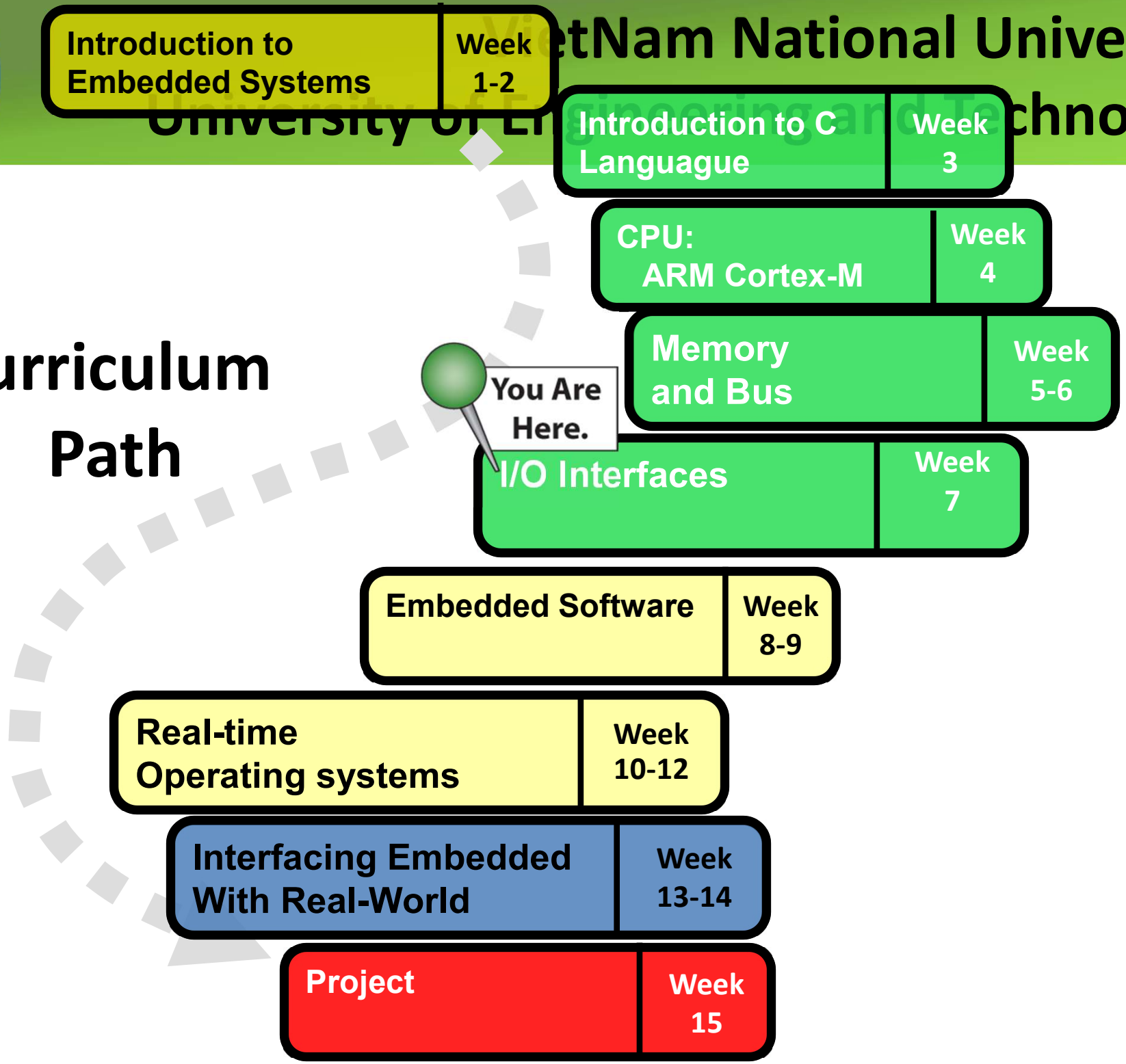
VietNam National University
University of Engineering and Technology

PROGRAMMING EMBEDDED AND REAL-TIME SYSTEMS
(INT3108, LẬP TRÌNH NHÚNG VÀ THỜI GIAN THỰC)

Dr. Nguyễn Kiêm Hùng
Email: kiemhung@vnu.edu.vn

Curriculum Path



Objectives

In this lecture you will be introduced to:

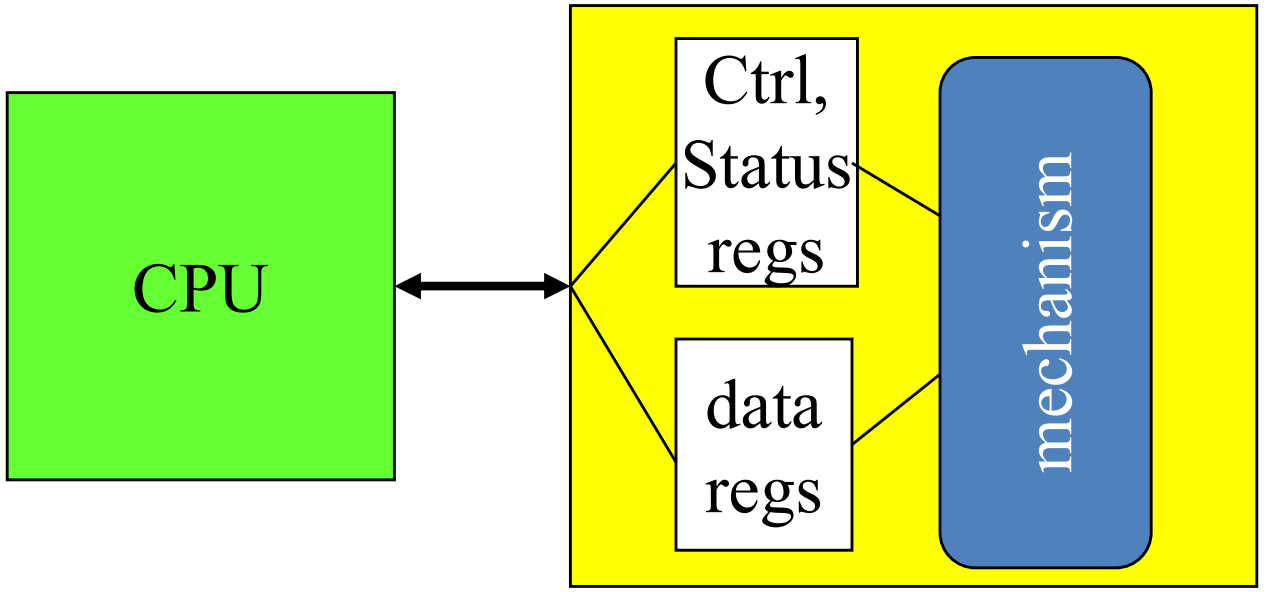
- Input and output mechanisms: Polling and Interrupt
- Interrupt Concepts and Operation

Outline

- **Input and Output Programming**
- I/O Mechanism: Polling vs. Interrupt
- Interrupt Mechanism
- Summary

I/O devices

- Usually includes some **analog or nonelectronic** component.
- Typical digital interface to CPU:
  - Includes data, status, and control registers accessed by CPU.
  - **Some registers may be read-only, while others may be readable or writable**



I/O Addressing: memory-mapped I/O and standard I/O

- Processor talks to both memory and peripherals using same bus – two ways to talk to peripherals
  - Memory-mapped I/O
    - Peripheral registers occupy addresses in same address space as memory
    - e.g., Bus has 16-bit address
      - lower 32K addresses may correspond to memory
      - upper 32K addresses may correspond to peripherals
  - Standard I/O (I/O-mapped I/O)
    - Additional pin (*M/IO*) on bus indicates whether a memory or peripheral access
    - e.g., Bus has 16-bit address
      - all 64K addresses correspond to memory when *M/IO* set to 0
      - all 64K addresses correspond to peripherals when *M/IO* set to 1

I/O Programming

- **Two types of instructions can support I/O:**
  - special-purpose I/O instructions;
  - memory-mapped load/store instructions.
- **Intel x86 provides in, out instructions.**
  - provide a separate address space for I/O devices
- **Most other CPUs (e.g. ARM processors) use memory-mapped I/O:**
  - Memory-mapped I/O enables software to view these registers as locations in memory.
  - These registers can be accessed using only **Load** and **Store** instructions
- **I/O instructions do not preclude memory-mapped I/O.**

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code:

```
LDR r1,#DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
MOV r0,#8 ; set up value to write
STR r0,[r1] ; write value to device
```

I/O Programming

- **A peripheral requires an initialization process before it can be used**
- **Initialization process includes:**
  - Programming the clock control circuitry
  - Configuring the operation mode of the I/O pins
    - multiplexed I/O pins that can be used for multiple purposes be necessary to configure to the expected modes (input/output direction, function, etc.) electrical characteristics (e.g. voltage, pull up/down, open drain, etc.).
  - Peripheral configuration
  - Interrupt configuration

Outline

- Input and Output Programming
- **I/O Mechanism: Polling vs. Interrupt**
- Interrupt Mechanism
- Summary

Polling vs. Interrupt

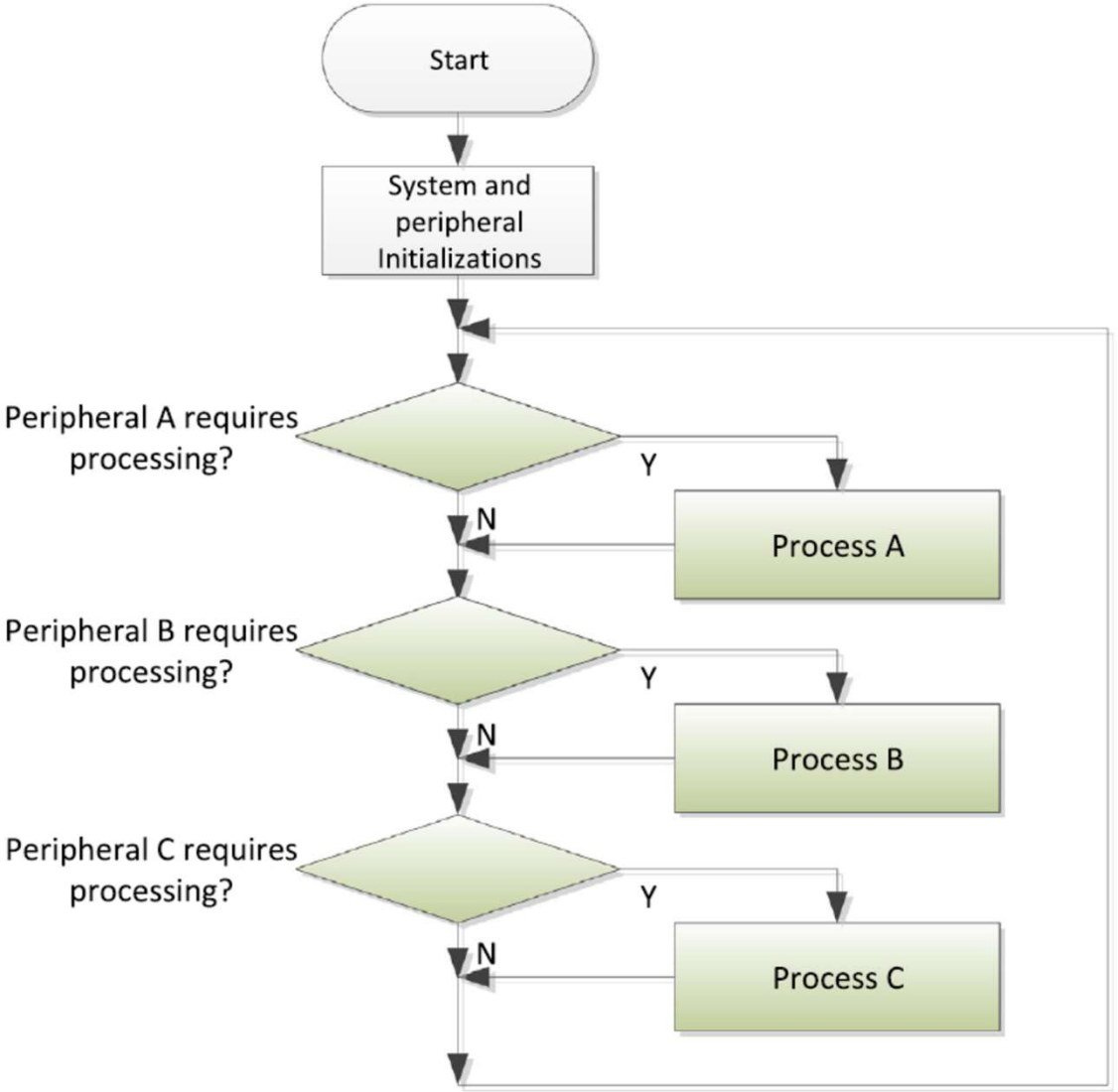
- Suppose a peripheral intermittently receives data, which must be serviced by the processor
  - The processor can *poll* the peripheral regularly to see if data has arrived – wasteful (S/W solution)
  - The peripheral can *interrupt* the processor when it has data
- Int (H/W solution) requires an extra pin or pins: If Int is 1, processor suspends current program, jumps to an Interrupt Service Routine (ISR)
  - Known as interrupt-driven I/O
  - Essentially, "polling" of the interrupt pin is built-into the hardware, so no extra time!

