI[0] ) &&
( ( DATA[15:8] == DBVnDBV[15:8] ) || ! BYTELANE[1] || DBCnBLM[1] || DBCnBAI[1] ) &&

......

( ( DATA[63:56] == DBVnDBV[63:56] ) || ! BYTELANE[7] || DBCnBLM[7] || DBCnBAI[7] ) &&
( ( DATA[71:64] == DBVSnDBV[71:64] ) || ! BYTELANE[8] || DBCSnBLM[8] || DBCSnBAI[8] )

&&
```

```
......

( ( DATA[127:120] == DBVSnDBV[127:120] ) || ! BYTELANE[15] || DBCSnBLM[15] ||
DBCSnBAI[15] )

// Odd Register Set Breakpoint Handling Pseudo-Code:

DB_match_odd =
(!DBCn+1TCuse || ( TC == DBCn+1TC ) ) &&
( ( ( TYPE == load ) && ! DBCn+1NoLB ) || ( ( TYPE == store ) && ! DBCn+1NoSB ) ) &&
DB_addr_match_odd && ( DB_no_value_compare_odd || DB_value_match_odd )

DB_addr_match_odd =
( ! DBCn+1ASIDuse || ( ASID == DBASIDn+1ASID ) ) &&
( ! DBASIDn+1UGID || ( GuestID == DBASIDn+1GuestID ) ) &&
( ( DBMn+1DBM | ~ ( ADDR ^ DBAn+1DBA ) ) == ~0 ) &&
( ( ~ DBCn+1BAI & BYTELANE ) != 0 )

DB_no_value_compare_odd =
( ( DBCn+1BLM | DBCn+1BAI | ~ BYTELANE ) == ~0 )

// Bytes 15:0 on data-bus are checked for match with odd register set. Value
// match is extended with new DBCS and DBVS registers of odd set.

DB_value_match_odd =
DBCnIVM ^
( ( DATA[7:0] == DBVn+1DBV[7:0] ) || ! BYTELANE[0] || DBCn+1BLM[0] || DBCn+1BAI[0] ) &&
( ( DATA[15:8] == DBVn+1DBV[15:8] ) || ! BYTELANE[1] || DBCn+1BLM[1] || DBCn+1BAI[1] )

&&

......
( ( DATA[63:56] == DBVn+1DBV[63:56] ) || ! BYTELANE[7] || DBCn+1BLM[7] || DBCn+1BAI[7])
( ( DATA[71:64] == DBVSn+1DBV[71:64] ) || ! BYTELANE[8] || DBCSn+1BLM[8] ||
DBCSn+1BAI[8] ) &&

......
( ( DATA[127:120] == DBVSn+1DBV[127:120] ) || ! BYTELANE[15] || DBCSn+1BLM[15] ||
DBCSn+1BAI[15] )

// Merging Odd and Even pseudo-code results:
// The equation assumes the matches are detected at different times, but
// are synchronized at some point, such as at graduation of the instruction.
// The pseudo-function IsMisAlignedAccess() functions as follows :
//          1: The address is not aligned to the type. E.g., a word load is not
// word-aligned, a SIMD load is not 16-byte aligned.
//          0: The address is aligned to the type. E.g., a word load is word-aligned, a
// SIMD load is 16-byte aligned.
if (DBCnGM=1 && IsMisAlignedAccess())then
          DB_match = DB_match_even && DB_match_odd
else
          // The register sets are independent and thus any may source match
          DB_match = DB_match_even || DB_match_odd
```

13.8.6 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be set up to generate a debug exception when the match condition is true, as described below.

13.8.6.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by the *BE* bit in the *IBCn* register, then a debug instruction break exception occurs if the IB_match equation is true. The corresponding *BS*[n] bit in the *IBS* register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the *DEPC* register and the *DBD* bit in the *Debug* register point to the instruction that caused the `IB_match` equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint cannot occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction; otherwise the debug instruction break exception reoccurs.

13.8.6.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by *BE* bit in the *DBCn* register, then a debug exception occurs when the DB_match condition is true. The corresponding *BS*[n] bit in the *DBS* register is set when the breakpoint generates the debug exception. A matching data breakpoint generates either a precise or imprecise debug exception.

Debug Data Break Load/Store Exception as a Precise Debug Exception

A precise debug data break exception occurs when a data breakpoint without value compare indicates a match. In this case the *DEPC* register and *DBD* bit in the *Debug* register points to the instruction that caused the `DB_match` equation to be true.

The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.
- A load transaction with no data value compare, i.e. where the `DB_no_value_compare` is true for the match, is not allowed to complete the load.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the rules shown in Table 13.5 apply with respect to updating the *BS*[n] bits.

Table 13.5 Rules for Update of BS Bits on Data Breakpoint Exceptions

Instruction	Breakpoints that Match		Update of BS Bits for Matching Data Breakpoints	
	Without Value Compare	With Value Compare	Without Value Compare	With Value Compare
Load/Store	One or more	None	BS bits set for all	(No matching break-points)
Load	One or more	One or more	BS bits set for all	Unchanged BS bits since load of data value does not occur so match of the breakpoint cannot be determined
Load	None	One or more	(No matching break-points)	BS bits set for all
Store	One or more	One or more	BS bits set for all	BS bits set for all
Store	None	One or more	(No matching break-points)	BS bits set for all

Any *BS*[n] bit set prior to the match and debug exception are kept set, since *BS*[n] bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

Debug Data Break Load/Store Exception as a Imprecise Debug Exception

An Debug Data Break Load/Store Imprecise exception occurs when a data breakpoint indicates an imprecise match. Imprecise matches are generated when data value compare is used. In this case, the *DEPC* register and *DBD* bit in the Debug register point to an instruction later in the execution flow rather than at the load/store instruction that caused the `DB_match` equation to be true.

The load/store instruction causing the Debug Data Break Load/Store Imprecise exception always updates the destination register and completes the access to the external memory system. Therefore this load/store instruction is not re-executed on return from the debug handler, because the *DEPC* register and *DBD* bit do not point to that instruction.

Several imprecise data breakpoints can be pending at a given time, if the bus system supports multiple outstanding data accesses. The breakpoints are evaluated as the accesses finalize, and a Debug Data Break Load/Store Imprecise exception is generated only for the first one that matches. Both the first and succeeding matches cause corresponding *BS* bits and *DDBLImpr*/*DDBSImpr* to be set, but no debug exception is generated for succeeding matches, because the processor is already in Debug Mode. Similarly, if a debug exception had already occurred at the time of the first match (for example, due to a precise debug exception), then all matches cause the corresponding *BS* bits and *DDBLImpr*/*DDBSImpr* to be set, but no debug exception is generated because the processor is already in Debug Mode.

The SYNC instruction, followed by appropriate spacing must be executed before the *BS* bits and *DDBLImpr*/*DDBSImpr* bits are accessed for read or write. This delay ensures that these bits are fully updated.

Any *BS* bit set prior to the match and debug exception remains set, because only debug software can clear the *BS* bits.

13.8.7 Breakpoint used as Triggerpoint

When an enabled instruction or data breakpoint matches, the corresponding bit in the *IBS.BS* or *DBS.BS* field is set. These fields are externalized on the *SI_Ibs* and *SI_Dbs* core outputs, respectively. These outputs are intended to be used to trigger external devices such as logic analyzers. Furthermore, breakpoint matches can also be used to start or stop PDtrace. See Section 13.11.8 "Enabling PDtrace" for details.

If the breakpoints are to be used only as trigger events, the signalling of the debug exception can be suppressed by clearing the *IBCn/DBCn.BE* field and setting the *IBCn/DBCn.TE* field.

13.9 Debug Vector Addressing

The debug vector address size is managed by the *Debug Vector Address* register as described in Section 13.14.1.2 "DebugVectorAddr Register". The *Debug Vector Address* register is a read/write register containing the base address of the debug exception vectors in bits 31:7, and a WG bit that determines whether the bits 31:30 of this field are a fixed value, or are programmable.

Bits 31:12 of the *DebugVectorAddress* register are concatenated with zeros to form the base of the debug exception vector. The exception vector base address comes from the fixed defaults for any EJTAG Debug exception. The reset state of bits 31:12 of the *DebugVectorAddress* register initialize the exception base register to `0xFC00.0480`.

The size of the *DebugVectorAddr* field depends on the state of the WG bit. At reset, the WG bit is cleared by default. In this case, the *DebugVectorAddr* field is comprised of bits 29:7. Bits 31:30 of the *DebugVectorAddr* Register are not writeable and are forced to a value of 2'b10 by hardware so that the debug exception handler will be executed from the *kseg0/kseg1* segments.

When the WG bit is set, bits 31:30 of the *DebugVectorAddr* field become writeable and are used to relocate the *DebugVectorAddr* field to other segments after they have been setup using the *SegCtl0* through *SegCtl2* registers. Note that if the WG bit is set by software (allowing bits 31:30 to become part of the *DebugVectorAddr* field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after WG was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the *DebugVectorAddr* register prior to clearing the WG bit if it wants to ensure that future debug exceptions will be executed from the kseg0 or kseg1 segments.

Note that the WG bit is different from the CV bit in the SegCtl0 register. Although their functions are similar, the CV bit applies only to cache error exceptions, whereas the WG bit applies to all exceptions.

If the value of the exception base register is to be changed, this must be done with *StatusBEV* equal to 1. The operation of the processor is **UNDEFINED** if the exception base field is written with a different value when *StatusBEV* is 0.

Table 13.6 shows the different debug exception vector locations that are possible.

Table 13.6 Debug Exception Vectors

$ECR_{ProbTrap}$	DCR_{RdVec}	$Config5_K$	SI_UseExceptionBase	Cache Error?	Debug Exception Vector
1	x	x	x	x	0xFFFF_FFFF_FF20_0200
0	1	0	x	0	0xFFFF_FFFF || 2'b10 || DebugVectorAddr[29:0]
0	1	1	x	0	0xFFFF_FFFF || DebugVectorAddr[31:0]
0	1	0	x	1	0xFFFF_FFFF || 3'b101 || DebugVectorAddr[28:0]
0	1	1	x	1	0xFFFF_FFFF || DebugVectorAddr[31:0]

Table 13.6 Debug Exception Vectors *(continued)*

$ECR_{ProbTrap}$	DCR_{RdVec}	$Config5_K$	SI_UseExceptionBase	Cache Error?	Debug Exception Vector
0	0	0	1	0	0xFFFF_FFFF || 2'b10 || SI_ExceptionBase[29:12] || 0x480
0	0	1	1	0	0xFFFF_FFFF || SI_ExceptionBase[31:12] || 0x480
0	0	0	1	1	0xFFFF_FFFF || 3'b101 || SI_ExceptionBase[28:12] || 0x480
0	0	1	1	1	0xFFFF_FFFF || SI_ExceptionBase[31:12] || 0x480
0	0	x	0	x	0xFFFF_FFFF_BFC0_0480

As shown in the table above, if the $ECR_{ProbeTrap}$ bit (14) is set in the EJTAG Control register, then all other bits or signals that determine the location of the debug vector address have no meaning and the location of the debug exception vector default to 0xFFFF_FFFF_FF20_0200. Note that the $ECR_{ProbeEn}$ bit (15) must be set in order for this bit to have meaning.

13.10 Test Access Port (TAP)

The TAP is used only when a probe is connected to the P6600 core.

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST_N*) interface which is compatible with IEEE Std. 1149.1.
- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.
- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through Processor Access (PA), and is used to eliminate the use of the system memory for debug routines.
- Support for both ROM based debugger and debugging both through TAP.

13.10.1 EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

Table 13.7 EJTAG Interface Pins

Pin	Type	Description
TCK	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the processor clock, so the EJTAG probe can drive *TCK* independently of the processor clock frequency. The core signal for this is called *EJ_TCK*
TMS	I	Test Mode Select Input The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*. The core signal for this is called *EJ_TMS*

Table 13.7 EJTAG Interface Pins*(continued)*

Pin	Type	Description
TDI	I	Test Data Input Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state. The core signal for this is called *EJ_TDI*
TDO	O	Test Data Output Serial output data is shifted from the Instruction or data register to the *TDO* pin on the falling edge of the *TCK* clock. When no data is shifted out, the *TDO* is 3-stated. The core signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*.
TRST_N	I	Test Reset Input (Optional pin) The *TRST_N* pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of *TRST_N*. The core signal for this is called *EJ_TRST_N* This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.

13.10.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 13.2. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

At power-up the TAP is forced into the *Test-Logic-Reset* by low value on *TRST_N*. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

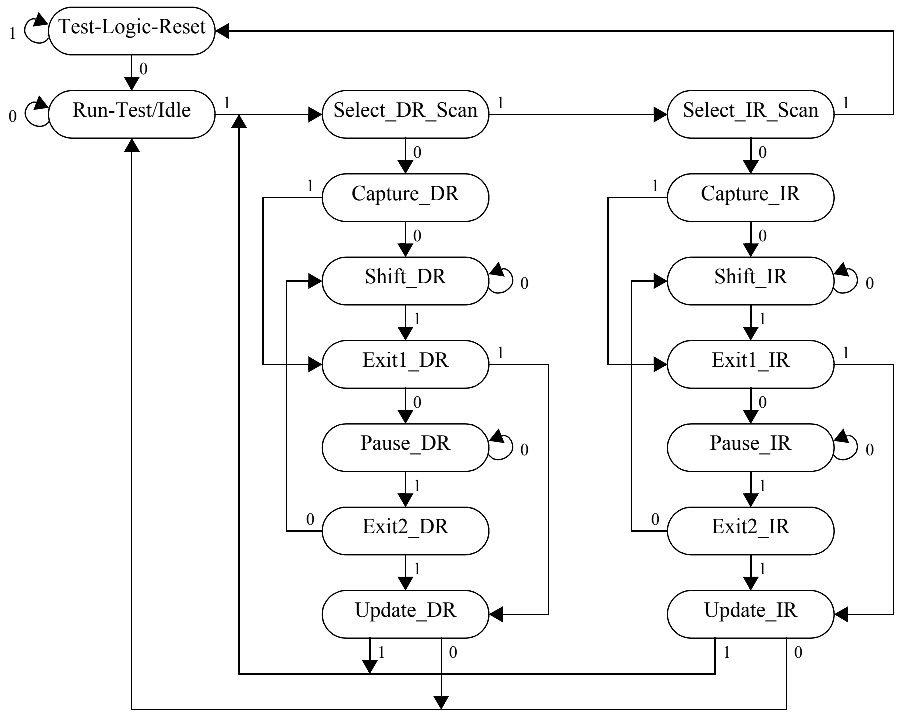
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 13.2.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the *Capture* states, the TAP transitions to either the *Shift* or *Exit1* states. Normally the *Shift* state follows the *Capture* state so that test data or status information can be shifted out for inspection and new data shifted in. Following the *Shift* state, the TAP either returns to the *Run-Test/Idle* state via the *Exit1* and *Update* states or enters the *Pause* state via *Exit1*. The reason for entering the *Pause* state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the Pause state shifting can resume by re-entering the *Shift* state via the *Exit2* state or terminate by entering the *Run-Test/Idle* state via the *Exit2* and *Update* states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the *Update-DR* or *Update-IR* state. The *Update* state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.

Figure 13.2 TAP Controller State Diagram



13.10.2.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the *TMS* input is held HIGH for at least five rising edges of *TCK*. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as *TMS* is HIGH.

13.10.2.2 Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as *TMS* is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When *TMS* is sampled HIGH on the rising edge of *TCK*, the controller transitions to the *Select_DR* state.

13.10.2.3 Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Capture_DR* state. A HIGH on *TMS* causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.4 Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller transitions to the *Capture_IR* state. A

HIGH on *TMS* causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.5 Capture_DR State

In this state the boundary scan register captures the value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift_DR*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.6 Shift_DR State

In this state the test data register connected between *TDI* and *TDO* as a result of the current instruction shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.7 Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

13.10.2.8 Pause_DR State

The *Pause_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_DR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.9 Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

13.10.2.10 Update_DR State

When the TAP controller is in this state the value shifted in during the *Shift_DR* state takes effect on the rising edge of the *TCK* for the register indicated by the Instruction register.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

13.10.2.11 Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001_2) on the rising edge of *TCK*. The data registers selected by the current instruction retain their previous state.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.12 Shift_IR State

In this state the instruction register is connected between *TDI* and *TDO* and shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state.

13.10.2.13 Exit1_IR State

This is a temporary controller state in which all registers retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

13.10.2.14 Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_IR* state. The instruction cannot change while the TAP controller is in this state.

13.10.2.15 Exit2_IR State

This is a temporary controller state in which the instruction register retains its previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

13.10.2.16 Update_IR State

The instruction shifted into the instruction register takes effect on the rising edge of *TCK*.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state.

13.10.3 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the *Shift-IR* state. Instructions are decoded and define the serial test data register path that is used to shift data between *TDI* and *TDO* during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions default to the BYPASS instruction.

Table 13.8 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation register
0x08	ADDRESS	Select Address register

Table 13.8 Implemented EJTAG Instructions *(continued)*

Value	Instruction	Function
0x09	DATA	Select Data register
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value
0x0D	NORMALBOOT	Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x10	TCBCONTROLA	Selects the *TCBTCONTROLA* register in the Trace Control Block
0x11	TCBCONTROLB	Selects the *TCBTCONTROLB* register in the Trace Control Block
0x12	TCBDATA	Selects the *TCBDATA* register in the Trace Control Block
0x13	TCBCONTROLC	Selects the *TCBTCONTROLC* register in the Trace Control Block
0x14	PCSAMPLE	Selects the *PCSAMPLE* register
0x15	TCBCONTROLD	Selects the *TCBTCONTROLD* register in the Trace Control Block
0x16	TCBCONTROLE	Selects the *TCBTCONTROLE* register in the Trace Control Block
0x17	FDC	Select Fast Debug Channel
0x1F	BYPASS	Bypass mode

13.10.3.1 BYPASS Instruction

The required BYPASS instruction allows the processor to remain in a functional mode and selects the Bypass register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the processor from *TDI* to *TDO* without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

13.10.3.2 IDCODE Instruction

The IDCODE instruction allows the processor to remain in its functional mode and selects the Device Identification (ID) register to be connected between *TDI* and *TDO*. The Device ID register is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the processor. Also, access to the Identification Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional *TRST_N* pin.

13.10.3.3 IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

13.10.3.4 ADDRESS Instruction

This instruction is used to select the 64-bit Address register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 64 bits through the *TDI* pin into the Address register and shifts out the captured address via the *TDO* pin.

13.10.3.5 DATA Instruction

This instruction is used to select the 64-bit Data register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 64 bits of *TDI* data into the Data register and shifts out the captured data via the *TDO* pin.

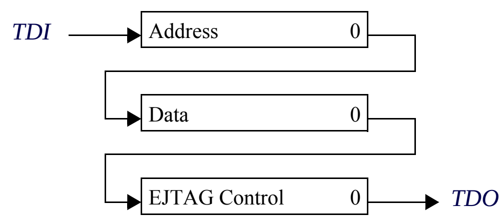
13.10.3.6 CONTROL Instruction

This instruction is used to select the EJTAG Control register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the EJTAG Control register and shifts out the EJTAG Control register bits via *TDO*.

13.10.3.7 ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register (ECR) between *TDI* and *TDO*. It can be used in particular to minimize the overhead in switching the instruction in the instruction register. The first bit shifted out is bit 0 of the ECR.

Figure 13.3 Concatenation of the EJTAG Address, Data and Control Registers



13.10.3.8 EJTAGBOOT Instruction

EJTAGBOOT provides a means to enter debug mode just after a reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which has no code in ROM.

When the EJTAGBOOT instruction is given and the Update-IR state is left, the EJTAGBOOT indication will become active. When EJTAGBOOT is active, a core reset will set the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register to 1. This will cause a debug exception that is serviced by the probe immediately after reset is deasserted.

This EJTAGBOOT indication is effective until a NORMALBOOT instruction is given, *TRST_N* is asserted or a rising edge of *TCK* occurs when the TAP controller is in Test-Logic-Reset state.

The Bypass register is selected when the EJTAGBOOT instruction is given.

13.10.3.9 NORMALBOOT Instruction

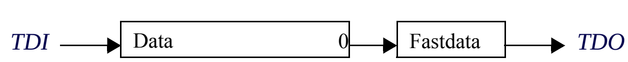
When the NORMALBOOT instruction is given and the Update-IR state is left, then the EJTAGBOOT indication will be cleared. When NORMALBOOT is active (EJTAGBOOT is not active), a core reset will set the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register to 0.

The Bypass register is selected when the NORMALBOOT instruction is given.

13.10.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 13.4.

Figure 13.4 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected



The FASTDATA access is used for efficient block transfers between dmseg (on the probe) and target memory (on the processor). An “upload” is defined as a sequence of processor loads from target memory and stores to dmseg. A “download” is a sequence of processor loads from dmseg and stores to target memory. The “Fastdata area” specifies the legal range of dmseg addresses (0xFFFF.FFFF.FF20.0000 - 0xFFFF.FFFF.FF20.000F) that can be used for uploads and downloads. The Data + Fastdata registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The PrAcc (processor access pending bit) will be 1 indicating the probe is required to complete the access. Both upload and download accesses are attempted by shifting in a zero *SPrAcc* value (to request access completion) and shifting out *SPrAcc* to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from dmseg’s Fastdata area, while uploads will shift out the data being stored to dmseg’s Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed. These are:

- *PrAcc* must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in dmseg (0xFFFF.FFFF.FF20.0000 to 0xFFFF.FFFF.FF20.000F).

Table 13.9 shows the values of the *PrAcc* and *SPrAcc* bits and the results of a Fastdata access.

Table 13.9 Operation of the FASTDATA Access

Probe Operation	Address Match Check	PrAcc in the Control Register	LSB (SPrAcc) Shifted In	Action in the Data Register	PrAcc Changes to	Lsb Shifted Out	Data Shifted Out
Download using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There is no restriction on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code and the probe software. Note that the most efficient transfer size is a 64-bit double-word.

The Rocc bit of the Control register is not used for the FASTDATA operation.

13.10.3.11 TCBCONTROLA Instruction

This instruction is used to select the TCBCONTROLA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

13.10.3.12 TCBCONTROLB Instruction

This instruction is used to select the TCBCONTROLB register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

13.10.3.13 TCBCONTROLC Instruction

This instruction is used to select the TCBCONTROLC register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

13.10.3.14 TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

13.10.3.15 PCSAMPLE Instruction

This instruction is used to select the PCSAMPLE register to be connected between *TDI* and *TDO*. This register is always implemented.

13.10.3.16 TCBCONTROLD Instruction

This instruction is used to select the TCBCONTROLD register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

13.10.3.17 TCBCONTROLE Instruction

This instruction is used to select the TCBCONTROLE register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

13.10.3.18 FDC Instruction

This instruction is used to select the Fast Debug Channel register to be connected between *TDI* and *TDO*. This register is always implemented.

13.10.4 TAP Processor Accesses

The TAP modules support handling of fetches, loads and stores from the CPU through the dmseg segment, whereby the TAP module can operate like a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via a load or store) which is located on the EJTAG Probe. This occurs in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFFFF.FFFF.FF20.0000 to 0xFFFF.FFFF.FF2F.FFFF, the ProbEn bit is set, and the processor is in debug mode (DM=1). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the ProbTrap bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to PrAcc or by a reset.

13.10.4.1 Fetch/Load and Store From/To the EJTAG Probe Through dmseg

1. The internal hardware latches the requested address into the Address register (in case of the Debug exception: 0xFF20.0200).

2. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 0 (selects processor read operation)
   Psz[1:0] = value depending on the transfer size

3. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

4. The EJTAG Probe checks the PRnW bit to determine the required access.

5. The EJTAG Probe selects the Address register and shifts out the requested address.

6. The EJTAG Probe selects the Data register and shifts in the instruction corresponding to this address.

7. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the instruction is available.

8. The instruction becomes available in the instruction register and the processor starts executing.

9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFFFF_FFFF_FF20.0000 to 0xFFFF_FFFF_FF2F.FFFF, the ProbEn bit must be set and the processor has to be in debug mode (DM=1). The sequence of actions is found below:

1. The internal hardware latches the requested address into the Address register

2. The internal hardware latches the data to be written into the Data register.

3. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 1 (selects processor write operation)
   Psz[1:0] = value depending on the transfer size

4. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

5. The EJTAG Probe checks the PRnW bit to determine the required access.

6. The EJTAG Probe selects the Address register and shifts out the requested address.

7. The EJTAG Probe selects the Data register and shifts out the data that was written.

8. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the write access is finished.

9. The EJTAG Probe writes the data to the appropriate address in its memory.

10. The processor detects that PrAcc bit = 0, which means that it is ready to handle a new access.

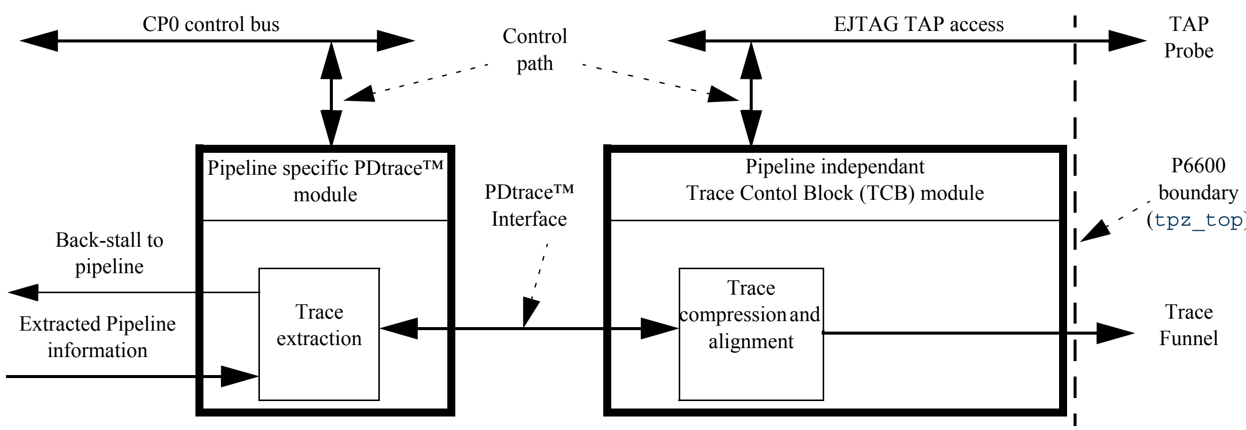
The above examples imply that no reset occurs during the operations, and that Rocc is cleared.

13.11 PDTrace

PDTrace enables the ability to trace program flow, load/store addresses and load/store data. Several run-time options exist for the level of information which is traced, including tracing only when in specific processor modes (e.g., UserMode or KernelMode).

There are two primary blocks involved in the PDtrace solution. The pipeline specific part of PDtrace is called the PDtrace module. It extracts the trace information from the processor pipeline, and presents it to a pipeline-independent module called the Trace Control Block (TCB). While working closely together, the two parts of PDtrace are controlled separately by software. Figure 13.5 shows an overview of the PDtrace modules within the core.

Figure 13.5 MIPS® Trace Modules in the P6600™ Core



To some extent, the two modules both provide similar trace control features, but the access to these features is quite different. The PDtrace controls can only be reached through access to CP0 registers. In general, the PDtrace control registers select what information is captured for tracing.The TCB controls can be reached through EJTAG TAP access or through load/store access to registers mapped in drseg space. The TCB registers control what is traced through the PDtrace™ Interface.

Before describing the PDtrace implemented in the P6600 core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

13.11.1 Processor Modes

Tracing can be enabled or disabled based on various processor modes. This section precisely describes these modes. The terminology is then used elsewhere in the document.

