MIPS32® Architecture for Programmers Volume IV-i: Virtualization Module of the MIPS32® Architecture

Document Number: MD00846
Revision 1.03
April 29, 2013

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Template: nB1.02, Built with tags: 2B ARCH IMPL MIPS32 MIPS32andIMPL

Table of Contents

List of Figures

List of Tables

Chapter 1

About This Book

The MIPS32® Architecture for Programmers Volume IV-i: Virtualization Module of the MIPS32® Architecture comes as part of a multi-volume set.

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS32® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS32™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS32™ instruction set
- Volume III describes the MIPS32® and microMIPS32™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. It is not applicable to the MIPS32® document set nor the microMIPS32™ document set. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture .
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture

1.1 Typographical Conventions

This section describes the use of *italic*, **bold** and `courier` fonts in this book.

1.1.1 Italic Text

- is used for *emphasis*
- is used for *bits*, *fields*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*
- is used for the memory access types, such as *cached* and *uncached*

1.1.2 Bold Text

- represents a term that is being **defined**
- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1
- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

`Courier` fixed-width font is used for text that is displayed on the screen, and for examples of code and instruction pseudocode.

1.2 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this book to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

1.2.1 UNPREDICTABLE

UNPREDICTABLE results may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause a result to be generated or not. If a result is generated, it is **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

- **UNPREDICTABLE** operations must not halt or hang the processor

1.2.2 UNDEFINED

UNDEFINED operations or behavior may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. **UNDEFINED** operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. **UNDEFINED** operations or behavior may cause data loss.

UNDEFINED operations or behavior has one implementation restriction:

- **UNDEFINED** operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state

1.2.3 UNSTABLE

UNSTABLE results or values may vary as a function of time on the same implementation or instruction. Unlike **UNPREDICTABLE** values, software may depend on the fact that a sampling of an **UNSTABLE** value results in a legal transient value that was correct at some point in time prior to the sampling.

UNSTABLE values have one implementation restriction:

- Implementations of operations generating **UNSTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
$\leftarrow$	Assignment
$=, \neq$	Tests for equality and inequality
$\|\|$	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
$+, -$	2's complement or floating point arithmetic: addition, subtraction

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
*, ×	2's complement or floating point multiplication (both used for either)
div	2's complement integer division
mod	2's complement modulo
/	Floating point division
<	2's complement less-than comparison
>	2's complement greater-than comparison
≤	2's complement less-than or equal comparison
≥	2's complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
not	Bitwise inversion
&&	Logical (non-Bitwise) AND
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[SRSCtl*$_{CSS}$*, x]*.
SGPR[s,x]	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR$_{RE}$ and User mode).

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 32-bit address all of which are significant during a memory reference.
ISA Mode	In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning**: 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e or microMIPS instructions In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and MIPSr3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 Release 1 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call.

1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL: http://www.mips.com

For comments or questions on the MIPS32® Architecture or this document, send Email to support@mips.com.

Chapter 2

The Virtualization Module of the MIPS32® Architecture

2.1 Base Architecture Requirements

The Virtualization Application-Specific Extension (Module) requires the following base architecture support:

- **The MIPS32 Architecture**: The Virtualization Module requires a compliant implementation of the MIPS32 Architecture, Release 5.00 or later.
- A TLB-based MMU is required.
- Coprocessor 0 registers *KScratch1* and *KScratch2* are required

2.2 Software Detection of the Module

Software can determine if the Virtualization Module is implemented by checking the state of the VZ bit in the *Config3* CP0 register.

2.3 Compliance and Subsetting

The Virtualization Module to the MIPS32 Architecture provides hardware support for software-controlled platform virtualization. A subset of Virtualization Module instructions and registers must be implemented, but certain instructions and machine state are defined to be optional and may be omitted.

2.4 Overview of the Virtualization Module

The Virtualization Module extends the MIPS32® 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.

2.5 Instruction Bit Encoding

Table 2.2 through Table 2.5 describe the instruction encodings used for the Virtualization Module. Table 2.1 describes the meaning of the symbols used in the tables. These tables only list the instruction encodings for the Virtualization Module instructions. See Volume I of this multi-volume set for a full encoding of all instructions.

Table 2.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Modules. If the Module is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.

Table 2.2 Virtualization Module Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010	*COP0* δ							
3	011								
4	100								
5	101								
6	110								
7	111								

Table 2.3 Virtualization Module COP0 Encoding of rs field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	β	∗	Vδ	MTC0	β	∗	∗
1	01	*	*	∗	∗	*	∗	∗	∗
2	10	C0 δ							
3	11								

Table 2.4 MIPS32 *COP0* Encoding of Function Field When *rs*=*V*

V	*bits 10..8*						
0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
βMFGC0	β	βMTGC0	β	∗	∗	∗	∗

Table 2.5 Virtualization Module COP0 Encoding of Function Field When *rs*=*CO*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	∗	TLBR	TLBWI	TLBINV	TLBINVF	∗	TLBWR	∗
1	001	TLBP	TLBGR	TLBGWI	TLBGINV	TLBGINVF	∗	TLBGWR	∗
2	010	TLBGP	∗	∗	∗	∗	∗	∗	∗
3	011	ERET	∗	∗	∗	∗	∗	∗	DERET
4	100	WAIT	∗	∗	∗	∗	∗	∗	∗
5	101	HYPCALL	∗	∗	∗	∗	∗	∗	∗
6	110	∗	∗	∗	∗	∗	∗	∗	∗
7	111	∗	∗	∗	∗	∗	∗	∗	∗

Chapter 3

Overview of Virtualization Support

3.1 Overview

The Virtualization Module defines a set of extensions to the MIPS32 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.

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, 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, 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.

Minimum Requirements for Virtualization

The first implementations of platform virtualization used ‘trap-and-emulate’ software techniques, which rely on certain properties of the underlying hardware. To be considered ‘classically virtualizable’ an architecture must have the following characteristics:

- At least two operating modes - including privileged and unprivileged
- System resources can only be controlled through privileged instructions while executing in privileged mode
- Execution of a privileged instruction in unprivileged mode will cause an exception (trap), returning control to privileged mode software
- Address translation is performed on the entire address space when in unprivileged mode

In the ‘classic’ approach, the guest operating system kernel is ‘de-privileged’ and is executed in the 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. This approach is also known as ‘trap and emulate’ virtualization.

The base MIPS32 architecture satisfies all the requirements for classic virtualization, except that address translation is not provided for the entire address space in user mode. User mode programs can only run from kuseg, located in the lower portion of the virtual address space. The kernel is typically compiled to run from kseg0, which is located in the upper portion of the virtual address space, and is accessible only in kernel mode. An operating system kernel compiled to work with instructions and data located in kseg0 cannot efficiently execute in user mode.

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.

In addition to addressing the minimum requirements for virtualization, the Virtualization Module provides features designed to reduce the number of hypervisor traps required, and to reduce the length of each hypervisor intervention.

For an outline of virtualization support and for a description of each included feature, see Chapter 4, “The Virtualization Privileged Resource Architecture” on page 13.

For a description of how each feature is intended to be used by software, see Section 4.13 “Virtualization Module features and Hypervisor Software”.

For a description of recommended features, see Table 4.7.

Chapter 4

The Virtualization Privileged Resource Architecture

4.1 Introduction

The MIPS32 Privileged Resource Architecture (PRA) defines a set of environments and capabilities on which the Instruction Set Architecture operates. This includes definitions of the programming interface and operation of the system coprocessor, CP0. The Virtualization Module defines extensions to the MIPS32 PRA that are desirable for the execution of guest Operating Systems in a fully virtualized environment. This document describes these extensions. It is not intended to be a stand-alone PRA specification, and must be read in the context of the MIPS32 Privileged Resource Architecture specification.

4.2 Overview

The Virtualization Module defines extensions to MIPS32 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. This Coprocessor 0 context is referred to by the term ‘context’ throughout this document.

The presence of the Virtualization Module is indicated by the *Config3*$_{VZ}$ field.See Section 5.9 “Configuration Register 3 (CP0 Register 16, Select 3)”.

4.3 Compliance

Features described as *Required* in this document are required of all processors claiming compatibility with the Virtualization Module. Any features described as *Recommended* should be implemented unless there is an overriding need not to do so. Features described as *Optional* are features that may or may not be appropriate for a particular Virtualization Module processor implementation. If such a feature is implemented, it must be implemented as described in this document if a processor is to claim compatibility with the Virtualization ModuleModule.

In some cases, there are features within features that have different levels of compliance. For example, if there is an *Optional* field within a *Required* register, this means that the register must be implemented, but the field may or may not be, depending on the needs of the implementation. Similarly, if there is a *Required* field within an *Optional* register, this means that if the register is implemented, it must have the specified field.

4.4 Operating Modes

Fundamental to the Virtualization Module is a limited-privilege guest operating mode. Guest mode consists of new operating modes guest-kernel, guest-user and guest-supervisor - orthogonal to the existing kernel, user and supervisor modes.

The pre-existing (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.

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.

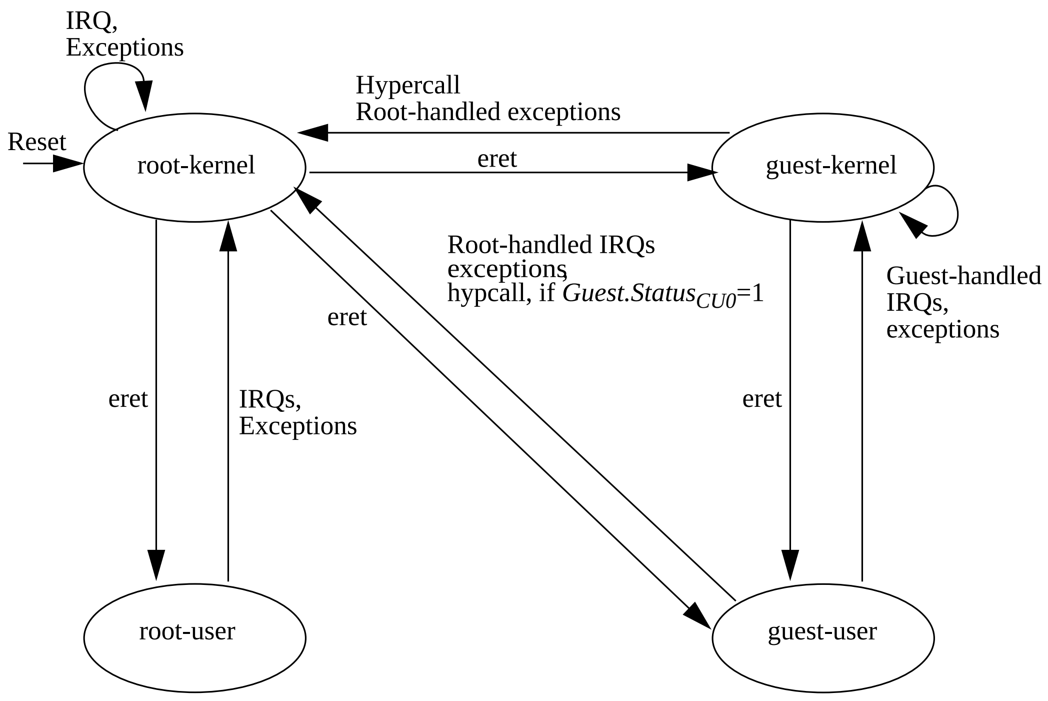
The separation between root mode and the limited-privilege guest mode allows root mode software to be in full control of the machine at all times even when a guest is running. Backward compatibility is retained for existing software running in root mode.

The *GuestCtl0* register, described in Section 5.2, 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.

See also Section 4.4.3 "Definition of Guest Mode".

Figure 4.1 shows the state transitions betwee n operating modes.

Figure 4.1 State Transitions between Operating Modes

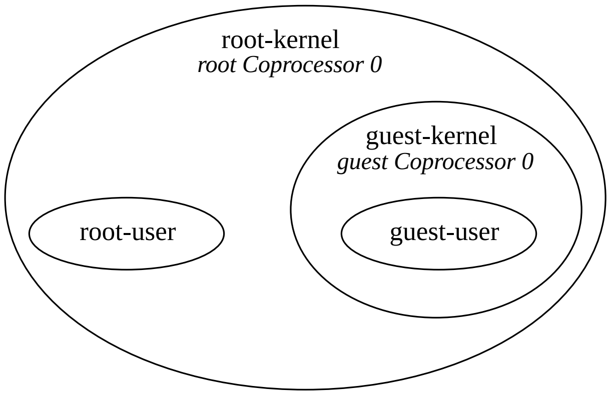


4.4.1 The Onion Model

The Virtualization Module applies an ‘onion model’ to address translation and exception handling for guests. Three operating modes are required to execute a virtualized guest operating system: unprivileged guest-user, limited-privilege guest-kernel and full-privilege root-kernel. The root-user mode is used to execute non-virtualized software. At each layer within the onion, any operation must be permitted by all the outer layers.

Figure 4.3 shows the logical arrangement of operating modes.

Figure 4.2 Virtualization Module Onion Model



In a MIPS32 processor, Coprocessor 0 contains system control registers, and can be accessed only by privileged instructions. A processor implementing the Virtualization Module physically replicates a subset of the Coprocessor 0 register set for use by the Guest Operating System. Root mode operation uses one set of Coprocessor 0 registers and Guest mode operation the other. The term ‘context’ refers to the software visible state held within each Coprocessor 0 register set. 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.

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.

During guest mode execution, 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.

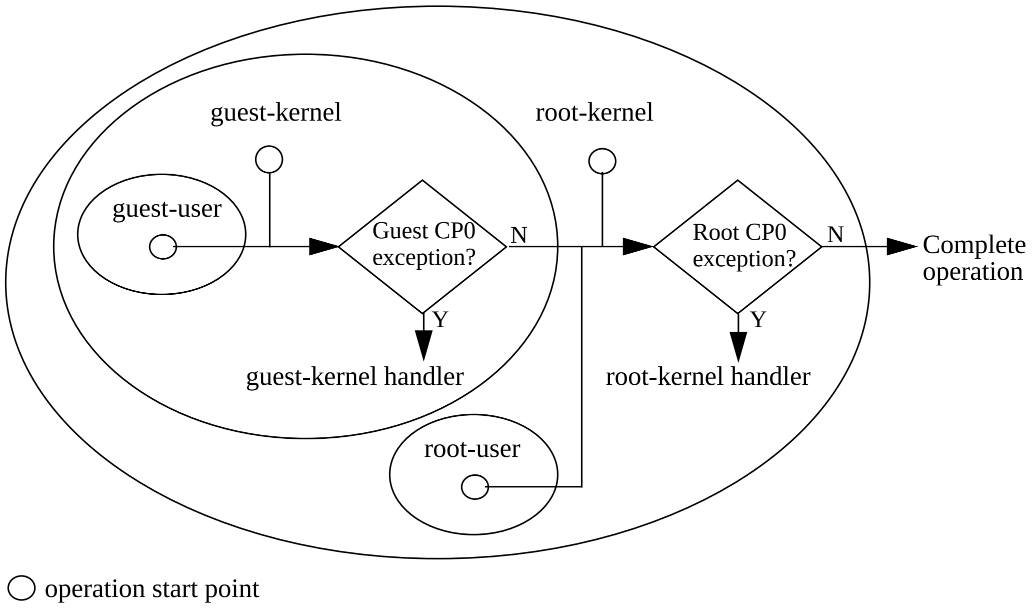
Exceptions are handled in the mode whose context triggered the exception. An exception triggered by the guest CP0 context will be handled in guest mode. An exception triggered by the root CP0 context will be handled in root mode.

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 presence of two Coprocessor 0 contexts allows for an immediate switch between guest and root modes - without requiring a context switch to/from memory. Simultaneously active contexts for the guest and root Coprocessor 0 allows guest-kernel privileged code to execute with the minimum hypervisor intervention, and ensures that key root-mode machine systems such timekeeping, address translation and external interrupt handling continue to operate without major changes during guest execution.

Figure 4.3 shows the how the Virtualization Module 'onion model' is applied to operations starting in each of the operating modes (supervisor modes are omitted for clarity).

Figure 4.3 Virtualization Module Onion Model and exceptions



An operation executed in guest-user mode must travel from the inside of the onion to the outside.

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 taken in guest mode.

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.

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.

External interrupts must travel from the outside of the onion to the inside - first being parsed by the root CP0 context, and if passed on by the hypervisor software, by the guest CP0 context.

4.4.2 Terminology

When executing in guest mode, both the root and guest Coprocessor 0 contexts are in active use. See Section 4.4.1 "The Onion Model". A prefix is used to distinguish between registers located in the guest and root contexts.

For example - *Root.Status* refers to the status register from the root context, and *Guest.Compare* refers to the timer compare register in the guest context.

Pseudocode in this document uses object-oriented terminology to describe processes which can be applied to multiple contexts. A prefix is used to indicate which context is to be operated on by the process. In object-oriented terminology, the subroutines shown are akin to methods provided by a CP0 class.

For example:

```
# Perform TLB lookup using Root CP0 context
# - exceptions taken in root context
Root.TLBLookup(.., .., ..)

# Perform TLB lookup using Guest CP0 context
# - exceptions taken in guest context
Guest.TLBLookup(.., .., ..)

# Perform TLB lookup using context defined by ‘object’ variable
# - exceptions taken in ‘object’ context
object.TLBLookup(.., .., ..)

# Perform TLB lookup using context of the caller
TLBLookup(.., .., ..
```

4.4.3 Definition of Guest Mode

4.4.3.1 Definition

The processor is in guest mode (guest-user, guest-supervisor or guest-kernel) when:

- *Root.GuestCtl0*$_{GM}$ = 1 and *Root.Status*$_{EXL}$=0 and *Root.Status*$_{ERL}$=0 and *Root.Debug*$_{DM}$=0.

Guest mode operation is determined as follows. This subroutine will be used in pseudo-code to test whether processor is in guest-mode.

```
subroutine IsGuestMode() :
    if (GuestCtl0_GM=1) and (Root.Debug_DM=0) and
        (Root.Status_ERL=0) and (Root.Status_EXL=0) then
        return(true)
    else
        return(false)
    endif
endsub
```

In contrast, the following subroutine is to be used in pseudo-code to test whether processor is in root-mode.

```
subroutine IsRootMode() :
    if (
        (GuestCtl0_GM=0) or
        ((GuestCtl0_GM=1) and not ((Root.Debug_DM=0) and
        (Root.Status_ERL=0) and (Root.Status_EXL=0))
        ) then
        return(true)
    else
        return(false)
    endif
endsub
```

4.4.3.2 Entry to Guest mode

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.

Instructions executed in root mode use the root context. When an ERET instruction is executed in root mode and *Root.Status*$_{ERL}$=0, the target address is obtained from *Root.EPC* and the exception-level bit EXL is cleared in *Root.Status*. After the ERET instruction execution is completed, the processor will be in guest mode if the *Root.GuestCtl0*$_{GM}$ bit was set.

The behavior of ERET, and DERET and their usage of *EPC*, *ErrorEPC* and *DEPC* registers are unchanged from the base architecture. The determination of Guest vs. Root mode is the result of setting the Root register fields *GuestCtl0*$_{GM}$, *Status*$_{EXL}$, *Status*$_{ERL}$ and *Debug*$_{DM}$ to the Guest mode definition state (*Root.GuestCtl0*$_{GM}$ = 1 and *Root.Status*$_{EXL}$=0 and *Root.Status*$_{ERL}$=0 and *Root.Debug*$_{DM}$=0).

4.4.3.3 Exit from Guest mode

When an interrupt or exception is to be taken in root mode, the bits *Root.Status*$_{EXL}$ or *Root.Status*$_{ERL}$ are set on entry, before any machine state is saved. As a result, execution of the handler will take place in root mode, and root mode exception context registers are used, including *Root.EPC*, *Root.Cause*, *Root.BadVAddr*, *Root.Context*, *Root.EntryHi*.

The HYPCALL instruction is provided for controlled guest-to-root transitions. This instruction triggers a Hypercall Exception, taken in root mode. See Section 4.7.11 "Hypercall Exception".

The ERET instruction cannot be used to enter root mode from guest mode. No root-mode state is accessible from guest mode, thus the guest cannot change the *Root.GuestCtl0*, *Root.Status* or *Root.Debug* registers.

4.4.3.4 Guest mode execution

When running in guest mode, the distinction between guest-user, guest-supervisor and guest-kernel is made using $Guest.Status_{ERL}$, $Guest.Status_{EXL}$ and $Guest.Status_{KSU/UM}$, following the rules described in the base architecture.

When an interrupt or exception is to be taken in guest mode, the bits $Root.Status_{EXL}$ or $Root.Status_{ERL}$ remain unaltered on entry. As a result, execution of the handler will take place in guest mode, and guest mode exception context registers are used, including *Guest.EPC*, *Guest.Cause*, *Guest.BadVAddr*, *Guest.Context*, *Guest.EntryHi*.

4.4.3.5 Reset

At reset, $Root.Status_{ERL}$=1, thus a MIPS32 processor will always start in root mode.

In addition, $Root.GuestCtl0_{GM}$=0 on reset, ensuring that the operation of existing software is unchanged.

4.4.3.6 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.

4.4.3.7 Fields affecting processor mode

Table 4.1 describes the fields affecting the processor mode.

Table 4.1 Guest, Root and Debug modes

Root					Guest			
$Debug_{DM}$	$Status_{ERL}$	$Status_{EXL}$	$Status_{KSU}$	GuestCtl0 GM	$Status_{ERL}$	$Status_{EXL}$	$Status_{KSU}$	Mode
1	Don't care							Debug

Table 4.1 Guest, Root and Debug modes

Root					Guest			
Debug$_{DM}$	**Status$_{ERL}$**	**Status$_{EXL}$**	**Status$_{KSU}$**	**GuestCtl0 GM**	**Status$_{ERL}$**	**Status$_{EXL}$**	**Status$_{KSU}$**	**Mode**
0	1	Don’t care						Root-Kernel
	0	1	Don’t care					
		0	00	0	Don’t care			
			01					Root-Supervisor
			10					Root-User
			Don’t care	1	1	Don’t care		Guest-Kernel
					0	1	Don’t care	
						0	00	
							01	Guest-Supervisor
							10	Guest-User
					Don’t care		11	**UNPREDICTABLE**
Don’t care			11	Don’t care				**UNDEFINED**

4.4.4 The Guest Context

The Virtualization Module provides root-mode software with controls over the instructions that can be executed, the registers which can be accessed, and the interrupts and exceptions which can be taken when in guest mode. These controls are combined with new exceptions that return control to root mode when intervention is required. The overall intent is to allow guest-mode software to perform the most common privileged operations without root-mode intervention - including transitions between guest kernel and guest user mode, controlling the virtual memory system (the TLB) and dealing with interrupt and exception conditions. Controls allows root-mode software to enforce security policies, and allow for virtualized features to be provided using direct access or trap-and-emulate approaches.

The features added by the Virtualization Module are primarily concerned with virtualizing the privileged state of the machine and dealing with related exception conditions. Hence most features are related to guest-mode interaction with Coprocessor 0. A partially-populated Coprocessor 0 context is added for guest-mode use. See Section 4.6 "Coprocessor 0".

The Virtualization Module provides controls to trigger an exception on any access to Coprocessor 0 from the guest, access to a particular register or registers, or to trigger an exception after a particular field has been changed. See Section 5.2 "GuestCtl0 Register (CP0 Register 12, Select 6)".

The guest Coprocessor 0 context includes its own interrupt system. Root-mode software can directly control guest interrupt sources, and can also pass through one or more external hardware interrupts to the Guest. Guest mode software can enable or disable its own interrupts to enforce critical regions. The root-mode interrupt system remains active, allowing timer and external interrupts to be dealt with by root-mode handlers at any time. See Section 4.8 "Interrupts".

The guest context includes its own TLB. This is useful for fully virtualized systems, where direct guest access to the TLB is necessary to maintain performance. A two-level address translation system is present, along with the related exception system. This system is used to manage guest mode access to virtual and physical memory, and then to relate those accesses to the real machine's physical memory. See Section 4.5 "Virtual Memory".

All MIPS32 unprivileged instructions and registers can be used by guest mode software without restriction. This includes the General Purpose Registers (GPRs) and multiplier result registers *hi* and *lo*. See Section 4.9 "Instructions and Machine State, other than CP0".

MIPS defines optional architecture features and Modules which add machine state and instructions to the base MIPS32 architecture. Some examples include the Floating Point Unit, the DSP Module, and the *UserLocal* register. The presence of these optional features and Modules within the machine is indicated by read-only configuration bits in the $Root.Config_{0..7}$ registers.

The Virtualization Module allows implementations to choose which optional features are available to the guest context. The optional features available to the guest are indicated by fields in the $Guest.Config_{0..7}$ registers. An implementation can further choose to allow run-time configuration of the features available to the guest by allowing root-mode writes to fields in the $Guest.Config_{0..7}$ registers.

Root-mode software can control guest writes to the *Guest.Config* registers when $GuestCtl0_{CF}=0$. This allows Root to control changes to Guest configuration, or be informed of changes to Guest configuration. See Section 4.6.6 "Guest Config Register Fields".