Polling

Polling

- Continuously checking the status of a peripheral; e.g. read data from an input keyboard.

- Polling is relatively straightforward in design and programming with the sacrifice of system performance.



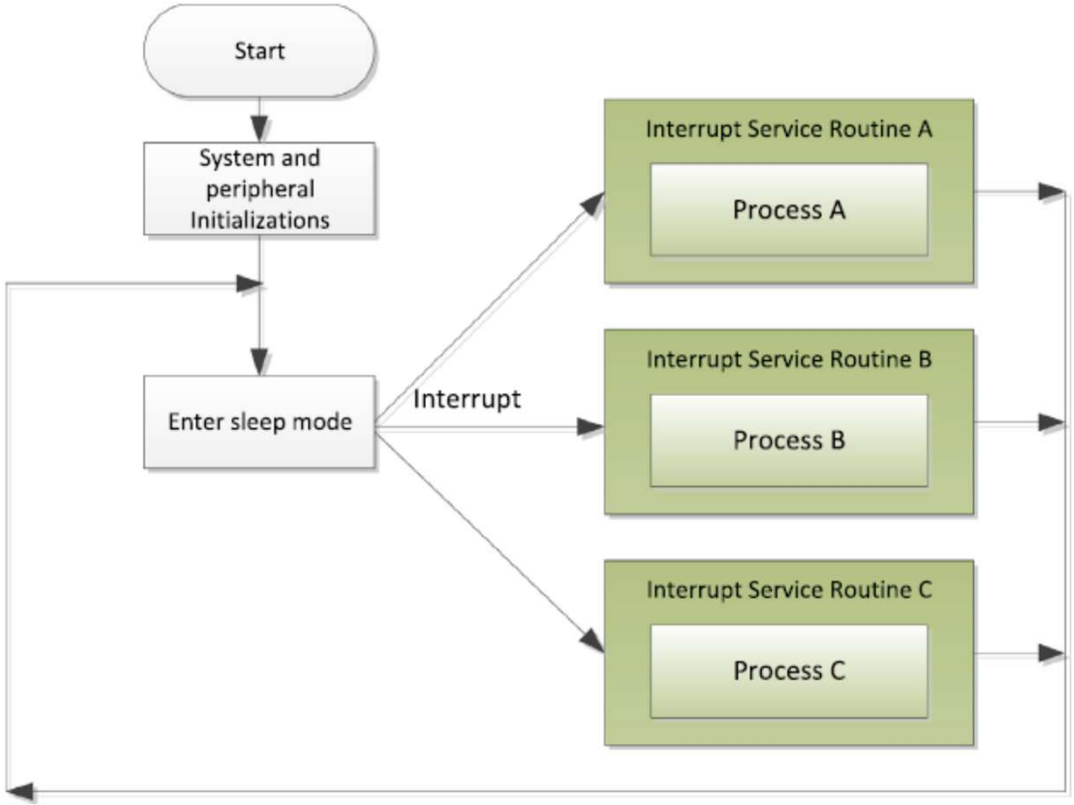
Interrupt

- Polling is very inefficient.
  - CPU can't do other work while testing device, therefore **hard to do simultaneous I/O**.
  - **it is difficult to define priorities between different services using polling**
  - **it is not energy efficient**
- Interrupts allow a device to change the flow of control in the CPU.
  - Causes subroutine call to handle device (**interrupt handler**).

Interrupt

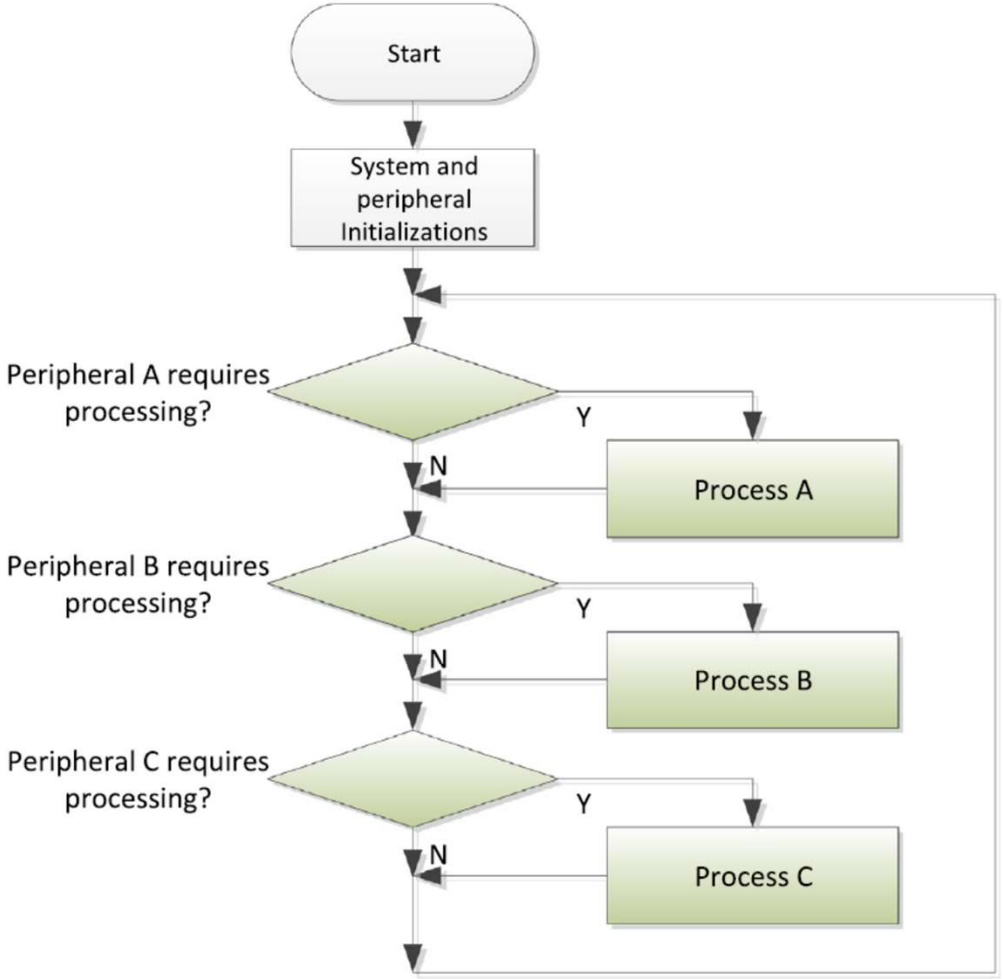
Interrupt Process

- Device "interrupts" CPU to indicate that it needs service. (*These events only occur if the interrupt is enabled.*)
- CPU waits until the current instruction has finished being executed.
- Save the contents of internal registers of the CPU & the state information within Control Unit
- The PC is loaded with address of the Interrupt Service Routine (ISR)
- ISR is executed.
- CPU returns to where it left off in the main program.

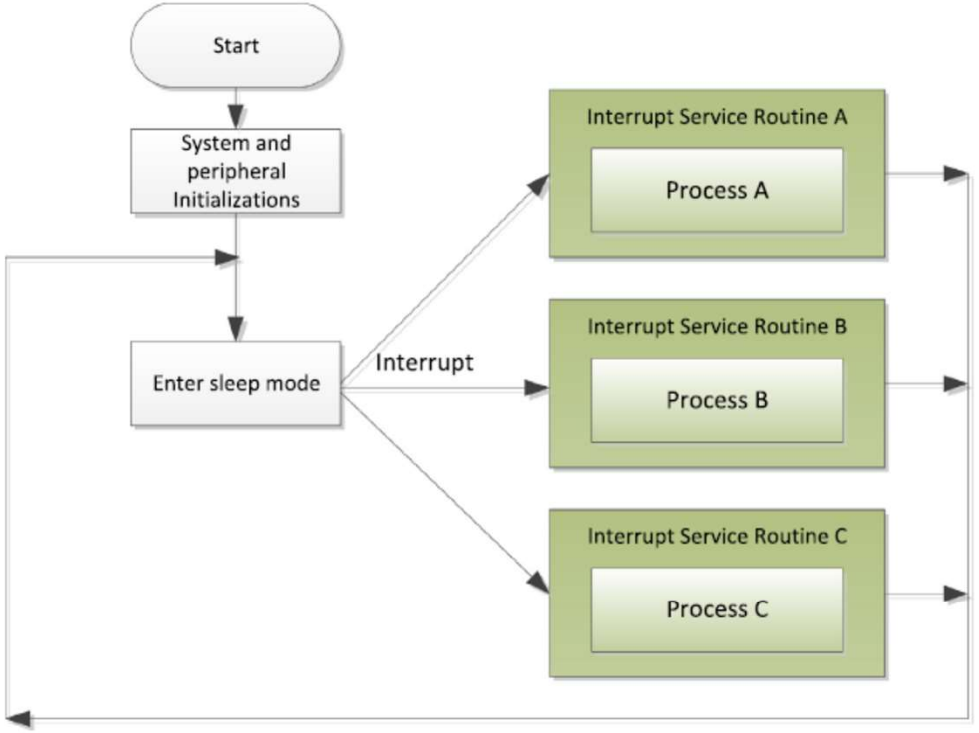


Polling vs. Interrupt

Polling



Interrupt



Outline

- Programming Input and Output
- Polling vs. Interrupt
- **Interrupt Mechanism**
- Summary

Interrupt

Interrupt Handler Features:

- Differs from subroutine because it is executed at any time due to interrupt, not due to Call

- Should be implemented as small as possible

- Should be executed in short-time.

Debugging interrupt code

- What if you forget to save registers?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Foreground Code:

***y* = *Ax* + *b*:**

```
for (i = 0; i < M; i++) {
y[i] = b[i];
for (j = 0; j < N; j++)
y[i] = y[i] + A[i,j]*x[j];
}
```

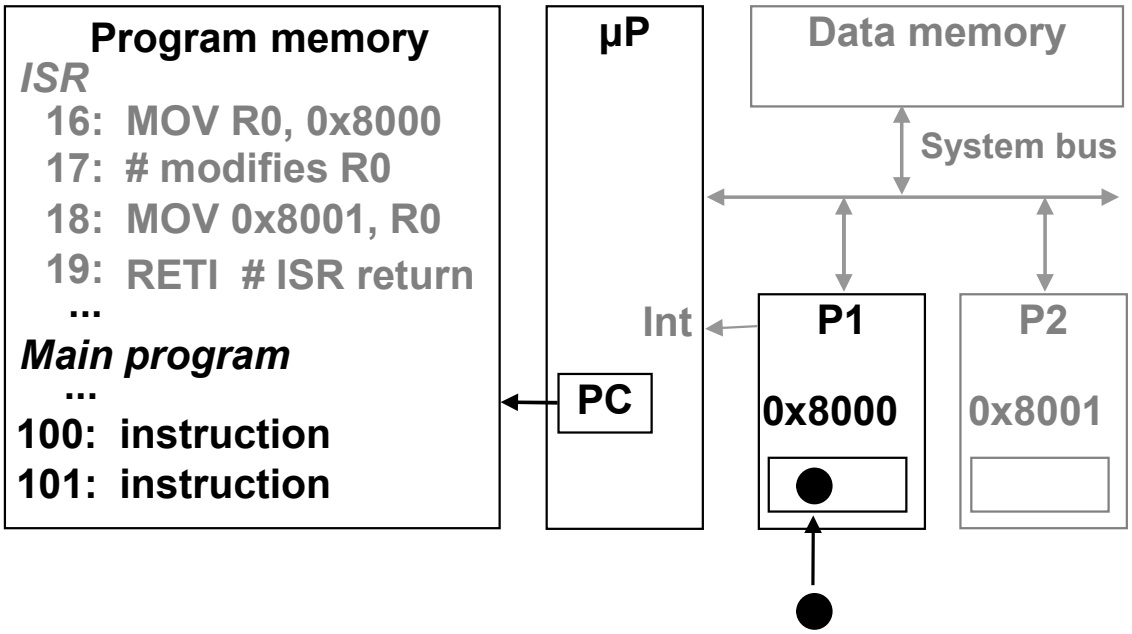
Interrupt

- How to determine the address (interrupt address vector) of the ISR?
  - Fixed interrupt
    - Address built into microprocessor, cannot be changed
    - Either ISR stored at address or a jump to actual ISR stored if not enough bytes available
  - Vectored interrupt
    - Peripheral must provide the address
    - Common when microprocessor has multiple peripherals connected by a system bus

