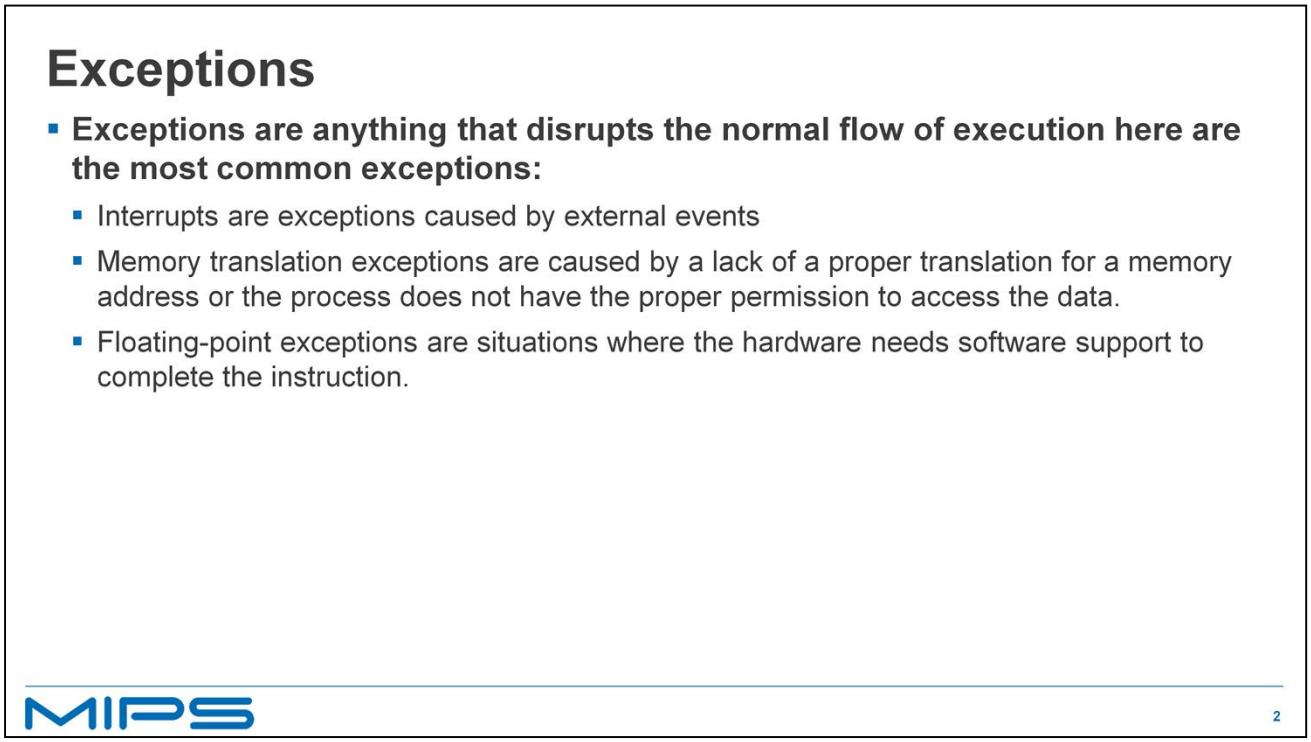
This section covers Exceptions.



+ An exception is an event the disrupts the normal flow of the execution of your code.

The CPU the exception routine is a special piece of code that tries to figure out what is wrong by check its status, see if it can be corrected and then continue along executing the normal code like nothing happened.

Here are some of the most common causes of an exception:

+ An interrupt is caused by a device that is external to the CPU. For example a clock interrupt or a packet of information coming in from the network. It's like a delivery man ringing your doorbell with a package for you. You'll go to the door sign for the package maybe open it. Eventually you'll get back to doing what ever it was before the doorbell rang.

+ Another common cause is a TLB miss and the exception needs to update the TLB with the proper translation.

+ A floating point exception which happens when the CPU doesn't know how to execute the instruction it has been given.

Exceptions

- **Exceptions continued:**
  - Program or hardware-detected errors: Such as nonexistent instructions, instructions that are illegal at user privilege level, coprocessor instructions executed with the appropriate SR flag disabled, integer overflow, address alignment errors, and accesses outside kuseg in user-mode.
  - Cache parity errors
  - System Calls
  - Full table can be found in the Software Users Manual of the core you are using. See "Cause Register ExcCode Field" Table

MIPS

+ Program or hardware detected errors most of the time these are problems with the execution of an instruction. For example these exceptions happen when a user mode process tries to execute an instruction that only can be executed in kernel mode. Sometimes a programming error can cause a code area to be over written with data so the instructions in that area are now garbage so when the CPU tries to execute this code it will generate this exception.

+ If your cache is designed for it the cache can generate a exception when the CPU access a cashed instruction or data and the parity isn't correct.

+ A system call is a way for a user mode program to transfer control to a OS running in kernel mode usually to request a OS service for example writing data to a disk drive.

+ There are more exceptions consult the Software User's manual for the core you are using for the full list.

Exceptions

- **Precise Exceptions**
  - Most exceptions are precise in that the instruction that caused the exception can be determined or there is a correct place to restart after the exception is handled.
    - All instructions in the pipeline before the exception will be completed.
    - All other instructions in the pipeline including the offending instruction will be restarted when the processor returns from exception processing.
    - Exceptions in a Branch or Jump Delay slot will be restarted at the branch.

MIPS

4

Determining where the exception happened can be an issue. You may need to know what instruction caused the exception to be able to fix it and you also need to know where to restart normal execution.

It's the difference between falling asleep while reading a book, when you wakeup you usually know what happen, you fell asleep, and you can figure out where you left off in your book. Now say you were reading and you get knocked out when you come to you can't figure out what happened because you don't know what hit you and you may not even remember what you were doing.

+Luckily most of the time the CPU can tell you precisely the instruction you were executing so you can determine what happened and where to restart normal execution. These are called Precise exceptions.

+ When a precise exception happens you can be sure that all instructions in the execution stream before the exception is reported have been executed.

+ On return from execution handler the CPU will start execution with the instruction that was set to execute at the time the exception happened.

+ To not cause any ambiguous execution, instructions in branch or jump delay slots are paired with the branch or jump. The return from exception execution will always execute the branch or jump even if the exception happened due to the instruction in the delay slot.

Exceptions

- **Non-precise Exceptions**
  - Instruction that caused the exception cannot be determined automatically by the CPU.
    - Bus (EC and OCP) Error. (Except for M5100 Cores)
    - Cache Errors
    - L2 Cache Error

MIPS

There are times the CPU cannot precisely determine what instruction caused the exception or where to restart execution.

+ It can only tell you what instruction was executing when the exception was detected An exception is imprecise when EPC/ErrorEPC/DEPC does not point to the instruction that caused the exception.

For example, if a load instruction misses in all of the caches for the requested data, and the cache hierarchy is non-blocking, execution may proceed pass the load. An interrupt may be recognized and accepted on an instruction subsequent to the load. While the interrupt handler is being executed, the response of the load returns and the response signals a Bus Error. In that case, a nested exception would occur, but the EPC for the bus error would not hold the address of the faulting load instruction. If the EXL bit is set at the time the Bus Error exception is recognized, the EPC would not be updated: for this case, the EPC would point to an instruction within the interrupt handler. A similar case can occur for late-arriving Floating-Point exceptions.

