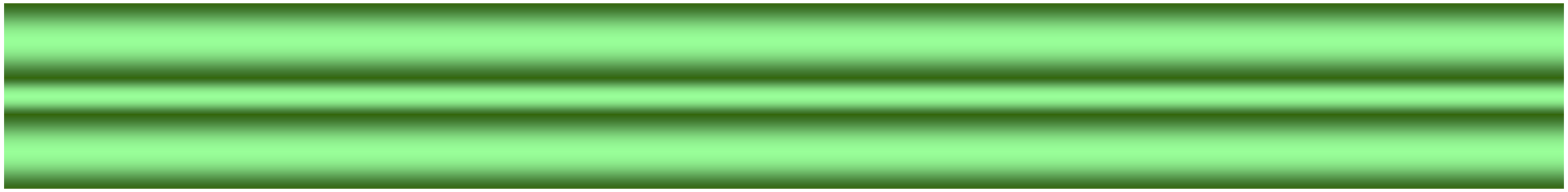
EE414 Embedded Systems

Ch 6. Interfacing

Part 2/4: Interrupts



Byung Kook Kim
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Overview

- 6.1 Introduction
- 6.2 Communication Basics
- 6.3 Microprocessor interfacing: I/O Addressing

- **6.4 Interrupts**

- 6.5 Interrupts in AM3359
- 6.6 Exceptions in ARM
- 6.7 Timers & Counters
- 6.8 Timer & Clock in AM3359
- 6.9 Direct memory access
- 6.10 Arbitration
- 6.11 Multilevel Bus Architectures

6.4 Interrupts

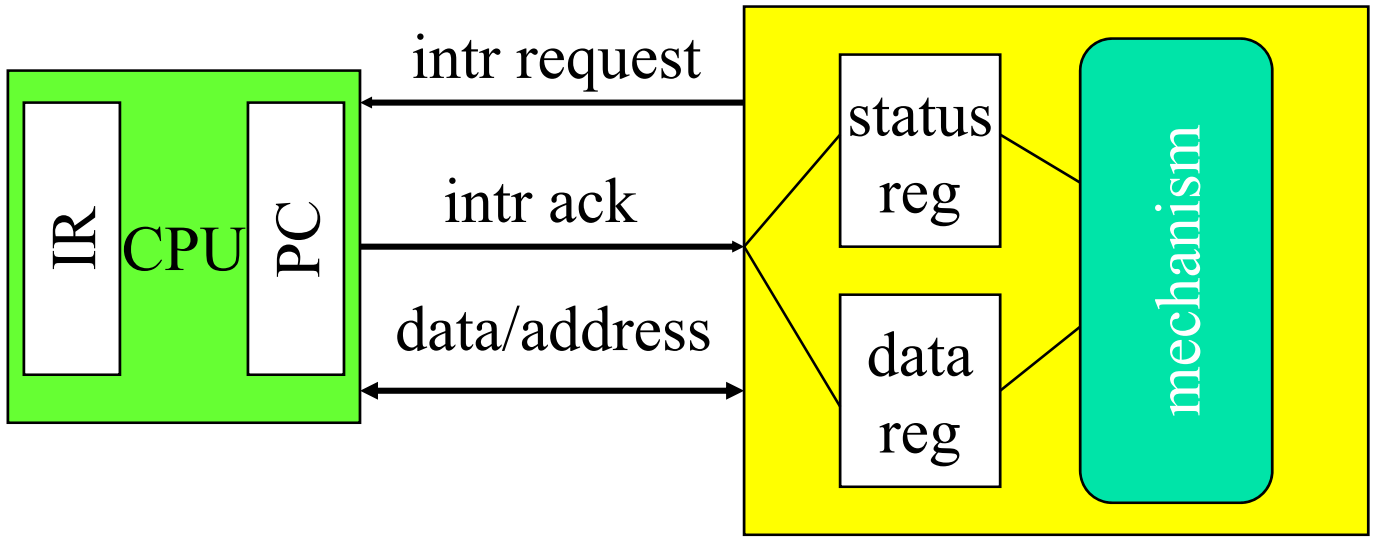
- **What is an interrupt?**
  - Analogy in everyday life:
    - Phone call while reading a book
  - *An event either from an internal or external source where*
    - *a processor will stop its current processing thread,*
    - *switch to a different instruction sequence (interrupt service routine),*
    - *and then resume its current processing.*

Interrupt I/O

- *Polling (Busy/wait)* is *very inefficient.*
  - CPU can't do other work while testing the device.
  - Hard to do simultaneous I/O.
    - Two serial inputs: Where to wait?

- *Interrupts* allow a device to change the flow of control in the CPU.
  - Requires an extra pin or pins: Int (IRQ)
  - If Int is 1, processor suspends current program, jumps to an Interrupt Service Routine, or ISR
  - Known as interrupt-driven I/O
  - Essentially, "polling" of the interrupt pin is built-into the *hardware*, so no extra time!