Interrupt-driven I/O using fixed ISR location

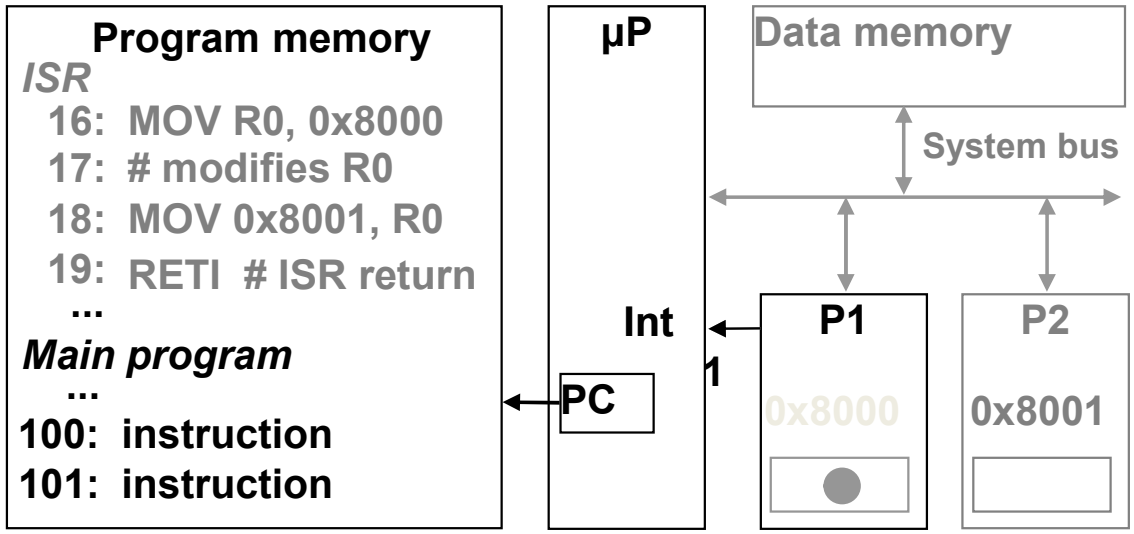
1(a): μP is executing its main program

1(b): P1 receives input data in a register with address 0x8000.



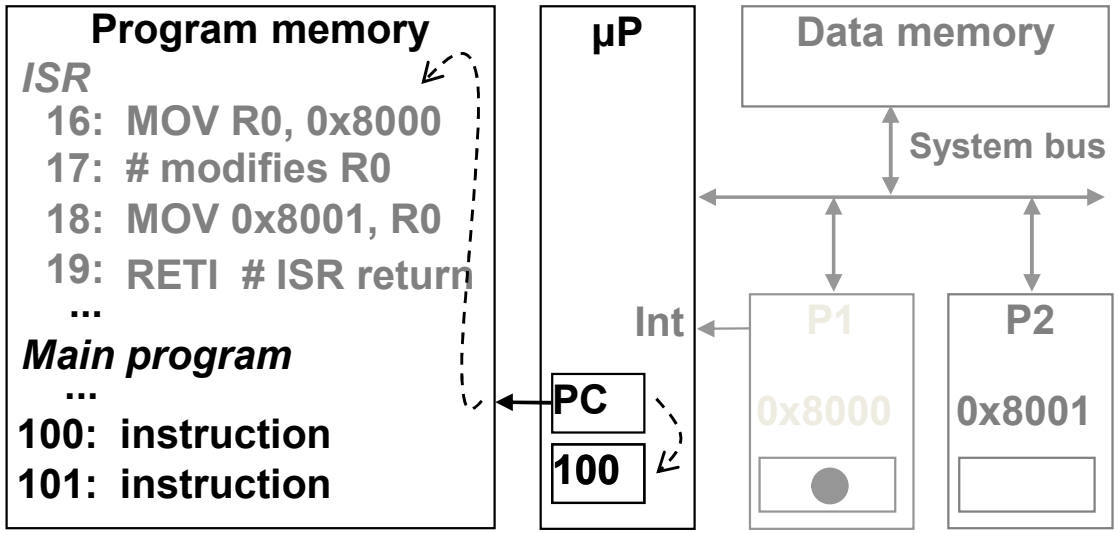
Interrupt-driven I/O using fixed ISR location

2: P1 asserts *Int* to request servicing by the microprocessor



Interrupt-driven I/O using fixed ISR location

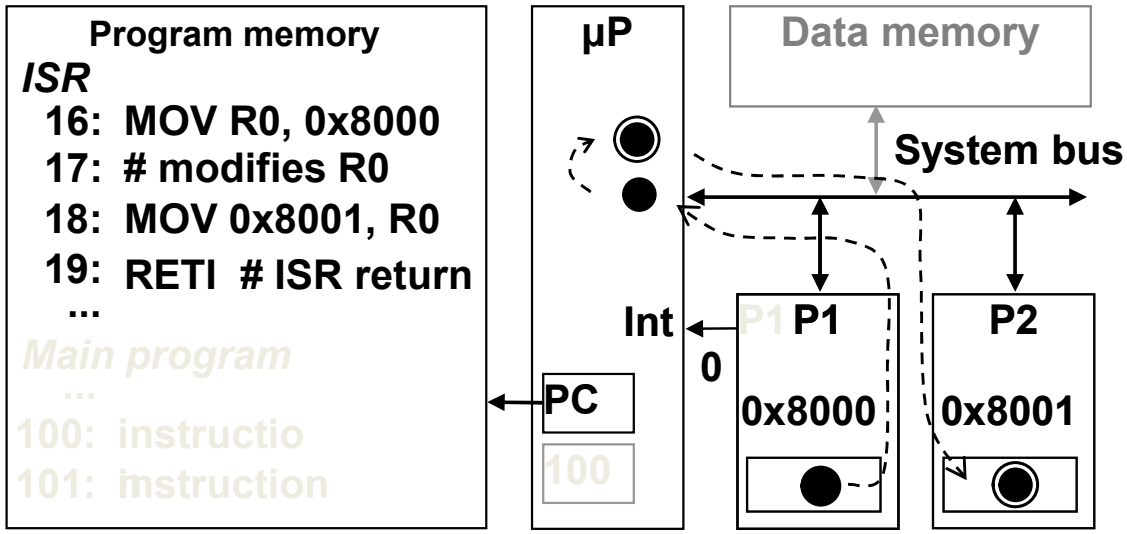
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location

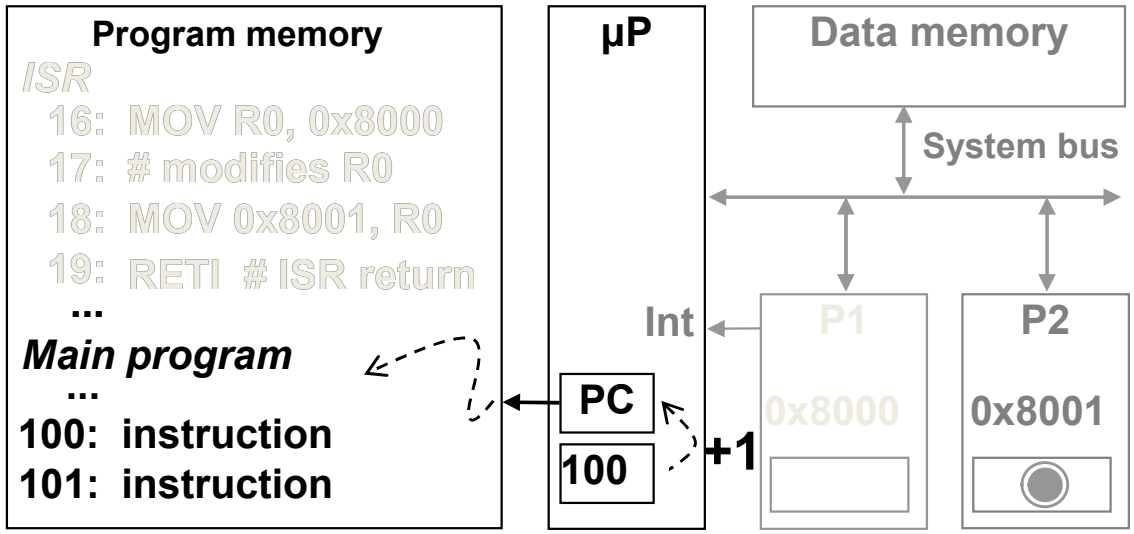
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

4(b): After being read, P1 deasserts *Int*.

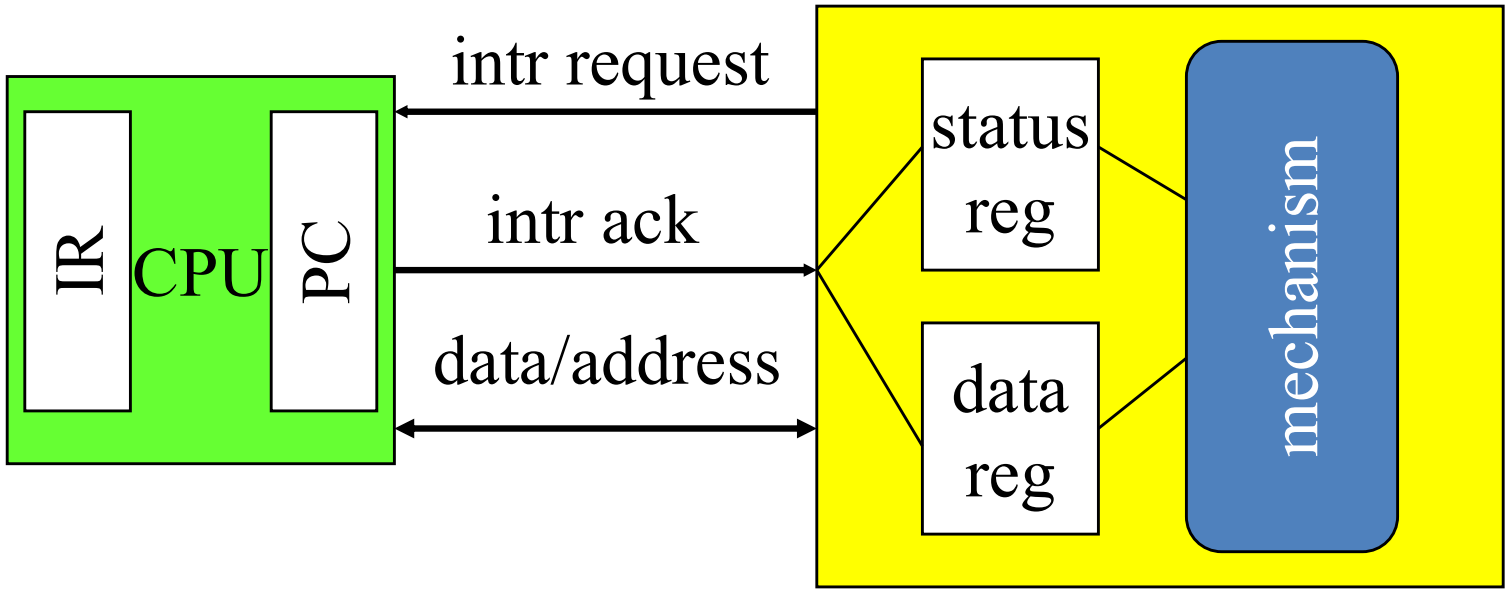


Interrupt-driven I/O using fixed ISR location

5: The ISR returns, thus restoring PC to 100+1=101, where μP resumes executing.