+ A bus, Cache and L2 Cache errors can be one of these exceptions because a load is non blocking and allows other instructions to execute until a instruction tries to execute that is dependent on the data being loaded. So the bus error for the load can occur on another instruction. A

fetch can cause a bus error but since most cores fetch ahead of execution the bus error may also happen while executing another instruction.

Only reads from the bus can cause bus error exceptions. With that in mind you might think that a store could not cause a bus error because it is after all a write to memory. However stores to cached memory can indirectly cause bus errors. When data is being written to the data cache and there is a cache miss for the cache line that contains the data the cache controller will request a read of the entire cache line from memory. It will then merge the word being stored. This read from memory can be the source of a bus error.

Most of the time this is where things go terribly wrong. Recovery from these errors can be tricky. First you would need to correct the error or try the bus operation again. Unless you have designed your code to check point itself as it goes so you can determine a good restart point it will be hard to do anything but restart the entire process. Most of the time this to will fail in the same place and you will be left with trying to report the problem to the user.

Exception Status ERL

ERL – Error Level or error exception mode

- Set when an error happens as opposed to an interrupt pin exception
  - **Reset, NMI, Cache Error**

- **When set CPU is forced into Kernel mode and**
- **all interrupts (exceptions) are disabled**

MIPS

There is a special error condition for Reset, NMI and Cache errors called Error Level. You can tell if the CPU is in this state by checking the ERL bit in the status register.

+ The ERL bit is bit two of the Status register.

+ When this bit is set the CPU is in Kernel Mode and all interrupts have been disabled.

Exception Status ERL

- **Code Vectors for ERL exceptions**
  - When ERL exceptions occur the CPU will start fetching instructions from the following locations:
    - Reset and NMI 0xBFC0 0000 (boot exception vector)
    - Cache error 0xBFC0 0300 or 0xA000 0100 if BEV is cleared.
    - The return address from these errors is stored in the ErrorEPC register. The ERET instruction will use the return address held in ErrorEPC instead of EPC

MIPS

7

+ the reset and NMI exception vector is usually BF C0 00 00 hex

+ The cache error exception vector is different depending on the Boot Exception Vector bit in the status register. If Boot Exception Vector bit is set, as it is at boot time, then the cache error vector is BF C0 03 00. If it is not set then the vector is A0 00 01 00. The idea here is you may take a cache exception when you are booting from rom or flash but once you have initialized your system you can clear the BEV bit and cache exceptions will go to what can be faster memory. The A0 00 01 00 vector is still of course in a uncached region of memory, kseg1.

The ERET instruction will use the return address held in ErrorEPC instead of EPC

Exception EBase Register (BEV = 0)

- **When Boot Exception Vector bit (BEV) in the status register is clear, the exception base changes to the setting in the EBase register.**

31	30	29 12	11 10	9 0
1	0	Exception Base	00	CPU #

- Bit 30 and 31 are always set forcing vectors to a KSEG0 address except for cache error which is forced to a KSEG1 address (non cached).
- Bits 12 – 29 are program settable which allows you to differ the vectors in a multi processor system. This defaults to 0. (0x8000 0000)
- Bits 0 – 9 are hardwired by static input pins that can represent unique values for CPUs in a multi processor system. This can be used to identify a processor in a multiprocessor system.

MIPS

8

On power up, Reset or NMI the processors come up in exception mode and fetch the first instruction at the Boot Exception Vector. The address for the boot exception vector is determined by a configuration option when the core is built and usually falls in the range of a boot flash. There is a bit in the status register called BEV (boot exception Vector) it is set by a power on, reset or NMI. Part of the boot process is to relocate the exception vectors so they will be placed in RAM for faster processing. Once the vectors are placed in RAM the boot code can set the Exception Base Register to the relocated address. The code then clears the BEV bit in the Status Register so that the processor will use the address in the Exception Base Register from that point on.

The EBase register also contains a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

+ Bit thirty and 31 one are hardwired to force all vectors into the cacheable Kseg0 address space with the exception of a cache error exception which will be forced into the un-cached Kseg1 address space.

+ Bits 12– 29 work in conjunction with bits thirty one and thirty to specify the base address of the exception vectors.

Bits eleven and ten are reserved bits and must be written as zeros. They will always return zeros when read.

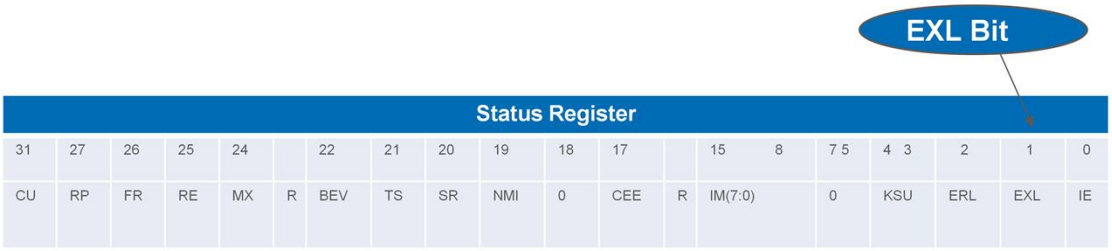
+ Bits zero through nine are usually set when the processor is implemented with a unique value to distinguish one processor from another. Your code can use this to determine which processor it is dealing with.

Bits zero through eleven do not figure into the exception base address.

Exception Status EXL

EXL – Exception Level (or regular exception mode)

- Set by the processor when any exception other than Reset, NMI or Cache Error, the most common being an interrupt



- When set, CPU is in kernel mode and all Hardware and Software interrupts are disabled.

MIPS 9

+ Exception Level bit in the Status Register is set by the processor when any exception other than Reset, NMI or Cache Error exception are taken.

+ The EXL bit set means the CPU is in a normal exception mode set to process an exception that can be returned from using the Exception Program Counter register or EPC for short. In this mode the Exception Program Counter contains the Program Counter that the CPU was processing when the exception happened. Also there is a bit call the Branch Delay bit located in the cause register that will be set if the CPU was executing the instruction in a branch or jump delay slot. The exception code should used this bit to determine what instruction might have caused the exception. If the Branch Delay bit is set either the instruction at Exception Program Counter or the instruction in the Branch delay slot could have caused the exception.

+ All interrupt exceptions both software and hardware will be disabled by the CPU before execution of the exception routine. Exception other than interrupts can still occur.

Note: The Exception PC, Cause register's Branch Delay bit and Shadow Register set register's Control bit will not be updated if another exception is taken.

Exception Vectors 32 bit mode

non MPU

Exception	BEV = 1 (Bootstrap) Address	BEV = 0 (Normal mode) Address
Boot Exception, Reset, NMI	0xBFC0_0000	
TLB Miss (EXL = 0)	0xBFC0_0200	0x8000_0000
Interrupt (Cause(IV) = 1	0xBFC0_0400	0x8000_0200
Cache Error	0xBFC0_0300	0xA000_0100
General Exception	0xBFC0_0380	0x8000_0180
Debug (ECR[ProbTrap] = 0)	0xBFC0_0480	
Debug (ECR[ProbTrap] = 1)	0xFF20_0200	

BEV - Boot Exception Vector
ECR - EJTAG Control Register

MIPS

10

Let take a moment to review what I have talked about so far by using is a table of the exception vectors.

