Lecture 10:
CPU – exception

Hunjun Lee
hunjunlee@hanyang.ac.kr

CPUs must prepare for unexpected events

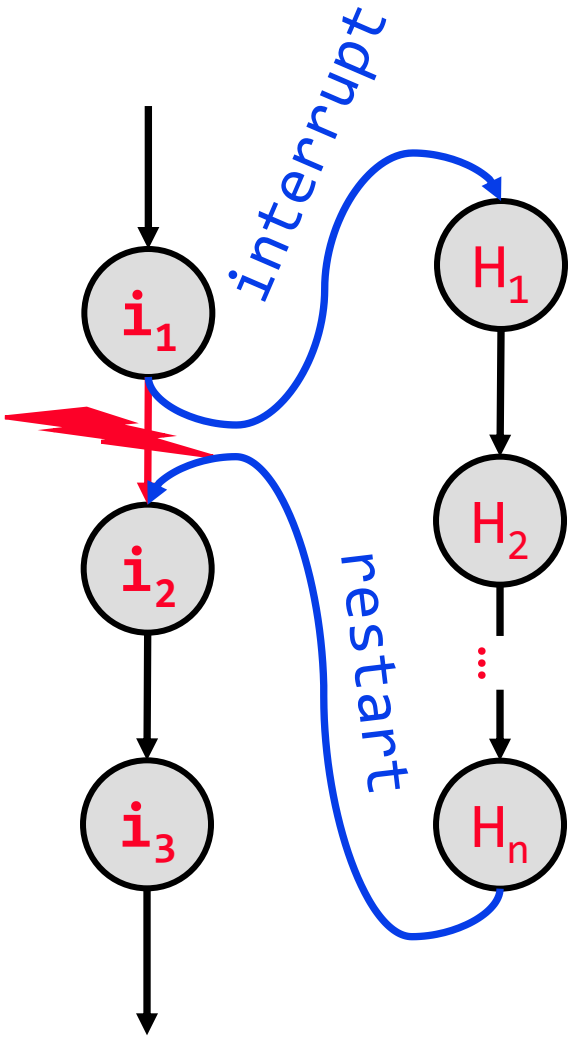
- The CPUs must detect and handle unexpected conditions
  - Instructions may fail and cannot complete
  - External I/O devices

- Option #1: The SW continuously checks every possible unexpected events (i.e., polling)
  - Acceptable for simple embedded systems

- Option #2: The HW dynamically detects unexpected events
  - Transparently transfer control to an exception handler (that knows how to resolve the condition and move back to your program)

Interrupt control transfer

- Unlike a normal function call, the interrupted thread (we'll get to this later) does not anticipate control transfer
- Control should be later returned to the main thread
- The control transfer should be 100% transparent to the interrupted thread



Types of Interrupts

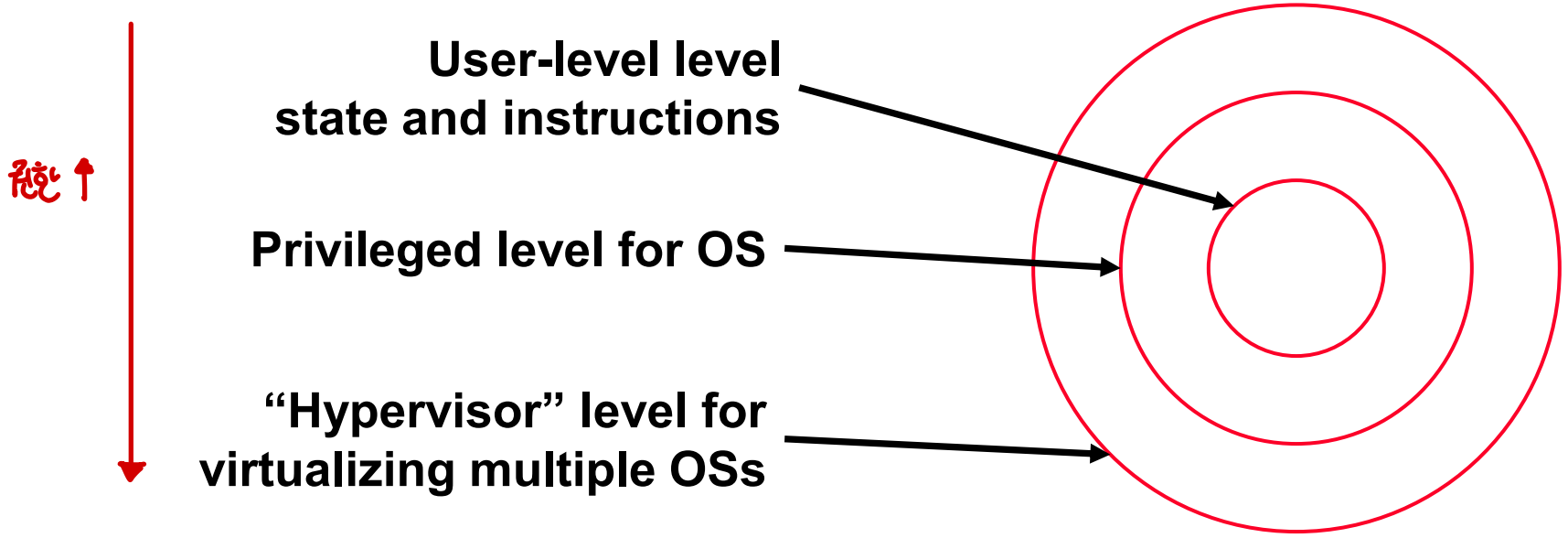
- Exceptions: Synchronous interrupts
  - Caused by (Tied to) a specific instruction
    - Illegal opcode, div by zero, arithmetic overflow, page fault …
  - **Should be handled immediately!**