Interrupt Interface

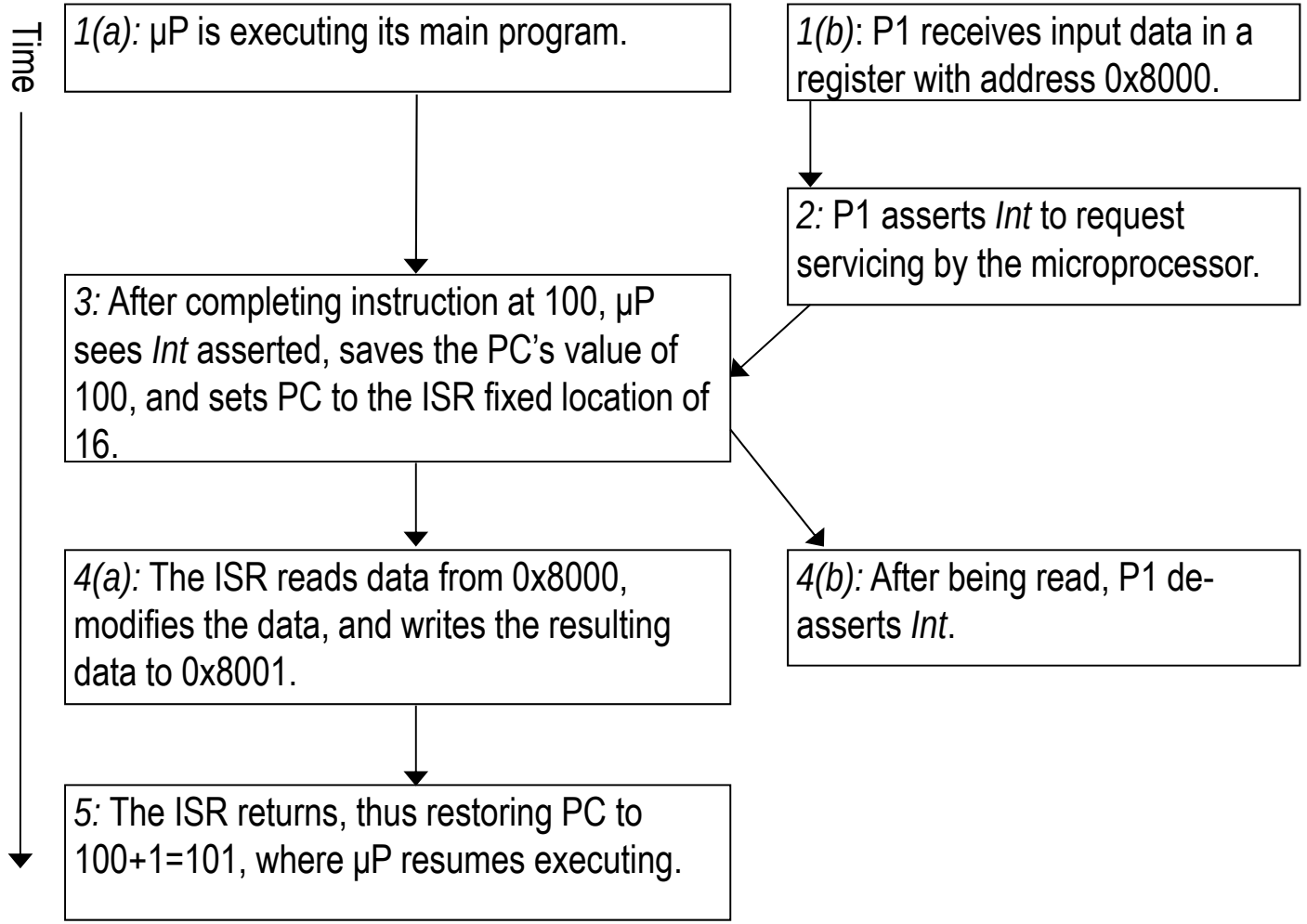


- CPU and device are connected by CPU bus.
- CPU and device **handshake**:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.

Interrupt Behavior

- Based on subroutine call mechanism.
  - Interrupt Service Routine (ISR)
    - Background program
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
  - Return address is saved to resume executing foreground program.

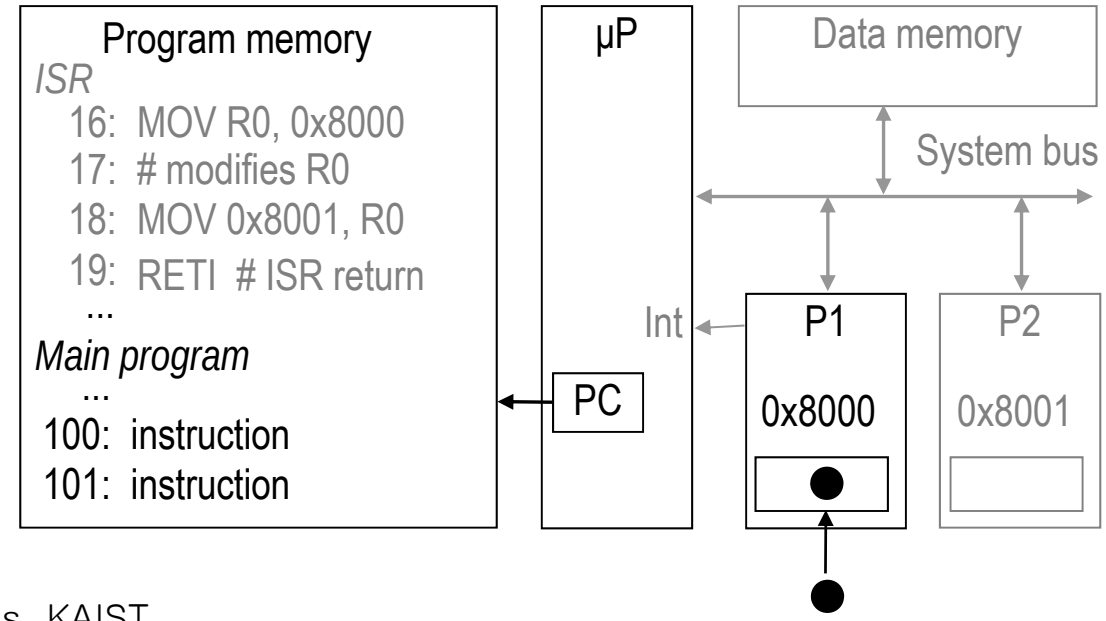
Interrupt-driven I/O using fixed ISR location



Interrupt-driven I/O using fixed ISR location (I)