+ The first vector is commonly called the boot exception vector because the CPU at power up will always start fetching it's first instruction at this vector. This vector is also used for Reset, Soft-reset and Non-maskable interrupts. All of these will cause the system to reboot. As a reminder the vector BF C0 00 00 hex is in the non-cacheable Kseg1 address space which is directly mapped to physical address 1f C0 00 00 hex.

The next two sections of the table depend on the boot exception vector bit in the status register.

+ The boot exception vector bit is set on a cold boot.

+ Usually your system will have a boot rom or flash rams that occupies some amount of memory that is located at the boot exception vector physical address of 1F C0 00 00, remember that's virtual address BF C0 00 00. The range of this rom or flesh will run at least through the address of the exception vectors shown here and the area need for the boot code. The boot code starts at the boot exception vector and initializes the cache and the tlb and sets up exception routines in the cacheable Kseg0 address space. There will be more on Boot code in the Boot section of this course.

+ Once the boot code has done enough to be able to use Kseg0 the boot exception vector bit is cleared

+ aside from cache exceptions any exceptions that come after this will go to their cacheable address in Kseg0. The cache vector changes but remains in the uncached Kseg1 address space.

+ last, are the special vectors for use when the CPU is implemented with a EJTAG tap controller.

The address segments in red can be altered by the use of the Exception Base register, Ebase.

General Exception Vector

- **0x0180 General exception vector**
  - For general exceptions the CPU begins fetching instructions from the same location
  - The Cause Register is used to determine which exceptions to service

Cause															
31	30	29 28	27	26		23	22		15 10		9 8	7	6 2		1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP[7:2]		IP1..IP0	0	Exc Code		0

MIPS

11

I will cover interrupt exceptions in detail right after I talk briefly about General exceptions that use the general exception vector.

General exceptions are any exceptions that are not external interrupts, TLB refills, cache, NMI or resets. All general exceptions go to the general exception vector.

To determine which general exception has happened the code must examine the Exception Code field of the coprocessor zero Cause register.

General Exception Vector

- **Machine Check**
  - The detection of multiple matching entries in the TLB.
- **Watch Exception**
  - Instruction or data reference matches the address information programmed into (stored) in the *WatchHi* and *WatchLo* registers
- **Address Error Exception**
  - Unaligned load, store or instruction fetch
  - Reference to the kernel address space from user mode
- **Bus Error**
  - An instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error
- **System Call**
  - SYSCALL instruction is executed

MIPS

12

Lets talk about some of the more common general exceptions

+ A machine check exception happens when duplicate entries have been written to TLB. If this occurs the exception routine needs to read all the TLB entries and over write or invalidate the duplicate entries.

+ A Watch Exception happens due to an instruction or data access that matches the address specified in the watch registers. Watch exceptions are not taken if the CPU is already in exception mode instead, these watch events are remembered, and result in a Deferred Watch exception, which taken as soon as the CPU leaves exception mode.

+ A Address error exception is taken when a load or store tries to access data that is not aligned to the data type of the instruction. For example the destination address of a Store Half Word instruction must be aligned to a halfword boundary. An instruction fetch from a non thirty two bit aligned address for normal instructions or a sixteen bit address for MIPS16 instructions will cause this exception. This can happen if the EPC or EEPC register were overwritten with an unaligned address. Any reference to kernel address space while the CPU is in user mode will also cause this exception.

+ Bus error is generated by an external device when there is a problem in completing a memory request.

+ A system call instruction is coded into you programs by the compiler when your program references a function the needs an OS service. For more information on the system call instruction refer to the MIPS Instruction set section of this course.

General Exception Vector

- **Reserved Instruction**
  - A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed
- **Coprocessor unusable**
  - Coprocessor unit that has not been marked usable or trying to use a CP0 instruction when not in kernel mode.
- **CorExtend block Unusable**
  - Attempt was made to execute a CorExtend instruction when the CEE bit in the Status register is not set.
- **Floating Point Exception**
  - Exception is initiated by the floating point coprocessor
- **Integer Overflow**
  - Selected integer instructions result in a 2's complement overflow

MIPS

13

+ A reserve instruction exception will happen if there is an instruction that the CPU cannot execute. This can be by design, in the case of Floating point instruction that are intended to be handled by software or it can be due to a corrupted code section.

+ Co processor Unusable exception happens when the CPU tries to execute a coprocessor instruction for a coprocessor that is not enabled or when it's in user mode and tries to execute a CoProcessor zero instruction.

+ Core Extent Unusable exception happens when the Core Extend Enable bit is not set. Core extend instructions are user definable instruction that are incorporated into the CPU design the Core Extend Enable bit offers programmatic control over the use of these instructions. For example in a virtual processor core such as the 34K these instruction can be limited to one of the thread processing units so other units would cause this exception if the tried to execute a core extend instruction.

+ Floating Point exceptions are any exception raised by the floating point unit. You will need to examine the status of the floating point unit to determine the cause.

+ The integer overflow instruction happens on selected instructions when the result over flows the size of an integer.

General Exception Vector

- **Trap**
  - A trap instruction resulted in a TRUE value
- **TLB Modified Exception on Data Access**
  - The dirty bit was zero in the TLB entry mapping the address referenced by a store instruction
- **TLB refill**
  - In the case of TLB refill EXL being set means that the CPU was already in the process of processing another exception (most likely a second TLB exception)
- **Breakpoint**
  - BREAK instruction is executed
- **Miscellaneous**
  - There are additional exceptions related to debug that are not covered here. Consult the Core specific Software Users Manual for more information

MIPS

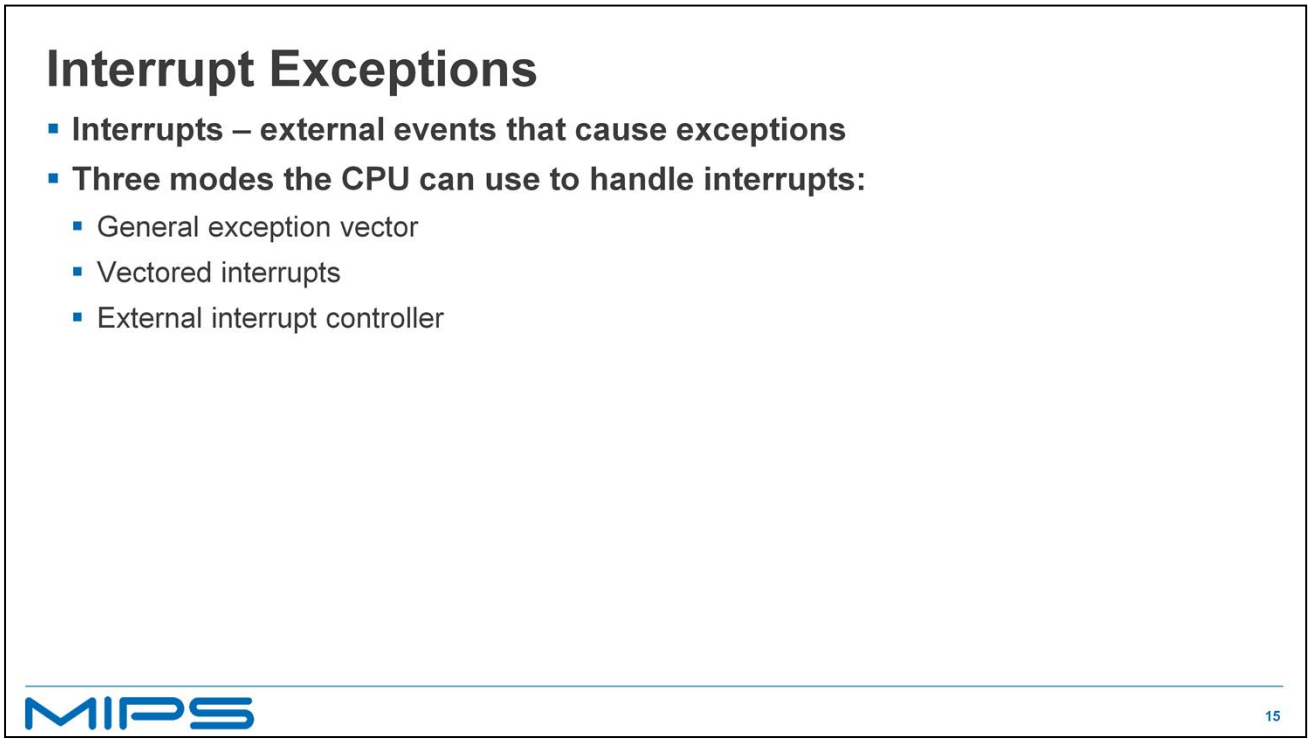
+ The trap instruction is raised when a trap instruction results in a true condition. The MIPS instruction set section has more information on the trap instructions.