- Interrupts: Asynchronous interrupts
  - External events (not tied to a particular instruction)
    - Ex) I/O events, power failure, …
  - **Can be postpone to some extent**
    - However, there exists a priority depending on the type

- System call/trap instruction
  - An instruction whose only purpose is **to raise an exception** (e.g., print …)

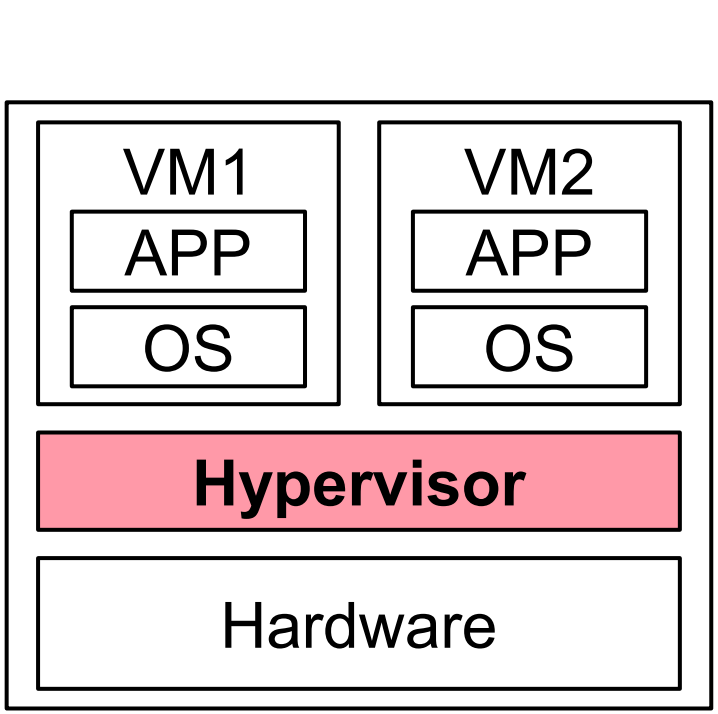
Privilege levels

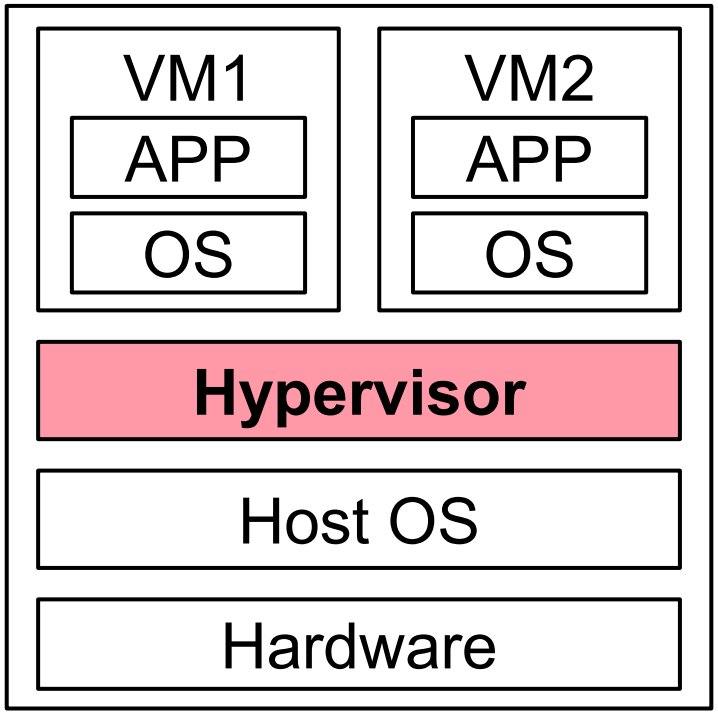
- The OS must somehow be more powerful to ensure transparency!
- There are multiple privilege modes depending on what should be handled!



Virtualization

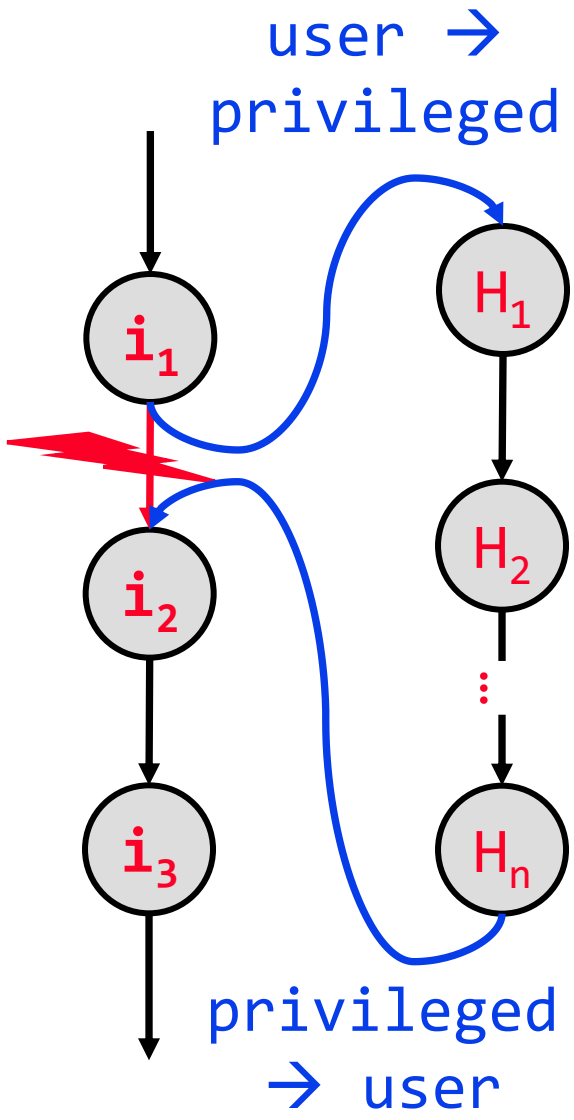
- Virtualization is a software to create **virtual representations of servers, storage, networks, and other physical machines**
  - This enables the software to control the number of allocated resources





