National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Interrupt

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Interrupt
- Hardware Interrupt
- Interrupt Workflow
- Software Interrupt -- Exception
- Interrupt Vector
- Interrupt Descriptor Table
- Interrupt Stack Table

What is an interrupt?

- **What is an interrupt?**
  - An interrupt is a hardware signal from a device to the CPU
  - Tells the CPU that the device needs attention
  - CPU should stop performing what it is doing and respond to the device
- **Interrupt handler?**
  - Service the device and stop it from interrupting
- **What kinds of interrupts do we have?**
  - Hardware interrupt
  - Software interrupt

What is an interrupt?

- **What is the job of an interrupt handler?**
  - Save additional CPU context (written in assembly)
  - Process interrupt (communicate with I/O devices)
  - Invoke kernel scheduler
  - Restore CPU context and return (written in assembly)

What is an interrupt?

- **Synchronous interrupt**
  - Produced by the CPU control unit while executing instructions
  - The control unit issues interrupt only after terminating the execution of an instruction
- **Asynchronous interrupt**
  - Generated by other hardware devices at arbitrary times with respect to the CPU clock signals

What is an interrupt?

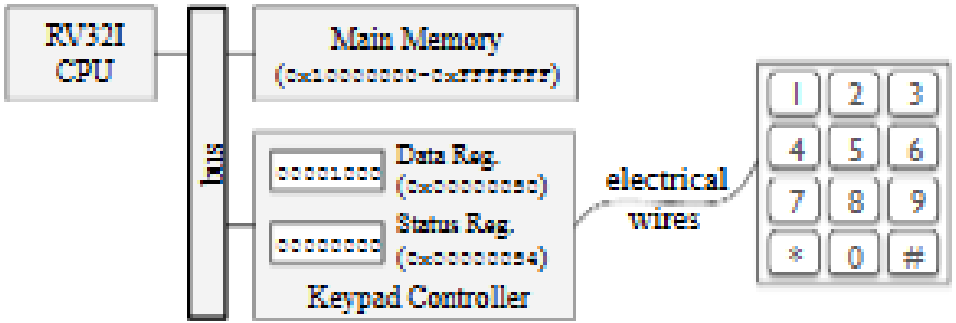
- **When an interrupt occurs …**
  - **Preempt current task**
    - The kernel must pause the execution of the current process
  - **Execute interrupt handler**
    - Search for the handler of the interrupt and transfer control
  - **After the interrupt handler completes execution**
    - The interrupted process can resume execution

Hardware Interrupt

- **Why do we need the hardware interrupt?**
  - Several devices connected to the CPU
    - E.g. keyboards, mouse, network card, etc.
  - These devices occasionally need to be serviced by the CPU
    - Tell the CPU that a key has been pressed
    - Interrupts can occur at any time
    - Need a way for the CPU to determine when a device needs attention

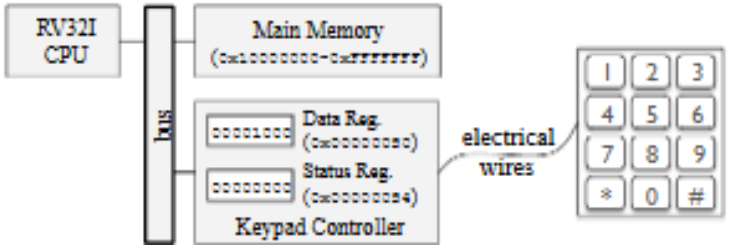
External Interrupt

- **When do we need interrupt?**
  - When the keypad is pressed
  - The keypad controller registers this information
  - This information requires the CPU attention and perform some actions
  - There must be a way to inform the CPU that peripheral needs its attention

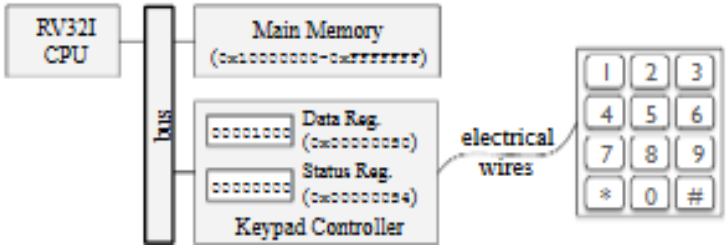


External Interrupt

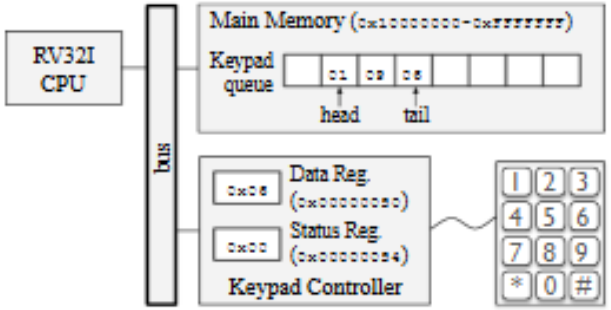
- The keypad controller contains two 8-bit register
  - The data register
    - Indicates the last key pressed on the keypad
  - The status register
    - Indicates the keypad's current status
    - Its LSB bit is the REDAY bit that indicates whether the keypad was pressed since the last time the CPU read a value from the data register



External Interrupt



- The keypad controller contains two 8-bit register
  - The status register
    - Its second LSB bit is called the OVRN bit that indicates whether the keypad was passed more than once
  - Data overrun (OVRN)
    - The keypad controller has only one data register
    - If keypad is pressed more than once before the CPU gets the chance to read the data register, one or more key values are lost
    - How to resolve this OVRN problem?



