

## CS61C : Machine Structures

### Lecture 6.2.2 Interrupts

2004-07-29

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c



CS 61C L6.2.2 Interrupts (1)

K. Meinz, Summer 2004 © UCB

## Big Idea

- How to choose between associativity, block size, replacement policy?
- Design against a performance model
  - Minimize: **Average Memory Access Time**
    - = Hit Time
    - + Miss Penalty x Miss Rate
  - influenced by technology & program behavior
  - Note: **Hit Time encompasses Hit Rate!!!**
- Create the illusion of a memory that is large, cheap, and fast - on average



CS 61C L6.2.2 Interrupts (2)

K. Meinz, Summer 2004 © UCB

## Example

- Assume
  - Hit Time = 1 cycle
  - Miss rate = 5%
  - Miss penalty = 20 cycles (on top of hit)
  - Calculate AMAT...
- Avg mem access time
  - =  $1 + 0.05 \times 20$
  - = 1 + 1 cycles
  - = 2 cycles



CS 61C L6.2.2 Interrupts (3)

K. Meinz, Summer 2004 © UCB

## Ways to reduce miss rate

- Larger cache
  - limited by cost and technology
  - hit time of first level cache < cycle time
- More places in the cache to put each block of memory – associativity
  - fully-associative
    - any block any line
  - k-way set associated
    - k places for each block
    - direct map: k=1

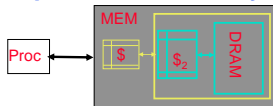


CS 61C L6.2.2 Interrupts (4)

K. Meinz, Summer 2004 © UCB

## Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM
  - ⇒ 200 processor clock cycles!



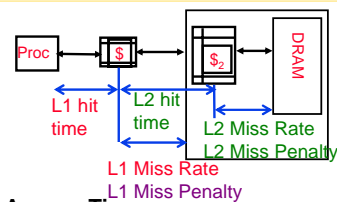
Solution: another cache between memory and the processor cache: **Second Level (L2) Cache**



CS 61C L6.2.2 Interrupts (5)

K. Meinz, Summer 2004 © UCB

## Analyzing Multi-level cache hierarchy



$$\text{Avg Mem Access Time} = \text{L1 Hit Time} + \text{L1 Miss Rate} * \text{L1 Miss Penalty}$$

$$\text{L1 Miss Penalty} = \text{AMAT}_{L2} = \text{L2 Hit Time} + \text{L2 Miss Rate} * \text{L2 Miss Penalty}$$

$$\text{Avg Mem Access Time} = \text{L1 Hit Time} + \text{L1 Miss Rate} * (\text{L2 Hit Time} + \text{L2 Miss Rate} * \text{L2 Miss Penalty})$$



CS 61C L6.2.2 Interrupts (6)

K. Meinz, Summer 2004 © UCB

### Typical Scale

- L1
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%
- L2:
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%
- L2 miss rate is fraction of L1 misses that also miss in L2
- why so high?



### Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = **200 cycles**
- L1 miss penalty =  $5 + 0.15 * 200 = 35$
- Avg mem access time =  $1 + 0.05 * 35 = \underline{2.75 \text{ cycles}}$



### Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
- Avg mem access time =  $1 + 0.05 * 200 = \underline{11 \text{ cycles}}$
- **4x faster with L2 cache!** (**2.75** vs. **11**)



### Cache Summary

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - Write through v. write back?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

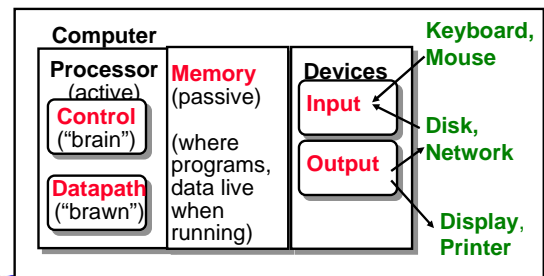


### Outline

- Exceptions
- Memory Mapped IO
- Exception Implementation



### 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



CS 61C L6.2.2 Interrupts (13)