Privilege transfer

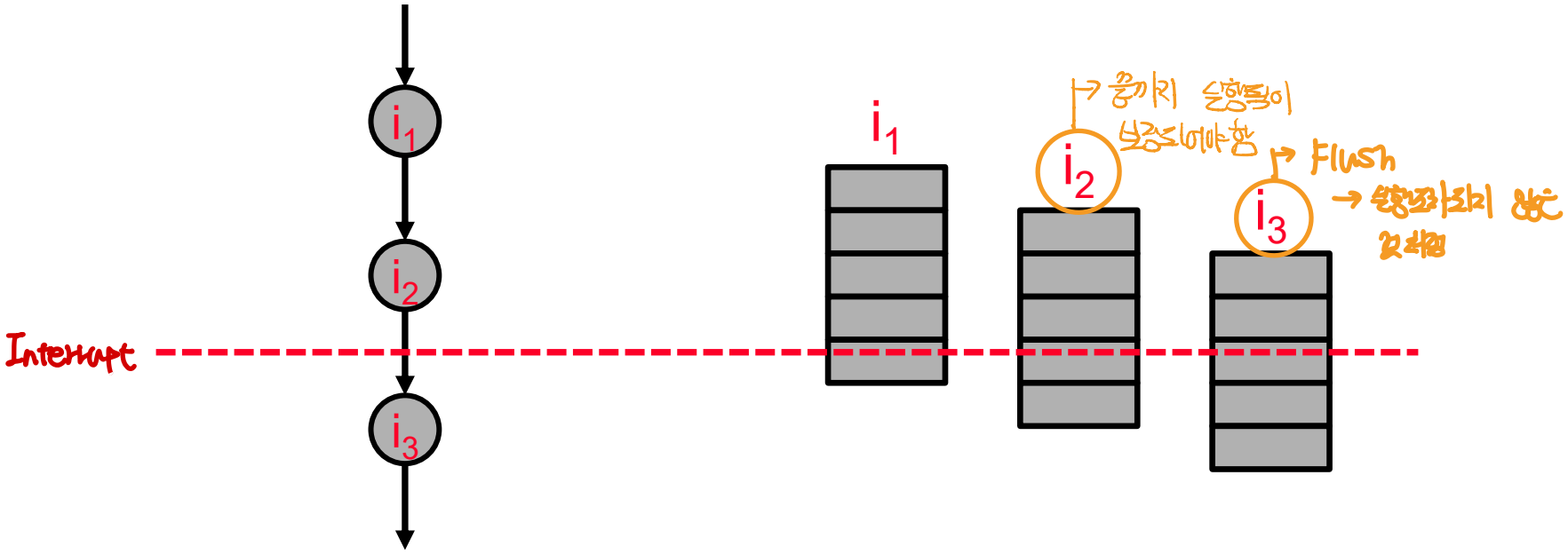
- User-level code does not run in the privileged mode
- CPU enters the privileged mode on interrupts (user code transfers control to the OS kernel)
- The handler restores privilege level back to the user mode (before restarting the user code)



Precise interrupts & execution

Interrupt 발생한 위치와 보장되는 Interrupt

- A precise interrupt appears to take place "exactly" in between two instructions
  - Older instructions complete
  - Younger instructions are flushed (as if never happened)
- For synchronous interrupts, execution stops just before the faulting instruction, and resume after handling is done



Supporting a precise exception

- When an instruction is detected to have caused an exception, there should be the following mechanisms
  - Flushes all younger instructions
  - Keep the architectural states precise (register file, PC, memory)
    - Saves PC and registers
  - Redirects the execution to the appropriate exception handling instructions

Why precise exceptions?