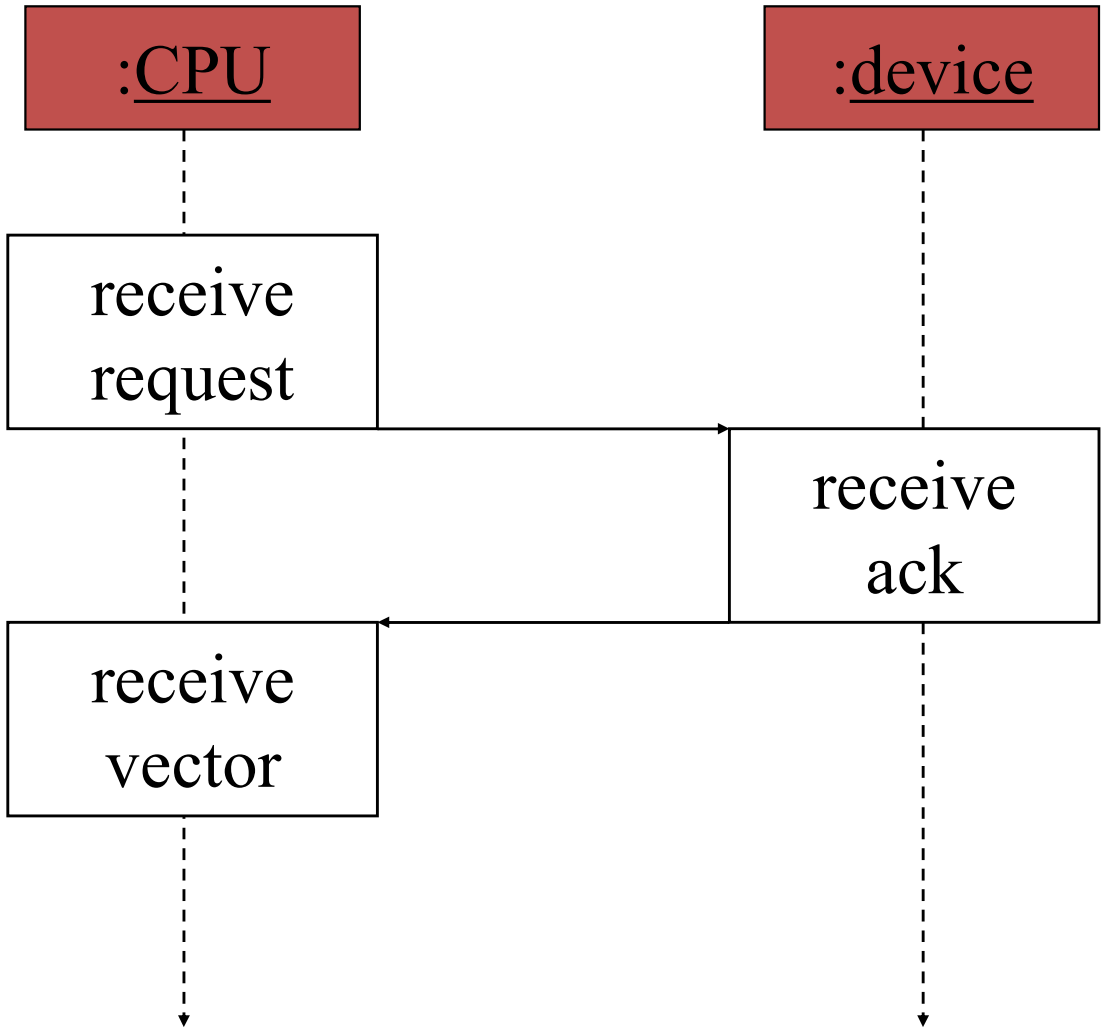


Interrupt-driven I/O using vectored interrupt



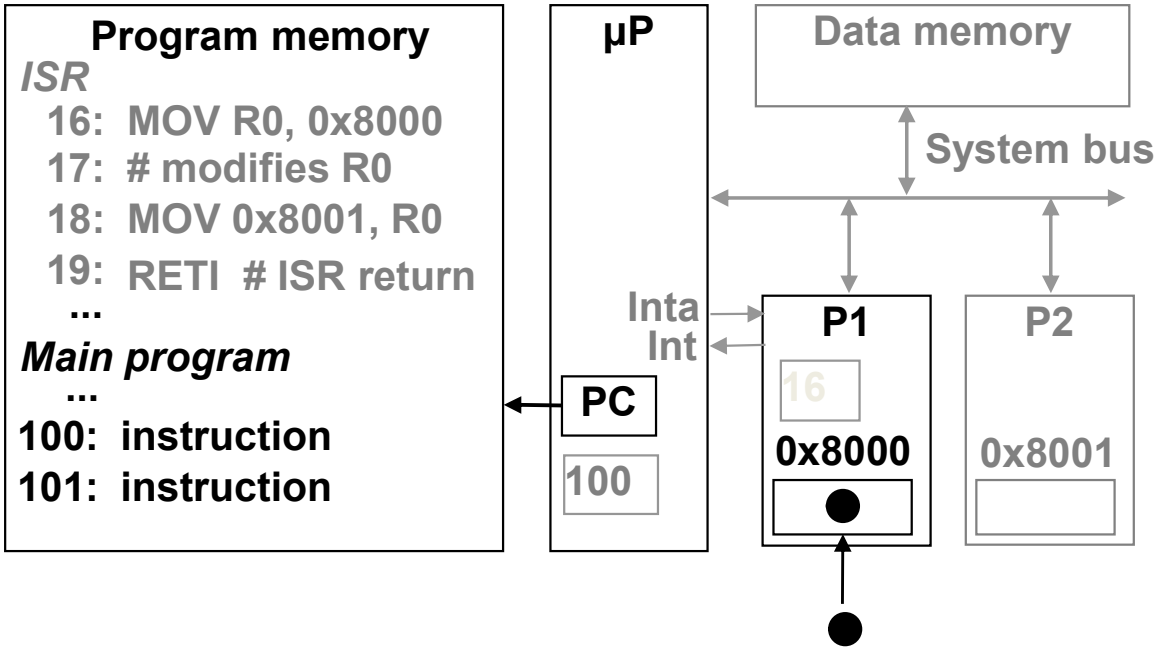
- CPU and device are connected by CPU bus.
- CPU and device handshake:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.
  - **peripheral provides this address on the data bus**

Interrupt-driven I/O using vectored interrupt



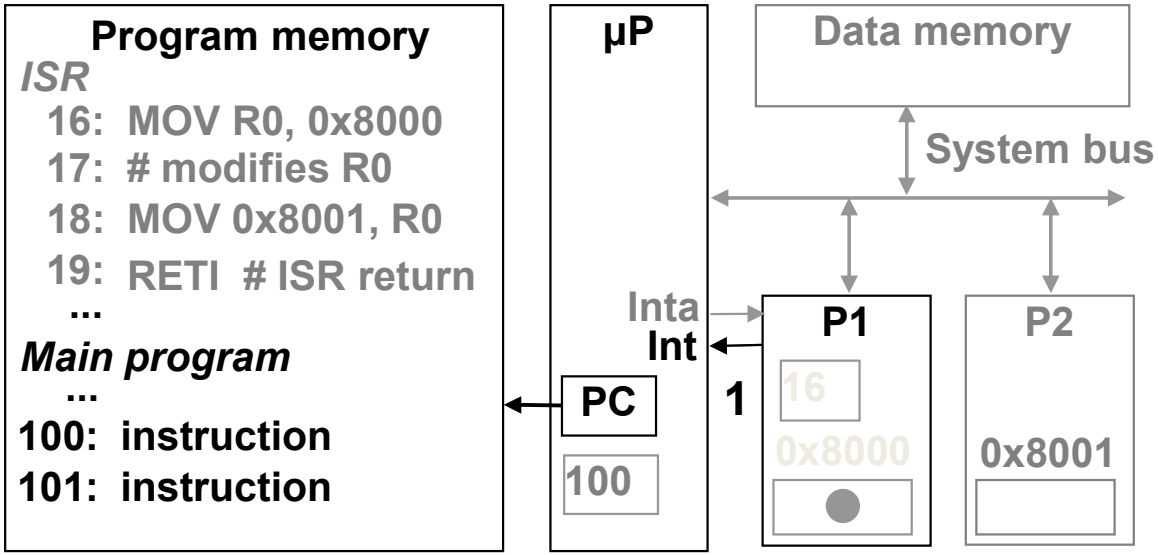
Interrupt-driven I/O using vectored interrupt

1(a): µP is executing its main program
1(b): P1 receives input data in a register with address 0x8000.



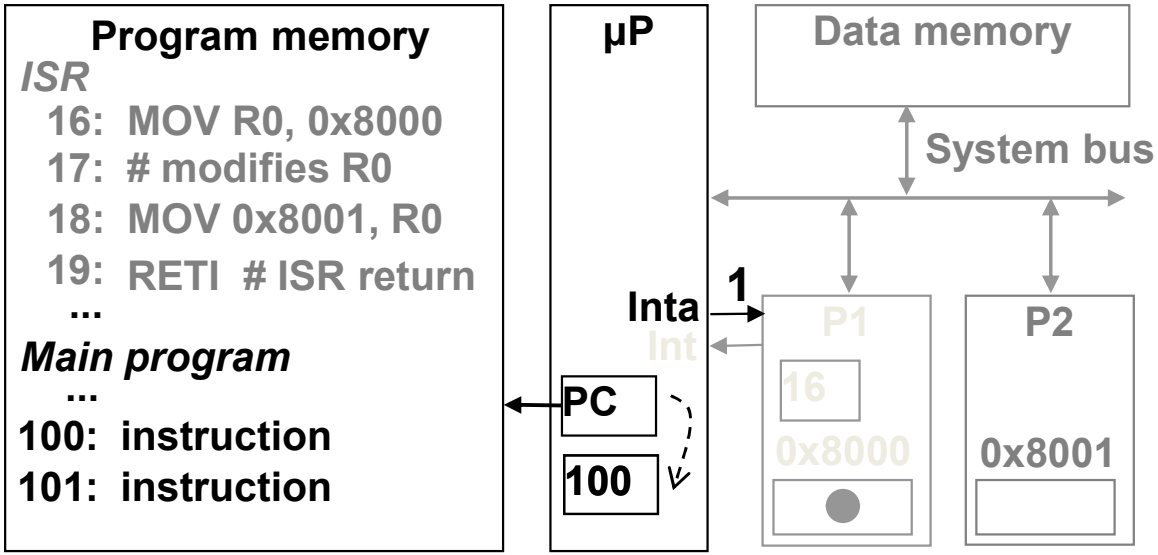
Interrupt-driven I/O using vectored interrupt

2: P1 asserts *Int* to request servicing by the microprocessor



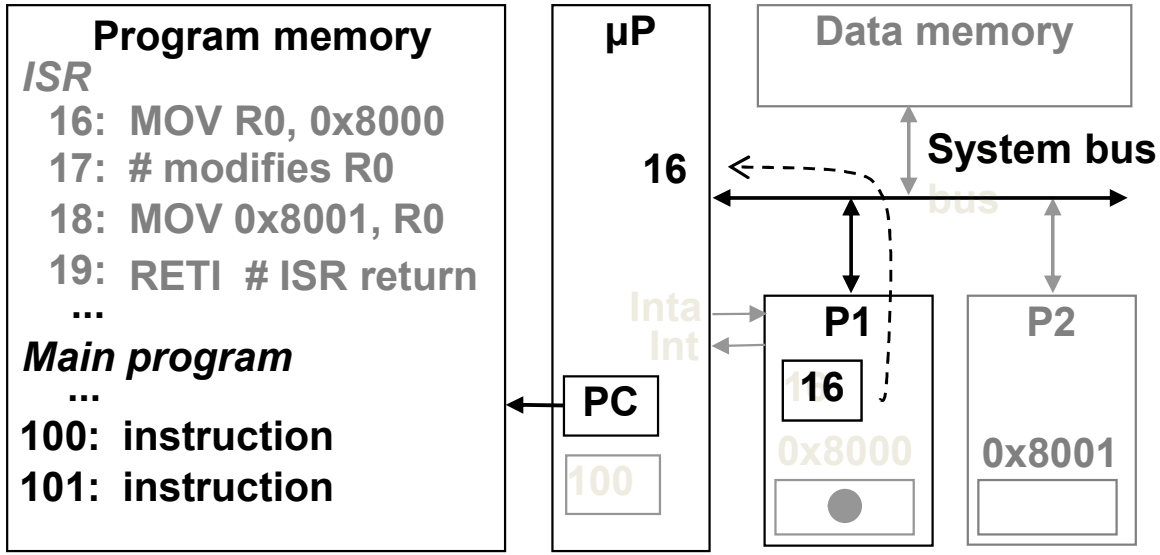
Interrupt-driven I/O using vectored interrupt

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*



Interrupt-driven I/O using vectored interrupt

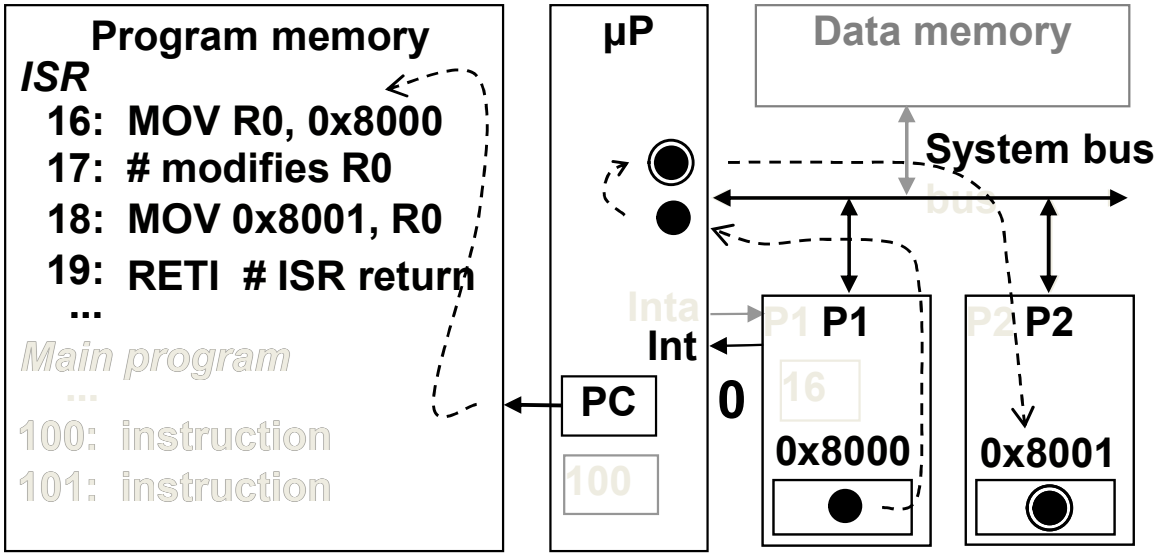
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus



Interrupt-driven I/O using vectored interrupt

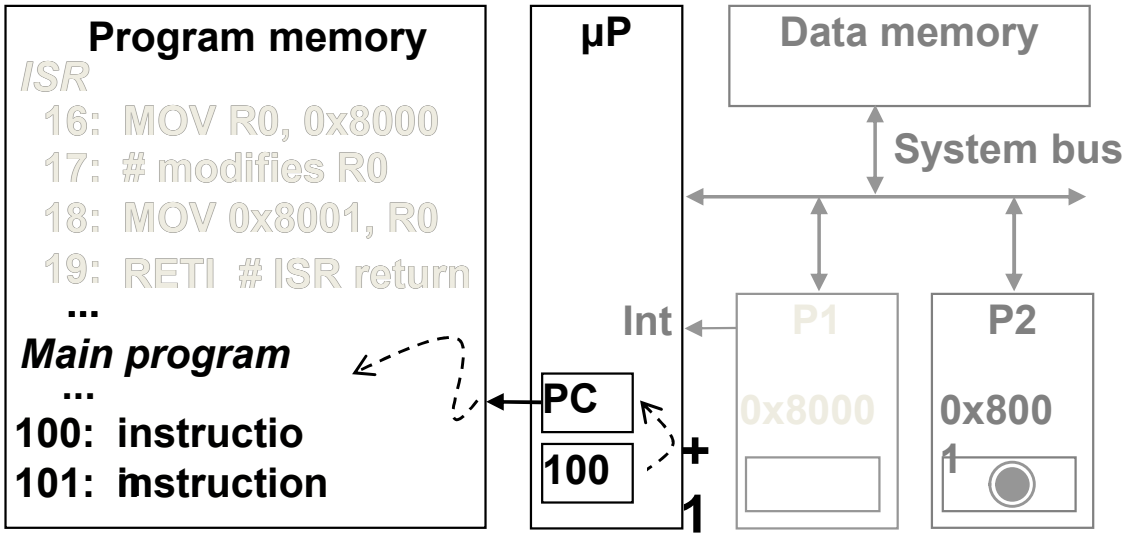
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

5(b): After being read, P1 deasserts *Int*.



Interrupt-driven I/O using vectored interrupt

6: The ISR returns, thus restoring the PC to 100+1=101, where the µP resumes



Quiz

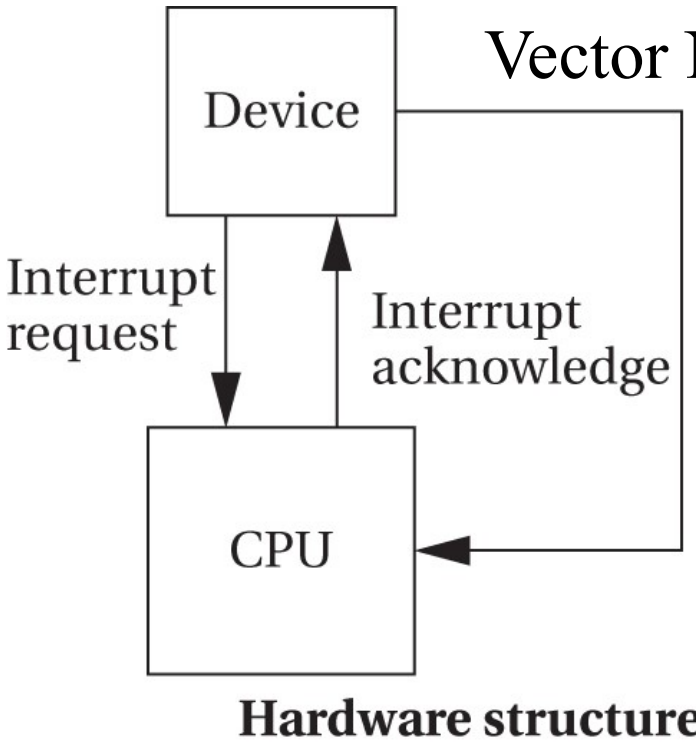
What Are the Difference between Exceptions and Interrupts ?

- **Exceptions/Interrupts are events that cause changes in program flow control outside a normal code sequence.**
- **The events could either be external or internal:**
  - External source : **Interrupt or interrupt request (IRQ)**
    - Interrupt handler, or interrupt service routine (ISR)
  - Internal source: **Exceptions**
    - Exception handler

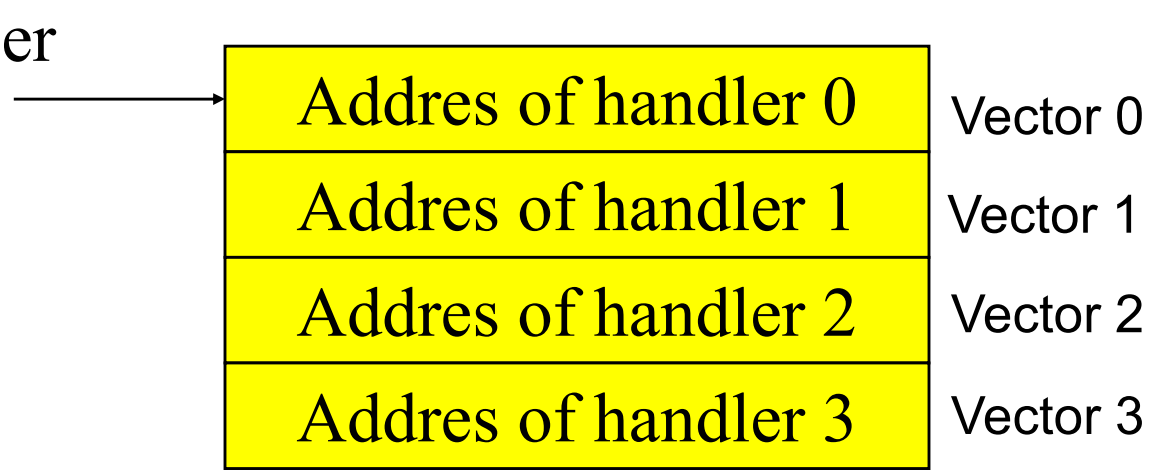
Priorities and vectors

- Two mechanisms allow us to make interrupts more **flexible**:
  - **Priorities** determine what interrupt gets CPU first.
  - **Interrupt vectors Table** allows the different interrupting devices to be handled by different handler
    - Vectors number (or Interrupt number) determine what code is called for each type of interrupt.
    - Handler address points to Interrupt entry point in memory
- Mechanisms are orthogonal: most CPUs provide both.

Interrupt vectors



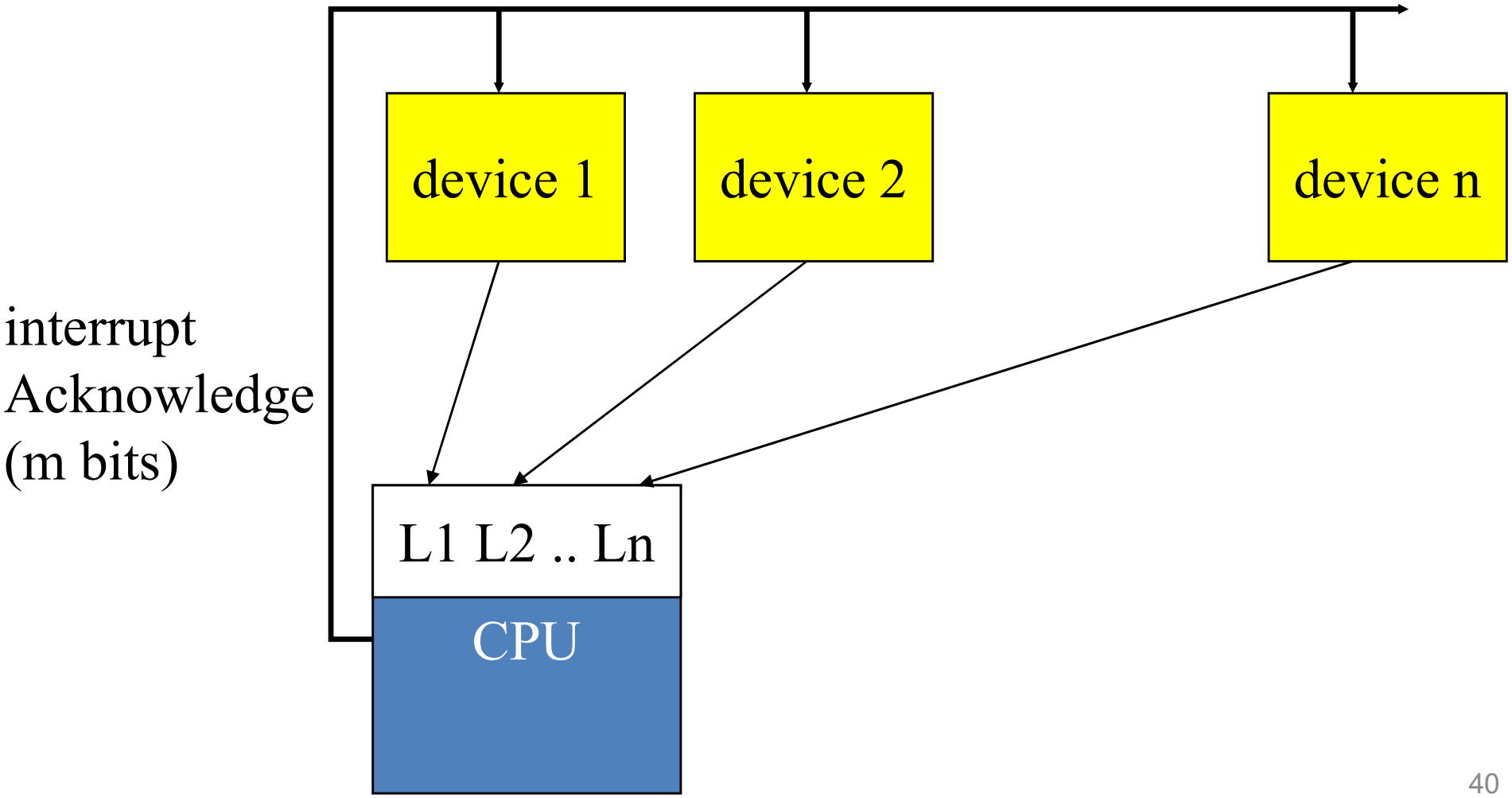
Hardware structure



- CPU acknowledges request.
- Device sends vector.
- CPU calls handler.
- Software processes request.
- CPU restores state to foreground program.

