inst.eecs.berkeley.edu/~cs61c/su06

CS61C : Machine Structures

Lecture #25: I/O

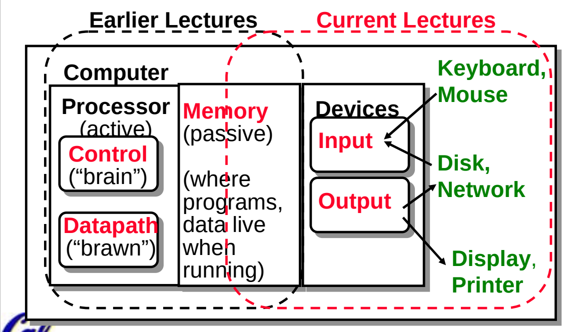
2006-08-10

Andy Carle

Review

- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
- Virtual Memory allows protected sharing of memory between processes with less swapping to disk

Recall : 5 components of any Computer



Motivation for Input/Output

- I/O is how humans interact with computers
- I/O gives computers long-term memory.
- I/O lets computers do amazing things:

  - Read pressure of synthetic hand and control synthetic arm and hand of fireman

  - Control propellers, fins, communicate in BOB (Breathable Observable Bubble)
- Computer without I/O like a car without wheels; great technology, but won't get you anywhere

I/O Device Examples and Speeds

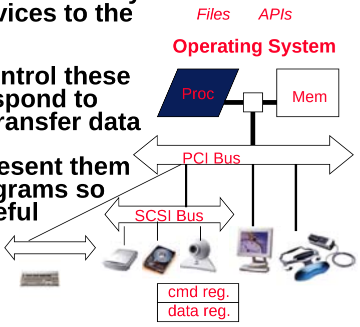
- I/O Speed: bytes transferred per second (from mouse to Gigabit LAN: 10-million-to-1)

• Device	Behavior	Partner	Data Rate (KBytes/s)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice output	Output	Human	5.00
Floppy disk	Storage	Machine	50.00
Laser Printer	Output	Human	100.00
Magnetic Disk	Storage	Machine	10,000.00
Wireless Network	I or O	Machine	10,000.00
Graphics Display	Output	Human	30,000.00
Wired LAN Network	I or O	Machine	125,000.00

When discussing transfer rates, use 10^x

What do we need to make I/O work?

- A way to connect many types of devices to the Proc-Mem
- A way to control these devices, respond to them, and transfer data
- A way to present them to user programs so they are useful

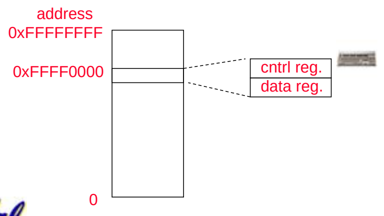


Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: reads a sequence of bytes
  - Output: writes a sequence of bytes
- Some processors have special input and output instructions
- Alternative model (used by MIPS):
  - Use loads for input, stores for output
  - Called "Memory Mapped Input/Output"
  - A portion of the address space dedicated to communication paths to Input or Output devices (no memory there)

Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices



Processor-I/O Speed Mismatch

- 1GHz microprocessor can execute 1 billion load or store instructions per second, or 4,000,000 KB/s data rate
  - I/O devices data rates range from 0.01 KB/s to 125,000 KB/s
- Input: device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act
- Output: device not be ready to accept data as fast as processor stores it
- What to do?

Processor Checks Status before Acting

- Path to device generally has 2 registers:
  - Control Register, says it's OK to read/write (I/O ready) [think of a flagman on a road]
  - Data Register, contains data
- Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg ($0 \Rightarrow 1$) to say its OK
- Processor then loads from (input) or writes to (output) data register
  - Load from or Store into Data Register resets Ready bit ($1 \Rightarrow 0$) of Control Register

SPIM I/O Simulation

- SPIM simulates 1 I/O device: memory-mapped terminal (keyboard + display)
  - Read from keyboard (receiver); 2 device regs
  - Writes to terminal (transmitter); 2 device regs

