

18-447 Lecture 13: Exceptions and Interrupts

James C. Hoe
Dept of ECE, CMU
March 1, 2010

Announcements: Spring break is almost here
Check grades on Blackboard
Midterm 1 graded

Handouts: Handout #9: Lab 3 (on Blackboard)

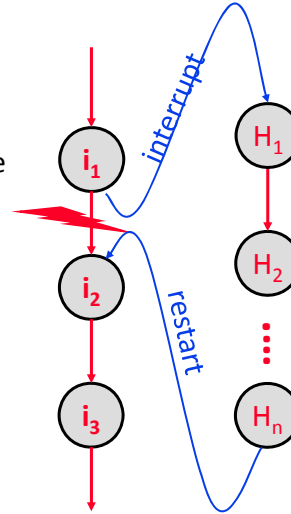
Exceptions and Interrupts

- ◆ A systematic way to handle exceptional conditions that are relatively rare, but must be detected and acted upon quickly
 - instructions may fail and cannot complete
 - external I/O devices may need servicing
 - quantum expiration in a time-shared system
- ◆ Option 1: write every program with continuous checks (a.k.a. polling) for every possible contingency
 - acceptable for simple embedded systems (toaster)
- ◆ Option 2: write “normal” programs for the best-case scenario where nothing unusual happens
 - detect exceptional conditions in HW
 - “transparently” transfer control to an exception handler that knows how to resolve the condition and then back to your program

Interrupt Control Transfer

- ◆ An interrupt is an “unplanned” function call to a system routine (aka, the interrupt handler)
- ◆ Unlike a normal function call, the interrupted thread cannot anticipate the control transfer or prepare for it in any way
- ◆ Control is later returned to the main thread at the interrupted instruction

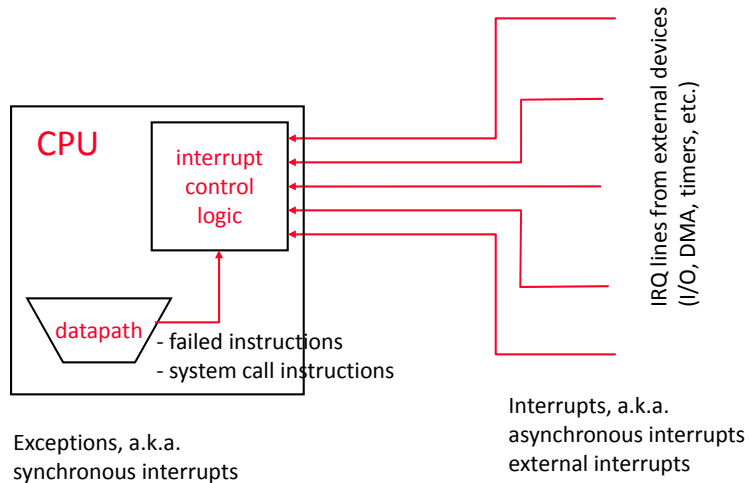
The control transfer to the interrupt handler and back must be 100% transparent to the interrupted thread!!!



Types of Interrupts

- ◆ Synchronous Interrupts (a.k.a. exceptions)
 - exceptional conditions tied to a particular instruction
 - e.g. illegal opcode, illegal operand, virtual memory management faults
 - the faulting instruction cannot be finished
 - no forward progress unless handled immediately
- ◆ Asynchronous Interrupts (a.k.a. interrupts)
 - external events not tied to a particular instruction
 - I/O events, timer events
 - some flexibility on when to handle it
 - cannot postpone forever or things start to “fall on the floor”
- ◆ System Call/Trap Instruction
 - an instruction whose only purpose is to raise an exception
 - whatever for?

