Lecture 11: Exceptions and Interrupts

Jangwoo Kim (Seoul National University)
jangwoo@snu.ac.kr

Announcement

- HW#3
  - Posted!
  - **Due: 4/19**
- Mid-term
  - **4/20 (6PM) @ Room 102**
- One more lecture before mid-term exam
  - **Pick between 4/14 (2pm, Saturday)** or **4/17 (5pm, Tuesday)**

This will help you better prepare for mid-term exam

CPU must prepare for '**unplanned**' events

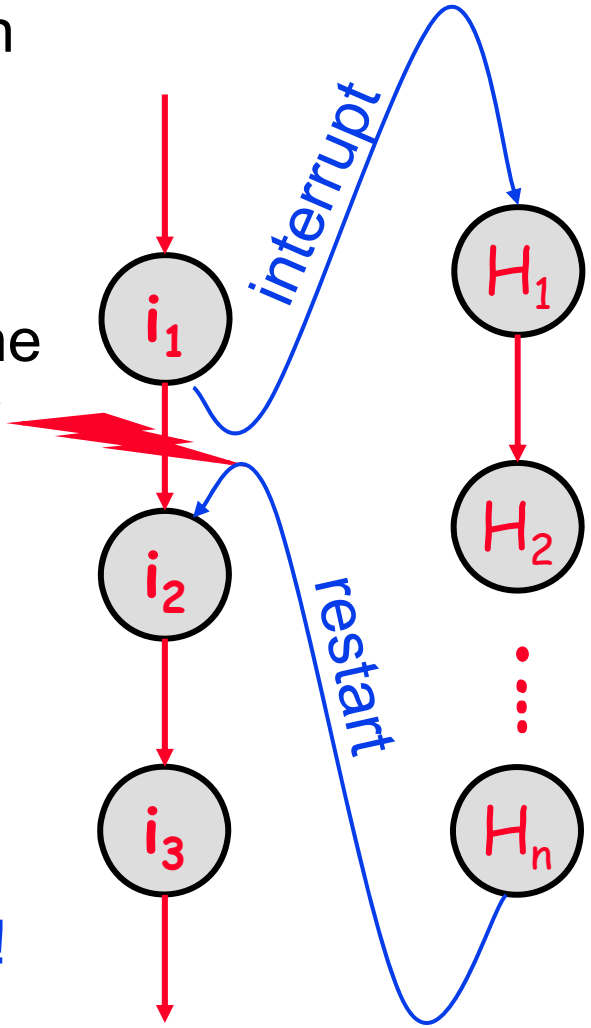
- A systematic way to handle exceptional conditions that are relatively rare, but must be detected and acted upon quickly
  - Instructions may fail and cannot complete
  - External I/O devices may need servicing
  - Quantum expiration in a time-shared system
- Option 1: write every program with continuous checks (a.k.a. polling) for every possible contingency

  acceptable for simple embedded systems (e.g., toaster)
- Option 2: write "normal" programs for the best-case scenario where nothing unusual happens
  - But, must detect exceptional conditions in HW
  - "Transparently" transfer control to an exception handler that knows how to resolve the condition and then back to your program

Interrupt Control Transfer

- An interrupt is an "unplanned" function call to a system routine (a.k.a., the interrupt handler)
- Unlike a normal function call, the interrupted thread cannot anticipate the control transfer or prepare for it in any way
- Control is later returned to the main thread at the interrupted instruction

The control transfer to the interrupt handler and back must be 100% transparent to the interrupted thread!!!



Types of Interrupts

- Synchronous Interrupts (a.k.a., **exceptions**)
  - Exceptional conditions **tied to a particular instruction**

