EE 457 Unit 8

Exceptions

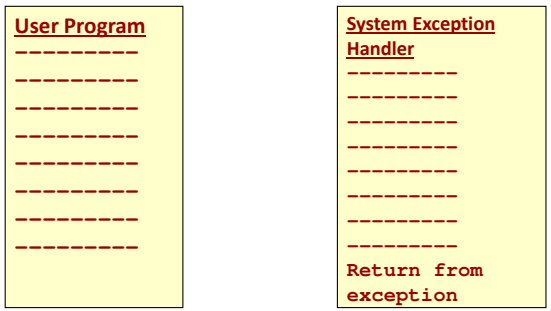
"What Happens When Things Go Wrong"

What are Exceptions?

- Exceptions are rare events triggered by the hardware and forcing the processor to execute a software handler
  - –
  - –
  - –
- Similar to conditional branches/subroutine calls
  - – Utilize the hardware already in place for branches
  - – Flush pipeline
  - – Fetch from entry point of software exceptions handler

Exception Processing

- Exception =
  - – Asynchronous (non-programmed) control transfer
  - – Synchronous system call/trap
- Must save PC of offending instruction, program state, and any information needed to return afterwards
- Restore upon return



Exception Examples 1

Example	Stage	Action
I/O Device Interrupt • A peripheral device requires action from the CPU (Interrupt I/O Driven)	WB	Take ASAP
Operating System Calls ("Traps") [e.g. File Open] • Trap instruction causes processor to enter kernel mode	____	Precise
Instruction Tracing and Breakpoints • When TRAP Bit is set all instructions cause exceptions • Particular instructions are flagged for exceptions (debugging)	ID	Precise
Arithmetic Exceptions • Overflow or Divide-by-0	EX	Precise

Exception Examples 2

Example	Stage	Action
Page Faults • Virtual memory access fault (no Page Table entry resident in memory)	_____	Precise
Misaligned Memory Address • Address is not multiple of operand size	____	Abort Process
Memory Protection Violations • Address is out of bounds; RWX violation	____	Abort Process
Undefined Instructions • Decode unit does not recognize opcode or other fields • Could be useful to extend the instruction set	____	Precise (Why not abort)
Hardware failure • Unrecoverable hardware error is detected; execution is compromised	WB	Take ASAP
Power Failure • Power has fallen below a threshold; Trap to software to save as much state as possible	WB	Take ASAP

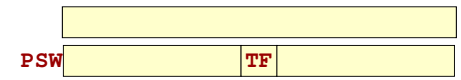
System Calls/Traps

- A controlled-method for user application calling OS services
- Switches processor to "kernel" mode where certain privileges are enabled that we would not want normal user apps to access

x86 System Call (old DOS OS call)

```
IN  AH, 01H
INT 20H     // getchar()
```

Instruction Tracing and Breakpoint
Single-stepping & Breakpoint in x86



Processor Status Word

Trap Flag

Exception Processing

- Save necessary state to be able to restart the process
  - Save ____of offending instruction
- Call an appropriate _____ to deal with the error / interrupt / syscall
  - Handler identifies cause of exception and handles it
  - May need to save more state
- Restore state and return to offending application (or kill it if recovery is impossible)

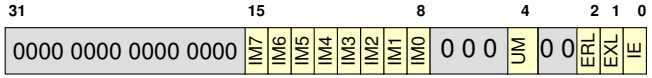
MIPS Coprocessor 0 Registers

- Status Register
  - Enables and disables the handling of exceptions/interrupts
  - Controls user/kernel processor modes
    - Kernel mode allows access to certain regions of the address space and execution of certain instructions
- Cause Register: Indicates which exception/interrupt occurred
- _____ Register
  - Indicates the address of the instruction causing the exception
  - This is also the instruction we should return to after handling the exception
- Coprocessor registers can be accessed via the 'mtc0' and 'mfc0' instructions
  - mfc0 $gpr,$c0_reg # R[gpr] = C0[c0_reg]
  - mtc0 $gpr,$c0_reg # C0[c0_reg] = R[gpr]