The base MIPS32 architecture includes access controls which allow kernel-mode code to limit access to optional or Module features. Examples include the $Status_{CU1}$ bit and the $Status_{MX}$ bit. The 'onion model' requires that both root-mode and guest-mode permissions are applied to guest-mode accesses. For example, access to the floating point unit must be enabled by the root ($Root.Status_{CU1}$) and the guest ($Guest.Status_{CU1}$) before exception-free accesses can be performed. See Section 4.9.4 “Floating Point Unit (Coprocessor 1)”. There are exceptions to the onion model, for example the *HWREna* register only applies in respective context for guest and root operations.

In a fully virtualized system, the virtual machine presented to the guest is a faithful copy of a real machine - all processor state, instructions, memory and peripherals operate as expected by the guest software.

Figure 4.4 shows a simplified MIPS32 processor during root mode execution. Shadow register controls determine which General Purpose Register set is used. Multiplier result registers are accessible in user and kernel modes. Address translation is performed using a TLB-based MMU and Segment Configurations. Access to the FPU is controlled by kernel-mode software using the *Status*$_{CU1}$ bit. Interrupts can result from external sources or the system timer. Exceptions can result from address translation, breakpoints, instruction execution, or serious errors such as NMI, Machine Check or Cache Error.

The example assumes a non-EIC interrupt system, and for reasons of clarity, omits Supervisor modes and *Config*$_{0..7}$ registers.

Figure 4.4 Simplified processor operation in root mode

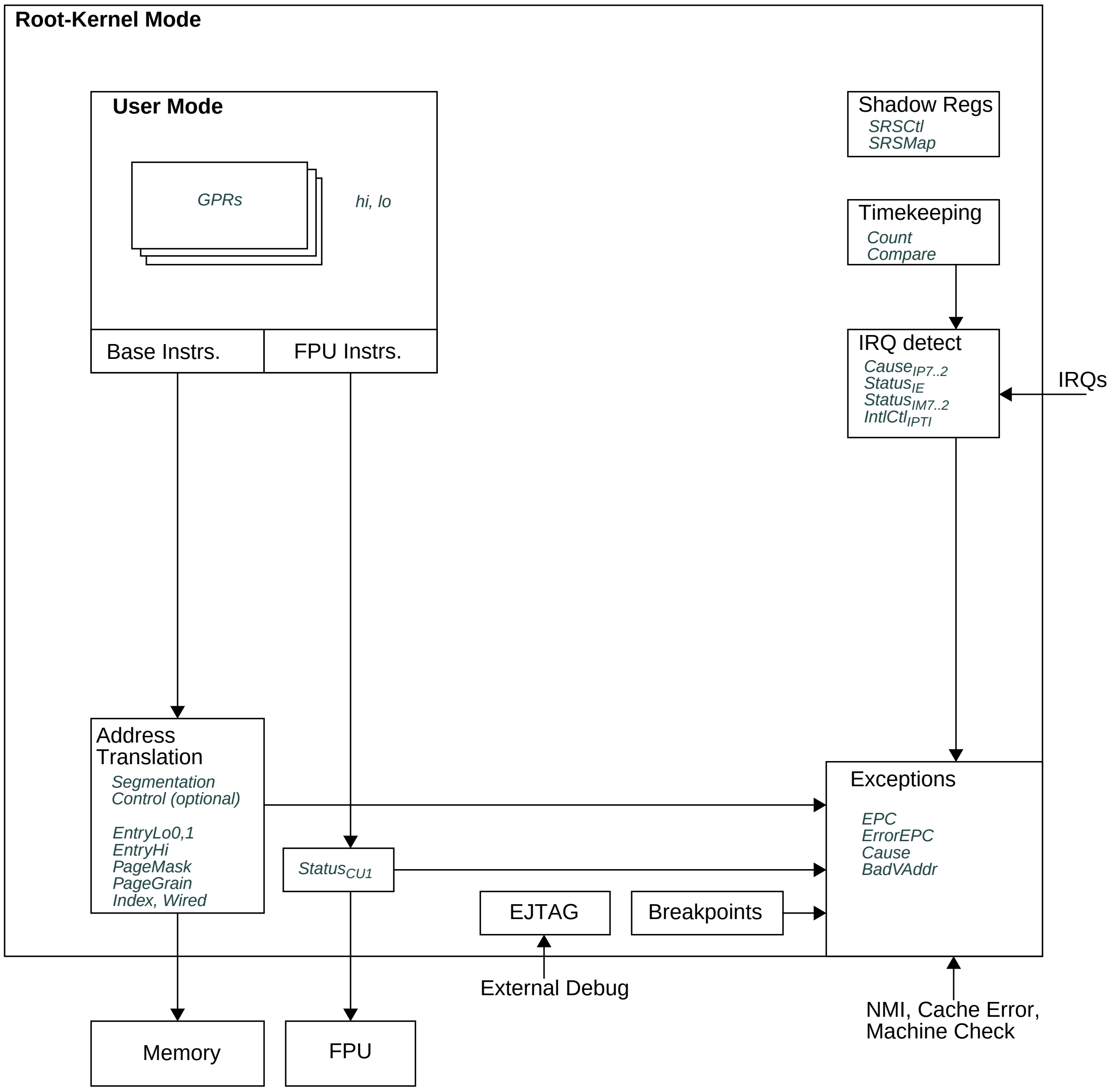
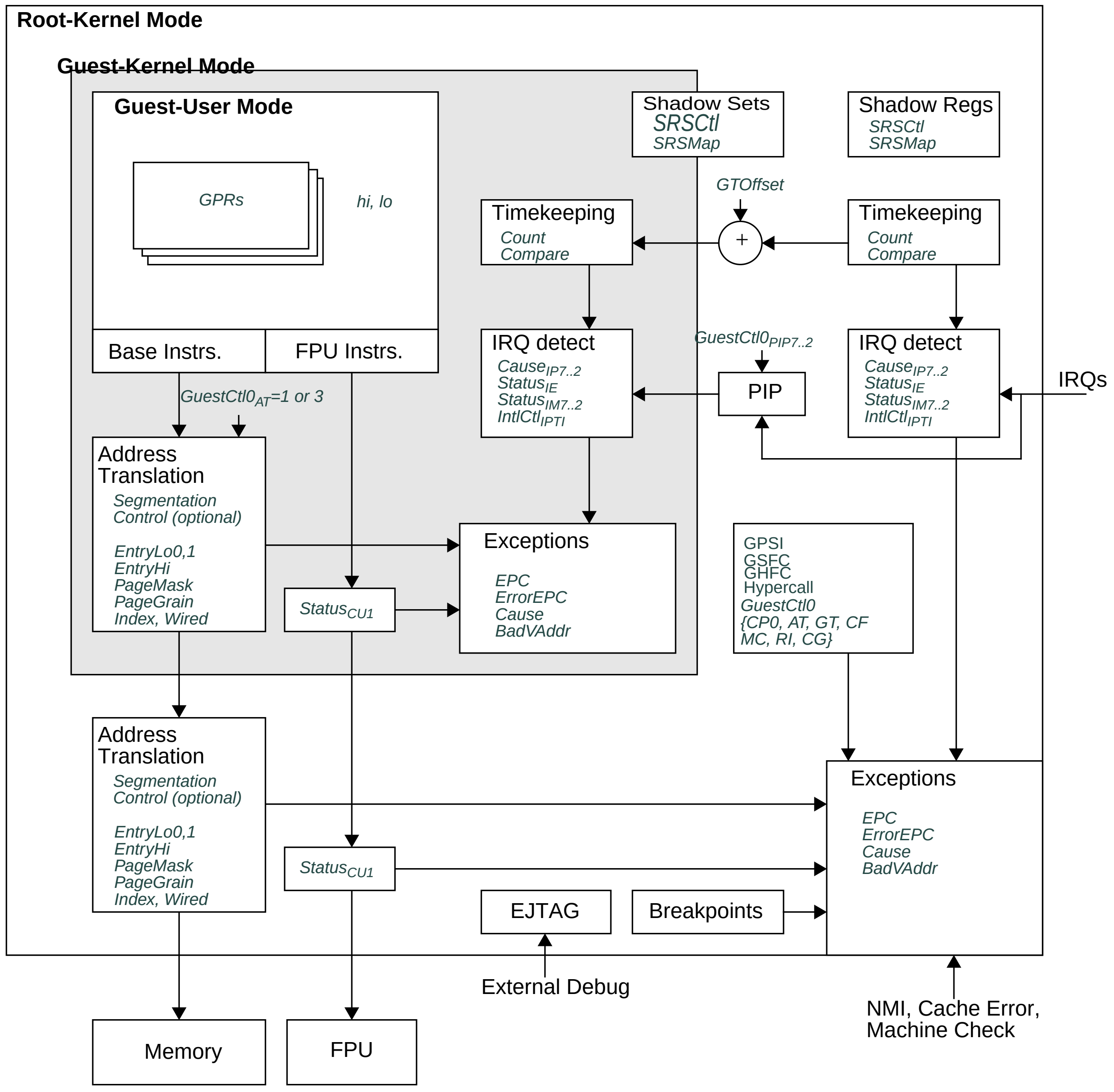


Figure 4.5 shows the Virtualization Module ‘onion model’ applied to the simplified MIPS32 processor from Figure 4.4, for a fully virtualized guest. Guest context shadow register controls determine which General Purpose Register set is used. Multiplier result registers are accessible in user and kernel modes. Address translation is performed first using the guest context (enabled by *GuestCtl0*$_{AT}$=1 or 3), then through the root context TLB. Note that root context Segment Configurations are not used - the root context TLB translates every address from the guest.

Exceptions detected by the guest context are handled in guest mode - from guest segmentation/translation, guest coprocessor enables, guest timekeeping, and IRQs - both external sources passed through by the root context, and IRQ sources directly asserted by root-mode software. Exceptions detected by the root context are handled in root mode - root timekeeping, IRQs, coprocessor enables and second-level address translation, plus new controls over guest behavior.

Figure 4.5 Virtualization Module Onion Model applied to simplified processor (full virtualization)



4.5 Virtual Memory

The Virtualization Module includes an option for two levels of address translation to be applied during guest-mode execution. The Virtualization Module requires that a TLB-based MMU is implemented in the root context.

The Virtualization Module provides a separate CP0 context for guest-mode execution. This context can optionally include segmentation controls and address translation (MMU). The guest MMU can be TLB-based, block address translation (BAT) or fixed mapping (FMT).

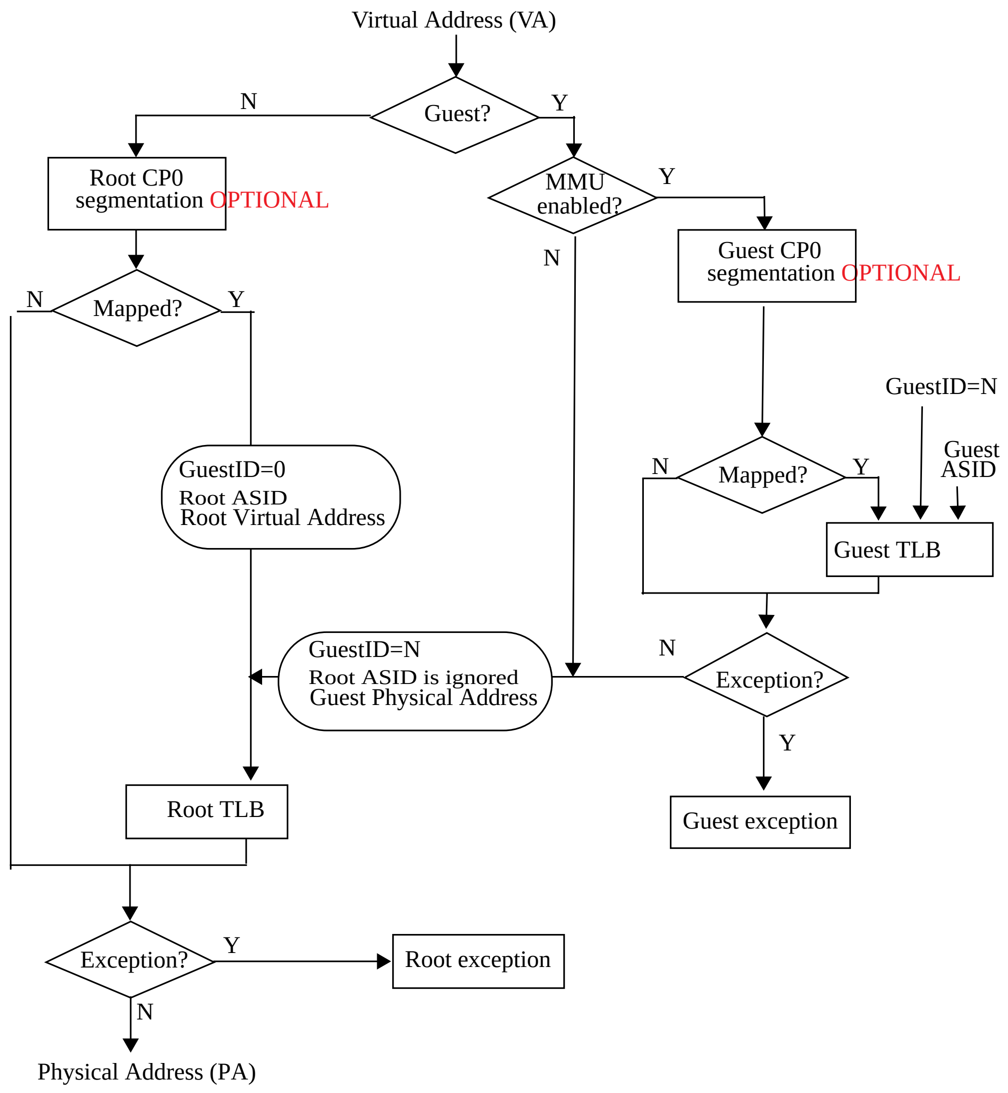
In guest mode when guest segmentation and translation are enabled ($GuestCtl0_{AT}$=1 or 3), two levels of address translation are performed. The first level uses the guest segmentation controls and the guest MMU. This translates an address from a Guest Virtual address (GVA) to a Guest Physical Address (GPA). The second level of translation uses the root TLB, using the GPA in place of the Virtual Address (VA) that would normally be used. This second translation results in a Physical Address (PA). The cache attribute used is that supplied by the guest context. In this second level of translation, exceptions in address translation are handled by Root.

When a TLB-based guest MMU is provided, it is recommended 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. We recommend that the Root TLB maintain an adequate amount of reserved TLB entries for its own use to avoid cascading TLB evictions (thrashing).

Figure 4.6 shows the outline of address translation in the Virtualization Module.

Figure 4.6 Outline of Address Translation



Implementation note: Processor designs incorporating the Virtualization Module and implementing a guest context MMU are unlikely to perform translation twice on each memory access. A hardware mechanism will be used to ensure that a Physical Address can be obtained from a Guest Virtual Address within the CPU pipeline in a single translation. The mechanism may use micro-TLBs - for example, on a micro-TLB refill, a guest TLB lookup would be followed by a root TLB lookup, to produce a one-step GVA-PA translation. Other methods are possible. The system must be arranged to allow for efficient execution and to appear to software that two independent translation steps are taking place for each memory access.

Guest mode segmentation controls and the guest mode MMU have no effect on the root mode address space.

The optional ‘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 implementation of a GuestID is recommended. Implementation complexity can be minimized by reducing the GuestID to 1 bit. This allows the Root TLB to distinguish between Root and Guest Entries, and flush either set of mappings in entirety with the TLBINVF instruction. Alternatively, GuestID can be eliminated by having Root virtual address space shared with Guest physical addresses.

The pseudocode below describes the complete address translation process for the MIPS32 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. See Section 4.5.1 "Virtualized MMU GuestID Use" for description of *RAD* and *DRG* terms.

```
/* Inputs
 * vAddr  - Virtual Address
 * IorD   - Access type - INSTRUCTION or DATA
 * LorS   - Access type - LOAD or STORE
 * pLevel - Privilege level - USER, SUPER, KERNEL
 *
 * Outputs
 * pAddr  - physical address
 * CCA    - cache attribute (valid when mapped)
 *
 * Exceptions: See called functions
 * Called from guest or root context.
 */
subroutine AddressTranslation(vAddr, IorD, LorS, pLevel)

   // Initialization.
   // GuestID is only applicable if $GuestCtl0_{RAD}$=0. Otherwise GuestID
   // is ignored (not applicable) in process of address translation.
   GuestID ← ignored

   if (IsGuestMode()) then
      // This is a Guest Address translation
      // step 1: Guest Virtual -> Guest Physical Address translation
      if ($GuestCtl0_{RAD}$=0)
            GuestID ← $GuestCtl1_{ID}$
      endif
      (mapped, addr, CCA) ← AddressDecode(vAddr, pLevel)
      if ($Config_{MT}$=1 or $Config_{MT}$=4) then // TLB type MMU
         if (mapped) then
            asid ← $Guest.EntryHi_{ASID}$
            (addr, CCA) ← Guest.TLBLookup(asid, GuestID, addr, IorD, LorS)
         endif
      else
         if ($Config_{MT}$=0) then
            # MMU=None case is undefined
            **UNDEFINED**
         else
            # Other MMU type, FMT or BAT. BAT will use LorS.
            (addr, CCA) ← Guest.OtherMMULookup(addr, CCA, LorS, pLevel)
         endif
      endif
      if (exception)
         Guest Exception
         // TLB exceptions may include Refill, Invalid, Execute-Inhibit for
         // Instruction, Refill, Invalid, Modified, Read-Inhibit for Data.
```

```
            // Guest segment map related exceptions may include Address Error
        endif


        // step 2: Guest Physical -> Root Physical Address translation
        // if $GuestCtl0_{RAD}$=0, then guest entry ASID is global in Root TLB.
        // H/W must set G=1 for guest entry for TLBWI and TLBWR.
        asid ← $Root.EntryHi_{ASID}$
        pAddr ← Root.TLBLookup(asid, GuestID, addr, IorD, LorS)
        if (exception)
            Root Exception
            // This is a Root exception initiated in guest context
            // This includes all TLB exceptions.
            // Segment map Address Error exception not included, as guest does not
            // lookup root segment map.
        endif

    else
        // This is a Root Address translation
        // Root Virtual -> Root Physical Address translation
        // If $GuestCtl0_{DRG}$=1,$GuestCtl0_{RID}$ 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 $GuestCtl0_{RID}$!=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(vAddr[31:29], pLevel)
        endif
        return (mapped, addr, CCA)
    endsub
```

See also Section 4.7.1 "Exceptions in Guest Mode" and Section 4.7.2 "Faulting Address for Exceptions from Guest Mode".

4.5.1 Virtualized MMU GuestID Use

The use of GuestID is optional as specified by the value of $GuestCtl0_{G1}$. Software can detect presence of *GuestCtl1* and thus $GuestCtl1_{ID}$ and $GuestCtl1_{RID}$ by reading $GuestCtl0_{G1}$.

For an implementation that supports $GuestCtl0_{RAD}$=0, $GuestCtl0_{G1}$ must be preset to 1, otherwise $GuestCtl0_{G1}$ must be preset to 0. $GuestCtl0_{RAD}$ is read-only - an implementation can support one or the other, but never both. On the other hand, $GuestCtl0_{DRG}$ is R/W. See Table 5.2 for description of R/W state of *DRG* and *RAD*.

$GuestCtl1_{ID}$ is used for guest-mode operation, while $GuestCtl1_{RID}$ is used for root-mode operation. Root address translation assumes GuestID=0 providing $GuestCtl0_{DRG}$=0.

The Guest TLB may or may not be shared by multiple guests. The Root TLB will be shared by Root and at least one unique Guest. Options to support dealiasing guest and root entries in Root TLB, and possibly multiple guests in the Guest TLB is described below.

A processor will support one of the two modes below. Software can determine the mode by reading $GuestCtl1_{RAD}$ described in Table 4.2

1. Dealiasing by GuestID

   GuestID is used to dealias multiple guest contexts in both Guest and Root TLB. Specifically, $GuestCtl1_{ID}$ is used for guest-mode operation, whereas $GuestCtl1_{RID}$ is used for root-mode operations. A guest or root-mode operation is an instruction or data translation, or TLB instruction.

   An implementation may choose to provide direct root-mode access to guest entries (GPA->RPA) in the Root TLB. Direct root-mode access is described by $GuestCtl0_{DRG}$ in Table 4.2. In the absence of this feature, root would have to probe the Root TLB with GPA, and subsequently read on match to obtain the RPA. If a miss occurs, then root must walk the guest shadow page tables in memory. Otherwise, with direct access, a miss will result in a hardware pagewalk, assuming a hardware pagewalker is supported.

   Root ASID for guest entries in the Root TLB are ignored because hardware will set the global bit on a write for such entries.

2. Dealiasing by Root ASID.

   This option should be used if no GuestID is implemented. Software can detect this mode by reading $GuestCtl1_{RAD.}$

Between Guest context-switches, the Guest and Root TLBs must be flushed of current guest context by root software. *Root.EntryHi$_{ASID}$* is used to dealias Root from Guest entries in the Root TLB. Root software must maintain a one is to one correspondence between allocated ASID and the unique Guest it represents.

Root ASID for guest entries in the Root TLB are not ignored unless software explicitly sets G=1 for the guest entry.

.

Table 4.2 GuestID Translation Related Usage Mode Control

Field	Description	Encoding	Meaning
GuestCtl0$_{RAD}$	RAD, or “Root ASID Dealias” mode determines the means that a Virtualized MMU implementation uses to dealias different contexts.		
		0	GuestID used to dealias both Guest and Root TLB entries in Root TLB.
		1	Root ASID is used to dealias Root TLB entries, while Guest TLB contains only one context at any given time.
GuestCtl0$_{DRG}$	DRG, or “Direct Root to Guest” access determines whether an implementation with *GuestCtl0$_{RAD}$*=0 provides root kernel the means to access guest entries directly in the Root TLB for access to guest memory. If GuestCtl0$_{DRG}$=1 then GuestCtl0$_{RID}$ must be used. If GuestID for root operation is non-zero, root is in kernel mode, Root.Status$_{EXL,ERL}$=0 and Debug$_{DM}$=0, then all root kernel data accesses are mapped, root SegCtl is ignored and Root TLB CCA is used. Access in root mode by other than kernel will cause an address error. H/W must set G=1 as if the access were for guest.		
		0	Root software cannot access guest entries directly.
		1	Root software can access guest entries directly.

The following pseudo-code indicates how to specify the ASID and GuestID(if present) interface to the Root and Guest TLBs for Guest and Root address translations, as a function of *GuestCtl0$_{RAD}$*. A field within a TLB entry needs to be compared with a “Key” as input to the interface to determine whether a match is has occurred.

Guest and Root TLB interfaces for GuestID dealiasing method (*GuestCtl0$_{RAD}$=0)*:

Guest TLB Interface:
```
if (Instruction or Load or Store)
        GuestTLB.Key[GuestID] = GuestCtl1_ID
endif
```

```
GuestTLB.Key[ASID] = $Guest.EntryHi_{ASID}$

Root TLB Interface:
if ( IsRootMode() )
        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 GuestCtl0$_{RID}$!=0 and !Instruction)
        if (!drg_valid) then
                // Instruction or Load or Store
                RootTLB.Key[GuestID] = 0
        else    // special mode - root access guest entries
                RootTLB.Key[GuestID] = $GuestCtl1_{RID}$
        endif
else    // Guest mode
        // Instruction or Load or Store
        RootTLB.Key[GuestID] = $GuestCtl1_{ID}$
endif
RootTLB.Key[ASID] = $Root.EntryHi_{ASID}$
```

With $GuestCtl0_{RAD}$=0, Guest entries in the Root TLB must ignore the ASID. For this reason, if $GuestCtl_{RID}$*!=0*, that is entry is a Guest entry, then Root mode execution of TLBWI and TLBWR sets the entry's G bit to 1 automatically. Otherwise, for Root entries, TLBWI and TLBWR must set/clear the G bit in accordance with the baseline architecture.

Guest and Root TLB interface for Root ASID dealiasing method ($GuestCtl0_{RAD}$=1*)* :

```
Guest TLB Interface:
        GuestTLB.Key[ASID] = $Guest.EntryHi_{ASID}$
Root TLB Interface:
        RootTLB.Key[ASID] = $Root.EntryHi_{ASID}$
```

$GuestCtl0_{DRG}$ has no effect on the Guest and Root address translations if $GuestCtl0_{RAD}$=1. *If* $GuestCtl0_{RAD}$=1, then $GuestCtl0_{DRG}$ must be read-only as 0.

For more detail on Guest and Root address translation, please refer to pseudo-code in Section 4.5 "Virtual Memory".

Table 4.3 specifies the association of GuestID with TLB instructions. For supporting information, refer to Section 4.6.2 "New CP0 Instructions".

Table 4.3 GuestID Use by TLB instructions.