- Because the ISA says so …
  - The programmer thinks that the instructions change the architectural states in order
- For debugging …
  - You'll know why if you have ever debugged a multithread program
- Easy recovery from exceptions
  - We do not need to change the states when handling exceptions
- Easy to restart the process after the exception
- Also, you can think of syscall (e.g., print)

Supporting a precise exception

- When an instruction is detected to have caused an exception, there should be the following mechanisms

  - Flushes all younger instructions

  - Keep the architectural states precise (register file, PC, memory)
    - Saves PC and registers

  - Redirects the execution to the appropriate exception handling instructions

Stopping and restarting a pipeline

Exception for I_3 detected @ ID stage

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
IF	I_0	I_1	I_2	I_3	I_4	bub	bub	I_h	I_{h+1}	I_{h+2}	I_{h+3}
ID		I_0	I_1	I_2	I_3	bub	bub	bub	I_h	I_{h+1}	I_{h+2}
EX			I_0	I_1	I_2	bub	bub	bub	bub	I_h	I_{h+1}
MEM				I_0	I_1	I_2	bub	bub	bub	bub	I_h
WB					I_0	I_1	I_2	bub	bub	bub	bub

Exception

Handling

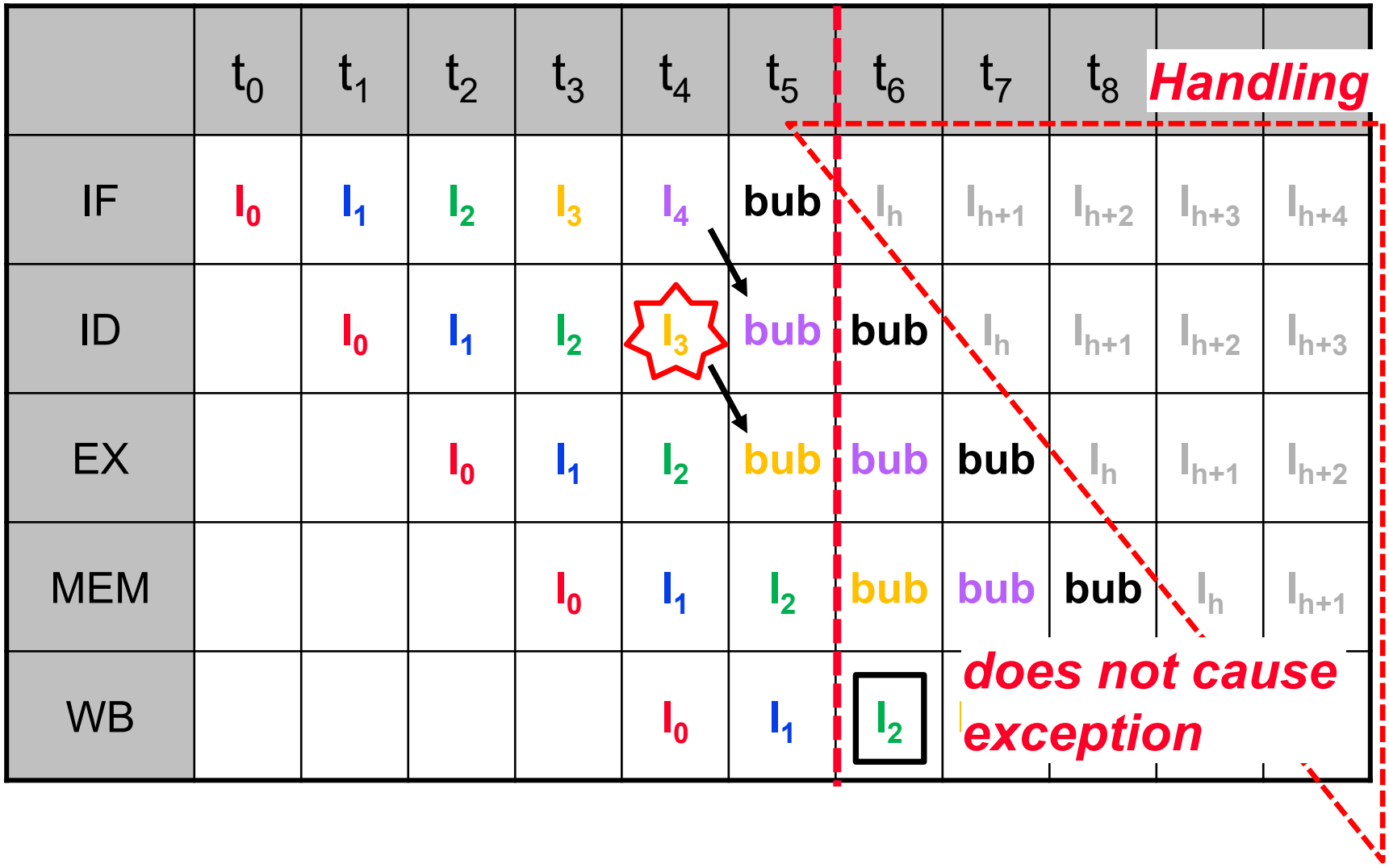
How would things look different for asynchronous interrupts

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, page fault
- **WB**
  - There is no exception by now …

Stopping and restarting a pipeline

Exception for I_3 detected @ ID stage



Supporting a precise exception

- When an instruction is detected to have caused an exception, there should be the following mechanisms

  - Flushes all younger instructions

  - Keep the architectural states precise (register file, PC, memory)
    - Saves PC and registers

  - Redirects the execution to the appropriate exception handling instructions

Interrupt handling @ MIPS - 1

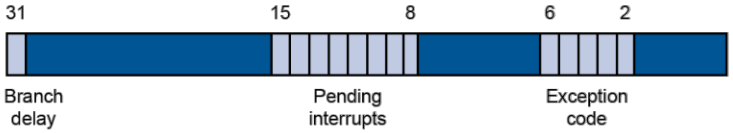
- Save some architectural states before calling an interrupt handler
  - Saves PC to the Exception program counter (EPC)
  - Saves r26, r27 registers ($k0~$k1: reserved for OS kernel)
- How to know the cause of the interrupt
  - CPU records the cause of the interrupt in a privileged registers
  - **Option #1)** Use cause register to indicate the problem
    - The initial handler uses the register to jump to another handler at the kernel space (flexible, but slow)
  - **Option #2)** Jump to different address depending on the cause
    - The HW determines which instruction to jump to (inflexible, but fast)
    - `Undefined opcode:` `8000 0000`
    - `Overflow:` `8000 0180`

