Embedded Systems Design: A Unified Hardware/Software Introduction

Interfacing

ESD_Cap6 --

Outline

- Interfacing basics
- Microprocessor interfacing
  - I/O Addressing
  - Interrupts
  - Direct memory access
- Arbitration
- Bus hierarchy

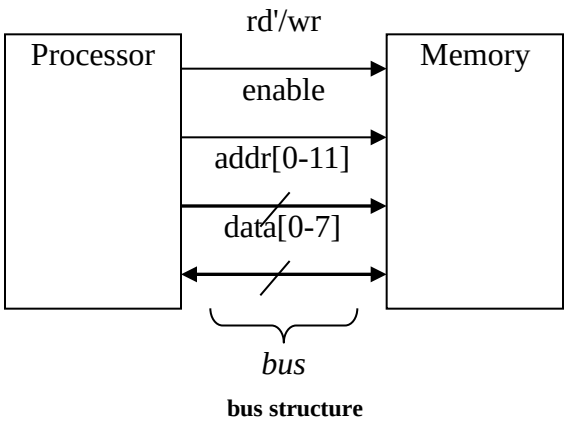
Introduction

- Embedded system functionality aspects
  - Processing
    - Transformation of data
    - Implemented using processors
  - Storage
    - Retention of data
    - Implemented using memory
  - Communication
    - Transfer of data between processors and memories
    - Mainly implemented using buses
    - The basic needed knowledge is related to *interfacing*

Interfacing Basics

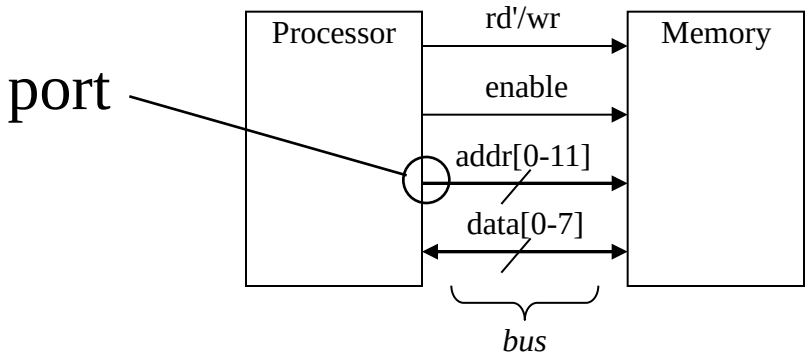
A simple bus

- Wires:
  - Uni-directional or bi-directional
    - Represented by lines
      - One line may represent multiple wires
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication



bus structure

Ports



- Conducting device on periphery
  - Connect the device with other ones
    - e.g. Connect bus to processor or memory
- Often referred to as a *pin*
  - Actual pins on periphery of IC package that plug into socket on printed-circuit board
  - Sometimes metallic balls instead of pins
  - Today, metal “pads” connecting processors and memories within single IC
- Single wire or set of wires with single function
  - E.g., 12-wire address port

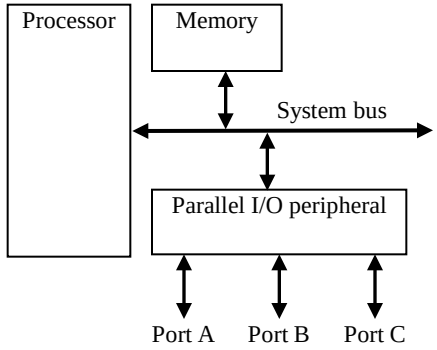
Microprocessor Interfacing

Microprocessor interfacing: I/O addressing

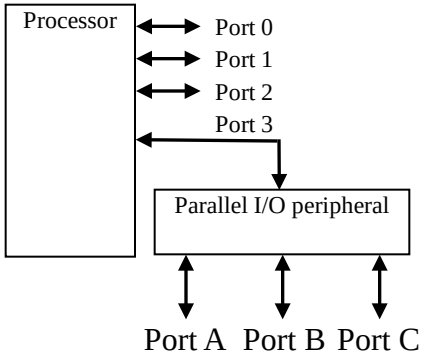
- A microprocessor communicates with other devices (memories or other processors) using some of its pins
  - Port-based I/O (Parallel I/O)
    - Processor has one or more N-bit ports
    - Processor's software reads and writes a port just like a register
    - E.g., P0 = 0xFF; v = P1.2; -- P0 and P1 are 8-bit ports
      - A sort of point-to-point connection with another device
  - Bus-based I/O
    - Processor has address, data and control ports that form a single bus
    - Communication protocol is built into the processor
    - A single instruction carries out the read or write protocol on the bus
      - A sort of multi-point connection by means of device addresses

Compromises/extensions

- Parallel I/O peripheral
  - When processor only supports bus-based I/O but parallel I/O needed
  - Each port on peripheral connected to a register within peripheral that is read/written by the processor
- Extended parallel I/O
  - When processor supports port-based I/O but more ports needed
  - One or more processor ports interface with parallel I/O peripheral extending total number of ports available for I/O
  - e.g., extending 4 ports to 6 ports in figure



Adding parallel I/O to a bus-based I/O processor



Extended parallel I/O

Types of bus-based I/O: 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

Memory-mapped I/O vs. Standard I/O

- Memory-mapped I/O
  - Requires no special instructions
    - Assembly instructions involving memory like MOV and ADD work with peripherals as well
    - Standard I/O requires special instructions (e.g., IN, OUT) to move data between peripheral registers and memory
- Standard I/O
  - No loss of memory addresses to peripherals
  - Simpler address decoding logic in peripherals possible
    - When number of peripherals much smaller than address space then high-order address bits can be ignored
      - smaller and/or faster comparators

Interrupts

Microprocessor interfacing: interrupts

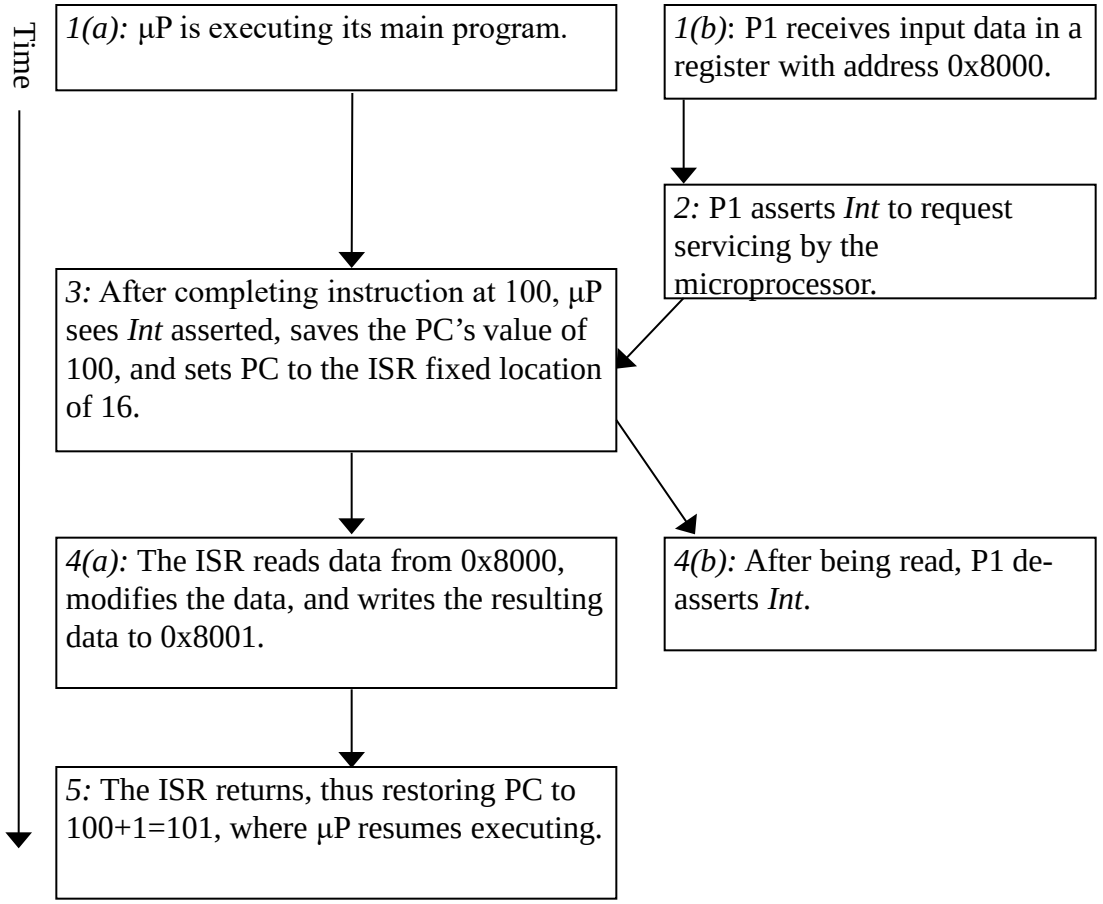
- Suppose a peripheral intermittently receives data, which must be serviced by the processor
  - The processor can *poll* the peripheral regularly to see if data has arrived – wasteful
  - The peripheral can *interrupt* the processor when it has data
- Requires an extra pin or pins: Int
  - If Int is 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!