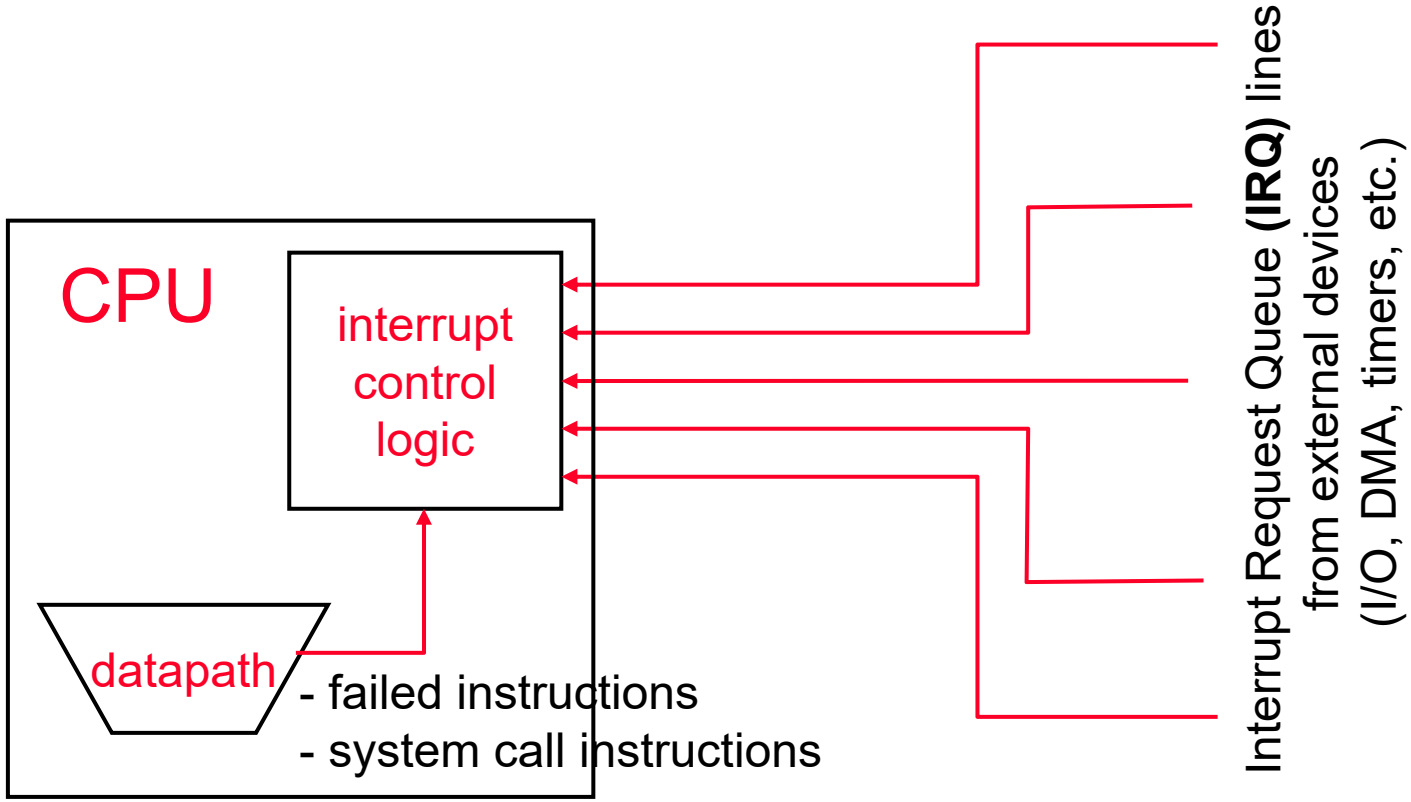
    e.g. illegal opcode, illegal operand, virtual memory management faults
  - The faulting instruction cannot be finished

  No forward-progress unless handled immediately
- Asynchronous Interrupts (a.k.a., **interrupts**)
  - External events **not tied to a particular instruction**

    e.g., I/O events, timer events
  - Some flexibility on when to handle it

  Some delay allowed, but still cannot postpone forever
- System Call/Trap Instruction
  - An instruction whose only purpose is **to raise an exception**

Interrupt Sources



Exceptions
(a.k.a. synchronous interrupts)

Interrupts
(a.k.a. asynchronous interrupts,
or external interrupts)