Interrupt Sources



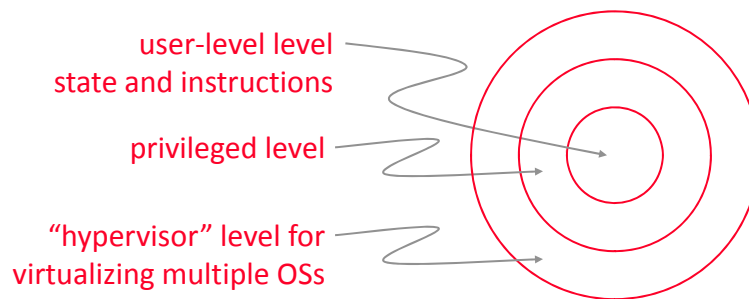
Virtualization and Protection

- ◆ Modern OS supports time-shared multiprocessing but each “user-level” process still thinks it is alone
 - each process sees a private set of user-level architectural states that can be modified by the user-level instruction set
 - each process cannot see or manipulate (directly) state and devices outside of this abstraction
- ◆ OS implements and manages a critical set of functionality
 - keep low-level details out of the user-level process
 - protect the user-level process from each other and itself

Do you want to access/manage harddisk directly? Do you trust your buddy or yourself to access the harddisk directly?

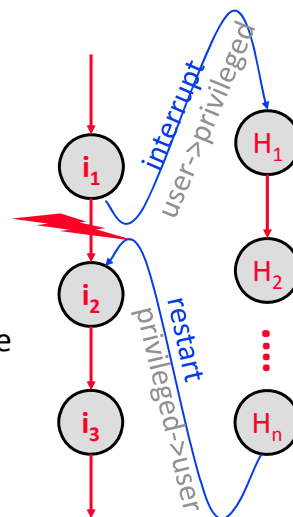
Privilege Levels

- ◆ The OS must somehow be more powerful to create and maintain such an abstraction, hence a separate privileged (aka protected or kernel) mode
 - additional architectural states and instructions, in particular those controlling virtualization/ protection/isolation
 - the kernel code running in the privileged mode has access to the complete “bare” hardware system



Control and Privilege Transfer

- ◆ User-level code never runs in the privileged mode
- ◆ Processor enters the privileged mode only on interrupts---user code surrenders control to a handler in the OS kernel
- ◆ The handler restores privilege level back to user mode before returning control to the user code

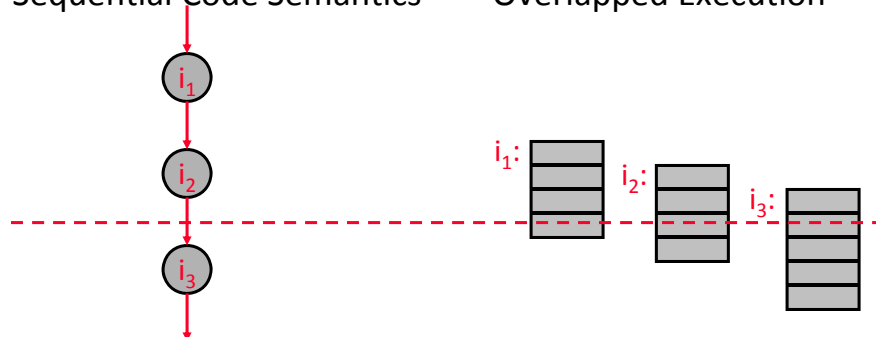


Implementing Interrupts

Precise Interrupt/Exception

Sequential Code Semantics

Overlapped Execution



A precise interrupt appears (to the interrupt handler) to take place exactly between two instructions

- older instructions finished completely
- younger instructions as if never happened
- on synchronous interrupts, execution stops just before the faulting instruction

Stopping and Restarting a Pipeline

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	bub	bub	I_h	I_{h+1}	I_{h+2}	I_{h+3}
ID		I_0	I_1	I_2	I_3	bub	bub	bub	I_h	I_{h+1}	I_{h+2}
EX			I_0	I_1	I_2	bub	bub	bub	bub	I_h	I_{h+1}
MEM				I_0	I_1	I_2	bub	bub	bub	bub	I_h
WB					I_0	I_1	I_2	bub	bub	bub	bub

What if I_0, I_1, I_2, I_3 and I_4 all generate exceptions in t_4 ?
How would things look different for asynchronous interrupts?