K. Meinz, Summer 2004 © UCB

## I/O Device Examples and Speeds

- I/O Speed: bytes transferred per second (from mouse to Gigabit LAN: 100-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	1,000,000.00

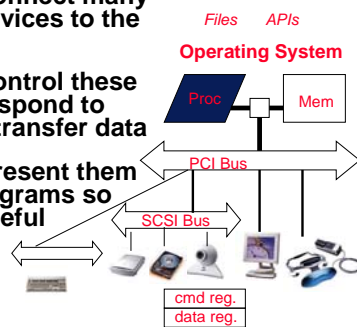


CS 61C L6.2.2 Interrupts (14)

K. Meinz, Summer 2004 © UCB

## 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



CS 61C L6.2.2 Interrupts (15)

K. Meinz, Summer 2004 © UCB

## 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)

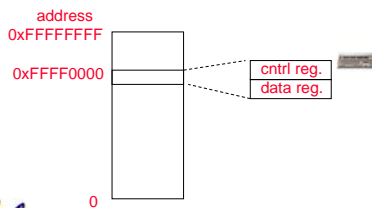


CS 61C L6.2.2 Interrupts (16)

K. Meinz, Summer 2004 © UCB

## Memory Mapped I/O

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



CS 61C L6.2.2 Interrupts (17)

K. Meinz, Summer 2004 © UCB

## 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 1,000,000 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?



CS 61C L6.2.2 Interrupts (18)

K. Meinz, Summer 2004 © UCB

### 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 ⇒ 1) to say its OK
- Processor then loads from (input) or writes to (output) data register
  - Load from or Store into Data Register → Proc resets Ready bit (1 ⇒ 0) of Control Register



CS 61C L6.2.2 Interrupts (19)

K. Meinz, Summer 2004 © UCB

### SPIM/Proj4 I/O Simulation

- Simulate 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)	Ready (1)
Receiver Data 0xffff0004	Unused (00...00)	Received Byte
Transmitter Control 0xffff0008	Unused (00...00)	Ready (1)
Transmitter Data 0xffff000c	Unused	Transmitted Byte



CS 61C L6.2.2 Interrupts (20)

K. Meinz, Summer 2004 © UCB

### SPIM I/O

- Control register rightmost bit (0): Ready
  - Receiver: Ready==1 means character in Data Register not yet been read; 1 ⇒ 0 when data is read from Data Reg
  - Transmitter: Ready==1 means transmitter is ready to accept a new character; 0 ⇒ 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



CS 61C L6.2.2 Interrupts (21)

