CS61C
Operating System Support and
Prioritized, Re-Entrant Interrupts

Lecture 13

March 3, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html



## Review 1/2

- I/O gives computers their 5 senses
- I/O speed range is million to one
- Processor speed means must synchronize with I/O devices before use
- Polling works, but expensive
- Interrupts works, more complex
- I/O control leads to Operating Systems



## Review 2/2: 4 Responsibilities leading to OS

- The I/O system is shared by multiple programs using the processor
- Low-level control of I/O device is complex because requires managing a set of concurrent events and because requirements for correct device control are often very detailed
- I/O systems often use interrupts to communicate information about I/O operations
- Would like I/O services for all user programs, under safe control



## Outline

- Instruction Set support for OS
- Administrivia, “What’s this stuff Good for”
- Prioritizing Interrupts
- Re-entrant Interrupt Routine
- Direct Memory Access
- Conclusion

## OS , I/O Communication Requirements

- The OS must be able to prevent:
  - The user program from communicating with the I/O device directly
- If user programs could perform I/O directly:
  - No protection to the shared I/O resources
- 3 types of communication are required:
  - The OS must be able to give commands to the I/O devices
  - The I/O device notify OS when the I/O device has completed an operation or an error
  - Data transferred between memory and I/O device



## Review Coprocessor Registers

- Coprocessor 0 Registers:

| *name* | *number* | *usage* |
|---|---|---|
| BadVAddr | `$8` | Bad memory address |
| Status | `$12` | Interrupt enable |
| Cause | `$13` | Exception type |
| EPC | `$14` | Instruction address |

- Different registers from integer registers, just as Floating Point is another set of registers independent from integer registers
  - Floating Point called "Coprocessor 1", has own set of registers and data transfer instructions



## Instruction Set support for OS

- How turn off interrupts during interrupt routine?
- Bit in Status Register determines whether or not interrupts enabled: <u>Interrupt Enable</u> <u>bit (IE)</u> ($0 \Rightarrow$ off, $1 \Rightarrow$ on)

| (described later) | IE |
|---|---|

Status Register



## Instruction Set support for OS

- How prevent user program from turning off interrupts (forever)?
  - Bit in Status Register determines whether in user mode or OS (kernel) mode: <u>Kernel/User bit (KU)</u> ($0 \Rightarrow$ kernel, $1 \Rightarrow$ user)

| (described later) | KU | IE |
|---|---|---|

Status Register

  - On exception/interrupt disable interrupts (IE=0) and go into kernel mode (UK=0)
- How remember old IE, U/K bits?
  - Hardware copies Current IE and UK bits (0-1) into Previous IE UK bits (2-3)

| (described later) | KU | IE | KU | IE | 0 | 0 |
|---|---|---|---|---|---|---|

Status Register

## How communicate between OS and user?

- OS to user
  - No restrictions on OS; can modify registers or memory visible to user program
  - To restore previous Kernel/User bits after interrupt, use Return from Exception (`rfe`)
- User to OS
  - `syscall` instruction: invoke the kernel (Go to 0x80000080, change to kernel mode)
  - By software convention, `$v0` has system service requested: OS performs request




## SPIM OS Services via Syscall

| Service | Code (put in `$v0`) | Args | Result |
|---|---|---|---|
| print_int | 1 | `$a0` = integer | |
| print_float | 2 | `$f12` = float | |
| print_double | 3 | `$f12` = double | |
| print_string | 4 | `$a0` = string | |
| read_int | 5 | | integer (in `$v0`) |
| read_float | 6 | | float (in `$f0`) |
| read_double | 7 | | double (in `$f0`) |
| read_string | 8 | `$a0` = buffer, `$a1` = length | |
| sbrk | 9 | `$a0` = amount | address(in `$v0`) |
| exit | 10 | | |

- Note: most OS services deal with I/O




## Example: User invokes OS (SPIM)

- Print “the answer = 5”
- First print “the answer =”:

```
.data
str:
.asciiz "the answer = "
.text
li  $v0, 4   # code for print_str
la  $a0, str # address of string
syscall      # print the string
```

- Now print 5

```
li  $v0, 1   # code for print_int
li  $a0, 5   # integer to print
syscall      # print it
```