TLB Operation	GuestID ($GuestCtl_{ID}$/$GuestCtl1_{RID}$)
TLBGINV	$GuestCtl1_{RID}$
TLBGINVF	$GuestCtl1_{RID}$
TLBGP	$GuestCtl1_{RID}$
TLBGR	$GuestCtl1_{RID}$
TLBGWI	$GuestCtl1_{RID}$
TLBGWR	$GuestCtl1_{RID}$

Table 4.3 GuestID Use by TLB instructions.

TLB Operation	GuestID (*GuestCtl$_{ID}$*/*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$_{ID}$*

4.5.2 Root and Guest Shared TLB Operation

An implementation may choose to share a common physical TLB amongst root and guest. In a TLB structure that incorporates a VTLB (Variable page size TLB) and FTLB (Fixed page size TLB), the VTLB must accommodate wired entries for both root and guest in a shared structure. In other implementations, the VTLB may be standalone without a supporting FTLB.

In a non-virtualized design, the number of wired entries is limited by the CP0 *Wired* register in either context. And the number of entries in the VTLB is determined by *Config1$_{MMUSize-1}$* and *Config4$_{VTLBSizeExt}$* or *Config4$_{MMUSizeExt}$*. For this purpose, it is required that any of these fields be writeable by root as given in Table 4.10.

In a recommended 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, if programmed appropriately. 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.

It is expected that root will allocate the appropriate number of wired entries to itself, and then write guest *Config1* and *Config4* related fields to set the available VTLB entries for guest. Root will read *Guest.Config4$_{MMUExtDef}$* to determine which of the guest *Config4* MMU size extension fields need to be written. Since the entries allocated for guest use also includes non wired entries shared by both root and guest, root software must be careful not to allocate all remaining non root-wired entries to guest. This prevents 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.

It is not required that hardware check for illegal values written to guest MMU size and extensions. A typical implementation will however check to ensure that any field write saturates at the maximum number of bits required to support the total number of entries in the shared TLB.

4.6 Coprocessor 0

Defined by the MIPS32 Privileged Resource Architecture (PRA), Coprocessor 0 (CP0) contains system control registers. Access to these registers is restricted and can only be performed using privileged instructions.

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.Config*$_{0\text{-}7}$ registers determine which architecture features are active during guest mode execution. The *GuestCtl0* register controls whether a guest access to a privileged feature will trigger an exception.

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.

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 as defined in the base architecture. On 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.

Root has the ability to define the presence of and control the contents of Guest CP0 registers. Therefore, if so configured, Guest access to guest CP0 state may cause a Guest Privileged Sensitive Instruction exception. Refer to Table 4.7, Section 4.6.6 “Guest Config Register Fields” and Section 4.7.7 “Guest Privileged Sensitive Instruction Exception” for further information.

Root may deconfigure guest CP0 registers by writing to guest configuration registers as defined in Table 4.10. Guest behavior in response to these modifications is defined in Table 4.8.

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.

4.6.1 New and Modified CP0 Registers

Coprocessor 0 registers are added by the Virtualization Module to control the guest context - *GuestCtl0, GuestCtl1* and *GTOffset*.

Table 4.4 describes CP0 registers introduced by the Virtualization Module.

Table 4.4 CP0 Registers Introduced by the Virtualization Module

Register Number	Sel	Register Name	Description	Reference	Compliance Level
12	6	*GuestCtl0*	Controls guest mode behavior.	Section 5.2	Required
10	4	*GuestCtl1*	Guest ID	Section 5.3	Optional
10	5	*GuestCtl2*	Virtual Interrupts	Section 5.4	Optional
10	6	*GuestCtl3*	Virtual Shadow Sets	Section 5.5	Optional
11	4	*GuestCtl0Ext*	Extension to GuestCtl0	Section 5.6	Optional
12	7	*GTOffset*	Offset for guest timer value	Section 5.7	Required

Table 4.5 describes CP0 registers modified by the Virtualization Module.

Table 4.5 CP0 Registers Modified by the Virtualization Module

Register Number	Sel	Register Name	Description	Reference	Compliance Level
13	0	*Cause*	Addition of hypervisor cause code.	Section 5.8	Required
16	3	*Config3*	Identifies Virtualization Module feature set.	Section 5.9	Required
19	0	*WatchHi*	Added support for Guest Watch.	Section 5.10	Optional
25	0	*PerfCnt*	Added support for Root/Guest performance count.	Section 5.11	Optional
31	2	*KScratch1*	Required in root context.	-	Required
31	3	*KScratch2*	Required in root context.	-	Required

4.6.2 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 4.6 describes CP0 instructions introduced by the Virtualization Module.

Table 4.6 CP0 Instructions Introduced by the Virtualization Module

Instruction	Description	Reference	Compliance Level
HYPCALL	Hypercall - call to root mode.	“HYPCALL” on page 124	Required
MFGC0	Move from Guest CP0	“MFGC0” on page 125	
MTGC0	Move to Guest CP0	“MTGC0” on page 127	
TLBGINV	Guest TLB Invalidate	“TLBGINV” on page 128	Optional
TLBGINVF	Guest TLB Invalidate Flush	“TLBGINVF” on page 130	Optional
TLBGP	Probe Guest TLB	“TLBGP” on page 133	Required when guest TLB present
TLBGR	Read Guest TLB	“TLBGR” on page 136	
TLBGWI	Write Guest TLB	“TLBGWI” on page 138	
TLBGWR	Write Random to Guest TLB	“TLBGWR” on page 140	

4.6.3 Guest CP0 registers

The Virtualization Module provides a partial set of CP0 registers for use by the guest, this is known as the *guest context*. Many guest context registers are optional or can be disabled under software control.

As in the base architecture, fields in *Guest.Config*, *Guest.Config1..7* registers define the architectural capabilities of the guest context. When a CP0 register does not exist in the guest context, or is disabled by a root-writable *Guest.Config* field, it can have no effect on guest behavior. See Section 4.6.6 "Guest Config Register Fields" for information on guest Config register fields which can be dynamically reconfigured by Root. Note that accesses to Guest CP0 registers in certain cases will trigger a Guest Privileged Sensitive Instruction (GPSI) exception as defined in Table 4.7.

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 guest CP0 context (thus controlling guest-mode execution) while denying guest-kernel access to the register. See Section 4.6.4 "Guest Privileged Sensitive Features".

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. This is done to properly manage the operation of the guest virtual machine. See Section 4.6.5 "Access Control for Guest CP0 Register Fields".

Table 4.7 lists the base architecture CP0 registers noting which may be implemented in the guest context.

Definitions of terms used in Table 4.7:

- Required - Must be implemented in the Guest context.
- Recommended - Should be implemented in the Guest context.
- Optional - Implementation dependent as to whether included in the Guest context.
- Not Available - Never implemented in the Guest context.

The guest CP0 context must include all CP0 registers from an optional feature or an Module if the associated *Guest.Config* field indicates that the feature or Module is available in the guest context. For any of these registers, guest access may be controlled by Root software. This is done by triggering a Guest Privileged Sensitive Instruction Exception on a guest-mode access. Guest Software Field Change and Guest Hardware Field Change exceptions can also be used.

See also Section 4.10 “Combining the Virtualization Module and the MT Module”.

Table 4.7 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	Compliance Level
0	0	*Index*	*Guest.Config*$_{MT}$=1 or *Guest.Config*$_{MT}$=4	*GuestCtl0Ext*$_{MG}$=1	Required for Guest context TLB
1	0	*Random*			
2	0	*EntryLo0*			
3	0	*EntryLo1*			
4	0	*Context*			
4	1	*ContextConfig*	*Guest.Config3*$_{SM}$=1 or *Guest.Config3*$_{CTXTC}$=1		Optional
4	2	*UserLocal*	*Guest.Config3*$_{ULRI}$=1	*GuestCtl0Ext*$_{OG}$=1	Recom-mended
5	0	*PageMask*	*Guest.Config*$_{MT}$=1 or *Guest.Config*$_{MT}$=4	*GuestCtl0Ext*$_{MG}$=1	Required for Guest context TLB
5	1	*PageGrain*		*GuestCtl0*$_{AT}$=1	
5	2	*SegCtl0*	*Guest.Config3*$_{SC}$=1		Optional
5	3	*SegCtl1*			
5	4	*SegCtl2*			
5	5	*PWBase*	*Guest.Config3*$_{PW}$=1		Optional
5	6	*PWField*			
5	7	*PWSize*			
6	0	*Wired*	*Guest.Config*$_{MT}$=1 or *Guest.Config*$_{MT}$=4		Required for Guest context TLB
6	6	*PWCtl*	*Guest.Config3*$_{PW}$=1		Optional
7	0	*HWREna*	*Guest.Config*$_{AR}$>=1	*GuestCtl0Ext*$_{OG}$=1	Required
8	0	*BadVAddr*	Always	*GuestCtl0Ext*$_{BG}$=1	
8	1	*BadInstr*	*Guest.Config3*$_{BI}$=1	*GuestCtl0Ext*$_{BG}$=1	Optional
8	2	*BadInstrP*	*Guest.Config3*$_{BP}$=1	*GuestCtl0Ext*$_{BG}$=1	Optional
9	0	*Count*	*Always*	*GuestCtl0*$_{GT}$=0	Required
10	0	*EntryHi*	*Guest.Config*$_{MT}$=1 or *Guest.Config*$_{MT}$=4	*GuestCtl0Ext*$_{MG}$=1	Required for Guest context TLB
11	0	*Compare*	Always	*GuestCtl0*$_{GT}$=0	Required
12	0	*Status*	Always	-	
12	1	*IntCtl*	*Guest.Config*$_{AR}$>=1	-	

Table 4.7 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	Compliance Level
12	2	*SRSCtl*	*Guest.Config$_{AR}$*>=1	Always	Optional
12	3	*SRSMap*	*Guest.Config$_{AR}$*>=1		
13	0	*Cause*	Always	-	Required
13	5	*NestedExc*	*Guest.Config5$_{NFExists}$*=1	-	Optional
14	0	*EPC*	Always	-	Required
14	2	*NestedEPC*	*Guest.Config5$_{NFExists}$*=1	-	Optional
15	0	*PRid*	-	Always	Not Available Emulated by Hypervisor
15	1	*EBase*	*Guest.Config$_{AR}$*>=1	-	Required
15	2	*CDMMBase*	*Guest.Config3$_{CDMM}$*=1	Always	Not Available Emulated by Hypervisor
15	3	*CMGCRBase*	*Guest.Config3$_{CMGCR}$*=1		
16	0	*Config*	Always	On write access when *GuestCtl0$_{CF}$*=0.	Required
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		
16	6	*Config6*	Implementation dependent	-	Optional
16	7	*Config7*			
17	0	*LLAddr*		*GuestCtl0Ext$_{OG}$*=1	Optional[1]
18	0	*WatchLo*	*Guest.Config1$_{WR}$*=1	Conditional, refer to Section 4.12 "Watchpoint Debug Support"	Optional
19	0	*WatchHi*	*Guest.Config1$_{WR}$*=1		

Table 4.7 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	Compliance Level
23	0	*Debug*	*Guest.Config1*$_{EP}$=1	Always	Not Available
24	0	*DEPC*	*Guest.Config1*$_{EP}$=1		
25	0-n	*PerfCnt*	*Guest.Config1*$_{PC}$=1	Conditional, refer to Section 4.8.4 "Performance Counter Interrupts"	
26	0	*ErrCtl*	-	Always	
27	0	*CacheErr*			
28	0	*TagLo*			
28	1	*DataLo*			
28	2	*TagLo*			
28	3	*DataLo*			
29	0	*TagHi*			
29	1	*DataHi*			
29	2	*TagHi*			
29	3	*DataHi*			
30	0	*ErrorEPC*	Always[2]	-	Required
31	0	*DESAVE*	*Guest.Config1*$_{EP}$=1	Always	Not Available
31	2	*KScratch1*	Always Defined by *Guest.Config4*$_{KScrExist}$	*GuestCtl0Ext*$_{OG}$=1	Optional
31	3	*KScratch2*			
31	4	*KScratch3*			
31	5	*KScratch4*			
31	6	*KScratch5*			
31	7	*KScratch6*			

1. LLAddr may optionally be implemented providing the Guest context has access to Guest Physical Addresses, else Not Available.
2. ErrorEPC is required in guest context because it used as scratch by some MIPS compatible OSes.

Table 4.7 indicates the conditions under which guest access of guest CP0 registers can cause a Guest Privileged Sensitive Instruction exception (GPSI) to Root. If a GPSI is taken for a guest CP0 register which may or may not be active in guest mode, the corresponding root CP0 register must be implemented. This is true because the guest CP0 context is always a subset of the root CP0 context. Otherwise, access to the corresponding guest CP0 register is UNPREDICTABLE.

If the configuration of a Guest accessible CP0 register can be modified by Root, then Guest access behavior is as specified in Table 4.8.

Root should not modify Guest configuration while the Guest is running. It is assumed that the Guest software will read its configuration registers during boot and not thereafter. Since Root can modify guest configuration, Root should maintain a copy of guest configuration at hardware reset so that it knows which guest CP0 registers are actually implemented. Once modified by Root, the guest configuration registers may not accurately reflect the physical existence of guest CP0 registers.

Table 4.8 Root Modification of Guest CP0 Configuration

Register Replicated in Guest Context?	Guest Configuration register bit Root writeable as per Table 4.10	Guest Configuration Register bit value on reset	Guest Configuration Register bit value after write by Root, if writeable	Interpretation of Configuration
No	No	0	N/A	The register does not exist in Guest. Reads and writes to this register are UNDEFINED.
Yes	No	1	N/A	The register is replicated in the Guest. Guest can access its version of the register without traps to Root excluding the cases identified in Table 4.7
No	Yes	0	0	The register exists in Root and is not replicated in the Guest context. In Guest mode, reads and writes to this register are UNDEFINED.
No	Yes	0	1	The register exists in Root and is not replicated in the Guest context. In Guest mode, reads and writes to this register throw a GPSI exception which allows Root to selectively emulate the register. Registers which conform to this definition are the Watch Registers (4.12) and Performance Registers (5.11).
Yes	Yes	1	1	The register exists in the Root context and is replicated in the Guest context. Guest can access its version of the register without exception excluding cases identified in Table 4.7
Yes	Yes	1	0	The register exists in the Root context and is replicated in the Guest context. Guest access to the register is disabled. Reads and writes to the register are UNDEFINED.

4.6.3.1 Guest Reserved Register Handling

This section defines the behaviour of guest access to reserved CP0 registers of different types.

1. Reserved for Architecture. These are CP0 registers reserved by the privileged architecture for future use.

2. Reserved for Implementation. These are CP0 registers reserved for implementations which may or may not be present in guest context.

   The list of registers is CP0 Register 9 (Selects 6 and 7), Register 11 (Selects 6 and 7), Register 16 (Selects 6 and 7), Register 22 (all Selects).

The behaviour of Reserved for Architecture registers follows.

```
if (GuestCtl0_CP0=0) {
        <GPSI>
} elsif (GuestCtl0Ext_OG=1) {
        <GPSI>
} elsif (is_MFC0) {
        MFC0 is UNPREDICTABLE
} else { // is_MTC0
        MTC0 is UNPREDICTABLE
}
```

The behaviour of Reserved for Implementation registers follows.

```
if (GuestCtl0_CPO=0) {
        <GPSI>
} elsif (is_MFC0) {
        MFC0 is UNPREDICTABLE
} else {
        MTC0 is UNPREDICTABLE
}
```

Reserved for Implementation registers are not qualified by *GuestCtl0Ext*$_{OG}$=1 because the requirements for implementation dependent registers is unknown.

4.6.4 Guest Privileged Sensitive Features

The *GuestCtl0* register controls which privileged features can be accessed from guest mode. See Section 5.2 “GuestCtl0 Register (CP0 Register 12, Select 6)”.

A hypervisor can limit guest access to privileged (CP0) registers and privileged sensitive instructions. A hypervisor exception is taken when a guest accesses a privileged feature which is ‘sensitive’. See Section 4.7.7 “Guest Privileged Sensitive Instruction Exception”.

4.6.5 Access Control for Guest CP0 Register Fields

The MIPS32 Privileged Resource Architecture includes register fields which are critical to machine behavior, where a Guest Hardware Field Change (GHFC) or Guest Software Field Change (GSFC) requires immediate hypervisor intervention. Guest Software Field Change and Guest Hardware Field Change detection mechanisms are provided in order to reduce the need for hypervisor exceptions for all CP0 writes, exceptions, interrupts and privileged instructions which could cause changes to critical fields.

The *GuestCtl0*$_{MC}$ field controls programmable change detection for certain guest CP0 fields. Changes to these fields will always result in a Guest Software Field Change or Guest Hardware Field Change exception.

See Section 4.7.8 “Guest Software Field Change Exception” and Section 4.7.9 “Guest Hardware Field Change Exception”.

Table 4.9 lists fields which can trigger a GSFC or GHFC exception. The architecture also provides the capability to disable GSFC and GHFC exceptions with *GuestCtl0Ext*$_{FCD}$. Table 4.9 assumes *GuestCtl0Ext*$_{FCD}$=0. See Section 4.14 “Lightweight Virtualization” and Table 5.8 for reference to *GuestCtl0Ext*$_{FCD}$.

Table 4.9 Guest CP0 Fields Subject to Software or Hardware Field Change Exception

Register	Field	Purpose	Exception Type
Status	CU2..CU1	Coprocessor access. *Status*$_{CU1}$ causes GSFC if *GuestCtl0*$_{SFC1}$=0 *Status*$_{CU2}$ causes GSFC if *GuestCtl0*$_{SFC2}$=0	GSFC
Status	RP	Reduced power mode. Guest value is ignored, *Root.Status*$_{RP}$ controls system power mode.	GSFC
Status	FR	Floating point register mode.	GSFC
Status	MX	Enable access to MDMX and DSP resources.	GSFC
Status	BEV	Bootstrap exception vector. Controls location of exception vectors, and is used to determine EIC vs non-EIC interrupt mode.	GSFC
Status	TS	TLB multiple match.	Both
Status	SR	Reset exception vector due to Soft Reset.	GSFC
Status	NMI	Reset exception vector due to Non-Maskable Interrupt.	GSFC
Status	Impl (17..16)	Implementation dependent.	GSFC
Status	UM/KSU	Operating mode. GSFC exception only when *GuestCtl0*$_{MC}$=1.	GSFC
Status	EXL	Exception level. GHFC exception only when *GuestCtl0*$_{MC}$=1.	GHFC
Status	ERL	Error level.	GSFC
Cause	DC	Disable Count. Root software should disable guest timer access and emulate a non-counting timer when this bit is set by the guest.	GSFC
Cause	IV	Interrupt Vector. Controls EIC vs non-EIC interrupt mode.	GSFC
IntCtl	VS	Vector spacing. Controls EIC vs non-EIC interrupt mode.	GSFC
PerfCnt	Event, EventExt	Performance Counter Control Event field. EventExt is *Optional* in implementations.	GSFC

4.6.6 Guest Config Register Fields

The *Guest.Config*$_{0-7}$ registers control the behavior of architecture features during guest execution. All fields follow base MIPS32 architecture definitions.

Virtualization Module implementations are permitted to choose whether to implement *Optional* MIPS32 features in the guest context. All *Required* features specified by the architecture revision (*Guest.Config*$_{AR}$) must be implemented. The operation of the guest context must always follow the setting of the *Guest.Config* register fields.

The guest context must be a subset of the root context - the guest context can only include features available in the root context.

The MIPS32 architecture defines many read-only *Config* register fields. For each read-only *Root.Config*$_{0-7}$ register field, the Virtualization Module implementation must choose a fixed value or allow dynamic reconfiguration in the corresponding *Guest.Config*$_{0-7}$ field.

Dynamic configuration is implemented by permitting root-mode writes to fields in *Guest.Config* registers. Only values supported by the implementation will be accepted on writes to read-only *Guest.Config* fields from root mode.

When an unsupported value is written, the field will remain unchanged after the write. The *Guest.Config* fields controlling dynamic reconfiguration are never writable from guest mode.

Root mode software can determine whether programmable features are available in the guest context by attempting to write values to *Guest.Config* fields.

Table 4.10 lists *Guest.Config* register fields which can be written from root mode in the MIPS32 Virtualization Module

The virtualization architecture does not require that hardware provide the capability to emulate different architectural releases for guest software that is different from the base implementation, due to complexity. For this reason, root cannot write $Guest.Config_{AR}$.

Table 4.10 Guest CP0 Read-only Config Fields Writable from Root Mode

Register	Field	Purpose	Root write
Config	M	*Config1* implemented	Optional
Config	MT	MMU Type	Optional
Config1	M	*Config2* implemented	Optional
Config1	MMU Size - 1	Number of entries in (guest) MMU	Required for-Shared TLB[1]
Config1	C2	Coprocessor 2 implemented	Optional
Config1	MD	MDMX implemented	Optional
Config1	PC	Performance Counter registers implemented	Optional
Config1	WR	Watch registers implemented	Optional
Config1	CA	Code compression (MIPS16e) implemented	Optional
Config1	FP	FPU implemented	Optional
Config2	M	*Config3* implemented	Optional
Config3	M	*Config4* implemented	Optional
Config3	BPG	Big pages feature implemented	Optional
Config3	ULRI	UserLocal implemented	Optional
Config3	DSP2P	DSP Module Revision 2 implemented	Optional
Config3	DSPP	DSP Module implemented	Optional
Config3	CTXTC	*ContextConfig* etc. implemented	Optional
Config3	ITL	IFlowTrace mechanism implemented	Optional
Config3	VEIC	External Interrupt Controller implemented	Optional
Config3	VInt	Vectored interrupts implemented	Optional
Config3	SP	Small pages feature implemented	Optional
Config3	CDMM	Common Device Memory Map implemented	Optional
Config3	MT	MT (MultiThreading) Module implemented	Optional
Config3	SM	SmartMIPS Module implemented	Optional
Config3	TL	Trace Logic implemented	Optional
Config4	M	*Config5* implemented	Optional

Table 4.10 Guest CP0 Read-only Config Fields Writable from Root Mode

Register	Field	Purpose	Root write
Config4	VTLBSizeExt	Extends $Config1_{MMUSize-1}$ if $Config4_{MMUExtDef}=3$	Required for Shared TLB[1]
Config4	MMUSizeExt	Extends $Config1_{MMUSize-1}$ if $Config4_{MMUExtDef}=1$	Required for Shared TLB[1]

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 as described in Section 4.5.2 .

4.6.7 Guest Context Dynamically Set Read-only Fields

The MIPS32 Privileged Resource Architecture includes register fields which 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.

Reserved (zero) bits and static configuration bits are not included. The *Random* register is not included.

Table 4.11 lists fields which are read-only to the guest and writable from root mode.

Table 4.11 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
SRSCtl	HSS	Highest Shadow Set
SRSCtl	EICSS	External Interrupt Controller Shadow Set
SRSCtl	CSS	Current Shadow Set
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.

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.

4.6.8 Guest Timer

Timekeeping within the guest context is controlled by root mode. The guest time value is generated from the root timer value *Root.Count* by adding the two's complement offset in the *Root.GTOffset* register. The guest time value can be read from the *Guest.Count* register, and is used to generate timer interrupts within the guest context.

When *GuestCtl0*$_{GT}$=1, guest mode can read and write the *Compare* register, and can read from the *Count* register. A guest write to *Count* always results in a Guest Privileged Sensitive Instruction exception.

When *GuestCtl0*$_{GT}$=0, all guest accesses to the *Count* and *Compare* registers result in a Guest Privileged Sensitive Instruction exception, including read via the RDHWR instruction.

The value of *Guest.Cause*$_{DC}$ has no direct effect on the calculation of the guest time value. A Guest Software Field Change (GSFC) exception results when an attempt is made to change the value of *Guest.Cause*$_{DC}$ from guest mode. Note that the value of *Root.Cause*$_{DC}$ affects the value of *Root.Count* during debug mode operation - this indirectly affects the value of *Guest.Count*.

The guest timer interrupt affects only the guest context - it cannot interrupt the root context. Similarly, the root timer interrupt cannot be directly assigned to the guest.

