MIPS

MIPS Training

Virtualization Module MIPS32®

www.mips.com

This course section covers the MIPS32 Architecture Virtualization Module for MIPS processors cores

Course Overview

- Introduction What and Why?
- Operating modes
- Virtual Memory
- Exceptions
- Interrupts
- Hypervisor Resource Control
- Virtualization and Multi Core
- Virtualization and Multi Threaded Cores
- System Control CP0 registers

MIPS

2

This section describes Virtualization Module support as it pertains to a MIPS Core implementation.

First I'll explain what Virtualization is and why hardware assisted virtualization is a good idea.

Operating modes will cover the new modes added for assisted virtualization.

The virtual memory section covers the use of the tlb's to control memory access

Exceptions covers exceptions within the context of virtualization

Interrupts covers interrupts within the contest virtualization

The Hypervisor resource section covers how core resources are configured and protected

In the Multi-Core section, I'll discuss how virtualization pertains to a Multi-Core System

In the Multi-Threading section I'll discuss how virtualization pertains to a Multi-Threaded Core and Multi-Threaded Multi Core

The System Control CP0 Registers section covers CP0 Registers that are related to virtualization

Introduction

- **What is Computer Virtualization?**
  - Turning one computer into several
    - Allows the running of multiple Virtual Machines (VM)
      - Each Context looks like an individual computer
      - Multiple Operating Systems
      - Embedding programs
      - Combination of both
    - Software running within a VM is separated from the underlying Hardware resources.

MIPS

3

Simply put Computer Virtualization makes one physical computer look like any number of physical computers.

+ It allows the running of multiple operating systems, multiple embedded programs or a combination of both.

+ It does this by separating the software running in these virtual machines or VMs from the underlying hardware resources.

Introduction

- **Host, Root or Processor**
  - The actual machine connected to the physical hardware
- **Virtual Machine - Virtual context of a processor**
- **Root Context – context of the processor**
- **Guest Context - Run in one or more VMs**
- **Host Software**
  - Called – Hypervisor, runs in the Root Context
  - Creates and controls the VMs

MIPS 4

Here are some terms I will be using:

+ A Host or Processor is connected to the actual physical hardware. A processor is synonymous with a Virtual Processor or VPE in a Core that supports the Multi Threaded ASE.

+ A Virtual machine or VM is a Virtual context of a processor created by software.

+ The root context is the context of the physical system.

+The Guest context is a context of a virtual machine. There can be several virtual machines each running a different Guest OS with their own context.

+The software that creates and controls the VMs is call a Hypervisor. The Hypervisor runs in the Root Context. It has direct control of the hardware and thus creates and maintains the trusted execution environment. The Hypervisor is responsible for loading and controlling the software running in a guest context. The software running in the guest context will not be able to tell it is not running on actual hardware

but all hardware interfacing that this software does can be controlled by the Hypervisor running on the host machine.

Introduction

- **Popek and Goldberg virtualization requirements:**
  - Equivalence / Fidelity
    - A program running under the Hypervisor should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
  - Resource control / Safety
    - The Hypervisor must be in complete control of the virtualized resources.
  - Behavior sensitive instructions
    - A statistically dominant fraction of machine instructions must be executed without Hypervisor intervention.
  - For more on this see: http://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements

MIPS 5

Taking a small step back, the idea for computer virtualization is not new. In 1974 Gerald Popek and Robert Goldberg wrote an article call "Formal Requirements for Virtualizeable Third Generation Architectures“. In it they laid out the 3 main requirements for virtualization .

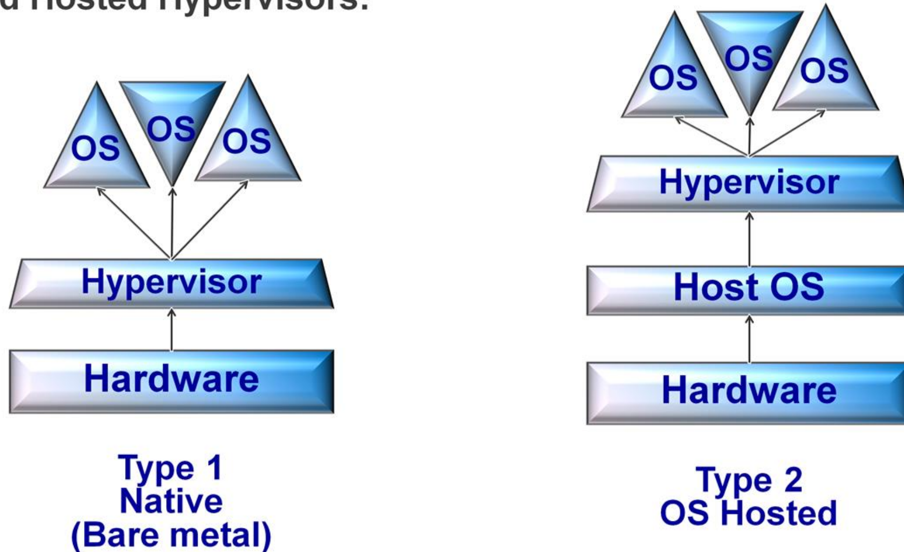
+ Equivalence - where running on the virtual machine should look identical to running on the hardware directly.

+ Resource control - were the Hypervisor must be in control of the virtualized resources.

+ Last - most of the instructions executed should not require Hypervisor intervention.

Introduction

- **Native and Hosted Hypervisors:**



MIPS

6

There are 2 types of Hypervisors:

+ A Type 1 or native Hypervisor runs directly on the hardware or processor and the Guest software such as an OS runs in VMs in the level above. Microsoft Hyper-V is an example of this type.

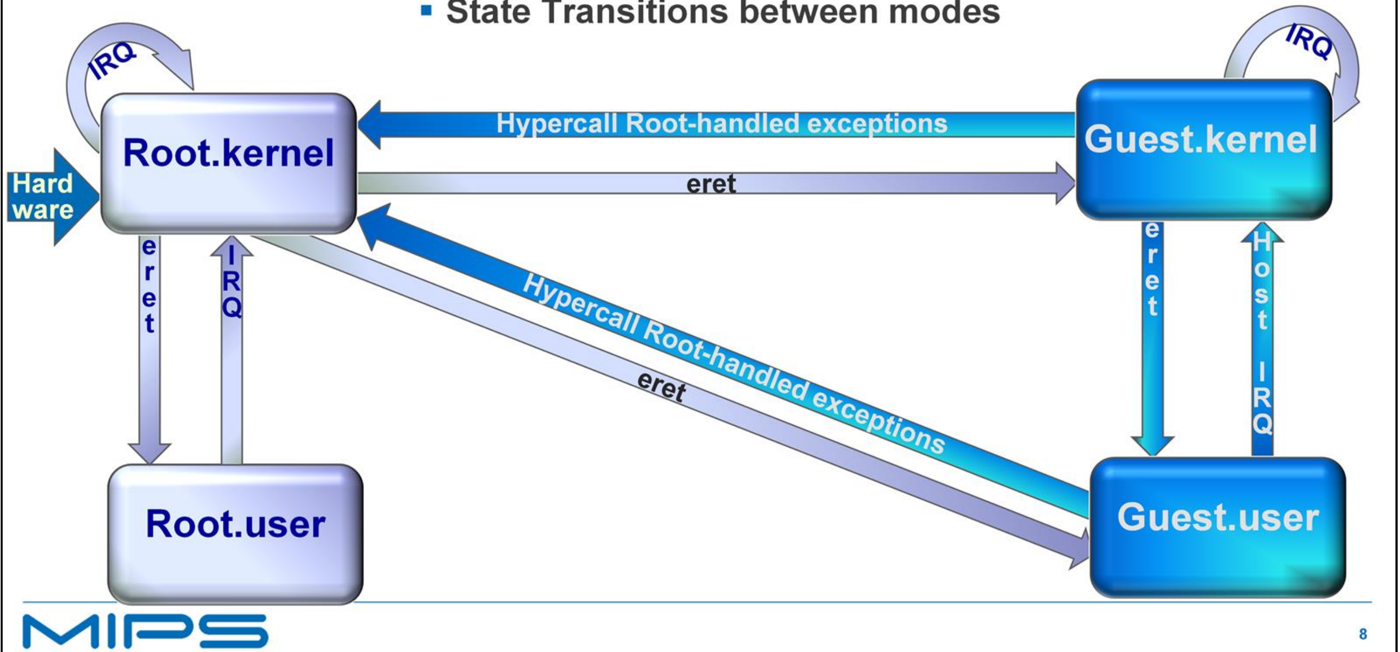
+ A Type 2 or hosted Hypervisor runs within a conventional operating system environment. These are also referred to as trap and emulate because all actions by a Guest OS that need to effect actual physical resources are trapped and the access is emulated. A common example of this is VMware Workstation that runs in the Microsoft Windows environment. On a Standard MIPS core the type 2 Hypervisor would run in Kernel Mode and the Guest OSs would run in user mode. In this way any access the Guest OS tries to make to a privilege region of memory or execute a privilege instruction will cause an exception that is handled by the Hypervisor running in Kernel mode.

Introduction

- **Virtualization Module of the MIPS32 Architecture**
  - Hardware-assisted virtualization, which delivers increased virtualized system performance through changes in the CPU architecture, without requiring any changes to the Guest operating systems.
  - Enables efficient use of a Type 1 Hypervisor

MIPS 7

The virtualization module extends the standard MIPS32 architecture so that an efficient type 1 Hypervisor can be used. The remaining sections of this class will cover the details of the this extension.



Fundamental to the Virtualization Module is the addition of a limited-privilege Guest operating mode.

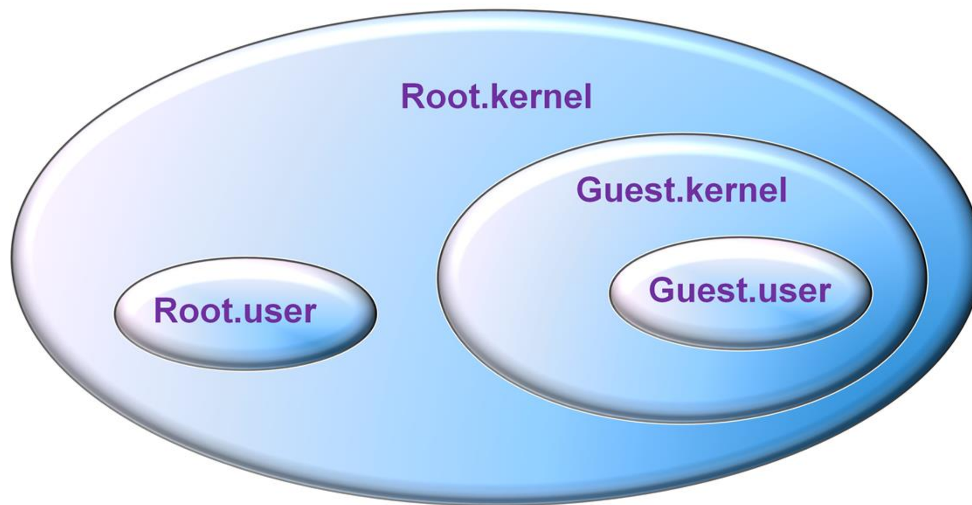
+ The Virtualization module keeps the existing privileged mode and calls it Root mode. For this mode, the classic Kernel, User and Supervisor operating modes will be referred to as Root.kernel, Root.user and Root.supervisor respectively. Note: The diagram does not include supervisor mode because it is seldom used.

+ The Virtualization module adds a Guest mode which consists of the new operating modes Guest.kernel, Guest.user and Guest.supervisor. These are orthogonal to the existing kernel, user and supervisor modes.

+ Virtualization module can selectively allow the Guest.kernel mode to handle some interrupts, Guest exceptions, and manage virtual memory for Guest.user mode processes.

+ The Hypervisor handles all exceptions that happen in the root context and can selectively handle exceptions that happen in the guest context.

Operating Modes



MIPS

9

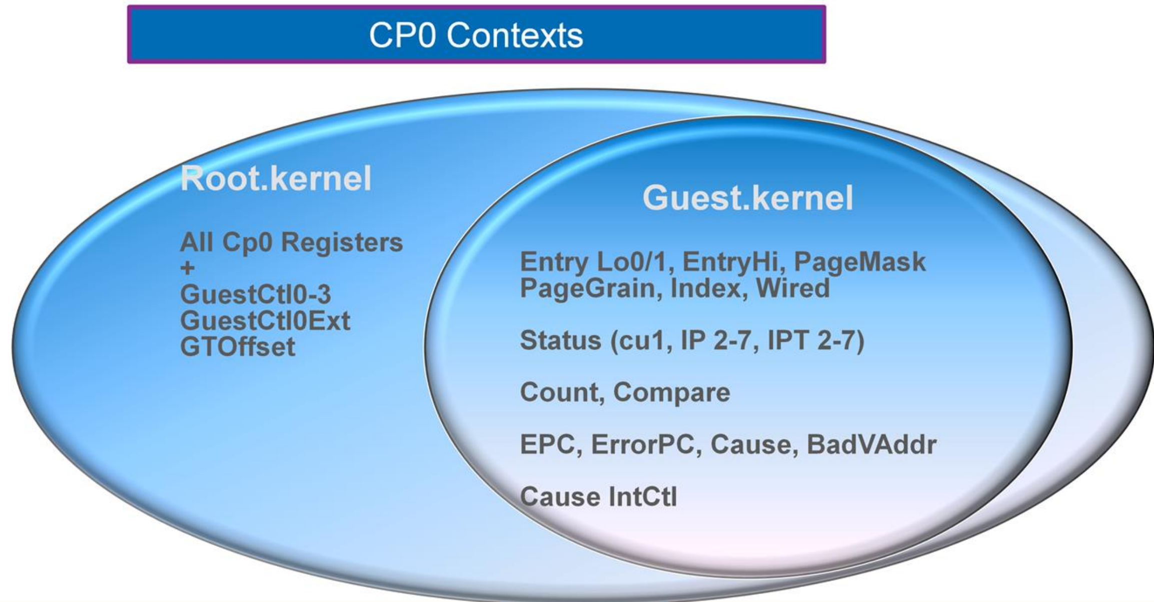
Three operating modes are required to execute a virtualized Guest operating system:

+ unprivileged Guest.user,

+ limited-privilege Guest.kernel

+ and full-privilege Root.kernel.