Status Register

- Register 12 in coprocessor 0
- Bit definitions
  - IM[7:0] – Interrupt Mask
    - 1 = Ignore interrupt / 0 = Allow interrupt
  - UM – User Mode
    - 1 = User mode / 0 = Kernel mode
  - ERL/EXL = Exception/Error Level
    - 1 = Already handling exception or error / 0 = Normal exec.
    - If either bit is '1' processor is also said to be in kernel mode
  - IE = Interrupt Enable
    - 1 = Allow unmasked interrupts / 0 = Ignore all interrupts

Status Register

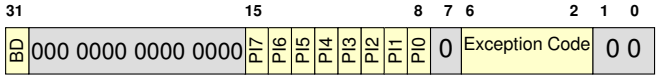


Cause Register

- Register 13 in coprocessor 0
- Bit definitions
  - BD – Branch Delay
    - The offending instruction was in the branch delay slot
    - EPC points at the branch but it was EPC+4 that caused the exception
  - PI[7:0] – Pending Interrupt
    - 1 = Interrupt Requested / 0 = No interrupt requested
  - Exception Code – Indicates cause of exception (see table)

Code	Cause
0	Interrupt (HW)
4, 5	Load (4), Store (5) Address Error
6, 7	Instruc. (6), Data (7) Bus Error
8	Syscall
9	Breakpoint
10	Reserved Instruc.
11	CoProc. Unusable
12	Arith. Overflow
13	Trap
15	Floating Point

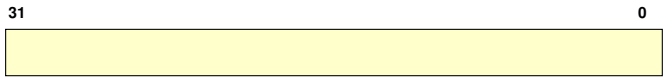
Cause Register



EPC Register

- Exception PC holds the address of the offending instruction
  - Can be used along with 'Cause' register to find and correct some error conditions
- _____ instruction used to return from exception handler and back to execution point in original code (unless handling the error means having the OS kill the process)
  - _____ Operation: PC = EPC

EPC = Exception PC



Problem of Calling a Handler

- We can't use explicit 'jal' instructions to call exception handlers since we don't

_____

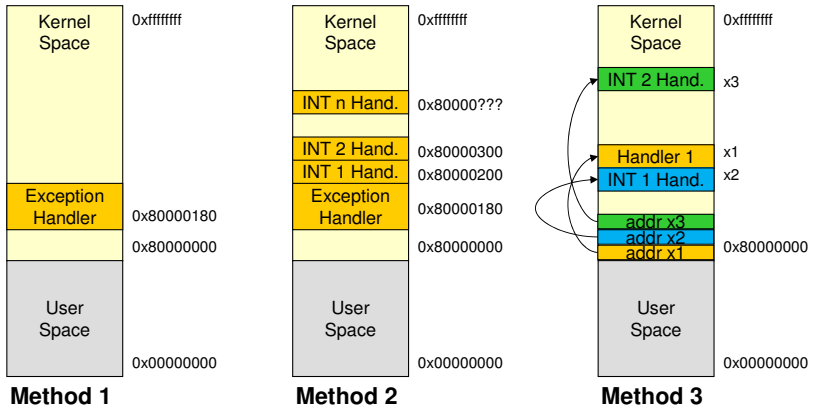
```
        .text
MAIN:   ----
        ----
        ----
        ----
        ----
        jr    $ra
```

Many instructions could cause an error condition. Or a hardware event like a keyboard press could occur at any point in the code.

Solution for Calling a Handler

- Since we don't know when an exception will occur there must be a preset location where an exception handler should be defined or some way of telling the processor in advance where our exception handlers will be located
- Method 1: _____ for master handler
  - Early MIPS architecture defines that the exception handler should be located at 0x8000_0180. Code there should then examine CAUSE register and then call appropriate handler routine
- Method 2: _____ (usually for interrupts)
  - Each interrupt handler at a different address based on interrupt number (a.k.a. vector) (INT1 @ 0x80000200, INT2 @ 0x80000300)
- Method 3: _____
  - Table in memory holding start address of exception handlers (i.e. overflow exception handler pointer at 0x0004, FP exception handler pointer at 0x0008, etc.)

Handler Calling Methods



Precise Exceptions

- Two conditions:
  - Synchronized with an instruction
    - A particular instruction caused the exception
    - Not an interrupt or some kind of failure
  - Must resume execution after handler
    - Restart the instruction causing the exception after exception handler returns
    - Not an exception that will cause the process to abort
- Not difficult in a processor executing one instruction at a time
- Very difficult in architectures in which multiple instruction execute concurrently (i.e. our 5-stage pipeline)

Why are Exceptions So Important?