External Interrupt

- To prevent data overruns
  - Copy the data register value to a FIFO queue located at the main memory as soon as the keypad is pressed
  - FIFO queue is an 8-element circular buffer and two pointers
    - Pointers points to queue's head and tail
    - Whenever a key is pressed, its value is pushed into the queue's tail
    - CPU must execute this routine as soon as possible to prevent data overruns

External Interrupt

- There are two main methods to direct the CPU handle events caused by external hardware
  - Polling
  - Hardware interrupt

Polling

- **Polling**
  - The CPU periodically checks whether peripherals need attention
  - Assuming the peripheral program is designed
  - Whenever there is a peripheral that needs attention, the program invokes a routine to handle the peripheral

Algorithm 3: Handling peripherals with polling.

```
1 while True do
      /* Handle peripherals */
2     for p in Peripherals do
3         if needsAttention(p) then
4             handlePeripheral(p) ;
5         end
6     end
7     PerformSomeComputation();
8 end
```

Polling

- **Polling**
  - keypadPressed()
    - checks if the keypad READY bit is set
  - getKey():
    - Read the contents of the data register
  - pushKeyOnQueue()
    - Push it to the queue's tail
  - Compute ()
    - Represents the work that is done by the program

Algorithm 4: Handling the keypad with polling.

```
1 while True do
2     if keypadPressed() then
3         k ← getKey() ;
4         pushKeyOnQueue(k) ;
5     end
6     Compute() ;
7 end
```

Polling

- **Polling**
  - The Compute() affects the frequency in which the keypad is checked
  - The longer Compute() -> increase the occurring data overrun
  - Could we directly remove Compute()?
  - Polling is usually not the best approach to check for and handle peripheral events, why?

Algorithm 4: Handling the keypad with polling.

```
1 while True do
2     if keypadPressed() then
3         k ← getKey() ;
4         pushKeyOnQueue(k) ;
5     end
6     Compute() ;
7 end
```

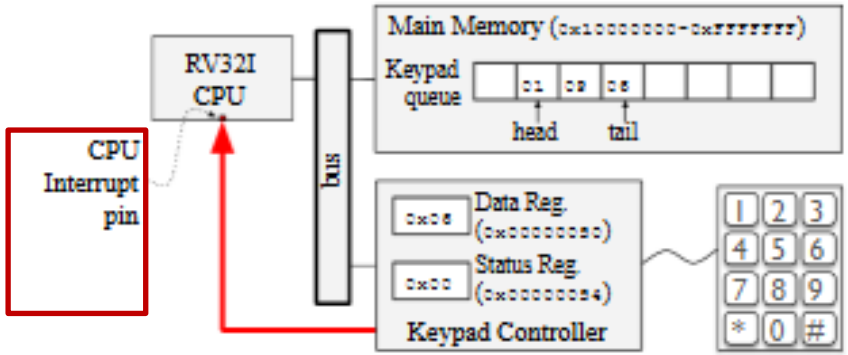
Hardware Interrupt

- **Hardware interrupt**
  - Allows hardware to inform the CPU they require attention
  - Caused by external (non-CPU) hardware such as peripherals to inform the CPU they require attention
  - The peripheral sends an interrupt to the CPU

Hardware Interrupt

- **Hardware interrupt**
  - Once the CPU receives this signal
    - The CPU saves the context of the current program
    - Invokes a routine to handle the hardware interrupt
    - Restores the context of the saved program and continues executing

Hardware Interrupt



- **Hardware interrupt**
  - The CPU contains an interrupt pin and the keypad controller is connected to the CPU interrupt pin
  - The interrupt pin as an input pin
    - Informs the CPU whether or not there is an external interrupt
    - The CPU constantly monitors the interrupt pin and interrupts the current execution flow to execute and ISR
    - The interrupt service routine (ISR) is a software routine that handle the interrupt

Hardware Interrupt

- **Detecting external interrupt**
  - Before fetching an instruction for execution
  - The CPU verifies if the interrupt.pin is set
  - If the CPU receives an interrupt signal & interrupts are enabled
    - The CPU saves the program counter (PC) into the SAVED_PC register
    - Set the PC register with (ISR_ADDRESS) and disable interrupts by clearing the interrupt.enabled register
    - Then, the next instruction (ISR) will be fetched