+In addition, a Root.user mode could be used to execute non-virtualized software that needs to be secure.



The Root and Guest mode each have their own contexts. The term ‘context’ refers to the software visible state held within each Coprocessor 0 register set.

+ The Virtualization Module physically replicates a subset of the Coprocessor 0 register set for use by the Guest Operating System. There is only one set of Guest CP0 registers per processor.

+The Root.kernel uses the full standard set of CP0 register and additional CP0 registers used to control Guest mode operation.

During Guest mode execution, both the Guest Coprocessor 0 and the Root Coprocessor 0 contexts are active. The presence of two simultaneously active Coprocessor 0 contexts is fundamental to the operation of the Virtualization Module. Exceptions can be handled in the mode whose context triggered the exception. An exception triggered by the Guest CP0 context can be handled in Guest mode and a exception triggered by the Root CP0 context will be handled in Root mode. For example a timer interrupt that is caused by the Root-CP0 count register reaching the Root.Compare register will be handled in the Root mode. A timer interrupt that is caused by the Guest.Count register reaching the Guest.Compare register will be handled in the Guest mode.

For a simple switch between Guest mode to Root Mode, the presence of two Coprocessor 0 contexts and a shadow register set allows for an immediate switch between Guest and Root modes – without requiring a context switch to or from memory. Simultaneously active CP0 contexts for the Guest and Root 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.

For a Hypervisor to switch from one Guest to another it must save the context of the Guest OS being switched out and restore context of the Guest OS being switched in. The Hypervisor must save and restore the software visible state and the physical resources which are shared between Guests, such as the Guest CP0 Registers, Root CP0 Guest Registers, Guest General Purpose Registers, Floating Point Registers, Hi/Lo accumulator registers and the TLB state . Some of the MIPS Cores may require less register saving. For example MT Cores and Cores with Shadow register sets can avoid the saving of GPRs. Cores with multiple FPU contexts can avoid saving FPU registers. The Virtualization Module also can be configured to avoid the saving of the TLB state.

Operating Modes

- **Virtualization Module control of Guest through Root mode software**
  - Allows Root to modify Guest configuration
    - Supports privileged software from different hardware platforms.
    - Allows Guest VM to have subset of features from the host system
  - Over privileged instructions that can be executed
  - Context Registers which can be accessed
  - Interrupts and exceptions which can be taken

MIPS

11

The MIPS Virtualization Module allows the Hypervisor to modify Guest configuration by writing the Guest configuration registers. In this manner, such a virtualized system can support privileged software from different hardware platforms by running them as Guests with different configurations. This allows a virtualized Guest to have features and capabilities which are a subset of the Root host machine and different from other Guests. For example if the host machine has a FPU the Hypervisor can program the Guest.Config[FP] field to not have a FPU.

+ The Virtualization Module provides Root-mode software with controls over privileged instructions that can be executed,

+ The CP0 Context registers which can be accessed,

+ and the interrupts and exceptions which can be taken when in Guest mode.

There will be more on this in upcoming sections.

Operating Modes

Function	Name	Bit	Read/Write	Reset State
Guest Mode	$GuestCtl0_{GM}$	CP0 Register 12, Select 6, bit 31	R/W	0
Description				
▪ The processor is in Guest mode when GM=1, Root.Status[EXL] = 0 and Root.Status[ERL] = 0 and Root.Debug[DM] = 0. ▪ Controlled access to Core resources ▪ Instruction monitoring ▪ 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.				

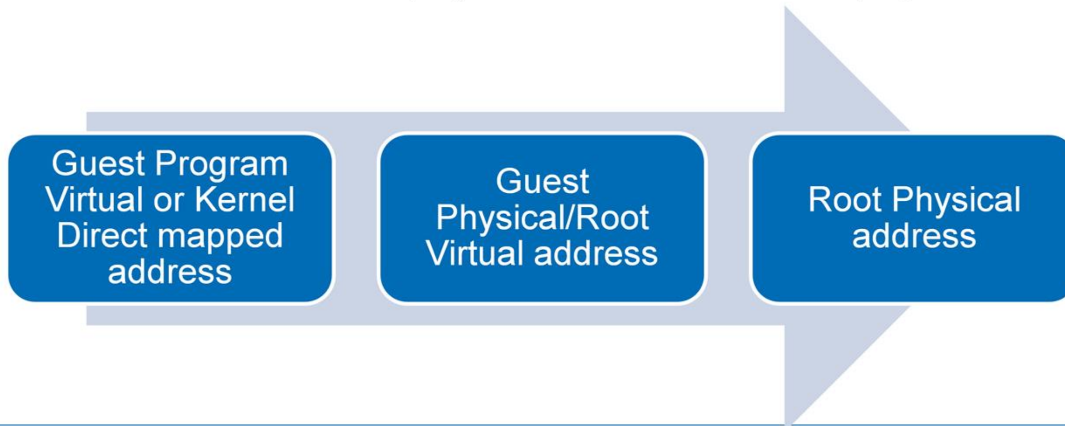
MIPS 12

Guest mode is controlled by the CP0 Root.GuestCtl0[GM] bit; it is used along with Root-mode exception and error status bits (StatusEXL, StatusERL) and the Debug Mode bit (DebugDM) to determine whether the processor is operating in Guest mode or Root mode.

The processor is in Guest mode when GM bit is set and the Root.Status [EXL], [ERL] and [DM] bits are cleared.

Virtual Memory

- **Guest VM has its own MMU either TLB or FMT**
  - Translates to Guest physical address
- **Root MMU translates Guest physical address to actual physical address**



MIPS 13

The Virtualization Module extension contains a hierarchy of memory management units which are used for the isolation of applications, operating systems and the Hypervisor. Specifically, a Guest MMU is managed by a Guest OS to isolate guest applications while a Root MMU is managed by the Hypervisor to isolate Guest OSes and itself. Only the hypervisor is allowed to interact with secure code that controls the Root MMU that does the actual translation to a physical address.

A Hypervisor divides actual physical memory up so that each Guest has its own non-shared physical memory and there is no overlap with the memory that the Hypervisor uses. All Guest address are controlled through the Root MMU even those that the Guest OS sees as being direct mapped.

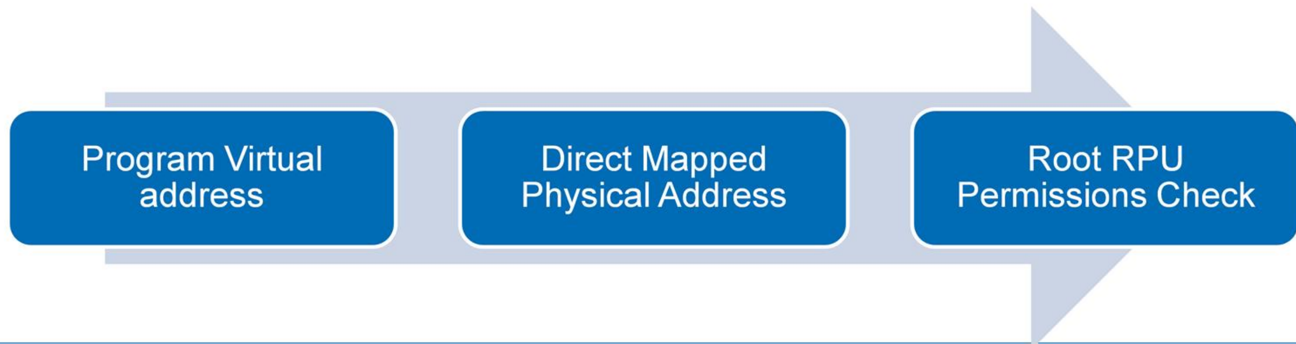
A program executing in Guest.user space works the same as a user space program in classic MIPS. It always uses virtual addresses that are mapped either, direct using a FMT MMU or mapped through a TLB MMU. The Guest OS translates the virtual address to a Guest.physical address. In the Virtualization module the Guest.physical address is a virtual address in the Root.mode context that the Root TLB will translate to a physical address.

Guest direct mapped segments such as KSEG0 or KSEG1 also map to a Guest.physical address which will further be translated buy the Root MMU.

Virtual Memory

Lightweight Virtualization

- **Lightweight Virtualization with Root Protection Unit (RPU)**
  - RPU controls Read, Write and Execute permissions for page size memory sections.
  - The Guest FMT/MMU translates Guest virtual address to direct mapped physical address.



MIPS

14

Lightweight Virtualization supports a Core with a FMT type MMU. This configuration provides memory protection without the need to manage TLB translations by the Hypervisor.

+ This is done through the use of a Root Protection Unit or RPU. The RPU is a de-featured Root TLB that checks the Guest physical address on a page basis for Execute-Inhibit, Read-Inhibit and Dirty page attributes note the Dirty Attribute write-Inhibits the page. It does not support address translation or the Cache Coherency Attribute.

+ The Guest FMT translation produces the direct mapped physical address. If the action being attempted is allowed by the page attributes then the Guest has access to related L1 Cache line, L2 Cache Line and physical memory. Otherwise the access will trap into the Hypervisor, using standard tlb exceptions.

The RPU is programmed just like a TLB however the Root EntryLo0 and EntryLo1 PFN and CCA fields are assumed read-only as 0.

Virtual Memory

Lightweight Virtualization

- RPU exceptions for memory protection

Name	Description	Exception
RI	Read Inhibit	TLB Invalid
XI	Execute Inhibit	TLB Invalid
D	Write Inhibit	TLB Modified exception

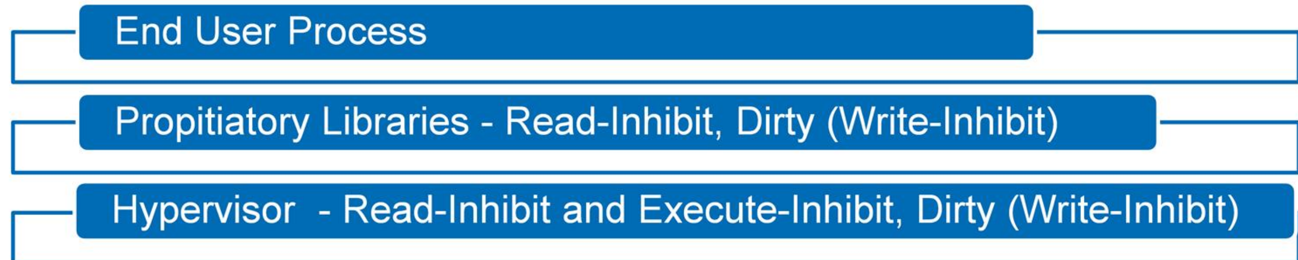
MIPS 15

This table shows you the exception that is generated on failure if a RPU is being used. BadVAddr contains the Physical Address of the source of the of the exception.

Virtual Memory

Lightweight Virtualization

- Lightweight Virtualization Provides Single Guest Support
  - In Guest mode
    - Protects Hypervisor from read, written and execute
    - Protects propitiatory code from being read or written



MIPS 16

Lightweight Virtualization is intended to support single hardware Guest context at a time. Software may swap context if you want to support more than one guest. With it you can protect the Hypervisor code and data from any kind of access from Guest mode. If there are propitiatory libraries that need to be protected from examination they can be set to read-inhibit so only instruction fetches can be done. Both the Hypervisor and propitiatory libraries can be write protected by setting the Dirty attribute.

Virtual Memory

TLB Virtualization

- **Guest ID Active if CP0 GuestCTL0[RAD] is clear (0)**
  - Used in both Guest and Root TBL to determine ownership of TLB entry
  - Benefit – Guest TLB does not need to be invalidated when switching between Guests
  - Hypervisor changes CP0 GuestCtl1[ID] when switching between Guests
- **Root ASID - Used when CP0 GuestCTL0[RAD] is set (1)**
  - Used in Root TLB - Guest ID is ignored, Root ASID is used instead
  - Down side - All entries in Guest TLB must be invalidated when switching between Guests
  - Hypervisor changes CP0 EntryHI[ASID] when switching between Guests

MIPS 17

A TLB type of MMU is used to support full virtualization features.

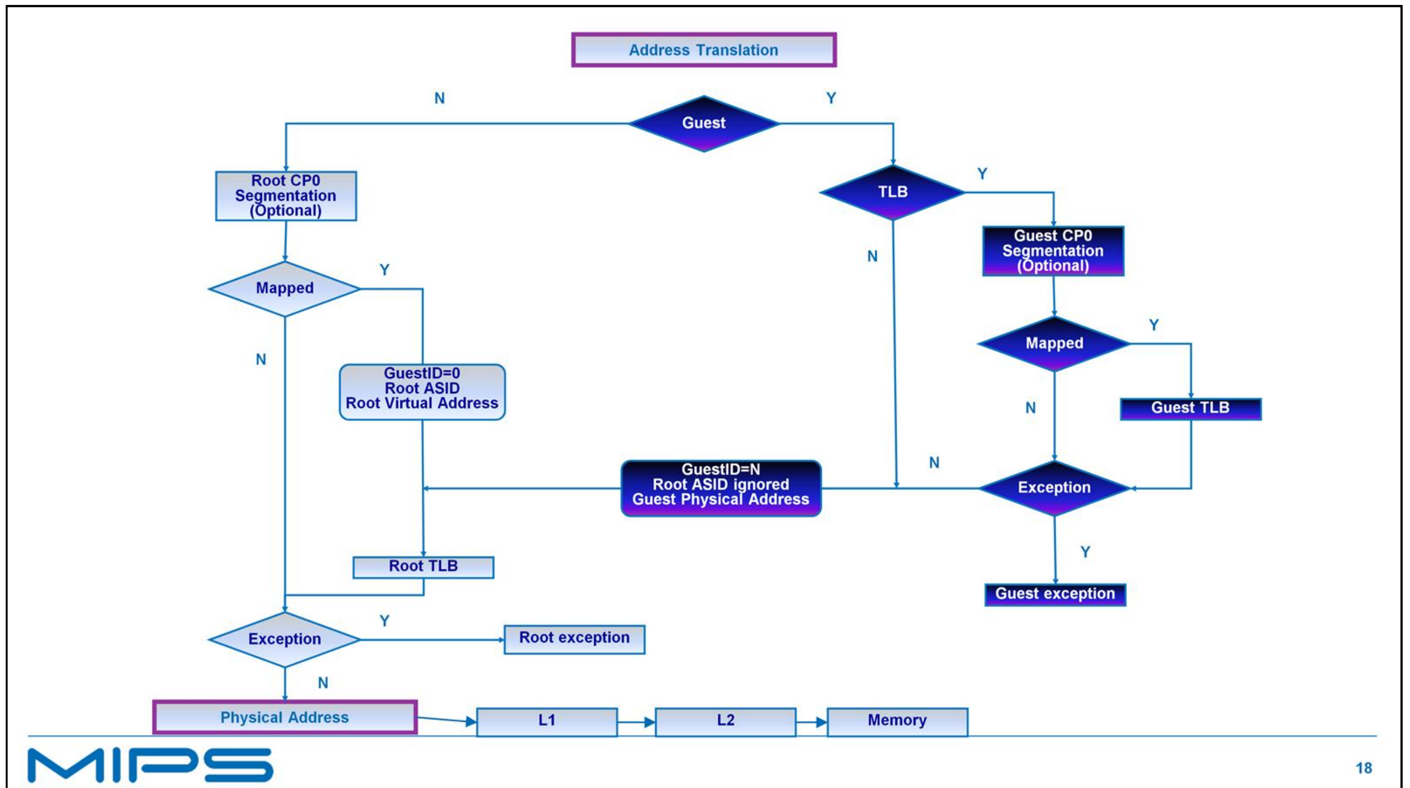
There are 2 ways to associate TLB entries with a specific context of Guest software.