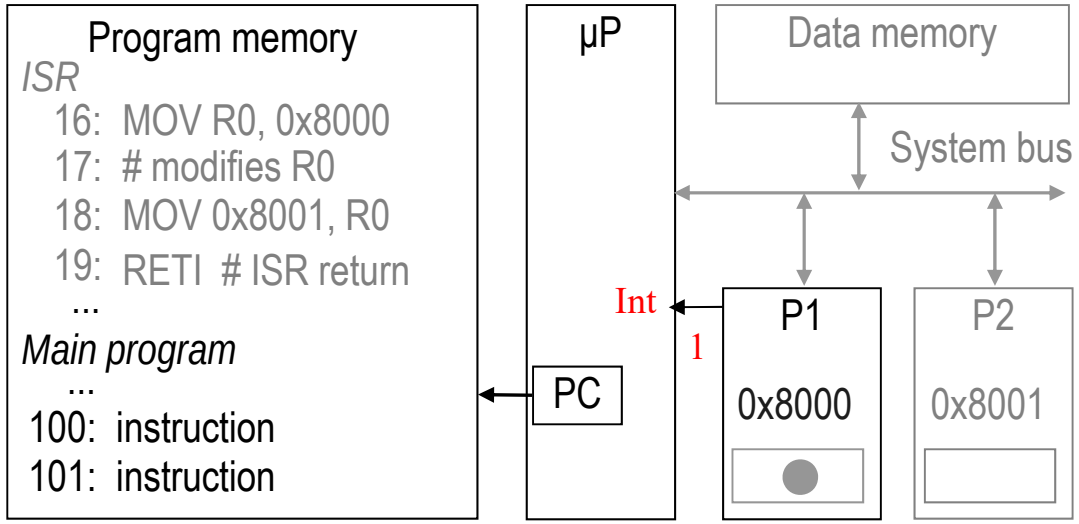
1(a): μP is executing its main program

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



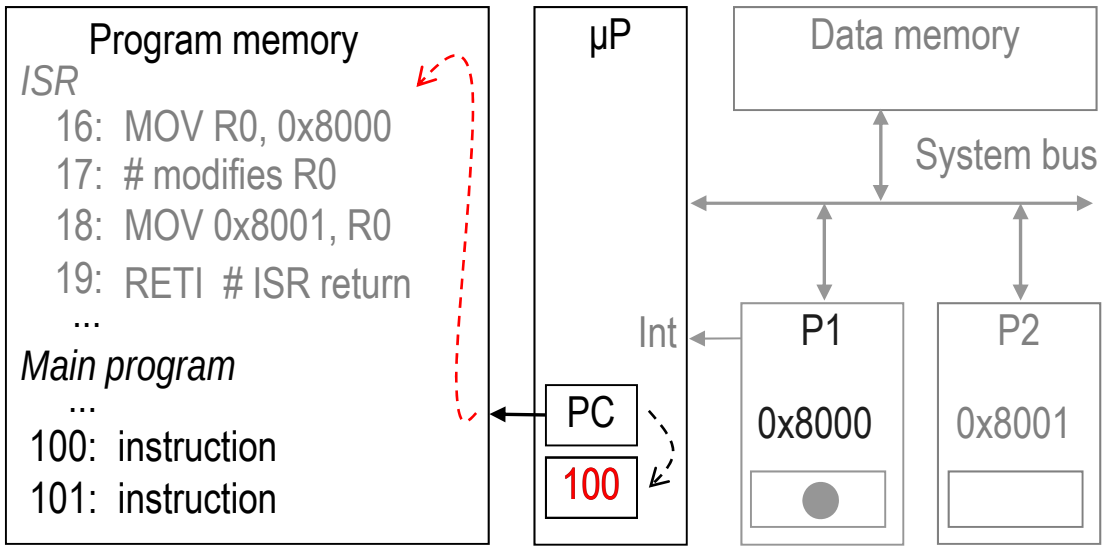
Interrupt-driven I/O using fixed ISR location (II)

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



Interrupt-driven I/O using fixed ISR location (III)

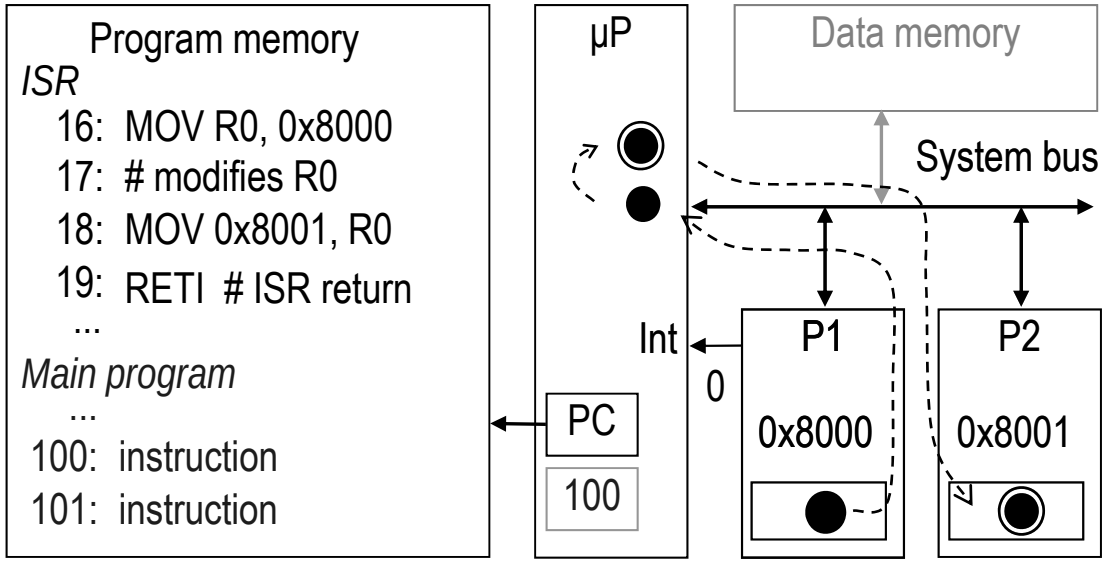
3: After completing instruction at 100,
μP sees *Int* asserted, saves the PC's
value of 100,
and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location (IV)

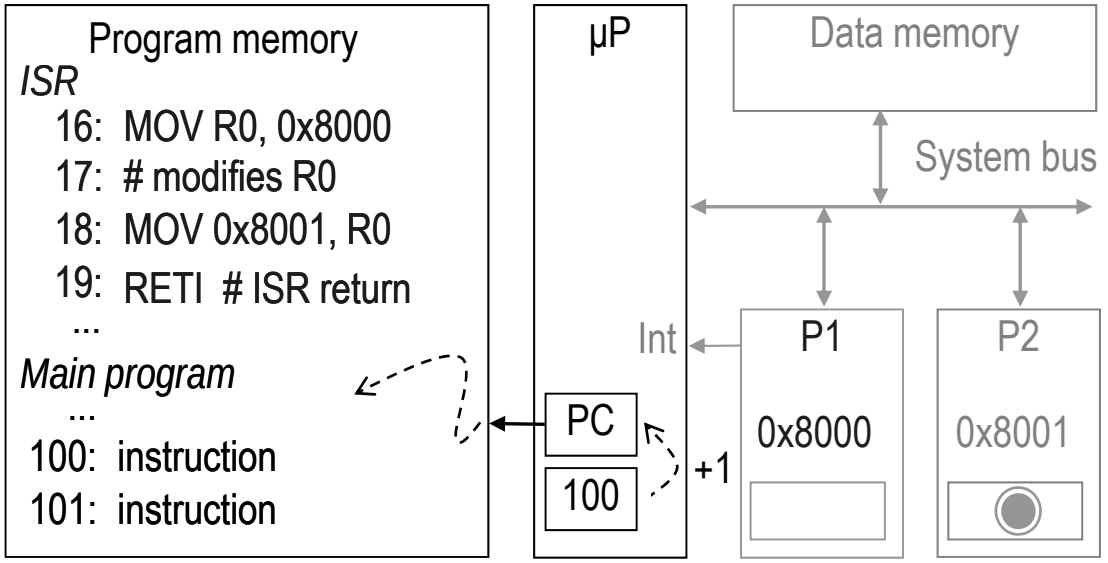
4(a): The ISR reads data from 0x8000,
modifies the data,
and writes the resulting data to 0x8001.

