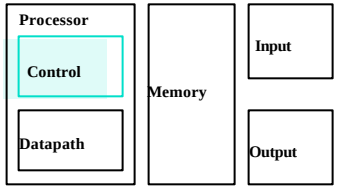
CS 152
Computer Architecture and Engineering
Lecture 12

Multicycle Controller Design
Exceptions

The Big Picture: Where are We Now?

° The Five Classic Components of a Computer



° Today's Topics:

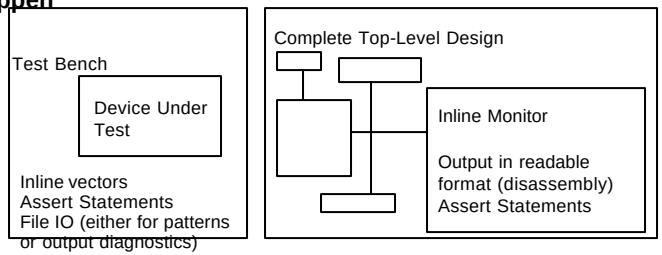
- Microprogramed control
- Administrivia; Courses
- Microprogram it yourself
- Exceptions
- Intro to Pipelining (if time permits)

Test benches

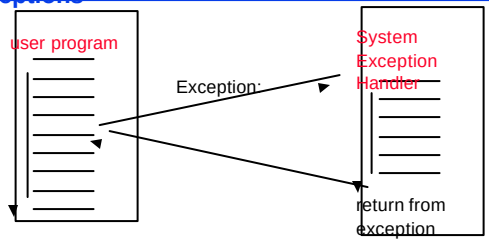
° Idea: wrap testing infrastructure around devices under test (DUT)

° Include test vectors that are supposed to detect errors in implementation. Even strange ones…

° Can (and probably should in later labs) include assert statements to check for "things that should never happen"



Exceptions



normal control flow:
sequential, jumps, branches, calls, returns

° Exception = unprogrammed control transfer

- system takes action to handle the exception
  - must record the address of the offending instruction
  - record any other information necessary to return afterwards
- returns control to user
- must save & restore user state

° Allows constuction of a "user virtual machine"

Two Types of Exceptions: Interrupts and Traps

° **Interrupts**
- caused by external events:
  - Network, Keyboard, Disk I/O, Timer
- asynchronous to program execution
  - Most interrupts can be disabled for brief periods of time
  - Some (like "Power Failing") are non-maskable (NMI)
- may be handled between instructions
- simply suspend and resume user program

° **Traps**
- caused by internal events
  - exceptional conditions (overflow)
  - errors (parity)
  - faults (non-resident page)
- synchronous to program execution
- condition must be remedied by the handler
- instruction may be retried or simulated and program continued or program may be aborted

Traps and Interrupts

° **exception means any unexpected change in control flow, without distinguishing internal or external;**

Type of event	*From where?*	*terminology*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Trap
Arithmetic overflow	Internal	Trap
Using an undefined instruction	Internal	Trap
Hardware malfunctions	Either	Trap or Interrupt

What happens to Instruction with Exception?

° **MIPS architecture defines the instruction as having no effect if the instruction causes an exception.**

° **When we get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.**

° **This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard**

Precise Interrupts

° **Precise þ state of the machine is preserved as if program executed up to 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
- Difficult in the presence of pipelining, out-ot-order execution, ...
- MIPS takes this position

° **Imprecise þ system software has to figure out what is where and put it all back together**

° **Performance goals often lead designers to not implement precise interrupts**
- system software developers, user, markets etc. usually wish they had not done this

° **Modern techniques for out-of-order execution and branch prediction help implement precise interrupts**

Big Picture: user / system modes

° By providing two modes of execution (user/system) it is possible for the computer to manage itself
  - operating system is a special program that runs in the privileged mode and has access to all of the resources of the computer
  - presents "virtual resources" to each user that are more convenient that the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
  - protects system from malicious users.
  - OS is assumed to "know best", and is trusted code, so enter system mode on exception.

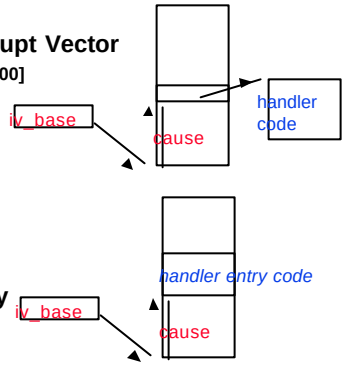
° Exceptions allow the system to taken action in response to events that occur while user program is executing:
  - Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
  - "Unimplemented instruction" used to emulate instructions that were not included in hardware

Addressing the Exception Handler

° Traditional Approach: Interupt Vector
  - PC <- MEM[IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, . . .

° RISC Handler Table
  - PC <– IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M88K, . . .

° MIPS Approach: fixed entry
  - PC <– EXC_addr
  - Actually very small table
    - RESET entry
    - TLB
    - other



Saving State

° Push it onto the stack
  - 68k, 80x86

° Shadow Registers
  - M88k
  - Save state in a shadow of the internal pipeline registers

° Save it in special registers
  - MIPS EPC, BadVaddr, Status, Cause

Additions to MIPS ISA to support Exceptions?