+ A Guest ID can be used to tag all entries in the TLB entry associated with a specific Guest. The Guest entry number is used when writing the TLB entry. It comes from the current value in the Root GuestCtl1[ID] field. The Guest ID is active if the CP0 GuestCtl0[RAD] field is 0. This Guest ID is use for both the Guest tlb entries and the Root tlb entries.

+ The benefit of using the Guest ID is that the Guest TLB doesn't not need to be flushed when the Hypervisor swaps one Guest software context for another.

+ Instead of using a Guest ID, the Root Address Space Identifier, ASID, can be used for the Root tlb entries to associate a specific Guest with a tlb entry in the Root TLB. In this case all entries in the Guest tlb will be assumed to belong to only 1 Guest context. The ASID use when writing a TLB entry comes from the CP0 EntryHi[ASID] field.

+ If the ASID is used the Hypervisor will need to invalidate the entire Guest TLB when it swaps one Guest software context for another. This is needed because there is only one Guest TLB and none of the entries are tagged to indicate which Guest context they belong to.



In Guest mode when Guest segmentation and translation are enabled two levels of address translation are performed by the hardware without software intervention.

+ The first level uses the Guest segmentation controls and the Guest TLB. This translates an address from a Guest Virtual address (GVA) or a Guest Direct mapped address 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.

The main take away from this is the Hypervisor completely controls the memory access to the L1 and L2 caches and physical memory by the Guest. In addition the Hypervisor divides memory up between Guests but makes it look to the Guest OS like it is in control of all physical memory.

Virtual Memory

Counterparts of TLB instructions for Guest TLB (Root mode)		
Mnemonic	Instruction	Description
TLBGINV	Guest TLB Invalidate	Trigger Guest TLB invalidate (ASID)
TLBGINVF	Guest TLB Invalidate Flush	Trigger Guest TLB invalidate (Index)
TLBGP	Probe Guest TLB for Matching Entry	Trigger Guest TLB probe
TLBGR	Read Indexed Guest TLB Entry	Trigger Guest TLB read
TLBGWI	Write Indexed Guest TLB Entry	Trigger Guest TLB write
TLBGWR	Write Random Guest TLB Entry	Trigger Guest TLB write

19

Here are the counter parts of the TLB instructions that operate on the Guest TLB from Root mode. These can be used by the Hypervisor to initialize the Guest TLB, monitor what is in the Guest TLB or to save a restore TLB entries on a Guest context switch if necessary. All instructions that change an entry will only effect tlb entries where the Guest id of the entry is equal to the CP0 GuestCtl1[RID].

Virtual Memory

Modified TLB instructions the operate on TLB (Guest or Root Mode)		
Mnemonic	Instruction	Description
TLBINV	Root TLB Invalidate	Invalidate matching Guest ID entries
TLBINVF	Root TLB Invalidate Flush	Invalidate matching Guest ID entries
TLBP	Probe Root TLB for Matching Entry	Probe Fails if Guest ID does not match
TLBR	Read Indexed Root TLB Entry	Read only entries matching Guest ID
TLBWI	Write Indexed Root TLB Entry	Write Guest ID to Entry
TLBWR	Write Random Root TLB Entry	Write Guest ID to Entry

20

These standard TLB instructions have been modified to use the Guest ID if GuestCTL0[RAD] is clear (0). These instructions are used by a Guest OS or the Hypervisor to manage their respective TLB.

Exceptions

- **In Guest mode execution, exceptions are always taken in the mode whose CP0 state triggered the exception**
  - First checked against the Guest CP0 context
    - resulting exceptions can be handled entirely within Guest mode
  - Then against the Root CP0 context.
    - resulting exceptions require a Root mode exception handler

MIPS 21

Now I'll cover Exceptions. Exceptions in the Virtualization Module are broken down into those are taken in Guest mode by the Guest OS and those that are taken in Root mode by the Hypervisor. Of course all exceptions while the Hypervisor is running are taken in Root mode by the Hypervisor. In Guest mode the Virtualization Module is designed so that the Guest OS can handle all exceptions that it is permitted to, to be as efficient as possible and not cause a mode switch. There are rules to determine which mode an exception is handled in.

+ First is regarding CP0 State exceptions, the exception is always taken in the mode whose CP0 state triggered the exception. For example say the Guest OS is Linux and one of the processes in Linux issues a FPU instruction which causes an exception,

+the Virtualization Module first checks to see if the Guest.Config1[FP] field is set to indicate there is a FPU and then it checks Guest.Status[CU1] to see if the FPU is usable. If either of these is not set then then a Guest exception will be raised and the Linux OS will handle the exception.

+If they are both set then the Root.CP0 context is check. There is a chance that the Root.Status[CU1] bit might not be set; then the exception will be raised in Root mode and the exception will be handled by the Hypervisor exception handler.

For this FPU example this is a way to control FPU sharing between Hypervisor

and multiple Guests. The Hypervisor needs only to save and restore the FPU registers, if a FPU instruction is actually used. In this case the Root exception handler would save the current FPU Register to the Guest context that last used them and then restore the FPU registers for the current Guest.

Exceptions

- **Mode switch is performed after the exception is detected and before any machine state is saved**
- **Reset, NMI, Memory Error and Cache Error**
  - Taken in Root mode and handled by Hypervisor exception routine
- **Hypervisor exception Uses General exception vector**
  - Cause Register ExcCode value 27, 0x1B,
  - GuestCtl0 Register GExcCode value 0, 0x00

MIPS

22

While running in Guest mode and there is an exception cause by the Root context, a mode switch is done to Root mode and then exception state is save to the Root.CP0 registers. The Hypervisor exception routine is expected to save what ever context it deems necessary so the context can be restored after the exception is processed.

+ The Reset, NMI, Memory Error and Cache Error exceptions are always handled in Root mode no matter what mode is executing.

+ A Hypervisor exception is cause by the Guest executing an instruction where the instruction is either not permitted in Guest mode or is not enabled in Guest mode. I'll cover these Guest Privileged Sensitive Instructions in the next slide.

Exceptions

- **Guest Privileged Sensitive Instruction Exception**

Guest CP0 Register Access	When:
Config0-7	GuestCtl0[CF]=0
Count or Compare	GuestCtl0[GT]=0
PageGrain, Wired, SegCtl0, SegCtl1, SegCtl2,	GuestCtl0[AT] != 3
SRSCtl or SRSMap	Always
Index, Random, EntryLo0, EntryLo1, Context, ContextConfig, PageMask, EntryHi	GuestCtl0Ext[MG] =1
BadVAddr, BadInstr, BadInstrP	GuestCtl0Ext[BG] =1
UserLocal, WREna, UserTraceData1, UserTraceData2, KScratch1-6	GuestCtl0Ext[OG] =1
Guest.Status[KSU]	GuestCtl0[MC] =1

23

The virtualization Module can be configured to give control to the Hypervisor when the Guest OS tries to execute a privileged instruction. The Guest Privileged Sensitive Instruction Exception can be triggered by the Guest accessing a sensitive CP0 register or execution a privileged instruction.

Any access to any CP0 register will cause this exception if the GuestCtl0[CP0] bit is not set. This table shows Guest CP0 Sensitive accesses even if the GuestCtl0[CP0] bit is set.

Access to any of the Config registers will cause an exception if the Config register access bit, CF is not set in the GuestCTL0 register

Access to the Compare registers will cause an exception if the timer register access GT bit is not set in the GuestCTL0 register.

Access to the PageGrain, Wired or any segmentation control register will cause a exception if the address translation field AT is not set in the GuestCTL0 register.

Access to any TLB related register if the MMU Guest bit, MG is set in GuestCtl0Ex

Access to any bad address register if the Bad Guest BG bit is set in GuestCtl0Ex

And Access to UserLocal, WREna, UserTraceData1, UserTraceData2, KScratch1-6 if the OG bit is set in GuestCtl0Ex

Exceptions

- **Guest Privileged Sensitive Instruction Exception (continued)**

Guest CP0 Register Access	When
SRSCtl, SRSMap	Always
Prid	Always
CDMMBase, CMGCRBase	Always
ErrCtl, CacheErr, TagLo, DataLo, TagLo, DataLo, TagHi, DataHi, TagHi, DataHi	Always
ErrorEPC	Always
DESAVE	Always
Count (write)	Always

Access to the registers that control the shadow register sets, Processor Identification, Common Device Memory Map, Cache, error

exceptions and Debug Exception Save register will always cause a Guest Privileged Sensitive Instruction Exception to the Hypervisor

Exceptions

- **Guest Privileged Sensitive Instruction Exception (continued)**
  - Triggered by the use of Guest Privileged Sensitive Instructions from Guest mode.

Instruction	
Wait	Any time
Cache, Cachee	GuestCtl0[CG]=0 GuestCtl0[CG]=1 if other than Effective address or GuestCtl0[CP0]=0
Any TLB instruction	GuestCtl0[AT] != 3. or GuestCtl0[CP0]=0
RDPGPR or WRPGPR	SRSCtl[HSS] = 0 or GuestCtl0[CP0]=0

25

This table shows the instructions that could cause the Guest Privileged Sensitive Instruction Exception

A wait instruction will always cause an exception to give the Hypervisor control over the power state of the core.

Cache instructions will generally cause an exception so that no Guest can read or write another Guest's or Hypervisor's cached instructions or data. The Cache Instruction Guest-mode enable, GuestCtl0[CG] bit allows the Guest it execute instructions which use an effective address. This effective address the will be translated by the Root TLB before the instruction executes so the Hypervisor is in control of the address being accessed. An effective address is also known as a program address which can be a virtual address that is mapped by the TLB or a direct mapped address from KSEG0 or KSEG1.

If the Cache Instruction Guest-mode Index enable, GuestCtl0Ext[CGI] field is set along with the GuestCtl0[CG] bit then the Guest can issue index invalidate instructions. Index invalidate instruction will just invalidate a cache line so it will do no harm to another Guest or the Hypervisor but could cause some performance degradation.

If the Address Translation, AT Field is not set so that the MMU is under Guest control then any TLB instruction will cause a exception.

If there are no additional shadow register sets or access to CP0 registers are not allowed the Read GPR from Previous Shadow Set or Write GPR from Previous Shadow Set instructions will cause the exception.

Exceptions

- **Guest Privileged Sensitive Instruction Exception (continued)**
  - Triggered by the use of Guest Privileged Sensitive Instructions from Guest mode If GuestCtl0[CP0]=0 any of the following:

CACHE, DI, EI, MTC0, MFC0, ERET, DERET, RDPGPR, RDHWR, WRPGPR, WAIT, all Enhanced Virtual Addressing (EVA) related instructions (e.g., LBE, LBUE), and all TLB related instructions.

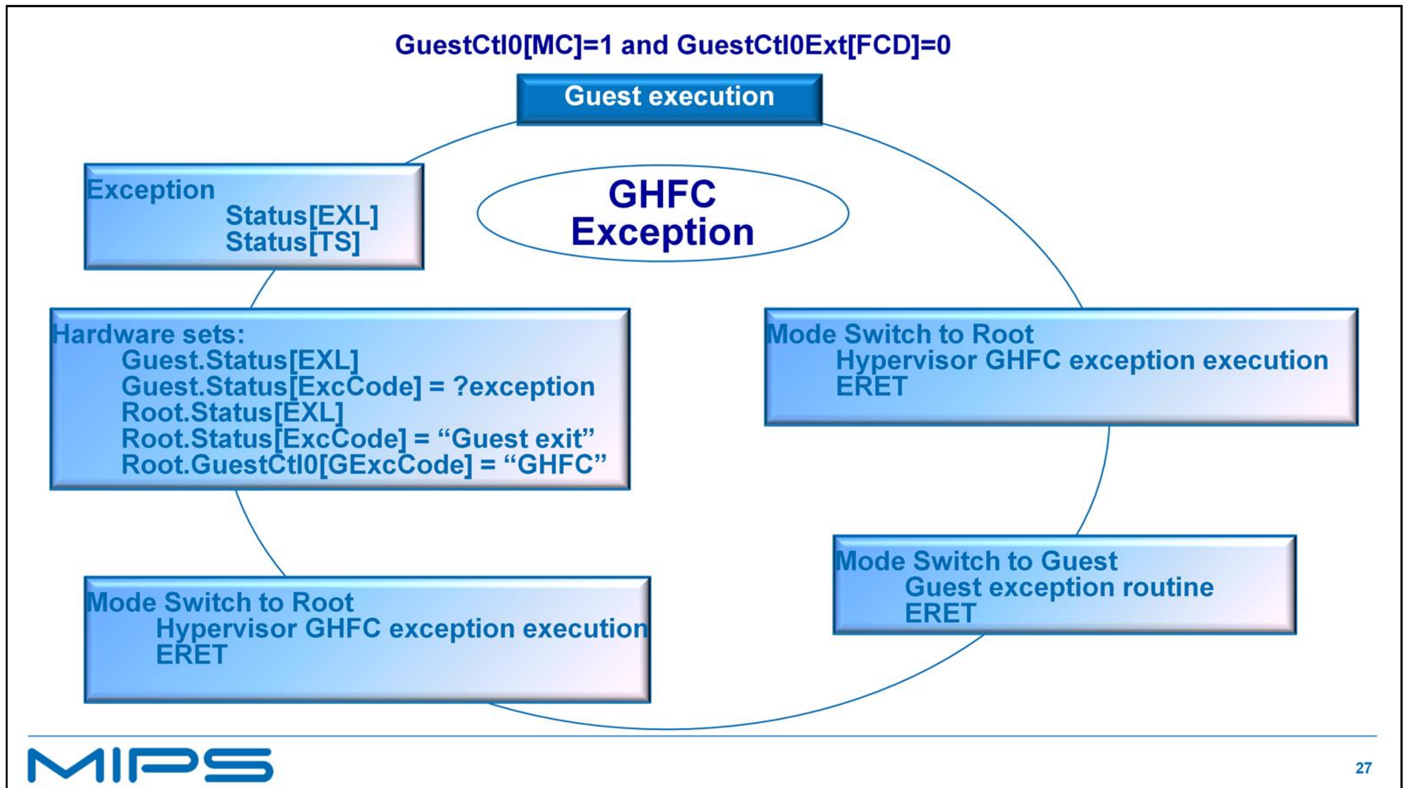
MIPS

26

If the Guest access to coprocessor 0, GuestCtl0[CP0] bit is not set then any Guest execution of a Privileged Instruction will cause an exception. Note: If this bit is 0 then it overrides any other setting and will always cause a Guest Privileged Sensitive Instruction exception for any Guest execution of a Privileged Sensitive Instruction.