```
DebugMode ← (Debug_DM = 1)
ExceptionMode ← (not DebugMode) and ((Status_EXL = 1) or (Status_ERL = 1))
KernelMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#00)
SupervisorMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#01)
UserMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#10)
```

13.11.2 Software Versus Hardware Control

In some of the specifications and in this text, the terms "software control" and "hardware control" are used to refer to the method for how trace is controlled. Software control is when the CP0 register *TraceControl* is used to select the modes to trace, etc. Hardware control is when the EJTAG register *TCBCONTROLA* in the TCB, via the PDtrace interface, is used to select the trace modes. The $TraceControl_{TS}$ bit determines whether software or hardware control is active.

13.11.3 Trace Information

The main object of trace is to show the exact program flow from a specific program execution or just a small window of the execution. In PDtrace this is done by providing the minimal cycle-by-cycle information necessary on the PDtrace™ interface for trace regeneration software to reproduce the trace. The following is a summary of the type of information traced:

- Only instructions which complete at the end of the pipeline are traced, and indicated with a completion-flag. The PC is implicitly pointing to the next instruction.
- Load instructions are indicated with a load-flag.
- Store instructions are indicated with a store-flag[1].
- Taken branches are indicated with a branch-taken-flag on the target instruction.
- New PC information for a branch is only traced if the branch target is unpredictable from the static program image.
- When branch targets are unpredictable, only the delta value from current PC is traced, if it is dynamically determined to reduce the number of bits necessary to indicate the new PC. Otherwise the full PC value is traced.
- When a completing instruction is executed in a different processor mode from the previous one, the new processor mode is traced.
- The first instruction is always traced as a branch target, with processor mode and full PC.
- Periodic synchronization instructions are identified with a sync-flag, and traced with the processor mode and full PC.

1. A SC (Store Conditional) instruction is not flagged as a store instruction if the load-locked bit prevented the actual store.

All the instruction flags above are combined into one 4-bit value to minimize the bit information to trace.

The target address is statically predictable for all branch and all jump-immediate instructions. If the branch is taken, then the branch-taken-flag will indicate this. All jump-register instructions and ERET/DERET are instructions which have an unpredictable target address. These will have full/delta PC values included in the trace information. Also treated as unpredictable are PC changes which occur due to exceptions, such as an interrupt, reset, etc.

Trace regeneration software is required to know the static program image in memory, in order to reproduce the dynamic flow with the above information. Only the virtual value of the PC is used. Physical memory location will typically differ.

It is possible to turn on PC delta/full information for all branches, but this should not normally be necessary. As a safety check for trace regeneration software, a periodic synchronization with a full PC is sent. The period of this synchronization is cycle based and programmable.

13.11.4 Load/Store Address and Data Trace Information

In addition to PC flow, it is possible to get information on the load/store addresses, as well as the data read/written. When enabled, the following information is optionally added to the trace.

- When load-address tracing is on, the full load address of the first load instruction is traced (indicated by the load-flag). For subsequent loads, a dynamically-determined delta to the previous load address is traced to compress the information which must be sent.
- When store-address tracing is on, the full store address of the first store instruction is traced (indicated by the store-flag). For subsequent stores, a dynamically-determined delta to the previous store address is traced.
- When load-data tracing is on, the full load data read by each load instruction is traced (indicated by the load-flag). Only actual read bytes are traced.
- When store-data tracing is on, the full store data written by each store instruction is traced (indicated by the store-flag). Only written bytes are traced.

Note that the P6600 core does not support full data tracing of 128 bit load/stores. In case of 128 bit data, only the lower 64 bits are traced together with an additional information for the regeneration software, informing about the missing upper 64 bits. For more details please refer to MIPS PDTrace specification (MIPS document number MD00439) Section 4.1.4.1 and Appendix F.10.

After each synchronization instruction, the first load address and the first store address following this are both traced with the full address if load/store address tracing is enabled.

13.11.5 Programmable Processor Trace Mode Options

To enable tracing, a global Trace On signal must be set. When trace is on, it is possible to enable tracing in any combination of the processor modes described in Section 13.11.1 "Processor Modes". In addition to this, trace can be turned on globally for all processes, or only for specific processes by tracing only specific masked values of the ASID found in $EntryHi_{ASID}$. Tracing can also be qualified with GuestID for VZ systems.

Additionally, an EJTAG Simple Break trigger point can override the processor mode and ASID selection and turn them all on. Another trigger point can disable this override again.

13.11.6 Programmable Trace Information Options

The processor mode changes are always traced:

- On the first instruction.
- On any synchronization instruction.
- When the mode changes and either the previous or the current processor mode is selected for trace.

The amount of extra information traced is programmable to include:

- PC information only.
- PC and cross product of load/store address/data
- Performance counter values, if the optional performance counter trace is enabled.

If the full internal state of the processor is known prior to trace start, PC and load data are the only information needed to recreate all register values on an instruction by instruction basis.

13.11.6.1 User Data Trace

Two special CP0 registers, *UserTraceData1* and *UserTraceData2,* can generate a data trace. When either of these registers is written, and the global Trace On is set, then the 32-bit data written is put in the trace as special User Data information. Since writing these registers is performed via an MTC0 operation, only one register is updated in any given cycle. Thus in the same cycle, only one of the UserTraceData registers is traced. However in back to back cycles, the tracing of the two registers can alternate, and is handled correctly.

Remark: The User Data is sent even if the processor is operating in an un-traced processor mode.

13.11.7 Enable Trace to Probe On-Chip Memory

When trace is On, based on the options listed in Section 13.11.5 "Programmable Processor Trace Mode Options", the trace information is continuously sent on the PDtrace™ interface to the TCB. The TCB must be enabled to transmit the trace information to the Trace funnel by having the $TCBCONTROLB_{EN}$ bit set. It is possible to enable and disable the TCB in a number of ways:

- Set/clear the $TCBCONTROLB_{EN}$ bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the $TCBCONTROLB_{EN}$ bit.
- Use the drseg mapping of *TCBCONTROLB* to clear $TCBCONTROLB_{EN}$ via a store to drseg space.

13.11.8 Enabling PDtrace

As there are several ways to enable tracing, it can be quite confusing to figure out how to turn tracing on and off. This section should help clarify the enabling of trace.

13.11.8.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

Hardware instruction/data simple breakpoints in the P6600 core can be used as triggers to start/stop trace. When used for this, the breakpoints need not also generate a debug exception, but are capable of only generating an internal trigger to the trace logic. This is done by only setting the TE bit and not the BE bit in the Breakpoint Control register.

Please see Section 13.14.2.5 “Instruction Breakpoint Control n (IBCn) Register” and Section 13.14.3.5 “Data Breakpoint Control n (DBCn) Register” for details on breakpoint control.

In connection with the breakpoints, the Trace BreakPoint Control (*TraceBPC*) register is used to define the trace action when a trigger happens. When a breakpoint is enabled as a trigger (TE = 1), it can be selected to be either a start or a stop trigger to the trace logic.

13.11.8.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, with the exception that the bits in the control register are used instead of the input enable signals from the TCB. The $TraceControl_{TS}$ bit controls whether hardware (via the TCB), or software (via the *TraceControl* register) controls tracing functionality.

Trace is turned on when the following expression evaluates true:

```
(
    (
        (TraceControl_TS and TraceControl_On) or
        ((not TraceControl_TS) and TCBCONTROLA_On)
    )
    and
    (MatchEnable or TriggerEnable of FilterDataTrace)
)
```

where,

```
MatchEnable ←
(
    TraceControl_TS
    and
    TraceControl_G or
        (((TraceControl_ASID xor EntryHi_ASID) and (not TraceControl_ASID_M)) = 0) and
        ((TraceCOntrol3_GV) or ((TraceControl3_GuestID xor EffectiveGuestID = 0) and
        (TraceControl3_GV = 1)))
    )
    and
    (
        (TraceControl_U and UserMode)       or
        (TraceControl_S and SupervisorMode) or
        (TraceControl_K and KernelMode)     or
        (TraceControl_E and ExceptionMode) or
        (TraceControl_D and DebugMode)
    )
)
or
(
    (not TraceControl_TS)
    and
    (TCBCONTROLA_G or (TCBCONTROLA_ASID = EntryHi_ASID))
    and
    (TCBCONTROLE_GV or (TCBCONTROLE_GUESTID = EntryHi_ASID))
    and
    (
        (TCBCONTROLA_U and UserMode)        or
        (TCBCONTROLA_S and SupervisorMode)  or
        (TCBCONTROLA_K and KernelMode)      or
        (TCBCONTROLA_E and ExceptionMode)   or
```

```
        (TCBCONTROLA_DM and DebugMode)
    )
)
```

and where,

```
TriggerEnable ←
(
    DBCi_TE          and
    DBS_BS[i]        and
    TraceBPC_DE      and
    (TraceBPC_DBPOn[i] = 1)
)
or
(
    IBCi_TE          and
    IBS_BS[i]        and
    TraceBPC_IE      and
    (TraceBPC_IBPOn[i] = 1)
)
```

and where,

```
FilterDataTrace <- TraceControl3FDT and
(Load_Address_Matches_Hardware_Breakpoint_Address or
Store_AddressS_Matches_Hardware_Breakpoint_Address)
```

As seen in the expression above, trace can be turned on only if the master switch $TraceControl_{On}$ or $TCBCONTROLA_{On}$ is first asserted.

Once this is asserted, there are three ways to turn on tracing. The first way, the *MatchEnable* expression, uses the input enable signals from the TCB or the bits in the *TraceControl* register. This tracing is done over general program areas. For example, all of the user-level code for a particular process (if ASID is specified), and so on.

The second way to turn on tracing, the *TriggerEnable* expression, is from the processor side using the EJTAG hardware breakpoint triggers. If EJTAG is implemented, and hardware breakpoints can be set, then using this method enables finer grain tracing control. It is possible to send a trigger signal that turns on tracingat a particular instruction. For example, it would be possible to trace a single procedure in a program by triggering on trace at the first instruction, and triggering off trace at the last instruction.

The third way to enable tracing is in Filtered Data Trace Mode. When this mode is enabled, data load and store addresses are compared to the hardware data breakpoint address, if the addresses match, the data value associated with that match along with the address are traced out.

The easiest way to unconditionally turn on trace is to asserteither hardware or software tracing and the corresponding trace on signal with other enables. For example, with $TraceControl_{TS}$ = 0, i.e., hardware controlled tracing, assert $TCBCONTROLA_{On}$, $TCBCONTROLA_{G}$, and all the other signals in the second part of expression *MatchEnable*. To only trace when a particular process with a known ASID is executing, assert $TCBCONTROLA_{On}$, the correct $TCBCONTROLA_{ASID}$ value, and all of $TCBCONTROLA_{U}$, $TCBCONTROLA_{K}$, $TCBCONTROLA_{E}$, and $TCBCONTROLA_{DM}$. (If it is known that the particular process is a user-level process, then it would be sufficient to only assert $TCBCONTROLA_{U}$ for example). When using the EJTAG hardware triggers to turn trace on and off, it is best if $TCBCONTROLA_{On}$ is asserted and all the other processor mode selection bits in *TCBCONTROLA* are turned off. This would be the least confusing way to control tracing with the trigger signals. Tracing can be controlled via software with the *TraceControl* register in a similar manner.

13.11.8.3 Turning Off PDtrace™ Trace

Trace is turned off when the following expression evaluates true:

```
(
    (TraceControl_TS and (not TraceControl_On)) or
    ((not TraceControl_TS) and (not TCBCONTROLA_On))
)
or
(
    (not MatchEnable)       and
    (not TriggerEnable)     and
    (not FilterDataTraceActive)and
    TriggerDisable
)
```

where,