Prioritized interrupts

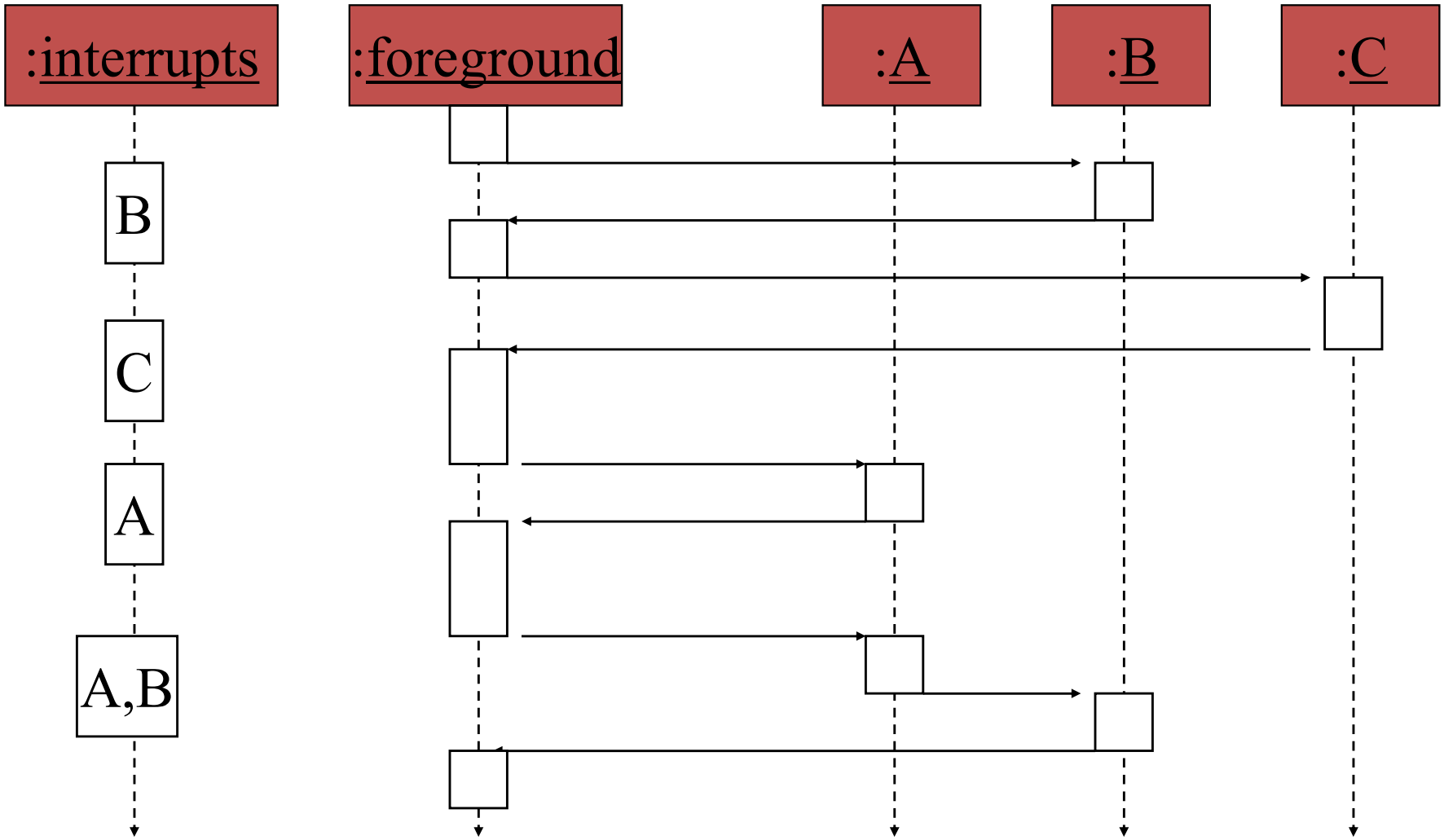
- Prioritized interrupts allow the CPU to ignore less important interrupt requests while it handles more important requests



Prioritized interrupts

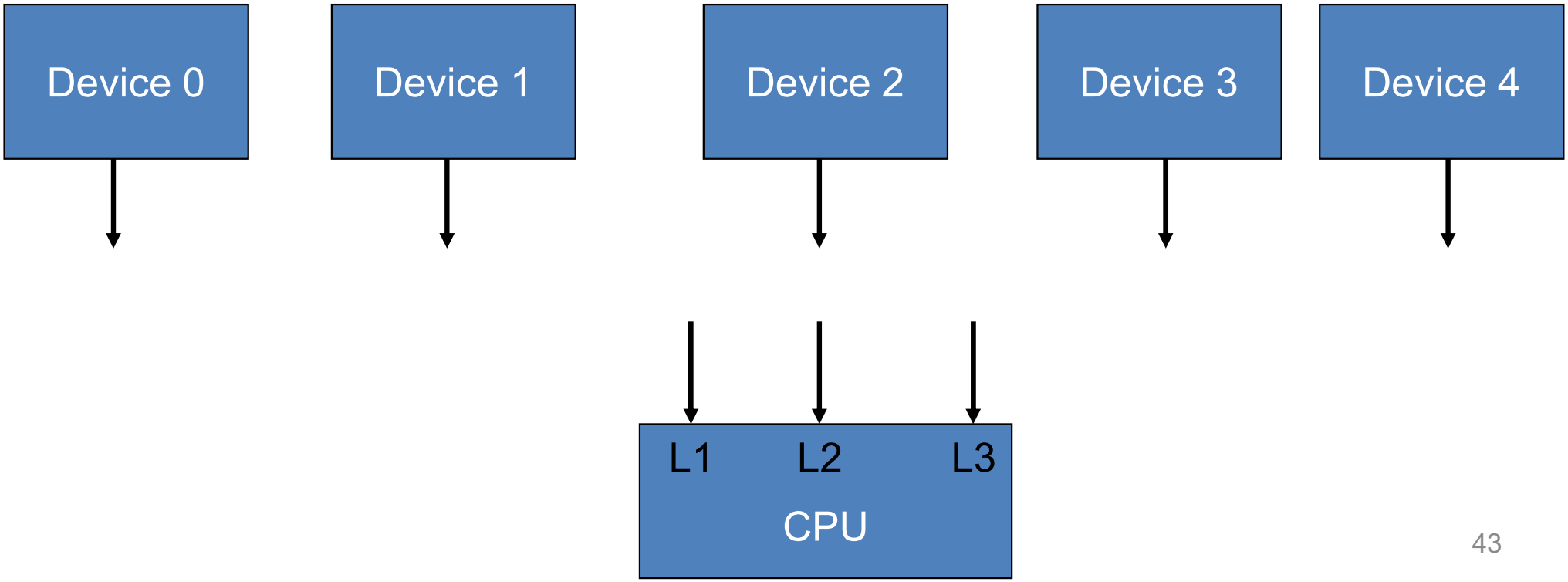
- **The priority mechanism must ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is being handled.**
- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- Non-maskable interrupt (NMI): highest-priority, never masked.
  - Often used for power-down.

Example: Prioritized I/O

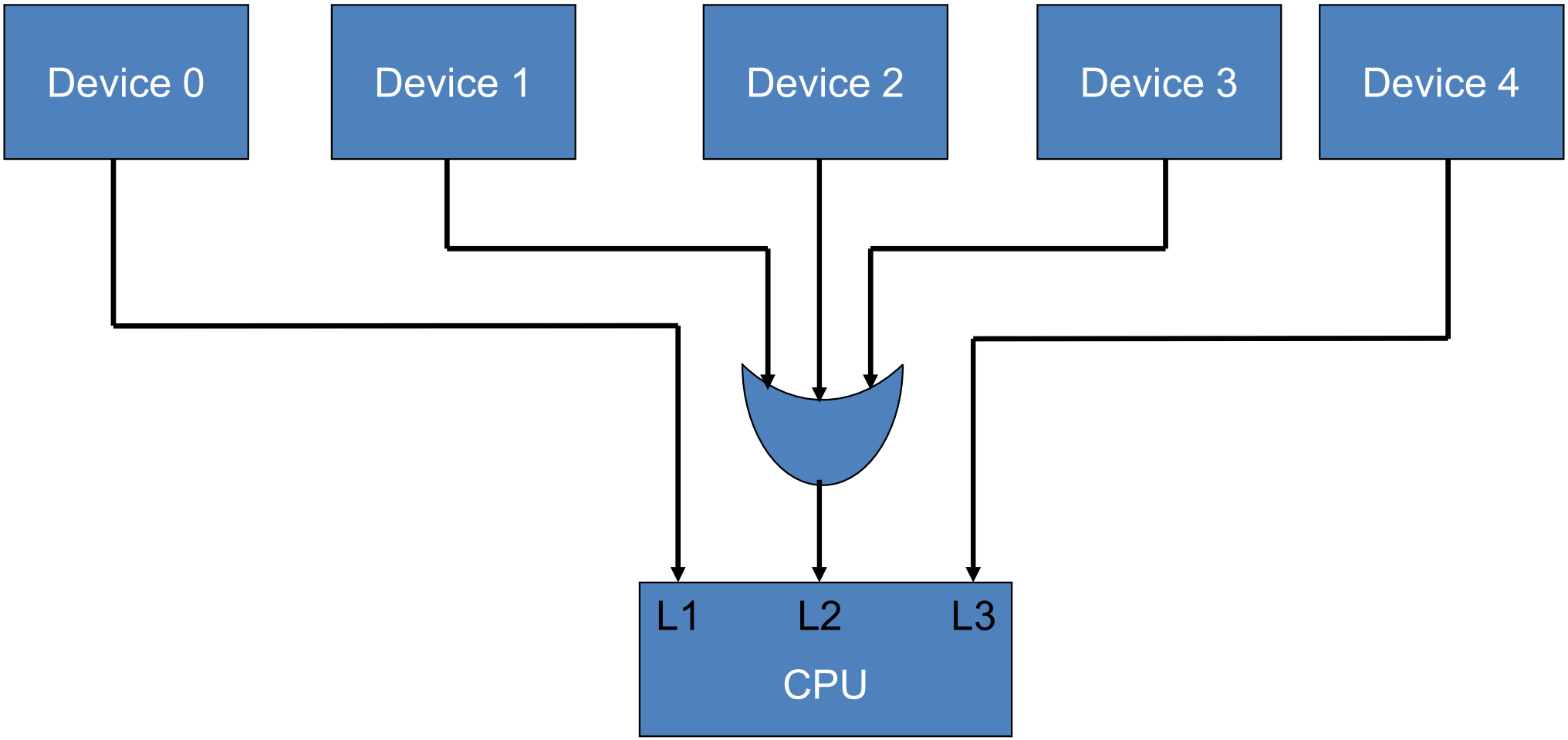


Prioritized interrupts

How to solve the case in which the number of I/O devices is more than the number of interrupt priority levels?



Prioritized interrupts



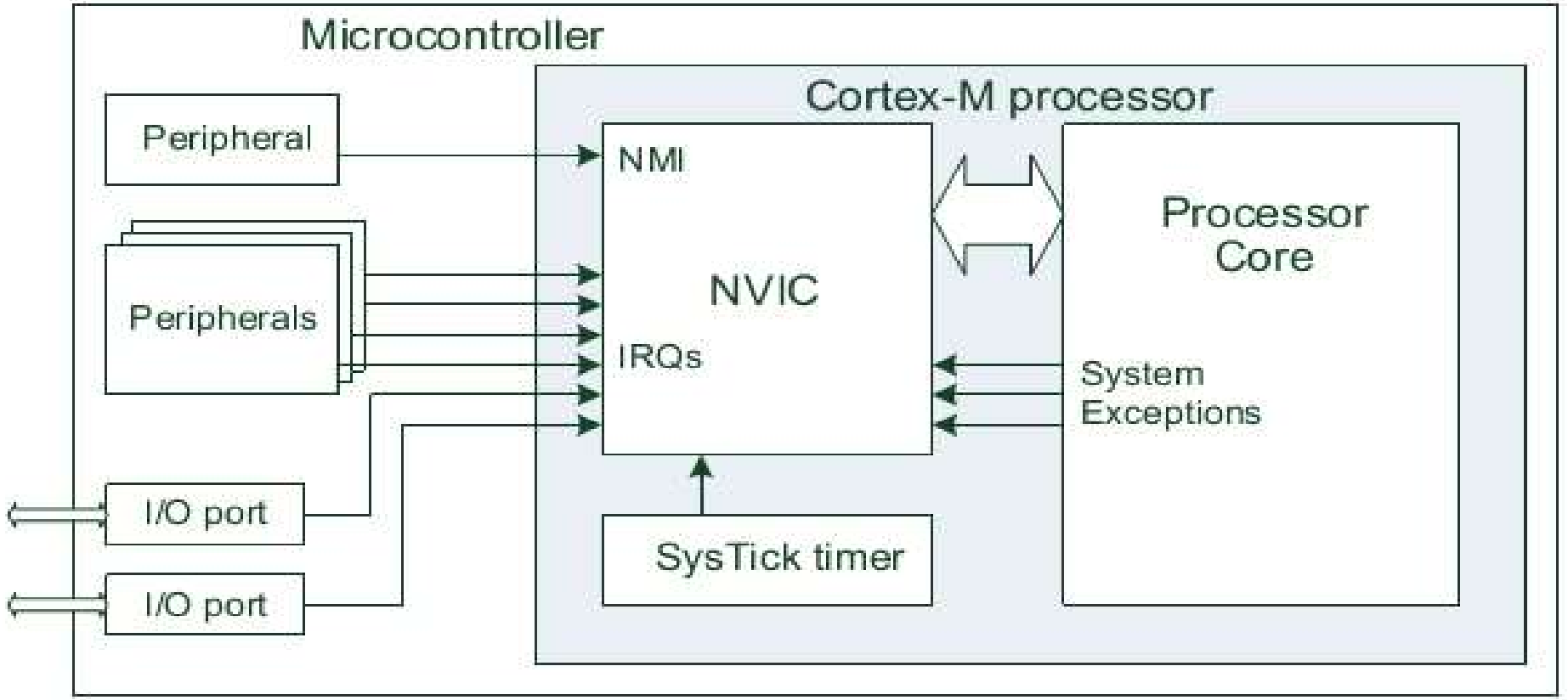
Use a mixture of interrupt-driven and polling methods to construct the program