## Relationship Memory Mapped I/O & Syscall?

- “Warning: Programs that use these syscalls to read from the terminal should not use memory-mapped I/O.” (Ap. A, p. A-49)
- Why?
- OS is using memory mapped I/O to provide a high level I/O abstraction to user programs; cannot violate abstraction and have it work

## Administrivia

- **Readings: I/O 8.9**
- **3rd Project: Today 7PM (Th. 8AM deadline)**
- **6th homework: Due 3/10 7PM**
  - **Exercises 8.3, 8.29 (skip challenge), Ap A.3**
- **4th Project: Friday 3/12 7PM (thank TAs!) (deadline Saturday 3/13 8AM)**
- **Upcoming events**
  - **Midterm Review Sunday 3/14 2PM, 1 Pimentel**
  - **Midterm on Wed. 3/17 5pm-8PM, 1 Pimentel**
  - **2nd online questionnaire when in lab 3/16-17**



## "What's This Stuff Good For?"

**Blind since birth, five-year-old Amy Stewart learns to read by computer, keeping up with the sighted kids in her first-grade class. The computer converts written lessons into Braille printouts.**

**Computers can't yet replace the human eye, but advanced technology is enabling the blind to join a world previously inaccessible to them. Kent Cullers, a prominent astrophysicist who is also blind, observes, "I interact as many other people do nowadays, through their machines. And the wonder of the technology is that I can do the job just as well as many other people can."** *One Digital Day*, 1998 (www.intel.com/onedigitalday)



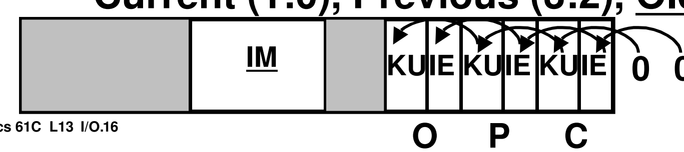
## Basic Interrupt Routine

- **Save several registers and `$ra` in memory for use in interrupt routine**
- **Get EPC and select exception code field from Cause Register**
- **Jump and link via jump table to appropriate interrupt routine based on (I/O interrupt, System call, Arithmetic Overflow)**
  - **Single interrupt address ⇒ jump table**
- **Return to code to restore registers, previous IE, UK bits (`rfe`) and return to instruction determined by old EPC**



## Prioritizing Interrupts

- **Some interrupts have higher priority**
  - **Alternative to blocking all interrupts?**
- **Categorize interrupts and exceptions into levels, and allow selective interruption via Interrupt Mask(IM) in Status Register: 5 for HW interrupts**
  - **Interrupt only if IE==1 AND Mask bit == 1 (bits 15:0 of SR) for that interrupt level**
  - **To support interrupts of interrupts, have 3 deep stack in Status for IE,K/U bits: Current (1:0), Previous (3:2), Old (5:4)**




**Status Register**

## Handling Prioritized Interrupts

- OS convention to simplify software:
  - Process cannot be preempted by interrupt at same or lower level
  - Return to interrupted code as soon as no more interrupts at a higher level
  - Any piece of code is always run at same priority level



## Interrupt Levels in MIPS?

- What are they?



- It depends what the MIPS chip is inside of: differ by app Casio PalmPC, Sony Playstation, HP LaserJet printer
- Hardware box designer associates I/O events with pins of MIPS chips according to needs of application
  - MIPS architecture enables priorities



## Interrupt Levels in MIPS Architecture

- Conventionally, from highest level to lowest level exception/interrupt levels:

  1) Bus error

  2) Illegal Instruction/Address trap

  3) High priority I/O Interrupt (fast response)

  4) Low priority I/O Interrupt (slow response)

(later in course will add more levels)

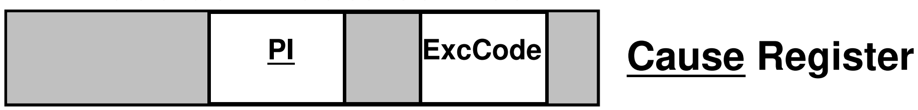


## Interrupt Levels in MIPS Software

- Conventionally, UNIX software system designed to have 4 to 6 Interrupt Priority Levels (IPL) that match the HW interrupt levels
- Processor always executing at one IPL, stored in a memory location and Status Register set accordingly
  - Processor at lowest IPL level, any interrupt accepted
  - Processor at highest IPL level, all interrupt ignored
  - Interrupt handlers and device drivers pick IPL to run at, faster response for some