Usage note: *Guest.Cause*$_{TI}$ is set when *Guest.Count* = *Guest.Compare*, even when the device is running in Root mode. In order to preserve the value of *Guest.Cause*$_{TI}$ while restoring *Guest.Cause*, the following approach may be taken:

```
#
Root.Status_EXL ← 1

# Calculate desired GTOffset value based on saved
# Guest.Count and current Root.Count values as well as hypervisor policies.
# GTOoffset has a few different purposes:
#      - To provide each guest a different value of Count.
#      - To restore a guest's virtual time between context switches.
# In the latter case, GTOffset allows Root to restore time to when a guest was
# switched out, by offsetting Root.Count by elapsed time.Or it allows guest Count
# to reflect elapsed time also.
#
# Under the simplest scheme, the new GTOffset must adjust current Root.Count
# for elapsed time between guest save an restore.

new_gt_offset ← calculate_gt_offset()
GTOffset ← new_gt_offset
# Restore Guest.Cause since Guest.Cause.TI may be 1.Guest.Cause must be saved
# after Guest.Count to provide most current Cause.TI.
Guest.Cause ← saved_cause

# after the following statement, the hardware might now set Guest.Cause[TI]

Guest.Compare ← saved_compare
current_guest_count ← Guest.Count

# set Guest.Cause_TI if it would have been set while the guest was sleeping.
# Since GTOffset for the guest and Guest.Compare restore is not atomic, this code
# is required to ensure that Guest.Cause.TI is set appropriately, since current
# Guest.Count could have raced ahead of saved_count before restoring Guest.Compare.
if (current_guest_count > saved_count) then
```

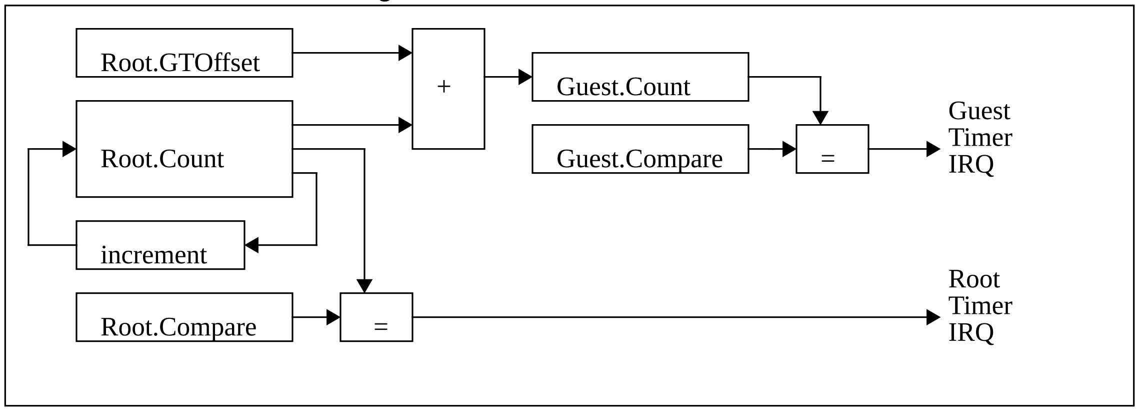
```
        if (saved_compare > saved_count && saved_compare < current_guest_count) then
            saved_cause[TI] ← 1
            Guest.Cause ← saved_cause
        endif
    else
        # The count has wrapped. Check to see if
        # Guest.Count has passed the saved_compare value.
        if (saved_compare > saved_count || saved_compare < current_guest_count) then
            saved_cause[TI] ← 1
            Guest.Cause ← saved_cause
        endif
    endif

    #The trick is to not overwrite the Guest.Cause here
    Root.GuestCtl_GM ← 1
    restore_register_state()
    eret
    #
```

Root-mode writes to *Guest.Count* are ignored.

See also Section 4.8 "Interrupts" and Section 5.7 "GTOffset Register (CP0 Register 12, Select 7)".

Figure 4.7 shows how the guest timer value is computed from the root timer.

Figure 4.7 Root and Guest Timers



4.6.9 Guest Cache Operations

A limited set of cache operations can be performed from guest mode, when the CACHE instruction is enabled by *GuestCtl0*$_{CG}$=1. For this case, any guest-mode cache operation using Effective Address Operand type other than 'Address' will result in a Guest Privileged Sensitive Instruction exception.

When *GuestCtl0*$_{CG}$=0, guest-mode execution of the CACHE instruction will result in a Guest Privileged Sensitive Instruction exception.

The above description also applies to the CACHEE instruction, which is optional in the baseline architecture.

See Section 4.7.7 "Guest Privileged Sensitive Instruction Exception".

4.6.10 UNPREDICTABLE and UNDEFINED in Guest Mode

The terms **UNPREDICTABLE** and **UNDEFINED** have specific meanings in MIPS architecture documents. See Section 1.3 "Special Symbols in Pseudocode Notation".

A distinction is drawn between **UNPREDICTABLE** and **UNDEFINED**. Unprivileged instructions can only have results which are **UNPREDICTABLE**.

This is to ensure that unprivileged code cannot:

- Compromise availability by preventing control being returned to the highest level of privilege on an interrupt or exception - for example by causing a hang or other indefinite stall.
- Compromise confidentiality by allowing data (machine state or memory) to be read without permission or detection.
- Compromise integrity by allowing data (machine state or memory) to be altered without permission or detection. This includes:
  - Altering data or instructions used by another process
    - e.g. alter a bank balance or bypass a license check
  - Altering data, instructions or machine state used by the highest level of privilege
    - e.g. to gain a higher level of privilege, or install an alternative interrupt handler
  - Compromised integrity also includes the case where one unprivileged process can communicate with another process without permission - a "covert channel". The channel can use data in memory, machine state which is not context switched, or the ability to cause timing changes detectable in another process.

The definition of **UNPREDICTABLE** requires that any result returned is produced only from data sources which are accessible in the unprivileged mode. This ensures that the **UNPREDICTABLE** result cannot be reproduced by another process - provided that the complete set of available data sources are context switched between unprivileged processes.

Hence process A might be able to perform an operation which produces a deterministic value where an **UNPREDICTABLE** result is defined by the architecture. Process A may even be able to control the value returned. However, if a full context switch is made between process A and process B, then process B will not be able to read hidden messages sent by process A. The value returned by the **UNPREDICTABLE** operation is dependent entirely on the state visible to process B, which has been fully context switched. No covert communication channel is allowed, and no data can be accidentally revealed from another process or from a higher level of privilege.

The definition of **UNDEFINED** only requires that the processor can be returned to a functioning state by application of the reset signal. This means that it is in theory possible to design a system which would allow information to be stored in hidden state, and communicated from one point in privileged code execution to another, even when it appears that all available machine state has been context switched.

The MIPS architecture requires that **UNDEFINED** operations can only result from operations performed in Kernel Mode or Debug Mode, or when the CP0 access bit is set (granting Kernel-level permissions). In other words, **UNDEFINED** operations can result only from operations at the highest level of privilege.

The Virtualization Module adds Guest Kernel Mode as a limited-privilege mode. Software executing in a Guest Mode (guest-kernel, guest-supervisor or guest-user) must never cause an **UNDEFINED** result.

Wherever a privileged operation is described by the MIPS architecture as having an **UNDEFINED** result, this must be interpreted as an **UNPREDICTABLE** result when executing in Guest Mode.

This mechanism ensures that guest operating systems cannot compromise the availability, confidentiality or integrity of the hypervisor, other guests or the system as a whole.

4.7 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.

4.7.1 Exceptions in Guest Mode

The Virtualization Module retains the exception-processing methodology of the base MIPS32 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 4.7.2 “Faulting Address for Exceptions from Guest Mode”.

```
# 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 MIPS32 Privileged Resource Architecture
   ctx.GeneralExceptionProcessing()
endif
```

4.7.2 Faulting Address for Exceptions from Guest Mode

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

- Address error
- TLB Refill
- TLB Invalid
- TLB Modified
- TLB Execute Inhibit
- TLB Read Inhibit

4.7.3 Guest initiated Root TLB Exception

When an exception is triggered as a result of a root TLB access during guest-mode execution, the handler 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. A Guest Virtual Address (GVA) unmapped by the Guest MMU is considered a GPA from the root's perspective.

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.

4.7.4 Exception Priority

Table 4.12 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 4.12 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

Table 4.12 Priority of Exceptions

Exception	Description	Type	Taken in mode
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
Watch - Instruction fetch	A root context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses. Refer to 'Watch Registers' - Section 4.12 "Watchpoint Debug Support".	Synchronous	Root
	A guest-context watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses. Refer to 'Watch Registers' - Section 4.12 "Watchpoint Debug Support".		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

Table 4.12 Priority of Exceptions

Exception	Description	Type	Taken in mode
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources, or was illegal: Coprocessor Unusable, Reserved Instruction, MSA disabled. If both exceptions occur on the same instruction, the Coprocessor Unusable, MSA disabled Exception takes 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
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.	Synchronous Hypervisor	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. Refer to Section 4.7.8 “Guest Software Field Change Exception”.	Synchronous Hypervisor	Root
Guest Hardware Field-Change	During guest execution, a hardware initiated set of *Status*$_{EXL/TS}$ occurred. See Section 4.7.9 “Guest Hardware Field Change Exception” for further information.	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

Table 4.12 Priority of Exceptions

Exception	Description	Type	Taken in mode
Watch - Data access	A root context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses. Refer to 'Watch Registers' - Section 4.12 "Watchpoint Debug Support"	Synchronous	Root
	A guest context watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses. Refer to 'Watch Registers' - Section 4.12 "Watchpoint Debug Support"		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
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 4.12 describes the type of exception. Table 4.13 explains the characteristics of each exception type.

Table 4.13 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.
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.

4.7.5 Exception Vector Locations

Exception vector locations are as defined in the base architecture.

The vector location is determined from the values of *EBase*, *Status*$_{EXL}$, *Status*$_{BEV}$, *IntCtl*$_{VS}$ and *Config3*$_{VEIC}$ obtained from the context in which the exception 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.

4.7.6 Synchronous and Synchronous Hypervisor Exceptions

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

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

Table 4.14 summarizes hypervisor conditions.

Table 4.14 Hypervisor Exception Conditions

Type	Root-mode Vector	Causes	Reference
Synchronous Hypervisor	General	Guest Privileged Sensitive Instruction	Section 4.7.7
Synchronous Hypervisor	General	Guest Software Field Change	Section 4.7.8
Synchronous Hypervisor	General	Guest Hardware Field Change	Section 4.7.9
Synchronous Hypervisor	General	Guest Reserved Instruction Redirect	Section 4.7.10
Synchronous Hypervisor	General	Hypercall	Section 4.7.11

4.7.7 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 term ‘sensitive’ refers to an instruction which may trigger a hypervisor exception when executed in guest-kernel 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*$_{CG}$ as described in Table 5.8. 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 using RDHWR when *GuestCtl0*$_{CP0}$=0 providing Guest CP0 registers are enabled for user access by guest *HWREna.*

- Write to *Count* register
- Access to *SRSCtl* or *SRSMap* CP0 registers when $SRSCtl_{HSS}$ > 0.
- Guest-kernel use of RDPGPR or WRPGPR instructions when $SRSCtl_{HSS}$ > 0.
- Any Privileged Instruction 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.

  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 as given in Table 4.7.

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).

4.7.8 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 bitfields always trigger the exception.

- *Guest.Status* bits: CU[2:1], RP, FR, MX, BEV, SR, NMI, UM/KSU, ERL, Impl (17..16), TS (always on clear, optionally on set),

  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.
- *Config5* : MSAEn. (Enable for MIPS SIMD Architecture module. Applicable if present)
- *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. *GuestCtl0*$_{SFC2}$ and *GuestCtl0*$_{SFC1}$ have been defined in Section 5.2 "GuestCtl0 Register (CP0 Register 12, Select 6)".

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 or TS bits 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. Root will handle hardware asserted SR/NMI as per Table 4.12.

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.

A D/MTC0 that attempts to clear TS will cause a GSFC, while setting of TS, caused by hardware, should result in a GHFC. Optionally, the setting of TS may cause a GSFC also instead of GHFC, for ease of implementation. However, it is recommended that setting of TS result in GHFC.

Clearing of TS will result in GSFC before the D/MTC0 completes. This should be contrasted with setting of TS as described in Section 4.7.9 "Guest Hardware Field Change Exception", which must set the value in *Guest.Status* before GHFC is taken.

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.

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).

4.7.9 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 $\mathit{GuestCtl0Ext}_{FCD}$=1 (refer to Section 5.6), 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 $\mathit{GuestCtl0}_{MC}$=1, changes to the following bitfields trigger this exception.

- Guest *Status* bits: EXL.

Set of the following bitfield triggers this exception.

- Guest *Status* bits: TS (set)

A change in value in any of these fields 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 which of TS or EXL have 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), $\mathit{Cause}_{ExcCode}$(=”Guest Exit”) and $\mathit{GuestCtl0}_{GExcCode}$(=”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.

Guest *$Status_{TS}$* is set by hardware, this exception is taken after TS is set and prior to start of the first instruction of the Guest machine-check exception handler. Therefore, the Guest Hardware Field Change exception handler will return to the first instruction of the Guest machine check exception handler.

See comment in Section 4.7.8 "Guest Software Field Change Exception". Setting of TS in guest context may optionally cause GSFC in lieu of GHFC. GHFC is however recommended response.

Cause Register ExcCode value

GE (27, 0x1B)

GuestCtl0 Register GExcCode value

GHFC(9, 0x09)

Entry Vector Used

General Exception Vector (offset 0x180).

4.7.10 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).

4.7.11 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. For specifics of Hypercall root-kernel and debug mode handling, refer to hypercall definition in Chapter 6, “Instruction Descriptions” .

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).

4.7.12 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
```

4.8 Interrupts

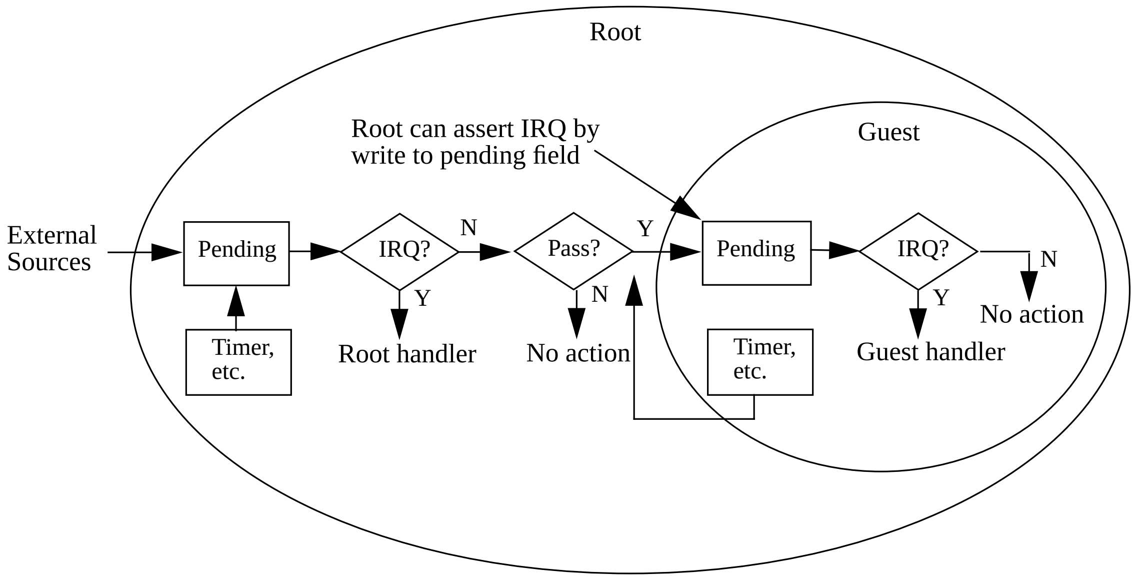
The Virtualization Module provides a virtualized interrupt system for the guest.

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 MIPS32 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 4.8 shows the how the Virtualization Module 'onion model' is applied to interrupt sources.

Figure 4.8 Interrupts in the Virtualization Module onion model



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.

4.8.1 External Interrupts

4.8.1.1 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])))
```

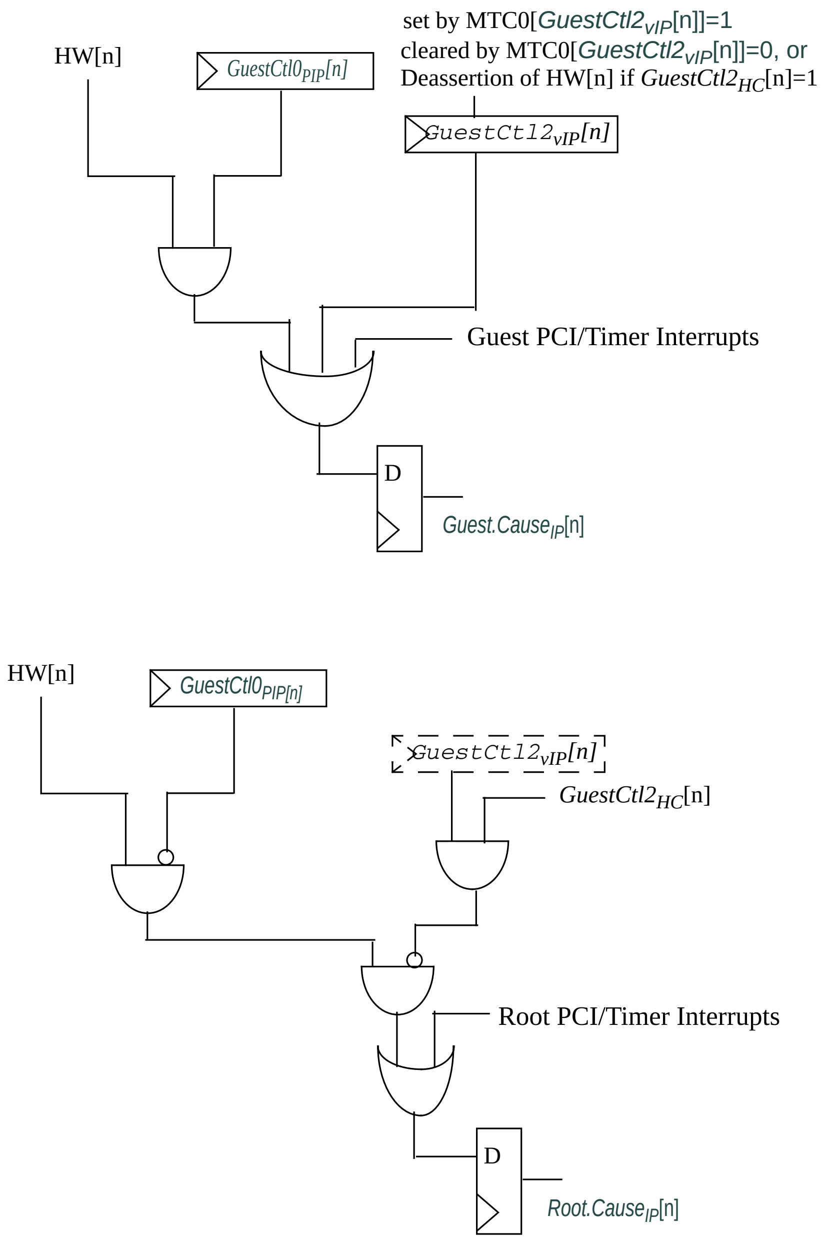
GuestCtl2$_{HC}$ is provided to control how *GuestCtl2*$_{vIP}$ is reset. If a bit of *GuestCtl2*$_{HC}$ is 1, then the deassertion of related external interrupt will always cause associated *GuestCtl2*$_{vIP}$ to be cleared. If a bit of *GuestCtl2*$_{HC}$ is 0 then the deassertion of HW[n] will not cause *GuestCtl2*$_{vIP}$ to be cleared. In this case, it is the responsibility of root software to clear by writing 0 to *GuestCtl2*$_{vIP\ [n]}$. See Section 5.4 "GuestCtl2 Register (CP0 Register 10, Select 5)"for further definition.

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.

Figure 4.9 shows virtualized management of the Guest and Root Cause register IP field . In the absence of support for *GuestCtl2*$_{vIP}$, a hardware-only version of *GuestCtl2*$_{vIP}$ should be considered to exist. Root may write a 1 to the hardware copy with MTGC0[CauseIP]. Root may also write a 0 to the hardware copy to clear the interrupt, whille deassertion of HW[n] will also clear corresponding bit in this hardware register. In presence of *GuestCtl2*$_{vIP}$, root writes to *Guest.Cause*$_{IP[7:2]}$ is considered optional. The mode of a hardware shadow copy should not be implemented if virtual interrupt capability is supported.

Figure 4.9 Guest and Root Cause$_{IP}$ (non-EIC) Virtualization


set by MTC0[GuestCtl2$_{vIP}$[n]]=1
cleared by MTC0[GuestCtl2$_{vIP}$[n]]=0, or
Deassertion of HW[n] if GuestCtl2$_{HC}$[n]=1
HW[n]
GuestCtl0$_{PIP}$[n]
GuestCtl2$_{vIP}$[n]
Guest PCI/Timer Interrupts
D
Guest.Cause$_{IP}$[n]
HW[n]
GuestCtl0$_{PIP[n]}$
GuestCtl2$_{vIP}$[n]
GuestCtl2$_{HC}$[n]
Root PCI/Timer Interrupts
D
Root.Cause$_{IP}$[n]

4.8.1.2 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 $GuestCtl1_{EID}$ as described in Table 5.4. The guest associated with the guest bus is by default equal to $GuestCtl1_{ID}$.

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. If GuestID is zero, then it uses 0x200+Vector as interrupt vector, otherwise it uses 0x200 as interrupt vector.

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 is described in Section 5.4 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.

The overhead of saving and restoring guest CP0 context can be minimized. Table 4.7 indicates which guest CP0 registers will cause a Guest Physical Senstive Instruction (GPSI) on guest access, and under what root configuration. Root software can opportunistically save/restore those guest CP0 registers which cause, or can be configured to cause a GPSI.

Guest GPR Shadow Sets are protected by virtual mapping to physical Shadow Sets. Section 4.9.1 “General Purpose Registers and Shadow Register Sets” describes how root enables virtual mapping for a guest. For the virtual map for Guest GPR Shadow Sets to be enabled, *GuestCtl3*$_{GLSS}$ must be written by root with appropriate value for the guest. It is assumed that *Guest.SRSCtl* is saved and restored.

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.

4.8.2 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 $\leftarrow$ EICGuestLevel
   else
      irq $\leftarrow$ GuestCtl2$_{GRIPL}$
      # h/w must clear if GuestCtl2$_{GRIPL}$ is source of interrupt.
      GuestCtl2$_{GRIPL}$ $\leftarrow$ 0
   endif
   # Guest.Cause$_{IP[1:0]}$ is incorporated in EIC.
   # State of Guest.Cause$_{IP[1:0]}$ is however preserved.
   r  $\leftarrow$(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]  $\leftarrow$ HW[5:0]
   if (GuestCtl0$_{PT}$=0)
      # All interrupts processed first by root.
      if (GuestCtl0$_{G2}$=1)
         # root software injects interrupts.
         r  $\leftarrow$ 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  $\leftarrow$ Root_HW_VIP[5:0]
      endif
   else
      # Guest interrupt passthrough supported.
      if (GuestCtl0$_{G2}$=1)
         r  $\leftarrow$ Root.GuestCtl2$_{vIP[5:0]}$ OR (irq[7:2] AND Root.GuestCtl0$_{PIP[5:0]}$)
      else
         r  $\leftarrow$ Root_HW_VIP[5:0] OR (irq[7:2] AND Root.GuestCtl0$_{PIP[5:0]}$)
      endif
   endif
   r        $\leftarrow$ r << 2
   r        $\leftarrow$ r OR (GuestTimerInterrupt << Guest.IntCtl$_{IPTI}$)
   r        $\leftarrow$ r OR (PCIEvent << Guest.IntCtl$_{IPPCI}$)
   r        $\leftarrow$ r OR Guest.Cause$_{IP[1:0]}$

endif
```

```
return(r)
endsub
```

The value returned by GuestInterruptPending() will subsequently be qualified by Guest *Status*$_{IM}$ in non-EIC mode or Guest *Status*$_{IPL}$ in EIC mode, as per the base architecture.

Fields in Guest *Config* registers indicate which interrupt options are available to the guest.

4.8.3 Timer Interrupts

Root may inject a timer interrupt in guest context by setting Guest *Cause*$_{TI}$ and indirectly Guest *Cause*$_{IP[IPTI]}$. This may happen under the scenario where a guest has been switched out, but its virtual timer, maintained by root, is triggered. Root would set Guest *Cause*$_{TI}$ before entering guest mode for the guest. Guest would take a timer interrupt, clear Guest *Compare*, which would then clear Guest *Cause*$_{TI}$. As per baseline MIPS architecture, a write to *Compare* will clear *Cause*$_{TI}$.

Root maintaining a virtual timer for a guest is recommended if there are multiple guests in operation. Otherwise, if there is only one guest, but the processor is in root mode, then a match on Guest *Count* and Guest *Compare* is allowed in an implementation to set Guest *Cause*$_{TI}$ and Guest *Cause*$_{IP[IPTI]}$. Once Root transitions to guest mode, then guest timer interrupt can be signaled in guest mode.

```
Root Injection of Guest TI:
        if (MTGC0[Guest.Cause_TI]=1)
                 Root.Guest.Cause_TI ← 1
        else if ((MTC0[Guest.Compare]))
                 Root.Guest.Cause_TI ← 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.Cause`$_{TI}$ `or "Other External and Internal interrupts".`

where “Other External and Internal interrupts” is defined in Section 4.8.2 “Derivation of Guest.CauseIP/RIPL”.

4.8.4 Performance Counter Interrupts

Root can configure the definition of performance counters in the Guest context via Guest *Config1*$_{PC}$ as follows:

- Guest *Config1*$_{PC}$=0, then performance counters are unimplemented in the guest context, access is **UNPREDICTABLE**.
- Guest *Config1*$_{PC}$=1, the performance counters are virtually shared by root and guest contexts.