+ 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.



The virtualization module can be set so the Hypervisor can intercept all exceptions

+ This is done by setting the mode change bit GuestCtl0[MC] and clearing the field change disable bit GuestCtl0Ext[FCD].

+ This is called a Guest Hardware field change exception.

+ The Guest Hardware field change exception is triggered by a hardware change to one of 2 status fields, EXL that indicates an exception has happened and TS indicating that the TLB is about to be shutdown due to a duplicate entry being placed into it.

+ When an exception occurs the Guest context is update with any information that is normally updated in accordance with the exception type and EXL is set. The Root context is modified by setting Exception level bit Root.Status[EXL] , setting the Exception Code field, Root.Cause[ExcCode] to “Guest Exit” and the Root.GuestClt[GExcCode] to indicate Guest Hardware Field Change Exception.

+ Next a mode switch will be done so the Hypervisor can do what it needs to do in accordance with the systems security policy.

+ Once the Hypervisor is finished and does an ERET the mode switches and the Guest begins processing the exception.

+ Once the Guest is finished and it does a ERET then the mode switches again to Root execution before returning to normal Guest execution. Once the Hypervisor is finished it does a ERET and the mode switches back to the Guest.

Exceptions

- **Guest Reserve Instruction Redirect**
  - GuestCtl0[RI]=1
  - Guest mode Reserved Instruction Exception
  - Exception taken in Root
  - Guest context not changed
  - Root.BadInstr – updated with instruction encoding
  - Root.BadInstrP – updated with instruction address
  - Root.Cause[ExcCode] = GuestException (27, 0x1B)
  - GuestCtl0[GExcCode] = GuestReserveinstructionRedirect (3, 0x03)

MIPS 28

If the Reserve instruction bit GuestCtl0[RI] is set a Guest generated Reserve Instruction Exception, will be taken in Root mode with no context change to the Guest. For example the execution of a DSP instruction in Guest mode when the Config3[DSP2P or DSPP] bits are not set indicating that the DSP ASE is not implemented will cause a Reserve Instruction Exception.

Exceptions

- **Hypercall Exception**
  - Caused by the execution of the hypcall instruction
    - Recommended Guest mode execution only
  - Assembler: hypcall *immediate code value*
    - *immediate code value* 0 -3FF placed in Root.BadInstr
  - Root.Cause[ExcCode] = GuestException (27, 0x1B)
  - GuestCtl0[GExcCode] = Hypercall (2, 0x02)

MIPS 29

The Guest OS can exit to the Hypervisor with the instruction hypercall. The instruction takes a 10 bit immediate value which is placed in the CP0 Root.BadInstr register.

MIPS

MIPS Training
Part 2

Virtualization Module MIPS32®

www.mips.com

This course section covers the MIPS32 Architecture Virtualization Module for MIPS processors cores

Interrupts

- **This section covers interrupt delivery in VM:**
  - Three methods of delivery for interrupts
    - Interrupts for Root, handled by Hypervisor
    - Interrupts passed through or directed to Guest, handled by Guest OS
    - Root asserted Guest interrupts, handled by Guest OS
  - Two modes of interrupts
    - IV – Interrupt Vectored, vectors 0 – 5
    - EIC – External Interrupt Controller vectors 1 - 255

MIPS 31

This section will cover how interrupts are delivered using the Virtualization Module.

+The Hypervisor can configure the system to deliver interrupts in 3 different ways

+ First in can program itself to receive the interrupt directly using legacy methods

+ It can configure the interrupt to go directly to the Guest OS without any Hypervisor intervention. It does this in 2 different ways depending on the type of interrupt. I'll go into details later.

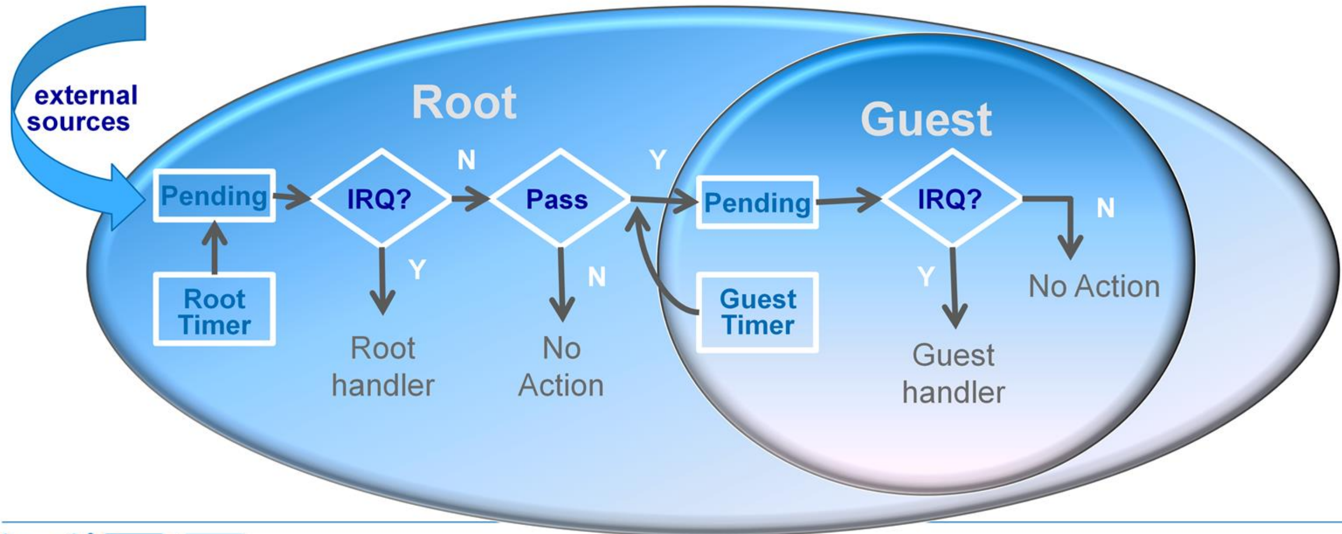
+ and last it can choose to receive the interrupt itself but then pass the interrupt on to the Guest OS

+ The Virtualization Module has to deal with the 2 modes of interrupts

+ Vectored interrupts where the is a 1 to 1 connection with a interrupt pin and an interrupt vector bit in the interrupt priority level field or IPL of the CP0 Cause register

+ Or Using an External interrupt controller that asserts vectors numbers into the Request Interrupt Priority Level field or RIPL of the CP0 Cause Register

The following slides will cover the 2 modes and how each is configured and works with the three delivery methods .



First I'll cover the Vectored Interrupt mode. I'm going to show you the big picture first and then I'll go into details on the setup and inter workings of Vectored interrupt Mode for Virtualization.

+ To start there are the external interrupts sources coming in to become a pending interrupt.

+ Next the pending interrupt is checked against the Root interrupt mask

+ and if it is masked then the Root will receive the interrupt

+ If it is not then a pass through mask is checked to see if it is to be passed through to the Guest

+ If that is set not nothing happens

+if it is set it moves along to pending for the Guest

+ the Guest interrupt mask is checked

+ if it is not set nothing happens

+ if it is set then it gets handled by the Guest OS

+ The Root Timer and the Guest Timer are handled in the same way

Interrupts

Vectored Interrupt Mode

- **Root assignment of External Interrupt**
  - Setting the interrupt mask (IM) bit associated with the interrupt pin in the CP0 Root.Status register.
  - Interrupt will be serviced in Root mode by Hypervisor.
- **Guest assignment of External Interrupt**
  - Setting the interrupt mask (IM) bit associated with the interrupt pin in the CP0 Guest.Status register.
  - Setting the associated bit in the Root.GuestCtl0[PIP] “Pending Interrupt Pass-through”
  - Interrupt will be serviced in Guest mode by Guest OS

MIPS 33

The assignment of an interrupt to Root is done by setting the interrupt bit associated with the interrupt pin in the Root.Status[IM] field to enable the interrupt to Root. This interrupt will interrupt the core place it into Root mode and start executing at the vector for the interrupt.

The assignment of an interrupt to a Guest is done by setting the interrupt bit associated with the interrupt pin in Guest.Status[IM] and setting the same bit in the “Pending Interrupt Pass-through” (PIP) field of the Root.GuestCtl0 register.

Interrupts

Vectored Interrupt Mode

- **Root asserted Guest interrupts**
  - Hypervisor monitoring of Guest interrupts
  - Hypervisor injecting a virtual interrupt into Guest context
  - Hypervisor receives a Guest interrupt for a Guest not resident

MIPS

The last method of delivering an interrupt to a Guest is through Root intervention. This is called Root asserted Guest interrupts. There are 3 reasons this might need to be done.

+ First if the Hypervisor wants to monitor Guest interrupts it can receive the interrupt and do what ever it needs then pass the interrupt along to the Guest.

+ second the Hypervisor my want to send a virtual interrupt to the Guest, like a pseudo device.

+ last When a Guest is not resident its interrupts will be redirected to the Hypervisor. When the Hypervisor receives an interrupt for a non resident Guest it will need to context switch the current Guest out and switch in the Guest that should receive the interrupt and set the interrupt up so the Guest will receive it.

The next slides will go into how this is done.

Interrupts

Vectored Interrupt Mode

- Hypervisor monitoring of Guest interrupts
  - Root context receives interrupt and does what it needs to do
  - Transfers the interrupt to the current Guest
    - Set the associated Hardware Clear (HC) in the Root.GuestCtl2 register will cause the associated “Interrupt Pending” (IP) bit in the Root.Cause register to be cleared when the Guest.Cause is de-asserted
    - Set the associated “Virtual Interrupt Pending” (VIP) bit in the GuestCtl2 register asserts the interrupt in the Guest once the ERET is done in the Hypervisor interrupt handler.

MIPS 35

If the Hypervisor is monitoring the Guest interrupts it will first receive the interrupt as if it were a Root context interrupt.

+ Then it will do what processing it needs to do.

+ Once that is done the Hypervisor will set the “Hardware Clear” or HC bit in the Root.GuestCtl2 register that is associated with the interrupt pin. This is necessary so the associated bit in the Root.Cause register will be cleared when the associated Guest.Cause bit is cleared by the Guest OS when it de-asserts the interrupt.

+ Next the Hypervisor will set the associated bit in the “Virtual Interrupt Pending” field of the Root.GuestCtl2 register. Once Guest Execution is enabled the Guest OS will receive the interrupt as long as the Guest.Status[IM] field is set to enable the particular interrupt.

If the Hypervisor wants to cause an interrupt in the Guest

+ it should make sure the Hardware Clear bit associated with the virtual interrupt pin is not set because there is no association with the Root.Cause register since this is not an external interrupt.

+ Next the Hypervisor will set the associated bit in the "Virtual Interrupt Pending" field of the Root.GuestCtl2 register. Once Guest Execution is enabled the Guest OS will receive the interrupt as long as the Guest.Status[IM] field is set to enable the particular interrupt.