+ TLB Modified instruction is caused by a TBL valid entry that is referenced with its dirty bit cleared. The TLB and the use of this bit is discussed in the TLB section of this class.

+ In the case of a TLB refill if EXL is set it means that the CPU was already in the process of processing another exception. The TLB Refill vector is normally 0x8000 0000. this commonly happens while already processing a TLB exception. For example when a page table is needed who's address is not already translated by an entry in the TLB.

+ A breakpoint exception is caused by the execution of a Break instruction. The MIPS instruction set section has more information on the break instructions.

+ For addition exception condition s consult the Software Users Manual of your Core.



I'll now go into detail on Interrupts.

+ Interrupt Exceptions are generated by external events.

+ There are three ways the CPU can handle these events.

+ First all interrupts can use the general exception vector. In this case the exception routine would check the exception code in the cause register and see that it was an interrupt exception. Then the code that would check the status of each device to see which one cause the interrupt. This is commonly called interrupt compatibility mode because it is the only way interrupts could be handled on very early MIPS processors and is still available so code written for them will still work on more current processors.

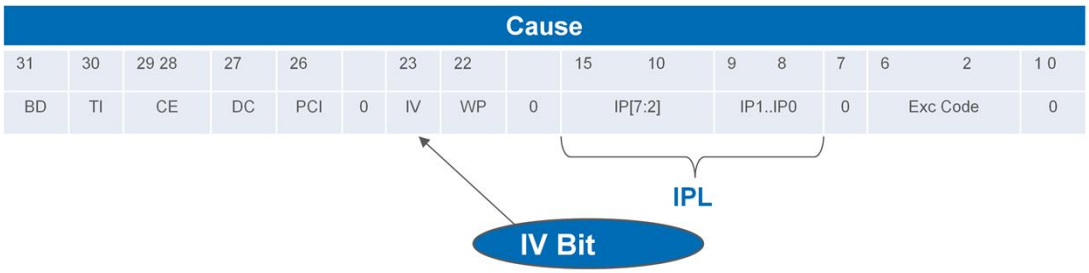
+ The second way the CPU could handle interrupts is the use of interrupt vectors The CPU has 6 external pins that can be wired to devices so they can signal the CPU when there is an interrupt. Each pins is associated with a program vector. When the CPU detects which pin is causing the interrupt it starts execution at the vector that pin is associated with. Sometimes there are more interrupt sources than 6 so some devices may

be wire to the same pin and when an interrupt comes in the code needs to query the devices it knows are connected to the same pin. The highest pin number is given the highest priority.

+ And Third is the external interrupt controller mode or EIC for short. An external interrupt controller can supply the CPU with a value from 0 to 63. Zero indicates there are no interrupts pending and 1 to 63 to indicate the vector of the interrupt to service so in this mode there can be 63 different interrupt vectors.

Enabling Interrupts

- **Using Vectored interrupts**
  - Vectored interrupts are enabled by setting the IV bit in the CP0 Cause register
  - There are 6 hardware interrupt pins and 2 internal software interrupt sources.



MIPS

16

Interrupts that use the general interrupt vector are pretty easy and handled like any other exceptions. Using the one general interrupt vector for interrupts is not used much because it offers the poorest performance in interrupt handling. It is really only there for compatibility with older code. I don't advise its use for new projects.

+ I'll now go into how vector interrupt mode is used.

+ To enable vectored interrupt mode the boot code needs to set the Interrupt Vector bit 23 in the cause register.

+ While we are here I would like to point out the Interrupt priority bits. These bits will give you status on what interrupts are active. You can check these bits at any time for pending interrupts although you don't need to do so for normal operation.

The interrupt pending bits are made up of 6 hardware pins and 2 virtual software pins. The software interrupts are tied to virtual interrupt pins 0 and 1 and can be made the lowest priority interrupts. The idea behind them is you can use these software interrupts to complete interrupt

processing that doesn’t need to be done at a higher priority level. You can be processing a interrupt say a level 5. Your code takes care of all the critical things but has some additional clean up to do that doesn’t need level 5 priority. You can raise a software interrupt that will trigger a software interrupt after all pending and enabled hardware interrupts have completed. You can do this by setting the virtual software pins, bits 8 or 9 in the cause register.

Enabling Individual Interrupts

- **Each interrupt vector is enabled by setting its bit in the Interrupt Mask field, IM of the CP0 Status register.**

+ You can use the Interrupt mask bits in the status register to control which individual interrupt pins will be enabled by the processor. When a interrupt pin is raised by an external device it will cause the cpu to start the interrupt routine only if the corresponding bit in the Interrupt mask field is set.

Interrupt Vector Addresses

- **Using Vectored interrupts (continued)**
  - Vectors start at address BEV + 0x400 / EBase + 0x200
  - Spacing between vectors is set in the CP0 Interrupt Control Register using the VS bits (example later in these slides)

Interrupt Control Register

31 29	28 26		9 5	
IPTI	IPPCI	0	VS	0

Encoding	Spacing between vectors	
Hex	Hex	Decimal
00	000	0
01	020	32
02	040	64
04	080	128
08	100	256
10	200	512

MIPS

18

The interrupt vectors start at address BF C0 04 00 during the boot process when the Boot Exception Vector bit is set and 80 00 02 00 during normal operation when the Boot Exception Vector bit is cleared. In both cases this address is after any other exception vectors.

+ Where the code for the next vector starts is programmable. You can select how much room there is between vectors. Usually there is just enough room for you to setup the interrupt stack, save registers as needed and call a C function to do the rest of the interrupt processing but you can have up to 512 bytes for code in between vectors. Note there is only one setting that applies to all interrupt vectors.

+ The setting of the vector spacing is done by using the Vector Spacing bits in the CP zero Interrupt control register. The boot code should set up this spacing before interrupts are enabled. You will see an example of this later in this section.