Interrupt handling @ MIPS - 2

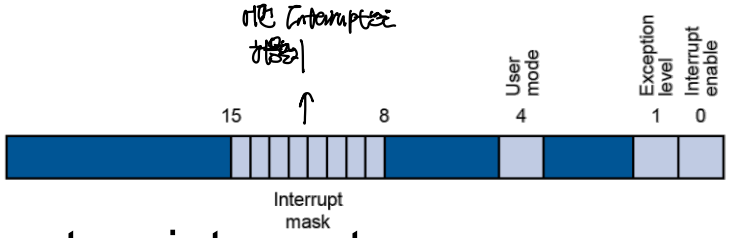
- If the interrupt is can be corrected ➔ take corrective action and return to the EPC
- Otherwise, terminate the program and report error

Interrupt handling @ MIPS - 3

- Details on interrupt handling registers
  - Exception program counter (EPC, CR14)
  - Interrupt cause register (CR13):
    - **What caused the interrupt?**

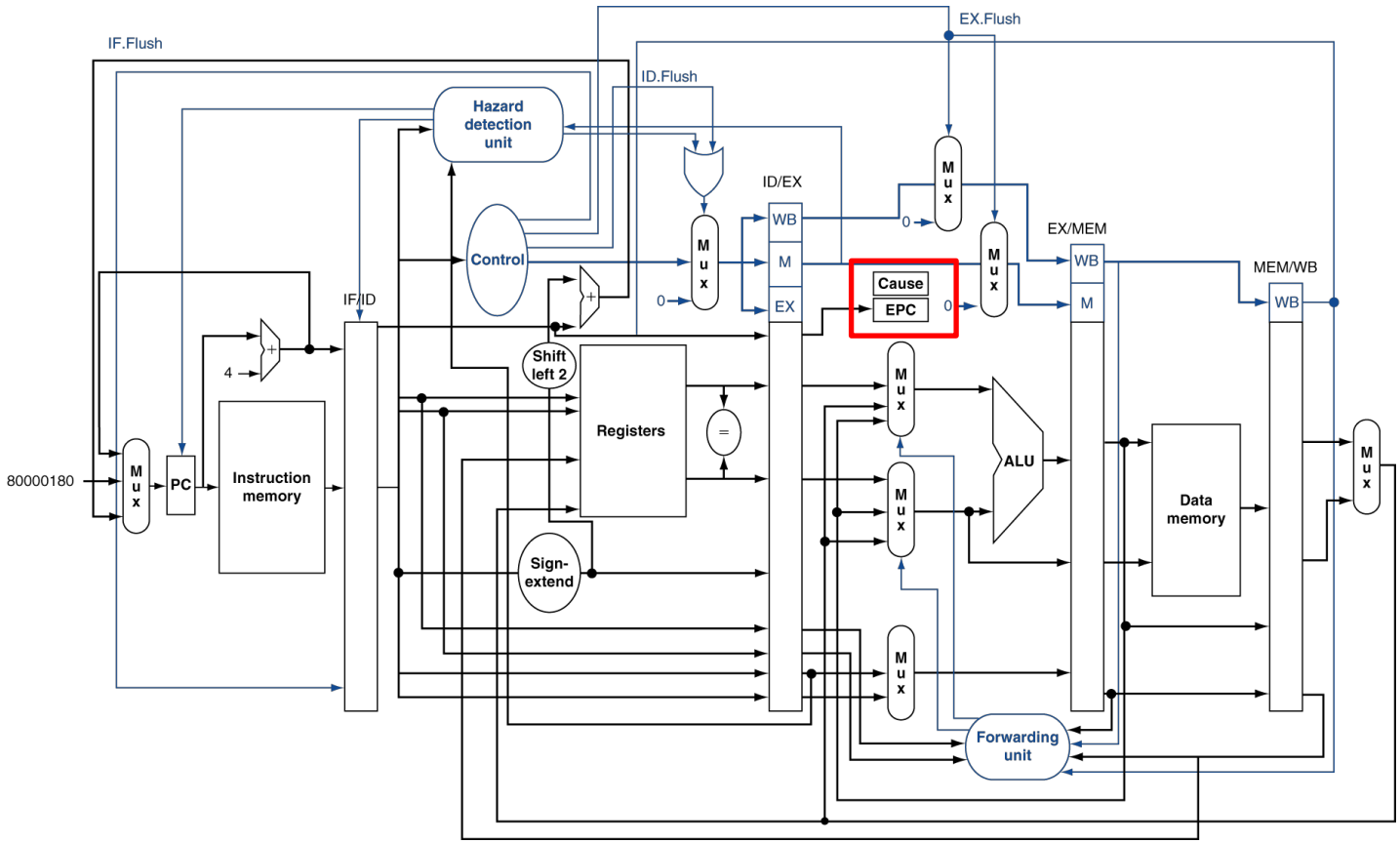


  - Interrupt status register (CR12):
    - **Enable/Disable interrupts + Set privilege modes**