Hardware Interrupt

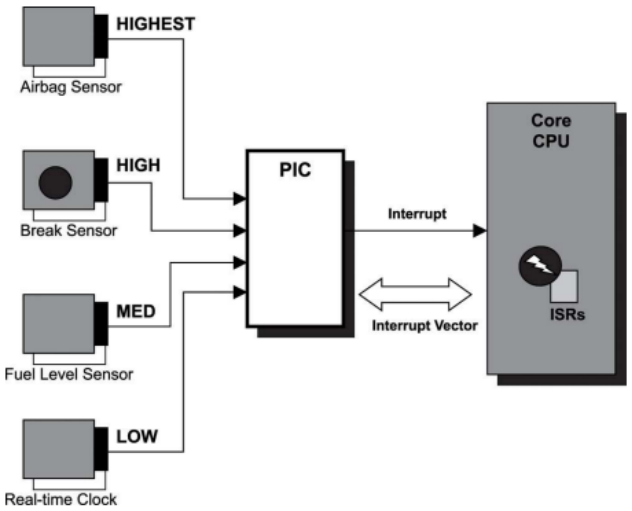
- **Invoking the proper ISR**
  - Each peripheral usually requires a specialized routine to handle its interrupts
  - SW-only design, the ISR is responsible for
    - Identifying which peripheral interrupted the CPU
    - Invoking the proper routine to handle the interrupt
    - Upon an interrupt
      - The CPU invokes a generic ISR doing above tasks
      - The ISR may have to interact with all peripherals to find out which one is requiring the CPU attention

Hardware Interrupt

- **Invoking the proper ISR**
  - SW-only design pros & cons
    - Simplifies the CPU hardware design
    - The ISR may take a long time trying to figure out which peripheral interrupted the CPU

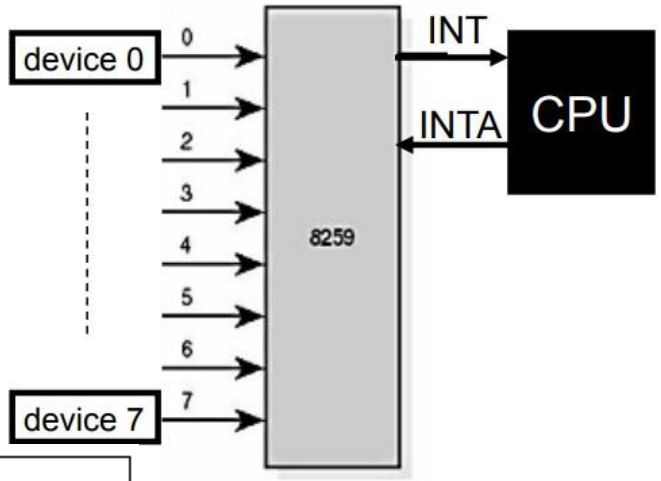
Hardware Interrupt

- **Programmable Interrupt Controller (PIC)**
  - Responsible for sequential multiple interrupt requests from devices
  - Advanced PIC (APIC)
    - Local APIC
      - Located on each CPU core
      - Handle interrupts from APIC-timer, thermal sensor
    - I/O APIC
      - Distributed external interrupts among the CPU cores



Hardware Interrupt

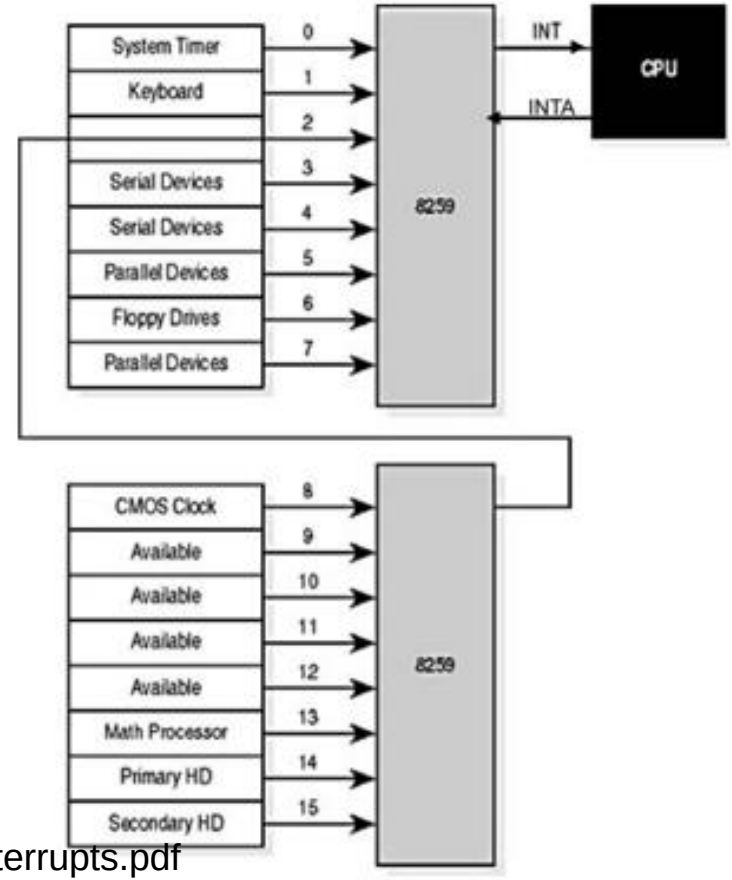
- 8259 PIC relays up to 8 interrupts to the CPU
  - Devices raise interrupts by an ‘interrupt request’ (IRQ)
  - CPU acknowledges and queries the 8259 to determine which device interrupted
  - Priorities can be assigned to each IRQ line
  - 8259s can be cascaded to support more interrupts
    - Two PICs and cascade buffer
    - IRQ2 -> IRQ9