```
TriggerDisable ←
(
    DBCi_TE          and
    DBS_BS[i]        and
    TraceBPC_DE      and
    (TraceBPC_DBPOn[i]  = 0)
)
or
(
    IBCi_TE          and
    IBS_BS[i]        and
    TraceBPC_IE      and
    (TraceBPC_IBPOn[i]  = 0)
)
```

Tracing can be unconditionally turned off by de-asserting the *$TraceControl_{On}$* bit or the *$TCBCONTROLA_{On}$* signal. When either of these are asserted, tracing can be turned off if all of the enables are de-asserted, irrespective of the $TraceControl_G$ bit (*$TCBCONTROLA_G$*) and $TraceControl_{ASID}$ (*$TCBCONTROLA_{ASID}$*) values. EJTAG hardware breakpoints can be used to trigger trace off as well. Note that if simultaneous triggers are generated, and even one of them turns on tracing, then even if all of the others attempt to trigger trace off, then tracing will still be turned on. This condition is reflected in presence of the "(not TriggerEnable)" term in the expression above.

13.12 PDtrace Cycle-by-Cycle Behavior

A key reason for using trace, and not single stepping to debug a software problem, is often to get a picture of the real-time behavior. However the trace logic itself can, when enabled, affect the exact cycle-by-cycle behavior,

13.12.1 FIFO Logic in PDtrace and TCB Modules

Both the PDtrace module and the TCB module contain a fifo. This might seem like extra overhead, but there are good reasons for this. The vast majority of the information compression happens in the PDtrace module. Any data information, like PC and load/store address values (delta or full), load/store data and processor mode changes, are sent on two 32-bit data busses to the TCB on the internal PDtrace™ interface. When an instruction requires more than 2x32 bits of information to be traced properly, the PDtrace fifo will buffer the information, and send it on subsequent clock cycles.

In the TCB, the on-chip trace memory is defined as a 128-bit wide synchronous memory running at core-clock speed. In this case the FIFO is not needed. For off-chip trace through the Trace Probe, the FIFO comes into play, because only a limited number of pins (16) exist. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the P6600 core. So for off-chip tracing, a specific TCB TW FIFO is needed.

13.12.2 Handling of FIFO Overflow in the PDtrace Module

Depending on the amount of trace information selected for trace, and the frequency with which the 2x32-bit data interface is needed, it is possible for the PDtrace FIFO to overflow from time to time. There are two ways to handle this case:

1. Allow the overflow to happen, and thereby lose some information from the trace data.

2. Prevent the overflow by back-stalling the core until the FIFO has enough empty slots to accept new trace data.

The PDtrace fifo option is controlled by either the $TraceControl_{IO}$ or the $TCBCONTROLA_{IO}$ bit, depending on the setting of $TraceControl_{TS}$ bit.

The first option is free of any cycle-by-cycle change whether trace is turned on or not. This is achieved at the cost of potentially losing trace information. After an overflow, the fifo is completely emptied, and the next instruction is traced as if it was the start of the trace (processor mode and full PC are traced). This guarantees that only the un-traced fifo information is lost.

The second option guarantees that all the trace information is traced to the TCB. In some cases this is then achieved by back-stalling the core pipeline, giving the PDtrace fifo time to empty enough room in the fifo to accept new trace information from a new instruction. This option can obviously change the real-time behavior of the core when tracing is turned on.

If PC trace information is the only thing enabled (in $TraceControl2_{MODE}$ or $TCBCONTROLC_{MODE}$, depending on the setting of $TraceControl_{TS}$), and Trace of all branches is turned off (via $TraceControl_{TB}$ or $TCBCONTROLA_{TB}$, depending on the setting of $TraceControl_{TS}$), then the fifo is unlikely to overflow very often, if at all. This is of course very dependent on the code executed, and the frequency of exception handler jumps, but with this setting there is very little information overhead.

13.12.3 Handling of FIFO Overflow in the TCB

The TCB also holds a FIFO, used to buffer the TW's which are sent off-chip through the Trace Probe. The data width of the probe is 16 pins and the speed of these data pins can range from core-clock speed to 1/10th of the core clock speed (the trace probe clock always runs at a double data rate multiple to the core-clock). See Section 13.12.3.1 "Probe Width and Clock-ratio Settings" for a description of probe width and clock-ratio options. The combination between the probe width and the data speed allows for different data rates through the trace probe. The high extreme is not likely to be supported in any implementation, but the low one might be.

The data rate is an important figure when the likelihood of a TCB fifo overflow is considered. The TCB will at maximum produce two 64-bit trace words per core-clock cycle. This is true for any selection of trace mode in $TraceControl2_{MODE}$ or $TCBCONTROLC_{MODE}$. The PDtrace module will guarantee the limited amount of data. If the TCB data rate cannot be matched by the off-chip probe width and data speed, then the TCB fifo can possibly overflow. Similar to the PDtrace module FIFO, this can be handled in two ways:

1. Allow the overflow to happen, and thereby lose some information from the trace data.

2. Prevent the overflow by asserting a stall-signal back to the core (*PDI_StallSending*). This will in turn stall the core pipeline.

As a practical matter, the amount of data to the TCB can be minimized by only tracing PC information and excluding any cycle accurate information. This is explained in Section 13.12.2 "Handling of FIFO Overflow in the PDtrace Module" and below in Section 13.12.4 "Adding Cycle Accurate Information to the Trace". With this setting, a data rate of 8-bits per core-clock cycle is usually sufficient. No guarantees can be given here, however, as heavy interrupt activity can increase the number of unpredictable jumps considerably.

13.12.3.1 Probe Width and Clock-ratio Settings

Note: the registers called out in this section are located in the Coherence Manager TAP described in Chapter 15, Multi-CPU Debug. All of these fields are reserved in the P6600 core TAP registers.

The actual number of data pins (16) is defined by the TAP $TCBCONFIG_{PW}$ field. Furthermore, the frequency of the Trace Probe can be different from the core-clock frequency. The trace clock (*TR_CLK*) is a double data rate clock. This means that the data pins (*TR_DATA*) change their value on both edges of the trace clock. When the trace clock is running at clock ratio of 1:2 (one half) of core clock, the data output registers are running a core-clock frequency. The clock ratio is set in the TAP$TCBCONTROLB_{CR}$ field. The legal range for the clock ratio is defined in TAP $TCBCONFIG_{CRMax}$ and TAP $TCBCONFIG_{CRMin}$ (both values inclusive). If the TAP$TCBCONTROLB_{CR}$ bit is set to an unsupported value, the result is UNPREDICABLE.

The maximum possible value for TAP TCBCONFIGCRMax field is 1:2 (TR_CLK is running at one second of the core-clock). The minimum possible value for TAP TCBCONFIGCRMin field is 1:20 (TR_CLK is running at one twentieth of the core-clock).

13.12.4 Adding Cycle Accurate Information to the Trace

Depending on the trace regeneration software, it is possible to obtain the exact cycle time relationship between each instruction in the trace. This information is added to the trace, when the $TCBCONTROLB_{CA}$ bit is set. The overhead on the trace information is a little more than one extra bit per core-clock cycle.

This setting only affects the TCB module and not the PDtrace module. The extra bit therefore only affects the likelihood of the TCB FIFO overflowing.

13.13 PC Sampling

The PC sampling feature enables sampling of the PC value periodically. This information can be used for statistical profiling of the program akin to gprof. This information is also very useful for detecting hot-spots in the code.

In PC sampling, the PC is sampled periodically and sent to the TAP register. Note that although the PC sampling function can be used both with and without a probe, if a probe is not connected, the sampled information cannot be read out since the TAP registers can only be read when a probe is connected. Therefore, MIPS recommends using the PC sampling capability only when a probe is connected.

The presence or absence of the PC Sampling feature is available in the Debug Control register as bit 9 (PCS).The sampled PC values are written into a TAP register. The old value in the TAP register is overwritten by a new value even if this register has not be read out by the debug probe. The sample rate is specified in a manner similar to the PDtrace synchronization period, with three bits. These bits in the Debug Control register are 8:6 and called PCSR (PC Sample Rate). These three bits take the value 2^5 to 2^{12} similar to SyncPeriod. Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in

WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application.

The sampled values includes a new data bit, the PC, the ASID of the sampled PC as well as the Enhanced Virtual Address (EVA) K/U bit. Figure 13.6 shows the format of the sampled values in the TAP register PCsample. The new data bit is used by the probe to determine if the PCsample register data just read out is new or already been read and must be discarded.

Figure 13.6 TAP Register PCsample Format

49 ... 42	41	40 ... 33	32 ... 1	0
R	K/U	ASID	PC	New

T he sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

13.13.1 PC Sampling in Wait State

When the processor is in a WAIT state to save power for example, an external agent might want to know how long it stays in the WAIT state. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 every time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, then counting is resumed as before.

13.14 EJTAG Registers

The following subsections describe the EJTAG register interface.

13.14.1 General Purpose Control and Status

The following register provide general control and status information for EJTAG.

13.14.1.1 Debug Control Register

The Debug Control Register (*DCR*) register controls and provides information about debug issues and is always provided with the P6600 core. The register is memory-mapped in drseg at offset 0x0.

The DataBrk and InstBrk bits indicate if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the INTE bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the NMIE bit, and a pending NMI is indicated through the NMIP bit.

The SRE bit allows implementation dependent masking of some sources for reset. The P6600 core does not distinguish between soft and hard reset, but typically only soft reset sources in the system would be maskable and hard sources such as the reset switch would not be. The soft reset masking should only be applied to a soft reset source if that source can be efficiently masked in the system, thus resulting in no reset at all. If that is not possible, then that soft reset source should not be masked, since a partial soft reset may cause the system to fail or hang. There is no automatic indication of whether the SRE is effective, so the user must consult system documentation.

The PE bit reflects the ProbEn bit from the EJTAG Control register (*ECR*), whereby the probe can indicate to the debug software that the probe will service dmseg accesses. The reset value in the table below takes effect on any CPU reset.

Figure 13.7 Debug Control Register Format

63	32
0	

31	30	29	28	27	26	25	24	23	22	21	19	18	17	16
0		ENM	0		PCIM	PCno ASID	DASQ	DASe	DAS	0		FDC Impl	Data Brk	Inst Brk

15	14	13	12	11	10	9	8	6	5	4	3	2	1	0
IVM	DVM	0		RD Vec	CBT	PCS	PCR		PCSe	IntE	NMIE	NMI pend	SRstE	Prob En

Table 13.10 Debug Control Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
0	63:30	Must be written as zeros; return zeros on reads.	0	0
ENM	29	Endianess in which the processor is running in kernel and Debug Mode. This bit is encoded as follows: 0: Little Endian 1: Bit Endian	R	Preset
0	28:27	Must be written as zeros; return zeros on reads.	0	0
PCIM	26	Configure PC Sampling to capture all executed addresses or only those that miss the instruction cache This feature is not supported and this bit will read as 0. This bit is encoded as follows: 0: All PC's captured. 1: Capture only PC's that miss in the cache.	R	0
PCnoASID	25	Controls whether the PCSAMPLE scan chain includes or omits the ASID field ASID is always included so this bit will read as 0. This bit is encoded as follows: 0: ASID included in PCSAMPLE scan 1: ASID omitted from PCSAMPLE scan	R	0
DASQ	24	Qualifies Data Address Sampling using a data breakpoint. Data address sampling is not supported so this bit will read as 0. This bit is encoded as follows: 0: All data addresses are sampled 1: Sample matches of data breakpoint 0	R	0

Table 13.10 Debug Control Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
DASe	23	Enables Data Address Sampling Data address sampling is not supported so this bit will read as 0. This bit is encoded as follows: 0: Data Address sampling disabled. 1: Data Address sampling enabled.	R	0
DAS	22	Indicates if the Data Address Sampling feature is implemented. Data address sampling is not supported so this bit will read as 0. This bit is encoded as follows: 0: No DA Sampling implemented 1: DA Sampling implemented	R	0
0	21:19	Must be written as zeros; return zeros on reads.	0	0
FDCImpl	18	Indicates if the fast debug channel is implemented. This bit is encoded as follows: 0: No fast debug channel implemented 1: Fast debug channel implemented	R	1
DataBrk	17	Indicates if data hardware breakpoint is implemented. This bit is encoded as follows: 0: No data hardware breakpoint implemented 1: Data hardware breakpoint implemented	R	Preset
InstBrk	16	Indicates if instruction hardware breakpoint is implemented. This bit is encoded as follows: 0: No instruction hardware breakpoint implemented 1: Instruction hardware breakpoint implemented	R	Preset
IVM	15	Indicates if inverted data value match on data hardware breakpoints is implemented. This bit is encoded as follows: 0: No inverted data value match on data hardware breakpoints implemented 1: Inverted data value match on data hardware breakpoints implemented	R	0
DVM	14	Indicates if a data value store on a data value breakpoint match is implemented. This bit is encoded as follows: 0: No data value store on a data value breakpoint match implemented 1: Data value store on a data value breakpoint match implemented	R	0
0	13:12	Must be written as zeros; return zeros on reads.	0	0
RDVec	11	Enables relocation of the debug exception vector. The value in the DebugVectorAddr register is used for EJTAG exceptions when ProbTrap = 0,and RDVec = 1.	R/W	0

Table 13.10 Debug Control Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
CBT	10	Indicates if complex breakpoint block is implemented. This bit is encoded as follows: 0: No complex breakpoint block implemented 1: Complex breakpoint block implemented	R	0
PCS	9	Indicates if the PC Sampling feature is implemented. This bit is encoded as follows: 0: No PC Sampling implemented 1: PC Sampling implemented	R	1
PCR	8:6	PC Sampling rate. Values 0 to 7 map to values 2^5 to 2^{12} cycles, respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles respectively. The external probe or software is allowed to set this value to the desired sample rate.	R/W	7
PCSe	5	If the PC sampling feature is implemented, then indicates whether PC sampling is initiated or not. That is, a value of 0 indicates that PC sampling is not enabled and when the bit value is 1, then PC sampling is enabled and the counters are operational.	R/W	0
IntE	4	Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms. This bit is encoded as follows: 0: Interrupt disabled 1: Interrupt enabled depending on other enabling mechanisms	R/W	1
NMIE	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode. This bit is encoded as follows: 0: NMI disabled 1: NMI enabled	R/W	1
NMIpend	2	Indication for pending NMI. This bit is encoded as follows: 0: No NMI pending 1: NMI pending	R	0
SRstE	1	Controls soft reset enable. This bit is encoded as follows: 0: Soft reset masked for soft reset sources dependent on implementation 1: Soft reset is fully enabled.	R/W	1
ProbEn	0	Indicates value of the ProbEn value in the ECR register. This bit is encoded as follows: 0: No access should occur to the dmseg segment 1: Probe services accesses to the dmseg segment Bit is read-only (R) and reads as zero if not implemented.	R	Same value as ProbEn in ECR

13.14.1.2 DebugVectorAddr Register

This register allows an alternate debug exception vector address to be specified, which can enable placing a debug monitor program into RAM for much faster execution than the default ROM address. This register is memory mapped at an offset of 0x00020 within the DRSEG memory segment.

Figure 13.8 shows the register format and Table 13.11 describes the fields in this register.

Figure 13.8 DebugVectorAddr Register Format

31 … 7	6	5 … 0
DebugVectorAddr	WG	0

Table 13.11 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
DebugVectorAddr	31	Programmable Debug Exception Vector Address. Note that bits 31:30 have default values of 1 and 0 respectively and can only be written when the WG bit is set. If the WG bit is cleared, these bits are read-only and retain their previous values. These two bits can be written whenever the WG bit is set, regardless of the state of *Config5*$_K$.	R/W	1
	30		R/W	0
	29:7		R/W	0x7f8009 (corresponds to 0xbfc00480)
WG	6	Write gate. When the WG bit is set, the DebugVectorAddr field is expanded to include bits 31:30 to facilitate programmable memory segmentation controlled by the *SegCtl0* through *SegCtl2* registers. When the WG bit is cleared, bits 31:30 of this register are not write-able and remain unchanged from the last time that WG was cleared.	R/W	Externally Set
0	5:0	Ignored on write, returns zero on read.	R	0

13.14.2 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in drseg with addresses as shown in Table 13.12.

Table 13.12 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	*IBS*	Instruction Breakpoint Status
0x1100	*IBA0*	Instruction Breakpoint Address 0
0x1108	*IBM0*	Instruction Breakpoint Address Mask 0
0x1110	*IBASID0*	Instruction Breakpoint ASID 0
0x1118	*IBC0*	Instruction Breakpoint Control 0
0x1200	*IBA1*	Instruction Breakpoint Address 1
0x1208	*IBM1*	Instruction Breakpoint Address Mask 1

Table 13.12 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1210	*IBASID1*	Instruction Breakpoint ASID 1
0x1218	*IBC1*	Instruction Breakpoint Control 1
0x1300	*IBA2*	Instruction Breakpoint Address 2
0x1308	*IBM2*	Instruction Breakpoint Address Mask 2
0x1310	*IBASID2*	Instruction Breakpoint ASID 2
0x1318	*IBC2*	Instruction Breakpoint Control 2
0x1400	*IBA3*	Instruction Breakpoint Address 3
0x1408	*IBM3*	Instruction Breakpoint Address Mask 3
0x1410	*IBASID3*	Instruction Breakpoint ASID 3
0x1418	*IBC3*	Instruction Breakpoint Control 3

13.14.2.1 Instruction Breakpoint Status (IBS) Register

The Instruction Breakpoint Status (*IBS*) register holds implementation and status information about the instruction breakpoints. The ASID applies to all the instruction breakpoints.

Figure 13.9 IBS Register Format

63 31	30	29 28	27 24	23 4	3 0
Res	ASIDsup	Res	BCN	Res	BS

Table 13.13 IBS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Res	63:31	Must be written as zero; returns zero on read.	R	0
ASIDsup	30	Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms. This bit is encoded as follows: 0: ASID compare not supported 1: ASID compare supported (IBASIDn register implemented)	R	1
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of instruction breakpoints implemented.	R	2 or 4
Res	23:4	Must be written as zero; returns zero on read.	R	0
BS	3:0	Break status for breakpoint n is at BS[n], with n from 0 to 3. The bit is set to 1 when the corresponding breakpoint is enabled and the condition has matched. If only two instruction breakpoints are implemented, bits 2 and 3 must be written as zero and will return zero on read.	R/W	Undefined

13.14.2.2 Instruction Breakpoint Address n (IBAn) Register

The Instruction Breakpoint Address *n* (*IBAn*) register has the address used in the condition for instruction breakpoint *n*, where n = breakpoint 0 - 3.

Figure 13.10 IBAn Register Format

63 … 0
IBA

Table 13.14 IBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
IBA	63:0	Instruction breakpoint address for condition.	R/W	Undefined

13.14.2.3 Instruction Breakpoint Address Mask n (IBMn) Register

The Instruction Breakpoint Address Mask *n* (*IBMn*) register has the mask for the address compare used in the condition for instruction breakpoint *n*, where n = breakpoint 0 - 3.

Figure 13.11 IBMn Register Format

63 … 0
IBMn

Table 13.15 IBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
IBMn	63:0	Instruction breakpoint address mask for condition. This bit is encoded as follows: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

13.14.2.4 Instruction Breakpoint ASID n (IBASIDn) Register

This register is used to define an ASID value to be used in the match expression, where n = breakpoint 0 - 3.

Figure 13.12 IBASIDn Register Format

63 … 32	31 … 24	23	22 … 8	7 … 0
R	GUESTID	UGID	R	ASID

Table 13.16 IBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
R	63:32	Must be written as zero; returns zero on read.	R	0
GUESTID	31:24	Indicates the GuestID. GuestID value used for match comparison. If GuestCtl0.G1 = 1, then the active width of this register field matches the number of writable bits of GuestCtl1.ID. If GuestCtl0.G1 = 0, then only the right-most bit of this register field is writable and the rest of the bits in this field are read-only as zero. A value of zero is used to select Root-mode execution.	R/W	Undefined
UGID	23	Use GuestID field. If this bit is set, a match only happens when the GuestID field within this register matches the GuestID of the memory request and the device is executing in GuestMode (GuestCtl0.GM = 1 & Root.Status.EXL = 0 & Root.Status.ERL = 0 & Root.Debug.DM = 0). If this bit is clear, the GuestID field of this register is not used for match calculation. If this bit is set, the GuestID field is used for the match calculation. Probe Software can determine if this feature is software configurable by writing and reading back this bit.	R/W	Undefined
R	19:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Instruction breakpoint ASID value for compare.	R/W	Undefined

13.14.2.5 Instruction Breakpoint Control n (IBCn) Register

The Instruction Breakpoint Control *n* (*IBCn*) register controls the setup of instruction breakpoint *n*, where n = breakpoint 0 - 3.

Figure 13.13 IBCn Register Format

63 … 24	23	22 … 3	2	1	0
R	ASIDuse	R	TE	R	BE

Table 13.17 IBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	63:24	Must be written as zero; returns zero on read.	R	0
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: This bit is encoded as follows: 0: Don't use ASID value in compare. 1: User ASID value in compare.	R/W	Undefined

Table 13.17 IBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	22:3	Must be written as zero; returns zero on read.	R	0
TE	2	Trigger-only Enable. This field is ignored when BE is set. When BE is cleared and TE is set, instruction breakpoint n is enabled, but will not signal a debug exception.	R/W	0
R	1	Must be written as zero; returns zero on read.	R	0
BE	0	Breakpoint Enable. When set, instruction breakpoint n is enabled and will signal a debug exception when its condition matches.	R/W	0

13.14.3 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used the setup the data breakpoints. All registers are in drseg, and the addresses are shown in Table 13.18.

Table 13.18 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	*DBS*	Data Breakpoint Status
0x2100	*DBA0*	Data Breakpoint Address 0
0x2108	*DBM0*	Data Breakpoint Address Mask 0
0x2110	*DBASID0*	Data Breakpoint ASID 0
0x2118	*DBC0*	Data Breakpoint Control 0
0x2120	*DBV0*	Data Breakpoint Value 0
0x2138	*DBCS0*	Data Breakpoint Control SIMD 0
0x2140	*DBVS0*	Data Breakpoint Value SIMD 0
0x2200	*DBA1*	Data Breakpoint Address 1
0x2208	*DBM1*	Data Breakpoint Address Mask 1
0x2210	*DBASID1*	Data Breakpoint ASID 1
0x2218	*DBC1*	Data Breakpoint Control 1
0x2220	*DBV1*	Data Breakpoint Value 1
0x2238	*DBCS1*	Data Breakpoint Control SIMD 1
0x2240	*DBVS1*	Data Breakpoint Value SIMD 1

13.14.3.1 Data Breakpoint Status (DBS) Register

The Data Breakpoint Status (*DBS*) register holds implementation and status information about the data breakpoints.

The ASIDsup field indicates whether ASID compares are supported.

Figure 13.14 DBS Register Format

63	32
R	

31	30	29	28	27 24	23 2	1 0
R	ASID	NoSVmatch	NoLVmatch	BCN	R	BS

Table 13.19 DBS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
R	63:31	Must be written as zero; returns zero on read.	R	0
ASID	30	Indicates that ASID compares are supported in data breakpoints. This bit is encoded as follows: 0: Don't use ASID value in compare 1: Use ASID value in compare	R	1
NoSVmatch	29	Indicates if a value compare on a store is supported in data breakpoints. This bit is encoded as follows: 0: Data value and address in condition on store 1: Address compare only in condition on store	R	0
NoLVmatch	28	Indicates if a value compare on a load is supported in data breakpoints. This bit is encoded as follows: 0: Data value and address in condition on store 1: Address compare only in condition on store	R	0
BCN	27:24	Number of data breakpoints implemented.	R	2
R	23:2	Must be written as zero; returns zero on read.	R	0
BS	1:0	Break status for breakpoint n is at BS[n], with n from 0 to 1. The bit is set to 1 when the condition for the corresponding breakpoint has matched and the condition has matched. If only one data breakpoint is implemented, bit 1 must be written as 0 and will return 0 on reads.	R/W0	Undefined

13.14.3.2 Data Breakpoint Address n (DBAn) Register

The Data Breakpoint Address n (*DBAn*) register has the address used in the condition for data breakpoint n, where n = breakpoint 0 - 1.

Figure 13.15 DBAn Register Format

63	0
DBA	

Table 13.20 DBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DBA	63:0	Data breakpoint address for condition.	R/W	Undefined

13.14.3.3 Data Breakpoint Address Mask n (DBMn) Register

The Data Breakpoint Address Mask n (*DBMn*) register has the mask for the address compare used in the condition for data breakpoint n, where n = breakpoint 0 - 1.

Figure 13.16 DBMn Register Format

63 ... 0
DBM

Table 13.21 DBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DBM	63:0	Data breakpoint address mask for condition. This bit is encoded as follows: 0: Corresponding address bit is compared 1: Corresponding address bit is masked	R/W	Undefined

13.14.3.4 Data Breakpoint ASID n (DBASIDn) Register

This register is used to define an ASID value to be used in the match expression. For this register, n = breakpoint 0 - 1.

Figure 13.17 DBASIDn Register Format

63 ... 32	31 ... 24	23	22 ... 8	7 ... 0
R	GUESTID	UGID	R	ASID

Table 13.22 DBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
R	63:32	Must be written as zero; returns zero on read.	R	0

Table 13.22 DBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
GUESTID	31:24	Indicates the GuestID. GuestID value used for match comparison. If GuestCtl0.G1 = 1, then the active width of this register field matches the number of writable bits of GuestCtl1.ID. If GuestCtl0.G1 = 0, then only the right-most bit of this register field is writable and the rest of the bits in this field are read-only as zero. A value of zero is used to select Root-mode execution.	R/W	Undefined
UGID	23	Use GuestID field. If this bit is set, a match only happens when the GuestID field within this register matches the GuestID of the memory request and the device is executing in GuestMode (GuestCtl0.GM = 1 & Root.Status.EXL = 0 & Root.Status.ERL = 0 & Root.Debug.DM = 0). If this bit is clear, the GuestID field of this register is not used for match calculation. If this bit is set, the GuestID field is used for the match calculation. Probe Software can determine if this feature is software configurable by writing and reading back this bit.	R/W	Undefined
Res	19:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Data breakpoint ASID value for compare.	R/W	Undefined

13.14.3.5 Data Breakpoint Control n (DBCn) Register

The Data Breakpoint Control SIMD *n* (*DBCn*) register controls the setup of data breakpoint *n*, where n = breakpoint 0 - 1.

Figure 13.18 DBCn Register Format

63 … 36	35	34 … 32
R	GM	0

31 … 24	23	22	21 … 14	13	12	11 … 4	3	2	1	0
R	ASIDuse	R	BAI	NoSB	NoLB	BLM	R	TE	R	BE

Table 13.23 DBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	63:36	Must be written as zero; returns zero on read.	R	0

Table 13.23 DBCn Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
GM	35	Ganged Mode. An even and odd numerically adjacent breakpoint register set are ganged together to form a memory-aligned breakpoint on misaligned data. In contrast aligned 32-bit data only requires one 32-bit breakpoint register set. GM is only set by software in the even register control, and hardware will infer the odd register in pair is ganged with it. This feature allows some breakpoint register sets to be ganged, others not. This bit is encoded as follows: 0: Ganged Mode is not implemented. All breakpoint register sets are independent. 1: Ganged mode is implemented	R/W	0
R	34:32	Must be written as zero; returns zero on read.	R	0
R	31:24	Must be written as zero; returns zero on read.	R	0
ASIDuse	23	Use ASID value in compare for data breakpoint n. This bit is encoded as follows 0: Don't use ASID value in compare 1: Use ASID value in compare	R/W	Undefined
R	22	Must be written as zero; returns zero on read.	R	0
BAI	21:14	Byte access ignore controls ignore of access to a specific byte. *BAI*[0] ignores access to byte at bits [7:0] of the data bus, *BAI*[1] ignores access to byte at bits [15:8], etc. This bit is encoded as follows: 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
NoSB	13	Controls if condition for data breakpoint is fulfilled on a store transaction. This bit is encoded as follows: 0: Condition may be fulfilled on store transaction 1: Condition is never fulfilled on store transaction	R/W	Undefined
NoLB	12	Controls if condition for data breakpoint is fulfilled on a load transaction. This bit is encoded as follows: 0: Condition may be fulfilled on load transaction 1: Condition is never fulfilled on load transaction	R/W	Undefined
BLM	11:4	Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc. This bit is encoded as follows: 0: Compare corresponding byte lane 1: Mask corresponding byte lane	R/W	Undefined
R	3	Must be written as zero; returns zero on reads.	R	0

Table 13.23 DBCn Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
TE	2	Trigger-only Enable. This field is ignored when *BE* is set. When *BE* is cleared and TE is set, data breakpoint n is enabled, but will not signal a debug exception.	R/W	0
R	1	Must be written as zero; returns zero on reads.	R	0
BE	0	Breakpoint Enable. When set, data breakpoint n is enabled and will signal a debug exception when its condition matches.	R/W	0

13.14.3.6 Data Breakpoint Control SIMD n (DBCSn) Register

The Data Breakpoint Control SIMD *n* (*DBCSn*) register controls the setup of SIMD data breakpoint *n*, where n = breakpoint 0 - 1. In the P6600 core, this register is used to data that is larger than 64 bits.

Figure 13.19 DBCSn Register Format

63 — 40	39 — 32	31 — 8	7 — 0
R	BAI[15:8]	R	BLM[15:8]

Table 13.24 DBCSn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	63:40	Must be written as zero; returns zero on read.	R	0
BAI[15:8]	39:32	Byte access ignore. Each bit of this field determines whether a match occurs on an access to a specific byte of the database. (BAI[8] controls matching for data bus bits 71:64; BAI[9] controls matching for data bus bits 79:72, etc. with the polarity of each bit, as follows: 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
R	31:8	Must be written as zero; returns zero on read.	R	0
BLM[15:8]	7:0	Byte lane mask for value compare on data breakpoint. BAI[8] controls matching for data bus bits 71:64; BAI[9] controls matching for data bus bits 79:72, etc. Each bit of this field is encoded as follows: 0: Compare corresponding byte lane 1: Mask corresponding byte lane	R/W	Undefined

13.14.3.7 Data Breakpoint Value n (DBVn) Register

The Data Breakpoint Value n (*DBVn*) register has the value used in the condition for data breakpoint n, where n = breakpoint 0 - 1.

Figure 13.20 DBVn Register Format

63	0
DBV	

Table 13.25 DBVn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DBV	63:0	Data breakpoint value for condition.	R/W	Undefined

13.14.3.8 Data Breakpoint Value SIMD n (DBVSn) Register

The Data Breakpoint Value SIMD n (DBVSn) register has the value used in the condition for data breakpoint n. It is located at drseg segment offsets 0x2140 (DVBS0) and 0x2240 (DVBS1). It is only required for 128-bit MSA (MIPS SIMD Architecture) data breakpoints. For break-pointing on 128-bit data, both DBVn and DBVSn are required for a total of 16 mask bits for 128-bits.

Figure 13.21 DBVSn Register Format

63	0
DBVS	

Table 13.26 DBVSn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DBVS	63:0	Data breakpoint data value for condition. Debug software must adjust for endianess when programming this field.	R/W	Undefined

13.14.3.9 Misaligned Load/Store Breakpoint Support

In the P6600 core, the breakpoint facility must have the ability to support misalignment of SIMD load/stores. Though each register set n allows for support up to 128-bits, two such register sets must be ganged together to support break-pointing on misaligned 128-bit data, which can span two successive 128-bit data bus transactions.

For this purpose, the ganged-mode has been introduced (see the DBC.GM bit above). Numerically adjacent even and odd register sets are paired, where the even breakpoint set applies to the access at the lower address, while the odd breakpoint set maps to the access at the upper address.

To enable breakpoint on SIMD load/stores, DBCnGM bit must be set to 1. All registers in either set are ganged together as indicated. In ganged mode, each register set of the pair may be considered independent for byte, half-word, word and doubleword load/stores. As an example, if two register sets are configured to breakpoint on 128-bit data that is aligned on a 64-bit boundary, then two 64-bit loads may breakpoint on either half of the data enabled in DBC and DBCS of each set. Thus, a set of ganged register sets may support break-points on a single SIMD load/store, or multiple non-SIMD load/stores simultaneously.

13.14.4 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

13.14.4.1 Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation the TAP controller selects the output of the Instruction register to drive the *TDO* pin. The shift register consists of a series of bits arranged to form a single scan path between *TDI* and *TDO*. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from *TDI* to *TDO*. Both the capture and shift operations occur on the rising edge of *TCK*. However, the data shifted out from the *TDO* occurs on the falling edge of *TCK*. In the Test-Logic-Reset and *Capture-IR* state, the instruction shift register is set to 00001_2, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in Table 13.8.

13.14.4.2 Data Registers Overview

The EJTAG uses several data registers that are arranged in parallel from the primary *TDI* input to the primary *TDO* output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from *TDI* to *TDO*. During a data register scan operation, the TAP selects the output of the data register to drive the *TDO* pin. The register is updated in the *Update-DR* state with respect to the write bits.

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Address Register
- Data Register
- FastData Register

13.14.4.3 Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the Bypass register provides a single bit scan path between *TDI* and *TDO*. The Bypass register allows abbreviating the scan path through devices that are not involved in the test. The Bypass register is selected when the Instruction register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

13.14.4.4 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 13.27 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction. Note that this register contains only device manufacturer information and should not be used in an attempt to determine the EJTAG or PDTrace revisions of the device.

Figure 13.22 Device Identification Register Format

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 13.27 Device Identification Register

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Version	31:28	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	*EJ_PartNumber[15:0]*
ManufID	11:1	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	*EJ_ManufID[10:0]*
R	0	reserved	R	1

13.14.4.5 Implementation Register

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset values are set by inputs to the core. The register is selected when the Instruction register is loaded with the IMPCODE instruction. The EJTAG probe uses this TAP register to determine the EJTAG version of the device. Software has no access to this register and must use the CP0 Debug register to determine the EJTAG version.

Figure 13.23 Implementation Register Format

31 29	28 25	24	23 21	20 17	16	15	14	13 11	10 1	0
EJTAGver	reserved	DINTsup	ASIDsize	reserved	MIPS16	0	NoDMA	TYPE	TYPEINFO	32/64

Table 13.28 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	Indicates EJTAG version 6.0.	R	6
reserved	28:25	Reserved. Must be written as zeros; returns zeros on reads.	R	0

Table 13.28 Implementation Register Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported. This bit is encoded as follows: 0: DINT signal from the probe is not supported. 1: Probe can use DINT signal to make debug interrupt.	R	*EJ_DINTsup*
ASIDsize	23:21	Size of ASID field in implementation. This bit is encoded as follows: 0: No ASID 1: Reserved 2: 8-bit ASID 3: Reserved	R	2
R	20:17	Reserved	R	0
MIPS16	16	Indicates whether MIPS16 is implemented. This bit is encoded as follows: 0: No MIPS16 support 1: MIPS16 implemented	R	1
R	15	Reserved. Must be written as zeros; returns zeros on reads.	R	0
NoDMA	14		R	1
TYPE	13:11	Indicates what type of entity is associated with this TAP and whether the TypeInfo field exists. This field is encoded as follows: 000: TYPEINFO field not implemented. Legacy value. 001: This TAP is attached to a CPU and the TYPEINFO field reflects $EBase_{CPUNUM}$. 010: This TAP is attached to a Trace-Master and the TypeInfo field is not used. 011 - 111: Reserved	R	1
TYPEINFO	10:1	Identifier information specific to the type of entity associated with this TAP. The attached entity is specified by the TYPE field. This field is encoded as follows: CPU: Reflects $EBase_{CPUNUM}$ of the associated CPU. Others: Reserved.	R	1
32/64	0	32/64 bit processor. This bit is always '1' in the P6600 core. 0: 32-bit processor 1: 64-bit processor	R	1

13.14.4.6 EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in the CONTROL instruction. Bits in the EJTAG Control register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This EJTAG Control register can only be accessed by the TAP interface.

The EJTAG Control register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0. This is in order to ensure proper handling of processor accesses.

The value used for reset indicated in the table below takes effect on CPU resets, but not on TAP controller resets (e.g. *TRST_N*). *TCK* clock is not required when the CPU reset occurs, but the bits are still updated to the reset value when the *TCK* is supplied. The first 5 *TCK* clocks after CPU reset may result in reset of the bits, due to synchronization between clock domains.

Figure 13.24 EJTAG Control Register Format

31	30 29	28 23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	R	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	R	DM	R

Table 13.29 EJTAG Control Register Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
Rocc	31	Reset Occurred The bit indicates if a CPU reset has occurred: 0: No reset occurred since bit last cleared. 1: Reset occurred since but last cleared. The Rocc bit will remain set to 1 as long as reset is applied. This bit must be cleared by the probe to acknowledge that the incident was detected. The EJTAG Control register is not updated in the *Update-DR* state unless Rocc is 0 or written to 0, in order to ensure proper handling of processor access following reset.	R/W	1
Psz[1:0]	30:29	Processor Access Transfer Size These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending. This field is encoded as follows: 00: Byte 01: Halfword 10: Word 11: Doubleword	R	Undefined
R	28:23	Reserved. Write as zero. Returns zero on reads.	R	0

Table 13.29 EJTAG Control Register Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Doze	22	Doze state The Doze bit indicates any type of low-power mode. The value is sampled in the Capture-DR state of the TAP controller: 0: CPU not in low power mode. 1: CPU is in low power mode. Doze includes the Reduced Power (RP) and WAIT power-reduction modes.	R	0
Halt	21	Halt state The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller: 0: Internal system clock is running. 1: Internal system clock is stopped.	R	0
PerRst	20	Peripheral Reset When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain takes effect in the CPU clock domain and in peripherals. When the bit is written to 0, it must also be read as 0 before it is guaranteed that the indication is also cleared in the CPU clock domain. This bit controls the *EJ_PerRst* signal on the core.	R/W	0
PRnW	19	Processor Access Read and Write This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while *PrAcc* is set: 0: Read transaction. 1: Write transaction.	R	Undefined
PrAcc	18	Processor Access (PA) Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending: 0: No pending processor access. 1: Pending processor access. The probe's software must clear this bit to 0 to indicate the end of the processor access. A write of 1 is ignored. A pending Processor Access is cleared when *Rocc* is set, but another PA may occur just after the reset if a debug exception occurs. Finishing a Processor Access is not accepted while the *Rocc* bit is set. This is to avoid a Processor Access occurring after the reset is finished because of an indication of a Processor Access that occurred before the reset. The FASTDATA access can clear this bit.	R/W0	0
R	17	Reserved. Write as zero. Returns zero on reads.	R	0

Table 13.29 EJTAG Control Register Descriptions *(continued)*

Fields				
Name	**Bit(s)**	**Description**	**Read / Write**	**Reset State**
PrRst	16	Processor Reset. When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the *EJ_PrRst* signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the *EJTAG Control* register is reset by a reset.	R/W	0
ProbEn	15	Probe Enable This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered: 0: Probe does not handle EJTAG memory transactions. 1: Probe does handle EJTAG memory transactions. It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the *ProbEn* to 0. The operation of the processor is UNDEFINED in this case. The ProbEn bit is reflected as a read-only bit in the ProbEn bit, bit0, in the Debug Control Register (DCR). The read value indicates the effective value in the DCR, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbEn prior to setting the EjtagBrk bit will have effect for the debug handler executed due to the debug exception. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT

Table 13.29 EJTAG Control Register Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
ProbTrap	14	Probe Trap This bit controls the location of the debug exception vector. This bit is encoded as follows: 0: In normal memory. Vector is located as described in Section 13.14.1.2 "DebugVectorAddr Register" 1: In EJTAG memory at 0xFFFF.FFFF.FF20.0200 in dmseg Valid setting of the ProbTrap bit depends on the setting of the ProbEn bit, see comment under ProbEn bit. The ProbTrap should not be set to 1 unless the ProbEn bit is also set to 1 to indicate that the EJTAG memory may be accessed. The read value indicates the effective value to the CPU, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbTrap bit prior to setting the EjtagBrk bit will have effect for the EjtagBrk. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:	R/W	0 or 1 from EJTAGBOOT
R	13	Reserved. Write as zero. Returns zero on reads.	R	0
EjtagBrk	12	EJTAG Break Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low power mode. This bit is cleared by hardware when the debug exception is taken. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: 0: No EJTAGBOOT indication given. 1: EJTAGBOOT indication given.	R/W	0 or 1 from EJTAGBOOT
R	11:4	Reserved. Write as zero. Returns zero on reads.	R	0
DM	3	Debug Mode This bit indicates the debug or non-debug mode: 0: Processor is in non-debug mode. 1: Processor is in debug mode. The bit is sampled in the *Capture-DR* state of the TAP controller. This bit is the equivalent debug mode indicator from the TAP interface.	R	0
R	2:0	Reserved. Write as zero. Returns zero on reads.	R	0

13.14.5 Processor Access Registers

13.14.5.1 Processor Access Address Register

The Address register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

13.14.5.2 Processor Access Data Register

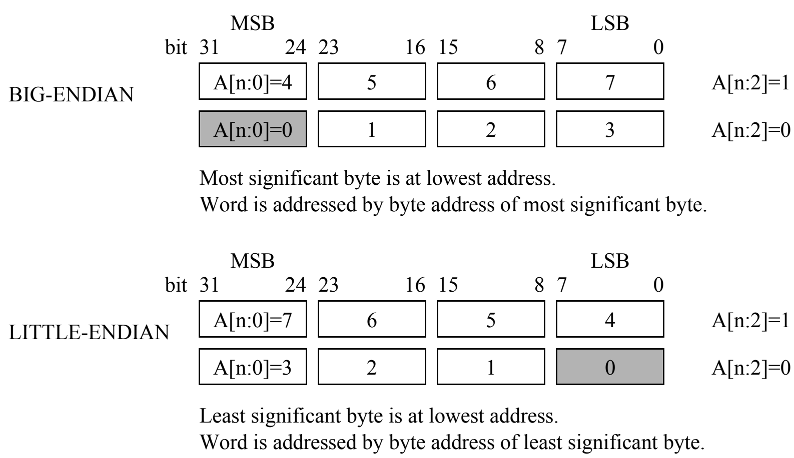
The Data register is used to provide data value to and from a processor access. The length of the Data register is 64 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value for a processor access read due to a CPU load or fetch from the dmseg. The register will be updated with a new value when a processor access write is pending.

The Data register is 64 bits wide. Data alignment is not used for this register, so the value in the Data register matches data on the internal bus. The unused bytes for a processor access write are undefined, and for a Data register read, 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the Data register depends on the endianess of the core, as shown in Figure 13.25. The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

Figure 13.25 Endian Formats for the Data Register



The size of the transaction and thus the number of bytes available/required for the Data register is determined by the Psz field in the *ECR*.

13.14.6 Fastdata Registers

13.14.6.1 Fastdata Register (TAP Instruction FASTDATA)

The width of the Fastdata register is 1 bit. During a Fastdata access, the Fastdata register is written and read, i.e., a bit is shifted in and a bit is shifted out. During a Fastdata access, the Fastdata register value shifted in specifies whether the Fastdata access should be completed or not. The value shifted out is a flag that indicates whether the Fastdata access was successful or not (if completion was requested).

Figure 13.26 Fastdata Register Format