  - Automatically set on interrupts
  - Also accessed by the “move from/to register-x”
    - `mfc0 $Rx, CRx (Rx ← CRx) / mtc0 $Rx, CRx (CRx ← Rx)`

Pipeline with Exceptions



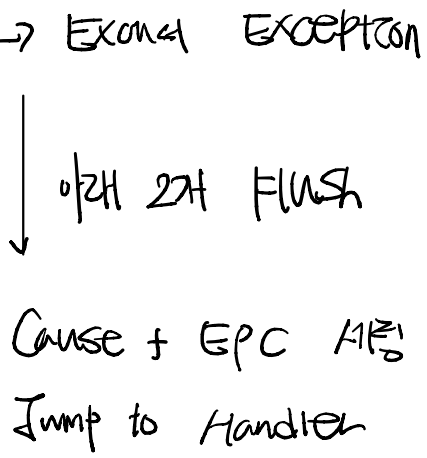
We need a flush mechanism till EX stage (to support exceptions at MEM stage)

→ WB에서 exception X, MEM 단계에서 Interrupt 발생했을 때, IF ~ EX에 있던 명령어 Flush
→ IF ~ EX는 Flush mechanism이 필요, 그러나 앞쪽 IF ~ EX의 명령어 Flush, 하고도 아님.

Exception example

- **Exception on add (overflow)**

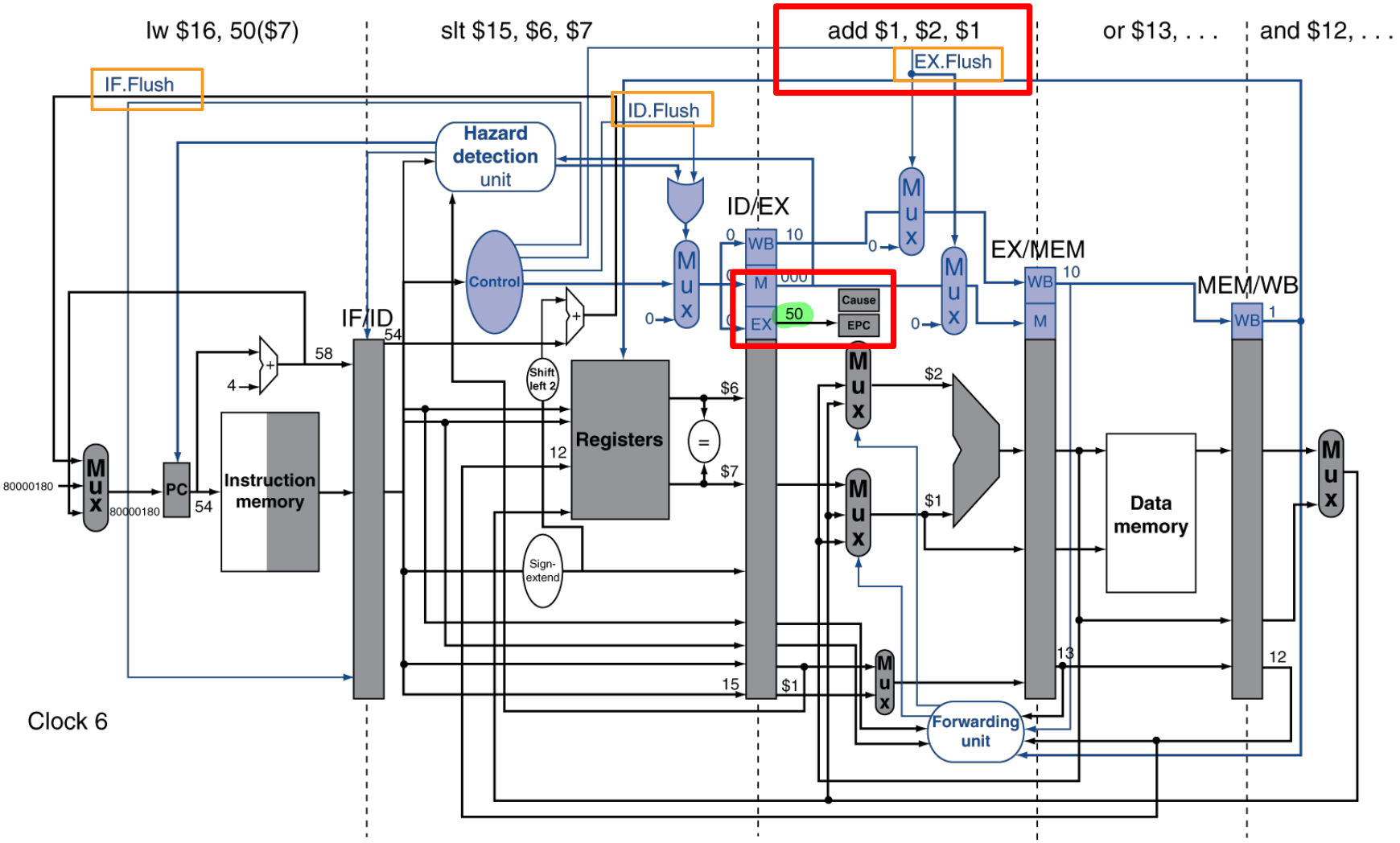
```
40    sub  $11, $2, $4
44    and  $12, $2, $5
48    or   $13, $2, $6
4C    add  $1,  $2, $1
50    slt  $15, $6, $7
54    lw   $16, 50($7)
…
```