° Exception state is kept in "coprocessor 0".
  - Use mfc0 read contents of these registers
  - Every register is 32 bits, but may be only partially defined

BadVAddr (register 8)
- register contains memory address at which memory reference occurred

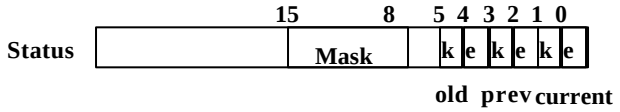
Status (register 12)
- interrupt mask and enable bits

Cause (register 13)
- the cause of the exception
- Bits 5 to 2 of this register encodes the exception type (e.g undefined instruction=10 and arithmetic overflow=12)

EPC (register 14)
- address of the affected instruction (register 14 of coprocessor 0).

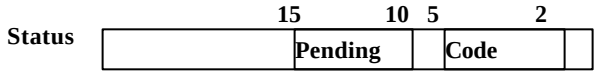
° Control signals to write BadVAddr, Status, Cause, and EPC
° Be able to write exception address into PC ($8000\ 0080_{hex}$)
° May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor): PC = PC - 4

Details of Status register



- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts
- k = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode
- e = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts were enabled
- When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0
  - run in kernel mode with interrupts disabled

Details of Cause register

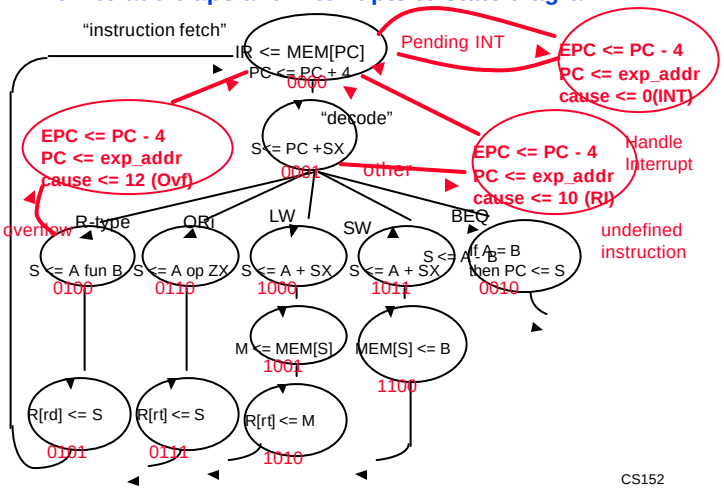


- Pending interrupt 5 hardware levels: bit set if interrupt occurs but not yet serviced
  - handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled
- Exception Code encodes reasons for interrupt
  - 0 (INT) => external interrupt
  - 4 (ADDRL) => address error exception (load or instr fetch)
  - 5 (ADDRS) => address error exception (store)
  - 6 (IBUS) => bus error on instruction fetch
  - 7 (DBUS) => bus error on data fetch
  - 8 (Syscall) => Syscall exception
  - 9 (BKPT) => Breakpoint exception
  - 10 (RI) => Reserved Instruction exception
  - 12 (OVF) => Arithmetic overflow exception

Example: How Control Handles Traps in our FSD

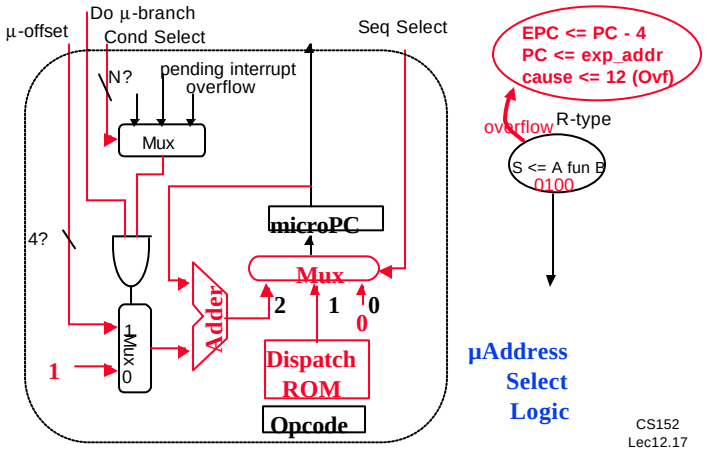
- Undefined Instruction–detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
  - Shown symbolically using "other" to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- Arithmetic overflow–detected on ALU ops such as signed add
  - Used to save PC and enter exception handler
- External Interrupt – flagged by asserted interrupt line
  - Again, must save PC and enter exception handler
- Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.
  - Complex interactions makes the control unit the most challenging aspect of hardware design

How to add traps and interrupts to state diagram



But: What has to change in our m-sequencer?

° Need concept of *branch* at micro-code level



Summary

° Microprogramming is a fundamental concept
- implement an instruction set by building a very simple processor and interpreting the instructions
- essential for very complex instructions and when few register transfers are possible
- *Control design reduces to Microprogramming*

° Exceptions are the hard part of control
- Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
- Providing clean interrupt model gets hard with pipelining!

° Precise Exception þ state of the machine is preserved as if program executed up to the offending instruction
- All previous instructions completed
- Offending instruction and all following instructions act as if they have not even started