The *PerfCnt* register(s) are never implemented in the Guest context. A Guest may have direct access to virtual performance counter registers under root software management when *Config1*$_{PC}$=1. If virtually shared, the encodings of *PerfCnt*$_{EC}$ as 0 or 1 cause a GPSI Exception to be raised on Guest access to a performance counter register. Root software may choose to configure performance counters for legal Guest access by encoding *PerfCnt*$_{EC}$ as 2 or 3.

Software may choose to assign all performance counters to Guest or Root, but not both. This is the recommended policy for sharing between Root and Guest. Root will typically configure Guest access when it initializes guest context. If assigned to Guest then Guest access will not cause a GPSI Exception.

Alternatively, an implementation may optionally choose to assign a subset of the total *PerfCnt* registers in Root CP0 context to Guest. Read of guest $PerfCnt(N)_M$ should return root $PerfCnt(N+1)_{EC[1]}$ to indicate *PerfCnt(N+1)* is owned by guest. If all *PerfCnt* pairs are allocated to guest, then guest read of the last M bit must return 0. Guest *PerfCnt* pairs assigned to Guest in this manner must be a contiguous range, starting from the least significant pair. It is further assumed that the allotment of performance counters to a guest is not dynamic - once established after initial guest access (which caused GPSI), then the allotment must remain as such for duration of guest.

Once assigned to Guest or Root (default) context, that context independently manages the performance counters, including interrupts. E.g., if the performance counters are enabled for Root, then Root $Cause_{PCI}$ and Root $Cause_{IP[IPPCI]}$ are set by hardware on counter overflow. Otherwise, counter overflow sets Guest.$Cause_{PCI}$ and Guest.$Cause_{IP[IPPCI]}$.

If Root software needs to inject a performance counter interrupt into Guest context, it must do so by setting the most-significant bit of the *PerfCnt* counter. Similarly Root may clear a guest performance counter interrupt by clearing the most-significant bit of the counter. Thus, Root does not require the ability to read/write $Guest.Cause_{PCI}$.

The $PerfCnt_{EC}$ field is Root only virtualization control and is not visible to the Guest.

PerfCnt use of *Status* register *K, S, U,* and *EXL* fields is taken from the current Root or Guest context.

PerfCnt interrupt behavior is solely governed by $PerfCnt_{IE}$, enabled context *Status* register interrupt masks and enable.

4.9 Instructions and Machine State, other than CP0

The Virtualization Module adds guest-mode context to duplicate privileged state, which is located in Coprocessor 0. Typically, all machine state located outside Coprocessor 0 is shared by guest and root contexts and thus would require save or restore by Root between context switches. Alternatively, in limited cases, state may be virtually shared among different contexts as in the case of GPR Shadow Sets.

4.9.1 General Purpose Registers and Shadow Register Sets

Guest *SRSCtl* and *SRSMap* are optional in guest CP0 context. The following cases apply to use and implementation of these CP0 registers.

1. No shadow sets are implemented. In this case, guest access to *SRSCtl and SRSMap,* or guest use of RDPGPR or WRPGPR always cause a GPSI. Root would return emulated Guest $SRSCtl_{HSS}$=0 in guest context to indicate to guest that no shadow sets are present.

2. Shadow sets are implemented in root context only. In this case, guest access to *SRSCtl and SRSMap,* or guest use of RDPGPR or WRPGPR always causes a GPSI. Root software would return emulated $SRSCtl_{HSS}$=0 on guest read of *SRSCtl* to indicate that no shadow sets are present in guest context. Hardware would return $SRSCtl_{HSS}$ =0 on root read of guest *SRSCtl,* while root writes to guest *SRSCtl* are ignored.

   Guest is provided $Root.SRSCtl_{CSS}$ as its set of GPRs.

3. Shadow sets are implemented in root context, and virtually shared between root and guest. In this case, guest *SRSCtl* and *SRSMap* must be present in guest CP0 context. Guest access to *SRSCtl and SRSMap* will cause GPSI to prevent guest from defining writeable *SRSCtl* fields specifically $SRSCtl_{ESS/PSS}$. Guest use of RDPGPR or WRPGPR will not cause a GPSI as these instructions refer to guest $SRSCtl_{PSS}$ which is writeable only by root - guest writes to $SRSCtl_{PSS}$ always cause a GPSI.

The case where Shadow Sets are implemented in guest context is not discussed in this section - it is not recommended due to the overhead of guest context save and restore of Shadow Sets. A mechanism of virtual sharing of a unique set of Shadow Sets amongst guests is thus not provided.

In the case of virtual sharing, the read-only field guest $SRSCtl_{HSS}$ must be writeable by root. This allows root software to set the total number of Shadow Set available to guest, which is equal to guest $SRSCtl_{HSS}$. The Lowest Shadow Set is specified by $GuestCtl3_{GLSS}$. Guest use will always assume $GuestCtl3_{GLSS}$ to $GuestCtl3_{GLSS}$ plus Guest $SRSCtl_{HSS}$ physical Shadow Sets as available to the guest. Root can write Guest $SRSCtl_{ESS/PSS}$ with (D)MTGC0 instructions.

A non-zero $GuestCtl3_{GLSS}$ is useful if a large number of Shadow Sets are implemented and can be physically partitioned among guests and root. Prior to guest entry, root would write $GuestCtl3_{GLSS}$ and guest $SRSCtl_{HSS}$ to define the continuous range of Shadow Sets available to the guest. This range should be non-overlapping with any other guests and root's range to avoid the overhead of save and restore. Root would also write Guest $SRSCtl_{ESS/PSS}$. Root may also choose to write guest $SRSCtl_{EICSS}$, taking the example of an EIC (External Interrupt Controller) interrupt. In this case, root would read $GuestCtl1_{EID}$ then write this value to $SRSCtl_{EICSS}$. unless hardware implements the transfer itself, as described in Section 4.8.1.2 .

Hardware must offset $SRSCtl_{ESS/PSS}$ by $GuestCtl3_{GLSS}$ before access of corresponding Shadow Set for guest. Similarly, the EIC, if supported, would drive a virtual EICSS. The virtual EICSS is registered and offset similarly before use.

A zero (default) $GuestCtl3_{GLSS}$ is useful is there are few Shadow Sets. Root may allocate one set for all guests, and one set for root. Any switch between guests would require a save and restore of the related Shadow Set.

Guest $SRSCtl_{EICSS}$ is set by EIC. EIC must be root managed since it is a shared resource and thus access must be virtualized amongst guests. Guest $SRSCtl_{EICSS}$ must always fall in guest range of Shadow Sets.

4.9.1.1 Pseudo-code for Shadow Set Handling

The pseudo-code below uses the logical term GSRSEn specifically to indicate whether Shadow Sets are available in guest context.

```
GSRSEn ← (Guest.SRSCtl.HSS > 0) ? 1 : 0;
```

Guest Shadow Sets are thus available if Shadow Sets are implemented in guest context (not recommended), or virtually-shared between root and guest (case 3).

Determination of Current and Previous Shadow Sets:

```
// Mode-specific CSS
Current_Shadow_Set ($SRSCtl_{CSS}$) ←
        guest_mode and GSRSEn ? Guest.$SRSCtl_{CSS}$ + $GuestCtl3_{GLSS}$ : Root.$SRSCtl_{CSS}$ ;
```

In the case where the processor is in guest mode and GRSEn=0 (e.g., case 2), guest will share $Root.SRSCtl_{CSS}$ Shadow Set with root .

```
// Mode-specific PSS, effective for RDPGPR/WRPGPR.
Previous_Shadow_Set (SRSCtl_PSS) ←
        guest_mode and GSRSEn  ? Guest.SRSCtl_PSS + GuestCtl3_GLSS :
        guest_mode and not GSRSEn ? <GPSI> : Root.SRSCtl_PSS ;
```

In the case where the processor is in guest mode and GRSEn=0 (e.g., case 2), guest use of RDPGPR/WRPGPR will cause a GPSI.

Events that update Root or Guest PSS and CSS:

Exception taken in root mode

```
Root.SRSCtl_PSS ← Root.SRSCtl_CSS;
Root.SRSCtl_CSS ← Root.SRSCtl_ESS/EICSS or Root.SRSMap_SSVx
```

This behavior is also applicable to exceptions taken in guest mode that cause a guest-exit to root mode.

Exception taken in guest mode, with GSRSEn = 1

```
Guest.SRSCtl_PSS ← Guest.SRSCtl_CSS
Guest.SRSCtl_CSS ← Guest.SRSCtl_ESS/EICSS or Guest.SRSMap_SSVx
```

In this case that the exception originates and is taken in guest mode.

Exception taken in guest mode, with GSRSEn = 0

```
Not Applicable.
```

ERET executed in root mode

```
Root.SRSCtl_CSS ← Root.SRSCtl_PSS
```

This is applicable to an exception taken in root mode, or an exception that causes a guest-exit to root mode.

ERET executed in guest mode, with GSRSEn=1:

```
Guest.SRSCtl_CSS ← Guest.SRSCtl_PSS
```

ERET executed in guest mode, with GSRSEn=0:

```
Not Applicable.
```

4.9.2 Multiplier Result Registers

The guest and root contexts share the multiplier result registers *LO* and *HI*.

4.9.3 DSP Module

The guest and root contexts share the DSP Module, if it is implemented. The DSP Module is available to the guest context when *Guest.Config3*$_{DSPP}$=1.

During guest mode execution, access to the DSP Module is controlled by the *$Status_{MX}$* bits from both the root and guest contexts. The DSP/MDMX enable bit *$Guest.Status_{MX}$* is checked first. If access is not granted, a DSP Module state unusable exception is taken in guest mode.

The *$Root.Status_{MX}$* bit is checked next. If access is not granted by the *$Root.Status_{MX}$* bit, a DSP Module state unusable exception is taken in root mode.

Root has the ability to deconfigure DSP resources in guest context by writing *$Config3_{DSPP}$* and *$Config3_{DSP2P}$* as given in Table 4.10. The writeable state of Guest.*$Status_{MX}$*, as visible in guest context, is dependent on Guest.*$Config3_{DSPP}$* only. An implementation may choose to limit root writeability to Guest.*$Config3_{DSPP}$* as selective enabling of DSP and DSP Revision 2 is not recommended in implementations. As a consequence of deconfiguration either all DSP resources are available to guest or none.

4.9.4 Floating Point Unit (Coprocessor 1)

The guest and root contexts share the Floating Point Unit, if it is implemented. 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.

4.9.5 Coprocessor 2

The guest and root contexts share coprocessor 2, if it is implemented. Coprocessor 2 is available to the guest context when *$Guest.Config1_{C2}$*=1.

During guest mode execution, access to the coprocessor 2 is controlled by the *$Status_{CU2}$* bits from both the root and guest contexts. The coprocessor enable bit *$Guest.Status_{CU2}$* is checked first. If access is not granted, a coprocessor unusable exception is taken in guest mode.

The *$Root.Status_{CU2}$* bit is checked next. If access is not granted by the *$Root.Status_{CU2}$* bit, a coprocessor unusable exception is taken in root mode.

4.9.6 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.

4.10 Combining the Virtualization Module and the MT Module

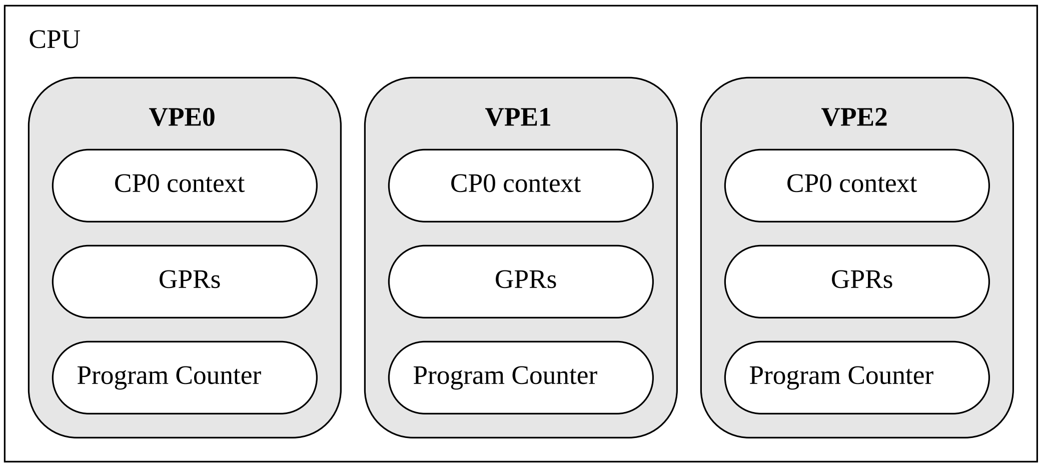
The MIPS MT Module defines a set of instructions and machine state which are used to implement multithreading. The presence of the MT Module is indicated by the *$Config3_{MT}$* field.

Like the Virtualization Module, the MT Module provides duplicate Coprocessor 0 state. A single MIPS CPU can contain multiple Virtual Processing Elements (VPEs). Each of these VPEs uses a separate set of general purpose registers (GPRs), and a separate CP0 context. Mechanisms for controlling one VPE from another are provided, to allow for system initialization and control.

Each VPE runs a separate and independent program - a ‘thread’. Switching between VPEs happens very rapidly - for example switching to a different VPEs on each cycle.

When used in a Symmetric Multi-Processing (SMP) configuration, the MT Module allows a single CPU core to appear to software as multiple CPU cores which are simultaneously executing, using the same physical address space accessed through a common set of L1 caches.

Figure 4.10 A MT Module processor equipped with three VPEs

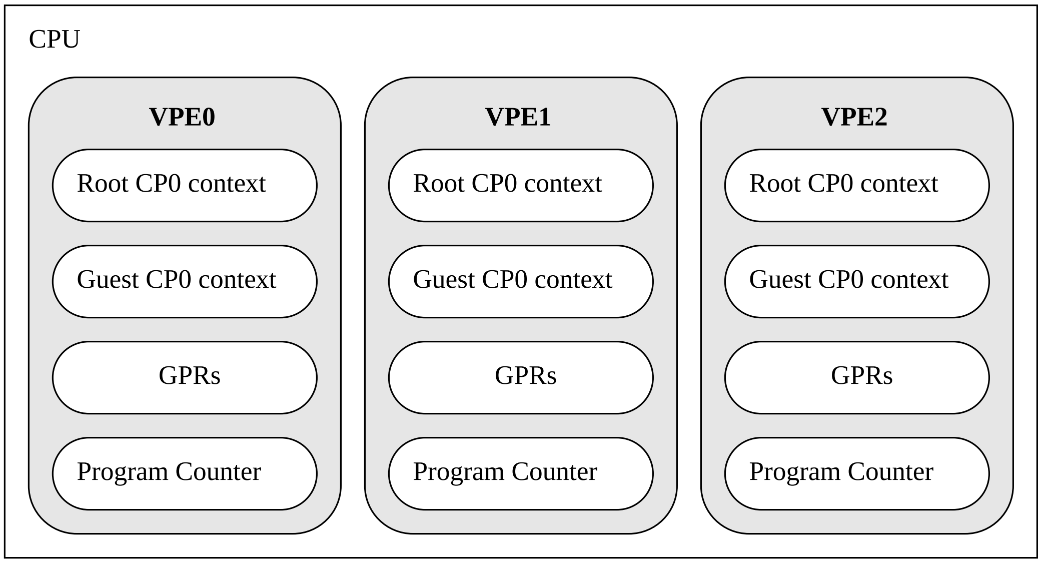


The Virtualization Module enables virtualization for a single thread of execution. Multiple CP0 contexts are present (guest and root), but general purpose registers (GPRs) and coprocessor registers are shared. A single thread of execution covers the hypervisor software, guest kernel software, and guest-user software.

The Virtualization Module and MT Module can co-exist in the same processor. Each VPE is treated like a separate processor - the pre-existing machine state of each VPE is accessible to root mode, and the new guest mode and guest CP0 context are added. In such a machine, *$Root.Config3_{MT}$*=1 and *$Root.Config3_{VZ}$*=1.

Figure 4.10 shows a MT Module processor equipped with three VPEs and the Virtualization Module.

Figure 4.11 A MT Module processor equipped with three VPEs and the Virtualization Module



The 'onion model' would in theory allow a processor to be built which would incorporate MT Module state and instructions within the guest context (*Guest.Config3*$_{MT}$=1), but this is not recommended. The guest context of a realistic machine will not contain the MT Module - hence *Guest.Config3*$_{MT}$=0. When G*uest.Config3*$_{MT}$=0, then (D)MTC0 and (D)MFC0 of MT Module CP0 registers are UNPREDICTABLE and attempts to execute MT Module instructions result in a Reserved Instruction exception in Guest context.

Hypervisor software running on each VPE manages the thread of execution for that VPE - as in a multi-core system. The hypervisor software controls the physical address space and privileges of each guest - for example whether the VPEs share a common physical address space (e.g. a SMP machine), or are configured to be entirely separate.

A trap-and-emulate approach is required for full virtualization of a guest which uses the MT Module (though this is not recommended). MT Module registers are never present in Guest CP0 context, even if the intent is to emulate. Root would write *Guest.Config3*$_{MT}$=1 to simulate presence of MT Module in guest context. Any guest-kernel access to MT Module registers, guest use of MT instructions will trigger a Guest Privileged Sensitive Instruction exception.

When multiple guest virtual machines are running on a single-threaded machine, switches between guests occur tens, hundreds or thousands of times per second. When a context switch takes place the outgoing guest's machine state is read out and saved, and the incoming guest's machine state is loaded and restored. The processor is controlled by one hypervisor instance, which is in control of the root context.

When multiple guest virtual machines are running on a multi-core machine, switches between guests on each core may still occur tens or hundreds of times per second, using the context switch method. However, multiple guests can be run simultaneously - one on each processor core. A distinct hypervisor instance on each processor is in control of that processor's root context - these hypervisor instances communicate to achieve shared goals, as in a traditional SMP system.

A similar arrangement is used when multiple guest virtual machines are running on a single-core multi-threaded machine. Switches between guests are achieved on a cycle-by-cycle basis - as the processor switches between VPEs. Multiple guests can run simultaneously - one on each VPE. A distinct hypervisor instance on each VPE is in control of that VPE's root context.

This concept can be further extended to a multi-threaded, multi-core machine. Each processor core features multiple VPEs, each of which has its own guest context. A distinct hypervisor instance is present on each VPE and in control of the root context.

The MT Module and Virtualization Module provide complementary feature sets, which allow hypervisor software the flexibility to schedule guest virtual machines on separate cores, on separate VPEs, and to schedule using traditional time-sharing methods.

4.11 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 4.1, The Debug privileged execution mode exists in the root context. A processor supporting virtualization operates in two contexts, Root and Guest. Within Guest, there are three privileged execution modes; kernel, supervisor and user, and in Root context, there are four; kernel, supervisor, user and debug.

Table 4.15 lists debug features and their application to the Virtualization Module.

Table 4.15 Debug Features and Application to Virtualization Module

Feature	Description	Reference
Debug mode	Guest mode is mutually exclusive with Debug mode. When in Debug mode (*Debug*$_{DM}$=1), the processor is not in guest mode.	Section 4.4.3 "Definition of Guest Mode"
	When the processor is running in Debug mode, it has full access to all resources that are available to Root-Kernel mode operation.	MIPS EJTAG Specification. Section 7.2.3 - Debug Mode Handling of Processor Resources
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 MIPS32 and EJTAG specifications.	MIPS EJTAG Specification. Section 7.2.2 - Debug Mode Address Space
Access to guest CP0 context	Debug tools access general purpose registers (GPRs) and coprocessor registers by executing instructions in the processor pipeline. Access to the guest CP0 context must use the Virtualization Module instructions provided to transfer data between the root and guest contexts - MTGC0 and MFGC0. Accesses to the guest TLB must use the instructions provided to initiate guest TLB operations from the root context - TLBGP, TLBGR, TLBGWI, TLBGWR. These operations are used to transfer data between the guest TLB and the guest CP0 context. When accessing the guest TLB in debug mode, a two-step process is required - to transfer data to/from the guest CP0 context and guest TLB, and to transfer data to/from the root CP0 context and guest CP0 context.	Section 4.6.2
Hardware Breakpoints	When implemented, hardware breakpoints are part of the root context. The root context remains active during guest mode execution, allowing hardware breakpoints to be used to debug guest software. Exceptions resulting from hardware breakpoints are of type Synchronous Debug or Asynchronous Debug. In both cases, the exceptions are handled in Debug mode.	Section 4.7.4

Table 4.15 Debug Features and Application to Virtualization Module

Feature	Description	Reference
Watch registers	Support for use of watchpoint from the Guest is optionally provided.	Refer to Section 4.12 “Watchpoint Debug Support”

4.12 Watchpoint Debug Support

Root and Guest Watchpoint debug support is provided by Coprocessor 0 *WatchHi* and *WatchLo* register pair(s). These registers are present in Root if Root $Config1_{WR}$=1 and in Guest if Guest $Config1_{WR}$=1 .

An implementation may choose to provide no Watch register support, Root-only Watch register support, or Root and Guest Watch register support.

In Table 4.16, the state of Guest $Config1_{WR}$. conveys what support is available to Guest.

Table 4.16 Guest Watchpoint Support

Guest $Config1_{WR}$ Value	R/W State	Function
0	R	No Guest Watch registers.
1	R	Guest Watch registers present.
0/1	R (Guest) R/W (Root)	Virtual Guest Watch support provided.

Root-only Watch registers (Root $Config1_{WR}$=1 and Guest $Config1_{WR}$=0) allows for Root Watch of Root Virtual Addresses (RVA), and optionally Guest Physical Addresses (GPA). Root Watch of GPA in this configuration is enabled through Root $WatchHi_{WM[0]}$.

If both Root and Guest Watch registers are present (Guest $Config1_{WR}$=1), then Root and Guest Watch will operate independently. Watch exceptions detected on match will be taken in respective modes.

The Virtualization Debug definition also allows for virtual Guest Watch via Root Watch registers (Guest $Config1_{WR}$=0/1). This feature is optional. Root Software can test R/W state of Guest $Config1_{WR}$ to determine whether virtual Guest Watch registers are supported.

Table 4.17 Watch Control

Guest $Config1_{WR}$ Value (in R/W State)	Root $WatchHi_{WM[1:0]}$	Function	Guest Exception on Access	Guest Exception on Match	Root Exception
0	X0	Root Watch RVA	UNPREDICTABLE	None	Watch
0	X1	Root Watch GPA (optional)	UNPREDICTABLE	None	Watch
1	00	Root Watch RVA	GPSI	None	Watch
1	01	Root Watch GPA (optional)	GPSI	None	Watch
1	10	Guest Watch GVA	None	Watch	None
1	11	Reserved	-	-	-

There is no support for Root emulation of Guest watch registers. Root emulation of Guest watch registers would require that every guest read and write trap to Root. In sharing mode, once a watch register pair is assigned to Guest, Guest can setup registers without Root intervention.

Referring to Table 4.17, if Guest $Config1_{WR}$=0, then no watch register pairs are enabled for Guest watch. A Guest access will be treated as as UNPREDICTABLE. Recommended implementations may either no-op both MTC0 and MFC0, trap to Root software with a GPSI, or no-op an MTC0 and return 0s on MFC0. If Guest $Config1_{WR}$=1, then a Guest access is treated normally except a MTC0 cannot modify $WatchHi_{WM}$, and an MFC0 will return 0s for $WatchHi_{WM.}$

If Guest $Config1_{WR}$=1, then selected Root Watch register pairs are enabled for Root or Guest watch. Referring to Table 4.17, this is determined by Root $WatchHi_{WM}$[1]. Root $WatchHi_{WM}$[0] determines whether Root is watching RVA or GPA. Root Watch of GPA is optional. If not supported, then a write of 1 to Root $WatchHi_{WM}$[1:0], will write 0, defaulting to RVA watch.

If under Guest control, Guest can only watch GVA. A write of 3 to Root $WatchHi_{WM}$[1:0], will write 2 in this configuration, defaulting to GVA watch. Root can take away privilege from Guest at any time by writing to Root Watch registers. Root access will thus not take an exception on access of a shared pair of registers under Guest control. If under Root control with Root $WatchHi_{WM}$[1]=0 then a Guest access will result in a GPSI. Root may choose to assign this register pair to Guest at this point, or return to the guest instruction following the move.

Guest watch is enabled strictly in guest mode as defined by the equation:

$$(Root.GuestCtl0_{GM} = 1 \text{ and } Root.Status_{EXL} = 0 \text{ and } Root.Status_{ERL} = 0 \text{ and } Root.Debug_{DM} = 0)$$

There is no facility for Guest to watch addresses related to Root intervention events. That is, events occuring when the following equation is true:

$$(Root.GuestCtl0_{GM} = 1 \text{ and } (Root.Status_{EXL} = 1 \text{ or } Root.Status_{ERL} = 1 \text{ or } Root.Debug_{DM} = 1))$$

In an implementation that supports virtual sharing between Root and Guest, Root software may choose to assign all *WatchHi* and *WatchLo* to Guest or Root, but not both. This is the recommended policy for sharing between Root and Guest. If assigned to Guest then Guest access will not cause a GPSI exception.