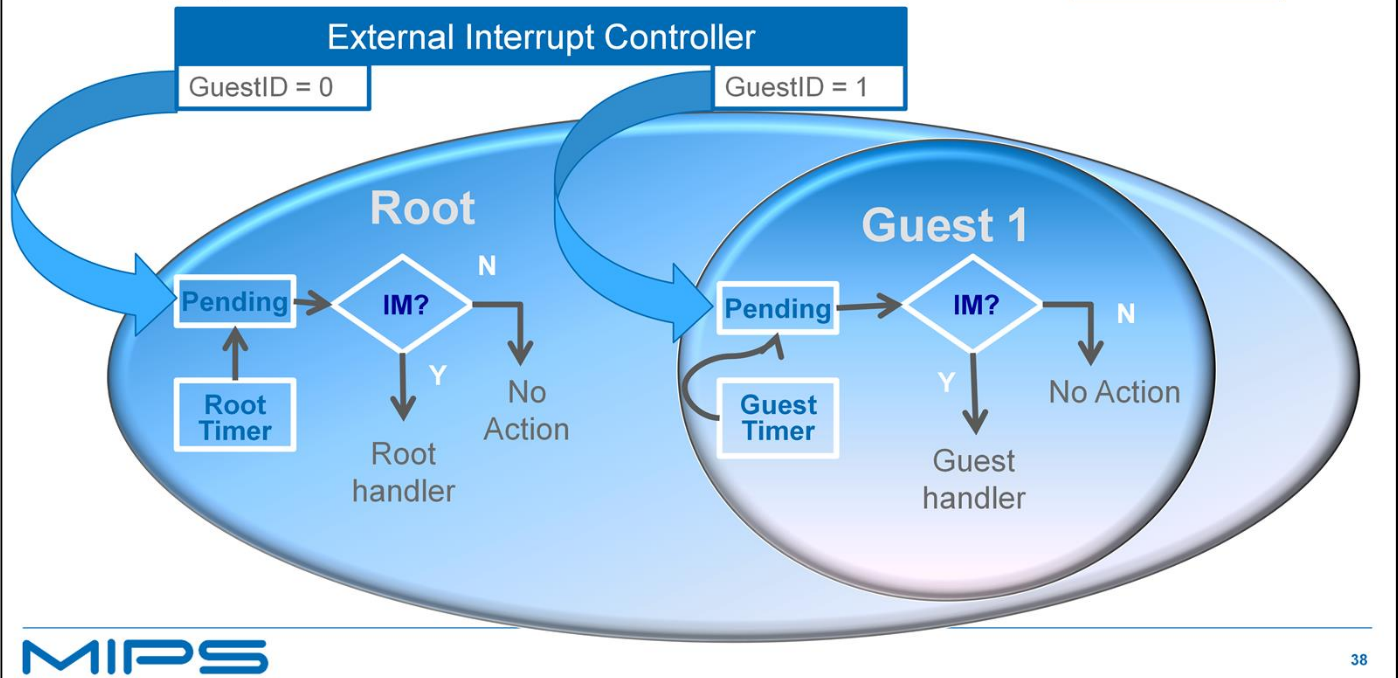
Interrupts

Vectored Interrupt Mode

- Hypervisor receives a Guest interrupt for a Guest not resident
  - Context switch in the target Guest
  - Transfers the interrupt to the Guest
    - Setting the associated Hardware Clear in the Root.GuestCtl2[HC] field will cause the associated "Interrupt Pending" bit in the Root.Cause[IP] field to be cleared when the bit in the Guest.Cause[IP] is de-asserted.
    - Setting the associated "Virtual Interrupt Pending" bit in the GuestCtl2[VIP] field asserts the interrupt in the Guest once the ERET is done in the Hypervisor interrupt handler.

MIPS

If the Hypervisor receives a Guest interrupt due to the fact that the target Guest is not the current Guest executing then the Hypervisor needs to first context switch in the target Guest and proceed to set the Guest up to receive the interrupt.



Now I'm gong to switch to explaining External Interrupt Controller Mode for Virtualization. The Virtualization Module requires a Guest ID associated with the interrupt. The Guest ID will be used to direct the interrupt directly to the Root interrupt bus if it is 0 or the Guest interrupt if the Guest id of the interrupt matches the resident Guest. There will be more on Guest IDs that don't match the resident Guest in the next slide.

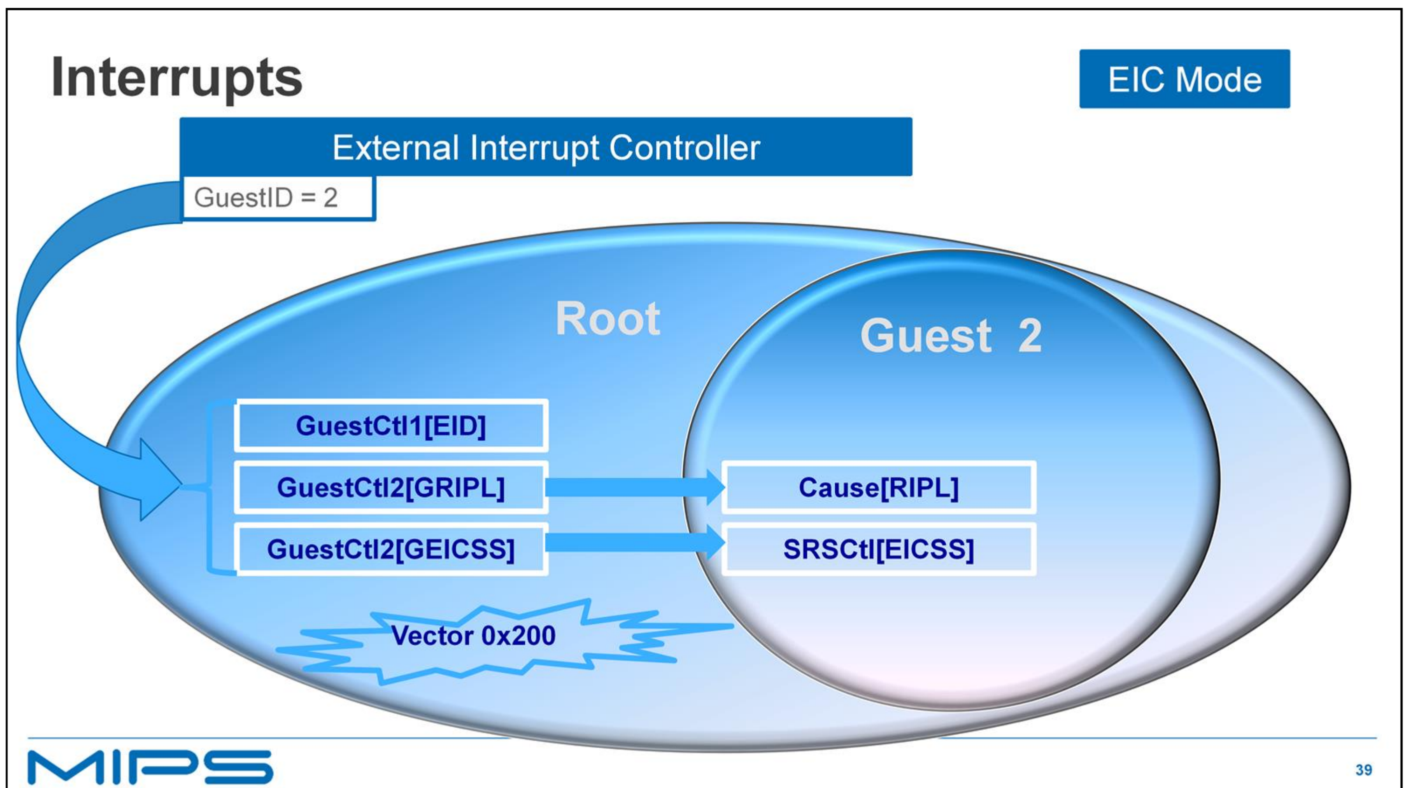
+ To start external interrupts come into the Root or Guest interrupt bus to become a pending interrupt.

+ Next the pending interrupt is checked against the associated interrupt mask

+ and if it is masked the associated context will receive the interrupt

+ If it is not nothing happens

+ The Root Timer and the Guest Timer are also routed directly to the associated interrupt bus



Now for a look at what happens if the interrupt is for a Guest that is not resident.

+ In this example the resident Guest is Guest 1 and the interrupt is for Guest 2. The External Interrupt Controller will check its target Guest ID against the current GuestCtl1[ID] and see that they do not match.

+ It will then deliver the interrupt to the Root bus and write the Guest intended to the External Interrupt Controller Guest ID field Root.GuestCtl1[EID], write the Request Interrupt Priority to the Guest Request Interrupt Priority field Root.GuestCtl2[GRIPL] register and write the EIC Shadow register Set number to the Guest EIC Shadow Set Field RootGuestCtl2[GEICSS].

+Then it interrupts the Root Context using interrupt vector 0x200. At that interrupt vector the Hypervisor knows the interrupt is for a non resident Guest.

+ The hypervisor uses the External Interrupt Controller Guest ID to context switch to that Guest

+Once the hypervisor has done the swap it will execute a return from exception which causes the processor to enter Guest Execution mode. On Guest entry the processor copies the Guest Request Interrupt Priority to the Guest.Context Cause[RIPL] field and the Guest EIC Shadow Set to the Guest.SRSCtl[EICSS] field. Once the Root interrupt does a Exception return the Guest 2 will be interrupted.

Context Switch from one Guest to another

- **Root can inject a timer interrupt in Guest context**
  - Virtual timer, maintained by Root to trigger when a Switch out of the Guest context
  - Hypervisor sets the Timer Interrupt (TI) bit to 1 in the Guest.Cause register
  - Guest OS takes the interrupt once Guest execution mode starts up again

MIPS 40

In systems where there are multiple Guest Contexts the Hypervisor should maintain a Virtual Timer to Switch out contexts.

+If the timer triggers while the Guest Context is switched out the Hypervisor should switch in the Guest context. The Hypervisor will need to set the timer interrupt bit in the Guest.Cause Register before completing the context switch.

+The Guest OS will take the timer interrupt once Guest execution is resumed.

Interrupts

- **Performance counters**
  - Only one set of Performance Counters
    - Located in Root Context never implemented in a Guest context
    - Hypervisor enable the Guest Context with access to the performance counters by setting Performance Counter field, Guest.Config1[PC] to 1
    - To enable a specific counter set the Performance Event Class field, PerfCnt [EC] to 2.
      - Guest will be able to access the counter and control it
      - Counter overflow exceptions will happen in the Guest context

MIPS

There is only one set of performance counters per processor (per VPE in a MT system). These are located in the Root context. The Hypervisor can configure a set of counters to be controlled by a Guest context.

+ The Hypervisor enables Guest use of performance counters by setting the Guest.config1[PC] bit in the Guest context. It should do this when it starts the Guest for the first time and not change this value so the Guest can read this when it boots.

+ The Hypervisor can control which set of performance counters a Guest has access to by using the Specific performance counters Event Class field. This field is only visible in the Root context a setting of 2 means that only Guest events will be counted, the counter is controllable in the Guest context and counter overflow exceptions will be take in the Guest Context.

Hypervisor Resource Control

- **Shadow Register Sets**
- **If the core has Shadow Register Sets**
  - Hypervisor can allocate some to each Guest Context
  - Set the Guest Lowest Shadow Set number field in Root.GuestCtl3[GLSS]
  - Set the Highest Shadow Set number field in the Guest.SRSCtl[HSS] register
  - Guest has the range between GLSS and GLSS+HSS.
  - Hardware offsets the Previous Shadow Set field, Guest.SRSCtl[PSS] by GLSS

MIPS 42

The Hypervisor can control access to the Cores resources these next slides will cover how that is done.

If the core is configured with Shadow Register sets then the Hypervisor can allocate them to Guest contexts. It does this by defining a range of shadow registers for a particular Guest.

+First it sets the Guest lowest Shadow Set field in the GuestCtl3 register with the starting shadow set number for the low end of the range.

+Then it sets the normally read only, Highest Shadow Set field in the Guest's SRSCtl register with the number of shadow set being allocated to the Guest.

+ The range for the Guest starts at GLSS and ends at GLSS+HSS.

+The hardware will automatically offset the starting number to the first shadow set for the Guest to 0.

Note: It will make context switching faster if the Hypervisor allocates at least one shadow set to each Guest to use as the Guest's main General Purpose Registers because the Hypervisor will not need to save the GPRs on a Guest context switch.

Hypervisor Resource Control

- **Multiplier Result Registers**
  - All Guest and Root contexts share the multiplier result registers LO and HI so the Hypervisor will need to save these registers when doing a Guest context switch.

MIPS 43

The Hi and Lo Multiplier registers are shared by both Guests and Root context so the Hypervisor must save these registers when switching Guest contexts.

Hypervisor Resource Control

- **DSP Module (if implemented)**
  - The Guest and Root contexts all share the DSP Module so the Hypervisor must save the HI/LO and the 3 HI/LO accumulator registers.
  - The Hypervisor sets the DSP Present field, Guest.Config3[DSPP] and to give a Guest access to the DSP Module.
  - If present but not enabled in the Guest Context (Guest.Status[MX] = 0) a DSP Module state unusable exception is taken in Guest mode.
  - Then If present but not enabled in the Root Context (Root.Status[MX] = 0) a DSP Module state unusable exception is taken in Root mode.

MIPS

If the DSP module is present it is shared by both Guests and Root contexts. The Hypervisor must save the HI/LO and the 3 HI/LO accumulator registers if enabled for the Guest Context.

+ The Hypervisor controls a Guest access to the DSP module by setting the normally read only DSP Present and DSP revision 2 Present fields in the Guest's Config3 register.

+ When a DSP instruction is issued and the DSP module is not enabled in the Guest's Status register then a DSP Module state unusable exception is taken in Guest mode.

+ If it is enabled in the Guest status register but not enabled in Root's Status register then a DSP Module state unusable exception is taken in Root mode.

Hypervisor Resource Control

- **Floating Point Unit**
  - The Guest and Root contexts all share the Floating Point Unit so the Hypervisor must save the floating point registers.
  - The Hypervisor sets the Floating Point field, Guest.Config1[FR] register to give a Guest access to the Floating Point Unit.
  - If present but not enabled in the Guest Context (Guest.Status[CU1] = 0) a coprocessor unusable exception is taken in Guest mode.
  - Then If present but not enabled in the Root Context (Root.Status[CU1] = 0) a coprocessor unusable exception is taken in Root mode.
- **Co-Processor 2 follows the same basic rules.**

MIPS 45

If a Floating Point Unit is present it is shared by both Guests and Root contexts. The Hypervisor must save the floating point registers if enabled in the Guest context.

+ The Hypervisor controls a Guest access to the Floating Point Unit by setting the normally read only Floating Point Present field in the Guest's Config1 register.

+ When a Floating Point instruction is issued and the Floating Point Unit is not enabled in the Guest's Status register then a coprocessor unusable exception is taken in Guest mode.

+ If it is not enabled in the Guest status register and not enabled in Root's Status register then a coprocessor unusable exception is taken in Root mode.

+ Co-Processor 2 follows the same rules but uses the Co-Processor 2 fields in the Config1 and Status registers

Hypervisor Resource Control

- **Floating Point register mode**
  - Shared.Status[FR] Controls the register mode of the processor
    - FR = 0 Floating-point registers can contain any 32-bit data type. 64-bit data types are stored in even-odd pairs of registers so there are 32, 32 bit floating point registers that need to be saved on a Guest Context switch.
    - FR = 1 Floating-point registers can contain any data type, 32 or 64 so there are 32, 64 bit floating point registers to save.
  - Shared.Status[FR] can be changed if Guest.Config5[UFR] = 1
  - Guest mode to change Config5[UFR] causes a GFSC exception.
    - Alerts the Hypervisor to check Guest.Status[FR] before saving Floating point registers on a contest switch.

MIPS 46

The floating point co processor is shared by the Root and Guest Contexts so the Hypervisor will need to save the floating point registers on a Guest context switch.