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



Interrupt-driven I/O using fixed ISR location (V)

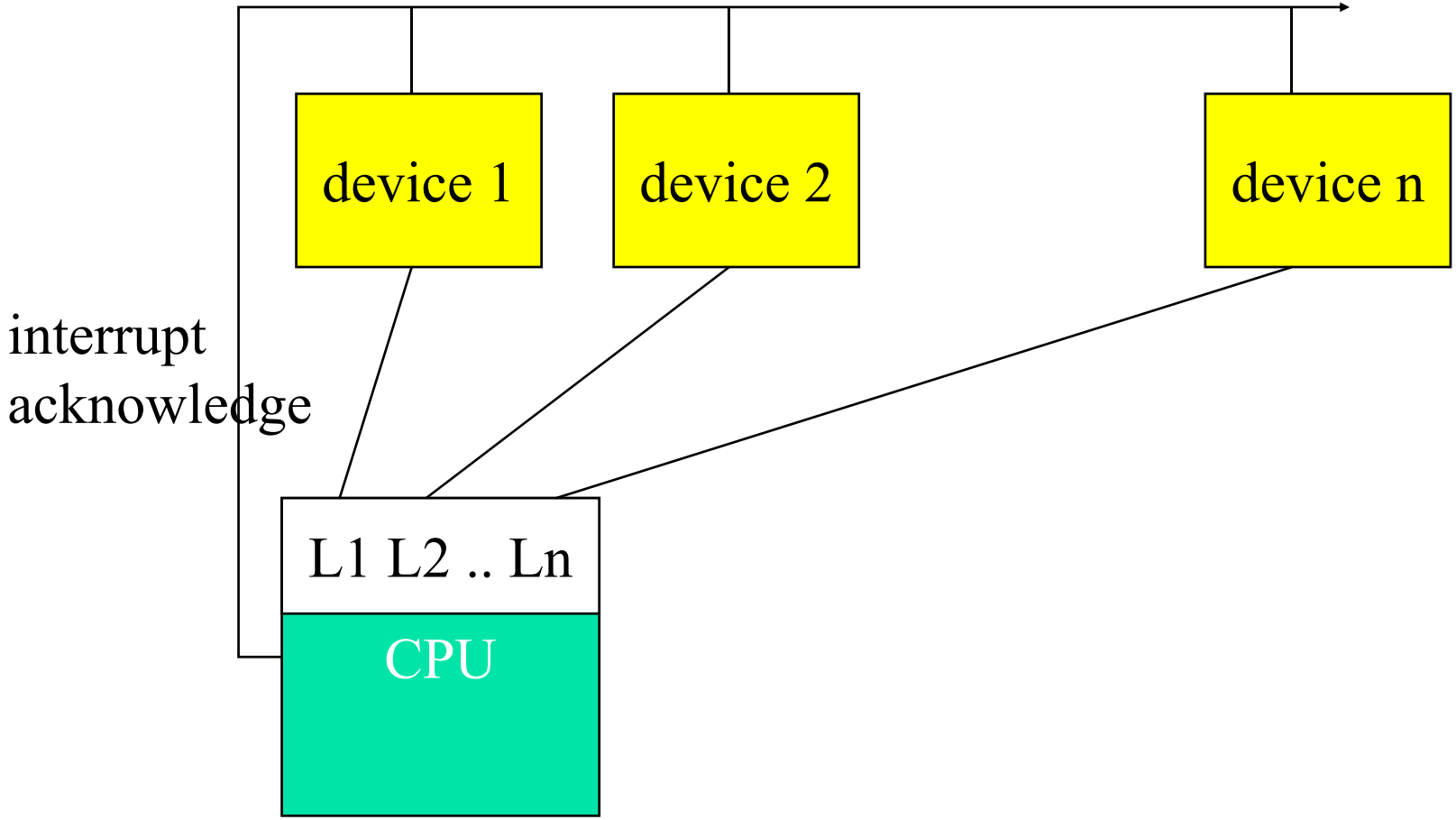
5: The ISR returns,
thus restoring PC to 100+1=101,
where μP resumes executing.