Microprocessor interfacing: 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
    - Peripheral must provide the address
    - Common when microprocessor has multiple peripherals connected by a system bus
  - Compromise: interrupt address table

The same concepts are involved in reset and exceptions (see Extra)!

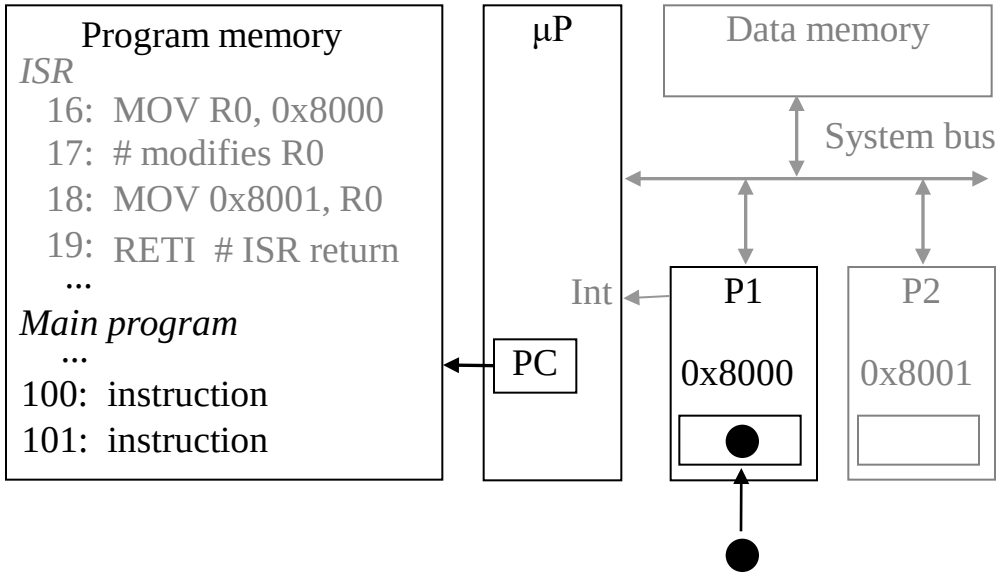
Interrupt-driven I/O using fixed ISR location



Interrupt-driven I/O using fixed ISR location

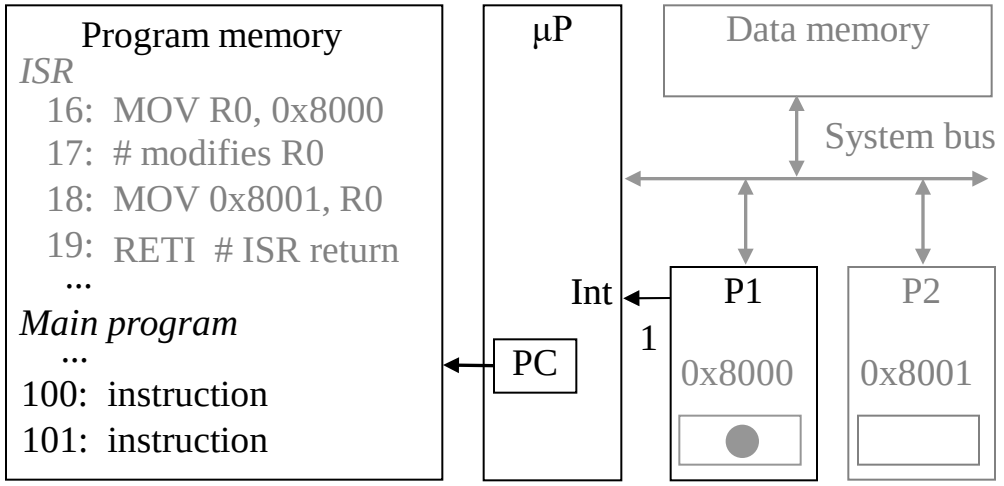
1(a): μP is executing its main program

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



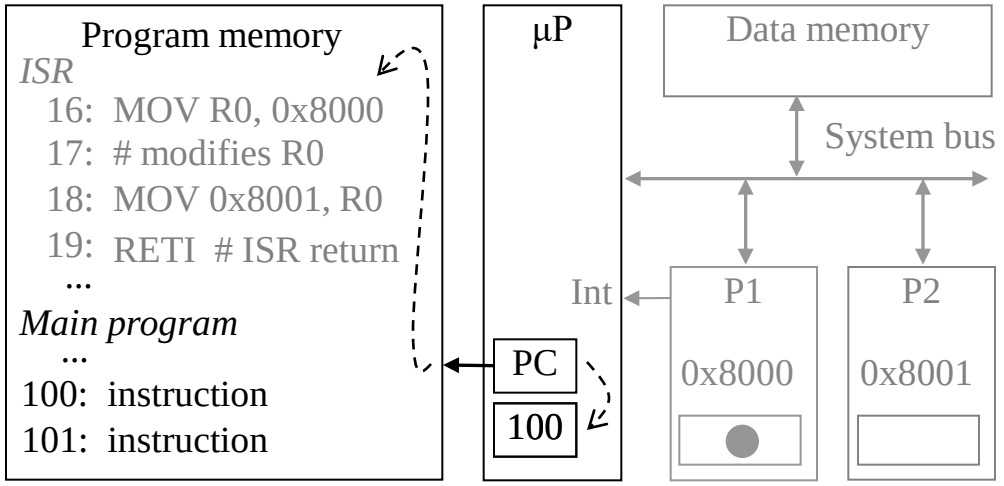
Interrupt-driven I/O using fixed ISR location

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



Interrupt-driven I/O using fixed ISR location

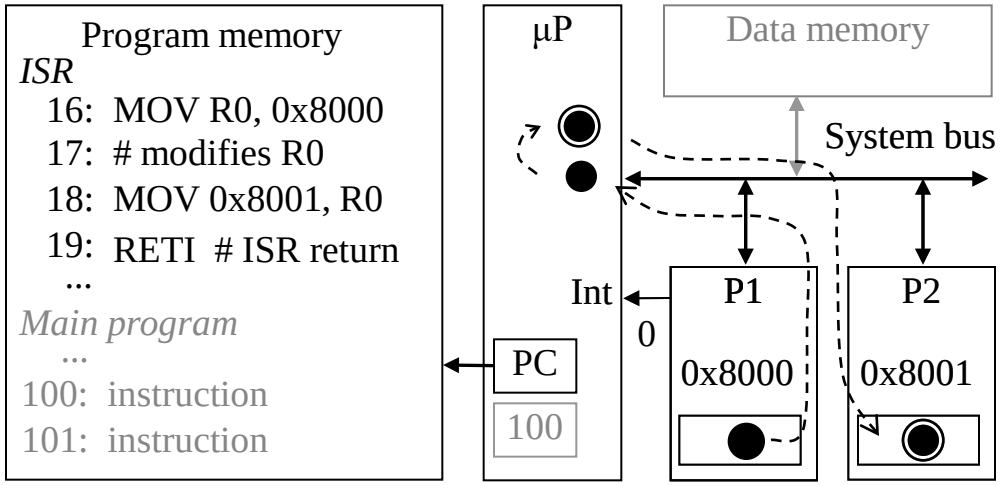
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location

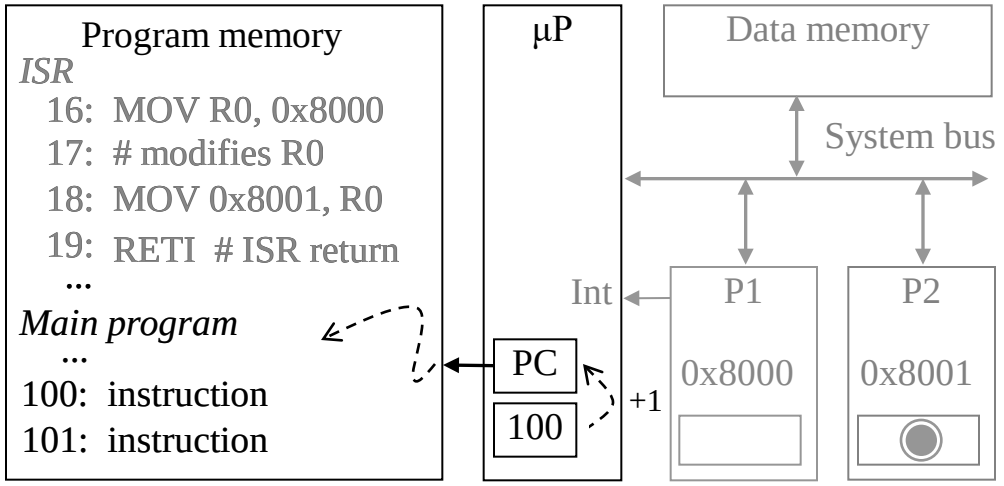
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