The Hypervisor will need to read the Floating Point Register mode field in the Status register to decide if 32 or 64 bit floating point registers need to be save on a Guest context switch.

+ If FR = 0 there are 32 32 bit registers to save

+ if FR =1 there are 32 64 bit registers to save

+ The ability to change FR in user mode is controlled by the setting of the UFR bit in the Config5 Register. If this bit is set then code operating in Guest User mode can make a change to the Guest.Status[FR] which is not visible to the Hypervisor.

+ To alert the Hypervisor when a Guest.user might have changed FR a GFSC exception is raised to Root on any access to Guest.Config5[UFR] field.

+If this access changes UFR to a 1 then the Hypervisor will need to check Guest.Status[FR] each time before it saves the floating point registers to know the size of the registers to save.

Hypervisor Resource Control

- **MSA (MIPS SIMD Architecture)**
  - The Guest and Root contexts all share the MSA Module, so the Hypervisor will need to save the MSA registers on a Guest Context switch.
  - The Hypervisor sets the MSA Enable field Guest.Config5[MSAEn] to give a Guest access to the MSA Module.
  - If present but not enabled in the Guest Context (Guest.Conifg5[MSAEn] = 0) MSA Module state unusable exception is taken in Guest mode.
  - Then If present but not enabled in the Root Context (Root. Conifg5[MSAEn] = 0) MSA Module state unusable exception is taken in Root mode.

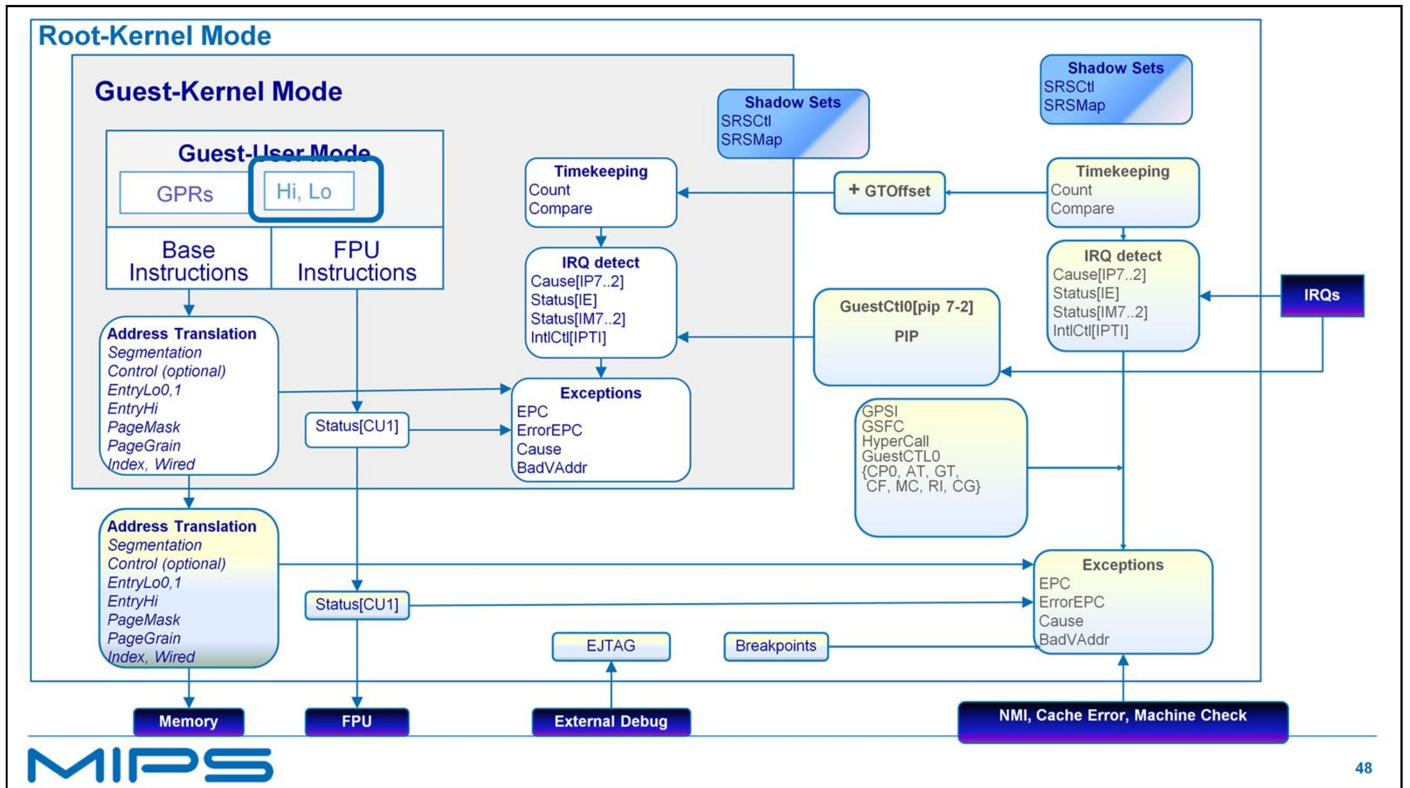
MIPS 47

If the MSA module is present it is shared by both Guests and Root contexts. The HI/LO registers and the 3 addition HI/LI accumulator pairs need to be saved on a Guest context switch if the MSA module has been enabled for that Guest.

+ The Hypervisor controls a Guest access to the MSA module by setting the normally read only MSA field in the Guest's Config5 register.

+ When a MSA instruction is issued from Guest mode and the MSA module is not enabled then a MSA Module state unusable exception is taken in Guest mode.

+ If it is enabled in the Guest Config5 register and not enabled in Root's Status register then a MSA Module state unusable exception is taken in Root mode.



Here is the whole picture to review.

+ If you have shadow set it's a good idea to assign at lease one for Guest use so when the processor switches to Root mode or when you context switch out a Guest you will not have to save the GPRs.

+ Multiplier result registers are accessible in user and kernel modes and are not protected. These shared registers must be saved/restored if necessary.

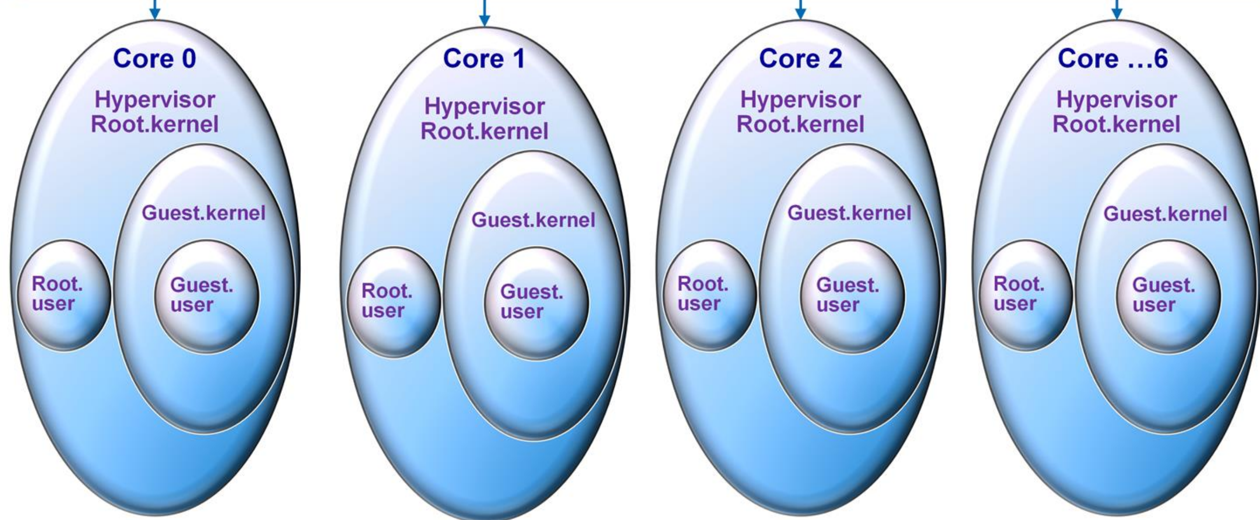
+ Address translation is performed first using the Guest TLB, enabled by setting the GuestCtl0[AT] to 1 or 3, then through the Root TLB. Note that Root context Segment Configurations are not used when translating a Guest Physical address - the Root context TLB translates every address from the Guest.

+ Access to the FPU is first checked in the Guest context and then in the Root context.

+ Exceptions detected by the Guest context are handled in Guest mode.

+ Guest timekeeping, and interrupts can be passed through by the Root context so that the Guest OS can handle them directly

+ Exceptions detected by the Root context are handled in Root mode by the Hypervisor.

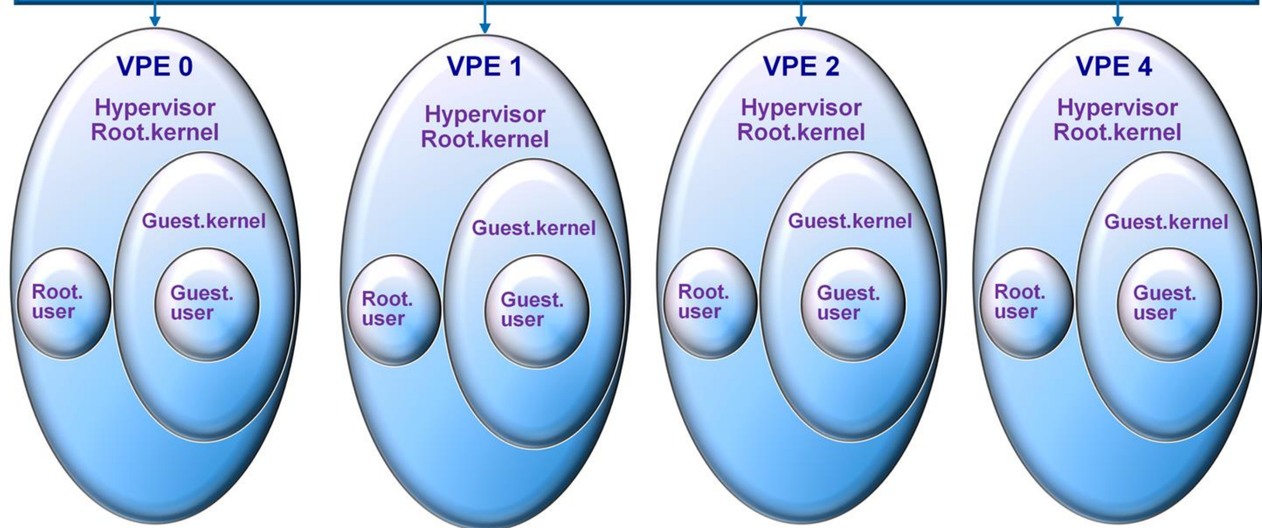


In a Multi Core system

+ Each Processor behaves just like a Single processor with Root and Guest Contexts.

+ Each Processor runs a distinct Hypervisor

+ Hypervisor instances communicate with each other to achieve shared goals, as in a traditional SMP system.. This can be done through shared memory segments, Inter thread Communication, inter-processor interrupts or a combination there of.



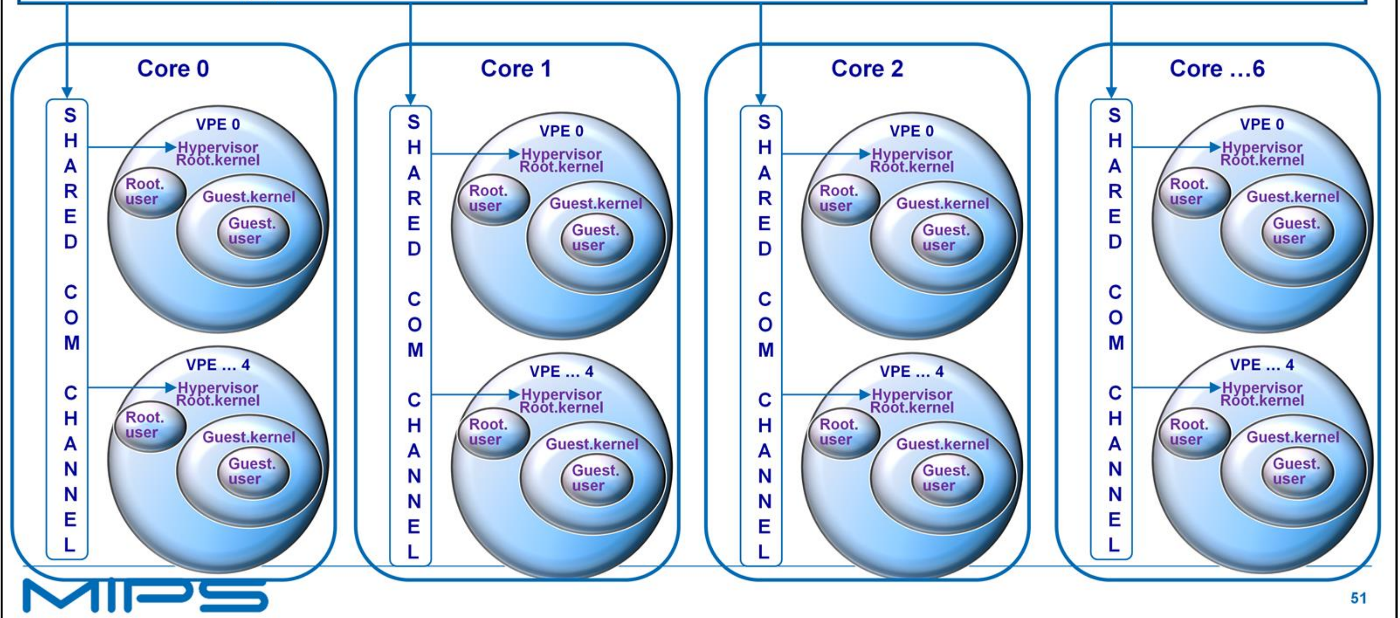
In a Multi Threaded Single Core system

+ Each VPE behaves just like a single processor with a Root and Guest Contexts.

+ Each VPE runs a distinct Hypervisor

+ The same as in a Mulit Core system the Hypervisor instances communicate with each other to achieve shared goals, as in a traditional SMP system.

Note: MT Module registers are never present in Guest CP0 context so a Guest cannot emulate a MT core.



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. All Hypervisors would communicate with each other over a shared channel.