Outline

- Programming Input and Output
- Polling vs. Interrupt
- **Interrupt Mechanism**
  - **Interrupt implementation in Cortex-M**
- Summary

Interrupt Management in Cortex-M Processors

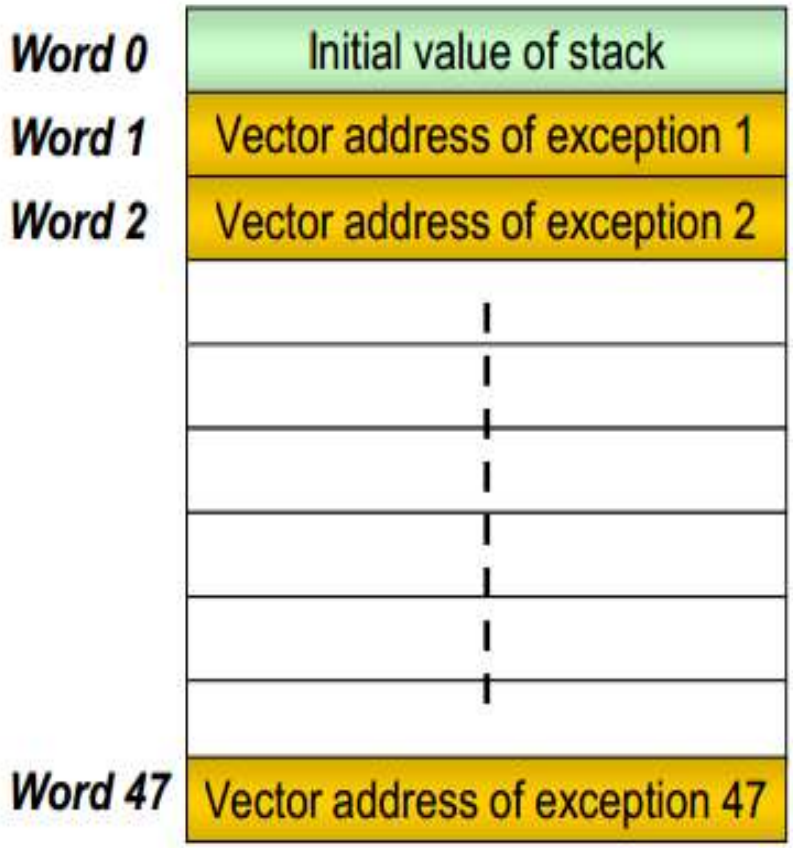
- **Nested Vectored Interrupt Controller (NVIC)**



Various sources of exceptions in a typical microcontroller

ARM Cortex-M's Exception Model

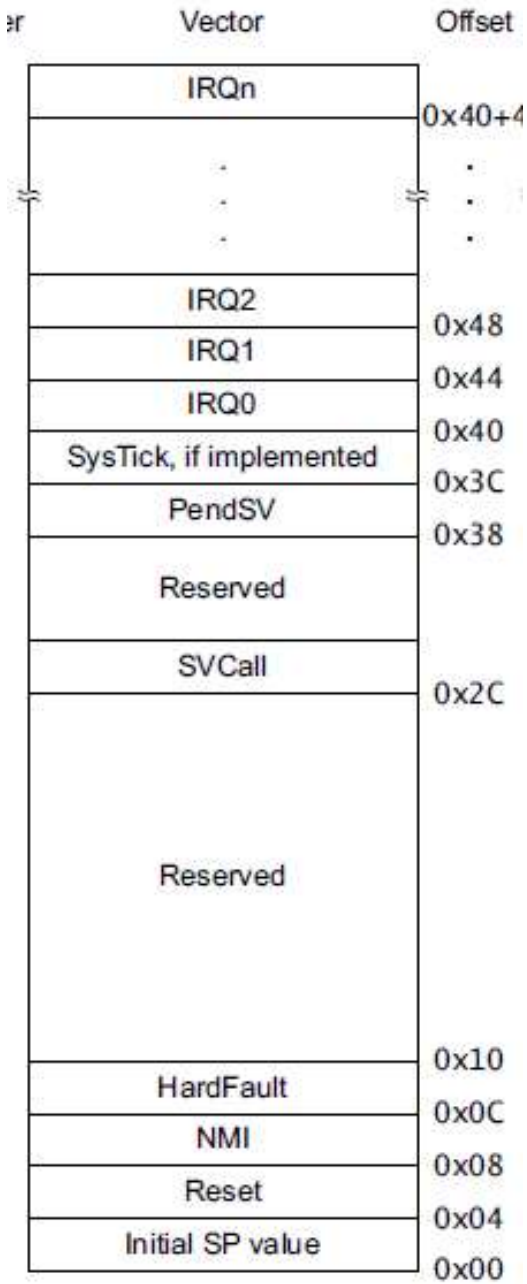
- An exception may be an internal interrupt or a hardware error.
- Each exception has exception number, priority number and vector address
- Vector table base address is fixed at 0x00
- Vector table is normally defined in the startup codes (startup.s)



Vector Table

ARM Cortex-M3's Exception Model

- SP value: the reset value of the stack pointer.
- Reset is invoked on power up or a warm reset
- A Non-Maskable Interrupt (NMI)
- A HardFault is an exception that occurs because of an error.
- A Supervisor Call (SVC) is an exception that is triggered by the SVC instruction
- PendSV is an interrupt-driven request for system-level service
- a SysTick exception is generated when the SysTick timer reaches zero



ARM Cortex-M's Exception Model

- **Interrupt management**

Excp. Number	Exception Type	Priority
1	Reset	-3 (highest)
2	NMI	-2
3	Hard Fault	-1
4	MemManage Fault	Programmable
5	Bus Fault	Programmable
6	Usage Fault	Programmable
7-10	Reserved	NA
11	SVC	Programmable
12	Debug Monitor	Programmable

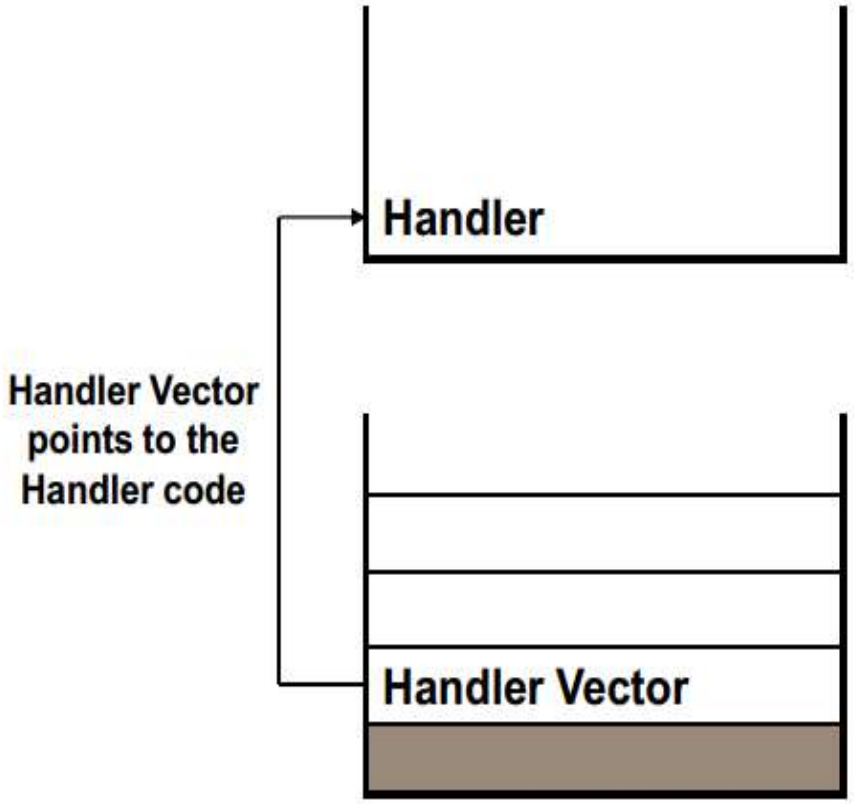
Excp. Number	Exception Type	Priority
13	Reserved	NA
14	PendSV	Programmable
15	SysTick	Programmable
16	Interrupt #0	Programmable
17	Interrupt #1	Programmable
…		
47	Interrupt #31	Programmable
…		
255	Interrupt #239	Programmable

ARM Cortex-M's Exception Model

Vector Table Usage:

In the case of an exception, the core:

- Reads the vector handler address for the exception from the vector table
- Branches to the handler



ARM Cortex-M's Exception Model

Vector Table Implementation (Startup.s in uVision):

```
; Vector Table Mapped to Address 0 at Reset

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size
__Vectors       DCD     __initial_sp  ; Top of Stack
                DCD     Reset_Handler  ; Reset Handler
                DCD     NMI_Handler  ; NMI Handler
                DCD     HardFault_Handler  ; Hard Fault Handler
                DCD     0  ; Reserved
                DCD     0  ; Reserved
                DCD     0  ; Reserved
                …
```

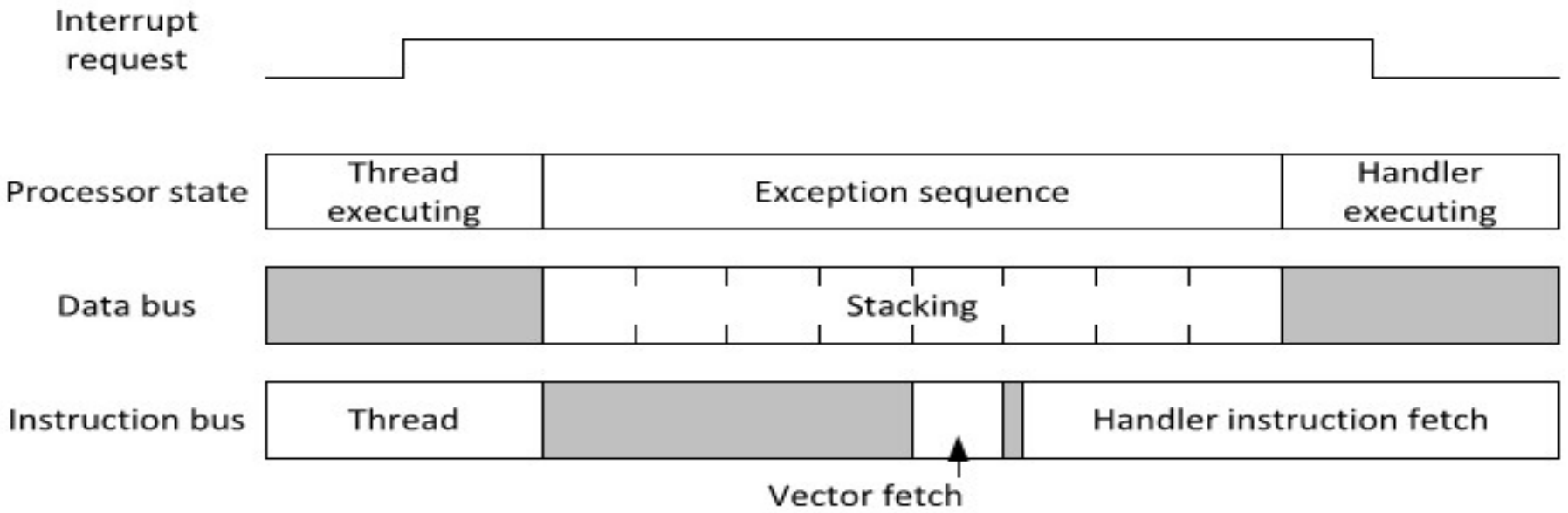
ARM Exception Sequence

- CPU actions:
  - Save PC. Copy CPSR to SPSR.
  - Force bits in CPSR to record interrupt.
  - Force PC to Exception handler.
- Handler responsibilities:
  - Restore proper PC.
  - Restore CPSR from SPSR.
  - Clear interrupt disable flags.

ARM Exception Sequence

An exception entrance sequence contains:

1. Stacking of a number of registers
2. Fetching the exception vector
3. Fetching the instructions for the exception handler to be executed.
4. Update NVIC and core registers



ARM Exception Sequence

Exception handler execution:

1. The processor is in Handler mode
   - The Main Stack Pointer (MSP) is used for stack operations
   - The processor is executing in privileged access level
2. **Nested Interrupt:** If a higher-priority exception arrives, the currently executing handler will be suspended and pre-empted

Interrupt Latency

- **Interrupt latency** refers to the delay from the start of the interrupt request to the start of the interrupt handler execution.
- The best-case latency for Cortex-M3/4 is 12 cycles, including:
  - Cycles for stacking the registers, vector fetch, and fetching instructions for the interrupt handler
  - Memory system has zero latency,
  - The bus system design allows vector fetch and stacking to happen at the same time.