31 … 1	0
R	SPrAcc

Table 13.30 Fastdata Register Field Description

Fields		Description	Read / Write	Power-up State
Name	**Bits**			
R	31:1	Reserved. Write as zero. Returns zero on reads.	R	0
SPrAcc	0	Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure. Shifting in a one does not complete the Fastdata access and the PrAcc bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete and a zero indicates the access would not have successfully completed.	R/W	Undefined

13.14.7 FDC TAP Register

The FDC TAP instruction performs a 38 bit bidirectional transfer of the FDC TAP register. The register format is shown in Figure 13.27 and the fields are described in Figure 13.31

Figure 13.27 FDC TAP Register Format

	37	36	35 … 32	31 … 0
In	Probe Data Accept	Data In Valid	ChannelID	Data
Out	Receive Buffer Full	Data Out Valid		

Table 13.31 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
Probe Data Accept	37	Indicates to core that the probe is accepting the data that was scanned out.	W	Undefined
Data In Valid	36	Indicates to core that the probe is sending new data to the receive FIFO.	W	Undefined
Receive Buffer Full	37	Indicates to probe that the receive buffer is full and the core will not accept the data being scanned in. Analagous to ProbeDataAccept, but opposite polarity	R	0

Table 13.31 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
Data Out Valid	36	Indicates to probe that the core is sending new data from the transmit FIFO	R	0
ChannelID	35:32	Channel number associated with the data being scanned in or out. This field can be used to indicate the type of data that is being sent and allow independent communication channels Scanning in a value with ChannelID=0xd and Data In Valid = 0 will generate a receive interrupt. This can be used when the probe has completed sending data to the core.	R/W	Undefined
Data	31:0	Data value being scanned in or out	R/W	Undefined

13.14.8 Fast Debug Channel Registers

This section describes the Fast Debug Channel registers. CPU access to FDC is via loads and stores to the FDC device in the Common Device Memory Map (CDMM) region. These registers provide access control, configuration and status information, as well as access to the transmit and receive FIFOs. The registers and their respective offsets are shown in Table 13.32

Table 13.32 FDC Register Mapping

Offset in CDMM device block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n ($0 \le n \le 15$)

13.14.8.1 FDC Access Control and Status (FDACSR) Register (Offset 0x0)

This is the general CDMM Access Control and Status register which defines the device type and size and controls user and supervisor access to the remaining FDC registers. The Access Control and Status register itself is only accessible in kernel mode. Figure 13.28 has the format of an Access Control and Status register (shown as a 64-bit register), and Table 13.33 describes the register fields.

Figure 13.28 FDC Access Control and Status Register

63 32	31 24	23 22	21 16	15 12	11 4	3	2	1	0
R	DevID	R	DevSize	DevRev	R	Uw	Ur	Sw	Sr

Table 13.33 FDC Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	63:32	Reserved. Write as zero. Returns zero on reads.	R	0
DevType	31:24	This field specifies the type of device.	R	DevType
R	23:22	Reserved. Write as zero. Returns zero on reads.	R	0
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. The value 0x2 indicates that this device uses 2 extra, or 3 total blocks.	R	0x2
DevRev	15:12	This field specifies the revision number of the device. The value 0x0 indicates that this is the initial version of FDC	R	0x0
R	11:4	Reserved. Write as zero. Returns zero on reads.	R	0
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled will return 0 and not change any state.	R/W	0
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in supervisor mode with access disabled is ignored.	R/W	0
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in supervisor mode with access disabled will return 0 and not change any state..	R/W	0

13.14.8.2 FDC Configuration (FDCFG) Register (Offset 0x8)

The FDC configuration register holds information about the current configuration of the Fast Debug Channel mechanism. Figure 13.29 has the format of the FDC Configuration register, and Table 13.34 describes the register fields.

Figure 13.29 FDC Configuration Register

31 … 20	19 … 18	17 … 16	15 … 8	7 … 0
R	Tx_IntThresh	Rx_IntThresh	TxFIFOSize	RxFIFOSize

Table 13.34 FDC Configuration Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
R	31:20	Reserved for future use. Read as zeros, must be written as zeros.	R	0

Table 13.34 FDC Configuration Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
TxIntThresh	19:18	Controls whether transmit interrupts are enabled and the state of the TxFIFO needed to generate an interrupt. This field is encoded as follows: 00: Transmit interrupt disabled 01: Empty 10: Not full 11: Almost empty. Either 0 or 1 entries in use. Refer to Section 13.15.2 for more information.	R/W	0
RxIntThresh	17:16	Controls whether receive interrupts are enabled and the state of the RxFIFO needed to generate an interrupt. This field is encoded as follows: 00: Receive interrupt disabled 01: Full 10: Not empty 11: Almost full. Either 0 or 1 entry free.	R/W	0
TxFIFOSize	15:8	This field holds the total number of entries in the transmit FIFO.	R	Preset
RxFIFOSize	7:0	This field holds the total number of entries in the receive FIFO.	R	Preset

13.14.8.3 FDC Status (FDSTAT) Register (Offset 0x10)

The FDC Status register holds up to date state information for the FDC mechanism. Figure 13.30 has the format of the FDC Status register, and Table 13.35 describes the register fields.

Figure 13.30 FDC Status Register

31 24	23 16	15 8	7 4	3	2	1	0
Tx_Count	Rx_Count	0	RxChan	RxE	RxF	TxE	TxF

Table 13.35 FDC Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Tx_Count	31:24	This optional field is not implemented and will read as 0	R	0
Rx_Count	23:16	This optional field is not implemented and will read as 0	R	0
0	15:8	Reserved. Must be written as zeros and read as zeros.	R	0
RxChan	7:4	This field indicates the channel number used by the top item in the receive FIFO. This field is only valid if RxE=0.	R	Undefined
RxE	3	If RxE is set, the receive FIFO is empty. If RxE is not set, the FIFO is not empty.	R	1
RxF	2	If RxF is set, the receive FIFO is full. If RxF is not set, the FIFO is not full.	R	0
TxE	1	If TxE is set, the transmit FIFO is empty. If TxE is not set, the FIFO is not empty.	R	1

Table 13.35 FDC Status Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
TxF	0	If TxF is set, the transmit FIFO is full. If TxF is not set, the FIFO is not full.	R	0

13.14.8.4 FDC Receive (FDRX) Register (Offset 0x18)

This register exposes the top entry in the receive FIFO. A read from this register returns the top item in the FIFO and removes it from the FIFO itself. The result of a write to this register is **UNDEFINED**. The result of a read when the FIFO is empty is also **UNDEFINED** so software must check the $FDSTAT_{RxE}$ flag prior to reading. Figure 13.31 has the format of the FDC Receive register, and Table 13.36 describes the register fields.

Figure 13.31 FDC Receive Register

31 ... 0
RxData

Table 13.36 FDC Receive Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
RxData	31:0	This register holds the top entry in the receive FIFO	R	Undefined

13.14.8.5 FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8*n)

These sixteen registers all access the bottom entry in the transmit FIFO. The different addresses are used to generate a 4b channel identifier that is attached to the data value. This allows software to track different event types without needing to reserve a portion of the 32b data as a tag. A write to one of these registers results in a write to the transmit FIFO of the data value and channel ID corresponding to the register being written. Reads from these registers are **UNDEFINED**. Attempting to write to the transmit FIFO if it is full has **UNDEFINED** results. Hence, the software running on the core must check the $FDSTAT_{TxF}$ flag to ensure that there is space for the write. Figure 13.32 has the format of the FDC Transmit register, and Table 13.37 describes the register fields.

Figure 13.32 FDC Transmit Register

31 ... 0
TxData

Table 13.37 FDC Transmit Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
TxData	31:0	This register holds the bottom entry in the transmit FIFO	W, Undefined value on read	Undefined

Table 13.38 FDTXn Address Decode

Addr	Chan		Addr	Chan		Addr	Chan		Addr	Chan
0x20	0x0		0x40	0x4		0x60	0x8		0x80	0xc
0x28	0x1		0x48	0x5		0x68	0x9		0x88	0xd
0x30	0x2		0x50	0x6		0x70	0xa		0x90	0xe
0x38	0x3		0x58	0x7		0x78	0xb		0x98	0xf

13.14.9 PDtrace™ Registers (Software Control)

The CP0 registers associated with PDtrace are listed in Table 13.39. Refer to Chapter 2, CP0 registers, for more information on these registers.

Table 13.39 A List of Coprocessor 0 Trace Registers

Register Number	Sel	Register Name
23	1	*TraceControl*
23	2	*TraceControl2*
24	2	*TraceControl3*
23	3	*UserTraceData1*
24	3	*UserTraceData2*

13.14.10 Trace Control Block (TCB) Registers (Hardware Control)

The TCB registers used to control its operation are listed in Table 13.40 and Table 13.41. These registers are accessed via the EJTAG TAP interface, or by software through mapping to drseg memory space.

Table 13.40 TCB EJTAG Registers

EJTAG Register	Name	Description	Implemented
0x10	TCBCONTROLA	Control register in the TCB mainly used for controlling the trace input signals to the core on the PDtrace interface. See Section 13.14.10.1 "TCBCONTROLA Register".	Yes
0x11	TCBCONTROLB	Control register in the TCB that is mainly used to specify what to do with the trace information. The *REG* [25:21] field in this register specifies the number of the TCB internal register accessed by the *TCBDATA* register. See Section 13.14.10.2 "TCBCONTROLB Register".	Yes
0x12	TCBDATA	This is used to access registers specified by the *REG* field in the *TCBCONTROLB* register. See Section 13.14.10.3 "TCBDATA Register".	Yes
0x14	TCBCONTROLC	Control Register in the TCB used to control and hold tracing information. See Section 13.14.10.4 "TCBCONTROLC Register".	Yes
0x13	TCBCONTROLD	Control Register in the TCB used to control and hold tracing information. See Section 13.14.10.5 "TCBCONTROLD Register".	Yes

Table 13.40 TCB EJTAG Registers *(continued)*

EJTAG Register	Name	Description	Implemented
0x16	TCBCONTROLE	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section 13.14.10.6 “TCBCONTROLE Register”.	Yes

13.14.10.1 TCBCONTROLA Register

The TCB is responsible for asserting or de-asserting the trace input control signals on the PDtrace interface to the core’s tracing logic. Most of the control is done using the *TCBCONTROLA* register.

The *TCBCONTROLA* register is written by an EJTAG TAP controller instruction, TCBCONTROLA (0x10). This register is also mapped to offset 0x3000 in drseg.

The format of the *TCBCONTROLA* register is shown below, and the fields are described in Table 13.41.

Figure 13.33 TCBCONTROLA Register Format

31 30	29 26	25 24	23	22 20	19	18	17	16	15	14	13	12 5	4	3	2	1	0
SyPExt	0	VModes	ADW	SyP	TB	IO	D	E	S	K	U	ASID	G	TFCR	TLSM	TIM	On

Table 13.41 TCBCONTROLA Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
SyPExt	31:30	These two bits used to be Implementation specific until PDtrace spec revision 06.00 when it reverts to architecturally defined bits to extend the SyP (sync period) field for implementations that need higher numbers of cycles between synchronization events. The value of SyP is extended by assuming that these two bits are juxtaposed to the left of the three bits of SyP (SyPExt.SyP). When only SyP was used to specify the synchronization period, the value was 2^x, where x was computed from SyP by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits just obtained by the juxtaposition of SyPExt and SyP. Sync period values greater than 2^{31} are UNPREDICTABLE. Since the value of 11010 represents the value of 31 (with +5), all values greater than 11010 are UNPREDICTABLE. Note that with these new bits, a sync period range of 2^5 to 2^{31} cycles can now be obtained.	R/W	0
0	29:26	Reserved. Must be written as zero; returns zero on read.	R	0
VModes	25:24	This field specifies the subset of tracing that is supported by the processor. This field is encoded as follows: 01: PC and load and store address tracing only All other values are invalid.	R	10

Table 13.41 TCBCONTROLA Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
ADW	23	*PDO_AD* bus width. 0: The width is 16 bits. 1: The width is 32 bits.	R	1
SyP	22:20	Used to indicate the synchronization period. The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown below. 000: 2^5 001: 2^6 010: 2^7 011: 2^8 100: 2^9 101: 2^{10} 110: 2^{11} 111: 2^{12} This field defines the value on the *PDI_SyncPeriod* signal.	R/W	000
TB	19	Trace All Branches. When set to one, this field indicates that the core must trace either full or incremental PC values for all branches. When set to zero, only the unpredictable branches are traced.	R/W	Undefined
IO	18	Inhibit Overflow. This bit is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full so that no trace records are ever lost.	R/W	Undefined
D	17	When set to one, this enables tracing in Debug mode, i.e., when the *DM* bit is one in the *Debug* register. For trace to be enabled in Debug mode, the *On* bit must be one, and either the *G* bit must be one, or the current process must match the *ASID* field in this register. When set to zero, trace is disabled in Debug mode, irrespective of other bits.	R/W	Undefined
E	16	This controls when tracing is enabled. When set, tracing is enabled when either of the *EXL* or *ERL* bits in the *Status* register is one, provided that the *On* bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the *ASID* field in this register. Note that if TraceControl3GV is set, the GuestID of instruction execution must match the TraceControl3GuestID register field for tracing to be enabled.	R/W	Undefined
S	15	When set, this enables tracing when the core is in Supervisor mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the *ASID* field in this register. Note that if TraceControl3.GV is set, the GuestID of instruction execution must match the TraceControl3.GuestID register field for tracing to be enabled.	R/W	Undefined

Table 13.41 TCBCONTROLA Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
K	14	When set, this enables tracing when the *On* bit is set and the core is in Kernel mode. Unlike the usual definition of Kernel Mode, this bit enables tracing only when the *ERL* and *EXL* bits in the *Status* register are zero. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the *ASID* field in this register. Noe that if TraceControl3.GV is set, the GuestID of instruction execution must match the TraceControl3.GuestID register field for tracing to be enabled.	R/W	Undefined
U	13	When set, this enables tracing when the core is in User mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the *On* bit (bit 0) is also set, and either the *G* bit is set, or the current process ASID matches the *ASID* field in this register. Noe that if TraceControl3.GV is set, the GuestID of instruction execution must match the TraceControl3.GuestID register field for tracing to be enabled.	R/W	Undefined
ASID	12:5	The ASID field to match when the *G* bit is zero. When the *G* bit is one, this field is ignored.	R/W	Undefined
G	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true.	R/W	Undefined
TFCR	3	When set, this indicates to the PDtrace interface that complete information about instruction if it can be a function call or return should be traced. It also indicates to the TCB that the optional Fcr bit must be traced in the appropriate trace formats	R/W	Undefined
TLSM	2	When set, this indicates to the PDtrace interface that complete information about Load and Store data cache miss should be traced. It also indicates to the TCB that the optional LSm bit must be traced in the appropriate trace formats.	R/W	Undefined
TIM	1	When set, this indicates to the PDtrace interface that complete information about instruction cache miss should be traced. It also indicates to the TCB that the optional Im bit must be traced in the appropriate trace formats.	R/W	Undefined
On	0	This is the global trace enable switch to the core. When zero, tracing from the core is always disabled, unless enabled by core internal software override. When set to one, tracing is enabled whenever the other enabling functions are also true.	R/W	0

13.14.10.2 TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (0x11). This register generally controls what to do with the trace information received. This register is also mapped to offset 0x3008 in drseg.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 13.42.

Figure 13.34 TCBCONTROLB Register Format

31	30 28	27 26	25 21	20	19	18	17	16 12	11	10 7	6 3	2	1	0
WE	0	TWSrcWidth	REG	WR	0	TRPAD	FDT	0	TLSIF	0	0	CA	0	EN

Table 13.42 TCBCONTROLB Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
WE	31	Write Enable. Only when set to 1 will the other bits be written in *TCBCONTROLB*. This bit will always read 0.	R	0
0	30:28	Reserved. Must be written as zero; returns zero on read.	R	0
TWSrc-Width	27:26	Used to indicate the number of bits used in the source field of the Trace Word, this is a configuration option of the core that cannot be modified by software. 00 - zero source field width 01 - two bit source field width 10 - four bit source field width 11 - reserved for future use This field can only be 10 for the P6600 core.	R	10
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 13.41.	R/W	0
WR	20	Write Registers: When set, the register selected by REG field is read and written when *TCBDATA* is accessed. Otherwise the selected register is only read.	R/W	0
0	19	Reserved. Must be written as zero; returns zero on read.	R	0
TRPAD	18	Trace RAM access disable bit, disables program software access to the on-chip trace RAM using load/store instructions. If probe access is not provided in the implementation, then this register bit must be tied to zero value to allow software to control access.	R/W	0
FDT	17	Filtered Data Trace Mode enable. When the bit is 0, this mode is disabled. When set to 1, this mode is enabled.	R/W	0
0	16:12	Reserved. Must be written as zero; returns zero on read.	R	0
TLSIF	11	When set, this indicates to the TCB that information about Load and Store data cache miss, instruction cache miss, and function call are to be taken from the PDtrace interface and trace them out in the appropriate trace formats as the three optional bits LSm, Im, and Fcr.	R/W	0
0	10:7	Reserved. Must be written as zero; returns zero on read.	R	0
TWSrcVal	6:3	These bits are used to indicate the value of the TW source field that will be traced if TWSrcWidth indicates a source bit field width of 2 or 4 bits. Note that if the field is 2 bits, then only bits 4:3 of this field will be used in the TW.	R	Preset
CA	2	Cycle accurate trace. When set to 1, the trace will include stall information. When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF's. The stall information included/excluded is: • TF6 formats with TCBcode 0001 and 0101. • All TF1 formats.	R/W	0

Table 13.42 TCBCONTROLB Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
OfC	1	This bit is always set to 1, indicating that the trace is sent to off-core coherency manager funnel.	R	1
EN	0	Enable trace. This is the master enable for trace to be generated from the TCB. This bit can be set or cleared, either by writing this register or from a start/stop trigger. When set to 1, Trace Words are generated and sent to the trace funnel. When set to 0, trace information is ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipe-line is flushed, and trace output is stopped.	R/W	0

13.14.10.3 TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the $TCBCONTROLB_{REG}$ field; see Table 13.40. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the $TCBCONTROLB_{WR}$ bit is set. For read-only registers, $TCBCONTROLB_{WR}$ is a don't care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 13.43. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 13.35 TCBDATA Register Format

31(63) … 0
Data

Table 13.43 TCBDATA Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	**Bits**			
Data	31:0 63:0	Register fields or data as defined by the $TCBCONTROLB_{REG}$ field	Only writable if $TCBCONTROLB_{WR}$ is set	0

13.14.10.4 TCBCONTROLC Register

The trace output from the processor on the PDtrace interface can be controlled by the trace input signals to the processor from the TCB. The TCB uses a control register, *TCBCONTROLC*, whose values are used to change the signal values on the PDtrace input interface. External software (i.e., debugger) can therefore manipulate the trace output by writing to this register.

The *TCBCONTROLC* register is written by the EJTAG TAP controller instruction, TCBCONTROLC (0x13). This register is also mapped to offset 0x3010 in drseg.

The format of the *TCBCONTROLC* register is shown below, and the fields are described in Table 13.44.

Figure 13.36 TCBCONTROLC Register Format

31 30	29 28	27 23	22 0
Res	NumDO	Mode	R

Table 13.44 TCBCONTROLC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Res	31:30	Reserved for future use. Must be written as zero; returns zero on read.	0	0
NumDO	29:28	Specifies the number of bits needed by this implementation to specify the DataOrder: 10 - Six bits	R	10
R	27:26	Reserved for future use. Must be written as zero; returns zero on read.	R/W	0
Mode	25:23	When tracing is turned on, this signal specifies what information is to be traced by the core. It uses 5 bits, where each bit turns on a tracing of a specific tracing mode. This field is encoded as follows: Bit 23: If set, trace the program counter (PC) Bit 24: If set, trace the load address. Bit 25: If set, trace the store address. If the corresponding bit is 0, then the Trace Value shown above is not traced by the processor.	R/W	0
R	22:0	Reserved for future use. Must be written as zero; returns zero on read.	R/W	0

13.14.10.5 TCBCONTROLD Register

The TCB includes a control register, TCBCONTROLD, whose values are used to enable tracing of the Coherence Manager. External software (i.e., debugger) can therefore manipulate the trace output by writing to this register. Each of the cores in the system has this register, and the *Core_CM_En* field is considered from each of the cores.

The *TCBCONTROLD* register is written by an EJTAG TAP controller instruction, *TCBCONTROLD* (0x14). This register is also mapped to offset 0x3018 in drseg. The format of the *TCBCONTROLD* register is shown below, and the fields are described in Table 13.45.

Figure 13.37 TCBCONTROLD Register Format

31 … 2	1	0
0	Core_CM_En	0

Table 13.45 TCBCONTROLD Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:2	Reserved. Must be written as zero; returns zero on read.	R	0
Core_CM_En	1	Core_CM_Enable: The CM looks at this bit coming from each of the cores. Allows cores other than the master to enable tracing if other conditions are met.	R/W	0
0	0	Reserved. Must be written as zero; returns zero on read.	R	0

13.14.10.6 TCBCONTROLE Register

The trace output from the processor on the PDtrace interface can be controlled by the trace input signals to the processor from the TCB. The TCB uses a control register, *TCBCONTROLE*, whose values are used to change the signal values on the PDtrace input interface. External software (i.e., debugger), can therefore manipulate the trace output by writing the *TCBCONTROLE* register.

The *TCBCONTROLE* register is written by an EJTAG TAP controller instruction, *TCBCONTROLE* (0x16).This register is also mapped to offset 0x3020 in drseg.

The format of the *TCBCONTROLE* register is shown below, and the fields are described in Table 13.46.

Figure 13.38 TCBCONTROLE Register Format

31 … 26	25	24	23	22	21 … 14	13	12	11	10 … 9	8	7 … 6	5	4	3	2	1	0
0	UPR	0	MSA	GV	GuestID	0	ADWB	ADWU	TdIDLE	0	PecOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC	

Table 13.46 TCBCONTROLE Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:26	Reserved for future use. Must be written as zero; returns zero on read.	0	0
UPR	25	Indicates that for 128-bit load/stores, only the lower 64 bits are traced and the lack of upper 64 bits is indicated by an additional bit in TF4. Example situations are MSA if tracing of 128-bit MSA load/store is not implemented (see bit TCBCONTROLE.MSA) and bonded 2x64-bit instructions.	R	1
0	24	Reserved for future use. Must be written as zero; returns zero on read.	0	0

Table 13.46 TCBCONTROLE Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
MSA	23	MSA Load/Store Data Trace. This bit is encoded as follows. 0 - MSA load/store data trace not implemented 1 - MSA load/store data trace implemented	R	1
GV	22	Enable trace for all GuestIDs or only 1 GuestID. 0 - trace enabled for all Guests 1 - trace enabled only for Guest specified by TCBControlE.GuestID	R	0
GuestID	21:14	The GuestID field to match when tracing. If GuestCtl0.G1 = 1, then the active width of the register field matches the number of writeable bits of GuestCtl1.ID and the rest of the bits of th is field are read-only as zero. If GuestCtl0.G1 = 0, then only the right-most bit of this register field is writeable and the rest of the bits of this field are read-only as zero. A value of zero is used to select Root-mode execution.	R/W	Undefined
0	13	Reserved for future use. Must be written as zero; returns zero on read.	R	0
ADWB	12:11	Number of bits used to encode ADW field in TF3/TF4: 0 - no ADW field in TF3/TF4 1 - 1 bit TF3/TF4.ADW field is present. 2 - 2 bit TF3/TF4.ADW field is present. 3 - 3 bit TF3/TF4.ADW field is present.	R	Preset
ADWU	10:9	Units of ADW width specifier entry in TF3/TF4. When ADWB is zero, there is no ADW field in TF3/ TF4, and ADWU gives the size of the AD field. When the ADWB and ADWU fields are both zero, TCBCONTROLA.ADW gives the size of the AD field.	R	Preset
TrIdle	8	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice-versa. The bit is read-only and updated by the trace hardware.	R	1
0	7:6	Reserved for future use; Must be written as zero; returns zero on read. (Hint to architect, Reserved for future expansion of performance counter trace events).	R	0
PeCOvf	5	Trace performance counters when one of the performance counters overflows its count value. Enabled when set to 1.	R/W	0
PeCFCR	4	Trace performance counters on function call/return or on an exception handler entry. Enabled when set to 1.	R/W	0
PeCBP	3	Trace performance counters on hardware breakpoint match trigger. Enabled when set to 1.	R/W	0

Table 13.46 TCBCONTROLE Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
PeCSync	2	Trace performance counters on synchronization counter expiration. Enabled when set to 1.	R/W	0
PeCE	1	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1.	Config Option	0
PeC	0	Specifies whether or not Performance Control Tracing is implemented. This bit is always set to 1 in the P6600 core.	R	1

The following registers are accessed by the $\text{TCBCONTROLB}_{\text{REG}}$ field.

13.14.10.7 TCBCONFIG Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. The format of the *TCBCONFIG* register is shown below, and the fields are described in Table 13.47. This register is also accessible at offset 0x3028 in DRSEG.

Figure 13.39 TCBCONFIG Register Format

31	30 25	24 21	20 17	16 14	13 11	10 9	8 6	5	4	3 0
CF1	0	TRIG	SZ	CRMax	CRMin	PW	PiN	0	OfT	REV

Table 13.47 TCBCONFIG Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
CF1	31	This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* does not exist and this bit always reads zero.	R	0
0	30:25	Reserved. Must be written as zero; returns zero on read.	R	0
TRIG	24:21	Number of triggers implemented. This also indicates the number of *TCBTRIGx* registers that exist.	R	Preset Legal values are 0 - 8
SZ	20:17	On-chip trace memory size. In the P6600 core, this field is not used since the trace memory is stored inside the Coherency Manager.	R	Undefined
CRMax	16:14	Off-chip Maximum Clock Ratio. This field indicates the maximum ratio of the CPU clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 13.48. This bit is reserved if off-chip trace option is not implemented.	R	Preset
CRMin	13:11	Off-chip Minimum Clock Ratio. This field indicates the minimum ratio of the CPU clock to the off-chip trace memory interface clock.The clock-ratio encoding is shown in Table 13.48. This bit is reserved if off-chip trace option is not implemented.	R	Preset

Table 13.47 TCBCONFIG Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
PW	10:9	ProbeWidth: Number of bits available on the off-chip trace interface data pins. The number of data pins is encoded as shown in the table. This field is encoded as follows: 00 - 01: Reserved 10: 16 bits 11: Reserved	R	10
PiN	8:6	Pipe number. Indicates the number of execution pipelines.	R	0
0	5	Reserved. Must be written as zero; returns zero on read.	R	0
OfT	4	When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module.	R	Preset
REV	3:0	Revision of TCB.	R	0x9

Table 13.48 Clock Ratio Encoding of the CR Field

Encoding of CR Field	Trace Clock:Core Clock Ratio
3’b000	1:20
3’b001	1:16
3’b010	1:12
3’b011	1:10
3’b100	1:2
3’b101	1:4
3’b110	1:6
3’b111	1:8

13.14.10.8 TCBTRIGx Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIGx*, where *x* is a single digit number from 0 to 7 (*TCBTRIG0* is Reg 16). The actual number of trigger registers implemented is defined in the $TCBCONFIG_{TRIG}$ field. An unimplemented register will read all zeros and writes are ignored.

Each Trigger Control register controls when an associated trigger is fired, and the action to be taken when the trigger occurs. Please also read Section 13.16 “TCB Trigger Logic”, for detailed description of trigger logic issues.

The format of the *TCBTRIGx* register is shown below, and the fields are described in Table 13.49. These registers are also accessible at offset 0x3200-0x3238 in DRSEG.

Figure 13.40 TCBTRIGx Register Format

31 — 24	23	22 — 16	15	14	13 — 7	6	5	4	3 — 2	1	0
TCBinfo	Trace	0	CHTro	PDTro	0	DM	CHTri	PDTri	Type	FO	TR

Table 13.49 TCBTRIGx Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
TCBinfo	31:24	This field is to be used in a possible TF6 trace format when this trigger fires.	R/W	0
Trace	23	When set, generate TF6 trace information when this trigger fires. Use *TCBinfo* field for the TCBinfo of TF6 and use *Type* field for the two MSB of the TCBtype of TF6. The two LSB of *TCBtype* are 00. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if the TF6 format was ever suppressed by a simultaneous trigger. If so, the read value will be 0. If the write value was 0, the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
0	22:16	Reserved. Must be written as zero; returns zero on read.	R	0
CHTro	15	When set, generate a single cycle strobe on *TC_ChipTrigOut* when this trigger fires.	R/W	0
PDTro	14	When set, generate a single cycle strobe on *TC_ProbeTrigOut* when this trigger fires.	R/W	0
0	13:7	Reserved. Must be written as zero; returns zero on read.	R	0
DM	6	When set, this Trigger will fire when a rising edge on the Debug mode indication from the core is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
CHTri	5	When set, this Trigger will fire when a rising edge on *TC_ChipTrigIn* is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0

Table 13.49 TCBTRIGx Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Names	Bits			
PDTri	4	When set, this Trigger will fire when a rising edge on *TC_ProbeTrigIn* is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
Type	3:2	Trigger Type: The Type indicates the action to take when this trigger fires. The encoding below shows the Type values and the Trigger action. 00: Trigger start; trigger start-point of trace. 01: Trigger end; trigger end-point of trace. 10: No effect 11: Trigger info; no action trigger, only for trace information. The actual action is to set or clear the $TCBCONTROLB_{EN}$ bit. A Start trigger will set $TCBCONTROLB_{EN}$, a End trigger will clear $TCBCONTROLB_{EN}$. If Trace is set, then a TF6 format is added to the trace words. For Start and Info triggers this is done before any other TF's in that same cycle. For End triggers, the TF6 format is added after any other TF's in that same cycle. If the $TCBCONTROLB_{TM}$ field is implemented it must be set to Trace-To mode (00), for the *Type* field to control on-chip trace fill. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if the trigger action was ever suppressed. If so the read value will be 11. If the write value was 11 the read value is always 11. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
FO	1	Fire Once. When set, this trigger will not re-fire until the *TR* bit is de-asserted. When de-asserted this trigger will fire each time one of the trigger sources indicates trigger.	R/W	0
TR	0	Trigger happened. When set, this trigger fired since the *TR* bit was last written 0. This bit is used to inspect whether the trigger fired since this bit was last written zero. When set, all the trigger source bits (bit 4 to 13) will change their read value to indicate if the particular bit was the source to fire this trigger. Only enabled trigger sources can set the read value, but more than one is possible. Also when set the *Type* field and the *Trace* field will have read values which indicate if the trigger action was ever suppressed by a higher priority trigger.	R/W0	0

13.14.11 Register Reset State

Reset state for all register fields is entered when either of the following occur:

1. TAP controller enters/is in Test-Logic-Reset state.
2. *EJ_TRST_N* input is asserted low.

13.15 Fast Debug Channel

The Fast Debug Channel (FDC) mechanism provides an efficient means to transfer data between the core and an external device using the EJTAG TAP pins. The FDC was created to allow for faster communication between the core and the probe. In previous generation MIPS processors, whenever the core wanted to communicate with the probe, the core would be halted and data send to the probe because the probe had no way to read the core. The FDC provides a mechanism using FIFO’s, whereby the probe can read the core without requiring that the core be halted. These FIFO’s provide a cross boundary between the core and the EJTAG regions of the P6600 core.

In the FDC, when the probe wishes to read and FDC register, the core gets an interrupt from the probe requesting this information. The core then places the requested information into the FIFO and continues operation. The core places information in the top of the FIFO, and the probe reads information from the bottom of the FIFO. The data contains information such as transmit versus receive, status of the operation, etc.

The external device would typically be an EJTAG probe and that is the term used here, but it could be something else. FDC utilizes two First In First Out (FIFO) structures to buffer data between thecore and probe. The probe uses the FDC TAP instruction to access these FIFOs, while the core itself accesses them using memory accesses. To transfer data out of the core, the core writes one or more pieces of data to the transmit FIFO. At this time, the core can resume doing other work. An external probe would examine the status of the transmit FIFO periodically. If there is data to be read, the probe starts to receive data from the FIFO, one entry at a time. When all data from the FIFO has been drained, the probe goes back to waiting for more data. The core can either choose to be informed of the empty transmit FIFO via an interrupt, or it can choose to periodically check the status. Receiving data works in a similar manner - the probe writes to the receive FIFO. At that time, the core is either interrupted, or finds out via polling a status bit. The core can then do load accesses to the receive FIFO and receive data being sent to it by the probe. The TAP transfer is bidirectional - a single shift can be pulling transmit data and putting receive data at the same time.

The primary advantage of FDC over normal processor accesses or fastdata accesses is that it does not require the core to be blocked when the probe is reading or writing to the data transfer FIFOs. This significantly reduces the core overhead and makes the data transfer far less intrusive to the code executing on the core.

The FDC memory mapped registers are located in the common device memory map (CDMM) region. FDC has a device ID of 0xFD.

13.15.1 Common Device Memory Map

Software on the core accesses FDC through memory mapped registers. These memory mapped registers are located within the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The base address and enabling of this region is controlled by the CDMMBase CP0 register.

13.15.2 Fast Debug Channel Interrupt

The FDC block can generate an interrupt to inform software of incoming data being available or space being available in the outgoing FIFO. This interrupt is handled similarly to the timer or performance counter interrupts. The $Cause_{FDCI}$ bit indicates that the interrupt is pending. The interrupt is also sent to the core outputs *SI_FDCI[_1]* where it is combined with one of the *SI_Int* pins. For non-EIC mode, the *SI_IPFDCI* input indicates which interrupt pin is has been combined with and this information is reflected in the $IntCtl_{IPFDCI}$ field. Note that this interrupt is a regular interrupt and not a debug interrupt.

The FDC Configuration Register (see Section 13.14.8.2 "FDC Configuration (FDCFG) Register (Offset 0x8)") includes fields for enabling and setting the threshold for generating each interrupt. Receive and transmit interrupt thresholds are specified independently, but transmit/receive interrrupts are ORed together to form a single interrupt per core.

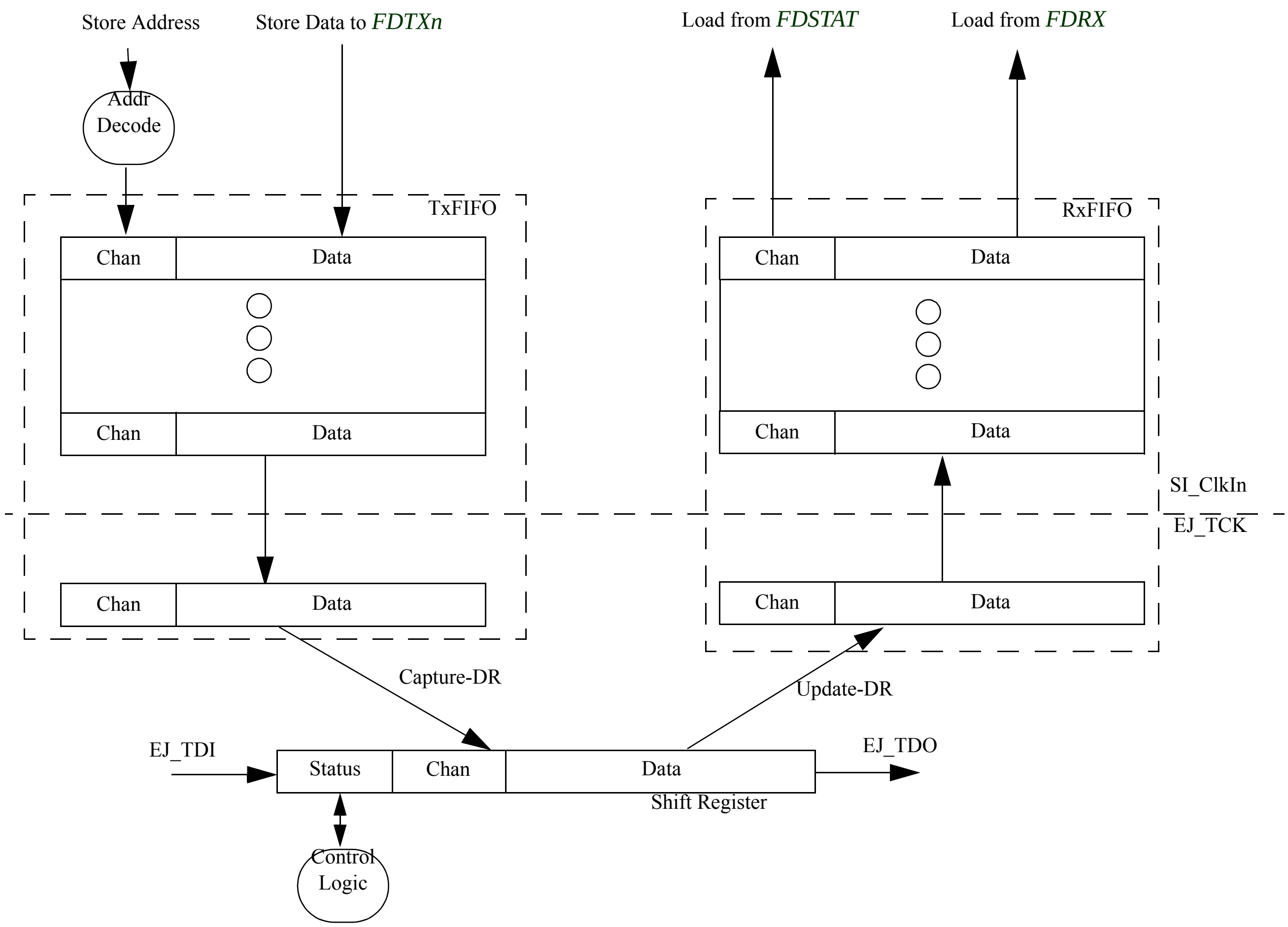
The following interrupt thresholds are supported:

- Interrupts Disabled: No interrupt will be generated and software must poll the status registers to determine if incoming data is available or if there is space for outgoing data.
- Minimum core Overhead: This setting minimizes the core overhead by not generating an interrupt until the receive FIFO (RxFIFO) is completely full or the transmit FIFO (TxFIFO) is completely empty.
- Minimum latency: To have the core take data as soon as it is available, the receive interrupt can be fired whenever the RxFIFO is not empty. There is a complimentary TxFIFO not full setting although that may not be quite as useful.
- Maximum bandwidth: When configured for minimum core overhead, bandwidth between the probe and core can be wasted if the core does not service the interrupt before the next transfer occurs. To reduce the chances of this happening, the interrupt threshold can be set to almost full or almost empty to generate an interrupt earlier. This setting causes receive interrupts to be generated when there are 0 or 1 unused RxFIFO entries. Transmit interrupts are generated when there are 0 or 1 used TxFIFO entries.

13.15.3 Core FDC Buffers

Figure 13.41 shows the general organization of the transmit and receive buffers on the P6600 core.

Figure 13.41 Fast Debug Channel Buffer Organization



One particular thing to note is the asynchronous crossings between the EJ_TCK and SI_ClkIn clock domains. This crossing is handled with a handshaked interface that safely transfers data between the domains. Two data registers are included in this interface, one in the source domain and one in the destination domain. The control logic actively manages these registers so that they can be used as FIFO entries. The fact that one FIFO entry is in the EJ_TCK clock domain is normally transparent, but it can create some unexpected behavior:

- TxFIFO availability: Data is first written into the *SI_Clk* FIFO entries, then it will move into the *EJ_TCK* FIFO entry. But, it takes several *EJ_TCK* cycles to complete the handshake and move the data. *EJ_TCK* is generally much slower than *SI_ClkIn* and may even be stopped (although that would be uncommon when this feature is in use). This can result in there not being space for new data, even though there are only N-1 data values queued up. To prevent the loss of data, the $FDSTAT_{TxF}$ bit is set when all of the *SI_ClkIn* FIFO entries are full. Software writing to the FIFO should always check the $FDSTAT_{TxF}$ bit prior to attempting a write and should not make any assumptions about being able to arbitrarily use all entries. ie. software seeing the $FDSTAT_{FxE}$ bit set should not assume that it can write $FDCFG_{TxCnt}$ data words without checking for full.

- TxFIFO Almost Empty Interrupt: As transmit data moves from *SI_ClkIn* to *EJ_TCK*, both of the flops will temporarily look full. This makes it difficult to determine when just 1 FIFO entry is in use. To enable a simpler condition, the almost empty TxInterrupt condition is set when all of the *SI_ClkIn* FIFO entries are empty. When this condition is met, there will be 0 or 1 valid entries. However, the interrupt will not be asserted when there is only one valid entry if it is an *SI_ClkIn* entry

- The RxFIFO has similar characteristics but these are even less visible to software since *SI_ClkIn* must be running to access the FDC registers.

13.15.4 Sleep mode

FDC data transfers do not prevent the core from entering sleep mode and will proceed normally in sleep mode. The FDC block monitors the TAP interface signals with a free-running clock. When new receive data is available or transmit data can be sent, the gated clock will be enabled for a few cycles to transfer the data and then allowed to stop again. If FDC interrupts are enabled, transferring data may cause an interrupt to be generated which can wake the core up.

13.16 TCB Trigger Logic

The TCB is optionally implemented with trigger unit. If this is the case, then the $TCBCONFIG_{TRIG}$ field is non-zero. This section will explain some of the issues around triggers in the TCB.

13.16.1 TCB Trace Enabling

The TCB must be enabled in order to produce a trace to the trace funnel, when trace information is sent on the PDtrace interface. The main switch for this is the $TCBCONTROLB_{EN}$ bit. When set, the TCB will send trace information to the trace funnel. The TCB can optionally include trigger logic, which can control the $TCBCONTROLB_{EN}$ bit.

13.16.2 Tracing a Reset Exception

Tracing a reset exception is possible. However, the $TraceControl_{TS}$ bit is reset to 0 at core reset, so all the trace control must be from the TCB (using *TCBCONTROLA* and *TCBCONTROLB*). The PDtrace fifo and the entire TCB are reset based on an EJTAG reset. It is thus possible to set up the trace modes, etc., using the TAP controller, and then reset the core.

13.16.3 Trigger Units Overview

TCB trigger logic features three main parts:

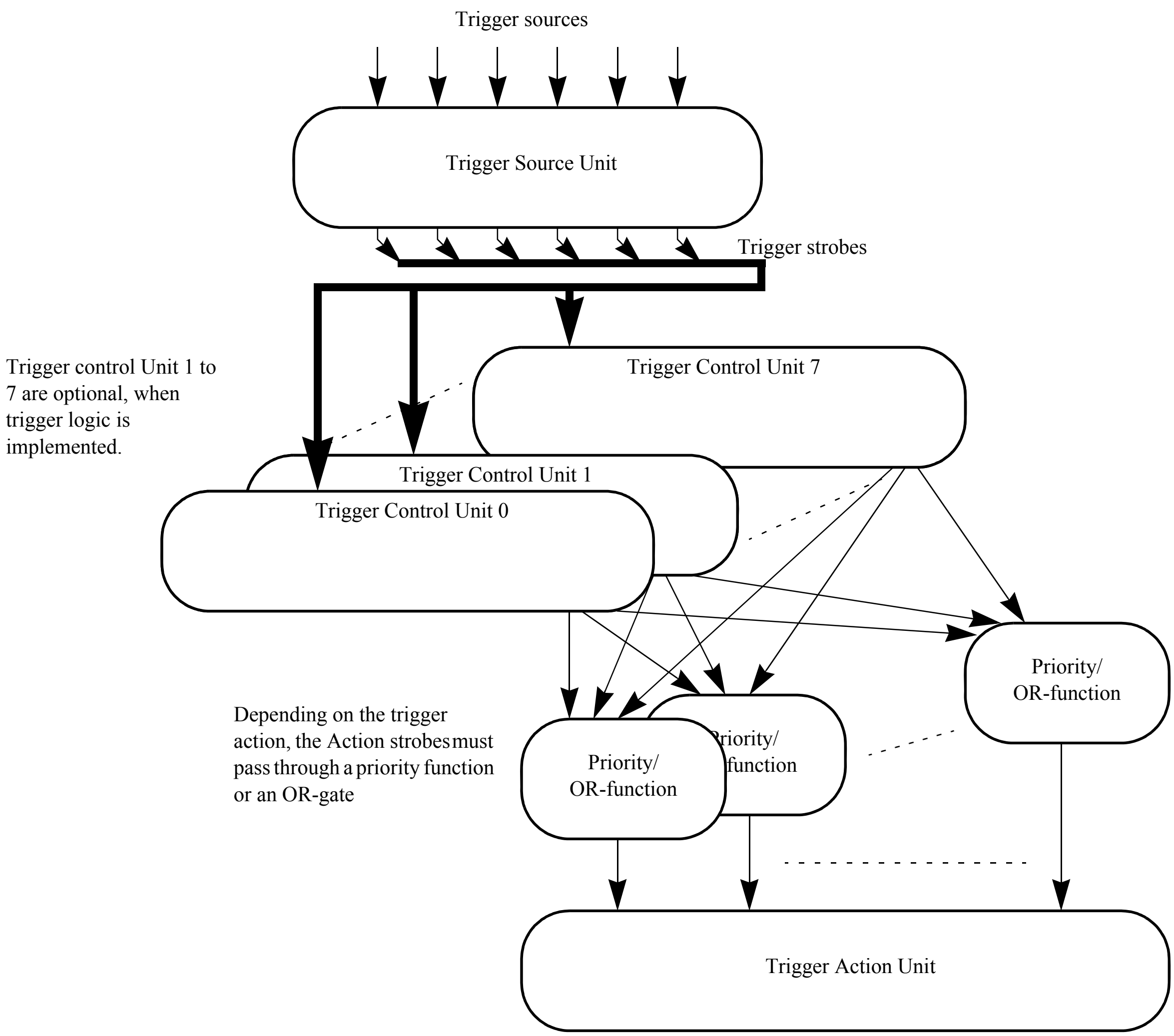
1. A common Trigger Source detection unit.

2. 1 to 8 separate Trigger Control units.

3. A common Trigger Action unit.

Figure 13.42 show the functional overview of the trigger flow in the TCB.

Figure 13.42 TCB Trigger Processing Overview



13.16.4 Trigger Source Unit

The TCB has three trigger sources:

1. Chip-level trigger input (*TC_ChipTrigIn*).
2. Probe trigger input (*TR_TRIGIN)*.
3. Debug Mode (DM) entry indication from the core.

The input triggers are all rising-edge triggers, and the Trigger Source Units convert the edge into a single cycle strobe to the Trigger Control Units.

13.16.5 Trigger Control Units

Up to eight Trigger Control Units are possible. Each of them has its own Trigger Control Register (*TCBTRIGx, x={0..7}*). Each of these registers controls the trigger fire mechanism for the unit. Each unit has all of the Trigger Sources as possible trigger event and they can fire one or more of the Trigger Actions. This is all defined in the Trigger Control register *TCBTRIGx* (see Section 13.14.10.8 "TCBTRIGx Register (Reg 16-23)").

13.16.6 Trigger Action Unit

The TCB has four possible trigger actions:

1. Chip-level trigger output (*TC_ChipTrigOut*).

2. Probe trigger output (*TR_TRIGOUT*).

3. Trace information. Put programmable information (TF6) into the trace stream from the TCB.

4. Start or End control of the $TCBCONTROLB_{EN}$ bit.

The basic function of the trigger actions is explained in Section 13.14.10.8 "TCBTRIGx Register (Reg 16-23)". Please also read the next Section 13.16.7 "Simultaneous Triggers".

13.16.7 Simultaneous Triggers

Two or more triggers can fire simultaneously. The resulting behavior depends on trigger action set for each of them, and whether they should produce a TF6 trace information output or not. There are two groups of trigger actions: Prioritized and OR'ed.

13.16.7.1 Prioritized Trigger Actions

For prioritized simultaneous trigger actions, the trigger control unit which has the lowest number takes precedence over the higher numbered units. The *x* in *TCBTRIGx* registers defines the number. The oldest trigger takes precedence over everything.

The following trigger actions are prioritized when two or more units fire simultaneously:

- Trigger Start and End ($TCBTRIGx_{Type}$ field set to 00 or 10), which will assert/de-assert the $TCBCONTROLB_{EN}$ bit.

- Triggers which produce TF6 trace information in the trace flow (Trace bit is set).

Regardless of priority, the $TCBTRIGx_{TR}$ bit is set when the trigger fires. This is so even if a trigger action is suppressed by a higher priority trigger action. If the trigger is set to only fire once (the $TCBTRIGx_{FO}$ bit is set), then the suppressed trigger action will not happen until after $TCBTRIGx_{TR}$ is written 0.

If a Trigger action is suppressed by a higher priority trigger, then the read value, when the $TCBTRIGx_{TR}$ bit is set, for the $TCBTRIGx_{Trace}$ field will be 0 for suppressed TF6 trace information actions. The read value in the $TCBTRIGx_{Type}$ field for suppressed Start/End triggers will be 11. This indication of a suppressed action is sticky. If any of the two actions (Trace and Type) are ever suppressed for a multi-fire trigger (the $TCBTRIGx_{FO}$ bit is zero), then the read values in Trace and/or Type are set to indicate any suppressed action.

13.16.7.2 OR’ed Trigger Actions

The simple trigger actions CHTro and PDTro from each trigger unit, are effectively OR’ed together to produce the final trigger. One or more expected trigger strobes on i.e. *TC_ChipTrigOut* can thus disappear. External logic should not rely on counting of strobes, to predict a specific event, unless simultaneous triggers are known not to occur.

Chapter 14

Multi-CPU Debug

This section describes the debug features of the P6600 Multiprocessing System. The following sections are included in this chapter:

- Section 14.1 “CM Performance Counters”
- Section 14.2 “Debug Mode Triggering”
- Section 14.3 “PDTrace Software Architecture”

14.1 CM Performance Counters

14.1.1 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 described in Chapter 6, “Coherence Manager” on page 325. This section describes the operation of those registers.

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 14.1. 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 14.1 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 14.2.

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 14.2):