CP0 Registers

- Root CP0 registers
  - Base set depending on processor implementation
  - Plus registers introduced by Virtualization module for Guest control
- 1 set of Guest CP0 registers
  - Base set depending on processor implementation - initial state defaults to the hardware reset state
  - Reconfigurable by Hypervisor (Root mode)
  - Access by Guest to CP0 registers can be monitored by Hypervisor

MIPS 52

This last section goes into more detail about the CP0 registers changes for the Virtualization Module. It covers the Added CP0 registers or CP0 registers with new fields.

+ The Root CP0 registers are a superset of the normal Processors CP0 registers with the addition of new registers to control Guest mode and some registers which have added fields. These changes are only found in the Root CP0 registers and not the Guest CP0 set.

+ The Guest CP0 registers are defaulted to the same values as the Root CP0 registers on the initial boot up. The Guest CP0 registers are reconfigurable by the Hypervisor. It can change some of the register fields in the Guest CP0 registers that would normally have been hard coded and read only. This allows the Hypervisor to give a Guest OS subset of resources available in the actual processor.

CP0 Registers

- **New registers introduced for Root CP0 set**

Register Number (number, select)	Register Name	Description
12,6	GuestCtl0	Controls Guest mode behavior
10,4	GuestCtl1	Guest IDs
10,5	GuestCtl2	Virtual Interrupts
10,6	GuestCtl3	Virtual Shadow Sets
11,4	GuestCtl0Ext	Extension to GuestCtl0
12,7	GTOffset	Offset for Guest timer value

MIPS

Here is a table of new registers added to the Root CP0 register set. These registers are only available to the Hypervisor in Root.Kernel mode. These registers configure the amount of control the Hypervisor will have over the Guest OS. The Guest OS will not be aware of any intervention by the Hypervisor.

Each will be covered in detail in the next slides.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
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	GExxCode	SFC2	SFC1

- **GM - 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.
  - 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.

MIPS 54

The next several slides will cover fields in the CP0 GuestCtl0 Register.

+The processor is in Guest mode when the Root Exception, Error, and Debug bits are all clear and the GM bit is set in the GusetCtl0 register.

+The Hypervisor will use this bit to transition the processor into Guest mode. It is recommended the Hypervisor use a return from exception instruction to enter Guest mode. The hypervisor sets the GM bit, stores the starting address for guest execution in the Root.EPC register and makes sure Root.Status[ERL]=0. Then it would execute an ERET instruction which will cause the program counter to be set to the address that is in the Root.EPC register and the exception-level bit EXL to be cleared in Root.Status. After the ERET instruction execution is completed, the processor will be in Guest mode fetching instructions from the PC.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **RI - Guest Reserved Instruction Redirect.**

  During Guest-mode execution:
  - 0 - Reserved Instruction exceptions are taken in Guest mode.
  - 1 - Reserved Instruction exceptions result in a Guest Reserved Instruction Redirect exception, taken in Root mode.

MIPS 55

The RI bit gives the Hypervisor control of what to do when a reserve instruction exception happens. A reserved instruction exception occurs when a reserved or undefined major opcode is execution or value in a function field is illegal . For example if a program were compiled to use hardware floating point and there is no hardware floating point hardware then a reserved instruction exception would be raised if it tries to execute a floating point instruction. In this example if the RI bit were 0 then the Guest OS can directly resolve the exception and for example emulate the floating point instruction. If RI were set to 1 the Hypervisor would receive an exception and take the appropriate action depending on the securities policies it follows.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
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	GExxCode	SFC2	SFC1

- **MC - Guest Mode-Change exception enable**

  During Guest-mode execution: 0 – No Change from classic MIPS

  - 1 – hardware initiated change to Guest.Status[EXL] will trigger a Guest Hardware Field Change Exception.
  - 1 - A software initiated change to Guest.Status[KSU] will trigger a Guest Software Field Change Exception.

MIPS 56

The MC bit gives the Hypervisor control when external hardware changes the status of the EXL (exception) field in the Guest Status register or Guest software changes the processor operation mode. The Hypervisor would use this exception to monitor when the Guest OS is externally forced into Guest.Kernel mode on exception or when the Guest OS transitions from User mode to Kernel which happens with the execution of a syscall instruction.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **CP0 - Guest access to Guest coprocessor 0.**

  During Guest-mode execution:
  - 1 – No Change from classic MIPS
  - 0 – Guest.kernel use of a Guest Privileged Sensitive instruction will trigger a Guest Privileged Sensitive Instruction exception..

MIPS 57

The CP0 bit gives the Hypervisor control over what happens when a Guest privileged sensitive instruction is executed. If CP0 is set to 1 Guest execution of privileged sensitive instructions are permitted subject to other configuration settings. If CP0 is cleared then any use of a privileged sensitive instruction will always cause an exception.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
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	GExxCode	SFC2	SFC1

- **AT - Guest Address Translation control.**
  - 1 - Guest MMU under Root control
  - 3 - Guest MMU under Guest control

MIPS

58

The Hypervisor can choose to directly control the Guest MMU. Setting AT to 1 allows Root to control Guest address translation directly. AT normally defaults to 3 allowing the Guest OS to control the Guest TLB however this is implementation dependent.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																		
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **GT - Timer register access**
  - 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 access to Count or Compare registers permitted

MIPS

If the GT bit is cleared the Hypervisor will monitor all accesses to the Guest count and compare registers.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **CG - Cache Instruction Guest-mode enable**
  - 0 - A Guest Privileged Sensitive Instruction exception will result from use the CACHE instruction.
  - 1 - The CACHE instruction can be used with an Effective Address Operand type of 'Address'. All other Address types will result in Guest Privileged Sensitive Instruction exception .

MIPS 60

The cache is not protected by Guest ID so entries in the cache should be monitored by the Hypervisor to be sure one Guest is not corrupting or snooping another's cache or the cache entries of the Hypervisor.

If CG is 0 then A Guest Privileged Sensitive Instruction exception will result any time a Guest issues a cache instruction.

If CG is 1 cache instructions that use an effective address will be allowed without causing an exception. This should be secure because the effective address is translated by the MMU to a physical address and the physical address is then used to address the cache. In this way only address that were assigned by the Hypervisor to the particular Guest can be accessed by this type of cache instruction.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **CF - Config register access**
  - 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

MIPS

61

Config0-7 register writes can be monitored by the Hypervisor. If cleared writes will cause Guest Privileged Sensitive instruction. If set writes by a Guest OS are permitted without causing an exception.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
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	GExxCode	SFC2	SFC1

- **G1 - GuestCtl1 register implemented.**
  - 0 - Unimplemented
  - 1 - Implemented.

MIPS 62

The G1 bit determines if the GuestCtl1 register has been Implemented. This determines if you can use GuestIDs. These IDs are used in identifying which Guest an external interrupt is for, which Guest a TLB entry is assigned to and which Guest OS is resident at the current time.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
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	GExxCode	SFC2	SFC1

- **Impl - Implementation defined.**
  - These bits are implementation dependent and not defined by the architecture.

These bits are implementation dependent and not defined by the architecture.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	l21 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	GExxCode	SFC2	SFC1

- **G0E - GuestCtl0[Ext] register implemented.**
  - 0 - Unimplemented
  - 1 - Implemented.

MIPS 64

The G0E bit determines if the GuestCtl0[Ext] register has been Implemented. This register gives the Hypervisor more control over cache instructions, CP0 state changes and writes to additional CP0 registers. More on this when I cover the GuestCtl0Ext register.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																		
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **PT - Pending Interrupt Pass-through feature**
  - 0 – Unimplemented GuestCtl0[PIP] not supported
  - 1 – Implemented GuestCtl0[PIP] supported

MIPS 65

The PT bit determines if the Pending Interrupt Pass-through feature has been Implemented. If this bit is set External interrupts are passed through from the Root context if enabled by the mask in the GuestCtl0[PIP] bits. This is the way the Hypervisor allows a Guest OS to directly handle a external interrupt with out Hypervisor intervention.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																		
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **ASE - Reserved for MCU Module Pending Interrupt Pass-through.**

MIPS 66

If the processor includes the MCU ASE module which expands the number of interrupt sources from 6 to 8 then these bits will be used as part of the Pending Interrupt Pass through mask.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
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	GExxCode	SFC2	SFC1

- **PIP – Pending Interrupt Pass-through Mask**
  - 0 - Corresponding interrupt request is not visible in Guest context
  - 1 - Corresponding interrupt request is visible in Guest context

MIPS

This is a mask of interrupts that will be passed through to the Guest OS without Hypervisor intervention if the processor is in non-EIC mode. If a bit is set that External interrupt will be directly handled by the resident Guest OS. This mask is enabled or disabled by the PT field in this same register.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **RAD – Root ASID De-alias mode**
  - 0 – Guest ID is used to de-alias both Guest and Root TLB entries
  - 1 - Root ASID is used to de-alias Root TLB entries when Guest TLB contains only one context at any given time

The Root ASID De-alias mode bit can be used if there is only one Guest possible then Hypervisor could use the ASID in the ROOT TLB to distinguish entries belonging to the Guest OS and Hypervisor. This is not recommended if there is more than one Guest possible because the Root TLB would need to be cleared when the Hypervisor switches between Guest contexts. In the case where RAD is set and the Global bit is also set for a TLB entry then that entry will be valid for both the Root and Guest contexts. You could also think of this as determining if the TLB is shared; if it is set to 1 the TLB is not sharable between different Guest OSs.

NOTE: If this bit is 0 then then the Hypervisor must use Guest IDs in the TLB entries so the G1 bit in this register must be set indicating the existence of the GuestCtl1 register which contains support for Guest IDs. Conversely if the bit is 1 then the G1 bit will be cleared and Guest ID cannot be used.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **DRG – Direct Root to guest TLB access**
  - 0 – No Direct Root to Guest TLB access
  - 1 - GuestCtl1.RID is used to de-alias TLB entries

MIPS 69

When the processor uses Guest IDs in the TLB entries (RAD = 0) and it is in Root mode but not in Error, Exception or Debug mode, setting the DRG bit will cause loads or stores to use TLB entries associated with the Root Control GuestID that is set in the GuestCtl1[RID] field.

In other words when GuestCtl0[DRG] field is set and the Hypervisor is performing TLB operations that are targeted to a specific Guest ID using the GuestCtl1.RID field. For example if the hypervisor (root mode) wants to use the entries in the TLB to directly access guest entries then the GuestCtl0.DRG bit should be set and the GuestCtl.RID set to the Guest ID of the guest entries it wants to access. Then loads and stores will use the GuestCtl.RID to select the correct TLB entry to use.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **G2 – *GuestCtl2 register implemented***
  - 0 – Unimplemented
  - 1 - Implemented.

MIPS

The G2 bit indicates the presents of a CP0 Root.GuestCtl2 register. This will always be set if the processor uses an External Interrupt Controller and may be set if it does not.

If the processor is not in EIC mode then the Hypervisor will use Root.GuestCtl2 register to assert and de-assert virtual interrupts to the resident Guest OS.

If the processor is in EIC mode then the Root.GuestCtl2 register will facilitate the Hypervisor with interrupts that should be routed to a Guest OS that is not currently resident.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
31	30	29	28	27 26	25	24	23	22	I21 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

Hypervisor Exception Cause Code	
0	Guest Privileged Sensitive instruction was attempted but not enabled from Guest.kernel mode
1	Guest Software Field Change event
2	Hypercall
3	Guest Reserved Instruction - Redirect GuestCtl0RI=1 and a reserved instruction exception was taken in Guest mode
8	Guest Virtual Address available but not the Guest Physical address on a Root TLB exception from Guest mode
9	Guest Hardware Field Change event
10	Guest Physical Address available on a Root TLB exception from Guest mode

71

When the Root.Cause[ExcCode] is set to GE a Hypervisor-intervention exception has occurred. The [GExcCode] field further qualifies this exception.

0 is a Guest Privileged Sensitive instruction; this happened 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 indicates a Guest Software Field Change Event happened; this was previously discussed in the exception section.

2 indicates a hypercall instruction was executed by the Guest.

3 indicates that Guest is getting a Reserved Instruction exception and the GuestCtl0[RI] bit is set causing the exception to be redirect to the Root context.

8 - GVA Indicates that a Guest mode access resulted in a Root TLB access and the Guest Physical Address is not available. Root.BadVaddr set to Guest Virtual address. This is allowed for a non-TLB refill exception; for example if a store was done to a clean page, Dirty bit not set so the translation in the TLB is correct and the exception is just telling the OS that the page is being written to for the first time. In this case an implementation is allowed to set Root.BadVaddr to the Guest Virtual address. The root TLB exception handler should check this bit and if it is set it must probe the Guest TLB entries using the TLBGP

instruction to find the index into the Guest TLB that contains the Guest Virtual Address. Then the handler would use the TLBGR instruction to read that indexed entry in the Guest TLB to determine the Guest Physical address. The handler would then use the Guest Physical address as the virtual address in finding the entry in the Root TLB and for this example, it can set that entries dirty bit.

NOTE: MIPS Core implementations use this method because it preserves the micro TLB entry and avoids a load miss for future loads.

9 indicates a Guest Hardware field change event happened.

And

10 - GPA indicates a Guest mode access resulted in a Root TLB access, the Guest Physical Address is available. Root.BadVaddr will be set to the Guest Physical address. This will always be the case for a guest TLB refill exception which will always set GPA in GuestCtl0GExcCode. If GPA is set, the root TBL exception handler can use Root.BadVaddr for the virtual address it needs to find a translation for.

MIPS

MIPS Training
END OF PART 2

Virtualization Module MIPS32®

www.mips.com

This course section covers the MIPS32 Architecture Virtualization Module for MIPS processors cores

MIPS

MIPS Training
Part 3

Virtualization Module MIPS32®

www.mips.com

This course section covers the MIPS32 Architecture Virtualization Module for MIPS processors cores

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																				
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **SFC2– Guest Software Field Change exception enable for Guest.Status[CU2]**
  - 0 - GSFC exception taken if Guest.Status[CU2] is modified by Guest
  - 1 - GSFC exception not taken if Guest.Status[CU2] modified by Guest

MIPS 74

The Hypervisor can allow the Guest OS to make changes to the Guest.Status[CU2] filed without intervention from the Hypervisor if the SFC2 field is set. Otherwise a change to the Guest.Status[CU2] field will result the a Field Change Exception to the Hypervisor. The CU2 field controls access to Co-Processor 2 and a Guest OS may want to limit access to one user process at a time. So when a Guest.user process causes a Coprocessor 2 unusable exception the Guest OS will need to do a context switch of the Coprocessor 2 registers. The security policy may find this acceptable and allow this without intervention from the Hypervisor by setting this the SCF2 bit.