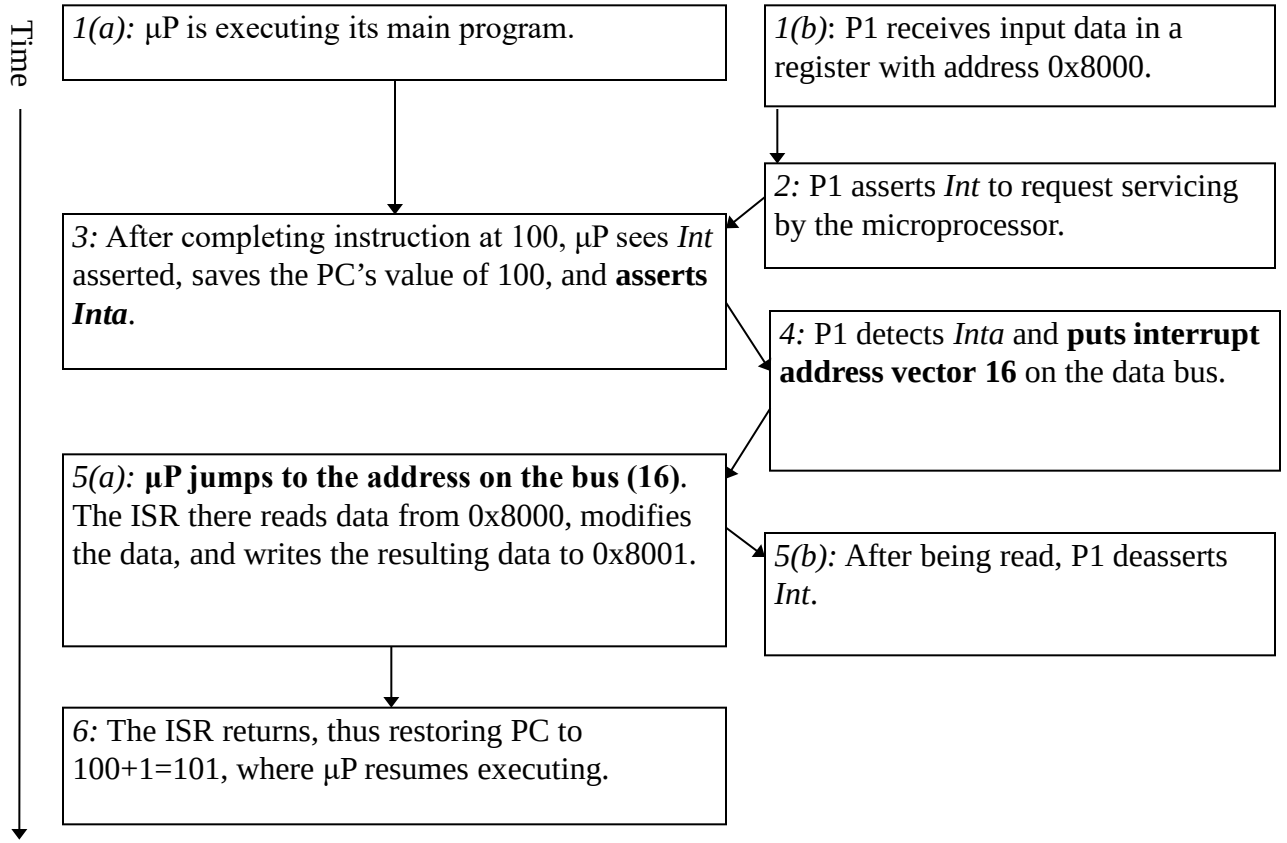


Interrupt-driven I/O using fixed ISR location

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



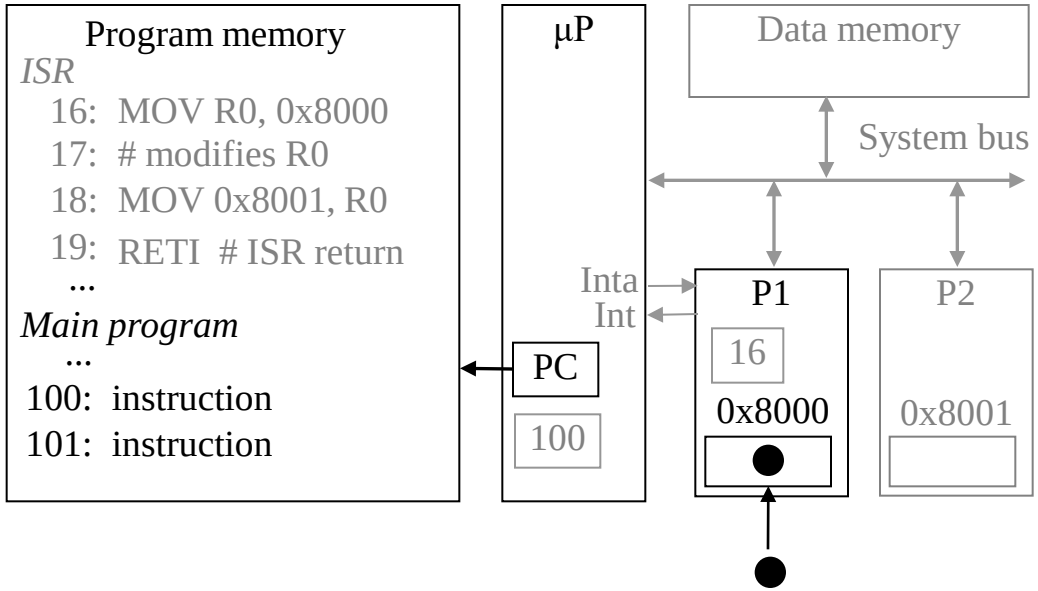
Interrupt-driven I/O using vectored interrupt



Interrupt-driven I/O using vectored interrupt

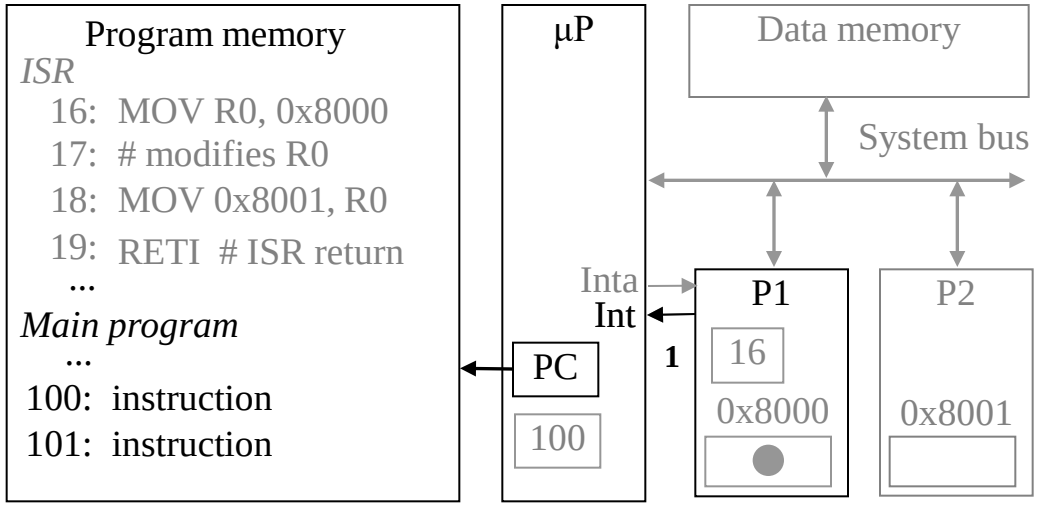
1(a): P is executing its main program

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



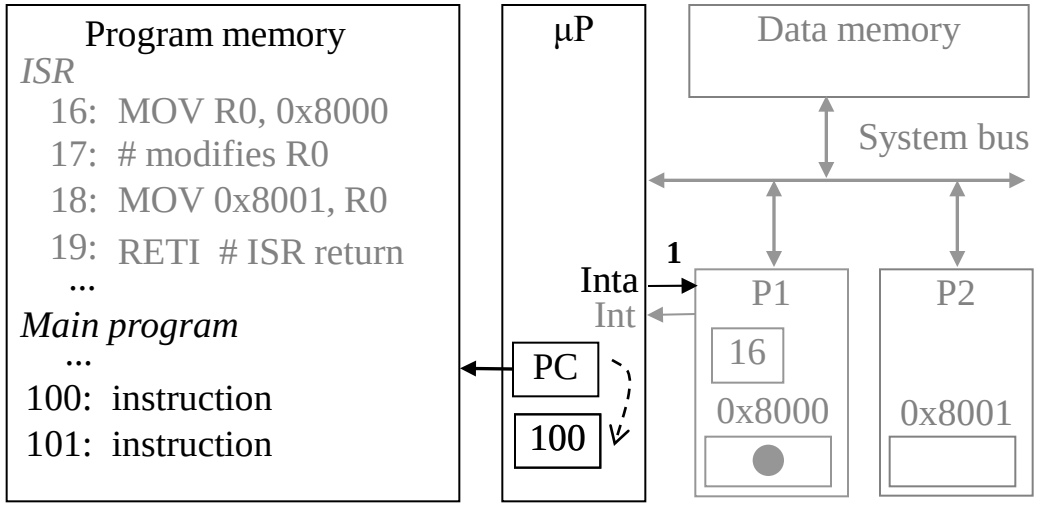
Interrupt-driven I/O using vectored interrupt