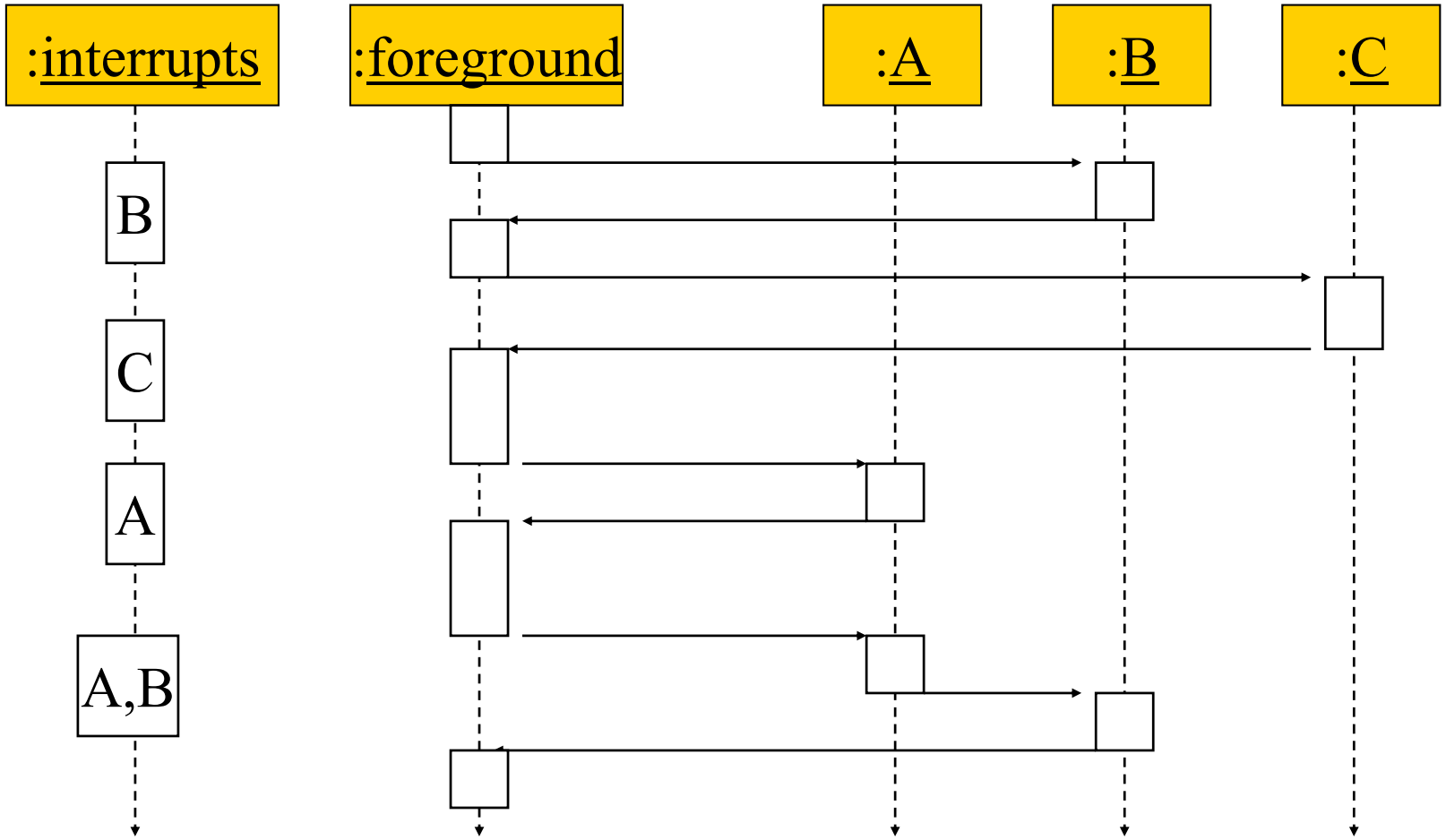
Priorities and Vectors

- Two mechanisms allow us to make interrupts more specific:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Mechanisms are orthogonal: most CPUs provide both.