## Interrupt levels

- Suppose there was an interrupt while the interrupt enable or mask bit is off: what should you do? (cannot ignore)
- Cause register has field--Pending Interrupts (PI)-- 5 bits wide (bits15:10) for each of the 5 HW interrupt levels
  - Bit becomes 1 when an interrupt at its level has occurred but not yet serviced
  - Interrupt routine checks pending interrupts ANDed with interrupt mask to decide what to service





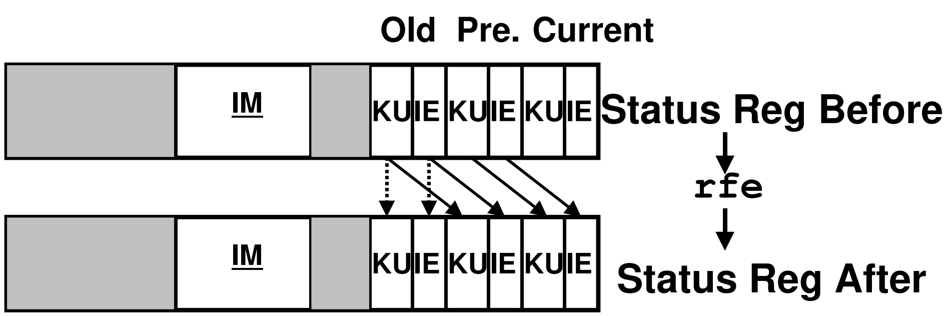

## Revised Interrupt Routine 1/2

- Get EPC and Cause Register
- Save EPC, CR, `$ra` and some registers in memory for use in interrupt routine
- Jump and link via jump table to appropriate exception/interrupt routine
- If I/O, Cause Register IP field ANDed Status Register IM field to find unmasked interrupts (maybe several); pick highest
- Change IM of Status Register to inhibit current level and lower priority interrupts
- Change Current IE of Status Register to enable interrupts (higher priority)



## Revised Interrupt Routine 2/2

- Jump to appropriate interrupt routine
- On Return, disable interrupts using Current IE bit of Status Register
- Then restore saved registers, previous IE/UK bits of Status (via `rfe`) and return to instruction determined by old EPC






## Re-entrant Interrupt Routine?

- How allow interrupt of interrupts and safely save registers?
- Stack?
  - Resources consumed by each exception, so cannot tolerate arbitrary deep nesting of exceptions/interrupts
- With priority level system only interrupted by higher priority interrupt, so cannot be recursive
- ⇒ Only need one save area ("exception frame") per priority level

## Improving Data Transfer Performance

- Thus far: OS give commands to I/O , I/O device notify OS when the I/O device completed operation or an error
- What about data transfer to I/O device?
  - Processor busy doing loads/stores between memory and I/O Data Register
- Ideal: specify the block of memory to be transferred, be notified on completion?
  - Direct Memory Access (DMA) : a simple computer transfers a block of data to/from memory and I/O, interrupting upon done



## Example: Direct Memory Access

- DMA from Disk Device to Memory at `Start`, for 4096 bytes

```
          .data
Count:    .word  4096
Start:    .space 4096
          .text
Initial: lw $s0, Count # No. chars
         la $s1, Start # @next char
Wait:    lw $s2, DiskControl
         andi $s2,$s2,1 # select Ready
         beq $s2,$0,Wait # spinwait
         lb $t0, DiskData # get byte
         sb $t0, 0($s1) # transfer
         addiu $s0,$s0,-1 # Count--
         addiu $s1,$s1,1  # Start++
         bne   $s0,$0,Wait # next char
```

- DMA “computer” in parallel with CPU



## “And in Conclusion..” 1/1

- Operating System started as shared I/O library
  - Support for OS abstraction: Kernel/User bit, stacked KU bits, `syscall`
  - MIPS follows coprocessor abstraction to add resources, instructions for OS
- Interrupt control: Interrupt Enable bit, stacked IE bits, Interrupt Priority Levels, Interrupt Mask
  - Re-entrant via restricting int. to higher priority
- DMA to accelerate data movement
- Next: Anatomy of I/O: disks, networks, ...