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



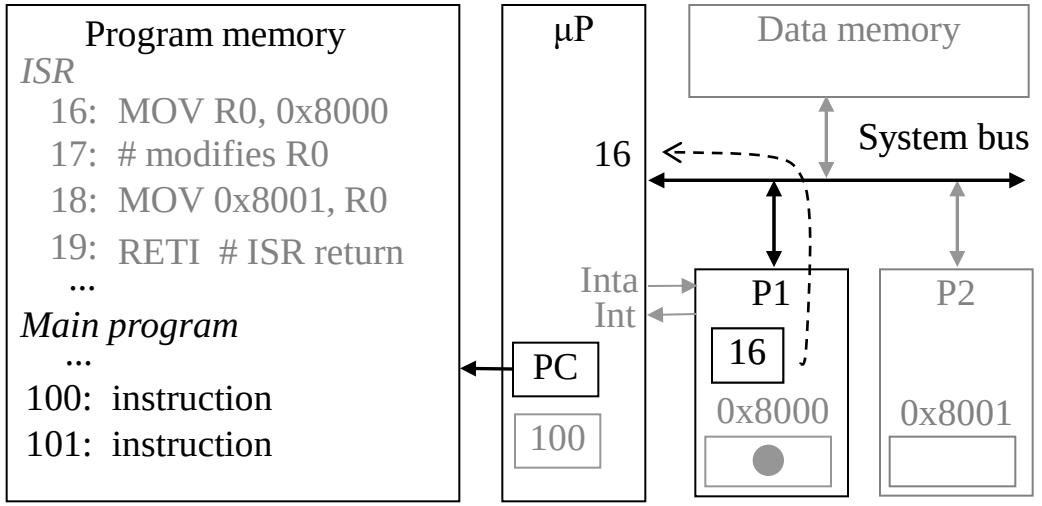
Interrupt-driven I/O using vectored interrupt

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

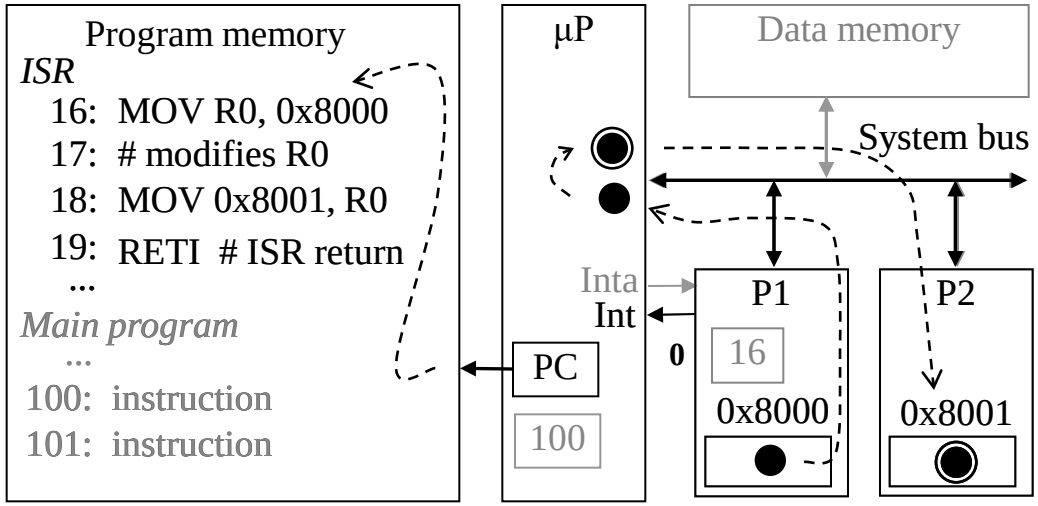
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-driven I/O using vectored interrupt

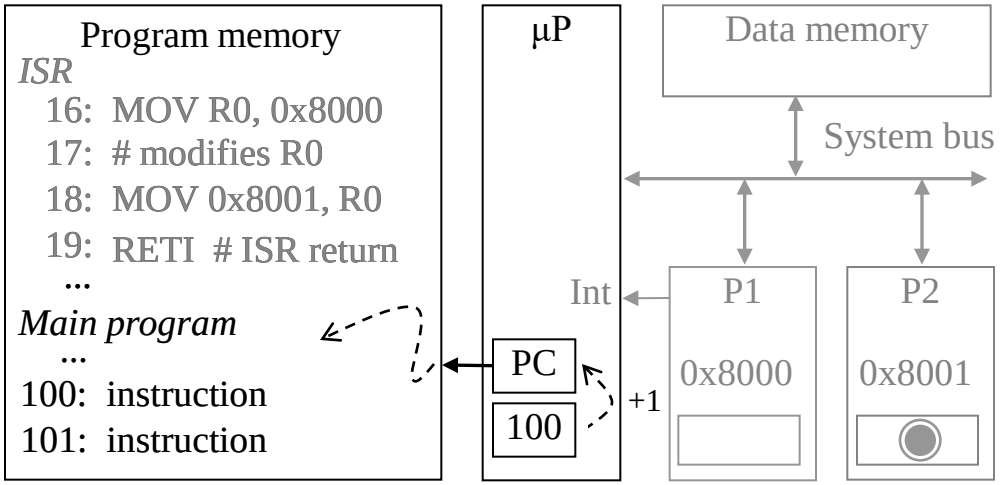
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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



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

Additional interrupt issues

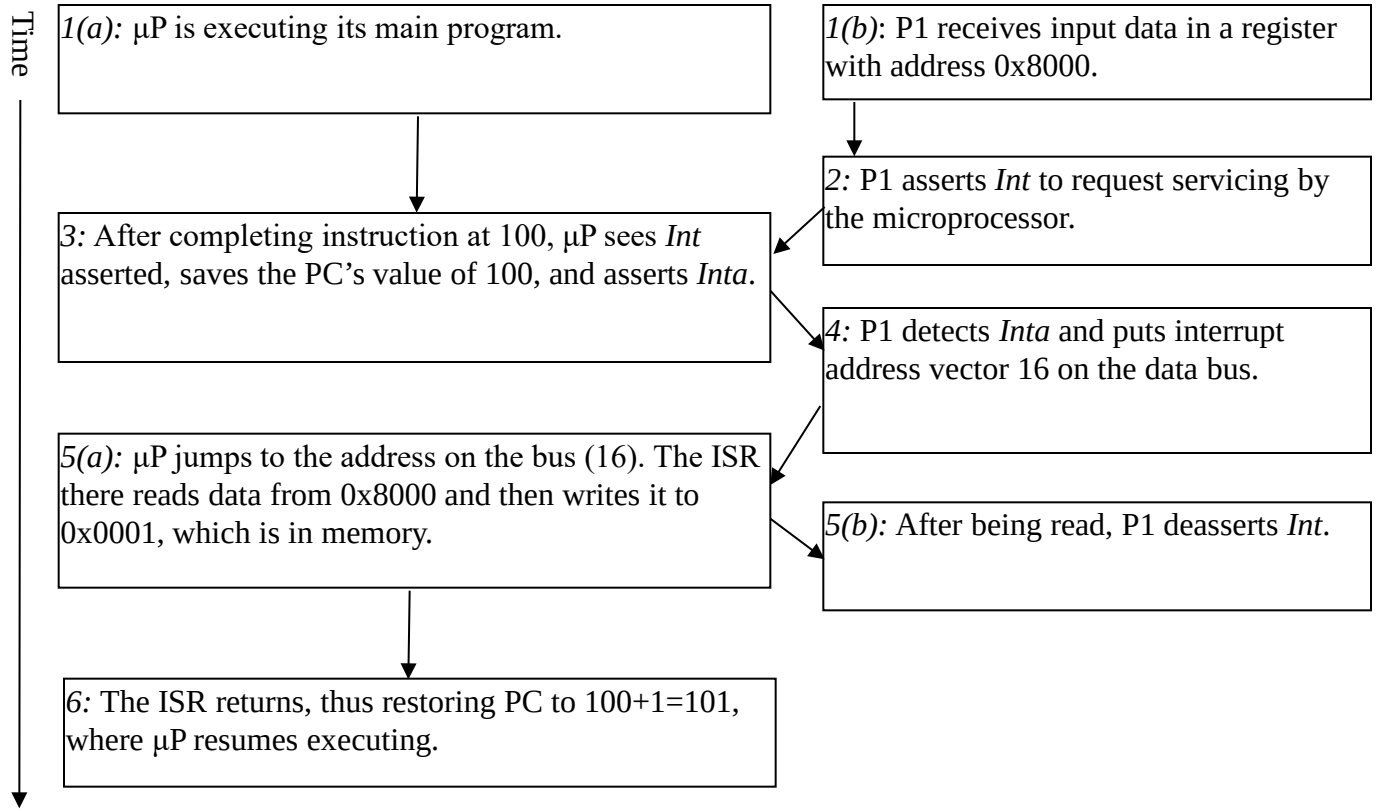
- Maskable vs. non-maskable interrupts
  - Maskable: programmer can set bit that causes processor to ignore interrupt
    - Important when in the middle of time-critical code and to manage interrupt nesting!
  - Non-maskable: a separate interrupt pin that can't be masked
    - Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory
- Jump 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