INTA is a signal used to identify that a CPU has an interrupt made by the interrupt controller.

Hardware Interrupt

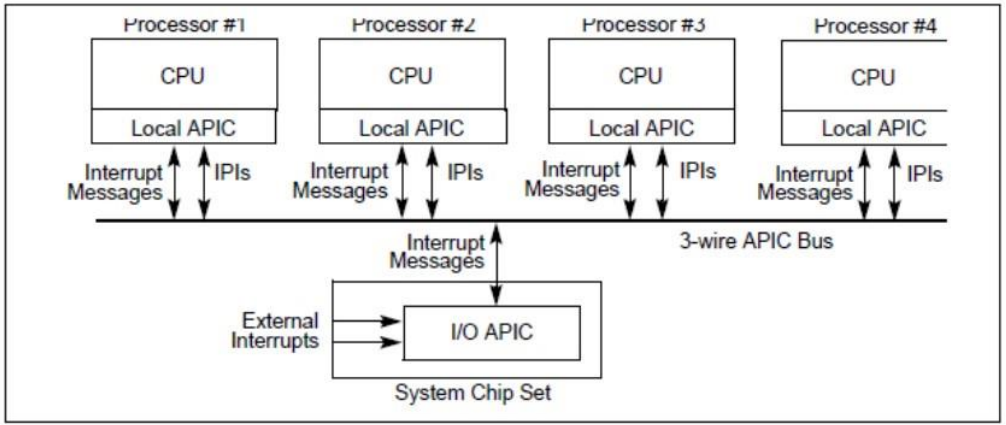
- IRQ 0 to IRQ 15, 15 possible devices
- Interrupt types
  - Edge
  - Level
- Limitations
  - Limited IRQs
  - Multi-processor support limited



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/6_Interrupts.pdf

Hardware Interrupt

- **Advanced PIC (APIC)**
  - External interrupts are routed from peripherals to CPUs in multi-processor systems through APIC
  - APIC distributes and prioritizes interrupts to processors
  - APICs communicates through a special 3-wire APIC bus



Hardware Interrupt

- **LAPIC**
  - Receives interrupts from I/O APIC and routes it to the local CPU
  - Can also receive local interrupts such as thermal sensors, internal timers, etc.
  - Send and receive IPIs (Inter-processor interrupts)
    - IPIs are used to distribute interrupts between processors or execute system-wide functions like booting, load distribution, etc.
- **I/O APIC**
  - Present in the chipset (northbridge)
  - Used to route external interrupts to local APIC

Takeaway Questions

- Who can issue an interrupt?
  - (A) Network card
  - (B) GPU
  - (C) Keyboard
- How do peripherals tell the CPU to know their requests?
  - (A) External interrupt
  - (B) Polling
  - (C) Exception

Interrupt Vector (1/3)

- **Interrupt vector**
  - The processor uses a unique number for recognizing the type of interruption or exception
  - Each interrupt/exception provided a number
  - Number used to index into an **interrupt descriptor table (IDT)**
  - IDT provides the entry point into an interrupt/exception handler
  - 0 to 255 vectors possible

Interrupt Vector (2/3)

- **Interrupt vector**
  - **0 to 31** correspond to **exception and nonmaskable interrupts (NMI,** handle non-recoverable error**)**
  - **32 – 47** are assigned to **maskable interrupts caused by IRQs**
  - **48 – 255** may be used to identify **software interrupts**
  - For example, Linux uses a 128 (0x80) vector that is used to make system calls to the kernel by other programs.
  - When a process in user mode executes int 0x80 assembly instruction, the CPU switches into kernel mode and starts executing the system_call() kernel function

Interrupt Vector (3/3)