Virtualization and Protection

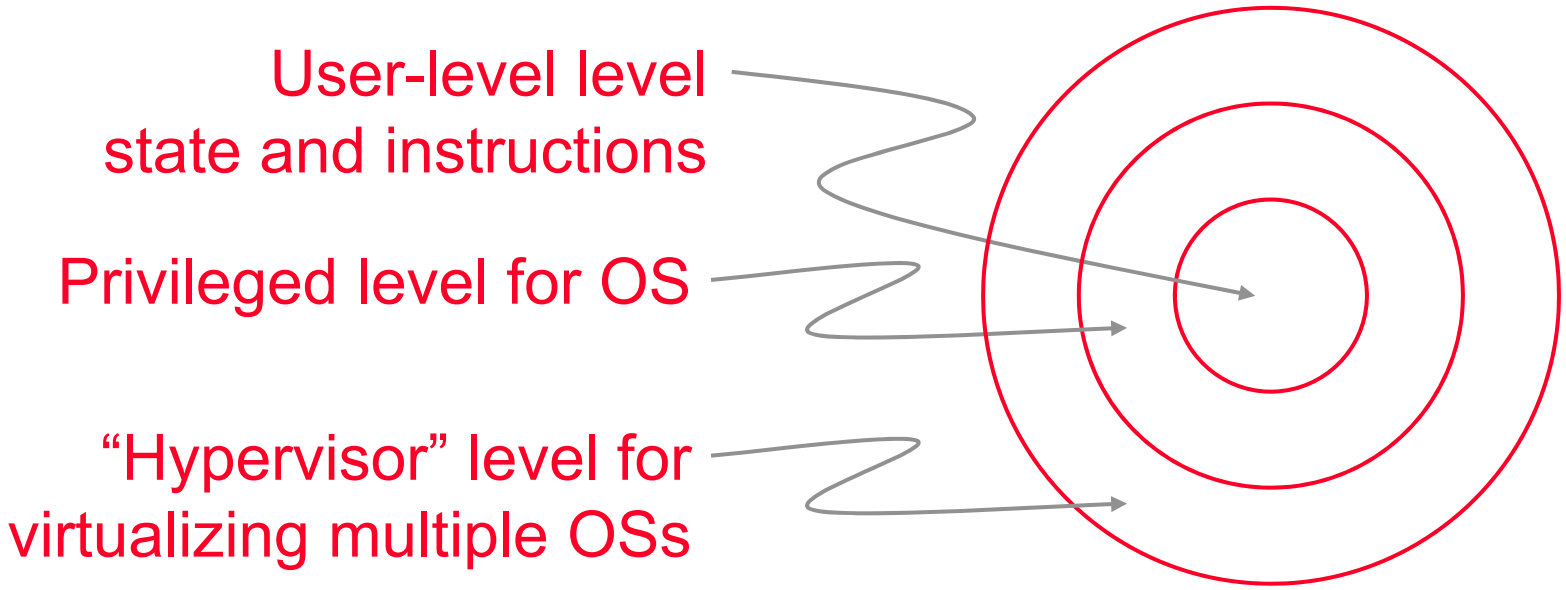
- Modern OS supports time-shared multiprocessing but each "user-level" process still thinks it is alone
  - Each process sees a private set of user-level architectural states that can be modified by the user-level instruction set
  - Each process cannot see or manipulate (directly) state and devices outside of this abstraction
- OS implements and manages a critical set of functionality
  - Keep low-level details out of the user-level process
    (e.g., HW device handler, thread scheduler, ..)
  - Protect the user-level process from each other and itself

Do you want to access/manage hard disk directly?

Do you trust your friend if he accesses the hard disk directly?

