CS61C - Machine Structures

Lecture 26 - Review of Interrupts

December 6, 2000

David Patterson

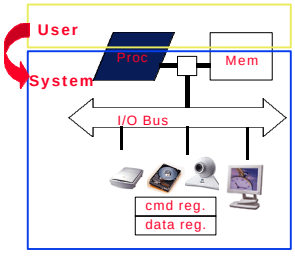
http://www-inst.eecs.berkeley.edu/~cs61c/

Outline

- Instruction Set Support for Interrupts
- Prioritized Interrupts
- Re-entrant Interrupt Routine
- Start course review (if time permits)

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



Reasons for Exceptions/Interrupts

- Hardware errors: memory parity error
- External I/O Event
  - High Priority and Low Priority I/O events
- Illegal instruction
- Virtual memory
  - TLB miss
  - Write protection violation
  - Page fault - page is on disk
  - Invalid Address - outside of address range
- Arithmetic overflow
  - Floating Point Exceptions
- System call (invoke Op Sys routine)

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

Software/Hardware Resources for Exceptions

- Registers to use in interrupt routine
  - $k1, $k2 reserved for use; don't have to be saved
- Enable/Disable Interrupt Bit
- Kernel/User mode to protect when can Disable Interrupt
- Register showing cause of interrupt
- PC address of interrupted instruction
- Register with virtual address if it caused an interrupt

Review of Coprocessor 0 Registers

° Coprocessor 0 Registers:

name	*number*	*usage*
BadVAddr	$8	Addr of bad instr
Status	$12	Interrupt enable
Cause	$13	Exception type
EPC	$14	Instruction address

° Different registers from integer registers, just as Floating Point has another set of registers independent from integer registers

- Floating Point called “Coprocessor 1”, has own set of registers and data transfer instructions

Nested Interrupt Support

° If going to support nested interrupts, what must be saved/restored on entry/exit of nested interrupt?

- Save/restore all things associated with current interrupt: Exception PC, BadVaddr, Cause, Interrupt Enable, Kernel/User state
- Any registers use beyond $k0, $k1

° What’s hard to save/restore?

- Interrupt Enable, Kernel/User, want to restore at same time, so put back into same state at same time: no chance Enable Interrupts before set Kernel/User properly
- Add instruction to restore both

Nested Interrupt Support

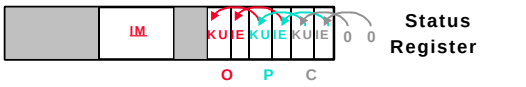
° How many levels deep must hardware support for IE, KU?

- Either max number of nested interrupts
- or just 2, since SW could save/restore once interrupted

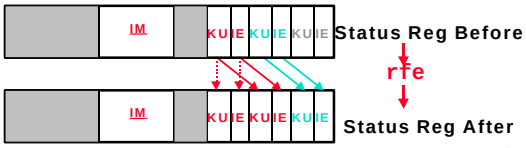
° MIPS Hardware saves restores user program, interupt routine, nested interrupt routine

Prioritizing Interrupts

° 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)



° Then restore previous KU,IE bits of Status (via rfe)



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

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

Administrivia: Rest of 61C

•Fri	12/8	Last Lecture: Stanford v. Cal
Sun	12/10	Final Review, 2PM (155 Dwinelle)
Tues	12/12	Final (5PM 1 Pimintel)
Mon	12/11	Beta test of Final

Contact Sumeet (cs61c-tf) to find place, time

- See TA ASAP about grade disagreements: scores should include lab6 as of today
- Final: Just bring pencils: leave home back packs, cell phones, calculators
- 2 sheets of paper, both sides, #2 pencils (no calculators)
  - Will check that notes are handwritten

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.

Supporting Multiple Interrupts in Software

- Exception/Interrupt behavior determined by combination of hardware mechanisms and operating system strategies
- same hardware with different OS acts differently
- A popular software model for multiple interrupts/exceptions, often used in Unix OS, is to set priority levels
  - This is an OS concept, not a HW concept
  - HW just needs mechanisms to support it

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)

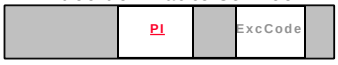
- Notes:
  - Disabling interrupts is dangerous
  - So we disable them for as short a time as possible: long enough to save vital info onto Exception Stack
- This new scheme allows us to handle many interrupts effectively.