- **Processor generates exception**

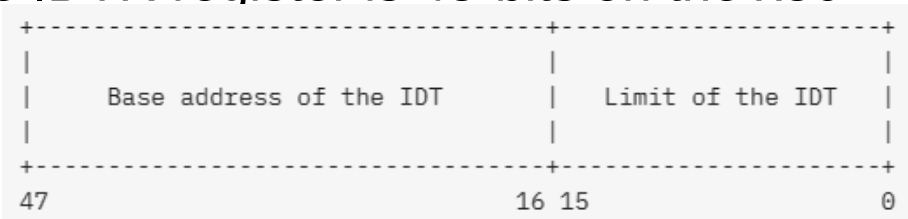
INT_NUM	Short description
0x00	Division by zero
0x01	Single-step interrupt
0x02	NMI
0x03	Breakpoint
0x04	Overflow
0x05	Bound range exceeded
0x06	Invalid opcode

Interrupt Descriptor Table (IDT)

- **Interrupt descriptor table**
  - Stores entry points of the interrupts and exceptions handlers
  - The IDT entries are called gates
    - Interrupt gates
    - Task gates
    - Trap gates
  - The IDT is an array of 8-byte gates (256 entries) on x86 and 16-byte gates on x86_64
  - Loaded the IDT with the null gates while transitioning into protected mode

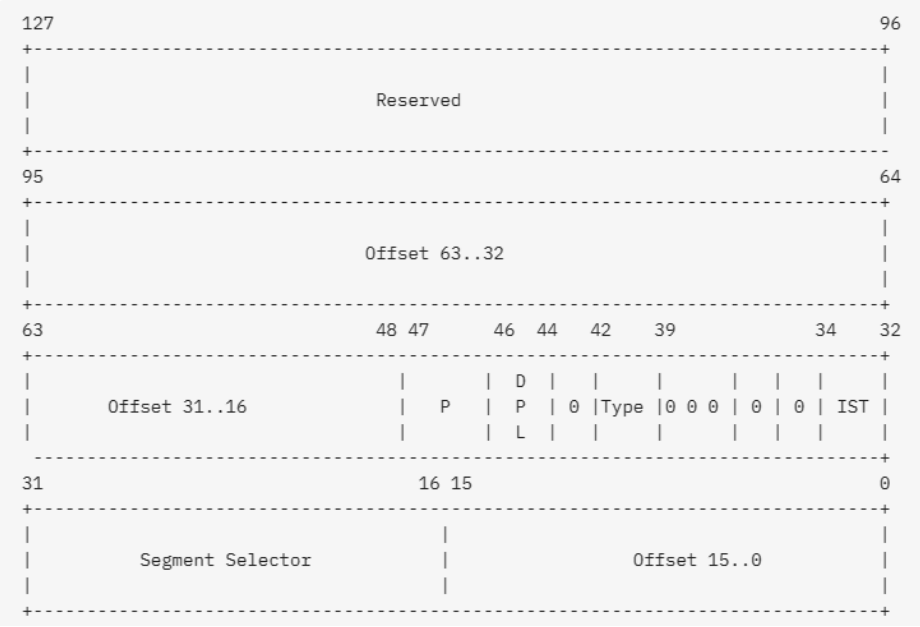
Interrupt Descriptor Table (IDT)

- **Interrupt descriptor table**
  - Can be located anywhere in the linear address space
  - The base address of it must be aligned on an 8-byte boundary on x86, a 16-byte boundary on x86_64
  - The base address of IDT is store in IDTR register
    - LIDT/SIDT instruction to read/write IDTR register
    - The IDTR register is 48-bits on the x86



Interrupt Descriptor Table (IDT)

- **The IDT entries** (16 bytes on x86_64)
  - **0-15 bits** as the base address of entry point of the interrupt handler
  - **16-31 bits** as the base address of the segment selector
  - DPL (Descriptor Privilege Level)



Interrupt Stack Table (IST)

- **Interrupt stack table**
  - New mechanism in x86_64
  - An alternative to legacy stack-switch mechanism
  - Unconditionally switches stacks when it is enabled and can be enabled for any interrupt
  - Seven IST pointers in the task state segment (TSS)
    - TSS contains information about a process
    - TSS is used for stack switching during the execution of an interrupt or exception handler

Interrupt Stack Table (IST)

- **Stack switching**
  - If the interrupt occurs when running in the user mode
    - The process switches from user stack to kernel stack
    - Then, switching to the interrupt stack
  - How to switch stack?
    - CPU should know the location of the new stack segment (SS) and ESP register
    - Done by task segment descriptor

Interrupt Stack Table (IST)

- **Task state segment (TSS)**
  - TSS is used to find the new stack
  - TSS resides in the memory
    - Processor register states -> used for task switching
    - I/O port permission bitmap -> specifies individual ports to accessible program
    - Inner-level stack pointer -> specifies the new stack pointer when a privilege level change occurs
    - Previous TSS link

Interrupts on RV32 I

- **Control and Status Registers (CSR)**
  - Special registers that expose the CPU status to the software
  - Allow the software to configure the CPU behavior
  - In RV32I ISA
    - `mstatus CSR` is a 32-bit register that exposes the current status of the CPU
    - `csrrw rd, csr, rs1` instruction
      - Atomically swaps the contents of register a0 and CSR
    - `csrr a0, mstatus` instruction
      - Copies the contents of the mstatus CSR into a0