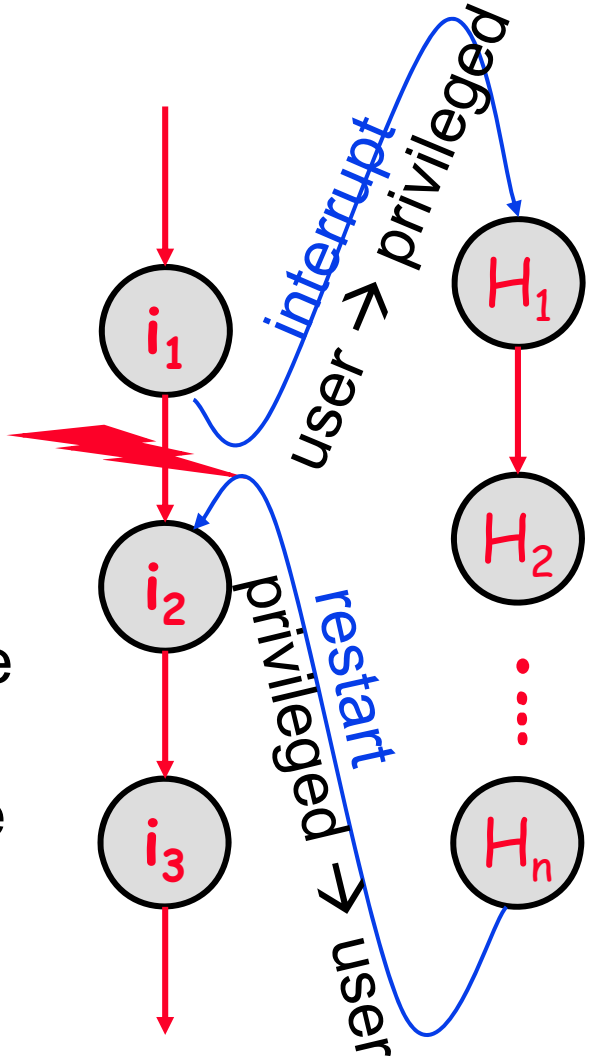
Privilege Levels

- The OS must somehow be more powerful to create and maintain such an abstraction, hence a separate privileged (a.k.a. protected or kernel) mode
  - Additional architectural states and instructions, in particular to control virtualization/protection/isolation
  - The kernel code running in the privileged mode has access to the complete "bare" hardware system



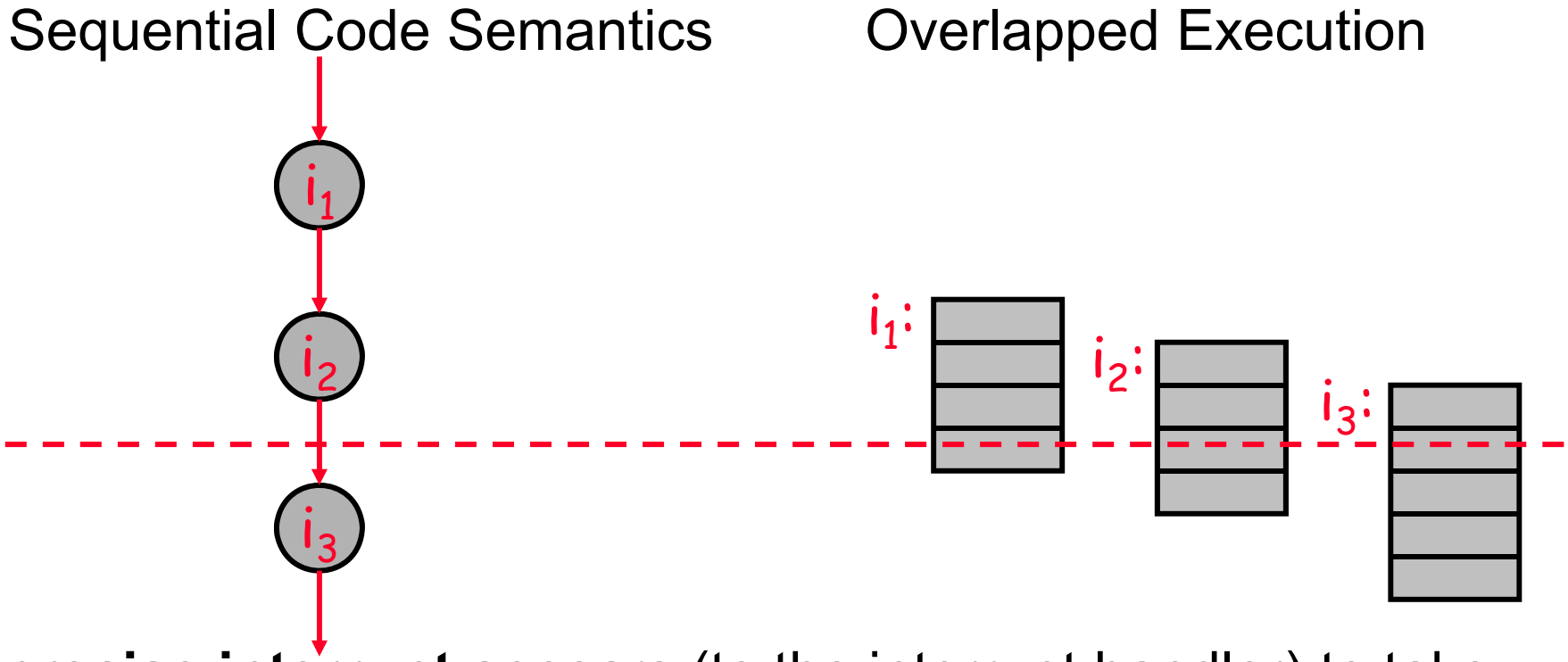
Control and Privilege Transfer

- User-level code never runs in the privileged mode
- Processor enters the privileged mode only **on interrupts**---user code surrenders control to a handler in the OS kernel
- The handler restores privilege level back to user mode before returning control to the user code