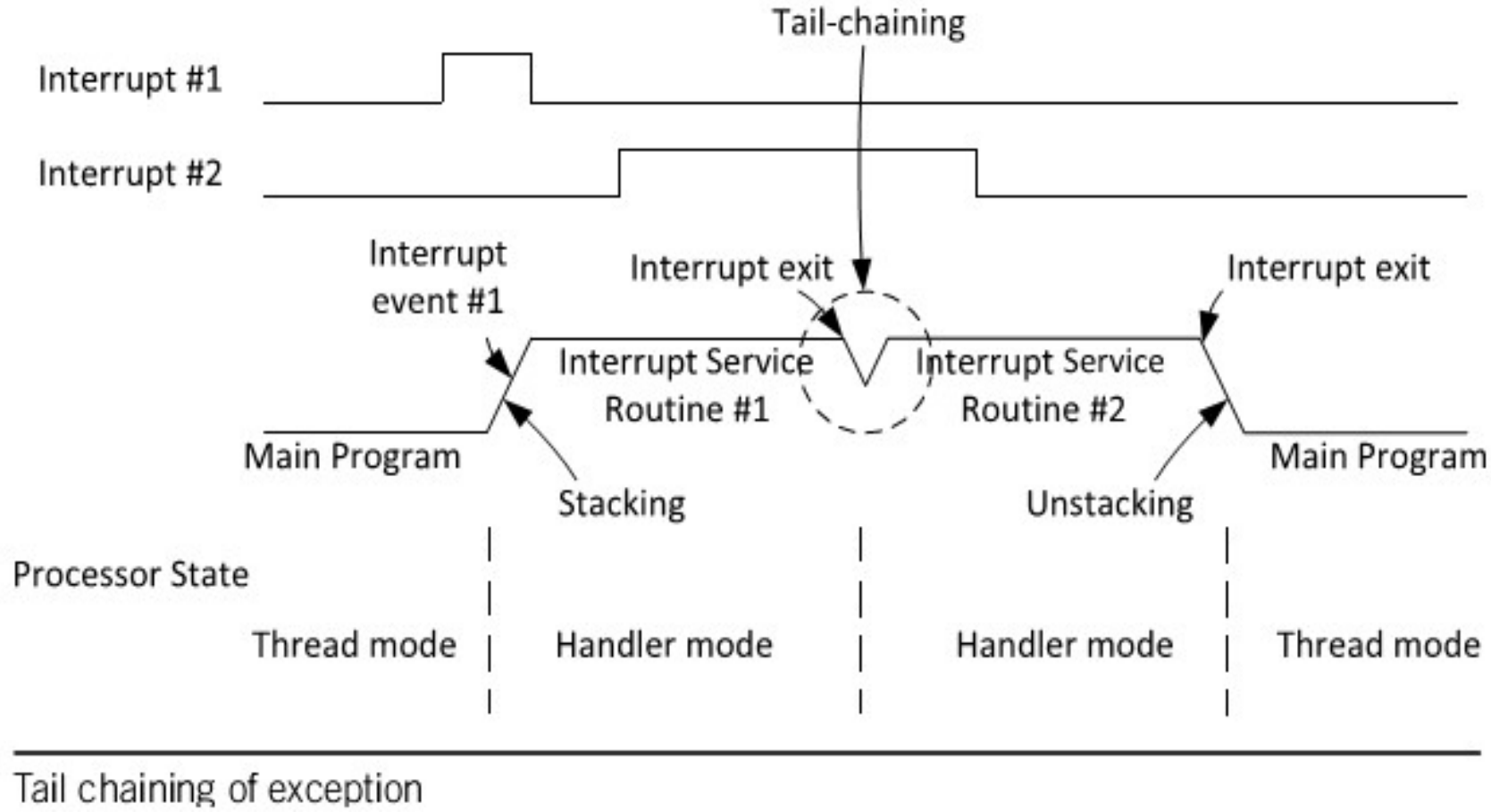
Exception handling optimization

Cortex-M processors use a number of methods to reduce the latency of servicing interrupts:

- Tail chaining
- Late arrival
- Pop preemption

Exception handling optimization

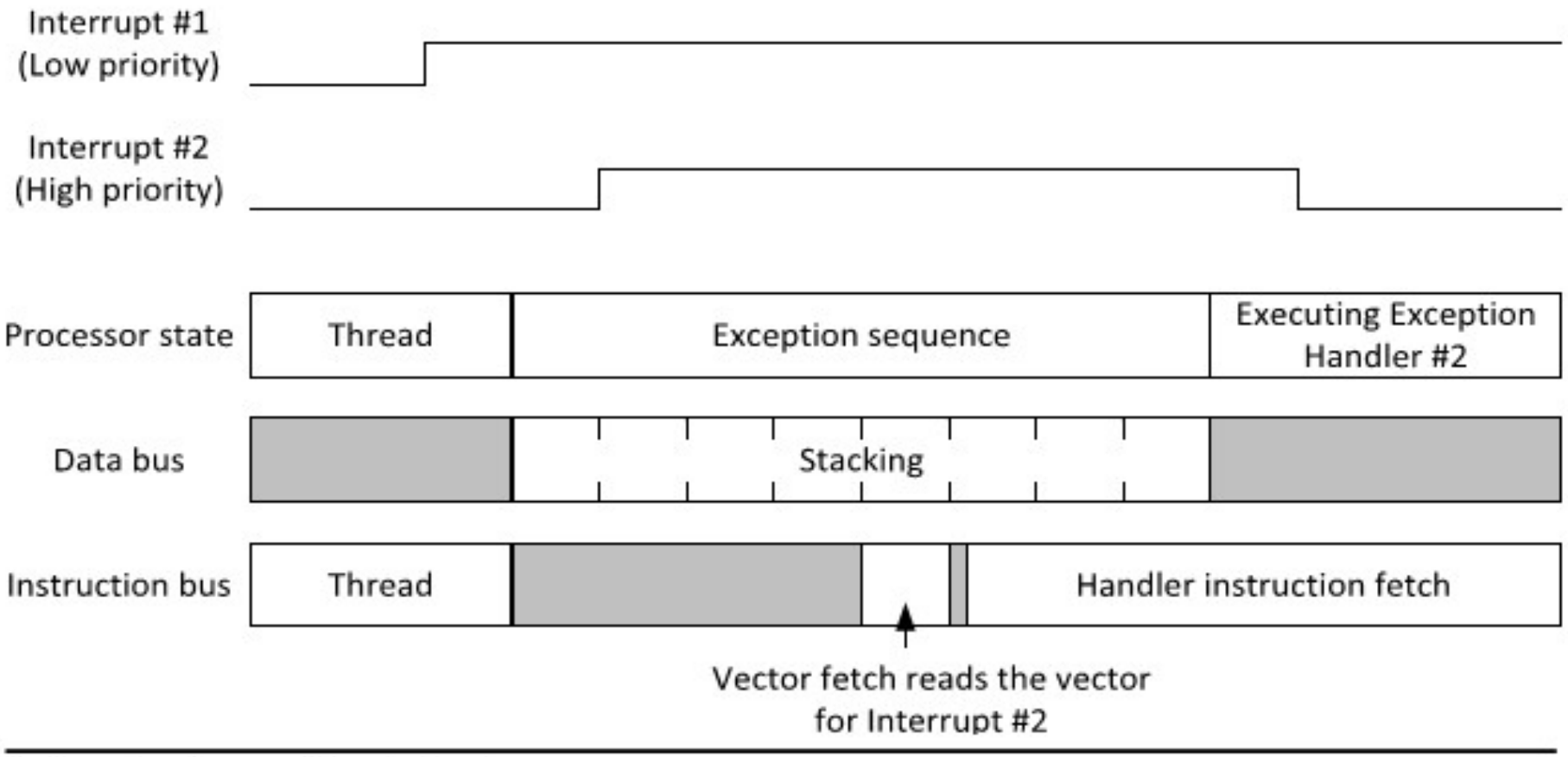
Tail chaining:



Tail chaining of exception

Exception handling optimization

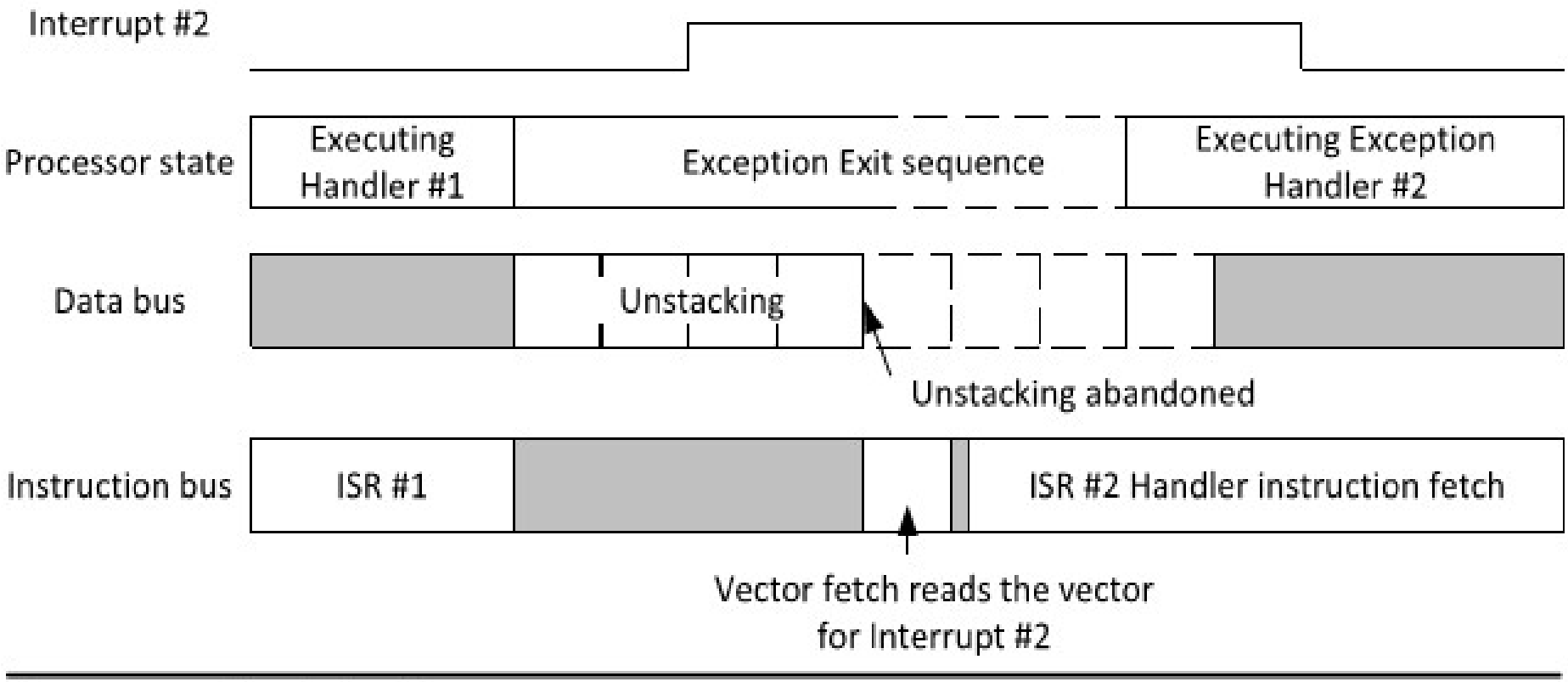
Late arrival:



Late arrival exception behavior

Exception handling optimization

Pop preemption:



Pop pre-emption behavior

Summary

- The two major styles of I/O are polled and interrupt driven
  - Interrupts may be vectorized and prioritized
  - Nested Vectored Interrupt Controller (NVIC) for interrupt handling

Quiz

Q1: Why do most computer systems use memory-mapped I/O?

Q2: Why do most programs use interrupt-driven I/O over polling?
Q3: Write ARM code that tests a register at location ds1 and continues execution only when the register is nonzero.
Q3: Write ARM code that waits for the low-order bit of device register ***ds1*** to become 1 and then reads a value from register ***dd1***.
Q4: What is polling? Draw a software flow diagram for polling three I/O peripheral devices? What are drawbacks of polling method?
Q5: What is interrupt-driven I/O? Draw a software flow diagram for interrupt-driven access of three I/O peripheral devices?
Q6: Describe steps performed by an CPU when it responds to an interrupt
Q7: Distinguish between interrupt, exception.

Quiz

Q9: **Why do most programs use interrupt-driven I/O over polling I/O? When would you prefer to use polling I/O over interrupt-driven I/O?**