Interrupts on RV32 I

- **Interrupt related CSR registers**
  - `mstatus:`
    - Provides information or control the interrupt handling
  - `mcause` (Machine Interrupt Cause):
    - Store the interrupt cause (a value that identifies whey an interrupt was generated)
  - `mtvec` (Machine Trap Vector)
    - Stores information that allows the CPU to identify the proper ISR

Interrupts on RV32 I

- **Interrupt related CSR registers**
  - `mip` (Machine Interrupt Pending)
    - Interrupts are pending (i.e., have been signaled but not handled by the CPU yet)
  - `mepc` (Machine Exception Program Counter):
    - The CPU saves the contents of the PC register into mpec
  - `mscratch` (Machine Scratch)
    - Is visible in machine mode

Interrupt Handling Flow

- **How does the CPU handle external interrupts**
  - 1. Check mstatus.MIE to verify if the CPU accept interrupts
  - 2. if mstatus.MIE = 1, the CPU first saves mstatus.MIE to mstatus.MPIE, then clear it so that new interrupts are ignored
  - 3. The CPU saves PC into mepc CSR and set mcause CSR
  - 4. ISR changes the PC register to point to the first instruction of the ISR

Interrupt Handling Flow

- **Implementing an ISR**

```
1  main_isr:
2    # Save the context
3    csrrw sp, mscratch, sp  # exchange sp with mscratch
4    addi  sp, sp, -64       # allocate space at the ISR stack
5    sw a0, 0(sp)            # save a0
6    sw a1, 4(sp)            # save a1
7    ...
8
9    # Handles the interrupt
10   csrr  a1, mcause        # read the interrupt cause and perform
11   ...                     # some action according to the cause.
12
13   # Restore the context
14   ...
15   lw a1, 4(sp)            # restore a1
16   lw a0, 0(sp)            # restore a0
17   addi  sp, sp, 64        # deallocate space from the ISR stack
18   csrrw sp, mscratch, sp # exchange sp with mscratch
19   mret                    # return from the interrupt
```