Implementing Interrupts

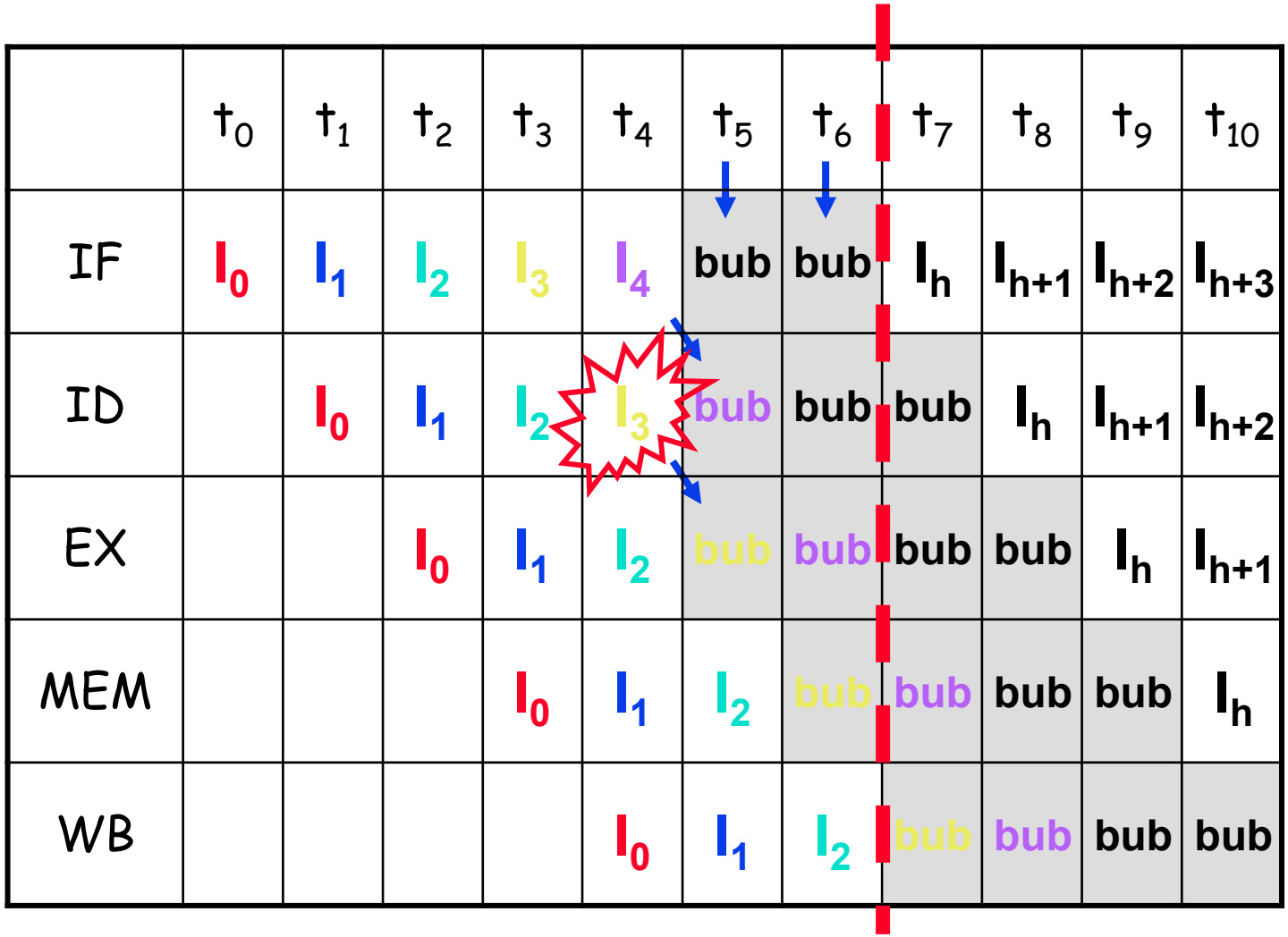
"Precise" Interrupt/Exception



A **precise interrupt** appears (to the interrupt handler) to take place exactly between two instructions

- Older instructions finished completely
- Younger instructions as if never happened
- On synchronous interrupts, execution stops just before the faulting instruction, and resume after interrupt handling done