Direct Memory Access

Direct memory access

- Buffering
  - Temporarily storing data in memory before processing
  - Data accumulated in peripherals commonly buffered
- Microprocessor could handle this with ISR
  - Storing and restoring microprocessor state inefficient
  - Regular program must wait
- DMA controller more efficient
  - Separate single-purpose processor
  - Microprocessor relinquishes control of system bus to DMA controller
  - Microprocessor can meanwhile execute its regular program
    - No inefficient storing and restoring state due to ISR call
    - Regular program need not wait unless it requires the system bus
      - Harvard archictecture – processor can fetch and execute instructions as long as they don't access data memory – if they do, processor stalls

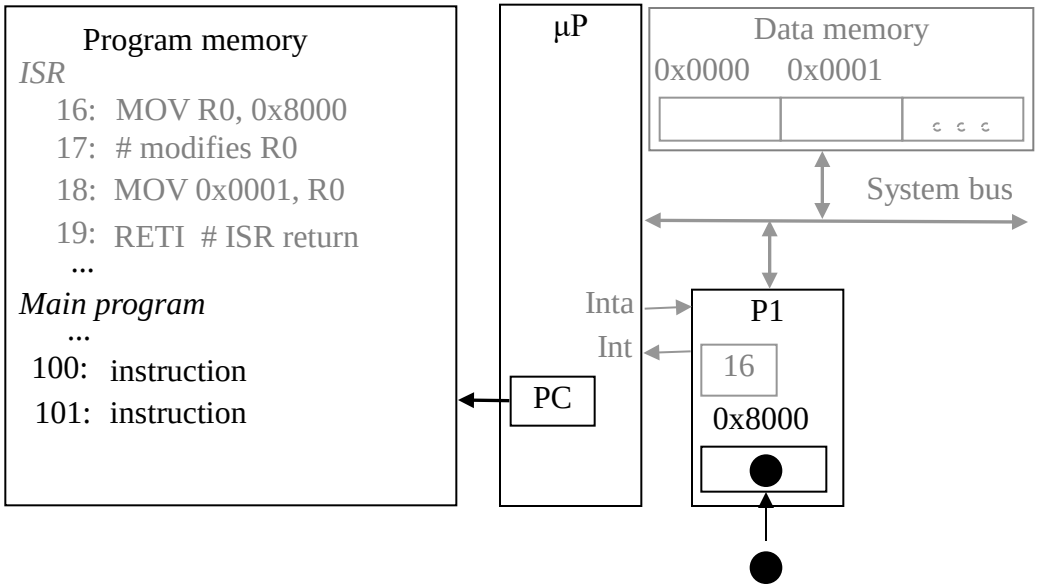
Peripheral to memory transfer *without* DMA, using vectored interrupt



Peripheral to memory transfer *without* DMA, using vectored interrupt

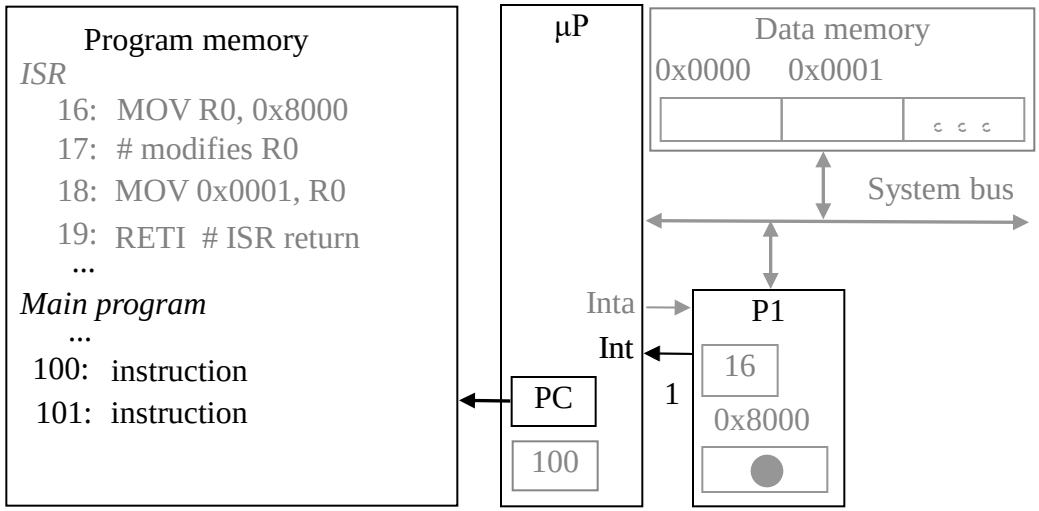
1(a): μP is executing its main program

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



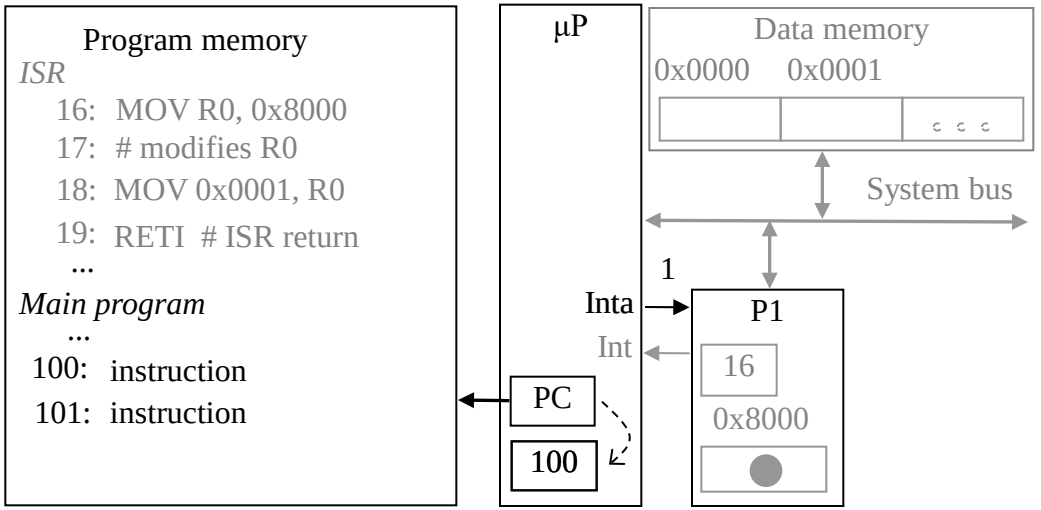
Peripheral to memory transfer *without* DMA, using vectored interrupt

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



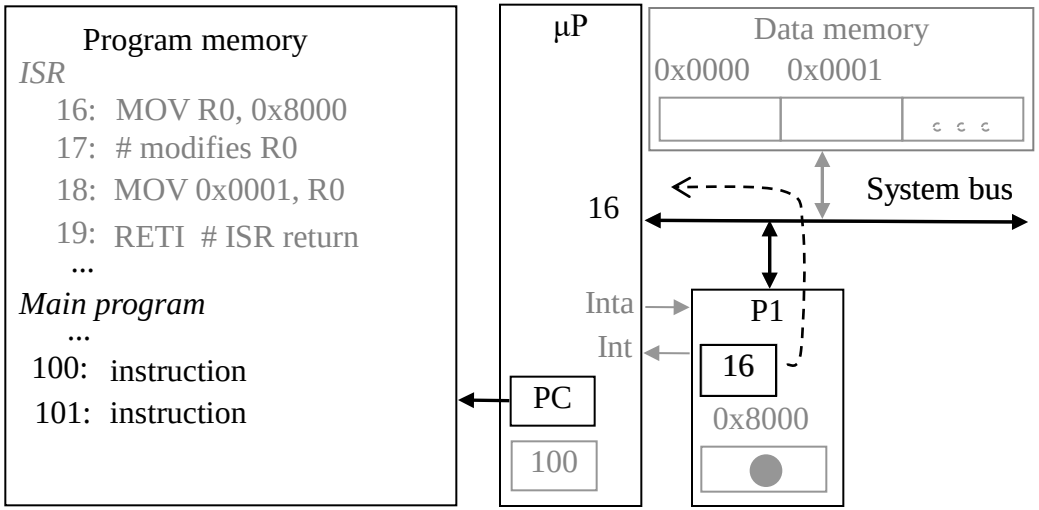
Peripheral to memory transfer *without* DMA, using vectored interrupt