Receiver Control `0xffff0000`	Unused (00...00)	(I.E.)	Ready
Receiver Data `0xffff0004`	Unused (00...00)	Received Byte	
Transmitter Control `0xffff0008`	Unused (00...00)	(I.E.)	Ready
Transmitter Data `0xffff000c`	Unused	Transmitted Byte	

SPIM I/O

- Control register rightmost bit (0): Ready
  - Receiver: Ready==1 means character in Data Register not yet been read;
    $1 \Rightarrow 0$ when data is read from Data Reg
  - Transmitter: Ready==1 means transmitter is ready to accept a new character;
    $0 \Rightarrow$ Transmitter still busy writing last char
    - I.E. bit discussed later
- Data register rightmost byte has data
  - Receiver: last char from keyboard; rest = 0
  - Transmitter: when write rightmost byte, writes char to display

I/O Example

- Input: Read from keyboard into `$v0`

```
              lui  $t0, 0xffff #ffff0000
Waitloop:     lw   $t1, 0($t0) #control
              andi $t1,$t1,0x1
              beq  $t1,$zero, Waitloop
              lw   $v0, 4($t0) #data
```

- Output: Write to display from `$a0`

```
              lui  $t0, 0xffff #ffff0000
Waitloop:     lw   $t1, 8($t0) #control
              andi $t1,$t1,0x1
              beq  $t1,$zero, Waitloop
              sw   $a0, 12($t0) #data
```

- Processor waiting for I/O called "Polling"
- "Ready" bit from processor's point of view!

Cost of Polling?

- Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling
  - Mouse: polled 30 times/sec so as not to miss user movement
  - Floppy disk: transfers data in 2-Byte units and has a data rate of 50 KB/second. No data transfer can be missed.
  - Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed.

% Processor time to poll [p. 677 in book]

Mouse Polling, Clocks/sec

= 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]

- % Processor for polling:

  $12*10^3$ [clocks/s] / $1*10^9$ [clocks/s] = 0.0012%

  ⇒ Polling mouse little impact on processor

Frequency of Polling Floppy

= 50 [KB/s] / 2 [B/poll] = 25K [polls/s]

- Floppy Polling, Clocks/sec

  = 25K [polls/s] * 400 [clocks/poll] = 10M [clocks/s]

- % Processor for polling:

  $10*10^6$ [clocks/s] / $1*10^9$ [clocks/s] = 1%

  ⇒ OK if not too many I/O devices

% Processor time to poll hard disk

Frequency of Polling Disk

= 16 [MB/s] / 16 [B] = 1M [polls/s]

- Disk Polling, Clocks/sec
  = 1M [polls/s] * 400 [clocks/poll]
  = 400M [clocks/s]
- % Processor for polling:

  $400*10^6$ [clocks/s] / $1*10^9$ [clocks/s] = 40%

  ⇒ Unacceptable

What is the alternative to polling?

- Wasteful to have processor spend most of its time "spin-waiting" for I/O to be ready
- Would like an unplanned procedure call that would be invoked only when I/O device is ready
- Solution: use exception mechanism to help I/O. Interrupt program when I/O ready, return when done with data transfer

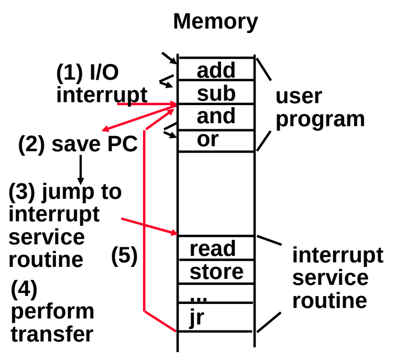
I/O Interrupt

- An I/O interrupt is like overflow exceptions except:
  - An I/O interrupt is "asynchronous"
  - More information needs to be conveyed
- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
  - I/O interrupt does not prevent any instruction from completion