K. Meinz, Summer 2004 © UCB

### I/O Example

- Input: Read from keyboard into \$v0

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

- Output: Write to display from \$a0

```
Waitloop:    lui    $t0, 0xffff #ffff0000
            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**"



CS 61C L6.2.2 Interrupts (22)

K. Meinz, Summer 2004 © UCB

### 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.



CS 61C L6.2.2 Interrupts (23)

K. Meinz, Summer 2004 © UCB

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

#### Mouse Polling, Clocks/sec

$$= 30 \text{ [polls/s]} * 400 \text{ [clocks/poll]} = 12\text{K} \text{ [clocks/s]}$$

- % Processor for polling:

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

⇒ Polling mouse little impact on processor

#### Frequency of Polling Floppy

$$= 50 \text{ [KB/s]} / 2 \text{ [B/poll]} = 25\text{K} \text{ [polls/s]}$$

- Floppy Polling, Clocks/sec

$$= 25\text{K} \text{ [polls/s]} * 400 \text{ [clocks/poll]} = 10\text{M} \text{ [clocks/s]}$$

- % Processor for polling:

$$10 * 10^6 \text{ [clocks/s]} / 1 * 10^9 \text{ [clocks/s]} = 1\%$$

⇒ OK if not too many I/O devices



CS 61C L6.2.2 Interrupts (24)

K. Meinz, Summer 2004 © UCB

## % Processor time to poll hard disk

### Frequency of Polling Disk

$$= 16 \text{ [MB/s]} / 16 \text{ [B]} = 1\text{M [polls/s]}$$

- Disk Polling, Clocks/sec  
 $= 1\text{M [polls/s]} * 400 \text{ [clocks/poll]}$   
 $= 400\text{M [clocks/s]}$
- % Processor for polling:  
 $400 * 10^6 \text{ [clocks/s]} / 1 * 10^9 \text{ [clocks/s]} = 40\%$   
 $\Rightarrow$  Unacceptable



CS 61C L6.2.2 Interrupts (25)

K. Meitz, Summer 2004 © UCB

## 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



CS 61C L6.2.2 Interrupts (26)

K. Meitz, Summer 2004 © UCB

## 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



CS 61C L6.2.2 Interrupts (27)

K. Meitz, Summer 2004 © UCB

## Definitions for Clarification

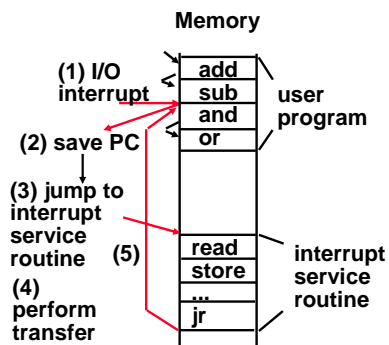
- **Exception**: signal marking that something “out of the ordinary” has happened and needs to be handled
- **Interrupt**: asynchronous exception
- **Trap**: synchronous exception
- **Note**: These are different from the book’s definitions.
  - All I care about: that you know the difference between sync and async.



CS 61C L6.2.2 Interrupts (28)

K. Meitz, Summer 2004 © UCB

## Interrupt Driven Data Transfer



CS 61C L6.2.2 Interrupts (29)

K. Meitz, Summer 2004 © UCB

## 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

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



CS 61C L6.2.2 Interrupts (30)

K. Meitz, Summer 2004 © UCB

## 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:  $500 \cdot 10^6 / 1 \cdot 10^9 = 50\%$
- Disk active 5%  $\Rightarrow$  5% \* 50%  $\Rightarrow$  2.5% busy



CS 61C L6.2.2 Interrupts (31)

K. Meitz, Summer 2004 © UCB

## 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.



CS 61C L6.2.2 Interrupts (32)

K. Meitz, Summer 2004 © UCB

## Generalizing Interrupts

- Must support:
  - Jumping to handler
    - On exception, proc sets nPC  $\leftarrow$  handlerAddr
  - Knowing what happened/bookkeeping
    - EPC: Reg holds -inst at which nrpt occurred
    - Cause: Holds the specific interrupt
  - Jumping back to user program
    - More bookkeeping (back to user mode)
    - ~ "jr \$EPC"



CS 61C L6.2.2 Interrupts (33)

K. Meitz, Summer 2004 © UCB

## Knowing what happened

- On exception, proc copies PC+4 into EPC:

```
assign exception = arith_excp | mem_excp | sys_excp
always @ (posedge clk)
  if (exception) EPC <= PC + 4;
```
- Proc copies exception number into cause:

```
always @ (posedge clk) {
  if (arith_excp) Cause <= 0x1;
  else if (mem_excp) Cause <= 0x2;
  else if (sys_excp) Cause <= 0x4; }
```



CS 61C L6.2.2 Interrupts (34)

K. Meitz, Summer 2004 © UCB

## Jumping to handler

New nPC mux:

```
assign nPC = (exception) ? 0x04000040
              : old_npc;
```

- Overrides all other nPC signals
  - Jump to handler takes priority over branch.



CS 61C L6.2.2 Interrupts (35)

K. Meitz, Summer 2004 © UCB

## In handler:

- Handler uses special regs/instrs to access exception data:
  - \$k0/\$k1 (Why not \$t0 ... \$t9?)
  - mfc0 \$reg \$EPC
  - mfc0 \$reg \$CAUSE
- When done, move EPC (or whatever) in \$k0 and jr \$k0.



CS 61C L6.2.2 Interrupts (36)

K. Meitz, Summer 2004 © UCB

## Proc Support for mfc0:

### • mfc0 \$reg X

Lots of options:

#### • In decode as “add \$reg X \$0”:

- Mux in front of BusA <- ForwardA, Cause, EPC
- Mux in front of BusB <- ForwardB, 0

#### • In wb:

- Mux in front of regdst:  
Regdst <- ALUout, MemOut, Cause, EPC



©Which one do you think pipelines better?

CS 61C L6.2.2 Interrupts (37)

K. Meina, Summer 2004 © UCB

## What about pipelining?!

• Precise Exceptions ⇒ State of the machine is preserved as if program executed up to (not including) the offending instruction

- All previous instructions **completed**
- Offending instruction and all following instructions act **as if they have not even started**
- Same system code will work on different implementations



CS 61C L6.2.2 Interrupts (38)

K. Meina, Summer 2004 © UCB

## Exception/Interrupts: Implementation

5 instructions, executing in 5 different pipeline stages!

### • Who caused the interrupt?

**Stage**    **Problem interrupts occurring**

IF Page fault on instruction fetch; misaligned memory access; memory-protection violation

ID Undefined or illegal opcode; syscall

EX Arithmetic exception

MEM Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

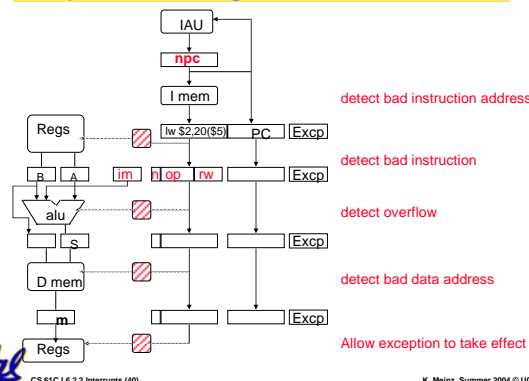
- How do we stop the pipeline? How do we restart it?
- Do we interrupt immediately or wait?
- **How do we sort all of this out to maintain preciseness?**



CS 61C L6.2.2 Interrupts (39)

K. Meina, Summer 2004 © UCB

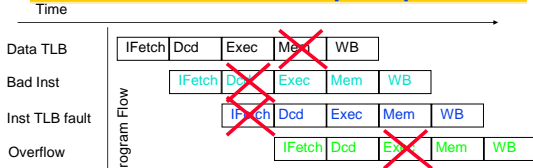
## Exception Handling



CS 61C L6.2.2 Interrupts (40)

K. Meina, Summer 2004 © UCB

## Another look at the exception problem



### • Use pipeline to sort this out!

- Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reaches WB stage
- When instruction reaches WB stage:
- Save PC ⇒ EPC, exc\_status ⇒ cause, handler addr ⇒ PC
  - Turn all instructions in earlier stages into noops!



CS 61C L6.2.2 Interrupts (41)

K. Meina, Summer 2004 © UCB

## Detailed implementation

### • New pipeline regs:

- Valid <- If instr. Is not valid, no writes to regfile or mem will occur. Invalids will not trigger exceptions
- iPC <- addr of the current instr. Will go into EPC
- Exception <- type of exception. Will go into cause.

• In each stage: if excepting, set except reg. **“flag exception”**

• In MEM, WB, only do write if Valid

• In WB: **“raise exception”** ...



CS 61C L6.2.2 Interrupts (42)

K. Meina, Summer 2004 © UCB

## Detailed implementation

---

- In WB:

```
if (ME/WB.Valid & ME/WB.Except) {  
  IF/DE.valid_in = DE/Ex.Valid_in =  
  EX/ME.Valid_in = ME/WB.valid_in = 0;  
  npc ← ME/WB.iPC;  
  cause ← Me/WB.Except;  
  EPC ← Me/WB.iPC;
```