```
    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 Registe*r 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).

14.1.2 Performance Counter Usage Models

There are several model for using the CM performance counters. This sections 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.

14.1.3 CM Performance Counter Event Types and Qualifiers

This section describes the Performance Counter Event Types and associated qualifiers.

Table 14.1 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 14.2	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 14.3	Gives a count of the specified coherent request and response types.
2	L2_WR_Data_Util	L2 Write Data Bus Usage	Accept State See Table 14.4	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 14.4	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_Util	L2 Read Data Bus Usage	L2 Data Width See Table 14.5	Counts number of cycles the L2/Memory read data bus is occupied. Qualifier determines if 64-bit cycles, 256-bit cycles, or both are counted.
5	Sharing_Miss	Sharing Frequency	Request Source Port Data Source Port See Table 14.6	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 14.7	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 14.8	Counts starts into the TA stage of the L2 pipeline.
9	L2_Hit	L2 Hit/Miss Usage	Hit/Miss Type Source Port See Table 14.9	Counts different types of L2 Cache Hits and Misses, crossed with Source Port ID.
16	IOCU_Request	IOCU Request	Transaction ID I/O Parking CM Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type See Table 14.10	Counts requests receive by the IOCU. The CM receives a sideband signal, SI_CMP_IOC_PerfInfo from the IOCU as described in Table 14.10.

Table 14.1 CM Performance Counter Event Types*(continued)*

Event #	Related Events	Use	Qualifiers	Description/Comments
17	IOCU1_Request	2nd IOCU Request	Transaction ID I/O Parking CM Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type See Table 14.10	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 14.10.

Table 14.2 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[1]	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
18	Burst Length[2] (# 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

Table 14.2 CM Performance Counter Request Count Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
15	Request Command	Legacy WR	Request is a legacy Write command. This is used for all non-coherent 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 CohReadShareAlways	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

1. CCA qualifier group is ignored on non-coherent cache-ops
2. Burst Length only used when Request Command is Legacy Read, Legacy Write, CohReadDiscard or CohWriteInval.

Table 14.3 CM Performance Counter Coherent Request/Response Qualifier

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

Table 14.3 CM Performance Counter Coherent Request/Response Qualifier*(continued)*

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	Request was 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	Request was 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.

Table 14.3 CM Performance Counter Coherent Request/Response Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
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 intervening request 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.
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 the command 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 14.4 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 14.5 CM Performance Counter L2 Data Width Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:2	Reserved		
1	L2 Data Width	256	Counts cycles where the L2/Memory returns data in 256-bit mode
0		64	Counts cycles where the L2/Memory returns data in 64-bit mode

Table 14.6 CM Performance Counter CM Data Source Qualifier

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
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 14.7 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 14.8 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 14.9 L2 Hit Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:20	Reserved		
19	Allocation (for Write or Read misses only)	Line allocated	A miss 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		Full line write hit	Full line write hit the L2 cache.
13		Partial line write hit	Partial line write hit the L2 cache. The line will be read, merged with the original write data, and replayed to complete the write.
12		Full line write miss	Full line write missed the L2 cache. Either allocates or writes through to memory depending on the L2 allocation policy.
11		Partial line write miss	Partial line write missed the L2 cache. Writes through to memory regardless of the L2 allocation policy.
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.

Table 14.9 L2 Hit Qualifier *(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
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 14.10 IOCU Performance Counter Request Count Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31	Reserved		
30:27	Transaction ID	TID	Value of IC_MTagID to match when the All_TID qualifier bit is set to 0. This field is unused when All_TID is 1.
26		All_TID	If 1 then the all values of IC_MTagID will match. If 0 then only transactions with IC_MTagID equal to the TID specified above will match.
25	I/O Parking	Start and Stop Parking	Request will start and stop I/O Parking.
24		Stop Parking	Request will stop I/O parking (but not start it).
23		Start Parking	Request will start I/O Parking (but not stop it).
22		No parking	Request will not start or stop I/O parking.
21	CM Transaction Count	5 CM Transactions	Request resulted in 5 CM transactions.
20		4 CM Transactions	Request resulted in 4 CM transactions.
19		3 CM Transactions	Request resulted in 3 CM transactions.
18		2 CM Transactions	Request resulted in 2 CM transactions.
17		1 CM Transaction	Request resulted in 1 CM transaction.
16	BurstLength	13-16	IC_MBurstLength is 13, 14, 15, or 16 dwords.
15		9-12	IC_MBurstLength is 9, 10, 11, or 12 dwords.
14		5-8	IC_MBurstLength is 5, 6, 7, or 8 dwords.
13		4	IC_MBurstLength is 4 dwords.
12		3	IC_MBurstLength is 3 dwords.
11		2	IC_MBurstLength is 2 dwords.
10		1	IC_MBurstLength is 1 dword.

Table 14.10 IOCU Performance Counter Request Count Qualifier *(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
9	L2 Allocation	L2 Allocation with Prepare for Store	Request will cause an L2 allocation and the request is a write with L2 Prepare For Store. This bit will never cause a match for read requests.
8		L2 Allocation without Prepare for Store	Request will cause an L2 allocation and the request is either a read or a write with L2 Prepare For Store not asserted.
7		No L2 Allocation	Request will not cause an L2 allocation.
6	Posted	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.

14.2 Debug Mode Triggering

This section describes the how to control the cores when entering debug mode.

14.2.1 Selecting CPUs to Enter Debug Mode

The P6600 Multiprocessing System contains a set of registers and logic that controls when the P6600 cores enter Debug mode. The logic allows software to:

- Specify which P6600 core enters debug mode on assertion of the *EJ_DINT_IN* signal (generally asserted by a debug probe).
- Force one or more P6600 cores to enter debug mode by writing to the *DINT Send to Group Register*.

14.2.2 Debug Mode Groups and Cross Triggering

The P6600 Multiprocessing System (MPS) allows software to define debug mode groups so that when one P6600 core enters debug mode, all other cores within the group also enter debug mode.

Software creates debug mode groups by writing to each core's *Core-Local DebugBreak Group Register*. Each bit in the *Join_DebugM* field of the *Core-Local DebugBreak Group Register* represents a core in the system. If the bit is set, the corresponding core will enter debug mode. If the bit is clear, the corresponding core is not affected by Debug Mode.

Only the positive edge of a core's *EJ<cpu>_DebugM* signal can cause the other CPUs to also enter the Debug Mode as a group. When there is no positive edge on the *DebugM* signals, the *Join_DebugM* fields in the *DebugBrk_Group* registers can be written without causing spurious glitches on the *EJ<cpu>_DINT* signals.

14.2.3 Debug Cross Trigger Facility and Power Management

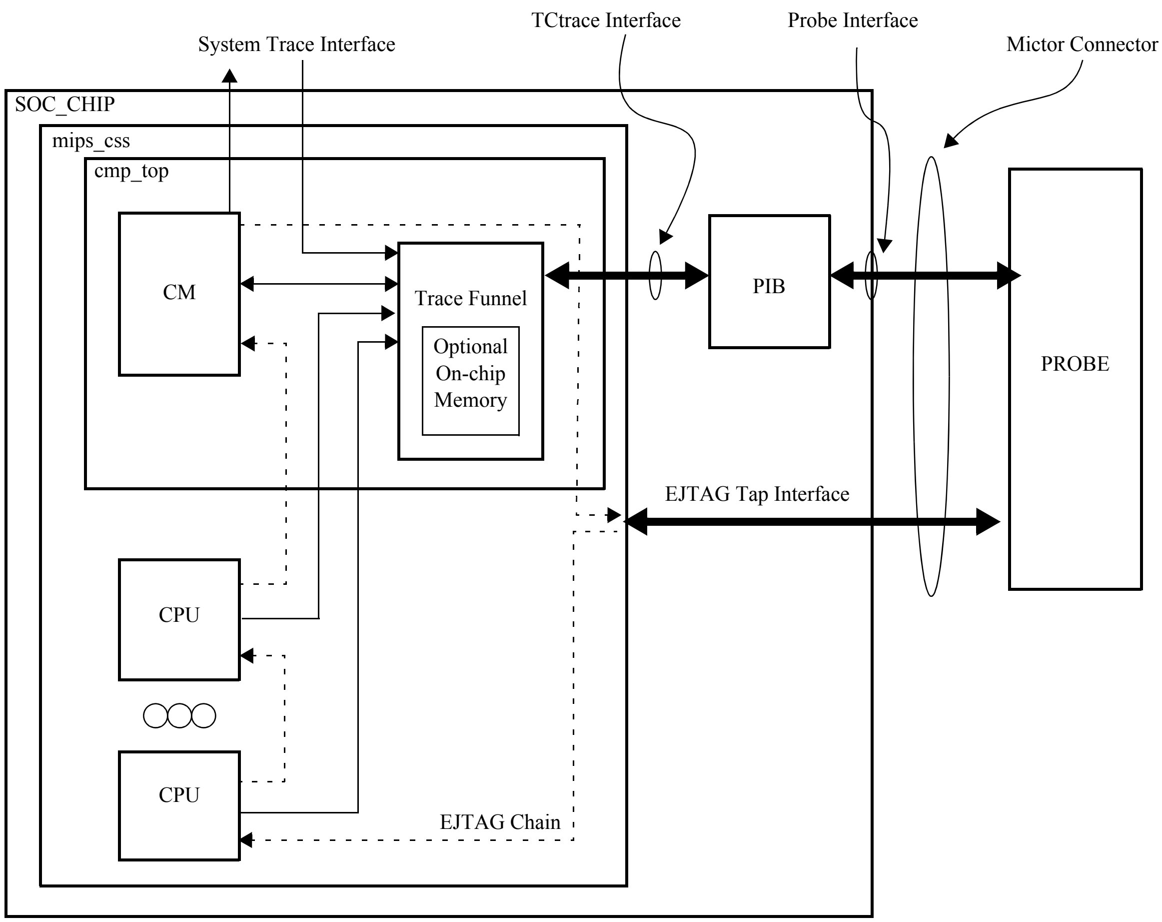
Due to power management of P6600 Multiprocessing System components, CPUs might not be powered or clocked when receiving a DINT via the debug cross trigger facility. However, the power controller observes all DINT events and will start up domains as requested. Depending on the programming of the power controller and time constants of the physical design, a delay between DINT event and a target CPU participating in the debug session might occur. To inquire about the current power status of a CPU, the debug handler can poll the power controller status registers. Generally, an EJTAG debug probe attached and recognized by the system will shorten the wake-up delay, while debug events without debug probe attachment might show more wake-up latency.