Point to the top of the previous program stack

ISR allocates space on the ISR stack

Identify the interrupt source by mcause CSR

Loading register's value from ISR stack

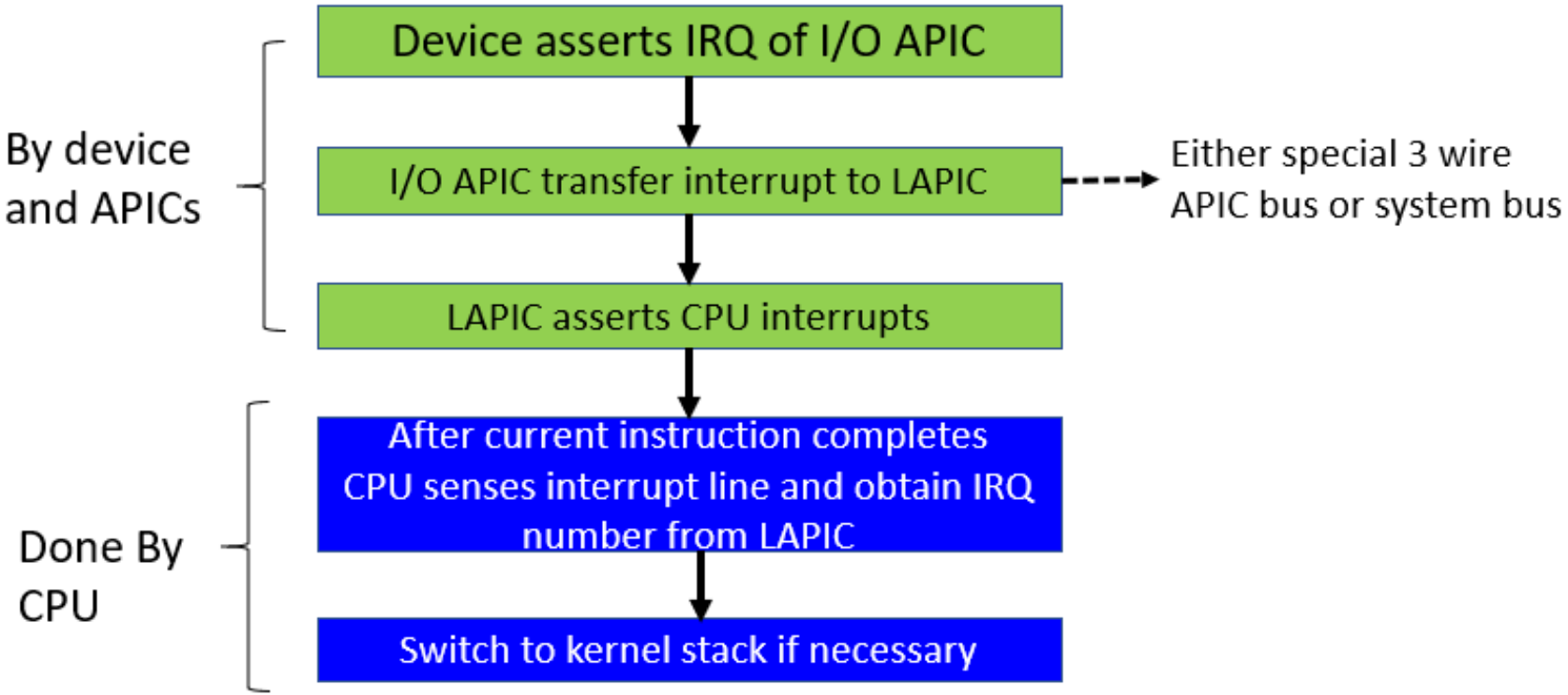
Interrupt Handling Flow

- **Enabling interrupt**
  - Once peripherals that generate interrupt signals are configured
  - The ISR and IST stack are set

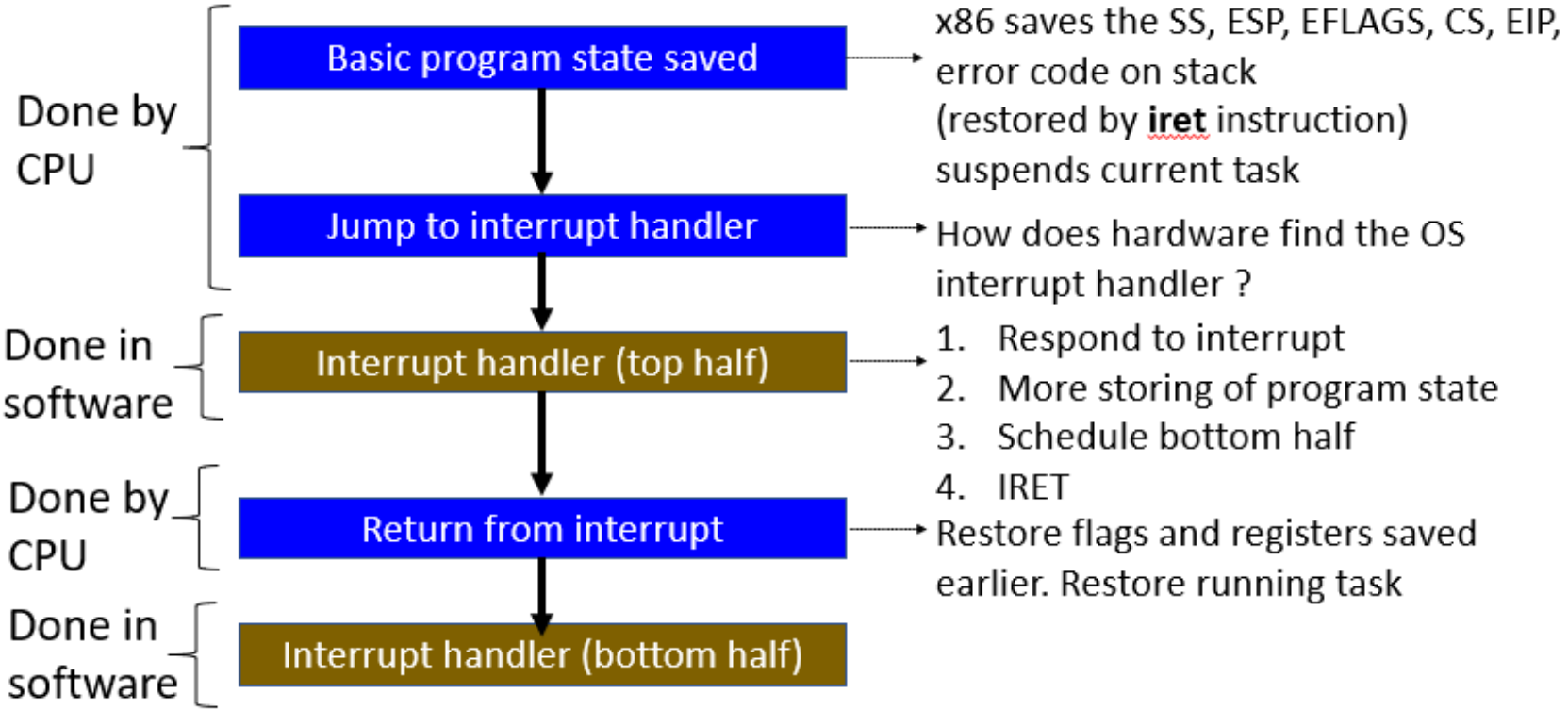
```
# Enable external interrupts (mie.MEIE <= 1)
csrr t0, mie        # read the mie register
li t2, 0x800        # set the MEIE field (bit 11)
or t0, t0, t2
csrw mie, t0        # update the mie register

# Enable global interrupts (mstatus.MIE <= 1)
csrr t0, mstatus    # read the mstatus register
ori t0, t0, 0x8     # set MIE field (bit 3)
csrw mstatus, t0    # update the mstatus register
```