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



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

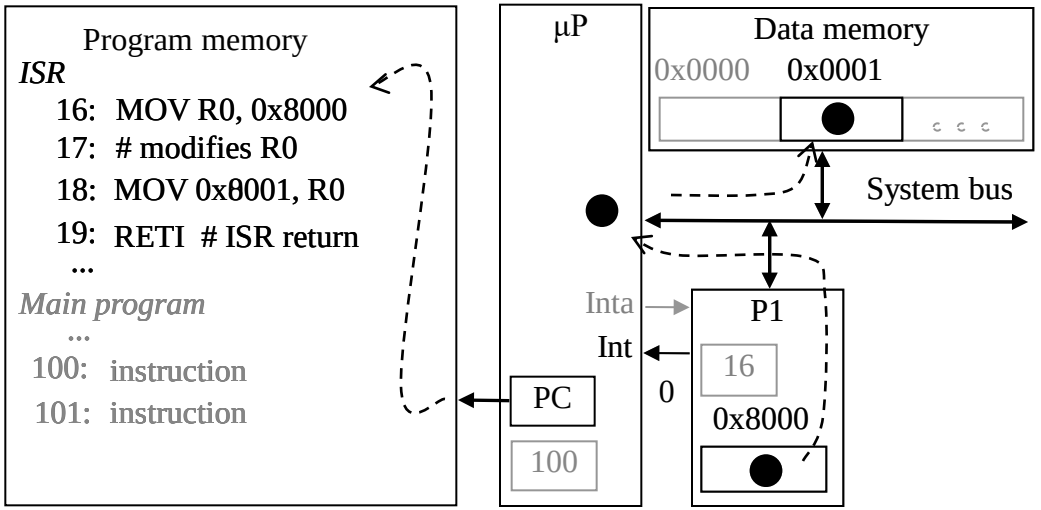
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

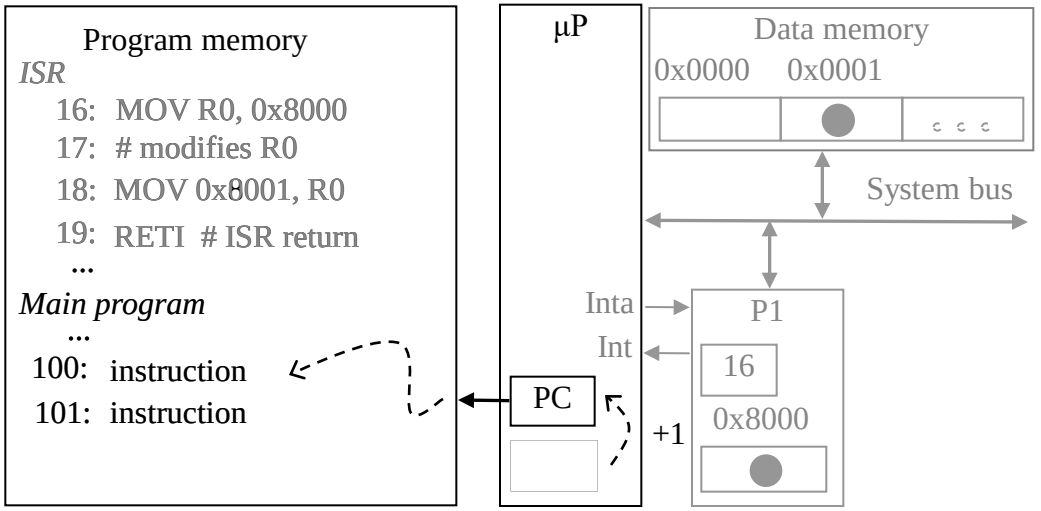
5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

5(b): After being read, P1 de-asserts *Int*.

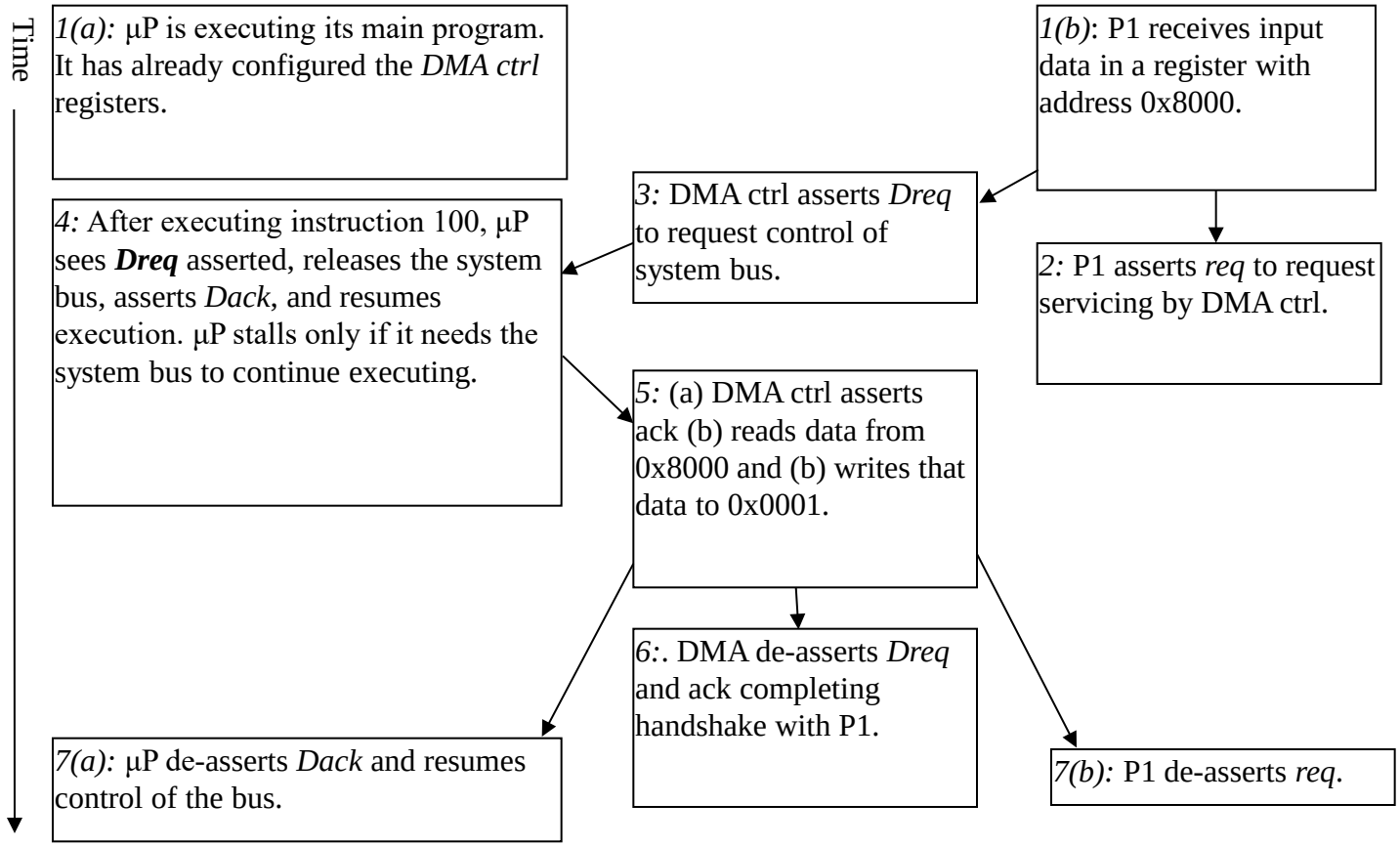


Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

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



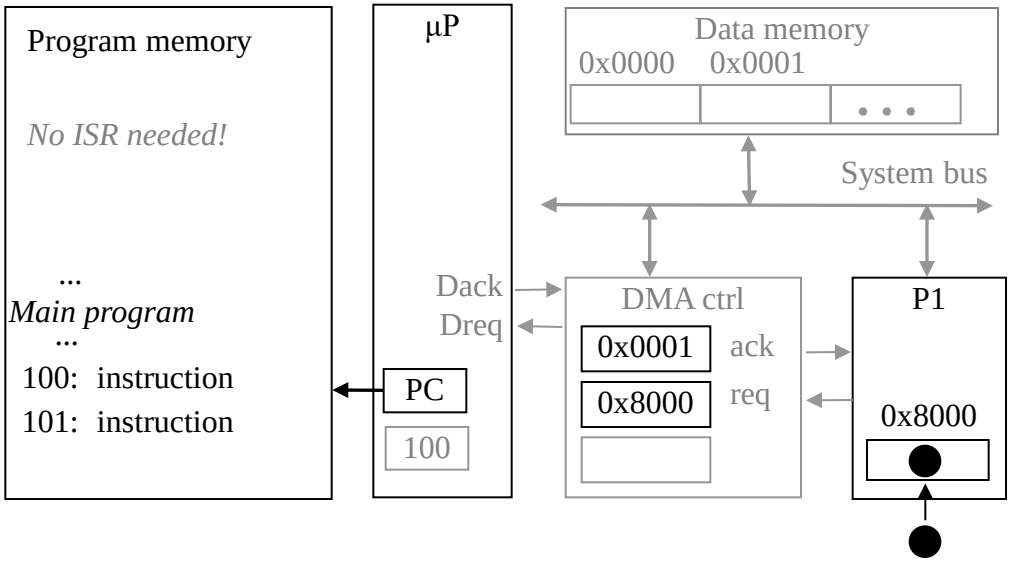
Peripheral to memory transfer with DMA



Peripheral to memory transfer with DMA (cont')