Alternatively, an implementation may optionally choose to assign a subset of the total *Watch* register pairs in Root CP0 context to Guest for simultaneous use by Guest and Root. Read of guest $WatchHi(N)_M$ should return root $WatchHi(N+1)_{WM[1]}$ to indicate to guest software that root *WatchLo/Hi(N+1)* is owned by guest. If all pairs are allocated to guest, then read by guest of the M bit in the last register pair should return 0. Initial access by guest to the Watch registers will result in a GPSI exception, allowing Root to configure *Watch* registers for guest use. *Watch* register pairs assigned to Guest in this manner must be a contiguous range, starting from the least significant pair. It is further assumed that the allotment of *Watch* registers to a guest is not dynamic - once established after initial guest access (which caused GPSI) or on guest configuration by root software, then the allotment must remain as such for duration of guest operation.

4.13 Virtualization Module features and Hypervisor Software

The Virtualization Module provides many features which are intended as optimizations to reduce the number of hypervisor traps required, and to reduce the length of each hypervisor intervention.

Table 4.18 describes an outline of the design intent of each feature, and how it is expected to be used in a virtualized system. It is intended to be treated as a guideline, and does not aim to specify how software should be implemented.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
Guest mode	The Guest Mode allows for a “limited privilege” kernel mode, in addition to the existing modes within the MIPS32 Privileged Resource Architecture. The separation of privileges between user and kernel modes is duplicated in guest mode, through the use of the guest-user and guest-kernel modes. This is intended to minimize virtualization overhead on mode transitions within a guest. A separation is introduced between the existing full-privilege kernel mode and the limited-privilege guest-kernel mode. This enables a hypervisor to selectively grant access to system resources through emulation, address translation or by granting direct access.
Separate Guest CP0 context	A partial CP0 context is provided for use when in guest mode. The guest CP0 context includes registers for processor status, exception state and timer access. Depending on the options chosen by the implementation, the guest CP0 context can also include registers to control segmentation and hardware page table walking within the guest context. The separate CP0 context for the guest reduces the context switch overhead when transitioning between root and guest modes. An interrupt or exception causing an exit from guest mode can be immediately handled using the original (root) CP0 context without additional context switching. The guest CP0 context is partially populated. Guest accesses to registers which are not included can be emulated by hypervisor handling of guest exceptions. The registers chosen to be included in the guest CP0 context are either necessary to control guest mode operation, or are so frequently accessed by guest kernels that trap-and-emulate is impractical.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
Simultaneously active guest and root CP0 contexts	During guest mode execution the guest CP0 context is used, but the original (root) CP0 context remains active. This permits an ‘onion model’ whereby guest activities are first checked against the guest CP0 context, and then against the root CP0 context. Exceptions are taken in the mode whose context triggered the exception. Systems controlled by the root CP0 context continue operating during guest mode execution. This includes CP0-controlled systems such as performance counters and breakpoints. It also includes logic which detects external interrupts and serious exceptions such as NMI, Bus Error or Cache Error. The onion model allows the pre-existing programming interface for these systems to be retained, and for their continued operation during guest mode execution. The addition of the guest-mode CP0 context allows an inner layer of systems to be used by the guest without hypervisor intervention. For example, the guest interrupt, timekeeping and address translation systems can be programmed and maintained by the guest kernel. Since these systems are active only during guest mode execution, and the pre-existing root-context systems remain active, little hypervisor intervention is required, as the guest cannot inflict damage to the root. When an exception returns control to root mode during guest mode execution, the guest context is immediately disabled. No context switch is required. The presence of two separate contexts allows for an immediate entry to the root-mode exception handler, using the root-mode exception state. On exit, an immediate return to the guest is possible. No time-consuming memory accesses for context switch are required. Following the rules of the ‘onion model’, access to coprocessors must be enabled by both the guest and original CP0 contexts. This allows for lazy context switch of coprocessors (for example, the floating point unit) when switching between guests.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
Dual-level address translation and guest TLB	In a fully virtualized system, the 'onion model' is applied to address translation. Memory accesses from the guest are translated using the guest context Segment Configurations and the guest context TLB. Exceptions or TLB refills resulting from this translation step are handled by the guest. The result is a 'guest physical' address (GPA). The root TLB (the original TLB) is used to perform a second level of translation - from the 'guest physical' address to a machine physical address. Exceptions or TLB refills resulting from this translation step are handled by the hypervisor, using the pre-existing TLB exceptions, or the new hardware page table walking system. This arrangement allows the guest kernel to maintain its own page tables which map guest-virtual to guest-physical addresses. The guest kernel can handle TLB refills and other exceptions without hypervisor intervention. The hypervisor maintains a separate page table which maps guest-physical addresses to machine physical addresses. The hypervisor is not required to parse or otherwise interpret the guest page tables, or to maintain a page table on behalf of the guest. No hypervisor knowledge of guest-virtual addresses is required. The two translation systems operate independently, greatly simplying the software architecture. Despite the two levels of translation, hardware implementations ensure that each memory access is translated only once within processor pipeline stages. This is done by dynamically creating single-level translations which combine the translations held within both guest and root TLBs. If the root TLB and guest TLB use the same page size, a guest TLB refill is likely to require a root TLB refill. When the root TLB uses page sizes larger than those used by the guest operating system, the number of root TLB refills can be reduced.
Guest context *Config*$_{0-7}$ registers	The guest context includes its own set of *Config*$_{0-7}$ registers. These are used for two purposes within a virtualized system. The first purpose is to indicate to hypervisor software how the guest context is configured in the particular hardware implementation. For example the hypervisor can determine the size of the guest TLB, and which optional features are included. The second purpose is for the hypervisor software to indicate to the hardware implementation how the guest context should behave. Hardware implementations can choose to allow writes to fields within guest context *Config*$_{0-7}$ registers. This allows the hypervisor to enable or disable certain architectural features, or to change the virtual machine behavior seen by the guest. The guest *Config*$_{0-7}$ register are primarily intended for use by hypervisor software, but access by guest kernels can be enabled. Given the infrequent access to *Config*$_{0-7}$ registers, it is likely that a hypervisor would choose to trap and emulate guest accesses.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
Interrupt delivery to guests	Global and individual interrupt enables are included in the guest context, along with interrupt-pending signals. Interrupt handlers are located at the standard entry points within the guest address space, or controlled by the guest context exception base register. Hypervisor software can deliver interrupts to a guest by writing the interrupt pending bits within the guest context. The hypervisor can enable immediate delivery of an external interrupt to a guest through direct assignment (pending interrupt passthrough). Guest kernels can implement critical regions using the normal interrupt enable/disable mechanisms, thus holding off delivery of interrupts to the guest context. External interrupts controlled by the root context cause an immediate exit from guest mode, returning control to a hypervisor interrupt handler. The guest cannot hold off these interrupts, as they are controlled by the root context.
Guest Timer system	Hypervisor software needs to control the passage of time as viewed by a guest. Guests need an efficient method to set up timer interrupts without incurring drift. The hypervisor can set a control bit to which allows a guest to read from the timer's *Count* register, and allows the guest to set up timer interrupts with the *Compare* register. The timer value seen by the guest is created by adding an offset to the real timer value, stored in *Root.GTOffset*. The guest does not have direct write access to its timer value - writes must be trapped and emulated by the hypervisor. It may be necessary for a hypervisor to disallow guest timer access when emulation is required. This may be the case if a guest kernel is booted on a system with one timer clock frequency, and is subsequently required to be re-scheduled on a core with a different timer clock frequency.
Secure, unique TLB entries based on GuestID.	An optional GuestID feature provides a Root programmable unique identifier for use in TLB entries eliminating the requirement for invalidation of TLB entries on virtual machine context switch. Refer to documentation on $GuestCtl1_{ID}$ and $GuestCtl1_{RID}$ fields in Section 5.3 "GuestCtl1 Register (CP0 Register 10, Select 4)".
Root control of Guest TLB mapping and Guest TLB resources. 1) mapping using Guest TLB 2) Guest TLB instructions/registers - $GuestCtl0_{AT}$	The $GuestCtl0_{AT}$ field provides control for whether the guest may use the privileged registers and instructions related to the MMU. This allows the situation where the guest TLB and Segmentation Control is part of the address translation, but any guest access to the control registers results in an exception ($GuestCtl0_{AT}$=1). This can be used both for hypervisor control and to debug guest behavior.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
Guest Software Field Change exceptions	The Guest Software Field Change exception system allows for hypervisor intervention before certain guest-context register fields are changed. The exception is taken prior to execution of the instruction which would have modified the field. Some guest register fields are implemented which correspond to fields in the root CP0 context, but are not actually connected to hardware. An example is the “reduced power” control bit *Status*$_{RP}$. When the guest kernel changes the value of such a field, it is expecting some change of behavior in the virtual machine. The field-change exception allows the hypervisor to respond appropriately. In other cases (e.g., *Cause*$_{IV}$) the field change would affect guest execution, but hypervisor intervention may be required in order to set up some other aspect of the virtual machine - for the example given, changes may be required to how external interrupts are passed to the guest.
Guest Hardware Field Change exception	The Guest Hardware Field Change exception is related to the Guest Software Field Change exception. It is used to trigger hypervisor intervention on a hardware initiated field change within a guest. This mechanism can be used for debug, security or emulation purposes by the hypervisor.
Guest Privileged Sensitive Instruction exceptions	The guest kernel mode is a limited privilege mode. The Guest Privileged Sensitive Instruction exception is the primary mechanism by which the hypervisor traps privileged instructions executed in guest mode. It can be used for emulation of non-existent CP0 registers, and emulation of accesses to registers which have been disabled by the hypervisor. The hypervisor is provided with a catch-all mechanism to trap on all guest privileged operations (*GuestCtl0*$_{CP0}$), and a number of more targeted enables. These targeted enables include fields to control access to guest address translation (*GuestCtl0*$_{AT}$), the guest timer (*GuestCtl0*$_{GT}$), limited cache operations (*GuestCtl0*$_{CG}$), and the *Config*$_{0-7}$ registers present in the guest context (*GuestCtl0*$_{CF}$). The ability to control access to these features allows the hypervisor to restrict guest permissions, or to emulate the hardware behavior expected by a guest - for example different *Config*$_{0-7}$ registers than are present in the machine.
Guest Reserved Instruction Redirect exception	A control bit is provided (*GuestCtl0*$_{RI}$) which allows guest RI exceptions to be redirected to hypervisor software. This enables emulation of instructions which are not available in the guest context.
New privileged instruction HYPCALL	A new instruction is provided, specifically to allow guest kernels to make API calls to the hypervisor software. This can be used from both guest-kernel and root-kernel modes.

Table 4.18 Virtualization Optimizations and their Intended Purpose

Virtualization Optimization	Description
New privileged instructions MFGC0, MTGC0 TLBGINV, TLBGINVF, TLBGR, TLBGWI, TLBGP, TLBGWR	New instructions are provided to allow access to the guest CP0 context for hypervisor software running in root mode. These instructions also provide access to the guest CP0 context for instructions executed in Debug mode, provided by the EJTAG debug system. The instructions MFGC0 and MTGC0 allow data to be transferred between general purpose registers (GPRs) and guest CP0 context registers. The instructions TLBGINV, TLBGINVF, TLBGP, TLBGR, TLBGWI and TLBGWR are used from root mode to access the guest context TLB using the TLB registers located in the guest context.

4.14 Lightweight Virtualization

4.14.1 Introduction

The Virtualization architecture provides support for a lightweight implementation. The focus of such an implementation is to reduce implementation cost and feature complexity. The added benefit of reduced feature complexity is that root software is simplified to the point where it need not be a complete hypervisor. For example, it may handle guest interrupts, guest exceptions and related context switching, but it wouldn't provide support for an added level of guest translation.

The lightweight virtualization specification may also support a different class of embedded applications. For example, where a Root Protection Unit (RPU) is used, the guests are not different OSes, but applications within an OS, where the applications are from different vendors who do not trust each other. Virtualization in this case has been extended to secure embedded applications.

4.14.2 Support for Lightweight Virtualization

4.14.2.1 Root Protection Unit (RPU)

The RPU is a defeatured Root TLB that does not translate a guest physical address to a root physical address, and thus does not require storage for root physical address. Instead it assumes that the guest physical address is identity mapped to physical memory. However, the RPU checks the guest physical address on a page basis, where the page is programmed by root software. If the page matches, then the guest has access to related physical memory. Otherwise the access will trap to root software, using standard exceptions.

The RPU and its software interface support all instructions and COP0 registers of the baseline architecture and extensions provided in the Virtualization Module. Root *EntryLo0* and *EntryLo1* PFN fields are assumed read-only as 0 since the RPU does not translate guest physical addresses. Similarly, the CCA(Cache Coherency Attribute) field is not supported. This field in *EntryLo0* and *EntryLo1* is read-only as 0 in hardware.

The RPU supports XI(Execute-Inhibit), RI(Read-Inhibit) along with D(Dirty) page attributes which are mandatory in an RPU implementation.

An RPU will support multiple page-sizes, though it is implementation dependent in the baseline architecture as to which page sizes are supported.

The RPU is only supported in a configuration with a root FMT (Fixed Mapping Table). Any addresses in root mode must use the Root FMT. Any guest addresses go through the guest FMT or TLB, and RPU.

An RPU is present in an implementation that supports virtualization (Root.$Config3_{VZ}$=1) and has a root FMT (Root.$Config_{MT}$=3). It is thus possible for the guest MMU to support a guest TLB with an RPU.

Refer to Table 4.19 for possible MMU configurations with an RPU.

Table 4.19 MMU Configurations with RPU

Guest Logical Address Translation		Root Logical Address Translation
1st Pass	2nd Pass	
FMT	RPU	FMT

Table 4.19 MMU Configurations with RPU

Guest Logical Address Translation		Root Logical Address Translation
1st Pass	2nd Pass	
TLB	RPU	FMT

4.14.2.2 Architectural Control

Additional software visible control has been added for lightweight virtualization.

1. *GuestCtl0Ext*$_{FCD}$

   This field disables hardware generation of Guest Hardware Field Change Exception, and Guest Software Field Change Exceptions. Consequently, root software does not need to support related exception handlers.

   See Section 5.6 for reference.

2. *GuestCtl3*$_{GLSS}$

   This field allows virtualization Shadow Set allocation among guests. This root managed field provides the lowest shadow set allocated to a guest, with the upper bounds provided by root-writeable Guest.*SRSCtl*$_{HSS}$. The context switch penalty is minimized as root need only write *GuestCtl3*$_{GLSS}$ when entering a new guest.

   See Section 5.5 and Section 4.9.1 for reference.

3. *GuestCtl0Ext*$_{MG,OG,BG}$

   These fields have been introduced to enable GPSI on guest access to specified guest CP0 registers. This is useful for fast guest context switching. In this case, root will save and restore limited guest CP0 registers, but in case the unsaved registers are accessed by guest, then an exception to root will allow root software to save and restore the effected registers opportunistically.

   See Section 5.6 for reference.

4. *GuestCtl2*$_{GRIPL,GEICSS,GVEC}$

   See Section 5.4 and Figure 5.4, for reference for reference.

   In EIC(External Interrupt Controller) mode for interrupt handling, *GuestCtl2* provides the capability of fast guest-to-guest interrupt switching capability. A guest interrupt on the root interrupt bus from the EIC will cause capture of interrupt related state (GRIPL,GEICSS,GVEC) in *GuestCtl2*. Guest entry will subsequently cause hardware to load GRIPL and GEICSS into guest context automatically, and GVEC would be used by the guest interrupt handler directly. The root interrupt handler thus does not have to copy state from *GuestCtl2* to guest context.

   See Section 4.8.1.2 for a description of EIC handling.

4.14.2.3 Optional Features of Virtualization Architecture

Certain features are optional in the virtualization architecture. An implementation may choose to support such features based on the class of applications that the product will support. An example being that an implementation need not support root write of all Configuration fields listed in Table 4.11.

Chapter 5

Coprocessor 0 (CP0) Registers

The Coprocessor 0 (CP0) registers provide the interface between the Instruction Set Architecture (ISA) and the Privileged Resource Architecture (PRA). The CP0 registers that are added or extended by the Virtualization Module are discussed below, with the registers presented in numerical order, first by register number, then by select field number.

5.1 CP0 Register Summary

Table 5.1 lists the CP0 registers affected by the Virtualization Module specification, in numerical order. The individual registers are described later in this document. Registers which are not described here follow the definitions from the MIPS32 Privileged Resource Architecture. The *Sel* column indicates the value to be used in the field of the same name in the MFC0 and MTC0 instructions.

Section 4.6.3 "Guest CP0 registers" describes CP0 register availability in guest mode.

Table 5.1 Virtualization Module Changes to Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Modification	Reference	Compliance Level
12	6	*GuestCtl0*	New Register. Controls guest mode behavior.	Section 5.2	Required
10	4	*GuestCtl1*	New Register. Guest ID	Section 5.3	Optional
10	5	*GuestCtl2*	New Register. Interrupt related	Section 5.4	Optional
10	6	*GuestCtl3*	New Register. GPR Shadow Set related.	Section 5.5	Optional
11	4	*GuestCtl0Ext*	Extension to GuestCtl0	Section 5.6	Optional
12	7	*GTOffset*	New Register. Guest timer offset.	Section 5.7	Required
13	0	*Cause*	Addition of hypervisor cause code.	Section 5.8	Required
16	3	*Config3*	Identifies Virtualization Module feature set.	Section 5.9	Required
19	0	*WatchHi*	Watch Debug.	Section 5.10	Optional
25	0	*PerfCnt*	Performance Counter, adds virtualization support.	Section 5.10	Optional
31	2	*KScratch1*	Required in root context.	-	Required
31	3	*KScratch2*	Required in root context.	-	Required

5.2 GuestCtl0 Register (CP0 Register 12, Select 6)

Compliance Level: *Required* by the Virtualization Module.

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.

The *GuestCtl0* register is instantiated per-VPE in a MT Module processor. This register is added by the Virtualization Module.

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 5.1 shows the format of the Virtualization Module *GuestCtl0* register; Table 5.2 describes the *GuestCtl0* register fields.

Figure 5.1 GuestCtl0 Register Format

31	30	29	28	27 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	0
GM	RI	MC	CP0	AT	GT	CG	CF	G1	Impl	G0E	PT	ASE	PIP	RAD	DRG	G2	GExcCode	SFC2	SFC1

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
GM	31	Guest Mode The processor is in guest mode when GM=1, *Root.Status*$_{EXL}$=0 and *Root.Status*$_{ERL}$=0 and *Root.Debug*$_{DM}$=0.	R/W	0	Required
RI	30	Guest Reserved Instruction Redirect. **Encoding** / **Meaning** 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	Required
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. **Encoding** / **Meaning** 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	Required

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CP0	28	Guest access to coprocessor 0. **Encoding** | **Meaning** 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 list of Guest Privileged Sensitive instructions which trigger a Guest Privileged Sensitive Instruction exception is given in Section 4.7.7 The CP0 bit has no other effect on the operation of coprocessor 0 in guest mode.	R/W	0	Required

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
AT	27:26	Guest Address Translation control. **Encoding** / **Meaning**: 0 – Reserved. 1 – Guest MMU under Root control. Guest and Root MMU both implemented and active in hardware. This mode is optional. 2 – Reserved 3 – Guest MMU under Guest control. Guest and Root MMU both implemented and active in hardware. This mode is required. Guest TLB resources are: • TLB related Instructions - TLBWR, TLBWI, TLBR, TLBP, TLB-INV, 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 (see Table 4.7), will trigger a Guest Privileged Sensitive Instruction exception, allowing Root to control Guest address translation directly. For additional information refer to Table 4.18, Entry: “Root control of Guest TLB mapping and Guest TLB resources.” In default mode ($GuestCtl0_{AT}$=3), the Guest TLB resources are active under Guest control. Refer to Section 4.5 “Virtual Memory” for additional information on guest virtual address translation.	R or R/W if more than default mode implemented.	Implementation defined	Required

Nested table within the AT description:

Encoding	Meaning
0	Reserved.
1	Guest MMU under Root control. Guest and Root MMU both implemented and active in hardware. This mode is optional.
2	Reserved
3	Guest MMU under Guest control. Guest and Root MMU both implemented and active in hardware. This mode is required.

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
GT	25	Timer register access. **Encoding** | **Meaning** 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	Required
CG	24	Cache Instruction Guest-mode enable. If R0, then GPSI exception will always occur. CG as an enable in thuis thus optional. CACHEE is optional in the baseline architecture. **Encoding** | **Meaning** 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.	R0, R/W	0	Optional
CF	23	Config register access. **Encoding** | **Meaning** 0 | Guest-kernel write access to *Config0-7* will trigger 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	Required

Table 5.2 GuestCtl0 Register Field Descriptions

Fields					
Name	Bits	Description	Read / Write	Reset State	Compliance
G1	22	*GuestCtl1* register implemented. Set by hardware. Encoding | Meaning 0 | Unimplemented 1 | Implemented.	R	preset	Required
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	Required
G0E	19	*GuestCtl0Ext* register implemented. Set by hardware. Encoding | Meaning 0 | Unimplemented 1 | Implemented.	R	preset	Required
PT	18	Defines the existence of the Pending Interrupt Passthrough feature. Encoding | Meaning 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. Implementation of the Pending Interrupt Passthrough feature is strongly recommended.	R	preset	Required
ASE	17..16	Reserved for MCU Module Pending Interrupt Passthrough.	0	0	Required for MCU Module; Otherwise Reserved

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PIP	15..10	Pending Interrupt Passthrough. 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. See Section 4.8. **Encoding** / **Meaning** 0: Corresponding interrupt request is not visible in guest context. 1: Corresponding interrupt request is visible in guest context.	R/W R0 if unimple-mented	0	Required
RAD	9	RAD, or “Root ASID Dealias” mode determines the means that a Virtualized MMU implementation uses Root ASID to dealias different contexts. **Encoding** / **Meaning** 0: GuestID used to dealias both Guest and Root TLB entries. 1: Root ASID is used to dealias Root TLB entries, while Guest TLB contains only one context at any given time.	R	0	Required
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. If GuestCtl0$_{DRG}$=1 then GuestCtl0$_{RID}$ must be used. If GuestID for root operation is non-zero, root is in kernel mode, Root.Status$_{EXL,ERL}$=0 and Debug$_{DM}$=0, then all root kernel data accesses are mapped, root SegCtl is ignored and Root TLB CCA is used. Access in root mode by other than kernel will cause an address error. H/W must set G=1 as if the access were for guest. DRG is R0 if only DRG=0 supported, otherwise it must be R/W. **Encoding** / **Meaning** 0: Root software cannot access guest entries directly. 1: Root software can access guest entries directly.	R0, R/W	0	Required

Table 5.2 GuestCtl0 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
G2	7	*GuestCtl2* register implemented. Set by hardware. Encoding / Meaning: 0 — Unimplemented 1 — Implemented.	R	preset	Required
GExc-Code	6..2	Hypervisor exception cause code. Described in Table 5.3. This field is UNDEFINED on a root exception.	R	Undefined	Required
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. Guest software may utilize *Status*$_{\mathrm{CU2}}$ for COP2 specific context switching. Encoding / Meaning: 0 — GSFC exception taken if CU[2] is modified by guest. 1 — GSFC exception not taken if CU[2] modified by guest.	R/W if imple-mented, 0 otherwise	0	Optional
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*$_{\mathrm{CU1}}$ for COP1 specific context switching. Encoding / Meaning: 0 — GSFC exception taken if CU[1] is modified by guest. 1 — GSFC exception not taken if CU[1] modified by guest.	R/W if imple-mented, 0 otherwise.	0	Optional

Table 5.3 describes the cause codes use for GExcCode.

Table 5.3 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

Table 5.3 GuestCtl0 GExcCode values

Exception code value		Mnemonic	Description
Decimal	Hexadecimal		
3	0x03	GRR	Guest Reserved Instruction Redirect. A Reserved Instruction 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.
11 - 31	0xB - 0x1f	-	Reserved

5.3 GuestCtl1 Register (CP0 Register 10, Select 4)

Compliance Level: *Optional* in the Virtualization Module.

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. GuestID value of 0 is reserved for Root.

Section 4.5.1 "Virtualized MMU GuestID Use" provides additional detail on GuestID usage as it applies to instruction and data address translation. Section 4.6.2 "New CP0 Instructions" describes the TLB instructions and their use of GuestID.

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.

The *GuestCtl1* register is instantiated per-VPE in a MT Module processor.

Figure 5.2 shows the format of the Virtualization Module *GuestCtl1* register; Table 5.4 describes the *GuestCtl1* register fields.