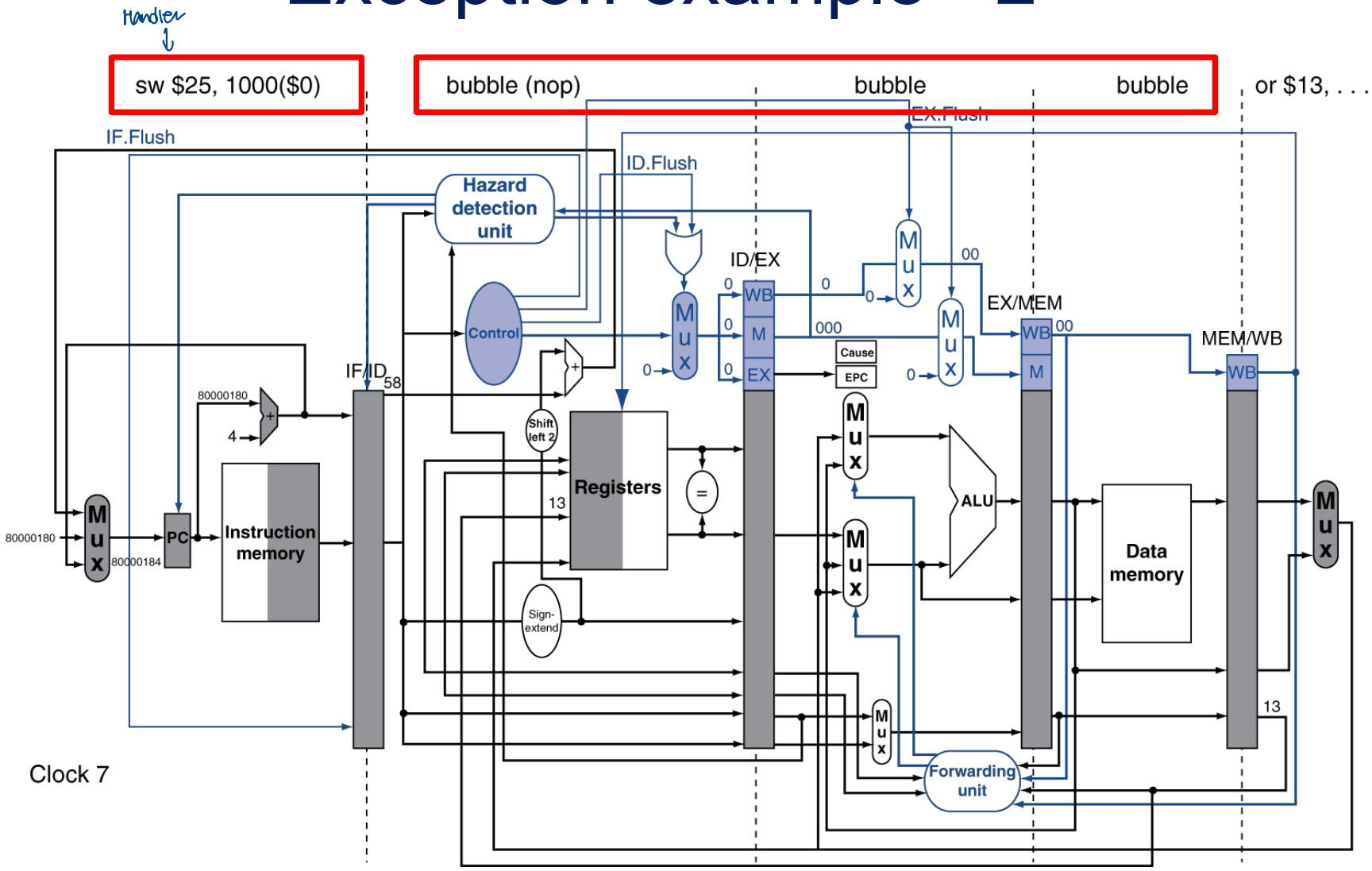
- **Handler**

```
80000180    sw   $25, 1000($0)
80000184    sw   $26, 1004($0)
…
```

Exception example - 1



Exception example - 2



Interrupt 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), 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

Registering interrupt handlers

- You can register custom interrupt handlers

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void my_handler(int sig) {
    printf("Caught signal %d: Handling FP exception!\n", sig);
    exit(EXIT_FAILURE); // Terminate the program
}

int main() {
    signal(SIGFPE, my_handler); // Register handler
    int x = 1 / 0; // This causes a divide-by-zero exception
    printf("This line will not execute.\n");
    return 0;
}
```

Returning from interrupts

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

(1) MIPS32 uses a special jump instruction

- ERET : MIPS 명령어
  - > To atomically restore the automatically saved CPU states
  - > To atomically restore the privilege level kernel to user
  - > 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 (control inst)
```

Another problem of delay slot

- What if the faulting address is in the delay slot?
  - Simply jumping back to EPC will cause error

```
beq $t0, $t1, target
lw $t2, $t3, 0($t4) // Branch Delay Slot (Cause exception)
…

target:
add $t5, t6, $t1    // This is what should be executed
```

- MIPS sets EPC = EPC – 4 to handle this type of errors
  - Beq is executed twice, but does not affect the functionality

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)

→ Nesting : EPC, Status register가 덮어씌워지는 것을.
→ Interrupt handler에서 exception, interrupt가 발생하면 안된다. ← MIPS도 막아버린다.