Prioritized Interrupts



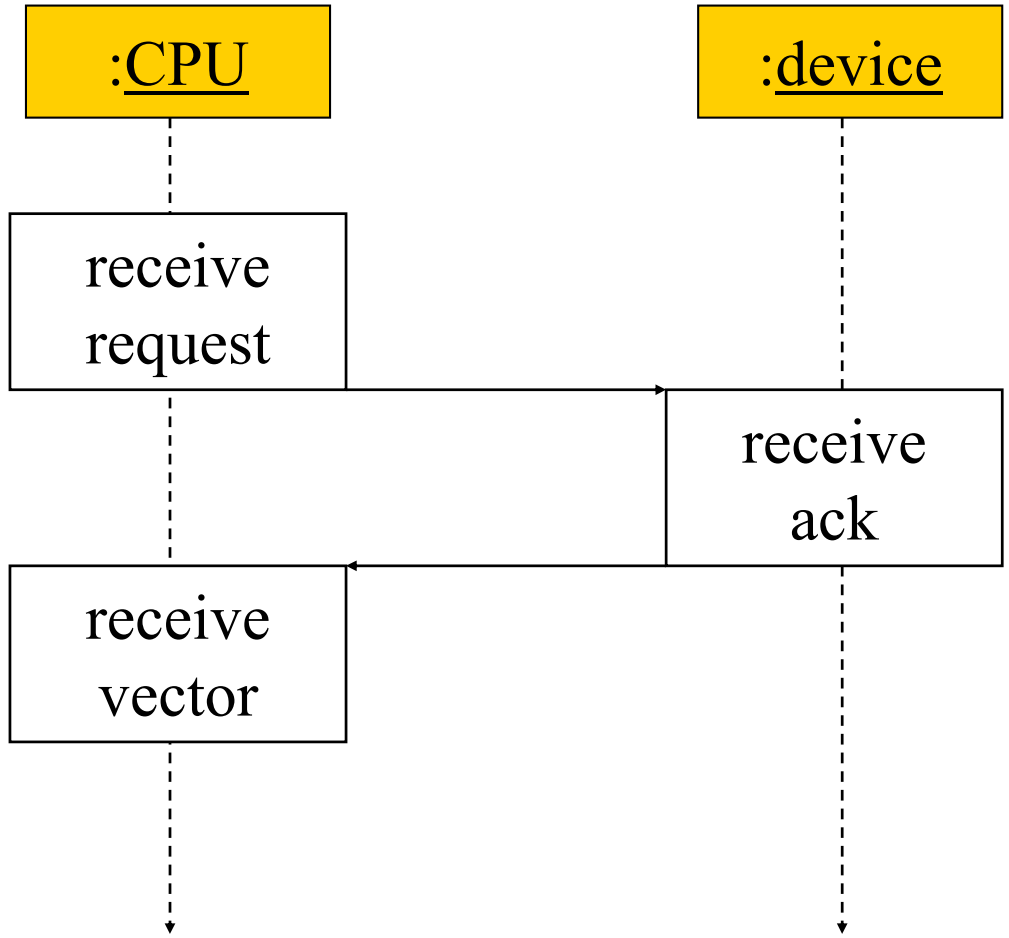
Example: Prioritized I/O



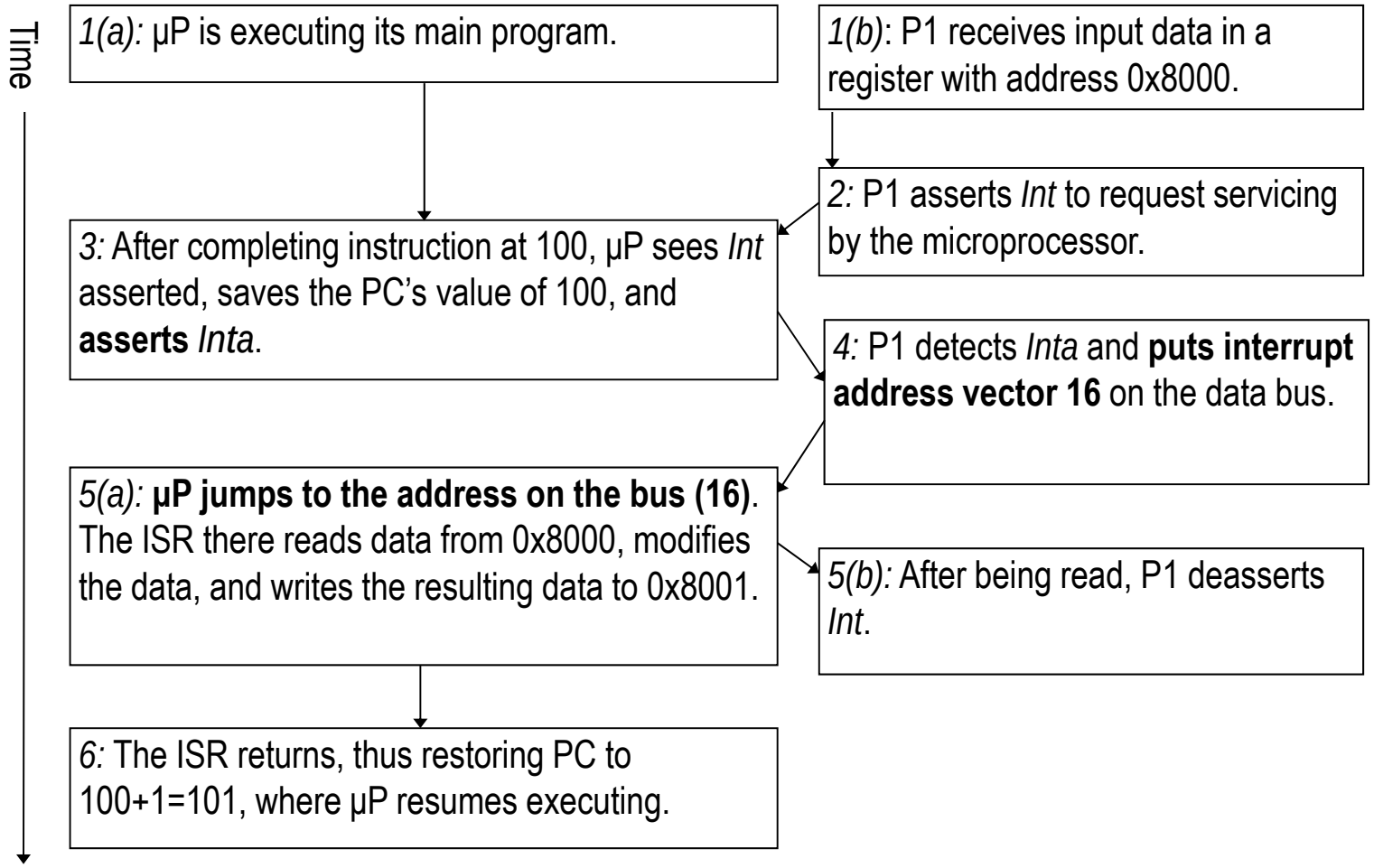
Vectored Interrupts

- What is 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***
    - Allow different devices to be handled by different code.
    - Peripheral must provide the address
    - Common when microprocessor has multiple peripherals connected by a system bus
  - Compromise: ***Interrupt address table***

Interrupt Vector Acquisition



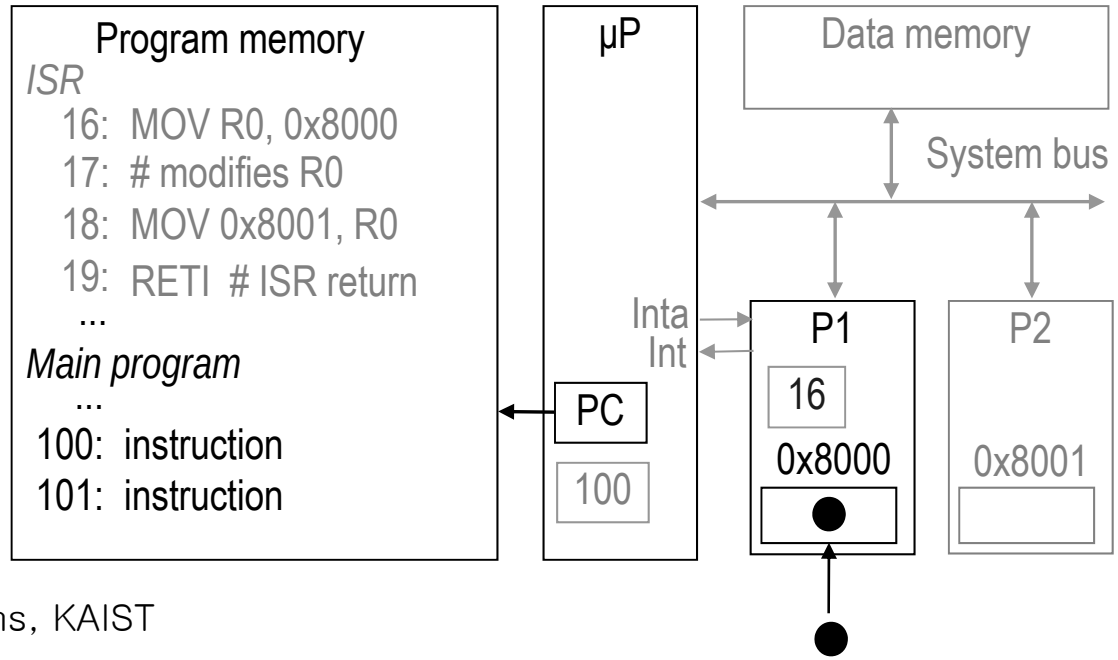
Interrupt-driven I/O using vectored interrupt



Interrupt-driven I/O using vectored interrupt (I)