Enable mie.MEIE and mstatus.MIE to allow CPU to handle external interrupts

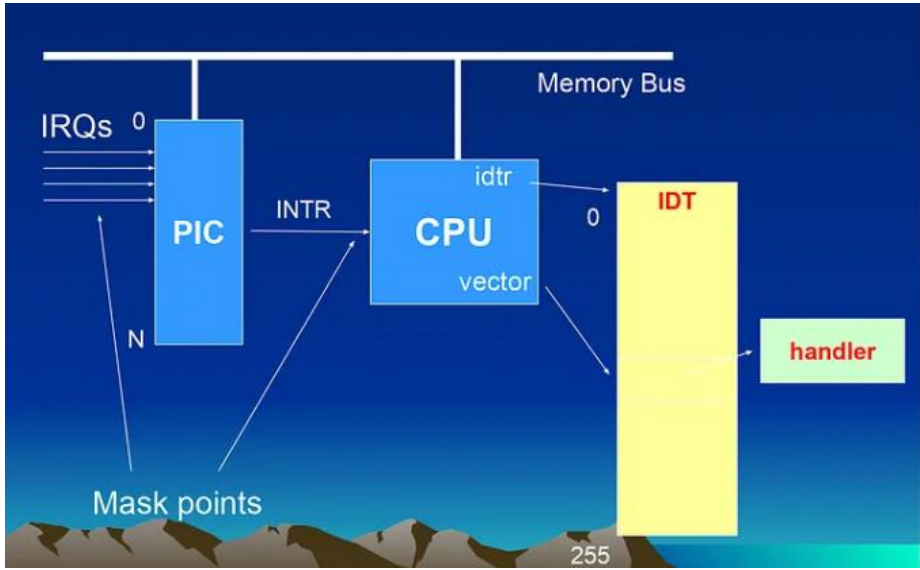
Interrupt Workflow



Interrupt Workflow



Interrupt Workflow



- Processing Interrupt
  - Device creates IRQ
  - PIC collects IRQs
  - PIC prioritizes IRQs
  - PIC issues interrupt to CPU
  - CPU saves interrupt states
  - CPU asks PIC interrupt number
  - CPU uses an interrupt vector number as an index to find IDT entry
  - ISR saves states in registers
  - Executing ISR
  - After completing, pass the EOI (End of Interrupt) command
  - Resume registers/iret command

ISA Privilege Level

- How to protect system from faulty or malicious user programs?
- Privilege level
  - Define which ISA resources are accessible by the software
  - U: User/Application
  - S: Supervisor
  - M: Machine

Exceptions

- Exceptions
  - Events generated by the CPU in response to exceptional conditions when executing instructions
  - Exceptions usually trigger an exception handling
    - The exceptional condition may be dealt with before the CPU continue executing the program
  - Causes the CPU to redirect the execution flow to a system routine

Exceptions

- Exceptions
  - Causes the CPU to redirect the execution flow to a system routine
    - Save the current program context
    - Handle the exceptional condition
    - Restore the context of the saved program to continue the exception

Exceptions

- Exceptions
  - The exception handling flow is similar to the hardware interrupt
  - RISC-V CPU uses the same mechanism to handle both interrupt and exceptions
  - The exception handling protects the system from illegal user code operations
    - The hardware is configured by system software
    - Generate exceptions in case the privilege mode is set as User/Application
    - The ISR may decide what to do with the user program

Software Interrupts

- Software interrupts
  - Events generated by the CPU when it executes special instructions
    - The environment call (ecall) and breakpoint (break) insns
    - Synchronous events that occur due to executing an insn

- Comparing to exception
  - Exception are only generated on exceptional conditions
  - Software interrupts are always generated when the CPU executes special instructions

Software Interrupt

- **Exception**
  - Caused by an exceptional condition in the processor itself
  - An example of an exceptional condition is division by zero
  - Exiting a program with syscall instruction
- Categories
  - **Faults**: an exception reported before the execution of a “faulty” instruction
  - **Traps**: an exception reported by the trap instruction
  - **Aborts**: an exception doesn’t always report the exact instruction which caused the exception

Summary

- Interrupt changes the sequence of instruction execution
- Exception occurs since the illegal operation
- Hardware interrupt – programmable interrupt controller
- Interrupt vector records interrupt commands

Takeaway Questions

- Which situations will raise the exception?
  - (A) External interrupt
  - (B) Software interrupt
  - (C) Divided by zero
- What are purposes of interrupt vector?
  - (A) Recognizing the type of interruption or exception
  - (B) Improve the performance of the CPU
  - (C) Raise the privilege of executed instructions