- However, for long-running handlers, interrupt must be enabled not to miss critical interrupts

→ Handler가 길면 중간에 Interrupt를 받도록 허용해줘야

  - 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 enable higher-priority interrupts
  - Higher-priority interrupt should not get delayed

Frequently enabling low-priority interrupts may critically affect the CPU performance (user-level instructions)

Short interrupt

```
_handler_shortest:
        // no prologue needed (Do not store registers)
        // use only r26 ($k0), r27 ($k1)

        … Short handler body …          // Use only r26, r27
                                        // Blocks other interrupts

        // epilogue
        mfc0 $r26, $epc                 // Set r26 to epc
        jr $r26                         // restore PC
        rfe                             // restore from exception mode
```

Longer interrupt (Not nested)

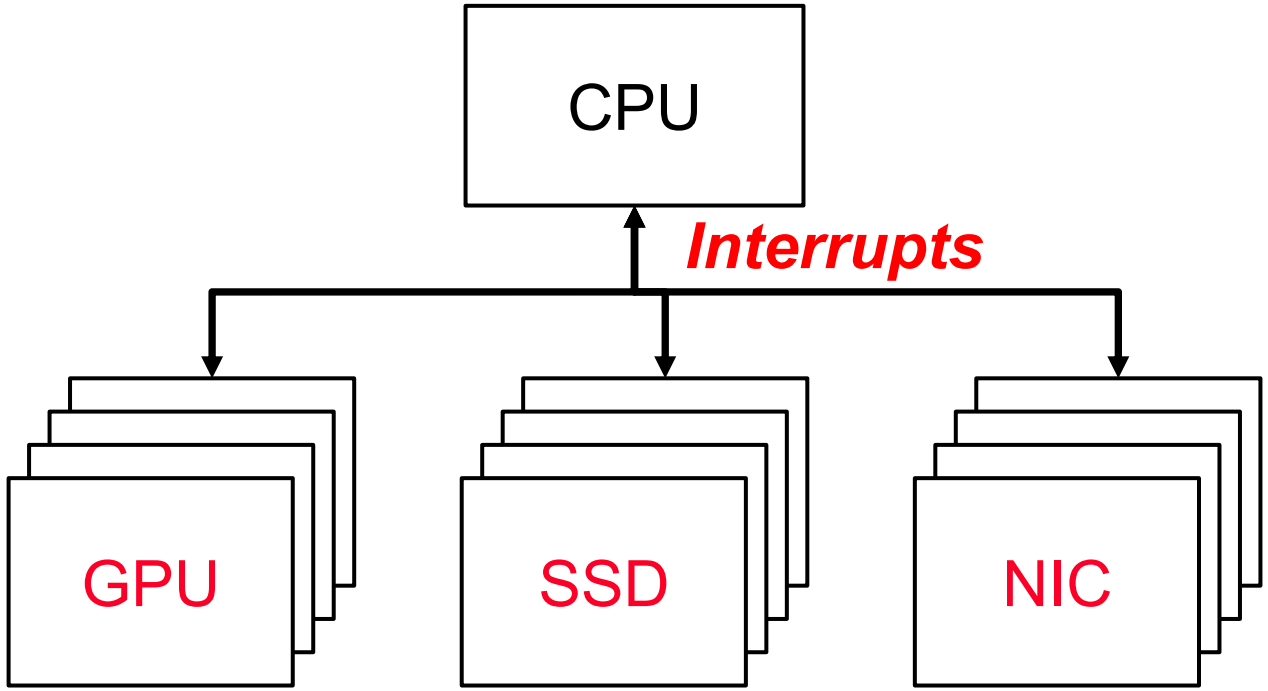
```
_handler_longer:
        // Prologue (save registers to use)
        // use r8/9 (callee-saved), r26 ($k0), r27 ($k1)
        addi $sp, $sp -8                    // Store r8/9
        sw $r8, 0($sp)
        sw $r9, 4($sp)
        … longer handler body …             // Use only r8/9, r26, r27
                                            // Blocks other interrupts
        // epilogue
        lw $r8, 0($sp)              // Restore r8/9
        lw $r9, 4($sp)
        addi $sp, $sp, 8
        mfc0 $r26, $epc                     // Set r26 to epc
        jr $r26                             // restore PC
        rfe                                 // restore from exception mode
```

Nestable handler

```
_handler_nestable:
        // Support nestable setup @ prologue
        addi $sp, $sp -4                    // Store EPC (to the stack)
        mfc0 $r26, $epc
        sw $r26, 0($sp)
        addi $r26, $r0, 0x405               // Enable nested interrupt
        mtc0 $r26, $status
        … longer handler body …             // These can be interrupted
                                            // Blocks other interrupts
        // epilogue
        addi $r26, $r0, 0x404               // disable nested interrupts
        mtc0 $r26, $status
        lw $r26, 0($sp)                     // Get EPC from the stack
        addi $sp, $sp, 8
        mfc0 r26,epc                        // Set r26 to epc
        jr r26                              // restore PC
        rfe                                 // restore from exception mode
```

Thinking of (external) interrupts

- The modern datacenter CPUs suffer from significantly large number of interrupts



Question?

Announcements:

Reading: *finish reading P&H Ch.4*

Handouts: *none*