Definitions for Clarification

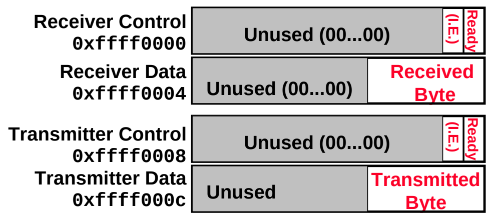
- **Exception: signal marking that something "out of the ordinary" has happened and needs to be handled**
- **Interrupt: asynchronous exception**
- **Trap: synchronous exception**
- **Note: Many systems folks say "interrupt" to mean what we mean when we say "exception".**

Interrupt Driven Data Transfer



SPIM I/O Simulation: Interrupt Driven I/O

- **I.E. stands for Interrupt Enable**
- **Set Interrupt Enable bit to 1 have interrupt occur whenever Ready bit is set**



Benefit of Interrupt-Driven I/O

- **Find the % of processor consumed if the hard disk is only active 5% of the time. Assuming 500 clock cycle overhead for each transfer, including interrupt:**
  - **Disk Interrupts/s = 16 MB/s / 16B/interrupt = 1M interrupts/s**
  - **Disk Interrupts, clocks/s = 1M interrupts/s * 500 clocks/interrupt = 500,000,000 clocks/s**
  - **% Processor for during transfer: $500*10^6 / 1*10^9$ = 50%**
- **Disk active 5% $\Rightarrow$ 5% * 50% $\Rightarrow$ 2.5% busy**

Generalizing Interrupts

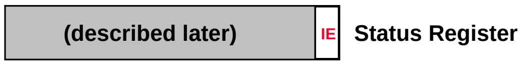
- **We can handle all sorts of exceptions with interrupts.**
- **Big idea: jump to handler that knows what to do with each interrupt, then jump back**
- **Our types: syscall, overflow, mmio ready.**

OS: I/O 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 transfers between memory and I/O device**

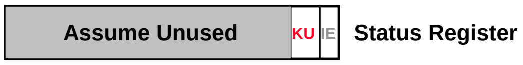
Instruction Set Support for OS (1/2)

- How to turn off interrupts during interrupt routine?
- Bit in Status Register determines whether or not interrupts enabled: **Interrupt Enable bit (IE)** (0 ⇒ off, 1 ⇒ on)



Instruction Set Support for OS (2/2)

- How to prevent user program from turning off interrupts (forever)?
  - Bit in Status Register determines whether in user mode or OS (kernel) mode: **Kernel/User bit (KU)** (0 ⇒ kernel, 1 ⇒ user)



  - On exception/interrupt disable interrupts (IE=0) and go into kernel mode (KU=0)

Administrivia

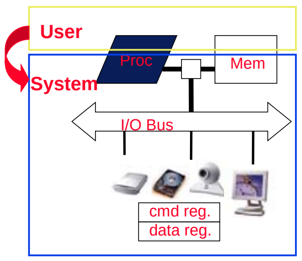
- Project 4 Due Tuesday
- Final Exam will be next Friday!

Kernel/User Mode

- Generally restrict device access to OS
- HOW?
- Add a "**mode bit**" to the machine: K/U
- Only allow SW in "**kernel mode**" to access device registers
- If user programs could access device directly?
  - could destroy each others data, ...
  - might break the devices, …

Crossing the System Boundary

- System loads user program into memory and 'gives' it use of the processor
- Switch back
  - SYSCALL
    - request service
    - I/O
  - TRAP (overflow)
  - Interrupt



Syscall

- How does user invoke the 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 = 42”
- First print “the answer =”:

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

- Now print 42

```
li  $v0,1    # 1=code for print_int
li  $a0,42   # integer to print
syscall      # print int
```

Handling a Single Interrupt (1/3)

- An interrupt has occurred, then what?
  - Automatically, the hardware copies PC into EPC ($14 on cop0) and puts correct code into Cause Reg ($13 on cop0)
  - Automatically, PC is set to 0x80000080, process enters kernel mode, and interrupt handler code begins execution
  - Interrupt Handler code: Checks Cause Register (bits 5 to 2 of $13 in cop0) and jumps to portion of interrupt handler which handles the current exception

Handling a Single Interrupt (2/3)

- Sample Interrupt Handler Code

```
.text 0x80000080
mfc0  $k0,$13   # $13 is Cause Reg
sll   $k0,$k0,26  # isolate
srl   $k0,$k0,28  #   Cause bits
```

- Notes:
  - Don’t need to save `$k0` or `$k1`
    - MIPS software convention to provide temp registers for operating system routines
    - Application software cannot use them
  - Can only work on CPU, not on cop0

Handling a Single Interrupt (3/3)

  - When the interrupt is handled, copy the value from EPC to the PC.
  - Call instruction `rfe` (return from exception), which will return process to user mode and reset state to the way it was before the interrupt
- What about multiple interrupts?