- Exceptions are part of the ISA (Instruction Set Architecture) specification
- Any implementation of an ISA must comply with its "Exception model"
- Precise exception handling constrains what the architecture can do
  - Exceptions are rare yet we must functionally support them
  - If we did not have to comply to the exception model architects would have a lot more freedom in their design

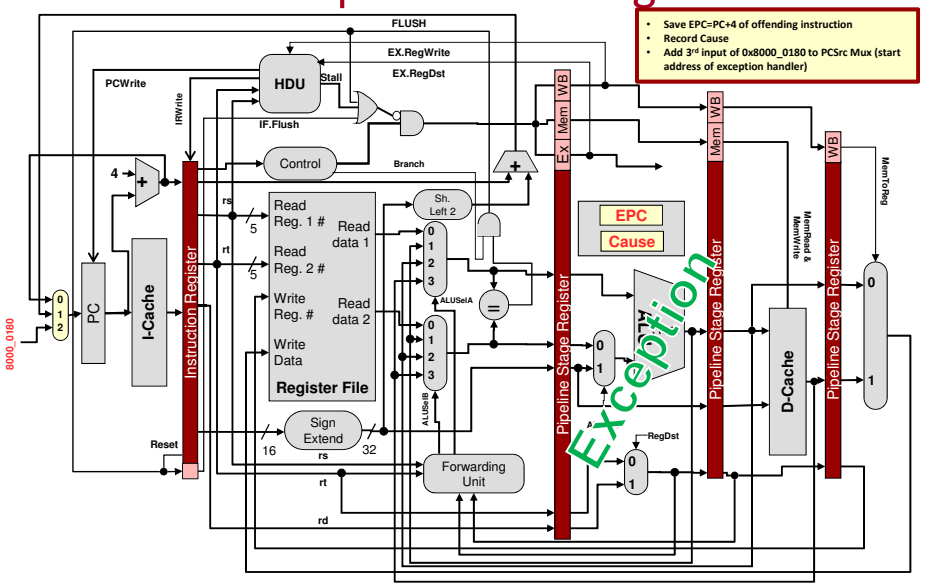
When designing micro-architectures for the common case, exceptions must always be in the back of your mind!

Exceptions in the 5-Stage Pipeline

- To support precise exceptions in the 5-stage pipeline we must…
  - Identify the pipeline stage and instruction causing the exceptions
    - Any stage can trigger an exception (except for the WB stage)
  - Identify the cause of the exception
  - Save the process state at the faulting instruction
    - Including registers, PC, and cause
    - Usually done by software exception handler
  - Complete the execution of instructions preceding the faulting instruction
  - Flush instruction following the faulting instruction plus the faulting instruction
  - Transfer control to exception handler

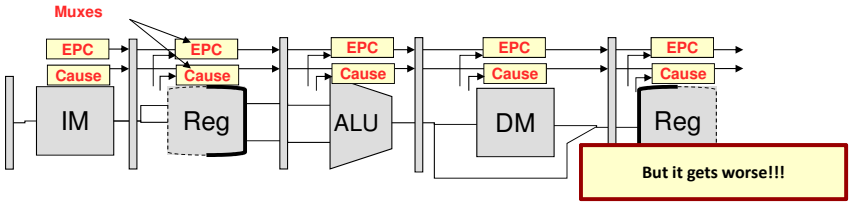
Use many of the same mechanisms as conditional branches.

Exception in EX stage



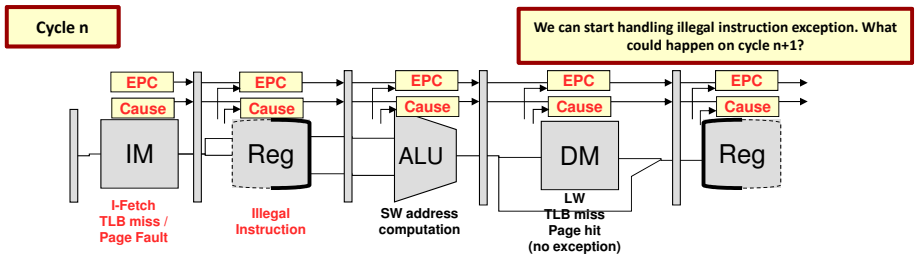
Exception Handling Complexities

- When the arithmetic exception is triggered in EX, we must flush __, ___and ___ and start fetching from 0x8000_0180
- Note that the handler's software must have access to CAUSE and EPC registers to figure out what to do
- Realize though exceptions may occur in all but the WB stage
  - 4 possible values of _____
  - Software needs to know which value is the actual cause and EPC
  - Depending on the stage where the exception occurs, we have to flush different stages