14.3 PDTrace Software Architecture

The P6600 MPS enables debug trace information from the P6600 cores, the Coherence Manager, and a System Trace Interface to be streamed off chip or stored in on-chip RAM. As shown in Figure 14.1, each P6600 core produces a 128-bit debug trace stream describing its program and data flow. The CM produces a 64-bit stream describing the flow of transactions within the CM. If a System Trace Interface is part of the build, it captures a 128-bit stream describing activity supplied externally by the System.

The Trace Funnel muxes the CPU, CM, and System Trace streams into a single debug trace stream which is either stored in an on-chip buffer or passed onto a Probe Interface Block (PIB). A PIB is the on-chip link between the Trace Funnel and debug probe interface, and may include functionality such as time multiplexing the 128-bit TCtrace data onto a narrower, slower probe interface.

Figure 14.1 PD Trace Architecture



The TCtrace stream consists of 64-bit trace words (TW). Each trace word trace is packed with one or more Trace Formats (TF). There are many trace format types produced by CPUs and the CM. The CPU TFs allow tracing of information such as the program counter, load/store addresses, and load/address data values. The CM TFs produce information such as the serialization order of requests and the results of L1 cache interventions.

The trace output of each CPU can be controlled by a set of EJTAG accessible registers located in the Trace Control Block (TCB) associated with that CPU.

14.3.1 CM Trace Functionality

This section describes the configuration and functionality of the CM debug trace.

14.3.1.1 CM Trace Configuration and Control

The CM Trace is controlled by the *CM TCBCONTROLD Register* as defined in Section "TCBCONTROLD Register". The enabling of the CM's Trace is determined by two fields in this register along with a field in each of the core's TCBCONTROLD register. Figure 14.11 shows that there are two ways to enable CM Trace. First, CM Trace can be enabled independent of the Cores' state by setting both *CM_EN* and *Global_CM_En* in the CM's *TCBCONTROLD* Register. Alternatively, by setting *CM_EN* and clearing *Global_CM_En*, the CM will only trace if at least one other core is tracing, i.e., *Core_CM_En* in at least one core's TCBCONTROLD register is set to 1. A core's *Core_CM_En* bit may be asserted/deasserted based on debug triggers as defined in *The MIPS64® P6600™ Processor Core Family Software User's Manual*. The value of each core's *Core_CM_En* bit is communicated to the CM on the *TC<core>_Trace_CM_En* signal.

Table 14.11 CM Trace Enable

CM TCBCONTROLD Reg		Cores' TCBCONTROLD Reg	CM PDTrace Enabled/Disabled
CM_EN	**Global_CM_En**	**Core_CM_En**	
0	x	x	Disabled
1	1	x	Enabled
1	0	All 0	Disabled
1	0	not All 0	Enabled

14.3.1.2 System Trace Interface Configuration and Control

The System Trace Interface stream is generated and controlled by external logic. The CM has control output pins to support design of this logic. There are 2 specific control outputs and one 32-bit user-defined output. These outputs and the trace data/contol pins associated with the trace stream are shown in Table 14.12. All the signals are timed relative to the *SI_CMClk*.

Table 14.12 System Trace Interface Stream and Control Pins

Signal	Direction/Type	Usage
SI_TC_Sys_Data[127:0]	CM stream input	System Trace stream data for 128-bit stream SI_TC_Sys_Data[71:68] must contain a Source Port ID and SI_TC_Sys_Data[7:4] must contain a Source Port ID. Legal values of either Source Port ID are: 4'hc or 4'hd. All other bits are completely user defined
SI_TC_Sys_Valid[1:0]	CM stream input	System Trace stream valid bits for upper and lower streams Bit 1 qualifies SI_TC_Sys_Data[127:64] Bit 0 qualifies SI_TC_Sys_Data[63:0] A value of 2'b10 is illegal
SI_TC_Sys_Stall	CM stream output	System Trace stream flow control.
SI_TC_Sys_Enable	CM control output	System Trace control advisory, driven from the *CM TCBCONTROLD*$_{ST\_En}$. Its purpose is to advise the external logic of the state of this control bit. If desired, external logic can stop generation of the stream if this output is a zero, and allow generation of the stream if it is a 1. However, external logic may choose to continue sending stream data after de-assertion until it has flushed all its collected stream data.
SI_TC_Sys_AnyCore_Enabled	CM control output	System Trace control advisory that at least one core is enabled to trace, derived from Cores' TCBCONTROLD register.
SI_TC_Sys_CM_Enabled	CM control output	System Trace control advisory that the CM2 is enabled to trace, derived from CM2's TCBCONTROLD register.
SI_TC_Sys_UserCtl[31:0]	CM control output	User defined control advisory bits, from TCBSYS. Bit 31 is a 1 when the Trace Funnel was configured with the System Trace present and is a 0 when the System Trace is not present. Bits [30:0] are completely user defined output values.

In addition to the System Trace Interface pins, there are internal control register bits that impact operation of the System Trace stream. Assertion of *CM TCBCONTROLB*$_{STCE}$ allows the System Trace funnel port to capture stream data; de-assertion of this bit causes the Trace Funnel to stop capturing the System Trace stream from within the Trace Funnel in case the external logic is problematic. In addition, de-assertion of *CM TCBCONTROLB*$_{EN}$ stops capture of all the streams (Cores, CM, System).

Thus the System Trace stream is enabled to capture the System Trace stream when these controls are asserted: *CM TCBCONTROLB*$_{STCE}$ and *CM TCBCONTROLB*$_{EN}$. The control outputs *SI_TC_Sys_Enable* and *SI_TC_Sys_UserCtl[31:0]* are available to the external logic to further control generation of the System Trace stream by allowing or disallowing assertion of the *SI_TC_Sys_Valid[1:0]* inputs. If any trace stream is being generated without enabling that stream to capture, then that stream is not captured and the data is dropped.

14.3.1.3 Trace Funnel Enable

When trace on the System, CM and/or Cores is enabled then trace information is continuously sent to the Trace Funnel. However, the trace funnel will only send the trace information to the trace probe or to the on-chip trace memory if it is enabled by setting the *CM TCBCONTROLB*$_{EN}$ bit. The Trace Funnel can be subsequently disabled by clearing the *CM TCBCONTROLB*$_{EN}$ bit. See "TCBCONTROLB Register Field Descriptions" on page 762 for more information.

14.3.1.4 CM Trace Formats

Trace information is captured at two points within the CM:

- Information about requests is captured by the Request Unit (RQU) after serialization, thus providing a view of the global order of requests.
- Information about L1 interventions is captured by the Intervention Unit (IVU) after all intervention responses have been received. This provides information about the state of the cache line in all L1 caches for coherent requests.

The type and amount of content in each Trace Format created by the CM depends on the source of the packet (RQU or IVU) and the configuration (TLev, AE, P<port>_Ctl control bits). Refer to *The PDtrace™ Interface and Trace Control Block Specification* for the detailed description of the CM Trace Formats.

14.3.1.5 CM / CPU Core Trace Correlation

In the P6600 core, trace information is provided from each of the cores as well as the Coherence Manager. In order to correlate transactions from the CM to the instruction stream, an identifier is used in both the core and CM traces.

The CM trace includes the core ID and CosID for each request. The CosID changes relatively slowly - it is generally incremented after PCSync in the core or if an overflow is detected in the CM. Typically several requests in a row will use the same CosID value, and the intermediate correlation is enabled by the requests appearing in the same order in the CM and core traces. Because of this, and the fact that the CosID is traced as a part of the instruction completion record, correlating instructions to CM transactions is possible only when PC tracing is enabled for all TCs executing on the core.

The PDtrace™ Interface and Trace Control Block Specification includes a more detailed description of the correlation process.

14.3.2 Controlling Trace in a Multi-CPU Multiprocessing System

The P6600 MPS enables debug trace information from the P6600 cores and the Coherence Manager to be streamed off chip or stored in on-chip RAM. As shown in Figure 14.1, each P6600 core produces a 64-bit debug trace stream describing its program and data flow. The CM produces a stream describing the flow of transactions within the CM2. The Trace Funnel muxes the CPU and CM trace streams into a single debug trace stream which is either stored in an on-chip buffer or passed onto a Probe Interface Block (PIB). A PIB is the on-chip link between the Trace Funnel and debug probe interface, and may include functionality such as time multiplexing the 64-bit TCtrace data onto a narrower, slower probe interface.

Since the P6600 core streams PDTrace data directly to the trace funnel, the core TCB system is configured as if only off-chip trace is present. Core TCB register bits which refer to control of on-chip trace resources will behave as it on-chip trace is not implemented.

The CM has its own set of TCBControl registers. It is designated as the ‘master’ which controls trace functionality for the CM, the on-chip trace buffer, and the PIB interface. In addition to the CM2 as trace master, the GCR block itself can function as the trace master in the P6600 core. This is done through memory mapped CM_GCR global control registers.

14.3.3 EJTAG Debug Support in the P6600 Coherence Manager

The EJTAG debug logic in the Coherence Manager is compliant with EJTAG Specification 6.0 and includes:

1. Standard Test Access Port (TAP) for a dedicated connection to a debug host

2. Optional PDtrace capability for program counter/data address/data value trace to On-chip memory or to Trace probe

The following sub-sections describe the TAP and EJTAG operation and registers.

14.3.3.1 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST_N*) interface which is compatible with IEEE Std. 1149.1.

- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.

EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

Table 14.13 EJTAG Interface Pins

Pin	Type	Description
TCK	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the CM clock, so the EJTAG probe can drive *TCK* independently of the CM clock frequency. The CM signal for this is called *EJ_TCK*

Table 14.13 EJTAG Interface Pins*(continued)*

Pin	Type	Description
TMS	I	Test Mode Select Input The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*. The CM signal for this is called *EJ_TMS*
TDI	I	Test Data Input Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state. The CM signal for this is called *EJ_TDI*
TDO	O	Test Data Output Serial output data is shifted from the Instruction or data register to the *TDO* pin on the falling edge of the *TCK* clock. When no data is shifted out, the *TDO* is 3-stated. The CM signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*.
TRST_N	I	Test Reset Input (Optional pin) The *TRST_N* pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the main CM logic. The CM's transaction processing logic is not reset by the assertion of *TRST_N*. The CM signal for this is called *EJ_TRST_N* This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.

Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 14.2. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

At power-up the TAP is forced into the *Test-Logic-Reset* by low value on *TRST_N*. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

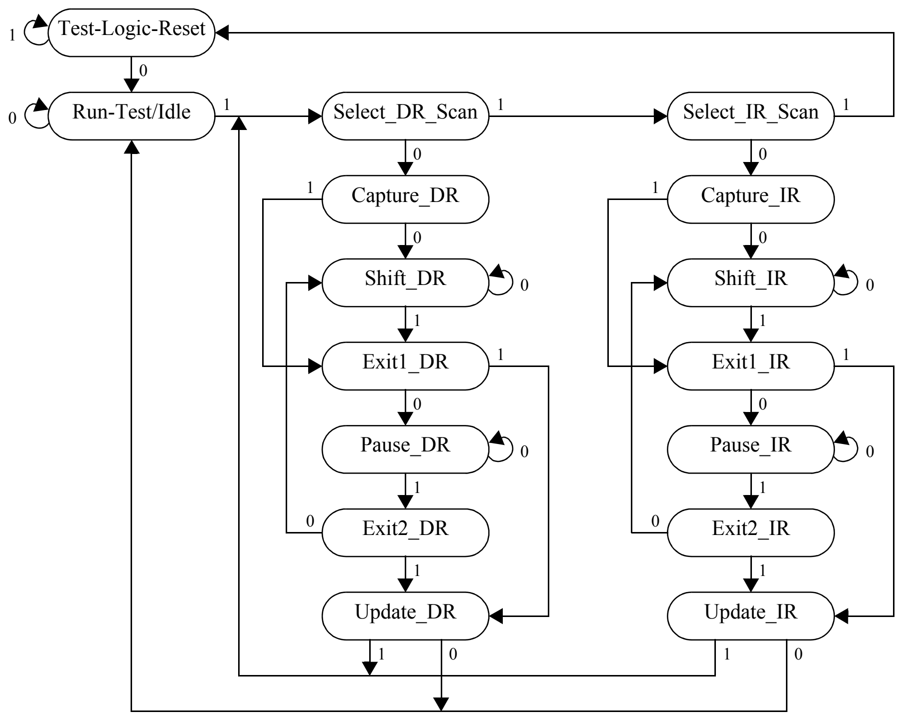
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 14.2.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the *Capture* states, the TAP transitions to either the *Shift* or *Exit1* states. Normally the *Shift* state follows the *Capture* state so that test data or status information can be shifted out for inspection and new data shifted in. Following the *Shift* state, the TAP either returns to the *Run-Test/Idle* state via the *Exit1* and *Update* states or enters the *Pause* state via *Exit1*. The reason for entering the *Pause* state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the Pause state shifting can resume by re-entering the *Shift* state via the *Exit2* state or terminate by entering the *Run-Test/Idle* state via the *Exit2* and *Update* states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the *Update-DR* or *Update-IR* state. The *Update* state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.

Figure 14.2 TAP Controller State Diagram



Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the *TMS* input is held HIGH for at least five rising edges of *TCK*. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as *TMS* is HIGH.

Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as *TMS* is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When *TMS* is sampled HIGH on the rising edge of *TCK*, the controller transitions to the *Select_DR* state.

Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Capture_DR* state. A

HIGH on *TMS* causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller transitions to the *Capture_IR* state. A HIGH on *TMS* causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

Capture_DR State

In this state the boundary scan register captures the value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift_DR*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

Shift_DR State

In this state the test data register connected between *TDI* and *TDO* as a result of the current instruction shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

Pause_DR State

The *Pause_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_DR* state. The instruction cannot change while the TAP controller is in this state.

Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

Update_DR State

When the TAP controller is in this state the value shifted in during the *Shift_DR* state takes effect on the rising edge of the *TCK* for the register indicated by the Instruction register.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001_2) on the rising edge of *TCK*. The data registers selected by the current instruction retain their previous state.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

Shift_IR State

In this state the instruction register is connected between *TDI* and *TDO* and shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state.

Exit1_IR State

This is a temporary controller state in which all registers retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_IR* state. The instruction cannot change while the TAP controller is in this state.

Exit2_IR State

This is a temporary controller state in which the instruction register retains its previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

Update_IR State

The instruction shifted into the instruction register takes effect on the rising edge of *TCK*.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state.

14.3.3.2 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the *Shift-IR* state. Instructions are decoded and define the serial test data register path that is used to shift data between *TDI* and *TDO* during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions default to the BYPASS instruction.

Table 14.14 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register.
0x03	IMPCODE	Select Implementation register.
0x08	Reserved	Instructions using this code select bypass register.
0x09	Reserved	Instructions using this code select bypass register.
0x0A	CONTROL	Select EJTAG Control register.
0x0B	Reserved	Instructions using this code select bypass register.
0x0C	Reserved	Instructions using this code select bypass register.
0x0D	Reserved	Instructions using this code select bypass register.
0x0E	Reserved	Instructions using this code select bypass register.
0x10	Reserved	Instructions using this code select bypass register.
0x11	TCBCONTROLB	Selects the *TCBCONTROLB* register in the Trace Control Block.
0x12	TCBDATA	Selects the *TCBDATA* register in the Trace Control Block.
0x13	Reserved	Instructions using this code select bypass register.
0x14	Reserved	Instructions using this code select bypass register.
0x15	TCBCONTROLD	Selects the *TCBCONTROLD* register in the Trace Control Block.
0x16	TCBCONTROLE	Selects the *TCBCONTROLE* register in the Trace Control Block.
0x17	Reserved	Instructions using this code select bypass register.
0x1F	BYPASS	Bypass register.

BYPASS Instruction

The required BYPASS instruction selects the Bypass register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the CM from *TDI* to *TDO* without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

IDCODE Instruction

The IDCODE instruction selects the Device Identification (ID) register to be connected between *TDI* and *TDO*. The Device ID register is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the CM. Also, access to the Identification Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional *TRST_N* pin.

IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

CONTROL Instruction

This instruction is used to select the EJTAG Control register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the EJTAG Control register and shifts out the EJTAG Control register bits via *TDO*.

TCBCONTROLB Instruction

This instruction is used to select the TCBCONTROLB register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

TCBCONTROLD Instruction

This instruction is used to select the TCBCONTROLD register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

TCBCONTROLE Instruction

This instruction is used to select the TCBCONTROLE register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

14.3.3.3 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation the TAP controller selects the output of the Instruction register to drive the *TDO* pin. The shift register consists of a series of bits arranged to form a single scan path between *TDI* and *TDO*. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from *TDI* to *TDO*. Both the capture and shift operations occur on the rising edge of *TCK*. However, the data shifted out from the *TDO* occurs on the falling edge of *TCK*. In the Test-Logic-Reset and *Capture-IR* state, the instruction shift register is set to 00001_2, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in Table 14.14.

14.3.3.4 Data Registers Overview

The EJTAG uses several data registers, which are arranged in parallel from the primary *TDI* input to the primary *TDO* output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from *TDI* to *TDO*. During a data register scan operation, the TAP selects the output of the data register to drive the *TDO* pin. The register is updated in the *Update-DR* state with respect to the write bits.

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)

Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the Bypass register provides a single bit scan path between *TDI* and *TDO*. The Bypass register allows abbreviating the scan path through devices that are not involved in the test. The Bypass register is selected when the Instruction register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 14.15 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the CM determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

Figure 14.3 Device Identification Register Format

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 14.15 Device Identification Register

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
Version	31:28	**Version** (4 bits) This field identifies the version number of the CM.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the CM.	R	*EJ_PartNumber[15:0]*
ManufID	11:1	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	*EJ_ManufID[10:0]*
R	0	reserved	R	1

Implementation Register

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset values are set by inputs to the CM2. The register is selected when the Instruction register is loaded with the IMPCODE instruction.

Figure 14.4 Implementation Register Format

31–29	28–14	13–11	10–1	0
EJTAGver	reserved	Type	TypeInfo	r

Table 14.16 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	Indicates EJTAG Version 6.0.	R	6
reserved	28:14	reserved	R	0
Type	13:10	Type of Entity associated with this TAP. 2: TAP is attached to a Trace-Master. TypeInfo field is not used.	R	2
TypeInfo	10:1	Identifier Information. Unused because this TAP is connected to a Trace-Master as indicated by the Type field.	R	0
reserved	0	reserved	R	0

EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in the CONTROL instruction. Bits in the EJTAG Control register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This EJTAG Control register can only be accessed by the TAP interface.

The EJTAG Control register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0.

The value used for reset indicated in the table below takes effect on CM2 resets, but not on TAP controller resets by e.g. *TRST_N*. *TCK* clock is not required when the CM2 reset occurs, but the bits are still updated to the reset value when the *TCK* is applied. The first 5 *TCK* clocks after CM2 resets may result in reset of the bits, due to synchronization between clock domains.

Figure 14.5 EJTAG Control Register Format

31	28–23	22	21	20–0
Rocc	Reserved	Doze	Halt	Reserved

Table 14.17 EJTAG Control Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Rocc	31	Reset Occurred The bit indicates if a CM reset has occurred: 0: No reset occurred since bit last cleared. 1: Reset occurred since bit last cleared. The Rocc bit will keep the 1 value as long as reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected. The EJTAG Control register is not updated in the *Update-DR* state unless Rocc is 0, or written to 0. This is in order to ensure proper handling of processor access.	R/W	1

Table 14.17 EJTAG Control Register Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
Res	30:23	Reserved	R	0
Doze	22	Tied to 0.	R	0
Halt	21	Halt state The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller: 0: Internal CM clock is running 1: Internal CM clock is stopped	R	0
Res	20:0	Reserved	R	0

14.3.3.5 CM2 Trace Control Block (TCB) Registers

The TCB registers used to control its operation are listed in Table 14.18 and Table 14.19. These registers, except for *TCBDATA*, are accessed via the EJTAG TAP interface as well as by the P6600 core via memory-mapped accesses to the Global Debug Control Block in the CM GCRs. *TCBDATA* can only be accessed via the EJTAG TAP interface. Note that the TCB registers are implemented only if PDTrace is selected at build time.

Table 14.18 TCB EJTAG Registers

EJTAG Register	Memory-Mapped Address*	Name	Description
0x11	0x0008	*TCBCONTROLB*	Control register in the TCB that is mainly used to specify what to do with the trace information. The *REG* [25:21] field in this register specifies the number of the TCB internal register accessed by the *TCBDATA* register. A list of all the registers that can be accessed by the *TCBDATA* register is shown in Table 14.19. See Section "TCBCONTROLB Register".
0x15	0x0010	*TCBCONTROLD*	Control register in the TCB used to control tracing from the Coherence Manager Section "TCBCONTROLD Register"
0x16	0x0020	*TCBCONTROLE*	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section "TCBCONTROLE Register".

Table 14.19 Registers Selected by $TCBCONTROLB_{REG}$

TCBCONTROLB$_{REG}$ Field	Memory Mapped Address*	Name	Reference	Notes
0	0x0028	TCBCONFIG	Section "TCBCONFIG Register (Reg 0)"	
4	0x0200/0x0208**	TCBTW	Section "TCBTW Register (Reg 4)"	These registers have no function if on-chip memory does not exist.
5	0x0108	TCBRDP	Section "TCBRDP Register (Reg 5)"	
6	0x0110	TCBWRP	Section "TCBWRP Register (Reg 6)"	
7	0x0118	TCBSTP	Section "TCBSTP Register (Reg 7)"	
17-29		reserved		
30	0x0040	TCBSYS	Section "TCBSYS Register (Reg 30)"	

Table 14.19 Registers Selected by TCBCONTROLB$_{REG}$ *(continued)*

TCBCONTROLB$_{REG}$ Field	Memory Mapped Address*	Name	Reference	Notes
31		TCBBYPASS		

* Memory-Mapped Address relative to the Global Debug Block in the CM GCRs.
** Memory-Mapped Access for TCBTW is split into two 32-bit registers: TCBTW_LO (address 0x0200) accesses TCBTW[31:0]. TCBTW_HI (address 0x0208) accesses TCBTW[63:32]

TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (EJTAG Register 0x11). This register generally controls what to do with the trace information received. This register is also mapped to offset 0x0008 in the Global Debug Block of the CM GCRs.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 14.20.

Figure 14.6 TCBCONTROLB Register Format

31	30 28	27 26	25 21	20	19	18	17	16	15	14	13 12	11	10 8	7	6 3	2	1	0
WE	0	TWSrcWidth	REG	WR	STCE	TRPAD	0	RM	TR	BF	TM	0	CR	Cal	0	CA	OfC	EN

Table 14.20 TCBCONTROLB Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
WE	31	Write Enable. Only when set to 1 will the other bits be written in *TCBCONTROLB*. This bit will always read 0.	R	0
Reserved	30:28	Reserved. Must be written as zero; returns zero on read.	R	0
TWSrc-Width	27:26	Used to indicate the number of bits used in the source field of the Trace Word. The value for the CM is always 0b10 indicationg a four bit source field width.	R	10
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 14.19. Note: Although this field can be written via memory-mapped GCR or EJTAG accesses, the *TCBDATA* register is only accessible via EJTAG access.	R/W	0
WR	20	Write Registers: When set, the register selected by REG field is read and written when *TCBDATA* is accessed. Otherwise the selected register is only read. Note: Although this field can be written via memory-mapped GCR or EJTAG accesses, the *TCBDATA* register is only accessible via EJTAG access.	R/W	0
STCE	19	System Trace capture enable. When asserted, the System Trace port of the Funnel is enabled to capture System Trace stream data. When not asserted, System Trace stream data is not captured regardless of *SI_TC_Sys_Valid[1:0]* input pin state.	R/W	0

Table 14.20 TCBCONTROLB Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
TRPAD	18	Trace RAM access disable bit. When set to 1 core reads and writes to the on-chip trace RAM using GCR accesses are inhibited. If TRPAD is set, memory-mapped writes to the GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI registers have no effect, and memory-mapped reads from GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI do not access the Trace RAM and 0 is returned. Also, when TRPAD is set, then memory-mapped writes to all CM TCB registers listed in Table 14.19 are inhibited.	R/W	0
Reserved	17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	16	Read on-chip trace memory. When written to 1, the read address-pointer of the on-chip memory in register *TCBRDP* is set to the value held in *TCBSTP*. Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer in register *TCBRDP* after each read. When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros. Once set to 1, writing 1 again will have no effect. The bit is reset by setting the TR bit or by reading the last Trace word in *TCBTW*. This bit has no function if on-chip memory is not implemented.	R/W1	0
TR	15	Trace memory reset. Trace memory reset. When written to one, the address pointers for the on-chip trace memory *TCBSTP, TCBRDP and TCBWRP* are reset to zero. Also the RM and BF bits are reset to 0. This bit is automatically reset back to 0, when the reset specified above is completed.	R/W1	0
BF	14	Buffer Full indicator that the TCB uses to communicate to external software in the situation that the on-chip trace memory is being deployed in the **trace-from** and **trace-to** mode. This bit is cleared when writing 1 to the *TR* bit. This bit has no function if on-chip memory is not implemented.	R	0

Table 14.20 TCBCONTROLB Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TM</td><td>13:12</td><td>Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace interface to start or stop trace.
<table><tr><th>TM</th><th>Trace Mode</th></tr><tr><td>00</td><td>Trace-To</td></tr><tr><td>01</td><td>Trace-From</td></tr><tr><td>10</td><td>Reserved</td></tr><tr><td>11</td><td>Reserved</td></tr></table>In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around and overwriting older Trace Words, as long as there is trace data coming from the core.
In Trace-From mode, the on-chip trace memory is filled from the point that the core starts tracing until the on-chip trace memory is full.
In both cases, de-asserting the EN bit in this register will also stop fill to the trace memory.
If a *TCBTRIGx* trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode.
These bits have no function if on-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>11</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CR</td><td>10:8</td><td>Off-chip Clock Ratio. Writing this field, sets the ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 14.21.
Note: As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per core clock rising edge.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>100_2</td></tr>
</table>

Table 14.20 TCBCONTROLB Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Cal</td><td>7</td><td>Calibrate off-chip trace interface.
If set to one, the off-chip trace pins will produce the following pattern in consecutive trace clock cycles. If more than 4 data pins exist, the pattern is replicated for each set of 4 pins. The pattern repeats from top to bottom until the Cal bit is de-asserted.
<table>
<tr><th colspan="5">Calibrations pattern</th></tr>
<tr><th></th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="13">This pattern is replicated for every 4 bits of TR_DATA pins.</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>
Note: The clock source of the TCB and PIB must be running.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>6:2</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>OfC</td><td>1</td><td>If set to 1, trace is sent to off-chip memory using TR_DATA pins.
If set to 0, trace info is sent to on-chip memory.
This bit is read only if a single memory option exists (either off-chip or on-chip only).</td><td>R/W</td><td>Preset</td></tr>
<tr><td>EN</td><td>0</td><td>Funnel Trace Enable. When this bit is set, the trace funnels accepts trace information from the CM and/or cores and writes the information to off-chip or on-chip memory.
When this bit is cleared, the trace funnel drops all new trace information from the CM and/or cores . The trace information already accepted by the trace funnel is sent to the off-chip or on-chip memory, but new trace information is dropped and not written out.</td><td>R/W</td><td>0</td></tr>
</table>

The Probe Interface Block (PIB) has been an available component with many previous MIPS cores, including the P6600 core. The P6600 core architecture brings two significant changes to the PIB. First, the PIB is now instantiated in mips_css. Second, this new version of the PIB, referred to as PIB2, provides additional clock ratios.

The PIB2 provides available TR_CLK to processor clock ratios of 1:2, 1:4, 1:6, 1:8, 1:10, 1:12, 1:16, and 1:20. The PIB1 supplied by MIPS has only the ratios 1:2, 1:4, 1:6, and 1:8. The PIB1 architecture also has provision for clock multiples, 1:1, 2:1, 4:1, and 8:1, but these are not supported in PIB2.