Interrupt Instructions

- **DI atomically clears the Status Register Interrupt Enable bit, returning the old value of Status Register in a general purpose register**
- **EI atomically sets the Status Register Interrupt Enable bit, returning the old value of Status Register Interrupt Enable in a general purpose register.**
  - Note of caution: It is better to restore the old value of Status Register Interrupt Enable returned by di, so your “disable interrupts” code will not malfunction if you accidentally invoke it when interrupts were already disabled.

MIPS

19

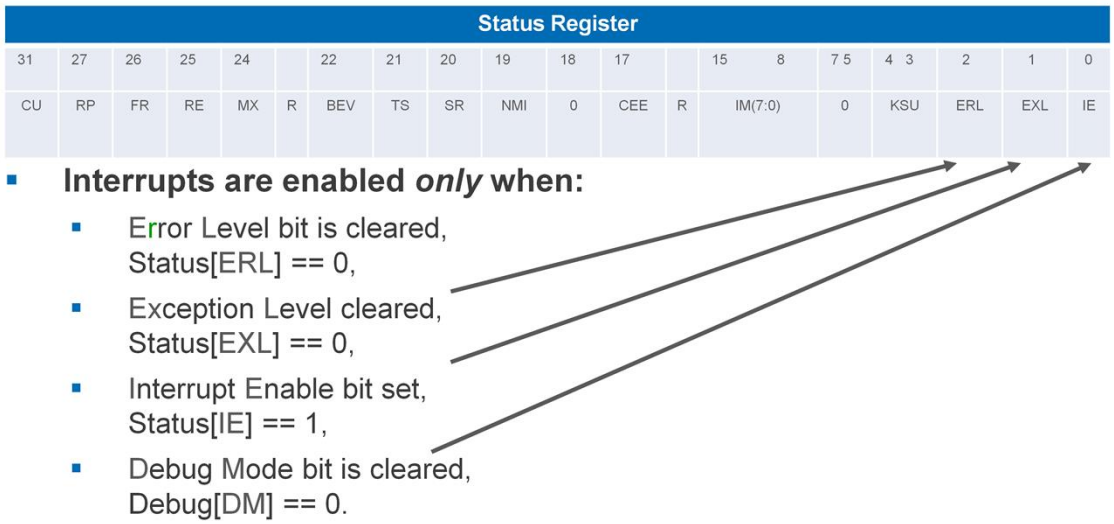
If you have viewed the MIPS instruction section of this course you know there are two instructions that make interrupt control easier. I think it’s a good idea to go over them here again.

+ First is the Disable interrupt instruction. This instruction helps by atomically saving the current value of the Status register to a general purpose register and clearing the Interrupt Enable bit in the CP0 Status register.

+ The Enable Interrupt instruction atomically saves the value of the status register to a general purpose register and sets the interrupt enable bit in the status register.

+ Note you might not always what to use the Enable interrupt instruction particularly if you are nesting interrupts. You might just what to restore the Status register with the value it had when it entered your interrupt function, so the Interrupt Enable bit in the status register will remain as it was and the CPU will continue with processing any lower priority interrupt function that may have interrupted.

Enabling All Interrupts



The boot code and interrupt handling code will need to enable interrupts.

+ To do so the code needs to make sure the following bits are set correctly:

+ Error Level bit is cleared

+ Exception Level Bit is cleared

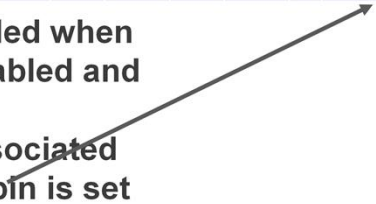
+ Interrupt Enable bit is set

+ and the DebugMode bit is cleared in the CP zero Debug Register

Enabling Individual Interrupts

Status Register																		
31	27	26	25	24		22	21	20	19	18	17		15 8	7 5	4 3	2	1	0
CU	RP	FR	RE	MX	R	BEV	TS	SR	NMI	0	CEE	R	IM(7:0)	0	KSU	ERL	EXL	IE

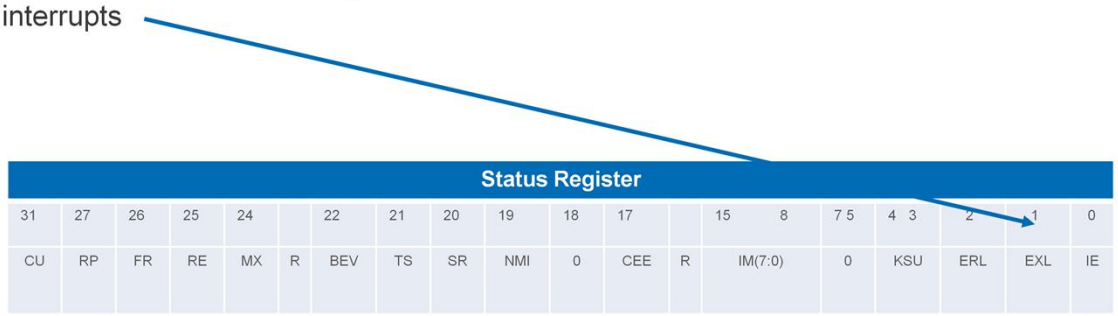
- An Interrupt is enabled when all Interrupts are enabled and the Interrupt Mask, Status[IM7-2] bit associated with the interrupt's pin is set



Next the code must enable individual interrupt vectors by setting the corresponding bit in the interrupt mask field of the CP zero status register.

CPU Interrupt Preparation

- **Automatic steps the CPU takes when it gets an interrupt exception**
  - The restart address is placed in the EPC register to the point to restart execution after the interrupt is processed
  - EXL is set in the Status Register and the CPU enters Kernel mode and disables all interrupts



Status Register																			
31	27	26	25	24		22	21	20	19	18	17		15 8	7 5	4 3	2	1	0
CU	RP	FR	RE	MX	R	BEV	TS	SR	NMI	0	CEE	R	IM(7:0)	0	KSU	ERL	EXL	IE

MIPS

22

Before I go into a specific code example I want to give you an overview to make it clear about what happens through the interrupt process.

+ first I'll start with what the CPU does.

+ Once an interrupt pin is raised and the pin is not masked off in the status register, the CPU sets the exception program counter to the restart address where normal processing should restart.

+ It sets the EXL bit in the Status Register. This has the effect of switching the processor to Kernel Mode and disabling all interrupts.

CPU Interrupt Preparation

Cause															
31	30	29 28	27	26		23	22		15 10	9	8	7	6 2	1	0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP[7:2]	IP1..IP0		0	Exc Code		0

- **Cause Register is updated to give information about pending interrupt interrupts and exception code**
- **The first instruction is fetched from exception vector location**

MIPS

+ The Co Processor Zero Cause Register is set to indicate the interrupt state. For example, the Interrupt Pending bit in the Cause register is set for this interrupt source and the exception code is set to indicate the cause of the exception.

If this were an exception you would have to consult the Exception Code field in the Cause Register to determine what exception happened.

If this were a vectored interrupt you would already know the cause was an interrupt so you won't have to check the exception code.

Also if the cpu is not using vectored interrupts the vector interrupt bit in the Cause Register is cleared there is no enforcement of interrupt priority. Your code can dictate which interrupt can be serviced first by check the Interrupt Priority field of the Cause register and selecting a pin to service. It would be best to stick to the priority scheme of the highest pin being the highest priority but you don't have to in this mode.