Figure 5.2 GuestCtl1 Register Format

31	30	29	28	27	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	0
EID								RID								0								ID							

Table 5.4 GuestCtl1 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
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. If implemented, the field is read-only and set by hardware. If not implemented then must be written as zero; returns zero on read.	R0 or R	0	Optional
RID	23..16	Root control GuestID. Used by root TLB operations, and when *GuestCtl0*$_{DRG}$=1 in root mode.	R/W	0	Required
0	15..8	Must be written as zero; returns zero on read.	R0	0	Reserved
ID	7..0	Guest control GuestID. Identifies resident guest. Applies to guest address translation.	R/W	0	Required

5.4 GuestCtl2 Register (CP0 Register 10, Select 5)

Compliance Level: *Optional* in the Virtualization Module.

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. Refer to Section 4.8.1 “External Interrupts” for a description of interrupt handling in EIC and non-EIC modes.

An implementation that supports the virtual interrupt functionality of *GuestCtl2* is not required to support root writes of *Guest.Cause*$_{IP}$*[7:2]* or *Guest.Cause*$_{RIPL}$ as described in Table 4.11.

GuestCtl2 is present in an implementation if *GuestCtl2*$_{G2}$=1.

The *GuestCtl2* register is instantiated per-VPE in a MT Module processor.

Figure 5.3 shows the format of the Virtualization Module *GuestCtl2* register in non-EIC mode. Table 5.5 describes the non-EIC mode *GuestCtl2* register fields.

Figure 5.4 shows the format of the Virtualization Module *GuestCtl2* register in EIC mode. Table 5.6 describes the EIC mode *GuestCtl2* register fields.

Figure 5.3 GuestCtl2 Register Format for non-EIC mode

31 30	29 28 27 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 0
ASE	HC	0	ASE	VIP	0	Impl

Figure 5.4 GuestCtl2 Register Format for EIC mode

31 30	29 28 27 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 0
ASE	GRIPL	0	GEICSS	0	GVEC

Table 5.5 non-EIC mode GuestCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ASE	31:30	MCU Module extension for HC. Must be written as zero; returns zero on read.	R0	0	Reserved
HC	29..24	Hardware Clear for *GuestCtl2*$_{VIP}$ This set of bits maps one to one to *GuestCtl2*$_{VIP.}$ HC may be bit-wise Read-only or R/W. If a bit is Read-only, then it may be preset to 0 or 1. Similarly, if a bit is R/W, then it may be reset to 0 or 1. The interpretation of 0 or 1 state follows.	R, R/W	0 or 1	Optional

Encoding	Meaning
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 h/w.

		In the case of HC=0, *Guest.Status*$_{IP}$[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. The choice of Read-only vs. R/W is implementation dependent. Root software can write then read field to determine supported configuration.			
0	25..18	Must be written as zero; returns zero on read.	R0	0	Reserved
ASE	17:16	MCU Module extension for VIP. Must be written as zero; returns zero on read.	R0	0	Reserved

Table 5.5 non-EIC mode GuestCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
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 the following manner:	R/W	0	Required

Encoding	Meaning
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

0	9..5	Must be written as zero; returns zero on read.	R0	0	Reserved
Impl	4:0	Implementation. These bits are implementation dependent and not defined by the architecture. If not implemented, they must be written as 0, 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 *GuestCtl2* register, modifies another field, and writes the updated value back to the *GuestCtl2* register.	R/W	0	Required

Table 5.6 EIC mode GuestCtl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ASE	31:30	MCU Module extension for GRIPL. Must be written as zero; returns zero on read.	R0	0	Reserved
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. GRIPL is 10 bits only for an implementation that complies with the MCU Module, otherwise it is 8 bits as in baseline architecture.	R/W	0	Required
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	Required
0	17:16	Must be written as zero; returns zero on read.	R0	0	Reserved
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. Root software can write the field if it needs to modify the EIC value before assigning to guest.	R/W	Undefined	Required

5.5 GuestCtl3 Register (CP0 Register 10, Select 6)

Compliance Level: *Optional* in the Virtualization Module.

The *GuestCtl3* register is optional. It is required only if Shadow GPR Sets are supported, and the Shadow Sets used by a guest are virtual and require mapping to physical Shadow Sets. With this mechanism, a pool of Shadow Sets can be physically shared by partitioning the sets among multiple guests and root, under root control.

Virtual mapping of Guest GPR set(s) is supported if Guest $SRSCtl_{HSS}$ is writeable by root. Presence of *GuestCtl3* can be detected by root software by writing any non-zero value less than or equal to root $SRSCtl_{HSS}$ to Guest $SRSCtl_{HSS}$. If a read returns the value written, then *GuestCtl3* is present.

The *GuestCtl3* register is instantiated per-VPE in a MT Module processor.

Figure 5.3 shows the format of the Virtualization Module *GuestCtl3* register; Table 5.7 describes the *GuestCtl3* register fields.

Figure 5.5 GuestCtl3 Register Format

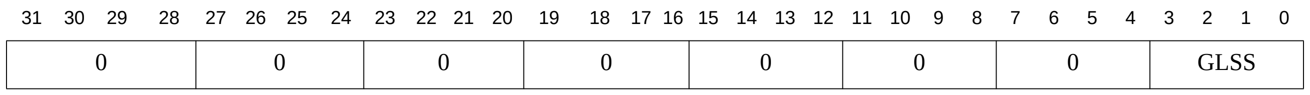


Table 5.7 GuestCtl3 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31:4	This bit must be written as zero, returns zero on read.	R0	0	Reserved
GLSS	3:0	Guest Lowest Shadow Set number. This determines the lowest physical Shadow Set number assigned by root to guest. Guest is thus assigned physical Shadow Sets GLSS to GLSS plus Guest $SRSCtl_{HSS}$. If this field is reserved, then all writes must be zero, and reads should return 0.	R0, R/W	0	Required

5.6 GuestCtl0Ext Register (CP0 Register 11, Select 4)

Compliance Level: *Optional* in the Virtualization Module.

GuestCtl0Ext is an optional extension to *GuestCtl0*. If not supported, the register must read as 0.

GuestCtl0$_{G0E}$ should be read by software to determine if *GuestCtl0Ext* is implemented.

The *GuestCtl0Ext* register is instantiated per-VPE in a MT Module processor.

Figure 5.6 shows the format of the Virtualization Module *GuestCtl0Ext* register; Table 5.8describes the *GuestCtl0Ext* register fields.

Figure 5.6 GuestCtl0Ext Register Format

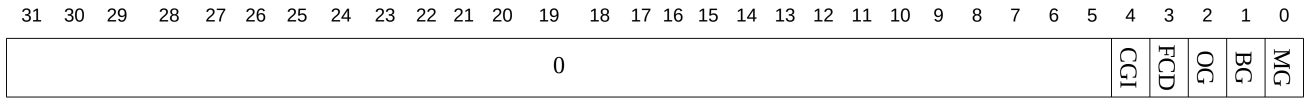


Table 5.8 GuestCtl0Ext Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
0	31:2	This bit must be written as zero, returns zero on read.	R0	0	Reserved
CGI	4	Related to *GuestCtl0*$_{CG}$. Allows execution of CACHE, CACHEE Index Invalidate operations in guest mode. **Encoding** / **Meaning**: 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. This field is R0 if feature is not implemented. The CACHEE instruction is optional in the baseline architecture.	R0, R/W	0	Optional
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, as in the baseline architecture. **Encoding** / **Meaning**: 0 — GSFC or GHFC event will cause exception. 1 — GSFC or GHFC event will not cause exception. This field is R0 if feature is not implemented.	R0, R/W	0	Optional

Table 5.8 GuestCtl0Ext Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
OG	2	Other GPSI Enable. Applies to *UserLocal, WREna, LLAddr, Reserved (for Architecture), UserTraceData1, UserTraceData2, KScratch1* through *KScratch6*, when implemented. If function is not supported, this field reads as 0. **Encoding** **Meaning** 0 GPSI not enabled for these registers unless GuestCtl0$_{CP0}$=0. 1 GPSI enabled for these registers. For a description of Reserved for Architecture registers, see Section 4.6.3.1 . *UserTraceData1, UserTraceData2* are optional CP0 registers defined in MIPS iFlowTrace specification. This field is R0 if feature is not implemented.	R0, R/W	0	Optional
BG	1	Bad register GPSI Enable. Applies to *BadVAddr, BadInstr, BadInstrP* when implemented. If function is not supported, this field reads as 0. **Encoding** **Meaning** 0 GPSI not enabled for these registers unless GuestCtl0$_{CP0}$=0. 1 GPSI enabled for these registers. This field is R0 if feature is not implemented.	R0, R/W	0	Optional
MG	0	MMU GPSI Enable. Applies to *Index, Random, EntryLo0, EntryLo1, Context, ContextConfig, PageMask, EntryHi*. If function is not supported, this field reads as 0. **Encoding** **Meaning** 0 GPSI not enabled for these registers unless GuestCtl0$_{CP0}$=0. 1 GPSI enabled for these registers. This field is R0 if feature is not implemented.	R0, R/W	0	Optional

5.7 GTOffset Register (CP0 Register 12, Select 7)

Compliance Level: *Required* by the Virtualization Module.

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.

The *GTOffset* register is instantiated per-VPE in a MT Module processor. This register is added by the Virtualization Module.

See Section 4.6.8 "Guest Timer".

Figure 5.7 shows the Virtualization Module format of the *GTOffset* register; Table 5.9 describes the *GTOffset* register fields.

Figure 5.7 GTOffset Register Format

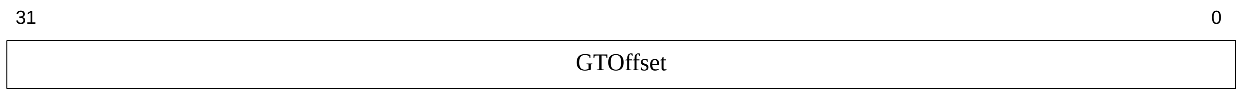


Table 5.9 GTOffset Register Field Descriptions

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

5.8 Cause Register (CP0 Register 13, Select 0)

Compliance Level: *Required* by the Virtualization Module.

As in MIPS32, the *Cause* register describes the cause of the most recent exception, and provides control of software interrupt requests and interrupt vector selection.

The behavior of the Cause register is changed by the Virtualization Module only by the addition of one new cause code.

The *Cause* register is instantiated per-VPE in a MT Module processor.

Figure 5.8 shows the format of the *Cause* register; Table 5.10 describes fields modified by the Virtualization Module.

Figure 5.8 Virtualization Module Cause 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
BD	TI	CE	DC	PCI	0	IV	WP	0	Mod-ule	IP7..IP2 / RIPL	IP1..IP0	0	ExcCode	0

Table 5.10 Cause Register Field Description, modified by Virtualization Module

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ExcCode	6..2	Exception Code - See Table 5.11. Addition of Hypervisor (GE) code.	R	Undefined	Required

Table 5.11 describes the new cause code value defined for ExcCode.

Table 5.11 Cause Register ExcCode values

Exception code value		Mnemonic	Description
Decimal	Hexadecimal		
27	0x1b	GE	Hypervisor Exception (Guest Exit). Hypervisor-intervention exception occurred during guest code execution. GuestCtl0$_{GExcCode}$ contains additional cause information.

5.9 Configuration Register 3 (CP0 Register 16, Select 3)

Compliance Level: *Required* by the Virtualization Module.

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

This register operates as described by the base architecture, except that the VZ field is added.

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.

In Guest context, the VZ field is reserved and read as 0.

Figure 5-9 shows the format of the *Config3* register; Table 5.12 describes the fields added to the *Config3* register by the Virtualization Module.

Figure 5-9 Config3 Register Format

31	30	29	28	27	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	0
M	BPG	CMGCR	MSA	BP	BI	SC	PW	VZ	IPLW		MMAR			MuCon	ISAOnExc	ISA		ULRI	RXI	DSP2P	DSPP	CTXTC	ITL	LPA	VEIC	VInt	SP	CDMM	MT	SM	TL

Table 5.12 Config3 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
VZ	23	MIPS® Virtualization Module implemented. This bit indicates whether the Virtualization Module is present. **Encoding** / **Meaning** 0 / Virtualization Module not implemented 1 / Virtualization Module is implemented	R	Preset (Always 0 in Guest context)	Required

5.10 WatchHi Register (CP0 Register 19)

Compliance Level: *Optional*.

The *WatchHi* register is as defined in the base architecture, except that it has been extended to optionally support watch management in virtualized guest and root contexts.

Figure 5-10 shows the format of the *WatchHi* register; Table 5.13 describes the added *WatchHi* register fields.

The WatchHi register has a 10b wide ASID field only if $Config4_{AE}$=1. Otherwise, the ASID field is 8b wide.

Figure 5-10 WatchHi Register Format

31	30	29 28	27 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	0
M	G	WM	0	ASID	0	Mask	I	R	W

Table 5.13 WatchHi Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
WM	29..28	This field is used for Root management of Watch functionality in an implementation supporting the Virtualization Module. 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. Refer to Section 4.12 "Watchpoint Debug Support"	R/W or R	0	Required (Release 3)

5.11 Performance Counter Register (CP0 Register 25)

Compliance Level: *Optional*.

The *PerfCnt* register(s) are as defined in the base architecture, except that the *EC* field has been added to optionally support performance measurement in virtualized guest and root contexts.

The Control Register associated with each performance counter controls the behavior of the performance counter. Figure 5-11 shows the format of the Performance Counter Control Register; Table 5.14 describes the new Performance Counter Control Register fields.

Figure 5-11 Performance Counter Control Register Format

31	30	29 25	24 23	22 16	15	14 11	10 5	4	3	2	1	0
M	W	Impl	EC	0	PCTD	EventExt	Event	IE	U	S	K	EXL

Table 5.14 New Performance Counter Control Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
EC	24:23	Event Class. Root only. Reserved, read-only 0 in all other contexts. An implementation 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. **Encoding** / **Meaning**: 0 — Root events counted. [default] Active in Root context. 1 — Root intervention events counted, Active in Root context. 2 — Guest events counted. Active in Guest context. 3 — 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 in Root mode. R0 in all others.	0	Optional

5.12 Note on future CP0 features

Implementation note: Addition of a new feature to the root context does not mean that it must be included in the guest context. However, when it becomes necessary to include a new architectural feature in the guest CP0 context, the following rules must be followed.

- A new architectural feature must have a corresponding *Guest.Config* field, which matches the *Root.Config* definition.
- The guest context must always be a subset of the root. No feature can be specified with a *Guest.Config* field which does not also exist in the root.
- It is recommended that the *Guest.Config* field be writable from root mode, to allow the feature to be disabled and become invisible to the guest.
- When the corresponding *Guest.Config* field indicates that a feature is present, it will operate as specified for root mode, and will only use state held in the guest context. The functional behavior of the feature will not be altered by fields in the root context. Timing may be affected.
- Root mode state can only be used to apply translations to the inputs or outputs of the feature, to check for exception conditions within the feature, or to check guest interaction with the feature. The *GuestCtl0* register should be used for single-bit exception-enable bits.
- Hypervisor exceptions can be triggered without the need for a *GuestCtl0* bit, if the exception always results from specified guest-mode interactions with the feature, or specified events within the feature itself. These exceptions will be taken in root mode.
- All memory accesses performed by the feature must be translated under root control. This will be through the root TLB unless another mechanism is provided (e.g. an IOMMU).
- Synchronous exceptions detected by the guest context have a higher priority than the equivalent exception detected by the root context. Synchronous exceptions originate from the ‘inside of the onion’ - the first boundary to be crossed is the guest context, then the root context.
- Asynchronous exceptions detected by the root context have higher priority than the equivalent exception detected by the guest context. Asynchronous exceptions (e.g. interrupts, memory error) originate from ‘outside of the onion’ - the first boundary to be crossed is the root context, and then the guest context.

Chapter 6

Instruction Descriptions

6.1 Overview

The Virtualization Module adds new and modifies existing instructions to allow root-mode access to the guest Coprocessor 0 context and the guest TLB. A new ‘hypercall’ instruction is added, to allow hypervisor calls to be made from guest mode.

Table 6.1 lists in alphabetical order the instructions newly defined or modified by the Virtualization Module.

Table 6.1 New and Modified Instructions

Mnemonic	Instruction	Description	Reference
HYPCALL	Hypercall	Trigger Hypercall exception.	“HYPCALL” on page 124
MFGC0	Move from Guest Coprocessor 0	Read guest coprocessor 0 into GPR.	“MFGC0” on page 125
MTGC0	Move from Guest Coprocessor 0	Write guest coprocessor 0 from GPR.	“MTGC0” on page 127
TLBGINV	Guest TLB Invalidate	Trigger guest TLB invalidate from root mode.	“TLBGINV” on page 128
TLBGINVF	Guest TLB Invalidate Flush	Trigger guest TLB invalidate from root mode.	“TLBGINVF” on page 130
TLBGP	Probe Guest TLB	Trigger guest TLB probe from root mode.	“TLBGP” on page 133
TLBGR	Read Guest TLB	Trigger guest TLB read from root mode.	“TLBGR” on page 136
TLBGWI	Write Guest TLB	Trigger guest TLB write from root mode.	“TLBGWI” on page 138
TLBGWR	Write Guest TLB	Trigger guest TLB write from root mode.	“TLBGWR” on page 140
TLBINV	TLB Invalidate	Modified TLB Invalidate behavior.	“TLBINV” on page 144
TLBINVF	TLB Invalidate Flush	Modified TLB Invalidate Flush behavior.	“TLBINVF” on page 142
TLBP	TLB Probe	Modified TLB probe behavior.	“TLBP” on page 145
TLBR	Read TLB	Modified TLB read behavior.	“TLBR” on page 147
TLBWI	Write TLB, Indexed	Modified indexed TLB write behavior.	“TLBWI” on page 150

Table 6.1 New and Modified Instructions

Mnemonic	Instruction	Description	Reference
TLBWR	Write TLB, Random	Modified random TLB write behavior.	"TLBWR" on page 170

31 26	25	24..21	20 11	10 6	5 0
COP0 010000	CO 1	0000	code	00000	HYPCALL 101000
6	1	4	10	5	6

Format: `HYPCALL` **MIPS32**

Purpose: Hypervisor Call

To cause a Hypercall exception

Description:

A hypervisor call (hypercall) exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as a software parameter. It can be retrieved by the exception handler from the *BadInstr* register, or by loading the contents of the memory word containing the instruction.

Restrictions:

This instruction is available to debug, root kernel and guest kernel modes.

Execution of Hypercall in debug mode is defined, but will not cause a mode transition to root. The processor will stay in debug mode ($Debug_{DM}$=1), and root COP0 state is unmodified.

Refer to MD00047, “EJTAG Specification”, for rules regarding Hypercall exception processing in debug mode. Hypercall exception falls into the category of “Other execution-based exceptions” in EJTAG Section 2.4.1. Debug-DExcCode is set to GE=27 (see Table 5.3), no COP0 state is modified, and other modifications to COP0 Debug state are made according to the rules in EJTAG Section 2.4.3.

Further, if root executes a hypercall in root mode, Root.$Cause_{ExcCode}$ gets set to GE=27 (even though its not a guest-exit) and $GuestCtl0G_{ExcCode}$ is set to HC=2. Root can distinguish a root hypercall from a guest hypercall by looking at $GuestCtl0_{GM}$. If it is set, then the hypercall must have come from a guest, if it is reset, then hypercall must have come from root since Root.$Status_{EXL}$ must have been 0, otherwise hypercall in root mode would not cause an exception.

Execution of hypercall in either root-kernel or debug mode is not recommended.