The PIB2 reports the minimum CR (TC_CRMin) as 3'b111 and maximum (TC_CRMax) as 3'b000 as shown in the table below. This is how software identifies a PIB2 as opposed to PIB.

Table 14.21 Clock Ratio Encoding of the CR field

TC_ClockRatio	TR_CLK : gclk
3'b000	1:20
3'b001	1:16
3'b010	1:12
3'b011	1:10
3'b100	1:2
3'b101	1:4
3'b110	1:6
3'b111	1:8

TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the $TCBCONTROLB_{REG}$ field; see Table 14.19. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the $TCBCONTROLB_{WR}$ bit is set. For read-only registers, $TCBCONTROLB_{WR}$ is a don't care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 14.22. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 14.7 TCBDATA Register Format

31(63) ... 0
Data

Table 14.22 TCBDATA Register Field Descriptions

Fields				
Names	Bits	Description	Read/Write	Reset State
Data	31:0 63:0	Register fields or data as defined by the $TCBCONTROLB_{REG}$ field	Only writable if $TCBCONTROLB_{WR}$ is set	0

TCBCONTROLD Register

The TCB includes a second control register, TCBCONTROLD (EJTAG Register 0x14), whose values are used to control the tracing functions of the Coherence Manager. External software (i.e., debugger) can therefore manipulate the trace output by writing to this register. This register is also mapped to offset 0x0010 in the Global Debug Block of the CM GCRs.

The format of the *TCBCONTROLD* register is shown below, and the fields are described in Table 14.8

Figure 14.8 TCBCONTROLD Register Format

31 26	25 24	23 22	21 20	19 18	17 16	15 12	9 8		6	5	4 3	2	1	0
Reserved	P4_Ctl	P3_Ctl	P2_Ctl	P1_Ctl	P0_Ctl	Reserved	TWSrcVal	WB	ST_En	IO	TLev	AE	Global_CM_En	CM_EN

Table 14.23 TCBCONTROLD Register Definitions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Reserved</td><td>31:30</td><td>Reserved for future use. Must be written as 0.</td><td>R</td><td>0</td></tr>
<tr><td>P6_Ctl</td><td>29:28</td><td>Implementation specific finer grained control over tracing Port 6 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P5_Ctl</td><td>27:26</td><td>Implementation specific finer grained control over tracing Port 5 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P4_Ctl</td><td>25:24</td><td>Implementation specific finer grained control over tracing Port 4 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P3_Ctl</td><td>23:22</td><td>Implementation specific finer grained control over tracing Port 3 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P2_Ctl</td><td>21:20</td><td>Implementation specific finer grained control over tracing Port 2 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P1_Ctl</td><td>19:18</td><td>Implementation specific finer grained control over tracing Port 1 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>P0_Ctl</td><td>17:16</td><td>Implementation specific finer grained control over tracing Port 0 traffic at the CM. See Table 14.24.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>15:12</td><td>Reserved for future use. Must be written as 0 and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>TWSrcVal</td><td>11:8</td><td>The source ID of the CM.</td><td>R/W</td><td>0</td></tr>
<tr><td>WB</td><td>7</td><td>When this bit is set, Coherent Writeback requests are traced. If this bit is not set, all Coherent Writeback requests are suppressed from the CM trace stream.</td><td>R/W</td><td>0</td></tr>
<tr><td>ST_En</td><td>6</td><td>System Trace Enable. Driven to the CM ouput pin SI_TC_Sys_Enable. External logic can use this output to control generation of the System Trace stream.</td><td>R/W</td><td>0</td></tr>
<tr><td>IO</td><td>5</td><td>Inhibit Overflow on CM FIFO full condition. When set to 1 the CM never drops trace words, but instead will stall the request and/or intervention processing until forward progress can be made.
When set to 0 the CM will drop trace words when the trace word FIFO overflows.</td><td>R/W</td><td>0</td></tr>
<tr><td>TLev</td><td>4:3</td><td>This defines the current trace level being used by CM tracing
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>No Timing Information</td></tr>
<tr><td>01</td><td>Include Stall Times, Causes</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>AE</td><td>2</td><td>When set to 1, address tracing is always enabled for the CM. This affects trace output from the serialization unit of the CM. When set to 0, address tracing may be enabled through the implementation specific P[x]_Ctl bits.</td><td>R/W</td><td>0</td></tr>
</table>

Table 14.23 TCBCONTROLD Register Definitions

Fields		Description	Read / Write	Reset State
Name	Bits			
Global_CM_En	1	Each CPU core can enable or disable CM tracing using this bit. This bit is not routed through the master core, but is individually controlled by each core. Setting this bit can enable tracing from the CM even if tracing is being controlled through software, if all other enabling functions are true.	R/W	0
CM_EN	0	This is the master trace enable switch to the CM. When zero, tracing from the CM is always disabled. When set to one, tracing is enabled if other enabling functions are true.	R/W	0

Table 14.24 P<port>_Ctl Trace Control Field

Value	Meaning
2'b00	Tracing Enabled, No Address Tracing, assuming AE = 0
2'b01	Tracing Enabled, Address Tracing Enabled, independent of AE
2'b10	Reserved
2'b11	Tracing Disabled

The *TCBCONTROLD.AE* bit enables addresses to be supplied when any request is serialized. This is not typically required because addresses issued from processor CPUs can be inferred from the CPU PDTrace stream.

The *TCBCONTROLB.TLev* bit controls the amount of information to be included the CM trace. Setting *TLev* to 1 may be useful when debugging performance problems.

The *TCBCONTROL.IO* bit determines the action taken by the CM with its internal trace buffers overflow. If the *IO* bit is 0 then trace information is lost when the trace buffer overflows. In this case, the CM temporarily stops producing trace messages, waits until the trace buffer becomes empty, performs a trace resynchronization with the CPUs and then starts producing new trace words.

However, if *TCBCONTROL.IO* bit is 1 then trace information is never lost, but the system performance may be impacted when the trace buffer becomes full and the additional trace words are required. In this case, the CM stalls the processing of requests and/or L1 intervention responses until a trace buffer becomes available.

The *TCBCONTROL.WB* determines if L1 writebacks are traced or not. L1 writebacks are not software visible and do not appear in the CPU PDTrace, so typically writebacks are not traced in the CM (WB set to 0).

The value in the *TCBCONTROLD.TWSrcVal* field appears in all trace words produced by the CM, thus tagging the trace word as coming from the CM. A unique value must be programmed in this field and *TCBCONTROLB.TWSrcVal* for all cores.

The five *P<port>_Ctl* fields in *TCBCONTROLD* give the ability to control the amount of trace information provided for requests received on the specified port. As shown in Table 14.24, requests from a given CM request port can be traced normally, always traced with addresses, or not traced. Typically, the CM request ports connected to CPUs will be traced normally (P0_Ctl, P1_Ctl, P3_Ctl, P4_Ctl set to 0) because the address is traced by the CPU itself. However, requests from the IOCU are only traced by the CM and therefore should have their addresses traced by the CM (P4_Ctl should be set to 2).

TCBCONTROLE Register

The *TCBCONTROLE* register is used top control tracing functions of the Coherence Manager performance counters. The *TCBCONTROLE* register is written by an EJTAG TAP controller instruction, *TCBCONTROLE* (0x16). This register is also mapped to offset 0x0020 in the Global Debug Block of the CM GCRs. The format of the *TCBCONTROLE* register is shown below, and the fields are described in Table 14.25.

Figure 14.9 TCBCONTROLE Register Format

31 … 9	8	7 … 1	0
0	TdIDLE	Res	PeC

Table 14.25 TCBCONTROLE Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:9	Reserved for future use. Must be written as zero; returns zero on read.	0	0
TrIdle	8	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware. TrIdle is set when the all cores and the CM have disabled PDTrace and the trace funnels has written all outstanding trace information to the off-chip or on-chip memory.	R	1
0	7:1	Reserved for future use; Must be written as zero; returns zero on read. (Hint to architect, Reserved for future expansion of performance counter trace events).	0	0
PeC	0	Performance counter tracing is not implemented.	R	0

TCBCONFIG Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. This register is also mapped to offset 0x0028 in the Global Debug Block of the CM GCRs. The format of the *TCBCONFIG* register is shown below, and the field is described in Table 14.26.

Figure 14.10 TCBCONFIG Register Format

31	30 … 25	24 … 21	20 … 17	16 … 14	13 … 11	10 9	8 … 6	5	4	3 … 0
CF1	0	0	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

Table 14.26 TCBCONFIG Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CF1</td><td>31</td><td>This bit is set if a TCBCONFIG1 register exists. In this revision, TCBCONFIG1 does not exist and this bit always reads zero.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>30:21</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>SZ</td><td>20:17</td><td>On-chip trace memory size. This field holds the encoded size of the on-chip trace memory.
The size in bytes is given by $2^{(SZ+8)}$, implying that the minimum size is 256 bytes and the largest is 8Mb.
This bit is reserved if on-chip memory is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>CRMax</td><td>16:14</td><td>Off-chip Maximum Clock Ratio.
This field indicates the maximum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 14.21.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>CRMin</td><td>13:11</td><td>Off-chip Minimum Clock Ratio.
This field indicates the minimum ratio of the core clock to the off-chip trace memory interface clock.The clock-ratio encoding is shown in Table 14.21.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>PW</td><td>10:9</td><td>Probe Width: Number of bits available on the off-chip trace interface TR_DATA pins. The number of TR_DATA pins is encoded, as shown in the table.
<table>
<tr><th>PW</th><th>Number of bits used on TR_DATA</th></tr>
<tr><td>00</td><td>4 bits</td></tr>
<tr><td>01</td><td>8 bits</td></tr>
<tr><td>10</td><td>16 bits</td></tr>
<tr><td>11</td><td>reserved</td></tr>
</table>
This field is preset based on input signals to the TCB and the actual capability of the TCB.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>PiN</td><td>8:6</td><td>Pipe number.
Indicates the number of execution pipelines.</td><td>R</td><td>0</td></tr>
<tr><td>OnT</td><td>5</td><td>When set, this bit indicates that on-chip trace memory is present. This bit is preset based on the selected option when the TCB is implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>OfT</td><td>4</td><td>When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module (TC_PibPresent asserted).</td><td>R</td><td>Preset</td></tr>
<tr><td>REV</td><td>3:0</td><td>Trace control buffer revision. Refer to the Release Notes for the most current PDTrace revision.</td><td>R</td><td>0x9</td></tr>
</table>

TCBTW Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The TW read is the one pointed to by the *TCBRDP* register. A side effect of reading the *TCBTW* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero.

This register is also mapped to offset 0x0200 (lower 32 bits) and 0x0208 (upper 32 bits) in the Global Debug Block of the CM GCRs.

The format of the *TCBTW* register is shown below, and the field is described in Table 14.27.

Figure 14.11 TCBTW Register Format

63 … 0
Data

Table 14.27 TCBTW Register Field Descriptions

Fields				
Names	Bits	Description	Read / Write	Reset State
Data	63:0	Trace Word	R/W	0

TCBRDP Register (Reg 5)

The *TCBRDP* register is the address pointer to on-chip trace memory. It points to the TW read when reading the *TCBTW* register. When writing the $TCBCONTROLB_{RM}$ bit to 1, this pointer is reset to the current value of *TCBSTP*.

This register is also mapped to offset 0x0108 in the Global Debug Block of the CM GCRs.

The format of the *TCBRDP* register is shown below, and the field is described in Table 14.28. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 14.12 TCBRDP Register Format

31 … n+1	n … 0
	Address

Table 14.28 TCBRDP Register Field Descriptions

Fields				
Names	Bits	Description	Read / Write	Reset State
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

TCBWRP Register (Reg 6)

The *TCBWRP* register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written.

This register is also mapped to offset 0x0110 in the Global Debug Block of the CM GCRs.

The format of the *TCBWRP* register is shown below, and the fields are described in Table 14.29. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

Figure 14.13 TCBWRP Register Format

31 n+1	n 0
	Address

Table 14.29 TCBWRP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

TCBSTP Register (Reg 7)

The *TCBSTP* register is the start pointer register. This pointer is used to determine when all entries in the trace buffer have been filled (when *TCBWRP* has the same value as *TCBSTP*). This pointer is reset to zero when the $TCBCONTROLB_{TR}$ bit is written to 1. If a continuous trace to on-chip memory wraps around the on-chip memory, *TSBSTP* will have the same value as *TCBWRP*.

This register is also mapped to offset 0x0118 in the Global Debug Block of the CM GCRs.

The format of the *TCBSTP* register is shown below, and the fields are described in Table 14.30. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 14.14 TCBSTP Register Format

31 n+1	n 0
	Address

Table 14.30 TCBSTP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

TCBSYS Register (Reg 30)

The *TCBSYS* register contents are driven to the *SI_TC_Sys_UserCtl[31:0]* output signals. This register is also mapped to offset 0x0040 in the Global Debug Block of the CM GCRs. Thus, any change to this register will be reflected in these output signals. The format of the *TCBSYS* register is shown below, and the fields are described in Table 14.31.

Figure 14.15 TCBSYS Register Format

31	30 ... 0
STA	UsrCtl

Table 14.31 TCBSYS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
STA	31	System Trace Available. Set to 1 if the System Trace Interface is present. Otherwise it is set to 0.	R	present: 1 not present: 0
UsrCtl	30:0	User-defined Control.	R/W	0

Register Reset State

Reset state for all register fields is entered when either of the following occur:

1. TAP controller enters/is in Test-Logic-Reset state.

2. *EJ_TRST_N* input is asserted low.

14.3.4 MIPS Trace Capability

There are several build-time options for trace support within the P6600 Multiprocessing System (MPS):

1. No trace logic included.

2. Trace logic to support an on-chip trace memory (embedded within the MPS).

3. Trace logic to support an off-chip trace probe (with off-chip trace memory).

4. Combination of options 2 and 3.

14.3.5 Memory-Mapped Access to PDtrace™ Control and On-Chip Trace RAM

PDtrace can be controlled entirely through software and the on-chip trace memory can be accessed directly by software using load and store instructions.

14.3.6 On-Chip Trace Buffer Usage

In order to direct trace data to the on-chip buffer instead of the off-chip interface, the OfC bit in the TCBControlB register of the trace master must be cleared. Once this is done, the trace funnel will combine trace data it receives from the CM and CPUs and write it to the on-chip memory. Tracing can be enabled or disabled on a per CM/CPU basis by setting or clearing the EN bits in the corresponding TCBControlB registers.

To initialize the on-chip trace buffer, the TR bit of the TCBControlB register of the trace master is set by software. This will initialize TCBRDP, TCBWRP and TCBSTP pointers to zero. These pointers do not have to explicitly written by software for initialization, the reset function that is caused by setting the TR bit is sufficient.

When it is desired to read out the Trace Words from the on-chip buffer, software first sets the RM bit.within TCBControlB. This will load the TCBRDP register with the value held in the TCBSTP register. The TraceWord pointed by TCBRDP can be then read out through the TCBTW register. The read will automatically update the TCBRDP value to point to the next newer entry. A subsequent read from TCBTW register will thus read out the next newer TraceWord. Software does not have to explicitly update the TCBRDP register.

If the TM field of TCBControlB register is set to Trace-From mode, the trace-buffer contents stop being updated when the trace-buffer is full (when TCBWRP points to the same entry as TCBSTP). This event is denoted by the BF bit of TCBControlB register. The BF bit can be polled by software to decide when to read out the trace buffer contents.

For production testing, such as stuck-at testing of memory cells within the trace buffer, the TCBRDP and TCBWRP registers can be explicitly written by software to write and read specific entries within the trace buffer. As previously stated, for normal usage these pointer registers do not have to be explicitly written by software.

Chapter 15

Instruction Latencies and Repeat Rates

This chapter provides the instructions latency and repeat rates for the following instruction types.

15.1 Definition of Terms

The terms *latency* and *repeat rate* are defined as follows:

Latency is defined as the minimum time between 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.

Repeat rate is measured as the minimum issue interval time between independent instructions. For example, a MUL instruction has a latency of 4 cycles and a repeat rate of 1 cycle. Consider the following code sequence:

```
MUL r4, r1, r2
MUH r5, r1, r2
```

The MUL instruction multiplies the r1 and r2 values and places the lower half of the result into r4. The MUH instruction multiples the r1 and r2 values and places the upper half of the result into r5. In this case the MUH can issue one cycle after the MUL instruction issues.

15.2 MTC0 Instruction Considerations

Any MTC0 instruction which can potentially change the operating mode (kernel, supervisor, user) or context (memory mapping) should be executed in the delay slot of a JALR.HB instruction to avoid hazards. Instructions following JALR.HB-MTC0 pair will thus be fetched and executed in the new mode. If the mode-changing MTC0 instruction is not placed in delay slot of JALR.HB instruction, it is not guaranteed that the following instruction will be fetched and executed in the new mode or context.

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.

15.3 Compact Branch Handling

Back-to-back compact branches in static code space are optimized in the P6600 for the following cases:

- BALC followed by any conditional compact branch
- BALC followed by BALC

The rest of the combinations of compact branches on the same cache-line maycause instruction fetch stall and related performance impact.

15.4 Integer Instruction Latencies and Repeat Rates

The following table shows the latency and repeat rates for integer instructions. Note that while the P6600 does have two ALU's, they are not identical. As such, certain instructions can only be executed in either ALU1 or ALU2. The ALU in which the instruction can be executed is shown in the Unit Type column.

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
ADD	Add word	1	1	ALU1/ALU2	2	
ADDIU	Add immediate unsigned word	1	1	ALU1/ALU2	2	
ADDIUPC	Add immediate to PC (unsigned, non-trapping)	2	1	ALU2	1	Y
ADDU	Add unsigned word	1	1	ALU1/ALU2	2	
ALIGN	Concatenate two GPRs, and extract a contiguous subset at a byte position. Operates on 32-bit words with a 2-bit byte position field.	2	1	ALU2	1	Y
ALUIPC	Aligned add upper immediate to PC	2	1	ALU2	1	Y
AND	Bitwise logical AND operation	1	1	ALU1	1	
ANDI	Bitwise logical AND immediate with a constant	1	1	ALU1/ALU2	2	
AUI	Add upper immediate	1	1	ALU2	1	Y
AUIPC	Add upper immediate to PC	2	1	ALU2	1	Y
B	Unconditional branch	n/a	1	CTI	1	
BAL	Branch and link	2	1	CTI	1	
BALC	Branch and link compact	2	1	CTI	1	Y
BC	Branch compact	n/a	1	CTI	1	Y
BC1EQZ	Branch if coprocessor 1 equal to zero	n/a	1	CTI	1	
BC1NEZ	Branch if coprocessor 1 not equal to zero	n/a	1	CTI	1	
BEQ	Branch on equal.	n/a	1	CTI	1	
BEQC	Compact branch if GPR values are equal.	n/a	1	CTI	1	Y
BEQZALC	Compact branch-and-link if GPR rt is equal to zero.	2	1	CTI	1	Y
BEQZC	Compact branch if GPR rs is equal to zero.	n/a	1	CTI	1	Y
BGEC	Compact branch if GPR rs is greater than or equal to GPR rt.	n/a	1	CTI	1	Y
BGEUC	Compact branch if GPR rs is greater than or equal to GPR rt, unsigned.	n/a	1	CTI	1	Y
BGEZ	Branch on greater than or equal to zero.	n/a	1	CTI	1	
BGEZALC	Compact branch-and-link if GPR rt is greater than or equal to zero.	2	1	CTI	1	Y
BGEZC	Compact branch if GPR rt is greater than or equal to zero.	n/a	1	CTI	1	Y
BGTC	Compact branch if GPR rt is greater than GPR rs (alias for BLTC). Assembly idiom with operands reversed.	n/a	1	CTI	1	Y

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
BGTUC	Compact branch if GPR rt is greater than GPR rs, unsigned (alias for BLTUC). Assembly idiom with operands reversed.	n/a	1	CTI	1	Y
BGTZ	Branch on greater than zero.	n/a	1	CTI	1	
BGTZC	Compact branch if GPR rt is greater than zero.	n/a	1	CTI	1	Y
BGTZALC	Compact branch-and-link if GPR rt is greater than zero.	2	1	CTI	1	Y
BITSWAP	Swaps (reverses) bits in each byte. Operates on all 4 bytes of a 32-bit GPR. See DBITSWAP instruction.	2	1	ALU2	1	Y
BLEC	Compact branch if GPR rt is less than or equal to GPR rs (alias for BGEC). Assembly idiom with operands reversed.	n/a	1	CTI	1	Y
BLEUC	Compact branch if GPR rt is less than or equal to GPR rt, unsigned (alias for BGEUC). Assembly idiom with operands reversed.	n/a	1	CTI	1	Y
BLEZ	Branch on less than or equal to zero.	n/a	1	CTI	1	
BLEZALC	Compact branch-and-link if GPR rt is less than or equal to zero.	2	1	CTI	1	Y
BLEZC	Compact branch if GPR rt is less than or equal to zero.	n/a	1	CTI	1	Y
BLTC	Compact branch if GPR rs is less than GPR rt.	n/a	1	CTI	1	Y
BLTUC	Compact branch if GPR rs is less than GPR rt, unsigned.	n/a	1	CTI	1	Y
BLTZ	Branch on less than zero.	n/a	1	CTI	1	
BLTZALC	Compact branch-and-link if GPR rt is less than zero.	2	1	CTI	1	Y
BLTZC	Compact branch if GPR rt is less than zero.	n/a	1	CTI	1	Y
BNE	Branch on not equal.	n/a	1	CTI	1	
BNEC	Compact branch if GPR value are not equal.	n/a	1	CTI	1	Y
BNEZALC	Compact branch-and-link if GPR rt is not equal to zero.	2	1	CTI	1	Y
BNEZC	Compact branch if GPR rs is not equal to zero.	n/a	1	CTI	1	Y
BOVC	Branch on overflow, compact.	n/a	1	CTI	1	Y
BNVC	Branch on no overflow, compact.	n/a	1	CTI	1	Y
BREAK	Breakpoint. To cause a breakpoint exception.	0	n/a	--	--	
CACHE	Perform a cache operation specified by the opcode.	n/a	1	LSU	1	
CFC1	Move control word from floating point	≥4	1	CTI/LSU	1	
CLO	Count number of leading ones in a word.	2	1	ALU2	1	
CLZ	Count number of leading zeros in a word.	2	1	ALU2	1	
CTC1	Move control word to floating point.	5	1	CTI/LSU	1	
DADD	Doubleword add. Add two 64-bit integers. Trap on overflow.	1	1	ALU1/ALU2	2	
DADDIU	Doubleword add immediate unsigned. Add a constant to a 64-bit integer.	1	1	ALU1/ALU2	2	
DADDU	Doubleword add unsigned. Add two 64-bit integers	1	1	ALU1/ALU2	2	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
DAHI	Doubleword add higher immediate	1	1	ALU2	1	Y
DALIGN	Concatenate two GPRs, and extract a contiguous subset at a byte position. Operates on 64-bit doublewords with a 3-bit byte position field.	2	1	ALU2	1	Y
DATI	Doubleword add top immediate	1	1	ALU2	1	Y
DAUI	Doubleword add upper immediate	1	1	ALU2	1	Y
DBITSWAP	Swaps (reverses) bits in each byte. Operates on all 8 bytes of a 64-bit GPR. See BITSWAP instruction.	2	1	ALU2	1	Y
DCLO	Count leading ones in doubleword.	2	1	ALU2	1	
DCLZ	Count leading zeros in doubleword.	2	1	ALU2	1	
DDIV DMOD	Divide 64-bit integers signed. Modulo 64-bit doublewords signed Divide the operands in GPR rs and GPR ft, and place the result into GPR rd. See the DIV instruction.	≥5	≥5	MDU	1	Y
DDIVU DMODU	Divide 64-bit unsigned integers. Modulo doublewords unsigned Divide the unsigned 64-bit operands in GPR rs and GPR rt, and place the result into GPR rd. See the DIVU instruction.	≥5	≥5	MDU	1	Y
DERET	Return from debug exception.	0	n/a	CTI	1	
DEXT	Doubleword extract bit field.	2	1	ALU2	1	
DEXTM	Doubleword extract bit field middle.	2	1	ALU2	1	
DEXTU	Doubleword extract bit field upper.	2	1	ALU2	1	
DI	Disable interrupts. Return the previous value of the CP0 Status register and disable interrupts.	0	n/a	--	--	
DINS	Doubleword insert bit field. Merge a right-justified bit field from the GPR rs field into the specified GPR rt field.	2	1	ALU2	1	
DINSM	Doubleword insert bit field middle.	2	1	ALU2	1	
DINSU	Doubleword insert bit field upper.	2	1	ALU2	1	
DIV MOD	Divide 32-bit integers signed. Modulo words signed Divide the operands in GPR rs and GPR ft, and place the result into GPR rd.	≥5	≥5	MDU	1	Y
DIVU MODU	Divide 32-bit unsigned integers. Modulo words unsigned Divide the unsigned 32-bit operands in GPR rs and GPR rt, and place the result into GPR rd. See the DDIVU instruction.	≥5	≥5	MDU	1	Y
DLSA	Doubleword load scaled address. Add two values from registers rs and rt. See LSA instruction.	2	1	ALU2	1	Y
DMFC0	Doubleword move from CP0 to GPR.	4	1	LSU	1	
DMFC1	Doubleword move from FPR to GPR.	4	1	LSU	1	
DMTC0	Doubleword move from GPR to CP0.	n/a	1	LSU	1	
DMTC1	Doubleword move from GPR to FPR.	n/a	1	LSU	1	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
DMUH	Multiply doublewords signed, high doubleword. Performs a signed 64-bit integer multiplication and places the high 64 bits of the result in the destination register.	4	1	MDU	1	Y
DMUL	Multiply doublewords signed, low doubleword. Perfoms a signed 64-bit integer multiplication and places the low 64 bits of the result in the destination register.	4	1	MDU	1	Y
DMUHU	Multiply doublewords unsigned, high doubleword. Performs an unsigned 64-bit integer multiplication and places the high 64 bits of the result in the destination register.	4	1	MDU	1	Y
DMULU	Multiply doublewords unsigned, low doubleword. Performs an unsigned 64-bit integer multiplication and places the low 64 bits of the result in the destination register.	4	1	MDU	1	Y
DROTR	Doubleword rotate right. Logical rotate right of a doubleword by a fixed amount — 0 - 31 bits.	1	1	ALU1/ALU2	2	
DROTR32	Doubleword rotate right plus 32. Logical rotate right of a doubleword by a fixed amount — 32 - 63 bits.	1	1	ALU1/ALU2	2	
DROTRV	Doubleword rotate right variable. Logical rotate right of a doubleword by a variable number of bits.	1	1	ALU1/ALU2	2	
DSBH	Doubleword swap bytes within halfwords. Swap the bytes within each halfword of GPR rt and store into GPR rd.	2	1	ALU2	1	
DSHD	Doubleword swap halfwords within doublewords. Swap the halfwords within each doubleword of GPR rt and store into GPR rd.	2	1	ALU2	1	
DSLL	Doubleword shift left logical. Logical left-shift of a doubleword by a fixed amount — 0 - 31 bits.	1	1	ALU1/ALU2	2	
DSLL32	Doubleword shift left logical plus 32. Logical left-shift of a doubleword by a fixed amount — 32 - 63 bits.	1	1	ALU1/ALU2	2	
DSLLV	Doubleword shift left logical variable. Logical left-shift of a doubleword by a variable number of bits.	1	1	ALU1/ALU2	2	
DSRA	Doubleword right shift arithmetic. Arithmetic right-shift of a doubleword by a fixed amount — 0 - 31 bits.	1	1	ALU1/ALU2	2	
DSRA32	Doubleword right shift arithmetic plus 32. Arithmetic right-shift of a doubleword by a fixed amount — 32 - 63 bits.	1	1	ALU1/ALU2	2	
DSRAV	Doubleword shift right arithmetic variable. Arithmetic right-shift of a doubleword by a variable number of bits.	1	1	ALU1/ALU2	2	
DSRL	Doubleword shift right logical. Logical right-shift of a doubleword by a fixed amount — 0 - 31 bits.	1	1	ALU1/ALU2	2	
DSRL32	Doubleword shift right logical plus 32. Logical right-shift of a doubleword by a fixed amount — 32 - 63 bits.	1	1	ALU1/ALU2	2	
DSRLV	Doubleword shift right logical variable. Logical right-shift of a doubleword by a variable number of bits.	1	1	ALU1/ALU2	2	
DSUB	Doubleword subtract. Subtract 64-bit integers. Trap on overflow.	1	1	ALU1/ALU2	2	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
DSUBU	Doubleword subtract unsigned. Subtract unsigned 64-bit integers. Trap on overflow.	1	1	ALU1/ALU2	2	
DVP	Disable virtual processor. Disable all virtual processors in a core except the one that issued the instruction.	0	n/a	--	--	Y
EHB	Execute hazard barrier. Stop instruction execute until all execution hazards have been cleared.	0	n/a	--	--	
EI	Enable interrupts. Return the previous state of the CP0 Status register and enable interrupts.	0	n/a	--	--	
ERET	Exception return. Return from interrupt, exception, or error trap.	4	n/a	LSU	1	
ERETNC	Exception return no clear. Return from interrupt, exception, or error trap without clearing the LL bit.	4	n/a	LSU	1	
EVP	Enable virtual processor. Enable all virtual processors in a core except the one that issued the instruction.	0	n/a	--	--	Y
EXT	Extract bit field. Extract a bit field from GPR rx and store it right-justified into GPT rt.	2	1	ALU2	1	
INS	Insert bit field. Merge a right-justified bit field from GPR rs into a specified field in GPR rt.	2	1	ALU2	1	
J	Jump. Branch within the current 256 MByte region.	n/a	1	CTI	1	
JAL	Jump and link. Execute a procedure call within the current 256 MByte region.	2	1	CTI	1	
JALR	Jump and link register. Execute a procedure call to an instruction address in a register.	2	1	CTI	1	
JALR.HB	Jump and link register with hazard barrier. Execute a procedure call to an instruction address in a register and clear all execution and instruction hazards.	n/a	1	CTI	1	
JIALC	Jump indexed and link, compact. The jump target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR rt.	n/a	1	CTI	1	Y
JIC	Jump indexed, compact. The branch target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR rt.	n/a	1	CTI	1	Y
JR	Jump register. Execute a branch to an instruction address in a register.	n/a	1	CTI	1	
JR.HB	Jump register with hazard barrier. Execute a a branch to an instruction address in a register and clear all execution and instruction hazards.	n/a	n/a	CTI	1	
LB	Load byte from memory as a signed value.	≥4	1	LSU	1	
LBU	Load byte from memory as an unsigned value.	≥4	1	LSU	1	
LD	Load doubleword from memory.	≥4	1	LSU	1	
LDC1	Load doubleword from memory to an FPR.	≥10	1	LSU	1	
LDPC	Load doubleword PC-relative. Load a doubleword from memory using a PC-relative address.	≥4	1	LSU	1	Y