Stopping and Restarting a Pipeline

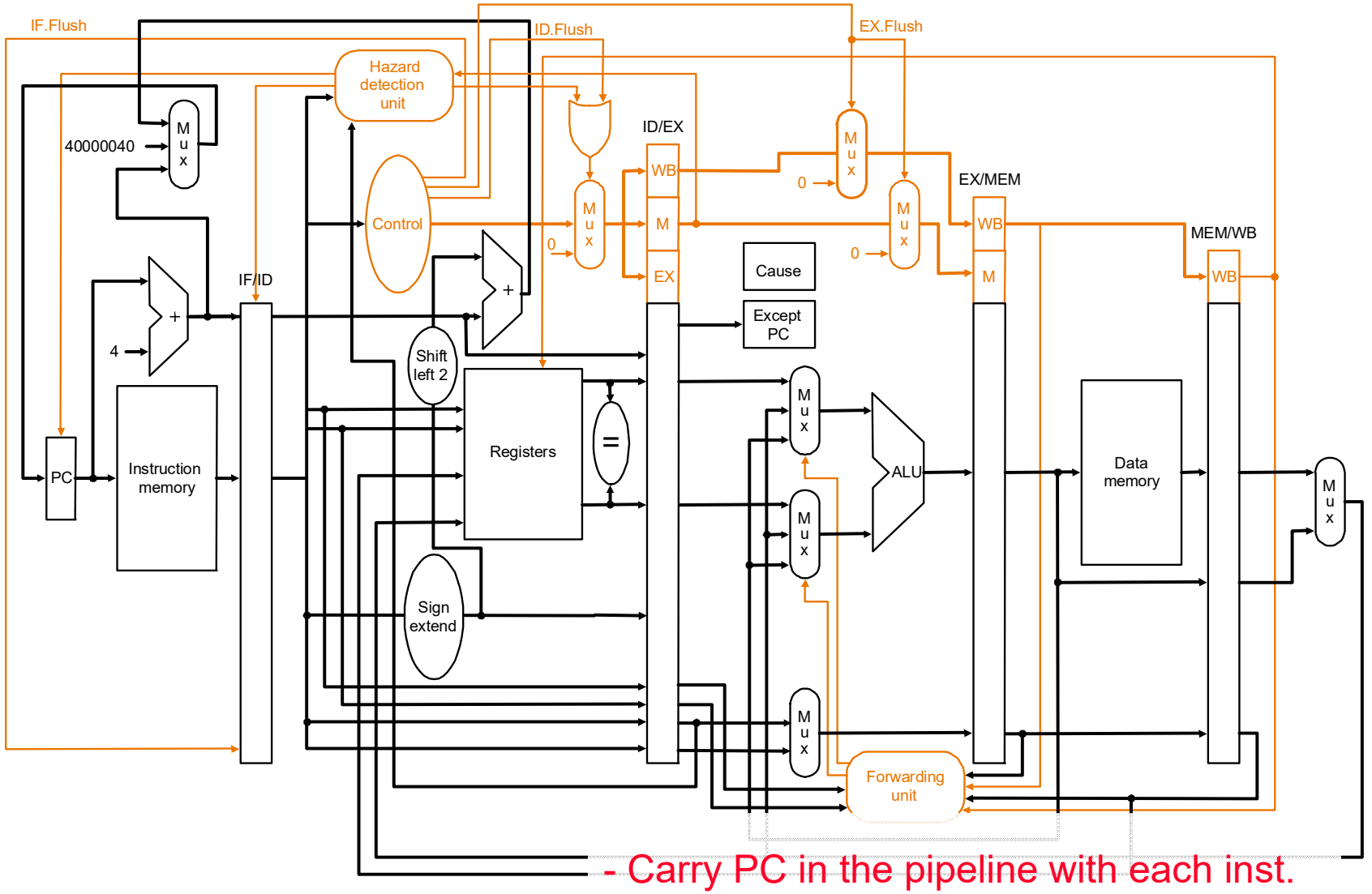


What if I_0, I_1, I_2, I_3 and I_4 all generate exceptions in t_4?
How would things look different for asynchronous interrupts?