+ The CPU then fetches the first instruction at the associated interrupt

vector.

Interrupt Routine

- GPRs K0 and K1 registers are free to use without saving
- Save the old interrupt mask bits in the Status Register to the stack
- Change the mask bits to ensure that the current interrupt and all interrupts your software regards as being of equal or lesser priority are inhibited

MIPS

24

Before I go into the actual code example I'll give you an overview of what your exception code should do

+ When exceptions are disabled you can use the General purpose registers K0 and K1 without saving them.

+ Save the mask bits in the Status register, The easiest way would be to save the whole status register.

+ Change the mask bits to mask out any interrupts of the same or lower priority then the one being serviced.

Interrupt Routine continued

- **Save the state for calling a C function.**
- **Change the CPU state to that appropriate for nested interrupts and exceptions.**
  - Set the Global Interrupt Enable bit in the status register to allow higher priority interrupts to be processed.
  - Change the CPU privilege level field to keep the CPU in kernel mode as you clear exception level,
  - Clear SR(EXL) itself to leave exception mode and expose the changes made in the status register.
- **Call your interrupt routine.**

MIPS

25

+ If the rest of your interrupt routine is in C you must first make room on the stack save all of the general purpose registers and then save them. If you are using a shadow register set for this vector you won't have to save the General purpose registers but you will need to copy the stack pointer and global pointer to the shadow set.

+ If you are going to allow for nested interrupts you need to change the CPU State:

+ Enable interrupts by setting the Interrupt enable bit in the Status Register

+ Set the kernel mode by clearing the User Mode and Supervisor mod bits in the Status Register that way when you leave exception level you will still be in Kernel mode to process the rest of your interrupt routine.

+ Leave exception mode by clearing the Exception Level bit , EXL in the Status register.

+ you can now jump to you c function and process

Interrupt Routine

- **On return Disable interrupts**
- **Clear interrupt source**
- **Restore state and prepare to return**
  - restore GPRs
- **Return from exception**
  - Jump to EPC and then change back from Kernel privilege level (if necessary) must be done atomically.
  - The instruction eret, (exception return) does this for you. It clears the SR(EXL) bit and jumps to the address stored in EPC

MIPS

+ On return you'll need to disable interrupts again so you can restore the pre-interrupt values of registers and resume execution of the interrupted task. You can do this by restores the Status Register with the copy that you saved off in the beginning of the interrupt routine.

+ Make sure that the interrupt has been acknowledge and clear on the device that caused the interrupt.

+ Restore CPU state so the interrupted code can continue unchanged.

+ Restore the General Purpose registers. On a side note since you saved all the General Purpose registers this step will restore the stack pointer and Global pointers of the interrupted code so you shouldn't use stack and global variables that are deference using these registers after this step in your interrupt code.

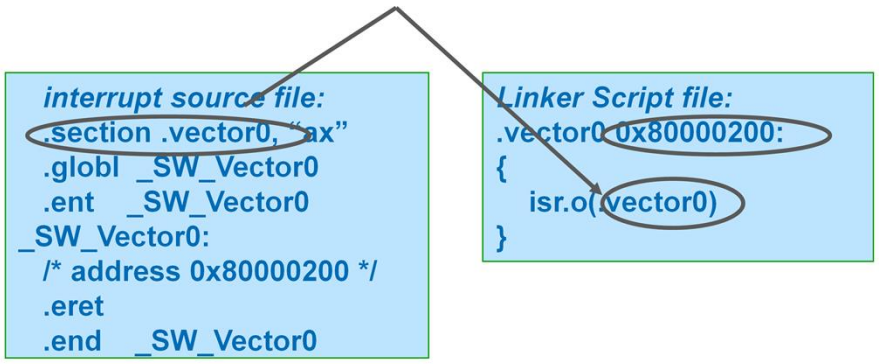
+ Now you are ready to return from the interrupt exception

+ The code needs to jump to the address in the Exception Program counter and enable exceptions.

+ You should use the ERET instruction to do this because, it will atomically clear the EXL bit in the Status register and Jump the address stored in the Exception Program Counter.

Placing interrupt code at vector address

- **Linker script**
  - .section directives for each vector are used by the linker file to place the section in the correct memory location.



MIPS

27

Link.ld Uses the

+ .section directives in the isr.s source file to

+ Link the code for each interrupt routines

+ To their vector address.

Order Setting in Code - reminder

- **.set noreorder**
  - The order of instructions is important. The noreorder setting tells the compiler not to optimize this section of code, so the programmer has control over what goes into the branch delay slots

MIPS

28

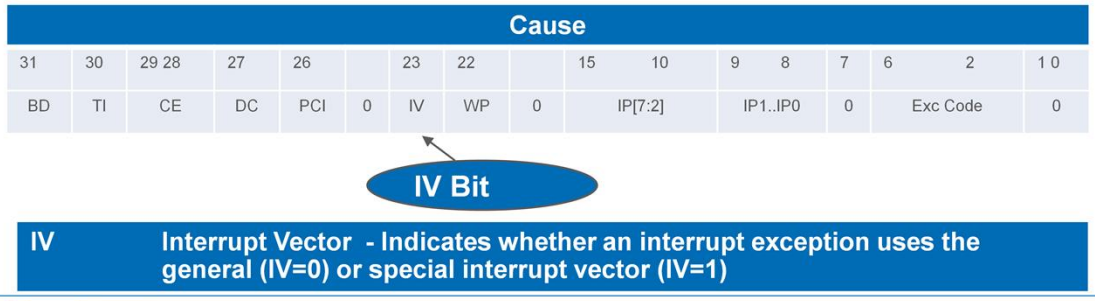
I will now cover the start.s file that initializes the interrupts in detail.

One of the first things you will see in the start.s file is dot set no reorder. This is done because I want to control what goes into the branch delay slots of the Assemble code.

Setting Vectored Mode

- **To configure the CPU for Vectored Interrupt mode set the IV bit, bit 23 in the Cause register.**

```
mfc0  t0, C0_CAUSE  # Get Cause register
li    t1, 1
ins   t0, t1, 23, 1     # set bit 23
mtc0  t0, C0_CAUSE  # Write it back to enable interrupts
ehb                 # Wait for change to take effect
```



The next piece of code will enable vector interrupt mode.

If vectored interrupt mode is not enabled all interrupts will go to the general interrupt vector.

+ To enable vectored interrupt mode I need to set bit 23 of the cause register.

+ First I read the cause register into the t0 register using the move from Coprocessor zero instruction

+ then the next 2 instructions will set bit 23 in the t0 register by first moving a 1 to the t1 register and then using the insert instruction, inserting the first bit of register t1 into bit 23 of the t0 register

+ Now that the bit is set I will writ the value back to the cause register using the move to coprocessor zero instruction

+ I'll use the ehb instruction to clear any hazard barrier that could exist

with the write to the cause register.

Set Interrupt Code Spacing

```
mfc0    t0, C0_INTCTL       # Get Interrupt Control register
li t1, 0x200                # Set Space between vectors to 512
ins t0, t1, 0, 10           # insert spacing
mtc0 t0, C0_INTCTL          # move the GPR to the CP0 register
ehb
```

Interrupt Control Register

31 29	28 26		9 5	
IPTI	IPPCI	0	VS	0

Encoding	Spacing between vectors	
Hex	Hex	Decimal
00	000	0
01	020	32
02	040	64
04	080	128
08	100	256
10	200	512

MIPS

30

You can set the spacing between the interrupt vectors to suit your needs. From no space which is usually not desirable because all you could do is fall into the next vector to 512 bytes where you might place a complete interrupt routine.