Exception Sources in Different Stages

- ◆ IF
 - instruction memory address/protection fault
- ◆ ID
 - illegal opcode
 - trap to SW emulation of unimplemented instructions
 - system call instruction (a SW requested exception)
- ◆ EX
 - invalid results: overflow, divide by zero, etc
- ◆ MEM
 - data memory address/protection fault
- ◆ WB
 - nothing can stop an instruction now...
- ◆ We can associate async interrupts (I/O) with any instruction/stage we like

Pipeline Flush for Exceptions

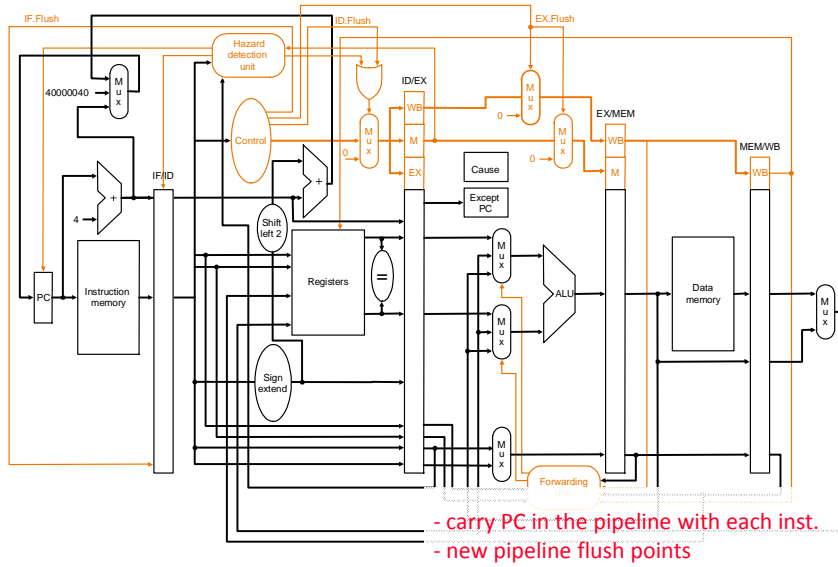
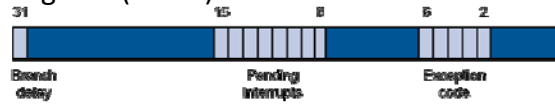


Figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

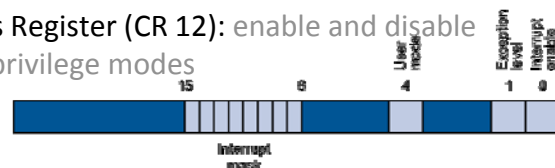
MIPS Interrupt Architecture

MIPS Interrupt Architecture

- ◆ Privileged system control registers
 - Exception Program Counter (EPC, CR14): which instruction did we stop on
 - Interrupt Cause Register (CR 13): what caused the interrupt



- Interrupt Status Register (CR 12): enable and disable interrupts, set privilege modes



- ◆ Loaded automatically on interrupt transfer events
- ◆ Also accessed by the “move from/to co-processor-0” instruction: “mfc0 Ry, CRx” and “mtc0 Ry, CRx”

Figures from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

MIPS Interrupt Architecture

- ◆ On an interrupt transfer, the CPU hardware saves the interrupt address to EPC
 - can't just leave frozen in the PC: overwritten immediately
 - can't use r31 as in a function call: need to save user value
- ◆ In general, CPU hardware must save any such information that cannot be saved and restored in software by the interrupt handler (very few such things)
- ◆ For example, the GPR can be managed in SW by the interrupt handler using a callee-saved convention
 - however, r26 and r27 are reserved by convention to be available to the kernel immediately at the start and the end of an interrupt handler