Exception Sources in Different Stages

- IF
  - Instruction memory address/protection fault
- ID
  - Illegal opcode
  - Trap to SW emulation of unimplemented instructions
  - System call instruction (an intended exception requested by SW)
- EX
  - Invalid results: overflow, divide-by-zero, …
- MEM
  - Data memory address, protection fault
- WB
  - Nothing can stop an instruction now…
- We can associate async interrupts (I/O) with any instruction/stage we like

Pipeline Flush for Exceptions

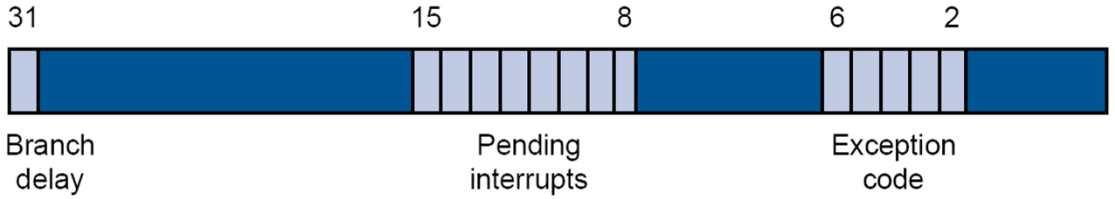


- Carry PC in the pipeline with each inst.
- New pipeline flush points

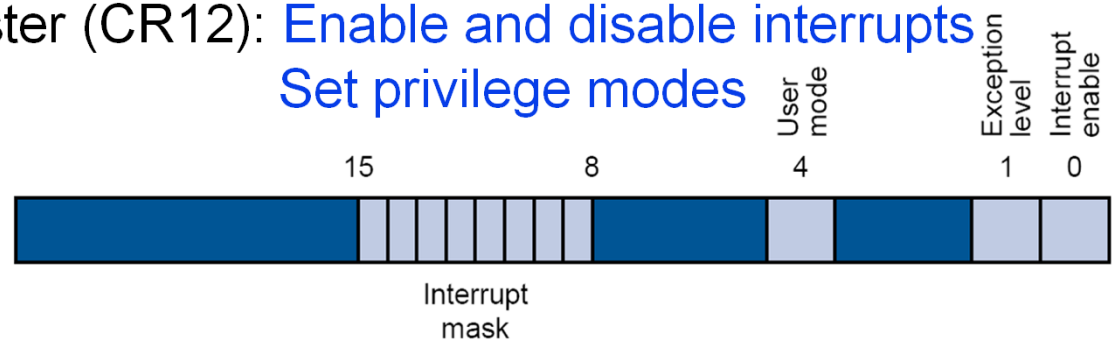
MIPS Interrupt Architecture

MIPS Interrupt Architecture

- Privileged system control registers
  - Exception Program Counter (EPC, CR14)
    - Which instruction did we stop on?
  - Interrupt Cause Register (CR13) : What caused the interrupt?



  - Interrupt Status Register (CR12): Enable and disable interrupts
    Set privilege modes



- Automatically set on interrupt transfer events
- Also accessed by the "move from/to control register-x" instruction: "`mfc0 Ry, CRx`" and "`mtc0 Ry, CRx`"

MIPS Interrupt Architecture

- On an interrupt transfer, the CPU hardware saves the interrupt address to EPC
  - Can't just leave frozen in the PC: overwritten immediately
  - Can't use r31 as in a function call: need to save user value
- In general, CPU hardware must save any such information that cannot be saved and restored in software by the interrupt handler (very few such things)
- For example, the GPR can be managed in SW by the interrupt handler using a **callee-saved** convention
  - However, r26 and r27 are reserved by convention to be available to the kernel immediately at the start and the end of an interrupt handler

    → r26 & r27 (or $k0~$k1: reserved for OS kernel)

Interrupt Servicing

- On an interrupt transfer, the CPU hardware records the cause of the interrupt in a privileged registers (Interrupt Cause Register, CR13)
- Option 1: Control is transferred to a pre-fixed default interrupt handler address **// flexible, but slow**
  - This initial handler examines the cause and branches to the appropriate handler subroutine to do the work
  - This address is protected from user-level process so one cannot just jump or branch to it
- Option 2: Vectored Interrupt **// not flexible, but fast**
  - A bank of privileged registers to hold a separate specialized handler address for each interrupt source
  - On an interrupt, hardware transfer control directly to the appropriate handler to save interrupt overhead

Example of Causes

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Handler Examples

- On asynchronous interrupts, device-specific handlers are invoked to service the I/O devices
- On exceptions, kernel handlers are invoked to either
  - Correct the faulting condition and continue the program (e.g., emulate the missing FP functionality, update virtual memory management), or
  - "signal" back to the user process if a user-level handler function is registered, or
  - Kill the process if the exception cannot be corrected
- "System call" is a special kind of function call from user process to kernel-level service routines