Operation:

```
if IsCoprocessorEnabled(0) then
   SignalException(HyperCall, 0)
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

HyperCall Exception

Coprocessor Unusable Exception

31 26	25 21	20 16	15 11	10 8	7 3	2 0
COP0 010000	V 00011	rt	rd	000	00000	sel
6	5	5	5	3	5	3

Format: `MFGC0 rt, rd` **MIPS32**
`MFGC0 rt, rd, sel` **MIPS32**

Purpose: Move from Guest Coprocessor 0

To move the contents of a guest coprocessor 0 register to a general register.

Description: `GPR[rt] ← Guest.CPR[0, rd, sel]`

The contents of the guest context coprocessor 0 register specified by the combination of *rd* and *sel* are loaded into general register *rt*. Note that not all guest context coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNDEFINED** if the guest context coprocessor 0 does not contain the register specified by *rd* and *sel*.

The guest context does not implement the Virtualization Module. Use of this instruction in guest-kernel mode will result in a Reserved Instruction exception, taken in guest mode.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
if IsCoprocessorEnabled(0) then
    if (Config3$_{VZ}$ = 0) then
        SignalException(ReservedInstruction, 0)
        break
    endif
    reg = rd
data ← Guest.CPR[0,reg,sel]
    GPR[rt] ← data
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25 21	20 16	15 11	10 8	7 3	2 0
COP0 010000	V 00011	rt	rd	010	00000	sel
6	5	5	5	3	5	3

Format: `MTGC0 rt, rd` **MIPS32**
`MTGC0 rt, rd, sel` **MIPS32**

Purpose: Move to Guest Coprocessor 0

To move the contents of a general register to a guest coprocessor 0 register.

Description: `Guest.CPR[0, rd, sel] ← GPR[rt]`

The contents of general register rt are loaded into the guest context coprocessor 0 register specified by the combination of *rd* and *sel*. Not all guest context coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be set to zero.

Restrictions:

The results are **UNDEFINED** if guest context coprocessor 0 does not contain the register as specified by *rd* and *sel* or the destination register is the *Guest.Count* register, which is read-only

The guest context does not implement the Virtualization Module. Use of this instruction in guest-kernel mode will result in a Reserved Instruction exception, taken in guest mode.

In a 64-bit processor, the MTGC0 instruction writes all 64 bits of register *rt* into the guest context coprocessor register specified by *rd* and *sel* if that register is a 64-bit register.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
if IsCoprocessorEnabled(0) then
   if (Config3$_{VZ}$ = 0) then
      SignalException(ReservedInstruction, 0)
      break
   endif
   data ← GPR[rt]
   reg ← rd
   Guest.CPR[0,reg,sel] ← data
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBGINV 001011
6	1	19	6

Format: `TLBGINV` **MIPS32**

Purpose: Guest TLB Invalidate

TLBGINV invalidates a set of guest TLB entries based on ASID and guest *Index* match. The virtual address is ignored in the match.

Implementation of the TLBGINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of *EntryHI*$_{EHINV}$ field is required for implementation of TLBGINV instruction.

Support for TLBGINV is recommended for implementations supporting VTLB/FTLB type TLB's.

Description:

On execution of the TLBGINV instruction, the set of guest TLB entries with matching ASID are marked invalid, excluding those guest TLB entries which have their G bit set to 1.

The *EntryHI*$_{ASID}$ field has to be set to the appropriate ASID value before executing the TLBGINV instruction.

Behavior of the TLBGINV instruction applies to all applicable guest TLB entries and is unaffected by the setting of the Guest.*Wired* register.

For JTLB-based MMU(*Config*$_{MT}$=1):

All matching entries in the guest JTLB are invalidated. *Index* is unused.

For VTLB/FTLB -based MMU(*Config*$_{MT}$=4):

A TLBGINV with *Index* set in guest VTLB range causes all matching entries in the guest VTLB to be invalidated. A TLBGINV with *Index* set in guest FTLB range causes all matching entries in the single addressed guest FTLB set to be invalidated.

If TLB invalidate walk is implemented in software (*Config4*$_{IE}$=2), then software must do these steps:

1. one TLBGINV instruction is executed with an index in guest VTLB range (invalidates all matching guest VTLB entries)
2. a TLBGINV instruction is executed for each guest FTLB set (invalidates all matching entries in guest FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4*$_{IE}$=3), then software must do these steps:

1. one TLBGINV instruction is executed (invalidates all matching entries in both guest FTLB & guest VTLB). In this case, *Index* is unused.

In an implementation supporting GuestID (*GuestCtl0*$_{G1}$=1), matching of guest TLB entries includes comparison of the TLB entry GuestID with the Root GuestID control field, *GuestCtl1*$_{RID}$.

Note that the TLBGINV instruction only invalidates guest virtual address translations in the guest TLB, invalidation of guest physical address translations requires execution of the equivalent TLBINV instruction sequence in the root TLB.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (for the case of *Config*$_{MT}$=4).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

For processors that do not include a TLB, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if (Guest.Config_MT=1 or
   (Guest.Config_MT=4 & Guest.Config4_IE=2 & Index ≤ Guest.Config1_MMU_SIZE-1))
   startnum ← 0
   endnum ← Guest.Config1_MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (Guest.Config_MT=4 & Guest.Config4_IE=2 & Index > Guest.Config1_MMU_SIZE-1)
   startnum ← start of selected Guest FTLB set // implementation specific
   endnum ← end of selected Guest FTLB set - 1 //implementation specifc
endif

if (Guest.Config_MT=4 & Guest.Config4_IE=3))
   startnum ← 0
   endnum ← Guest.Config1_MMU_SIZE-1 +
   ((Guest.Config4_FTLBWays + 2) * Guest.Config4_FTLBSets)
endif

if IsCoprocessorEnabled(0) then
   for (i = startnum to endnum)
      if ((Guest.TLB[i]_ASID = EntryHi_ASID) & (Guest.TLB[i]_G = 0))
         if (GuestCtl0_G1 = 1)
            if (Guest.TLB[i]_GuestID = GuestCtl1_RID)
               Guest.TLB[i]_hardware_invalid ← 1
            endif
         else
               Guest.TLB[i]_hardware_invalid ← 1
         endif
      endif
   endfor
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBGINVF 001100
6	1	19	6

Format: `TLBGINVF` **MIPS32**

Purpose: Guest TLB Invalidate Flush

TLBGINVF invalidates a set of Guest TLB entries based on *Index* match. The virtual address and ASID are ignored in the match.

Implementation of the TLBGINVF instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of the *EntryHI*$_{EHINV}$ field is required for implementation of TLBGINV and TLBGINVF instructions.

Support for TLBGINVF is recommend for implementations supporting VTLB/FTLB type TLB's.

Description:

On execution of the TLBGINVF instruction, all entries within range of guest *Index* are invalidated.

Behavior of the TLBGINVF instruction applies to all applicable guest TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU(*Config*$_{MT}$=1):

TLBGINVF causes all entries in the guest JTLB to be invalidated. *Index* is unused.

For VTLB/FTLB-based MMU(*Config*$_{MT}$=4):

TLBINVF with *Index* in guest VTLB range causes all entries in the guest VTLB to be invalidated.

TLBINVF with *Index* in guest FTLB range causes all entries in the single corresponding set in the guest FTLB to be invalidated.

If TLB invalidate walk is implemented in software (*Config4*$_{IE}$=2), then software must do these steps:

1. one TLBGINV instruction is executed with an index in guest VTLB range (invalidates all matching guest VTLB entries)
2. a TLBGINV instruction is executed for each guest FTLB set (invalidates all matching entries in guest FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4*$_{IE}$=3), then software must do these steps:

1. one TLBGINV instruction is executed (invalidates all matching entries in both guest FTLB & guest VTLB). In this case, *Index* is unused.

In an implementation supporting GuestID (*GuestCtl0*$_{G1}$=1), matching of guest TLB entries includes comparison of the TLB entry GuestID with the Root GuestID control field, *GuestCtl1*$_{RID}$.

Note that the TLBGINVF instruction only invalidates guest virtual address translations in the guest TLB, invalidation of guest physical address translations requires execution of the equivalent TLBINVF instruction sequence in the root TLB.

Restrictions:

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries visible as defined by the Config4 register.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if ( Guest.Config$_{MT}$=1 or
    (Guest.Config$_{MT}$=4 & Guest.Config4$_{IE}$=2 & Index $\leq$ Guest.Config1$_{MMU\_SIZE-1}$))
    startnum ← 0
    endnum ← Guest.Config1$_{MMU\_SIZE-1}$
endif
// treating VTLB and FTLB as one array
if (Guest.Config$_{MT}$=4 & Guest.Config4$_{IE}$=2 & Index > Guest.Config1$_{MMU\_SIZE-1}$)
    startnum ← start of selected Guest FTLB set // implementation specific
    endnum ← end of selected Guest FTLB set - 1 //implementation specifc
endif

if (Guest.Config$_{MT}$=4 & Guest.Config4$_{IE}$=3))
    startnum ← 0
    endnum ← Guest.Config1$_{MMU\_SIZE-1}$ +
    ((Guest.Config4$_{FTLBWays}$ + 2) * Guest.Config4$_{FTLBSets}$)
endif

if IsCoprocessorEnabled(0) then
    for (i = startnum to endnum)
        if (GuestCtl0$_{G1}$ = 1)
            if (Guest.TLB[i]$_{GuestID}$ = GuestCtl1$_{RID}$)
                Guest.TLB[i]$_{hardware\_invalid}$ ← 1
            endif
        else
                Guest.TLB[i]$_{hardware\_invalid}$ ← 1
        endif
    endfor
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBGP 010000
6	1	19	6

Format: `TLBGP` **MIPS32**

Purpose: Probe Guest TLB for Matching Entry

To find a matching entry in the Guest TLB, initiated from root mode.

Description:

The *Guest.Index* register is loaded with the address of the Guest TLB entry whose contents match the contents of the *Guest.EntryHi* register. If no Guest TLB entry matches, the high-order bit of the *Guest.Index* register is set.

In an implementation supporting GuestID (*GuestCtl0*$_{G1}$=1), if the GuestID read does not match *GuestCtl1*$_{RID}$, then the match fails.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

If an implementation detects multiple matches, and does not detect all multiple matches on TLB write, then a TLBGP instruction can take a Machine Check Exception if multiple matches occur.

For processors that do not include a TLB in the guest context, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if IsCoprocessorEnabled(0) then
    if (Config3$_{VZ}$ = 0) then
        SignalException(ReservedInstruction, 0)
        break
    endif
    Guest.Index ← 1 || UNPREDICTABLE$^{31}$

// If a set-associative TLB is used, then a single set may be probed.

for i in 0...Guest.TLBEntries-1
       if  (((Guest.TLB[i]$_{VPN2}$ and ~(Guest.TLB[i]$_{Mask}$)) =
              (Guest.EntryHi$_{VPN2}$ and ~(Guest.TLB[i]$_{Mask}$))) and
              (Config4$_{IE}$ and not TLB[i]$_{hardware\_invalid}$) and
              (Guest.TLB[i]$_{G}$ or (Guest.TLB[i]$_{ASID}$ = Guest.EntryHi$_{ASID}$)))then
                    if (GuestCtl0$_{G1}$ = 1)
                           if (Guest.TLB[i]$_{GuestID}$ = GuestCtl1$_{RID}$)
                                  Guest.Index ← i
                           endif
                    else
                       Guest.Index ← i
                 endif
          endif
    endfor
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Machine Check (implementation dependent)

Reserved Instruction

31–26	25	24–6	5–0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBGR 001001
6	1	19	6

Format: `TLBGR` **MIPS32**

Purpose: Read Indexed Guest TLB Entry

To read an entry from the Guest TLB into the guest context, initiated from root mode.

Description:

The *Guest.EntryHi*, *Guest.EntryLo0*, *Guest.EntryLo1*, and *Guest.PageMask* registers are loaded with the contents of the Guest TLB entry pointed to by the *Guest.Index* register. Note that the value written to the *Guest.EntryHi*, *Guest.EntryLo0*, and *Guest.EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the PFN field of the *EntryLo0* and *EntryLo1* registers may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.

In an implementation supporting GuestID, if the TLB entry is not marked invalid, the $GuestCtl1_{RID}$ field is written with the GuestID of the TLB entry read.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Guest.Index* register are greater than or equal to the number of TLB entries in the guest context.

If root-mode access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

The guest context does not implement the Virtualization Module. Use of this instruction in guest-kernel mode will result in a Reserved Instruction exception, taken in guest mode.

For processors that do not include a TLB in the guest context, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if IsCoprocessorEnabled(0) then
    if ($Config3_{VZ}$ = 0) then
        SignalException(ReservedInstruction, 0)
        break
    endif
    i ← Guest.Index
    if i > (Guest.TLBEntries - 1) then
```

```
            **UNDEFINED**
        endif
        if (Config4$_{IE}$ = 1 && Guest.TLB[i]$_{EHINV}$ = 1) then
            GuestCtl1$_{RID}$ ← 0
            Guest.Pagemask$_{Mask}$ ← 0
            Guest.EntryHi ← 0
            Guest.EntryLo1 ← 0
            Guest.EntryLo0 ← 0
            Guest.EntryHi$_{EHINV}$ ← 1
            break
        endif
        if (GuestCtl0$_{G1}$ = 1)
            GuestCtl1$_{RID}$ ← Guest.TLB[i]$_{GuestID}$
        endif
        Guest.PageMask$_{Mask}$ ← Guest.TLB[i]$_{Mask}$
        Guest.EntryHi ←
                (Guest.TLB[i]$_{VPN2}$ and not Guest.TLB[i]$_{Mask}$) || # Masking impl dependent
                0$^5$ || Guest.TLB[i]$_{ASID}$
        Guest.EntryLo1 ← 0$^2$ ||
                (Guest.TLB[i]$_{PFN1}$ and not Guest.TLB[i]$_{Mask}$) || # Masking impl dependent
                Guest.TLB[i]$_{C1}$ || Guest.TLB[i]$_{D1}$ || Guest.TLB[i]$_{V1}$ || Guest.TLB[i]$_{G}$
        Guest.EntryLo0 ← 0$^2$ ||
                (Guest.TLB[i]$_{PFN0}$ and not Guest.TLB[i]$_{Mask}$) || # Masking impl dependent
                Guest.TLB[i]$_{C0}$ || Guest.TLB[i]$_{D0}$ || Guest.TLB[i]$_{V0}$ || Guest.TLB[i]$_{G}$
    else
        SignalException(CoprocessorUnusable, 0)
    endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBGWI 001010
6	1	19	6

Format: `TLBGWI` **MIPS32**

Purpose: Write Indexed Guest TLB Entry

To write a Guest TLB entry indexed by the *Index* register, initiated from root mode.

Description:

The Guest TLB entry pointed to by the *Guest.Index* register is written from the contents of the *Guest.EntryHi*, *Guest.EntryLo0*, *Guest.EntryLo1*, and *Guest.PageMask* registers. The information written to the Guest TLB entry may be different from that in the *Guest.EntryHi*, *Guest.EntryLo0*, and *Guest.EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.
- In an implementation supporting GuestID, $GuestCtl1_{RID}$ is written in the TLB entry.

If EHINV is implemented, the TLBGWI instruction also acts as an explicit TLB entry invalidate operation. The Guest TLB entry pointed to by the Guest.Index register is marked invalid when guest $EntryHI_{EHINV}$=1.

When $EntryHI_{EHINV}$=1, no machine check generating error conditions exist.

Implementation of the TLBGWI invalidate feature is required if the TLBGINV and TLBGINVF instructions are implemented, optional otherwise.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Guest.Index* register are greater than or equal to the number of TLB entries in the guest context.

If access to the root Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

On an FTLB enabled system, if *Guest.Index* is in FTLB range and the page size specified does not match FTLB page size, recommended behavior is that the write not complete and a Machine Check Exception be signaled.

On an FTLB enabled system, for a write in FTLB range, if the VPN is inconsistent with Index, it is recommended that a Machine Check Exception be signaled.

It is implementation dependent whether multiple TLB matches are detected on a TLBGWI, though it is recommended. If a TLB write detects multiple matches, but not necessarily all multiple matches, then it is recommended that a TLB lookup or TLB probe operation signal a Machine Check Exception on detection of multiple matches.

If multiple match detection is implemented, then on detection, it is recommended that the multiple match be invalidated and the write completed. It is recommended that no Machine Check Exception be signaled.

The guest context does not implement the Virtualization Module. Use of this instruction in guest-kernel mode will result in a Reserved Instruction Exception, taken in guest mode.

For processors that do not include a TLB in the guest context, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if IsCoprocessorEnabled(0) then
    if (Config3_VZ = 0) then
        SignalException(ReservedInstruction, 0)
        break
    endif
    i ← Guest.Index
    if (Config4_IE = 1) then
        Guest.TLB[i]_hardware_invalid ← 0
        if ( EntryHI_EHINV=1 ) then
            Guest.TLB[i]_hardware_invalid ← 1
        endif
    endif
    Guest.TLB[i]_Mask ← Guest.PageMask_Mask
    Guest.TLB[i]_R ← Guest.EntryHi_R
    Guest.TLB[i]_VPN2 ← Guest.EntryHi_VPN2 and not Guest.PageMask_Mask # Impl dependent
    Guest.TLB[i]_ASID ← Guest.EntryHi_ASID
    Guest.TLB[i]_G ← Guest.EntryLo1_G and Guest.EntryLo0_G
    Guest.TLB[i]_PFN1 ← Guest.EntryLo1_PFN and not Guest.PageMask_Mask # Impl dependent
    Guest.TLB[i]_C1 ← Guest.EntryLo1_C
    Guest.TLB[i]_D1 ← Guest.EntryLo1_D
    Guest.TLB[i]_V1 ← Guest.EntryLo1_V
    Guest.TLB[i]_PFN0 ← Guest.EntryLo0_PFN and not Guest.PageMask_Mask # Impl dependent
    Guest.TLB[i]_C0 ← Guest.EntryLo0_C
    Guest.TLB[i]_D0 ← Guest.EntryLo0_D
    Guest.TLB[i]_V0 ← Guest.EntryLo0_V
    if (GuestCtl0_G1) then
        Guest.TLB[i]_GuestID ← GuestCtl1_RID
    endif
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Machine Check (disabled if guest *EntryHI*$_{EHINV}$=1.)

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWR 001110
6	1	19	6

Format: `TLBGWR` **MIPS32**

Purpose: Write Random Guest TLB Entry

To write a Guest TLB entry indexed by the *Random* register, initiated from root mode.

Description:

The Guest TLB entry pointed to by the *Guest.Random* register is written from the contents of the *Guest.EntryHi*, *Guest.EntryLo0*, *Guest.EntryLo1*, and *Guest.PageMask* registers.

The information written to the Guest TLB entry may be different from that in the *Guest.EntryHi*, *Guest.EntryLo0*, and *Guest.EntryLo1* registers, in that:

- The value written to the VPN2 field of the Guest TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *Guest.PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a Guest TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *Guest.PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a Guest TLB write.

- The single G bit in the Guest TLB entry is set from the logical AND of the G bits in the *Guest.EntryLo0* and *Guest.EntryLo1* registers.

- In an implementation supporting GuestID, $GuestCtl1_{RID}$ is written in the TLB entry.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

On an VTLB/FTLB enabled implementation, if the *Pagemask* register contains a page size differing from the FTLB page size defined in *Config4*, then the write goes into a random entry in the VTLB.

It is implementation dependent whether multiple TLB matches are detected on a TLBGWR, though it is recommended. If a TLB write detects multiple matches, but not necessarily all multiple matches, then a TLB lookup or TLB probe operation should signal a Machine Check Exception on detection of multiple matches.

If multiple match detection is implemented, then on detection, the multiple match should be invalidated and the write completed. No Machine Check Exception should be signaled.

The guest context does not implement the Virtualization Module. Use of this instruction in guest-kernel mode will result in a Reserved Instruction exception, taken in guest mode.

For processors that do not include a TLB in the guest context, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if IsCoprocessorEnabled(0) then
    if ($Config3_{VZ}$ = 0) then
```

```
            SignalException(ReservedInstruction, 0)
            break
        endif
        i ← Guest.Random
        if (Config4$_{IE}$ = 1) then
            Guest.TLB[i]$_{hardware\_invalid}$ ← 0
            if ( EntryHI$_{EHINV}$=1 ) then
                Guest.TLB[i]$_{hardware\_invalid}$ ← 1
            endif
        endif
        Guest.TLB[i]$_{Mask}$ ← Guest.PageMask$_{Mask}$
        Guest.TLB[i]$_{R}$ ← Guest.EntryHi$_{R}$
        Guest.TLB[i]$_{VPN2}$ ← Guest.EntryHi$_{VPN2}$ and not Guest.PageMask$_{Mask}$ # Impl. dependent
        Guest.TLB[i]$_{ASID}$ ← Guest.EntryHi$_{ASID}$
        Guest.TLB[i]$_{G}$ ← Guest.EntryLo1$_{G}$ and Guest.EntryLo0$_{G}$
        Guest.TLB[i]$_{PFN1}$ ← Guest.EntryLo1$_{PFN}$ and not PageMask$_{Mask}$ # Impl. dependent
        Guest.TLB[i]$_{C1}$ ← Guest.EntryLo1$_{C}$
        Guest.TLB[i]$_{D1}$ ← Guest.EntryLo1$_{D}$
        Guest.TLB[i]$_{V1}$ ← Guest.EntryLo1$_{V}$
        Guest.TLB[i]$_{PFN0}$ ← Guest.EntryLo0$_{PFN}$ and not PageMask$_{Mask}$ # Impl. dependent
        Guest.TLB[i]$_{C0}$ ← Guest.EntryLo0$_{C}$
        Guest.TLB[i]$_{D0}$ ← Guest.EntryLo0$_{D}$
        Guest.TLB[i]$_{V0}$ ← Guest.EntryLo0$_{V}$
        if (GuestCtl0$_{G1}$) then
            Guest.TLB[i]$_{GuestID}$ ← GuestCtl1$_{RID}$
        endif
    else
        SignalException(CoprocessorUnusable, 0)
    endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Machine Check (implementation dependent)

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBINVF 000100
6	1	19	6

Format: `TLBINVF` **MIPS32**

Purpose: TLB Invalidate Flush

Description:

The TLBINVF instruction is unmodified from the base architectural definition, except in an implementation supporting GuestID:

- When executing in Guest mode, if the GuestID read does not match *GuestCtl1*$_{ID}$, then the TLB entry is not modified.
- When executing in Root mode, if the GuestID read does not match *GuestCtl1*$_{RID}$, then the TLB entry is not modified. Note that this only applies to the root TLB, invalidation of guest virtual address translations requires execution of the equivalent TLBGINVF instruction sequence to modify the guest TLB.

Restrictions:

Unchanged from the base architecture.

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBINV 000011
6	1	19	6

Format: `TLBINV` **MIPS32**

Purpose: TLB Invalidate

Description:

The TLBINV instruction is unmodified from the base architectural definition, except in an implementation supporting GuestID:

- When executing in Guest mode, if the GuestID read does not match *GuestCtl1*$_{ID}$, then the TLB entry is not modified.
- When executing in Root mode, if the GuestID read does not match *GuestCtl1*$_{RID}$, then the TLB entry is not modified. Note that this only applies to the root TLB, invalidation of guest virtual address translations requires execution of the equivalent TLBGINV instruction sequence to modify the guest TLB.

Restrictions:

Unchanged from the base architecture.

Exceptions:

Unchanged from the base architecture.

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBP 001000
6	1	19	6

Format: `TLBP` **MIPS32**

Purpose: Probe TLB for Matching Entry

To find a matching entry in the TLB.

Description:

The TLBP instruction is unmodified from the base architectural definition, except in an implementation supporting GuestID:

- When executing in Guest mode, if the GuestID read does not match *GuestCtl1*$_{ID}$, then the match fails.
- When executing in Root mode, if the GuestID read does not match *GuestCtl1*$_{RID}$, then the match fails.

Restrictions:

Unchanged from the base architecture.

Operation:

```
if IsCoprocessorEnabled(0) then
    Index ← 1 || UNPREDICTABLE$^{31}$
    for i in 0...TLBEntries-1
        if ((TLB[i]$_{VPN2}$ & ~(TLB[i]$_{Mask}$)) = (EntryHi$_{VPN2}$ & ~(TLB[i]$_{Mask}$))) and
           (Config4$_{IE}$=1 && TLB[i]$_{hardware\_invalid}$ != 1) and
           ((IsRootMode() and (TLB[i]$_{GuestID}$ = GuestCtl1$_{RID}$)) or
           (IsGuestMode() and (TLB[i]$_{GuestID}$ = GuestCtl1$_{ID}$))) and
           ((TLB[i]$_{G}$ = 1) or (TLB[i]$_{ASID}$ = EntryHi$_{ASID}$))then
            Index ← i
        endif
    endfor
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Machine Check (implementation defined)

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	1	19	6

Format: `TLBR` **MIPS32**

Purpose: Read Indexed TLB Entry

To read an entry from the TLB.

Description:

The TLBR instruction is unmodified from the base architectural definition, except in an implementation supporting GuestID:

- When executing in Guest mode, if the GuestID read does not match *GuestCtl1*$_{ID}$, then the TLB related CP0 registers are zeroed and EHINV is set to 1.
- When executing in Root mode and the TLB entry is not marked as invalid, *GuestCtl1*$_{RID}$ is set to the GuestID of the TLB entry read.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. The preferred implementation is to signal a Reserved Instruction Exception.

Operation:

```
if IsCoprocessorEnabled(0) then
    i ← Index
    if i > (TLBEntries - 1) then
        UNDEFINED
    endif
    if (Config4$_{IE}$=1 && TLB[i]$_{hardware\_invalid}$ = 1) then
        GuestCtl1$_{RID}$ ← 0
        Pagemask$_{Mask}$ ← 0
        EntryHi ← 0
        EntryLo1 ← 0
        EntryLo0 ← 0
        EntryHi$_{EHINV}$ ← 1
        break
    endif
    PageMask$_{Mask}$ ← TLB[i]$_{Mask}$
    EntryHi ←
            (TLB[i]$_{VPN2}$ and not TLB[i]$_{Mask}$) || # Masking implementation dependent
            $0^5$ || TLB[i]$_{ASID}$
    EntryLo1 ← $0^2$ ||
            (TLB[i]$_{PFN1}$ and not TLB[i]$_{Mask}$) || # Masking mplementation dependent
            TLB[i]$_{C1}$ || TLB[i]$_{D1}$ || TLB[i]$_{V1}$ || TLB[i]$_G$
    EntryLo0 ← $0^2$ ||
            (TLB[i]$_{PFN0}$ and not TLB[i]$_{Mask}$) || # Masking mplementation dependent
            TLB[i]$_{C0}$ || TLB[i]$_{D0}$ || TLB[i]$_{V0}$ || TLB[i]$_G$
    # if in guest mode, if the TLB entry guest id != guest id
    # zero the result
    if (GuestCtl0$_{G1}$ = 1)
        if (GuestCtl0$_{GM}$=1) and (Root.Debug$_{DM}$=0) and
            (Root.Status$_{ERL}$=0) and (Root.Status$_{EXL}$=0) then
            if (TLB[i]$_{ID}$ != GuestCtl1$_{ID}$) then
                Pagemask$_{Mask}$ ← 0
                EntryHi ← 0
                EntryLo1 ← 0
                EntryLo0 ← 0
                EntryHi$_{EHINV}$ ← 1
            endif
        else #in root mode, save read GuestID
            GuestCtl1$_{RID}$ ← TLB[i]$_{GuestID}$
        endif
    endif
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWI 000010
6	1	19	6

Format: `TLBWI` **MIPS32**

Purpose: Write Indexed TLB Entry

To write a TLB entry indexed by the *Index* register.

Description:

The TLBWI instruction is unmodified from the base architecture, except in an implementation supporting GuestID:

- When executing in Guest mode, *GuestCtl1*$_{ID}$ is written in the guest TLB entry.
- When executing in Root mode *GuestCtl1*$_{RID}$ is written in the root TLB entry.

It is expected that a Guest entry in the Root TLB must have its Global(G) bit set to 1 on a TLB write. This is because the ASID field is not applicable for a Guest entry in the Root TLB.

If EHINV is implemented, the TLBWI instruction also acts as an explicit TLB entry invalidate operation. The TLB entry pointed to by the Index register is marked invalid when *EntryHI*$_{EHINV}$=1.

When *EntryHI*$_{EHINV}$=1, no machine check generating error conditions exist.

Restrictions:

Unmodified from the base architecture.

Operation:

```
if IsCoprocessorEnabled(0) then
    i ← Index
    if ( Config4$_{IE}$=1) then
        TLB[i]$_{hardware\_invalid}$ ← 0
        if (EntryHI$_{EHINV}$=1) then
            TLB[i]$_{hardware\_invalid}$ ← 1
        endif
    endif
    TLB[i]$_{Mask}$ ← PageMask$_{Mask}$
    TLB[i]$_{VPN2}$ ← EntryHi$_{VPN2}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{ASID}$ ← EntryHi$_{ASID}$
    if (GuestCtl0$_{G1}$) then
            if ((GuestCtl0$_{RAD}$=0) and IsRootMode() and (GuestCtl1$_{RID}$ != 0))
                TLB[i]$_G$ ← 1
            else
                TLB[i]$_G$ ← EntryLo1$_G$ and EntryLo0$_G$
            endif
    else
            TLB[i]$_G$ ← EntryLo1$_G$ and EntryLo0$_G$
    endif
    if ( IsRootMode() ) then
        TLB[i]$_{GuestID}$ ← GuestCtl1$_{RID}$
    else
        TLB[i]$_{GuestID}$ ← GuestCtl1$_{ID}$
    endif
    TLB[i]$_{PFN1}$ ← EntryLo1$_{PFN}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{C1}$ ← EntryLo1$_{C}$
    TLB[i]$_{D1}$ ← EntryLo1$_{D}$
    TLB[i]$_{V1}$ ← EntryLo1$_{V}$
    TLB[i]$_{PFN0}$ ← EntryLo0$_{PFN}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{C0}$ ← EntryLo0$_{C}$
    TLB[i]$_{D0}$ ← EntryLo0$_{D}$
    TLB[i]$_{V0}$ ← EntryLo0$_{V}$
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Unmodified from the base architecture.

Chapter 7

Notes

This Virtualization Module specification is a work in progress. Feedback and comments are welcomed on the functional behavior, and the explanations of that behavior.

7.1 Potential areas of improvement

The following items have been identified as potential areas of improvement in the specification.

- Extensions to EJTAG specification to allow additional control over hardware breakpoints used during guest execution.
- Consider options to allow for translation of 36-bit physical addresses
- Consider options to reduce the cost of guest0-guest1-guest0 context switching.
- Security: JTAG, DEBUG, Boot, IOMMU

Notes