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
LH	Load halfword from memory as a signed value.	≥4	1	LSU	1	
LHU	Load halfword from memory as an unsigned value.	≥4	1	LSU	1	
LL	Load linked word. Load a word from memory for an atomic read-modify-write.	≥4	1	LSU	1	
LLD	Load linked doubleword. Load a doubleword from memory for an atomic read-modify-write.	≥4	1	LSU	1	
LSA	Load scaled address. Add two values from registers rs and rt. See DLSA instruction.	2	1	ALU2	1	Y
LUI	Load upper immediate. Load a constant into the upper half of a word.	1	1	ALU1	1	
LW	Load word from memory as a signed value.	≥4	1	LSU	1	
LWC1	Load word from memory to an FPR.	≥10	1	LSU	1	
LWPC	Load word PC relative. Load a word from memory as a signed value using a PC-relative address.	≥4	1	LSU	1	Y
LWU	Load word from memory as an unsigned value.	≥4	1	LSU	1	
LWUPC	Load word unsigned PC relative. Load a word from memory as an unsigned value using a PC-relative address.	≥4	1	LSU	1	Y
MFC0	Move from CP0. Move the contents of a CP0 register to a general register.	4	1	LSU	1	
MFC1	Copy a word from an FPR to a GPR.	7	1	LSU	1	
MFHC0	Move from high CP0. Move the contents of the upper 32 bits of a CP0 register, extended by 32-bits, to a general register.	4	1	LSU	1	
MFHC1	Copy word from high half of an FPR to a GPR.	7	1	LSU	1	
MTCO	Move to CP0. Move the contents of the upper 32 bits of a general register to a CP0 register.	n/a	1	LSU	1	
MTC1	Move word from a GPR to an FPR.	n/a	1	LSU	1	
MTHC0	Move to high CP0. Move the contents of the upper 32 bits of a CP0 register, extended by 32-bits, to a general register.	5	1	LSU	1	
MTHC1	Copy word from a GPR to the high half of an FPR.	5	1	LSU	1	
MUH	Multiply words signed, high word. Performs a signed 32-bit integer multiplication and places the high 32 bits of the result in the destination register.	3	1	MDU	1	Y
MUHU	Multiply words unsigned, high word. Performs an unsigned 32-bit integer multiplication and places the high 32 bits of the result in the destination register.	3	1	MDU	1	Y
MUL	Multiply words signed, low word. Performs a signed 32-bit integer multiplication and places the low 32 bits of the result in the destination register.	3	1	MDU	1	Y
MULU	Multiply words unsigned, low word. Performs an unsigned 32-bit integer multiplication and places the low 32 bits of the result in the destination register.	3	1	MDU	1	Y
NAL	No-op and link. Used to read the PC.	2	1	CTI	1	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
NOP	No operation.	0	n/a	ALU1/ALU2	2	
NOR	NOT OR. Bitwise logical NOT OR.	1	1	ALU1	1	
OR	OR operation. Bitwise logical OR.	1	1	ALU1	1	
ORI	OR immediate. Bitwise logical or with a constant.	1	1	ALU1/ALU2	2	
PAUSE	Pause. Wait for the LL bit to clear.	n/a	1	LSU	1	
PREF	Prefetch. Move data between memory and cache.	≥4	1	LSU	1	
RDHWR	Read hardware register. Move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.	4	1	LSU	1	
RDPGPR	Read GPR from previous shadow set. Move the contents of a GPR from the previous shadow set to a current GPR.	1	1	ALU	1	
ROTR	Rotate word right. Logical right-rotate of a word by a fixed number of bits.	1	1	ALU1/ALU2	2	
ROTRV	Rotate word right variable. Logical right-rotate of a word by a variable number of bits.	1	1	ALU1/ALU2	2	
SB	Store byte. Store a byte to memory.	n/a	1	LSU	1	
SC	Store conditional word. Store a word to memory to complete an atomic read-modify-write.	≥4	1	LSU	1	
SCD	Store conditional doubleword. Store a doubleword to memory to complete an atomic read-modify-write.	≥4	1	LSU	1	
SD	Store a doubleword to memory.	n/a	1	LSU	1	
SDBBP	Software debug break point. Cause a debug breakpoint exception.	0	n/a	--	--	
SDC1	Store doubleword from FPR to memory	4	1	LSU	1	
SEB	Sign-extend byte. Sign-extend the least significant byte of GPR rt and store the value into GPR rd.	1	1	ALU1/ALU2	2	
SEH	Sign-extend halfword. Sign-extend the least significant halfword of GPR rt and store the value into GPR rd.	1	1	ALU1/ALU2	2	
SELEQZ	Select integer GPR value or zero. Condition true only if all bits in GPR rt are zero.	2	1	ALU2	1	Y
SELNEZ	Select integer GPR value or non-zero. Condition true only if any bit in GPR rt is non-zero.	2	1	ALU2	1	Y
SH	Store halfword to memory.	n/a	1	LSU	1	
SIGRIE	Signal reserved instruction exception.	n/a	n/a	--	--	Y
SLL	Shift word left logical by a fixed number of bits.	1	1	ALU1/ALU2	2	
SLLV	Shift word left logical by a variable number of bits.	1	1	ALU1/ALU2	2	
SLT	Set on less than. Record the result of a less-than comparison.	1	1	ALU1/ALU2	2	
SLTI	Set on less than immediate. Record the result of a less-than comparison with a constant.	1	1	ALU1	1	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
SLTIU	Set on less than immediate unsigned. Record the result of an unsigned less-than comparison with a constant.	1	1	ALU1	1	
SLTU	Set on less than unsigned. Record the result of a less-than comparison.	1	1	ALU1/ALU2	2	
SRA	Shift word right arithmetic. Execute an arithmetic right-shift of a word by a fixed number of bits.	1	1	ALU1/ALU2	2	
SRAV	Shift word right arithmetic variable. Execute an arithmetic right-shift of a word by a variable number of bits.	1	1	ALU1/ALU2	2	
SRL	Shift word right logical. Execute a logical right-shift of a word by a fixed number of bits.	1	1	ALU1/ALU2	2	
SRLV	Shift word right logical variable. Execute a logical right-shift of a word by a variable number of bits.	1	1	ALU1/ALU2	2	
SSNOP	Superscalar no-operation. Break superscalar issue.	0	n/a	--	--	
SUB	Subtract 32-bit integers. Trap on overflow.	1	1	ALU1	1	
SUBU	Subtract unsigned 32-bit integers.	1	1	ALU1/ALU2	2	
SW	Store word to memory.	n/a	1	LSU	1	
SWC1	Store word from FPR to memory	7	1	LSU	1	
SYNC	Synchronize shared memory. Order loads and stores for shared memory.	n/a	1	LSU	1	
SYNCI	Synchronize caches to make instruction writes effective.	n/a	1	LSU	1	
SYSCALL	System call. Cause a system call exception.	n/a	n/a	--	--	
TEQ	Trap if equal. Compare GPR's and do a conditional trap if equal.	n/a	1	ALU2	1	
TGE	Trap if greater or equal. Compare GPR's and do a conditional trap on greater or equal condition.	n/a	1	ALU2	1	
TGEU	Trap if greater or equal unsigned.	n/a	1	ALU2	1	
TLBINV	TLB invalidate. Invalidates TLB entry based on ASID and index match.	n/a	1	LSU	1	
TLBINVF	TLB invalidate flush.	n/a	1	LSU	1	
TLBP	TLB probe. Find a matching TLB entry.	n/a	1	LSU	1	
TLBR	TLB read. Read an entry from the TLB.	n/a	1	LSU	1	
TLBWI	TLB write indexed. Write or invalidate a TLB entry indexed by the CP0 Index register.	n/a	1	LSU	1	
TLBWR	TLB write random. Write a TLB entry indexed by an implementation-defined location.	n/a	1	LSU	1	
TLT	Trap if less than. Compare GPR's and trap on condition.	n/a	1	ALU2	1	
TLTU	Trap if less than unsigned. Compare GPR's and trap on condition.	n/a	1	ALU2	1	
TNE	Trap if not equal. Compare GPR's and trap on condition.	n/a	1	ALU2	1	
WAIT	Wait for event. Enter standby mode.	0	n/a	--	--	

Table 15.1 P6600 Integer Instructions — Latency and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate	Unit Type	Number of Units	New in R6
WRPGPR	Write to GPR in previous shadow set. Move the contents of a current GPR to a GPR in the previous shadow set.	2	1	ALU2	1	
WSBH	Word swap bytes within halfwords. Swap the bytes within each halfword of GPR rt and store the value into GPR rd.	2	1	ALU2	1	
XOR	Exclusive OR.	1	1	ALU1	1	
XORI	Exclusive OR immediate.	1	1	ALU1/ALU2	2	

15.5 Floating Point Instruction Latencies and Repeat Rates

The following table shows the latencies and repeat rates for the floating point unit (FPU) instructions.

Table 15.2 Floating Point Latencies and Repeat Rates

Instruction	Definition	Latency	Repeat Rate
ABS.fmt	Floating point absolute value	2	1
ADD.fmt	Floating point add	4	1
CEIL.L.fmt	Fixed point ceiling convert to long fixed point	4	1
CEIL.W.fmt	Fixed point ceiling convert to word fixed point	4	1
CLASS.fmt	Scalar floating point class mask	2	1
CMP.cond.fmt	Fixed point compare setting mask	2	1
CVT.D.fmt	Fixed point convert to double floating point	4	1
CVT.L.fmt	Fixed point convert to long fixed point	4	1
CVT.S.fmt	Fixed point convert to single floating point	4	1
CVT.W.fmt	Fixed point convert to word fixed point	4	1
DIV.fmt	Floating point divide	variable	variable
FLOOR.L.fmt	Fixed point floor convert to long fixed point	4	1
FLOOR.W.fmt	Fixed point floor convert to word fixed point	4	1
MADDF.fmt	Floating point fused multiply add[1]	4, 8	1
MAX.fmt	Scalar floating point maximum value	2	1
MAXA.fmt	Scalar floating point maximum value with input arguments	2	1
MIN.fmt	Scalar floating point minimum value	2	1
MINA.fmt	Scalar floating point minimum value with input arguments	2	1
MSUBF.fmt	Floating point fused multiply subtract[1]	4, 8	1
MUL.fmt	Floating point multiply	5	1
NEG.fmt	Floating point negate	2	1
RECIP.fmt	Floating point reciprocal	variable	variable
RINT.fmt	Scalar floating point round to integral floating point value	4	1
ROUND.L.fmt	Floating point round to long fixed point	4	1
ROUND.W.fmt	Floating point round to word fixed point	4	1
RSQRT.fmt	Floating point reciprocal square root	variable	variable
SEL.fmt	Select floating point values with	2	1
SELEQZ.fmt	Select floating point with conditions equal to zero	2	1
SELNEZ.fmt	Select floating point with conditions not equal to zero	2	1
SQRT.fmt	Floating point square root	variable	variable
SUB.fmt	Floating point subtract	4	1
TRUNC.L.fmt	Floating point truncate to long fixed point	4	1

Table 15.2 Floating Point Latencies and Repeat Rates

Instruction	Definition	Latency	Repeat Rate
TRUNC.W.fmt	Floating point truncate to word fixed point	4	1

1. 4 = latency to another MADDF add or subtract operand, 8 = latency to another MADDF multiply operand.

15.6 MSA Instruction Latencies and Repeat Rates

The following table shows the latency and repeat rates for the MIPS SIMD Architecture (MSA) instructions.

Table 15.3 MSA Instruction Latencies and Repeat Rates

Instruction	Definition	Latency	Repeat Rate
ADD_A.df	Vector add absolute values	2	1
ADDS_A.df	Vector saturated add of absolute values	2	1
ADDS_S.df	Vector saturated add of signed values	2	1
ADDS_U.df	Vector saturated add of unsigned values	2	1
ADDV.df	Vector add	2	1
ADDVI.df	Immediate add	2	1
AND.V	Vector and	2	1
ANDI.B	Immediate and	2	1
ASUB_S.df	Vector absolute values of signed subtract	2	1
ASUB_U.df	Vector absolute values of unsigned subtract	2	1
AVE_S.df	Vector signed average	2	1
AVE_U.df	Vector unsigned average	2	1
AVER_S.df	Vector signed average rounded	2	1
AVER_U.df	Vector unsigned average rounded	2	1
BCLR.df	Vector bit clear	2	1
BCLRI.df	Immediate bit clear	2	1
BINSRI.df	Immediate bit insert right	2	1
BINSL.df	Vector bit insert left	2	1
BINSLI.df	Immediate bit insert left	2	1
BINSR.df	Vector bit insert right	2	1
BMNZ.V	Vector move if not zero	2	1
BMNZI.B	Immediate move if not zero	2	1
BMZ.V	Vector move if zero	2	1
BMZI.B	Immediate move if zero	2	1
BNZ.df	Branch if all elements are non zero	2	1
BNZ.V	Branch if any element non zero	2	1
BNEG.df	Vector selected bit position negate	2	1
BNEGI.df	Immediate bit negate	2	1
BSEL.V	Vector bit select	2	1
BSELI.B	Immediate bit select	2	1
BSET.df	Vector bit set	2	1
BSETI.df	Immediate bit set	2	1
BZ.df	Branch if any element zero	2	1
BZ.V	Branch if all elements zero	2	1

Table 15.3 MSA Instruction Latencies and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate
CEQ.df	Vector compare equal	2	1
CEQI.df	Immediate compare equal	2	1
CFCMSA	GPR copy from MSA control register	n/a	1
CTCMSA	GPR copy to MSA control register	n/a	1
CLE_S.df	Vector compare signed less than or equal	2	1
CLE_U.df	Vector compare unsigned less than or equal	2	1
CLEI_S.df	Immediate compare signed less than or equal	2	1
CLEI_U.df	Immediate compare unsigned less than or equal	2	1
CLT_S.df	Vector compare signed less than	2	1
CLT_U.df	Vector compare unsigned less than	2	1
CLTI_S.df	Immediate compare signed less than or equal	2	1
CLTI_U.df	Immediate compare unsigned less than or equal	2	1
COPY_S.df	Element move to GPR signed	n/a	1
COPY_U.df	Element move to GPR unsigned	n/a	1
DIV_S.df	Vector signed divide. See MOD_S instruction	variable	variable
DIV_U.df	Vector unsigned divide. See MOD_U instruction	variable	variable
DOTP_S.df	Vector signed dot product	5	1
DOTP_U.df	Vector unsigned dot product	5	1
DPADD_S.df	Vector signed dot product and add	5	1
DPADD_U.df	Vector unsigned dot product and add	5	1
DPSUB_S.df	Vector signed dot product and subtract	5	1
DPSUB_U.df	Vector unsigned dot product and subtract	5	1
FADD.df	Vector FP add	4	1
FCAF.df	Vector FP compare always false	2	1
FCEQ.df	Vector FP compare equal	2	1
FCLASS.df	Vector FP class mask, record class (0, inf, qNaN, etc) of data	2	1
FCLE.df	Vector FP compare less than equal	2	1
FCLT.df	Vector FP compare less than	2	1
FCOR.df	Vector FP compare not equal	2	1
FCNE.df	Vector FP compare not equal	2	1
FCUEQ.df	Vector FP compare not equal	2	1
FCULE.df	Vector FP compare greater than	2	1
FCULT.df	Vector FP compare greater than equal	2	1
FCUN.df	Vector FP compare unordered	2	1
FCUNE.df	Vector FP compare not equal	2	1
FDIV.df	Vector FP divide	variable	variable
FEXDO.df	Vector FP down convert	4	1

Table 15.3 MSA Instruction Latencies and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate
FEXP2.df	Vector FP base 2 exponentiation ws is float df, wt is integer of size df	5	1
FEXUPL.df	Vector FP up convert left	4	1
FEXUPR.df	Vector FP up convert right	4	1
FFINT_S.df	Vector FP convert from signed integer	4	1
FFINT_U.df	Vector FP convert from unsigned integer	4	1
FFQL.df	Vector FP convert from fixed point left	4	1
FFQR.df	Vector FP convert from fixed point right	4	1
FILL.df	Replicate and move from GPR	2	1
FLOG2.df	Vector FP base 2 exponentiation ws is float df, wt is integer of size df	2	1
FMADD.df	Vector multiply add	4, 8	1
FMSUB.df	Vector multiply subtract	4, 8	1
FMAX.df	Vector FP maximum	2	1
FMAX_A.df	Vector FP maximum on absolute value	2	1
FMIN.df	Vector FP minimum	2	1
FMIN_A.df	Vector FP minimum on absolute value	2	1
FMUL.df	Vector FP multiply	5	1
FRCP.df	Vector FP reciprocal	variable	variable
FRINT.df	Vector FP round to integer value but retain float format	4	1
FRSQRT.df	Vector FP reciprocal-square root	variable	variable
FSAF.df	Vector FP signaling equal	2	1
FSEQ.df	Vector FP signaling equal	2	1
FSLE.df	Vector FP signaling less than equal	2	1
FSLT.df	Vector FP signaling less than	2	1
FSNE.df	Vector FP signaling not equal	2	1
FSOR.df	Vector FP compare not equal	2	1
FSQRT.df	Vector FP square root	variable	variable
FSUB.df	Vector FP sub	4	1
FSUEQ.df	Vector FP signaling not equal	2	1
FSULE.df	Vector FP signaling greater than	2	1
FSULT.df	Vector FP signaling greater than equal	2	1
FSUN.df	Vector FP signaling equal	2	1
FSUNE.df	Vector FP compare not equal	2	1
FTINT_S.df	Vector FP convert to signed integer	4	1
FTINT_U.df	Vector FP convert to unsigned integer	4	1
FTQ.df	Vector FP to fixed point	4	1
FTRUNC_S.df	Vector FP convert to signed integer	4	1

Table 15.3 MSA Instruction Latencies and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate
FTRUNC_U.df	Vector FP convert to unsigned integer	4	1
HADD_S.df	Vector signed horizontal add	2	1
HADD_U.df	Vector unsigned horizontal add	2	1
HSUB_S.df	Vector signed horizontal sub	2	1
HSUB_U.df	Vector unsigned horizontal sub	2	1
ILVEV.df	Vector interleave even	2	1
ILVL.df	Vector interleave left	2	1
ILVOD.df	Vector interleave odd	2	1
ILVR.df	Vector interleave right	2	1
INSERT.df	Move from GPR	n/a	1
INSVE.df	Move from element	2	1
LD.df	Vector load	≥10	1
LDI.df	Immediate load elements	2	1
MADD_Q.df	Vector fixed point madd	5	1
MADDR_Q.df	Vector fixed point multiply rounded and add	5	1
MADDV.df	Vector multiply add	5	1
MAX_A.df	Vector maximum of absolute value	2	1
MAX_S.df	Vector signed maximum	2	1
MAX_U.df	Vector unsigned maximum	2	1
MAXI_S.df	Immediate signed maximum	2	1
MAXI_U.df	Intermediate signed maximum	2	1
MIN_A.df	Vector min of absolute value	2	1
MIN_S.df	Vector signed minimum	2	1
MIN_U.df	Vector unsigned minimum	2	1
MINI_S.df	Immediate signed minimum	2	1
MINI_U.df	Immediate unsigned minimum	2	1
MOD_S.df	Vector signed remainder. See DIV_S instruction.	variable	variable
MOD_U.df	Vector unsigned remainder. See DIV_U instruction.	variable	variable
MOVE.V	Vector move	2	1
MSUB_Q.df	Vector fixed point msub	5	1
MSUBR_Q.df	Vector fixed point multiply rounded and subtracted	5	1
MSUBV.df	Vector multiply subtract	5	1
MUL_Q.df	Vector fixed point multiply	5	1
MULR_Q.df	Vector fixed point multiply rounded	5	1
MULV.df	Vector multiply	5	1
NLOC.df	Vector number of leading ones counted	2	1
NLZC.df	Vector number of leading zeros counted	2	1

Table 15.3 MSA Instruction Latencies and Repeat Rates *(continued)*

Instruction	Definition	Latency	Repeat Rate
NOR.V	Vector NOR	2	1
NORI.B	Immediate NOR	2	1
OR.V	Vector OR	2	1
ORI.B	Immediate OR	2	1
PCKEV.df	Vector pack even	2	1
PCKOD.df	Vector pack odd	2	1
PCNT.df	Vector number of bits set	3	1
SAT_S	Immediate signed saturate to width	3	1
SAT_U	Immediate unsigned saturate to width	3	1
SHF.df	Immediate set shuffle	2	1
SLD.df	Element slide	2	1
SLDI.df	Element slide	2	1
SLL.df	Vector shift left	2	1
SLLI.df	Immediate shift left	2	1
SPLAT.df	Element replicate	2	1
SPLATI.df	Element replicate	2	1
SRA.df	Vector shift right arithmetic	2	1
SRAI.df	Immediate shift right arithmetic	2	1
SRAR.df	Vector shift right arithmetic rounded	2	1
SRARI.df	Immediate shift right arithmetic rounded	2	1
SRL.df	Vector shift right logical	2	1
SRLI.df	Immediate shift right logical	2	1
SRLR.df	Vector shift right logical rounded	2	1
SRLRI.df	Immediate shift right logical rounded	2	1
ST.df	Vector store	≥3	1
SUBS_S.df	Vector signed saturated subtract of signed values	2	1
SUBS_U.df	Vector unsigned saturated subtract of unsigned values	2	1
SUBSUS_U.df	Vector unsigned saturated subtract of signed values	2	1
SUBSUU_S.df	Vector signed saturated subtract of unsigned values	2	1
SUBV.df	Vector subtract	2	1
SUBVI.df	Immediate signed saturated subtract of unsigned values	2	1
VSHF.df	Vector shuffle	2	1
XOR.V	Vector XOR	2	1
XORI.B	Immediate XOR	2	1

Chapter 16

Implementation-specific Instructions

This chapter describes the architectural definition for the following implementation-specific instructions in the P6600 Multiprocessing System.

- CACHE: Cache Operation
- PREF: Prefetch
- SYNC: Synchronize Shared Memory

For the actual instruction definition and opcode information, refer to *Volume II: MIPS64 Architecture for Programmer's Manual* included in the document suite. The following table lists the elements of each instruction and how they are specifically handled by the P6600 core.

Table 16.1 Implementation Specific Instruction Behavior in the P6600 Core

Instruction	Parameter	Function
CACHE	*op* field, bits 17:16	Encoding 2'b10. No support for tertiary cache. If this encoding appears in the op field in bits 17:16, it is ignored by the core.
	op field, bits 20:18	Encoding 3'b011. This implementation specific encoding is not implemented by the P6600 core and is treated as a no-operation (NOP).
		Encoding 3'b111. Fetch and Lock. Depends on the type of cache being accessed: *L1 instruction*: This encoding is not supported and is treated as a no-operation (NOP). *L1 data:* This encoding is not supported and is treated as a no-operation (NOP). *L2 cache*: This encoding is supported and is sent to the Coherency Manager (CM3) when the encoding appears. *L3 cache:* This encoding is ignored and is treated as a no-operation (NOP).
		Encoding 3'b110. Data Cache Hit Writeback: Encoding 3'b101. Data Cache Hit Writeback Invalidate: *L1 data*: HitWB or HitWBInv cache operations write back the cache line, irrespective of the state of the lock bit. Software should not rely on the state of the lock bit after the cache operation.

Table 16.1 Implementation Specific Instruction Behavior in the P6600 Core *(continued)*

Instruction	Parameter	Function
PREF	*hint* field, bits 20:16	Encoding 5’b00000 - 5’b00111 (0 - 7 decimal). These hint field encodings are treated as described in the PREF instruction in *Volume II: MIPS64 Architeccture for Programmer’s*.
		Encoding 5’b01000 - 5’b01111 (8 - 15 decimal). These hint field encodings perform the same function as hints 0 - 7 respectively, but operate on the L2 cache. As such, they are sent to the CM3 by the core. Refer to *Volume II: MIPS64 Architecture for Programmer’s* manual for the definition of hints 0 - 7 decimal.
		Encoding 5’b10000 - 5’b10111 (16 - 23 decimal). These hint field encodings are not supported by the P6600 core as the L3 cache is not supported. Each of these encoded values are ignored and treated as a no-operation (NOP).
		Encoding 5’b11000 - 5’b11111 (24 - 32 decimal). These hint field encodings are not supported by the P6600 core and cause a Reserved Instruction exception to be taken.
SYNC	*stype* field, bits 10:6	Encoding 0x0, 0x4, and 0x10 - 0x13. The P6600 core supports the standard mandatory SYNC encoding (0x0), as well as all of the optional SYNC encodings: 0x4 - SYNC_WMB/SYNC4 0x10 - SYNC_MB/SYNC16 0x11 - SYNC_ACQUIRE/SYNC17 0x12 - SYNC_RELEASE/SYNC18 0x13 - SYNC-RMB/SYNC19 For more information on these encodings, refer to Table 6.6 in the SYNC instruction definition in *Volume II: MIPS64 Architecture for Programmer’s*.
		Encoding 0x1 - 0x3, and 0x5 - 0xF. These implementation specific encoded values are not supported by the P6600 core and default to encoding 0x0 (SYNC).
		Encoding 0x14 - 0x1F. These encoded values are reserved by the P6600 core and default to encoding 0x0 (SYNC).

Appendix A

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
01.00	October 20, 2015	Initial release of P6600 SUM.
01.01	January 14, 2016	Update CP0 HWENa register.
01.10	July 14, 2016	Update PDTrace material throughout document.
01.20	July 22, 2016	Update document from internal review. Update integer latency and repeat rates table in Chapter 15.
01.21	August 26, 2016	Update document from internal review. Delete material on register controlled power management. Minor updates to MSA and Integer latency and repeat rate tables in Chapter 15.
01.22	November 1, 2016	Update latency and repeat rates in Chapter 15. Update CP0 chapter. Replace implementation specific instructions with table of differences in Chapter 16. Added Section 15.3 in Latency and Repeat Rates chapter 15.