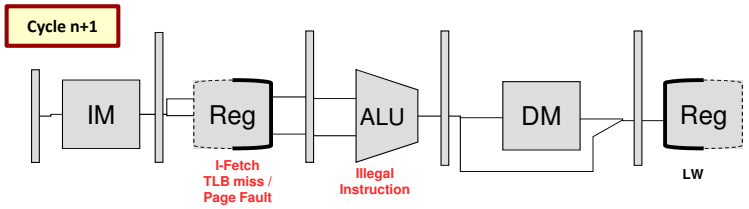
More Complex Complexities?

- What happens if multiple exceptions occur in the same cycle from different instructions in different stages
  - Should take the _____ exception in _____
  - "Program/process order" = Order if only 1 instruction were executed at a time (= Fetch order)
  - Thus oldest instruction is the one _____ into the pipeline
  - There is no point in dealing with all exceptions, just the oldest one
  - Let software deal with the oldest and then restart…if later instruction were going to generate an exception, then they will again upon restart and we can handle it then



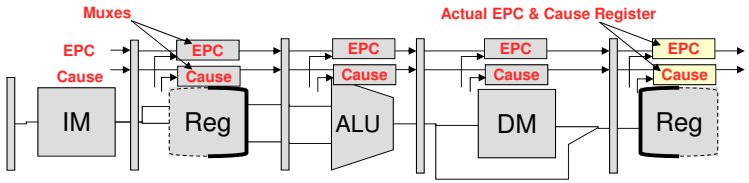
More Complex Complexities?

- Remember we must complete instruction preceding the faulting instruction
- Remember we are supposed to handle instruction in program order (not temporal order)



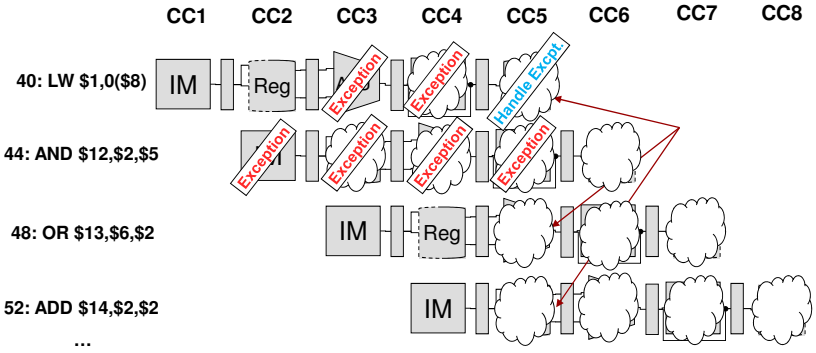
Simplify the Process

- It is not practical to take an exception in the cycle when it happens
  - Multiple exceptions in the same cycle
  - It is complex to take exception in various pipeline stages since we have to take them in program order and not temporal order
- Instead, we will just tag an instruction in the pipeline if it causes and exception (recording the cause and EPC)
  - Turn the offending instruction into a NOOP (bubble)
  - Let the instructions continue to flow down the pipeline and handle the offending instruction's execution in the WB stage
    - The cause and status info is carried down the pipe via stage registers
  - Exception remains _____ until it reaches the WB stage
  - Exceptions are then processed into the WB stage



Handling in WB Stage

- Handling in WB stage helps deal with temporal vs. program order issues



Simplified Processing

- Precise exceptions are now taken in WB along with other HW interrupts
- Faulting instructions "carry" their cause and EPC values through the pipeline stage registers
- Only one set of EPC and CAUSE registers in the WB stage
- When an instruction flagged as faulting reaches the WB stage
  - Flush IF, ID, EX, MEM
    - Make sure that if a SW is in MEM stage that it is not allowed to write
  - Load the handler address in the PC
  - Make sure EPC & Cause are software-readable (movable to GPR's)

This is a general approach to dealing with exceptions in the processor:
Wait until the faulting instruction exits the machine to trigger the handling procedure