Returning from Interrupt

- Software restores all architectural states saved at the start of the interrupt routine

(1) MIPS32 uses a special jump instruction

- ERET
  - > To atomically restore the automatically saved CPU states
  - > To atomically restore the privilege level
  - > To atomically jump back to the interrupted address in EPC

(2) MIPS R2000 uses a pair of instructions

```
(assume "mfc0 r26,epc" done)
jr r26      // jump to a copy of EPC in r26
rfe         // restore from exception mode
            // must be used in the delay slot!!
```

Branch Delay Slot and RFE

- What if the faulting address is a branch delay slot?
  - Simply jumping back to the faulting address won't continue correctly if the preceding branch was taken
  - We didn't save enough information to do the right thing
- MIPS's solution
  - CPU HW makes a note (in the Cause register) if the faulting address captured is in a delay slot
  - In these cases, the handler returns to the preceding branch instruction which gets executed twice
- Generally harmless except "JALR r31"
  - Explicitly disallowed by the MIPS ISA

  What would happen if it is allowed?

An Extremely Short Handler

```
_handler_shortest:
    # no prologue needed

    . . . short handler body . . .      # can use only r26 and r27
                                        # interrupt not re-enabled for
                                        # something really quick

    # epilogue
    mfc0 r26,epc                        # get faulting PC
    jr r26                                      # jump to retry faulting PC
    rfe                                 # restore from exception mode
                                                = restore arch. state
```

A Short Handler

```
_handler_short:
    # prologue
    addi sp, sp -0x8          # allocate stack space (8 byte)
    sw r8, 0x0(sp)            # back-up r8 and r9 for use in body
    sw r9, 0x4(sp)            #

    . . . short handler body . . .   # can use r26, r27, and r8, r9
                                     # interrupt not re-enabled

    # epilogue
    lw r8, 0x0(sp)            # restore r8, r9
    lw r9, 0x4(sp)            #
    addi sp, sp, 0x8          # restore stack pointer
    mfc0 r26,epc              # get EPC
    j r26                     # jump to retry EPC
    rfe                       # restore from exception mode
```

Nesting Interrupts

- On an interrupt control transfer, further exceptions or asynchronous interrupts are disabled automatically
  - Another interrupt would overwrite the contents of the EPC and Interrupt Cause and Status Registers

  (1) The handler must be carefully written not to generate synchronous exceptions itself during this window of vulnerability

  (2) The handler must disable further asynchronous interrupts using interrupt status register (CR12)

- However, for long-running handlers, interrupt must be re-enabled not to miss critical interrupts
  - The handler must save the contents of EPC/Cause/Status to memory (stack) before re-enabling asynchronous interrupt
  - Once interrupts are re-enabled, EPC/Cause/Status can be updated by the next interrupt

Interrupt Priority

- Asynchronous interrupt sources are ordered by priorities
  - Higher-priorities interrupts are more timing critical
  - If multiple interrupts are triggered, the handler handles the highest-priority interrupt first
- Interrupts from different priorities can be selectively disabled by setting the mask in the Status register
- When servicing a particular priority interrupt, the handler only re-enable higher-priority interrupts
  - Higher-priority interrupt should not get delayed

Re-enabling same/lower-priority interrupts may lead to an infinite loop if a device interrupts repeatedly

Nestable Handler

```
_handler_nest:
    # prologue
    addi sp, sp, -0x8          # allocate stack space for EPC
    mfc0 r26, epc              # get EPC
    sw r26, 0x0(sp)            # store EPC onto stack
    sw r8, 0x4(sp)             # allocate a register for use later
    addi r26, r0, 0x405        # set interrupt enable bit
    mtc0 r26, status           # write into status reg

    . . . interruptible        # could free-up more registers
        longer handler body . . .  # to stack if needed

    # epilogue
    addi r8, r0, 0x404         # clear interrupt enable bit
    mtc0 r8, status            # write into status reg
    ld r26, 0x0(sp)            # get EPC back from stack
    ld r8, 0x4(sp)             # restore r8
    addi sp, sp, 0x8           # restore stack pointer
    j r26                      # jump to retry EPC
    rfe                        # restore from exception mode
```

Question?

Announcements: Homework #3 will be collected (due: 4/19)
Mid-term on 4/20 (starting at 6:00PM)
Reading: Finish P&H Ch. 4 & Early part of Ch. 5
Handouts: None