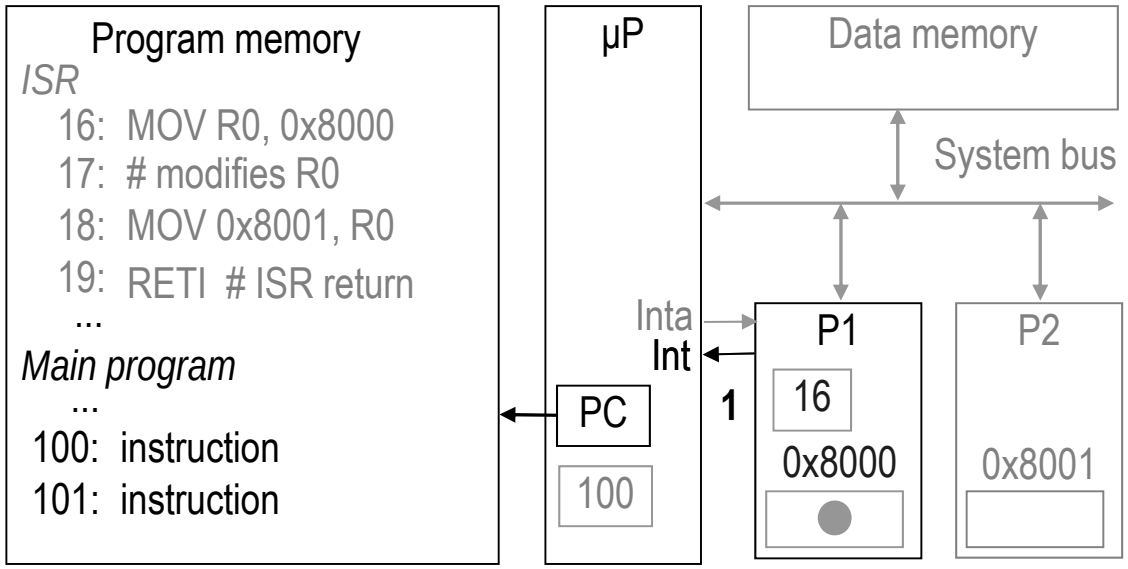
1(a): P is executing its main program

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



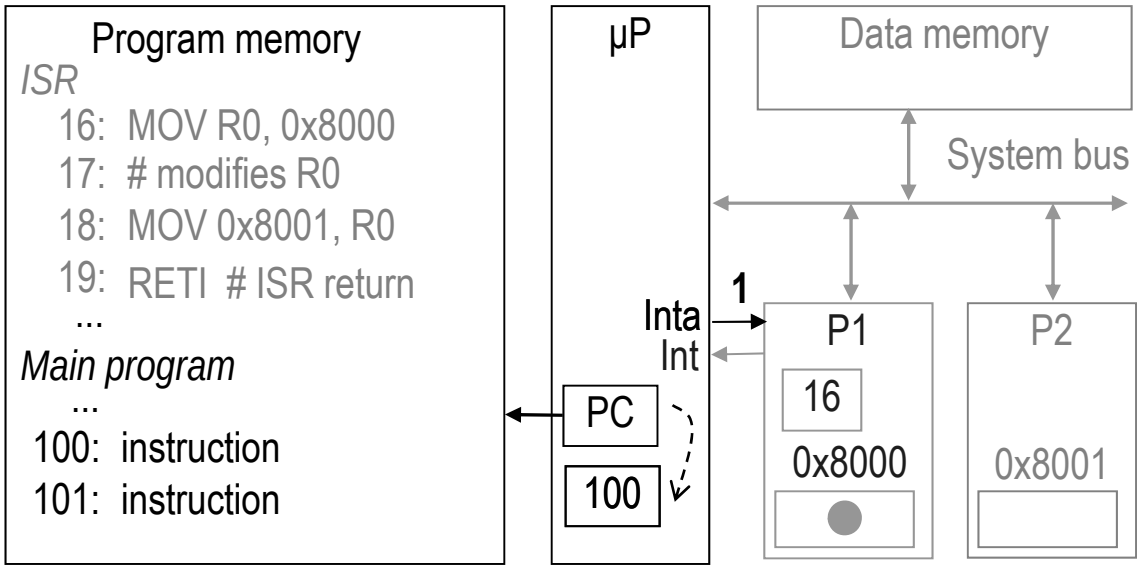
Interrupt-driven I/O using vectored interrupt (II)

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



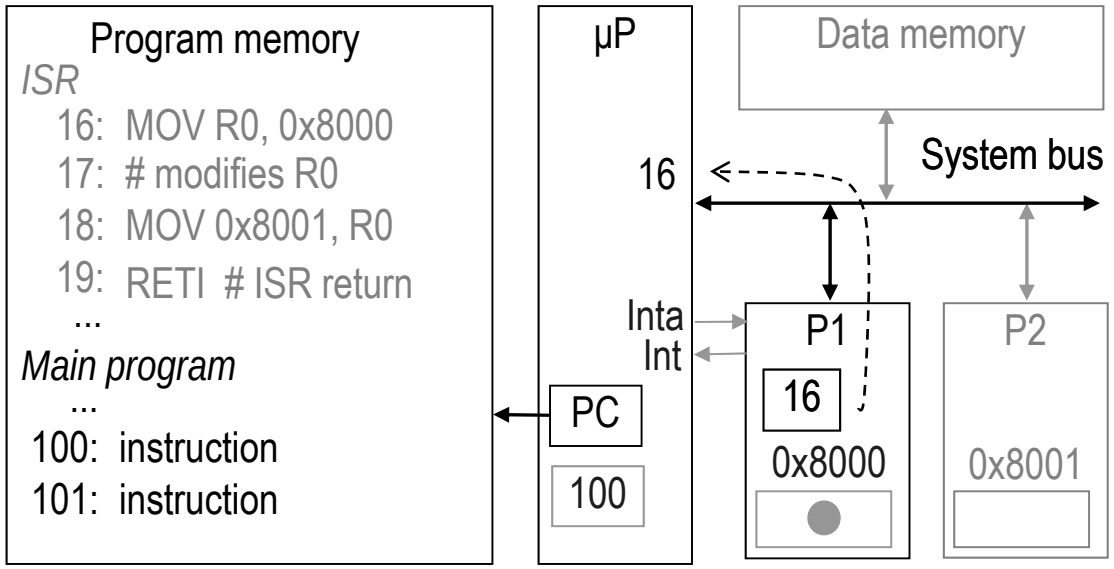
Interrupt-driven I/O using vectored interrupt (III)

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 (IV)

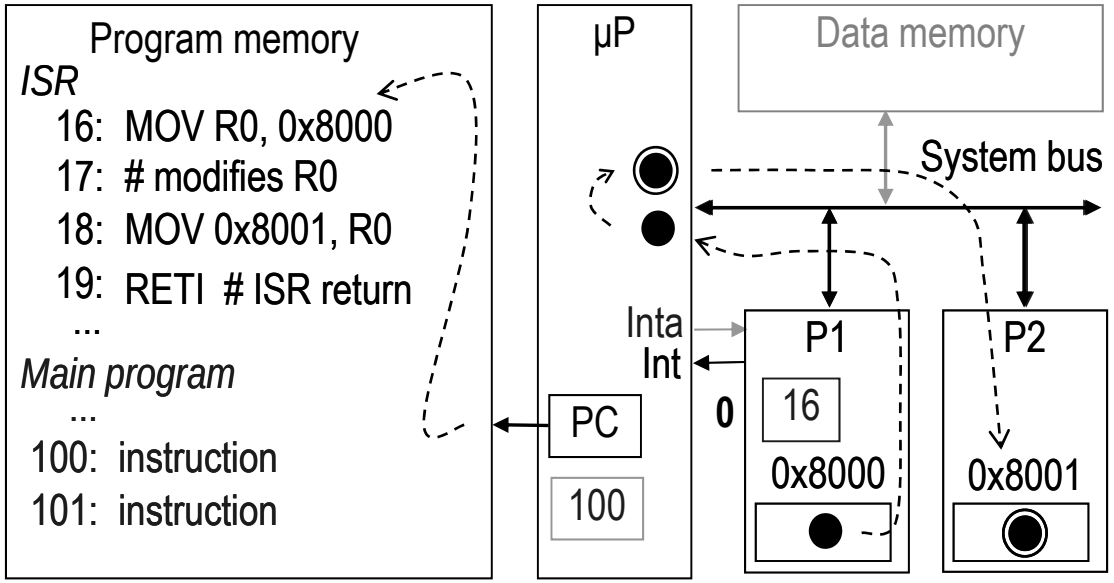
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-driven I/O using vectored interrupt (V)