Multiple Interrupts

- Problem: What if we’re handling an Overflow interrupt and an I/O interrupt (printer ready, for example) comes in?
- Options:
  - drop any conflicting interrupts: unrealistic, they may be important
  - simultaneously handle multiple interrupts: unrealistic, may not be able to synchronize them (such as with multiple I/O interrupts)
  - queue them for later handling: sounds good

Prioritized Interrupts (1/3)

- Question: Suppose we're dealing with a computer running a nuclear facility. What if we're handling an Overflow interrupt and a Nuclear Meltdown Imminent interrupt comes in?
- Answer: We need to categorize and prioritize interrupts so we can handle them in order of urgency: emergency vs. luxury.

Prioritized Interrupts (2/3)

- 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
  - When an interrupt is handled, take the highest priority interrupt on the queue
    - may be partially handled, may not, so we may need to save state of interrupts(!)

Prioritized Interrupts (3/3)

- To implement, we need an Exception Stack:
  - portion of address space allocated for stack of "Exception Frames"
  - each frame represents one interrupt: contains priority level as well as enough info to restart handling it if necessary

Modified Interrupt Handler (1/3)

- Problem: When an interrupt comes in, EPC and Cause get overwritten *immediately* by hardware. Lost EPC means loss of user program.
- Solution: Modify interrupt handler. When first interrupt comes in:
  - disable interrupts (in Status Register)
  - save EPC, Cause, Status and Priority Level on Exception Stack
  - re-enable interrupts
  - continue handling current interrupt

Modified Interrupt Handler (2/3)

- When next (or any later) interrupt comes in:
  - interrupt the first one
  - disable interrupts (in Status Register)
  - save EPC, Cause, Status and Priority Level (and maybe more) on Exception Stack
  - determine whether new one preempts old one
    - if no, re-enable interrupts and continue with old one
    - if yes, may have to save state for the old one, then re-enable interrupts, then handle new one

Modified Interrupt Handler (3/3)

- Notes:
  - Disabling interrupts is dangerous
  - So we disable them for as short a time as possible: long enough to save vital info onto Exception Stack
- This new scheme allows us to handle many interrupts effectively.

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
- MIPS architecture enables priorities for different I/O events

Interrupt Levels in MIPS Architecture

- Conventionally, from highest level to lowest level exception/interrupt levels:
  - Bus error
  - Illegal Instruction/Address trap
  - High priority I/O Interrupt (fast response)
  - Low priority I/O Interrupt (slow response)
  - Others

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: code in DMA controller

- DMA code from Disk Device to Memory

```
         .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

Details not covered

- MIPS has a field to record all pending interrupts so that none are lost while interrupts are off; in Cause register
- The Interrupt Priority Level that the CPU is running at is set in memory
- MIPS has a field in that can mask interrupts of different priorities to implement priority levels; in Status register
- MIPS has limited nesting of saving KU,IE bits to recall in case higher priority interrupts; in Status Register

Implementation?

- Take 150 & 152

Peer Instruction

A. A faster CPU will result in faster I/O.

B. Hardware designers handle mouse input with interrupts since it is better than polling in almost all cases.

C. Low-level I/O is actually quite simple, as it's really only reading and writing bytes.

"And in conclusion…"

- I/O gives computers their 5 senses
- I/O speed range is 100-million to one
- Processor speed means must synchronize with I/O devices before use
- Polling works, but expensive
  - processor repeatedly queries devices
- Interrupts works, more complex
  - devices causes an exception, causing OS to run and deal with the device
- I/O control leads to Operating Systems