1(a): μP is executing its main program. It has already configured the DMA ctrl registers

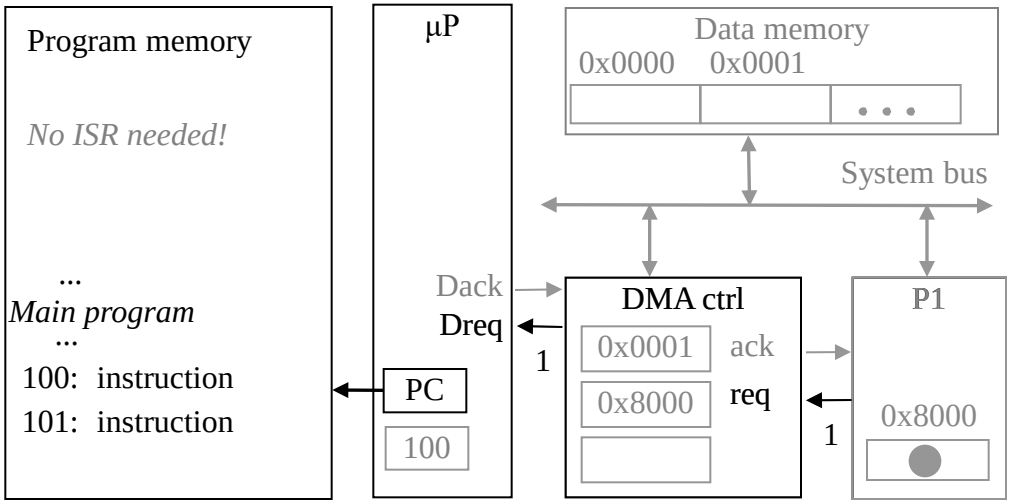
1(b): P1 receives input data in a register with address 0x8000.



Peripheral to memory transfer with DMA (cont')

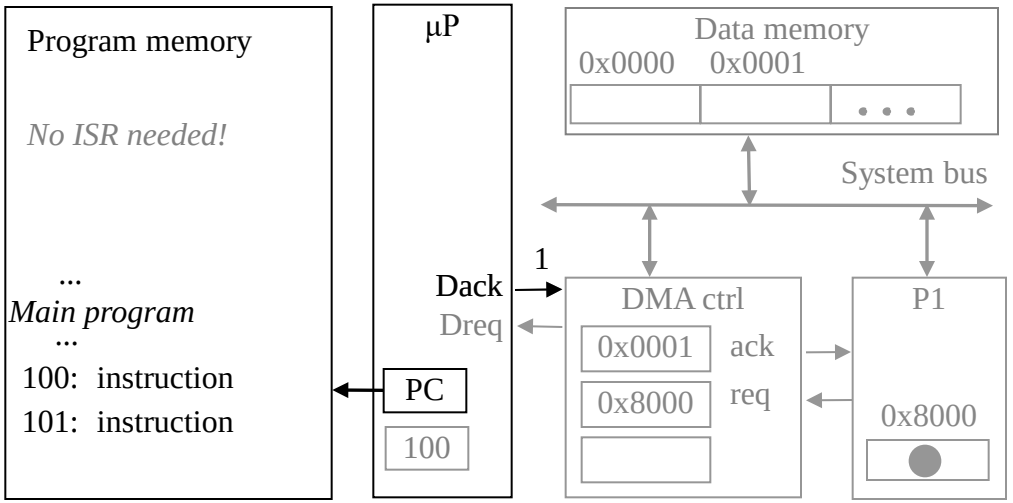
2: P1 asserts *req* to request servicing by DMA ctrl.

3: DMA ctrl asserts *Dreq* to request control of system bus



Peripheral to memory transfer with DMA (cont')

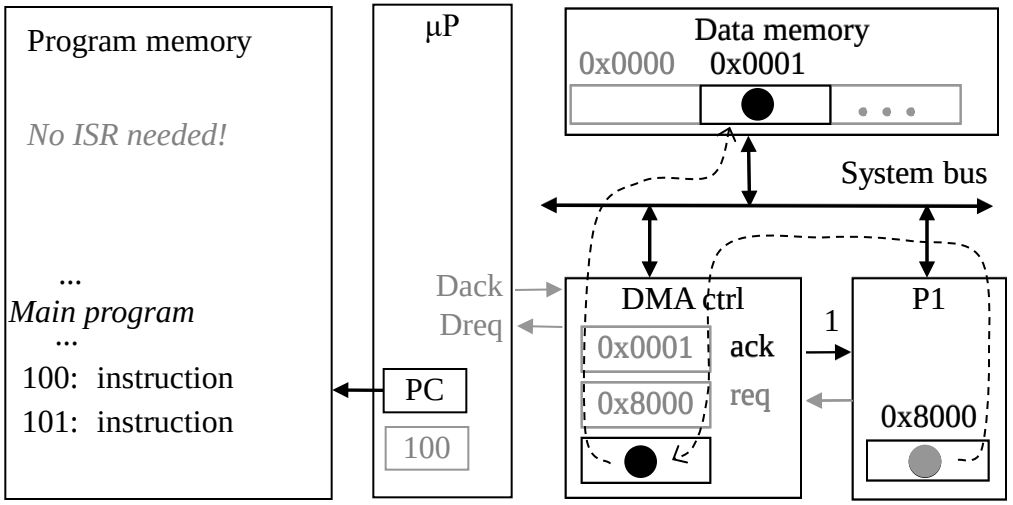
4: After executing instruction 100, μP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, μP stalls only if it needs the system bus to continue executing.



Peripheral to memory transfer with DMA (cont')

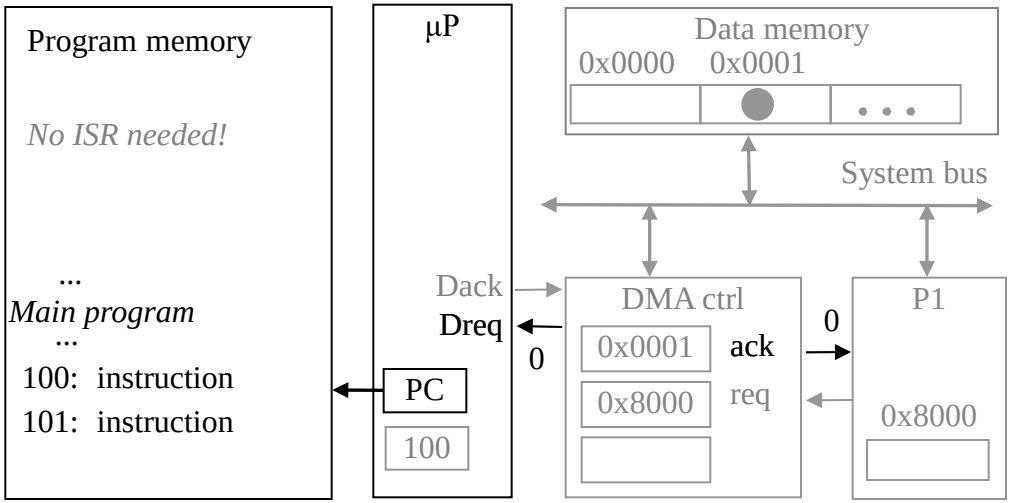
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)



Peripheral to memory transfer with DMA (cont')

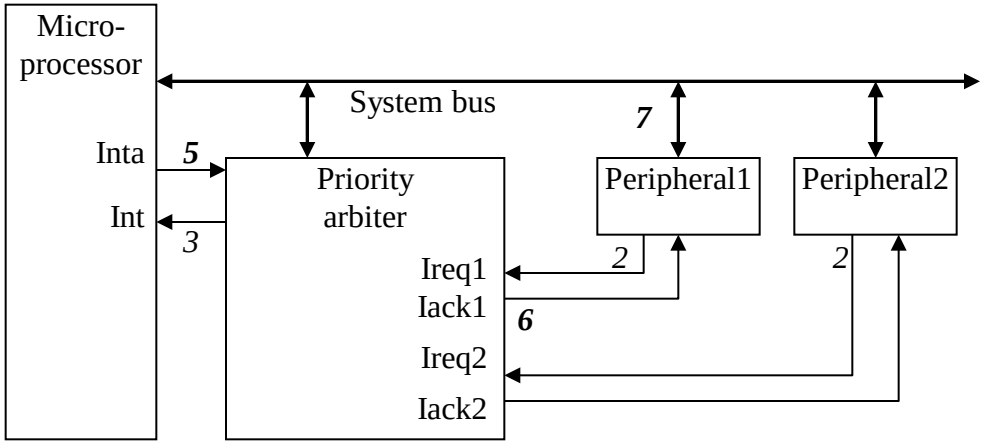
6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