CP0 Registers

GuestCtl0 Register (CP0 Register 12, Select 6)																			
31	30	29	28	27 26	25	24	23	22	I21 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	GExxCode	SFC2	SFC1

- **SFC1– Guest Software Field Change exception enable for Guest.Status[CU1]**
  - 0 - GSFC exception taken if [CU1] is modified by Guest
  - 1 - GSFC exception not taken if [CU1] modified by Guest

MIPS 75

Similar to the SFC2 field, the Hypervisor can allow the Guest OS to make changes to the Guest.Status[CU1] filed without intervention from the Hypervisor if the SFC1 field is set. Otherwise a change to the Guest.Status[CU1] field will result the a Field Change Exception to the Hypervisor. The CU1 field controls access to Co-Processor 1 which is by convention the FPU and a Guest OS may want to limit access to one user process at a time. So when a Guest.user process causes a Coprocessor 1 unusable exception the Guest OS will need to do a context switch of the FPU registers. The security policy may find this acceptable and allow this without intervention from the Hypervisor by setting this the SFC1 bit.

CP0 Registers

GuestCtl1 Register (CP0 Register 10, Select 4)			
31 24	23 16	15 8	7 0
EID	RID	0	ID

- **EID – written by EIC with Guest ID**
- **RID - Root control GuestID. Used by Root TLB operations, and when GuestCtl0[DRG]=1 in Root mode to access Guest TLB entries.**
- **ID - Guest control GuestID. Identifies resident Guest. Applies to Guest address translation.**

MIPS 76

The GuestCtl1 register contains 3 IDs

EID is the external Guest ID set by the External Interrupt Controller when a interrupt is raised for a non resident Guest. This is used to determine which Guest the interrupt is for. The Hypervisor can check the EID against the ID to determine if the Guest currently resident is the Guest the interrupt is intended for. If it isn't then the Hypervisor should swap in the intended Guest. This was covered in detail in the interrupt section.

RID is used when the GuestCtl0[DRG] field is set and the Hypervisor is performing TLB operations that are targeted to directly access a specific Guests entries given by this RID field and not Root TLB entries.

The GuestCtl.ID is used in Guest mode only to select TLB entries. In root mode the GuestCtl.ID indicates the ID of the current Guest context. When a hypervisor switches in a Guest context it changes the GuestCtl.ID and from that point forward all guest access to the TLB will only have access to TLB entries with a guest ID that matches the GuestCtl.ID.

CP0 Registers

GuestCtl2 Register (CP0 Register 10, Select 5) non EIC mode						
31 30	29 24	23 18	17 16	15 10	9 5	4 0
0	HC	0	0	VIP	0	0
0		0	0		0	0

- **HC - Hardware Clear for GuestCtl2[VIP] (bit wise mask)**
  - 0 – de-assertion external IRQ
  - 1 – de-assertion external IRQ and VIP
- **VIP - Virtual Interrupt Pending (bit wise mask)**
  - 0 – Guest.StatusIP bit cannot be set by Hypervisor
  - 1 – Guest.StatusIP bit can be set by Hypervisor

MIPS

77

The meaning of the fields in the GuestCtl2 register depend on the interrupt mode:

If the core is in Vectored Interrupt mode and not external interrupt controller mode this register is used by the Hypervisor to send a interrupt to the Guest OS. The Hypervisor will receive an external interrupt is if an interrupt happened for a Guest that was not resident. It would then switch in the intended Guest context. This was covered in the interrupt section.

+ Setting the associated Hardware Clear in the Root.GuestCtl2[HC] field will cause the associated "Interrupt Pending" bit in the Root.Cause[IP] register to be cleared when the bit in the Guest.Cause[IP] field is de-asserted. This field may be hardwired as a read only bit set to 1 or 0.

+ Set the associated "Virtual Interrupt Pending" bit in the GuestCtl2[VIP] Field asserts the interrupt in the Guest once the ERET is done in the Hypervisor interrupt handler.

CP0 Registers

GuestCtl2 Register (CP0 Register 10, Select 5) EIC mode								
31 30	29 24	23 22	21 18	17 16	15 10	9 5	4 0	
ASE	GRIPL	0	GEICS	0	GVEC			

- **GRIPL - Guest RIPL**
- **GEICSS - Guest EICSS**
- **GVEC - Guest Vector (only for Guest interrupts taken by Root)**
  - Root interrupt vector for non resident guest interrupt

If the core is in external interrupt controller mode:

These fields are written only when an interrupt received on the Root interrupt bus for a non resident Guest is taken.

The External interrupt Controller writes the follow fields:

+ GRIPL is the Requested Interrupt Priority Level in the MIPS EIC implementation this is treat as the vector number for the processor once it enters Guest mode.

+ GEICSS is the External Interrupt Controller Shadow set number

+ The GVEC is the Guest Vector Offset that could be provided by the EIC which would be used directly as the interrupt vector once the processor enters Guest mode. NOTE: The GVEC is not used in EIC implementations of current MIPS Cores; The GRIPL as describe previously is used as a vector instead.

Details for non resident Guest interrupts were provided in the Interrupt section.

CP0 Registers

GuestCtl3 Register (CP0 Register 10, Select 6)	
	3 0
	GLSS
	0

- **GLSS - Guest Lowest Shadow Set number.**

Guest Lowest Shadow Set number determines the lowest physical Shadow Set number assigned by Root to Guest. Guest SRSCtl[HSS] is the highest number shadow register set for the Guest in the Guest context. Guest is thus assigned physical Shadow Set range between GLSS and GLSS plus Guest SRSCtl[HSS]. NOTE: the Guest lowest shadow set is the set the Guest will use for GPRs. If you have shadow set it's a good idea to assign at lease one for Guest use so when the processor switches to Root mode or when you context switch out a Guest you will not have to save the GPRs.

CP0 Registers

GuestCtl0Ext Register (CP0 Register 11, Select 4)					
	4	3	2	1	0
	CGI	FDC	OG	BG	MG

- **CGI –CACHE/CACHEE Index Invalidate OK**
- **FCD –GSFC or GHFC event will not cause an exception.**
- **OG –Other GPSI Enable Enables GPSI for**
  - WREna, UserTraceData1, UserTraceData2, KScratch1 through KScratch6
- **BG –Bad register GPSI Enable Enables GPSI for**
  - BadVAddr, BadInstr, BadInstrP
- **MG - MMU GPSI Enable Enable Enables GPSI for**
  - Index, Random, EntryLo0, EntryLo1, Context, ContextConfig, PageMask, EntryHi

80

GuestCtl0Ext is an optional extension to GuestCtl0. It adds additional control features to the virtualization module. GuestCtl0G0E should be read by software to determine if GuestCtl0Ext is implemented.

CGI Allows execution of CACHE and CACHEE Index Invalidate operations in Guest mode. 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 Guest Privileged Sensitive Instruction (GPSI).

FCD Disables Guest Software/Hardware Field Change Exceptions (GSFC/GHFC).

OG adds Guest CP0 registers, UserLocal, WREna, UserTraceData1, UserTraceData2, KScratch1 through KScratch6 to the list of register access that will cause a Guest Privileged Sensitive Instruction exception.

BG Bad register GPSI Enable. adds Guest CP0 registers, BadVAddr, BadInstr, BadInstrP to the list of register access that will cause a Guest Privileged Sensitive Instruction exception.

MG adds Guest CP0 registers Index, Random, EntryLo0, EntryLo1, Context, ContextConfig, PageMask and EntryHi to the list of register access that will cause a Guest Privileged Sensitive Instruction exception.

CP0 Registers

- **GTOffset Register (CP0 Register 12, Select 7)**
  - The Guest.Count register is relative to the Root.Count register.
  - The GTOffset register is use to offset these count registers

MIPS 81

The Guest.Count register is always relative to the Root.Count register. The GTOffset register is used to offset the Root.Count register; the result can be read from the Guest.Count register.

Any writes to the Guest.Count register always result in a Guest Privileged Sensitive Instruction exception so the Hypervisor can make the proper adjustment and write it to the GTOffset.

CP0 Registers

- **Setting up Guest.Count**
  - Guest OS executes mtc0 $0, Count (zero out count register)
    - Causes GPSI exception (Root.GuestCtl0[GExcCode] = 0)
      - Hypervisor exception decodes Root.BadInstr to find out instruction was mtc0 $0 Count
      - Moves the Root.Count into a GPR and calculates its Two's complement
      - Adds to the two's complement value the value in the source register ($0)
      - Move the resulting value to Root.GTOffset
      - Add 4 to Root.BadInstrp and move it to Root.EPC
      - Clear hazard barrier (ehb instruction)
      - Execute ERET
  - Guest execution resumes at instruction after the mtc0

MIPS 82

Here is what happens when a Guest OS tries to set it's Count register.

When the Guest OS executes a mtc0 to write its Count register a Guest privileged sensitive instruction exception is raised giving control to the hypervisor.

+ The Hypervisor decodes the instruction

+ The hypervisor calculates the two's complement of the current value of the Root.Count Register

+ then it adds that value to the Guest setting

+ and moves the result to the GTOffset register

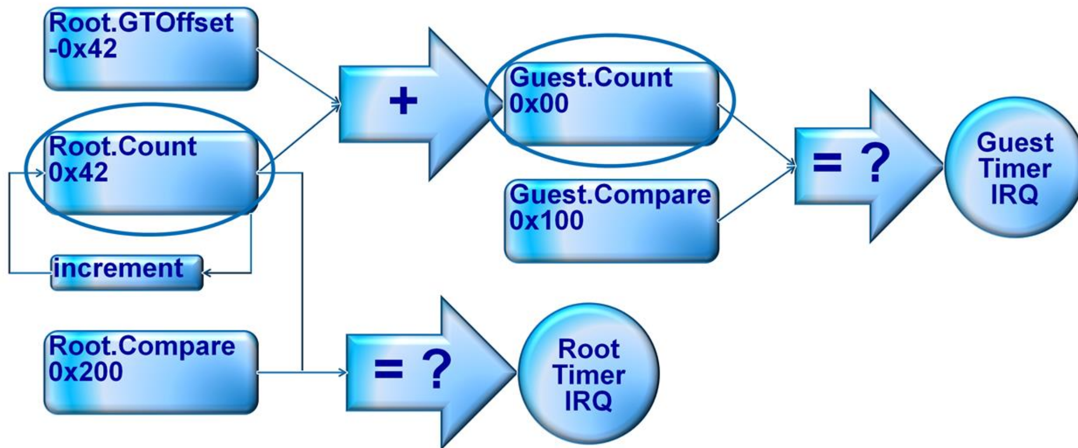
+ Then it calculates the new PC for the Guest to continue execution from, by adding 4 to the Bad Instruction Pointer and moving that value to the EPC register.

+ after issuing a Hazard barrier instruction to make sure the write of the GTOOffset register has taken effect it executes an Exception Return instruction.

+ Guest OS execution continues none the wiser.

CP0 Registers

To make Guest.Count 0:
Root.GTOffset = ((~Root.Count) + 1); //two's compliment
Guest.Count = 0 = (Root.Count + Root.GTOffset)



83

Here is a diagram of a timer interrupt in the Guest and Root Contexts. Assume that Root.Count register is a hex 42 to start

+ Let's assume to set up the timer the Guest will zero out the its Count register so the GTOffset will be set to a negative 42

+ and the Guest will set it's compare register to hex 100

+ and Root has sets its compare register to hex 200 at the same time.

+ each cycle the Root.Count register will increment

+ which has the effect of incrementing Guest.Count

+ 100 cycles later the Guest.Count will be 100 equaling the Guest.Compare register and the Guest OS will take a timer interrupt

+ 100 cycles after that the Root.Count will equal the Root.Compare register and the Hypervisor will get a timer interrupt

CP0 Registers

- **Cause Register (CP0 Register 13, Select 0)**
  - additional ExcCode value
    - GE - Hypervisor Exception (Guest Exit). Hypervisor-intervention exception occurred during Guest code execution. GuestCtl0[GExcCode] contains additional cause information.

MIPS

84

There is a new code for the Root.Cause Exception Code field to indicate a Guest Exit or GE. The Hypervisor exception routine would decode the Exception Code field and finding it is GE would then decode Guest Exception Code to further refine which exception it is. Guest Exception Code was cover earlier in this section.

CP0 Registers

- **Configuration Register 3 (CP0 Register 16, Select 3)**
  - Additional Field
  - VZ – bit 23, MIPS® Virtualization Module implemented. This bit indicates whether the Virtualization Module is present. (Root Context only).

MIPS

The Root.Config3 register has an new field called VZ which when set indicates that the Virtualization is implemented for the processor.

CP0 Registers

- **New instructions**
  - MFGC0 – Move from Guest CP0 register to GPR
  - MTGC0 – Move to Guest CP0 register from GPR

MIPS

86

There are 2 new instructions to move from and to the Guest set of CP0 registers to a General purpose register.

CP0 Registers

- **Guest CP0 Read-only Config fields writeable by Root**
  - Only if also present in the Root CP0 Context.

Register	Field	Purpose
Config1	C2	Co-processor 2 implemented
Config1	PC	Performance Counter register implemented
Config1	WR	Watch Registers implemented
Config1	FP	FPU implemented
Config3	DSPP/DSP2P	DSP ASE implemented
Config3	ITL	Iflow trace implemented
Config3	CDMM	Common device memory map implemented

MIPS 87

The Guest CP0 Context is somewhat configurable by Root in that it can be a subset of the Root context. Some of the fields in the configuration registers of the Guest context that are normally read only, can be de-configured by the Hypervisor. The intent is for the Hypervisor to do this de-configuration before starting the Guest code and not change it once the Guest has started. The Virtualization architecture allows for many of the Config registers to have many writable fields by Root however, the Actual MIPS IP Cores implementations limit the Root writeable fields to the list shown here.

This ends the Virtualization Module section. Writable

MIPS

MIPS Training
End of Part 3

Virtualization Module MIPS32®

www.mips.com

This course section covers the MIPS32 Architecture Virtualization Module for MIPS processors cores