Interrupt Servicing

- ◆ On an interrupt transfer, the CPU hardware records the cause of the interrupt in a privileged registers (Interrupt Cause Register)
- ◆ Option 1: Control is transfer to a pre-fixed default interrupt handler address
 - this initial handler examines the cause and branches to the appropriate handler subroutine to do the work
 - this address is protected from user-level process so one cannot just jump or branch to it
- ◆ Option 2: Vectored Interrupt
 - a bank of privileged registers to hold a separate specialized handler address for each interrupt source
 - On an interrupt, hardware transfer control directly to the appropriate handler to save interrupt overhead

MIPS uses a 7-instruction handler for TLB-miss

Example of Causes

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Handler Examples

- ◆ On asynchronous interrupts, device-specific handlers are invoked to service the I/O devices
- ◆ On exceptions, kernel handlers are invoked to either
 - correct the faulting condition and continue the program (e.g., emulate the missing FP functionality, update virtual memory management), or
 - “signal” back to the user process if a user-level handler function is registered, or
 - kill the process if the exception cannot be corrected
- ◆ “System call” is a special kind of function call from user process to kernel-level service routines

Returning from Interrupt

- ◆ Software restores all architectural state saved at the start of the interrupt routine
- ◆ MIPS32 uses a special jump instruction (ERET) to atomically
 - restore the automatically saved CPU states
 - restore the privilege level
 - jump back to the interrupted address in EPC
- ◆ MIPS R2000 used a pair of instructions


```
jr r26    // jump to a copy of EPC in r26
rfe      // restore from exception mode
          // must be used in the delay slot!!
```

Branch Delay Slot and RFE

- ◆ What if the faulting address is a branch delay slot?
 - simply jumping back to the faulting address won't continue correctly if the preceding branch was taken
 - we didn't save enough information to do the right thing
- ◆ MIPS's solution
 - the CPU HW makes a note (in the Cause register) if the faulting address captured is in a delay slot
 - in these cases, the handler returns to the preceding branch instruction which gets executed twice (as the last instruction before and first instruction after)
- ◆ Generally harmless except "JALR r31"
 - explicitly disallowed by the MIPS ISA
 - think about what would happen in that case

An Extremely Short Handler

`_handler_shortest:`

`# no prologue needed`

```
... short handler body ... # can use only r26 and r27
                          # interrupt not re-enabled for
                          # something really quick
```

`# epilogue`

`mfc0 r26,epc`

`jr 26`

`rfe`

`# get faulting PC`

`# jump to retry faulting PC`

`# restore from exception mode`

Note: You can find more examples in the book CD. If you are really serious about it, take a look inside Linux source. It is not too hard to figure out once you know what to look for.

A Short Handler

```

_handler_short:
    # prologue
    addi sp, sp, -0x8           # allocate stack space (8 byte)
    sw r8, 0x0(sp)            # back-up r8 and r9 for use in body
    sw r9, 0x4(sp)            #

    . . . short handler body . . . # can use r26, r27, and r8, r9
    . . .                       # interrupt not re-enabled

    # epilogue
    lw r8, 0x0(sp)            # restore r8, r9
    lw r9, 0x4(sp)            #
    addi sp, sp, 0x8          # restore stack pointer
    mfc0 r26, epc             # get EPC
    j r26                      # jump to retry EPC
    rfe                       # restore from exception mode

```

Nesting Interrupts

- ◆ On an interrupt control transfer, further asynchronous interrupts are disabled automatically
 - another interrupt would overwrite the contents of the EPC and Interrupt Cause and Status Registers
 - the handler must be carefully written to not generate synchronous exceptions itself during this window of vulnerability
- ◆ For long-running handlers, interrupt must be re-enabled to not miss additional interrupts
 - the handler must save the contents of EPC/Cause/Status to memory (stack) before re-enabling asynchronous interrupt
 - once interrupts are re-enabled, EPC/Cause/Status is clobbered by the next interrupt (contents appear to change for no reason)