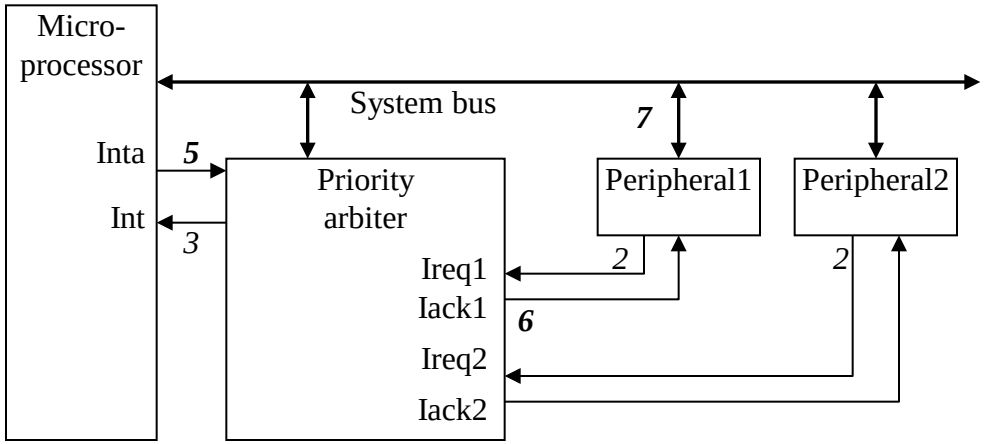
Arbitration

Arbitration: Priority arbiter

- Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller) simultaneously - which gets serviced first?
- Priority arbiter
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for configuration only



Arbitration using a priority arbiter



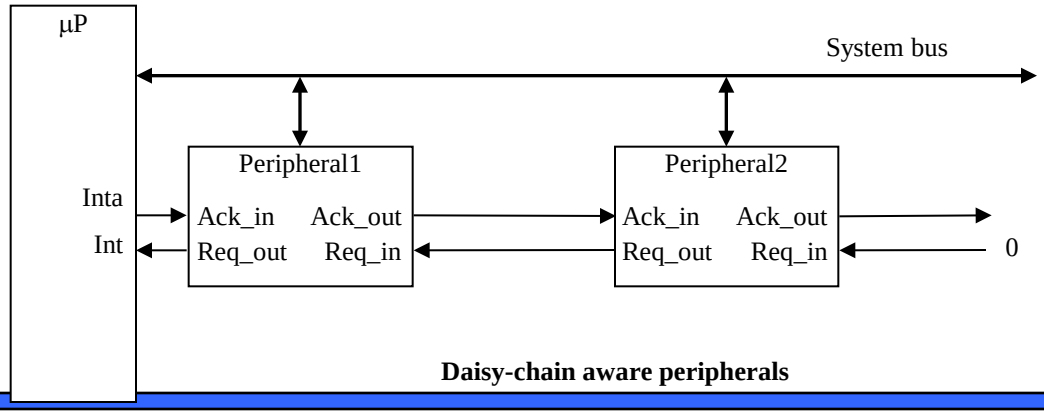
1. 1. Microprocessor is executing its program.
2. 2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. 3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. 4. Microprocessor stops executing its program and stores its state.
5. 5. Microprocessor asserts *Inta*.
6. 6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. 7. Peripheral1 puts its interrupt address vector on the system bus
8. 8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns
9. (and completes handshake with arbiter).
10. 9. Microprocessor resumes executing its program.

Arbitration: Priority arbiter

- Types of priority
  - Fixed priority
    - each peripheral has unique rank
    - highest rank chosen first with simultaneous requests
    - preferred when clear difference in rank between peripherals
  - Rotating priority (round-robin)
    - priority changed based on history of servicing
    - better distribution of servicing especially among peripherals with similar priority demands

Arbitration: Daisy-chain arbitration

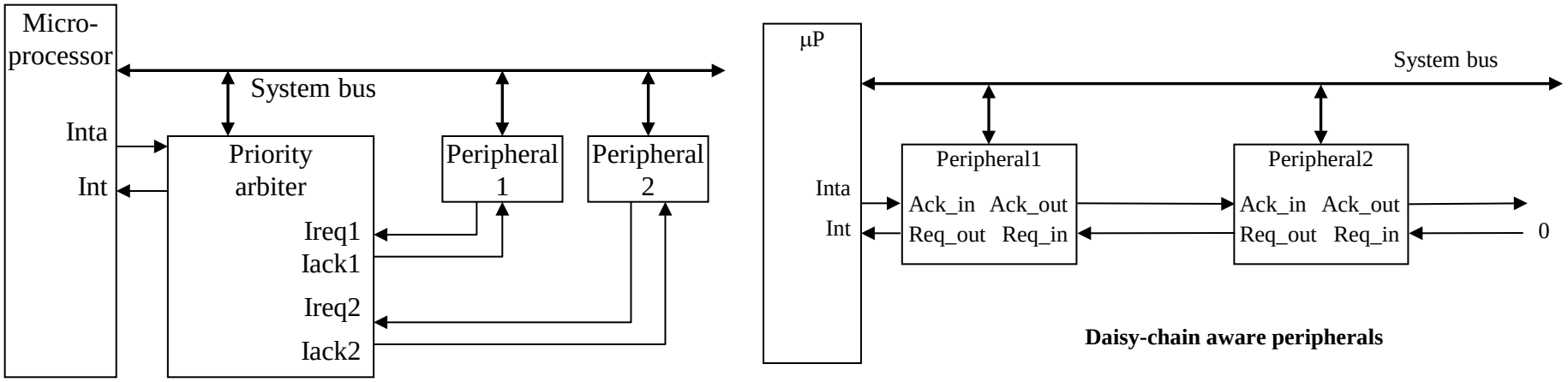
- Arbitration done by peripherals
  - Built into peripheral or external logic added
    - *req* input and *ack* output added to each peripheral
- Peripherals connected to each other in daisy-chain manner
  - One peripheral connected to resource, all others connected “upstream”
  - Peripheral’s *req* flows “downstream” to resource, resource’s *ack* flows “upstream” to requesting peripheral
  - Closest peripheral has highest priority



Daisy-chain aware peripherals

Arbitration: Daisy-chain arbitration

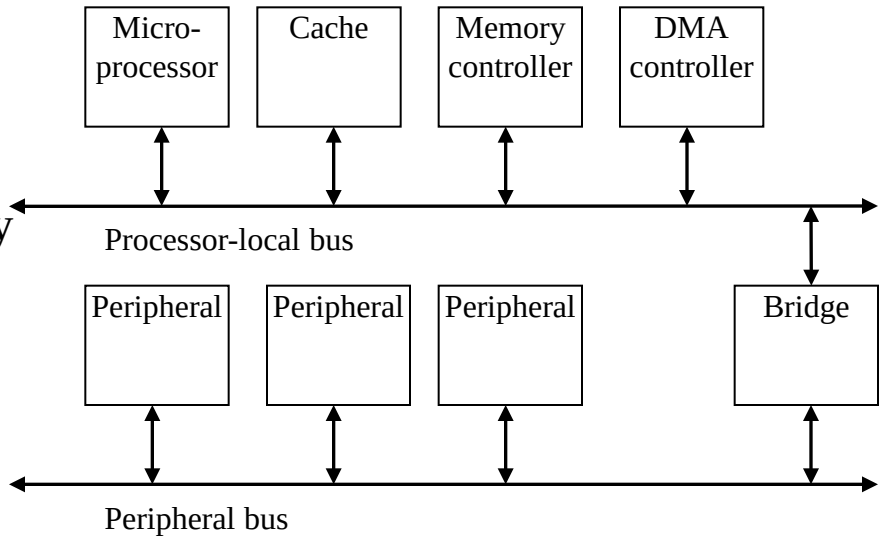
- Pros/cons
  - Easy to add/remove peripheral - no system redesign needed
  - Does not support rotating priority
  - One broken peripheral can cause loss of access to other peripherals



Multi Level Bus Architectures

Multilevel bus architectures

- Don't want one bus for all communications
  - Peripherals would need high-speed, processor-specific bus interface
    - excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus
- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc.
- Peripheral bus
  - Lower speed, narrower, less frequent communication
  - Typically industry parallel standard bus (ISA, PCI, PCIe) for portability
- Bridge
  - Single-purpose processor converts communication between busses



Summary

- General-purpose processors
  - Port-based or bus-based I/O
  - I/O addressing: Memory mapped I/O or Standard I/O
  - Interrupt handling: fixed or vectored
  - Direct memory access
- Arbitration
  - Priority arbiter (fixed/rotating) or daisy chain
- Bus hierarchy