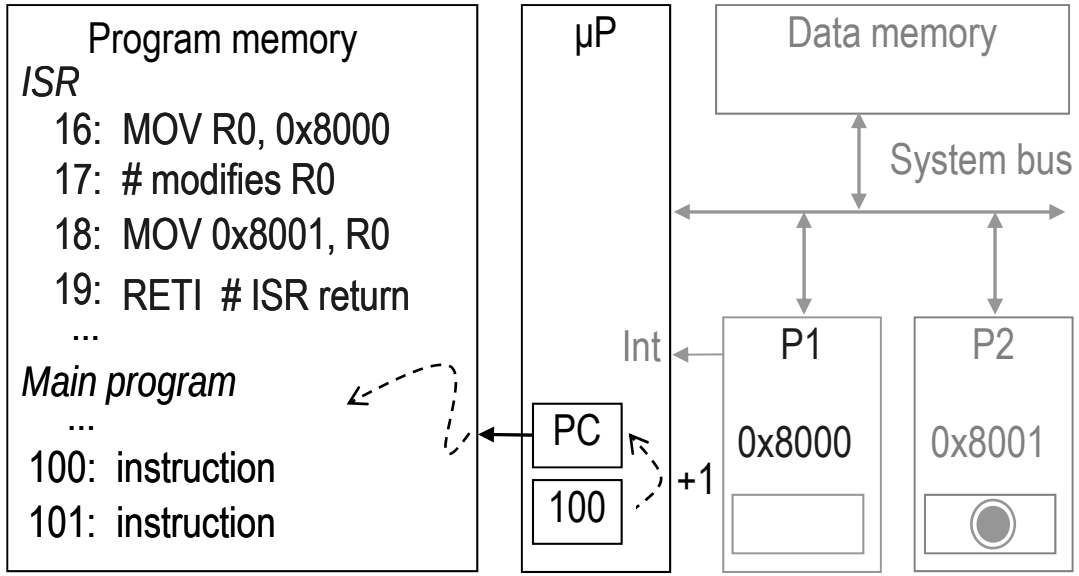
5(a): PC jumps to the address on the bus (16).
The ISR there reads data from 0x8000,
modifies the data, and
writes the resulting data to 0x8001.

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



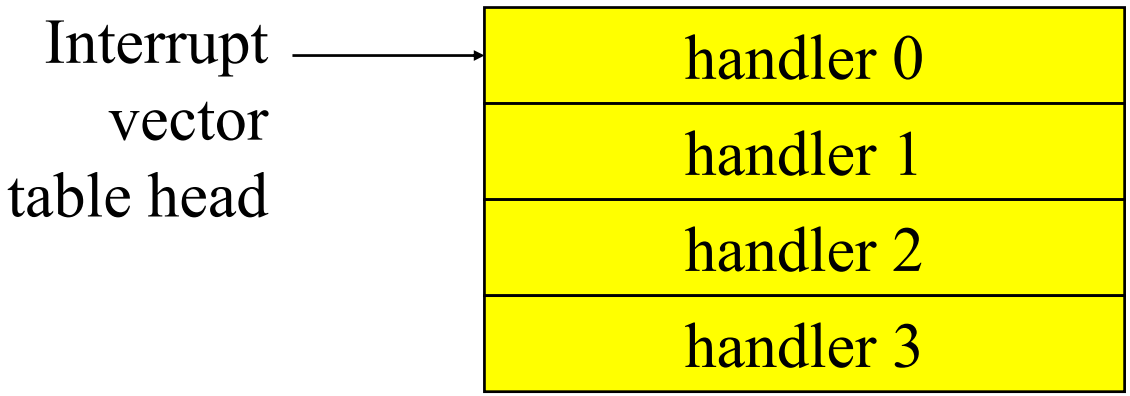
Interrupt-driven I/O using vectored interrupt (VI)

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



Interrupt Address Table

- Compromise between fixed and vectored interrupts
  - One interrupt pin
  - Table in memory holding ISR addresses (maybe 256 words)
  - Peripheral doesn't provide ISR address, but rather index into table
    - Fewer bits are sent by the peripheral
    - Can move ISR location without changing peripheral
- Interrupt vector table:



Additional Interrupt Issues

Maskable vs. non-maskable interrupts

- Maskable interrupt:
  - programmer can set bit that causes processor to ignore interrupt
    - Important when in the middle of time-critical code
- Non-maskable interrupt (NMI):
  - A separate interrupt pin that can't be masked
    - Highest-priority, never masked.
  - Typically reserved for drastic situations, like power failure
    - Requiring immediate backup of data to non-volatile memory.

Additional Interrupt Issues (II)

- **Register saving** before jumping to ISR
  - Some microprocessors treat jump same as call of any subroutine
    - Complete state saved (PC, registers) – may take hundreds of cycles
  - Others only save partial state, like PC only
    - Thus, ISR must not modify registers, or else must save them first
    - Assembly-language programmer must be aware of which registers stored
    - Ex. FIQ in ARM

Sources of Interrupt Overhead

- Handler execution time.
- Interrupt mechanism overhead.
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

Debugging Interrupt Code

- What if you forget to change registers in ISR using assembly language program?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Interrupt in C Language on Linux Module

```
// In init_module()
// Define ISR
request_irq(IRQ#, ISR_name, ... );
```

```
// In module program
// Interrupt Service Routine
void ISR_name(int irq, void *dev_id, struct pt_regs *regs)
{
    // ISR body
    // Check status register (Optional: to make sure)
    // Input or output data register.
}
```

References

- [1] Frank Vahid, “Embedded system design: A unified hardware/software introduction”, John Wiley & Sons, 2002.