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 (since it's a Software Concept v. HW)
- MIPS has a field in that can mask interrupts of different priorities to implement priority levels; in Status register

Interrupts while serving interrupts?

- 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 (bits15:11) 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



Cause Register

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

From First Lecture

- 15 weeks to learn big ideas in CS&E
  - Principle of abstraction, used to build systems as layers
  - Compilation v. interpretation to move down layers of system
  - Pliable Data: a program determines what it is
  - Stored program concept: instructions are data
  - Principle of Locality, exploited via a memory hierarchy (cache)
  - Greater performance by exploiting parallelism
  - Principles/pitfalls of performance measurement

Stored program concept: instructions as data

- Allows computers to switch personalities
- Simplifies compile, assembly, link, load
- Distributing programs easy: on any disk, just like data
  - ⇒ binary compatibility, upwards compatibility (8086, 80286, 80386, 80486, Pentium I, II, III, 4)
- Makes it easier for viruses: Send message that overflows stack, starts executing code in stack area, take over machine

Principle of Locality

- Exploited by memory hierarchy
- Registers assume Temporal Locality: data in registers will be reused
- Disk seeks faster in practice: short seeks are much faster, so disk accesses take less time ⇒ due to Spatial Locality
- Disks transfer in 512 Byte blocks assuming spatial locality: more than just 4 bytes useful to program
- Networks: most traffic is local, so local area network vs. wide area network

Cache (1/2)

- Memory hierarchy
- Spatial locality vs. temporal locality
- N-way set associative
  - N=1: direct mapped
  - N=# cache blocks: fully associative
- Block size tradeoff
- Average access time
  - Hit time, hit rate, miss penalty, miss rate

Cache (2/2)

- Cache misses
  - Compulsory, conflict, capacity
- Replacement policy
  - LRU, random
- Multi-level caches
- Write-through vs. write-back

Greater performance by exploiting parallelism

- Pipelining
  - Overlap execution to increase instruction throughput vs. instruction latency
- Input/Output
  - Overlap program execution with I/O, only interrupt when I/O complete
  - DMA data while processor does other work
- RAID (Redundant Array of Inexp. Disks)
  - Replace a few number of large disks with a large number of small disks ⇒ more arms moving, more heads transferring (even though small disks maybe slower)

Pipeline (1/2)

- Remember laundry analogy!
- Latency vs. throughput
- Structural hazards
  - exist in register file and memory
  - fix memory by adding another level 1 $
  - fix register by specifying first half cycle writes and second half cycle reads

Performance measurement Principles/Pitfalls

- Processors
  - only quoting one factor of 3-part product: clock rate but not CPI, instruction count
  - Benchmarks v. Clock rate of Pentium 4
  - Cache miss rate vs. Average memory time
- Networks
  - only looking peak bandwidth, not including software start-up overhead for message
- Disks
  - Seek time much better than quoted (3X)
  - Data transfer rate worse than quoted (0.75X)

Rapid Change AND Little Change

- Continued Rapid Improvement in Computing
  - 2X every 1.5 years (10X/5yrs, 1000X/15yrs)
  - Processor speed, Memory size - Moore's Law as enabler (2X transistors/chip/1.5 yrs); Disk capacity too (not Moore's Law)
  - Caches, Pipelining, Branch Prediction, ...
- 5 classic components of all computers
  1. Control
  2. Datapath
  3. Memory
  4. Input
  5. Output

  } Processor (or CPU)

Things to Remember

- Kernel Mode v. User Mode: OS can provide security and fairness
- Syscall: provides a way for a programmer to avoid having to know details of each I/O device
- To be acceptable, interrupt handler must:
  - service all interrupts (no drops)
  - service by priority
  - make all users believe that no interrupt has occurred