I am going to set it for the max of 512 bytes.

+ First I read the interrupt control register into the t0 register using the move from Coprocessor zero instruction

+ load the value to be inserted

+ Notice how the register is laid out so you can set the lower bits to the actual spacing you want so all I have to do is insert the spacing value into the t0 register.

+ then I just move the to register back to the interrupt control register using the move to coprocessor zero instruction.

Interrupt Routine - Stack Setup

- **Interrupt Routine function**
  - This is a vectored interrupt function example that uses the system's GPR set
- **Setup stack for saving registers and calling a C function**

```
addiu   k1, sp, -180  #Add 180 bytes to the stack
/* align to word boundary */
ori     k1, k1, 0x7
xori    k1, k1, 0x7
```

MIPS

31

Now for the interrupt functions themselves. I am going to show you an interrupt function that will call a C function to do most of the work. In this example there is not much to do so the C function is very small but is will serve as an example for how you would go about calling a c function from a interrupt routine.

+ The first one I will go through is the non shadow set version.

+ as already covered in the assemble section of this training course, registers K0 and K1 are always free for an interrupt routine to use as long as interrupts are disabled. I will use them in this interrupt routine to adjust the stack pointer. This function will continue to use the current stack to store registers and call a C function. To do this I need to allocate room on the stack for the registers I want to save.

+ I use the stack pointer stored in the general purpose register sp and since stacks grow downward I will subtract the amount of stack space I need from the sp register.

+ I also need to make sure the pointer is aligned to a word boundary.

Interrupt Routine - Saving GPRs

- **Need to save all GPRs except k0 and k1 which we can use freely for interrupt handling (not used by C functions)**
  - NOTE: allow space (16 bytes) for C arguments in compliance with the ABI.

```
sw    at, 20(k1)
sw    v0, 24(k1)
sw    v1, 28(k1)
sw    a0, 32(k1)
sw    a1, 36(k1)
sw    a2, 40(k1) …….. Save rest
```

MIPS

Now that the stack is setup I can start saving context.

+ One note there are 16 bytes that need to be available for C argument storage by C functions so I will leave the first 16 bytes of the stack free so the base register offset value for saving values to the stack will start at 20.

+ will use the store word instruction using the K1 register that has the adjusted stack pointer as a base register and increment a offset value as I save the registers.

Interrupt Routine - Save Critical CP0 Registers

```
mfc0   t1, C0_STATUS      # Get status register
sw     t1, 136(k1)        # Save status register value to stack
mfc0   t2, C0_EPC         # Get EPC register`
sw     t2, 140(k1)        # Save EPC register value to stack
```

- **Reset the interrupt source so interrupt will not be raised agin once interrupts are enabled**

MIPS

33

I want to show you what you need to do to nest interrupts to do this there are some coprocessor 0 register that need to be saved.

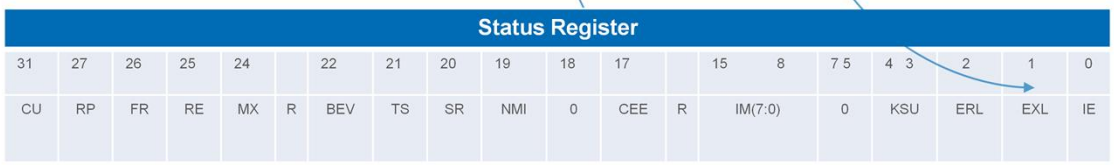
+ Here I will read the values of the status register and the Error Program Counter.

+ And save them to the stack.

Before allowing nested interrupts reset the interrupt source so this interrupt won't be raised again when interrupts are enabled

Interrupt Routine - Enable Interrupts

- **Clear EXL to allow nested interrupts**
  - ins t1, zero, 1, 1
  - mtc0 t1, C0_STATUS
- **NOTE: You should also mask particular interrupts before doing this according to your interrupt priorities.**



Status Register																		
31	27	26	25	24		22	21	20	19	18	17		15 8	7 5	4 3	2	1	0
CU	RP	FR	RE	MX	R	BEV	TS	SR	NMI	0	CEE	R	IM(7:0)	0	KSU	ERL	EXL	IE

MIPS

To enable interrupts I must clear the exl bit of the status register. Register t1 holds the value of the status register so I don't need to get it again and since I have already saved the value to the stack I can make changes to the t1 register. I use the insert instruction to insert a 0 into bit 1 of the register which is the EXL bit and the move it back to the status register.

+ one thing to note you should also change the interrupt mask to mask out interrupts of equal to and less than the priority level you are servicing.

Interrupt Routine – Adjust stack pointer

- **Set stack pointer and jump to C function with argument**

```
move    sp, k1          # Copy new stack pointer to sp
jal C_function          # jump to c function
li a0, 2                # put argument into a0 (BD slot)
```


Last, I copy the stack pointer I have been using that is in the K1 register to the sp register and put an argument into the first argument register. The argument will be used in the C function to increment a counter in the global array.

Interrupt Routine - Post Processing

- **Restore status and EPC registers from stack**

```
move   k1, sp                  # use k1 for stack pointer
lw     k0, 136(k1)             # retrieve the value of C0_STATUS
mtc0   k0, C0_STATUS           # restore C0_STATUS
ehb

lw     t2, 140(k1)             # retrieve the value of C0_EPC
mtc0   t2, C0_EPC              # Restore EPC register
ehb                            # Wait for change to take effect
```

MIPS

After the code returns from the called C function I need to restore the state of the processor so it can continue processing what ever it was doing before being interrupted.

+ I copy the stack pointer to K1 and use K1 as a base register to load values from the stack.

+ I'll restore the status register, Note this also has the effect of disabling interrupts since EXL should be set when this value was saved. Interrupts need to be disabled while we restore the rest of the registers.

+ I'll restore the Error PC which has the address where processing will continue after the error return

Interrupt Routine - Post Processing continued

- **Restore GPRs**

```
lw    at, 20(k1)
lw    v0, 24(k1)
lw    v1, 28(k1)
lw    a0, 32(k1)
lw    a1, 36(k1)
lw    a2, 40(k1)
lw    a3, 44(k1)
lw    t0, 48(k1)
lw    t1, 52(k1)
lw    t2, 56(k1)  ……. Restore all
```

MIPS

Next restore all the GPR registers

Interrupt Routine - Post Processing continued

- **Return from interrupt exception**
  - eret will clear EXL in status register

```
eret                    # Exception Return
```


And last use the error return instruction to atomically clear the EXL bit in the status register and returning execution to the point before the interrupt.

Shadow Register Sets

- **Additional General Purpose Register set Assigned to interrupt vector (only for vectored mode)**
  - Interrupt does not use processes General Purpose Registers
  - Cuts overhead of saving the processes GPRs to the stack

This core can have additional General purpose register set that can be used by interrupts instead of using the normal register set, GPR set 0The next slide will show how a register set is assigned to a particular interrupt vector.

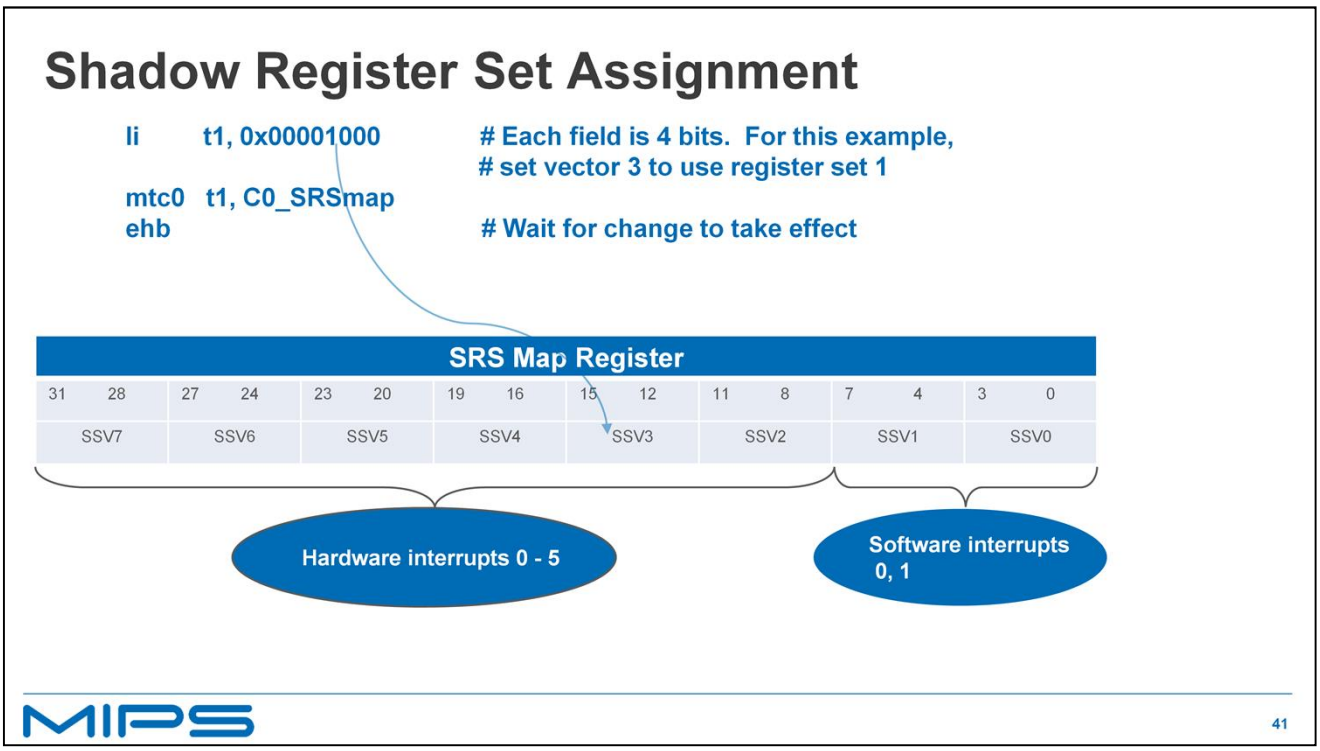
Shadow Register Sets

- **Configured into the core at core build time**
  - Up to 3 shadow register set

MIPS

40

Shadow register set are configured into the core at core build time. These do take up more space in the core so they do add to the cost of the Chip.



Each interrupt vector can be assigned a specific register set. The default is the normal register set, GPR set 0.

In the next examples or a interrupt routine, I will be using a shadow register set so I need to configure which register set will be used by that interrupt vector. I want hardware interrupt 1 to use shadow register set 1. Hardware interrupt 1 goes to interrupt vector 3 since the first two interrupt vectors, zero and one are for software interrupts.

Each field in the Shadow register set map register is 4 bits, Each contains the number of the register set to use with a zero indicating the general propose register set and a one through fifteen indicating a shadow register set. Note most Cores only allow 4 shadow register set so some of the values will be illegal. You can check how many shadow register sets a core has by reading the HSS field in the Shadow Register Set Control register.

In this example only one vector will be using a shadow register set so I can just write the number of the shadow set to the correct field and set the rest to zero.

+ I do this by setting bit 12 of register t1 which effectively writes a 1 to

SSV3 so vector 3 will use shadow register set 1 and the rest will use the general purpose registers.

+ Then I move this value to the Shadow Register Set Map register using the Move to Coprocessor zero instruction

Interrupt Routine using a Shadow Register

- **This is a vectored interrupt function that uses a shadow register set.**
  - This makes the code smaller and faster
- **Setup stack for arguments and CP0 register saves; NOTE: much less stack space**

```
addiu   sp, sp, -28          # push stack for registers +args
```

MIPS

Now I'll go through the code for hardware vector one. Hardware vector 1 is set up with a shadow register set.

Using the shadow register set is less overhead and makes it quicker to start servicing the interrupt device due to the fact that you don't need to save all the general purpose registers.

I'll go through quick the steps involved because they are much the same as I have shown you for hardware vector zero just a few less.

+ first you need to create a stack frame to save some of the registers and argument area before calling a C function

Interrupt Routine using a Shadow Register continued

- **Save critical CP0 registers**

```
mfc0   t1, C0_STATUS      # Get the value of the status register
sw     t1, 20(sp)         # Store status value on the stack
mfc0   t2, C0_EPC         # Get the value of EPC
sw     t2, 24(sp)         # Store the value of EPC on the stack
```


You need to save the status and error Program counter to the stack the same as before but you don't need to save any other registers.

Interrupt Routine using a Shadow Register continue

- **Reset the interrupt source so interrupt will not be raised again once interrupts are enabled**
- **Mask this interrupt and enable nested interrupts**

```
ins   t1, zero, 1, 1          # clear EXL (bit 1)
ins   t1, zero, 11, 1         # clear (mask off) HW Vector 1 (bit 11)
mtc0  t1, C0_STATUS
ehb
```

- **Jump to C function with argument**

```
jal     C_function    # Jump to C function
li      a0, 3         # put argument into a0 (BD Slot)
```

MIPS

Next you need to reset the interrupt source so interrupt will not be raised again once interrupts are enabled

+ then you are ready to enable interrupts. Here I show the masking of the interrupt vector of the interrupt we are servicing

+ once that is done you can jump to the C function that will handle the rest of the interrupt processing

Interrupt Routine - Post Processing using a Shadow Register

- **Reset EPC and Status registers**

```
lw      t2, 24(sp)          # retrieve the saved value of C0_EPC
mtc0    t2, C0_EPC          # Write it back to C0_EPC
ehb                         # Wait for change to take effect

lw      t1, 20(sp)          # retrieve the value of C0_STATUS
mtc0    t1, C0_STATUS       # Write the value to C0_STATUS
ehb                         # Wait for change to take effect

addiu   sp, sp, 28          # set the stack pointer back to were
                            # it was when interrupt routine was entered
eret                        # Exception Return
```

MIPS

45

After the interrupt device has bee serviced

+restore the Error Program Counter and Status register

+ reset the stack pointer and return from the interrupt.

External Interrupt Mode

- The vector spacing is set up the same as it is in vectored interrupt mode.
- All other rules apply the same as they are in vectored interrupt mode
- For system equipped with the MIPS GIC there will be more information on programming the GIC in a later section of the class.

+Vector spacing works the same as in vectored interrupt mode

+ and the interrupt routines and calling of C functions work the same

This section covers Exceptions.