
Chapter 6: Interfacing

Outline

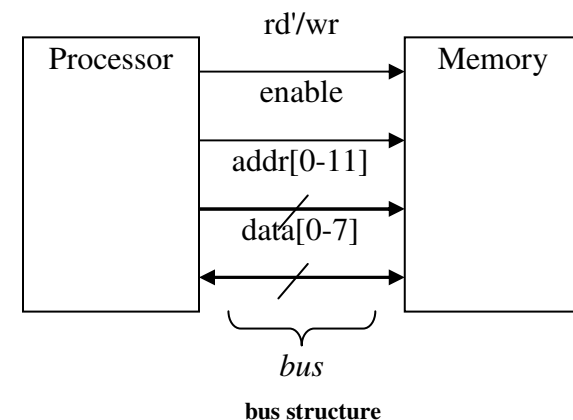
- Interfacing basics
- Microprocessor interfacing
 - I/O Addressing
 - Interrupts
 - Direct memory access
- Arbitration
- Hierarchical buses
- Protocols
 - Serial
 - Parallel
 - Wireless

Introduction

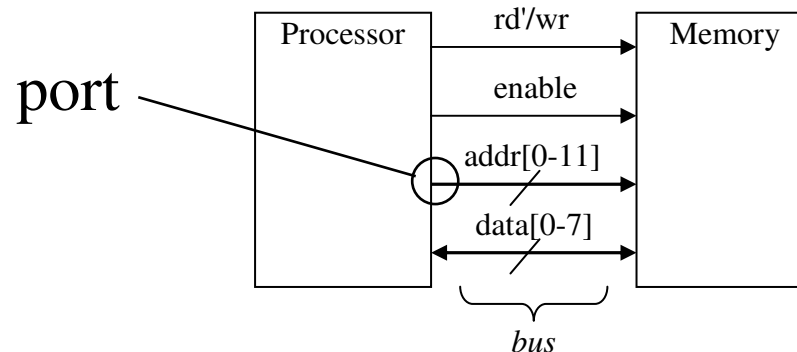
- Embedded system functionality aspects
 - Processing
 - Transformation of data
 - Implemented using processors
 - Storage
 - Retention of data
 - Implemented using memory
 - Communication
 - Transfer of data between processors and memories
 - Implemented using buses
 - Called *interfacing*

A simple bus

- Wires:
 - Uni-directional or bi-directional
 - 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

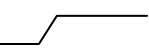
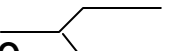


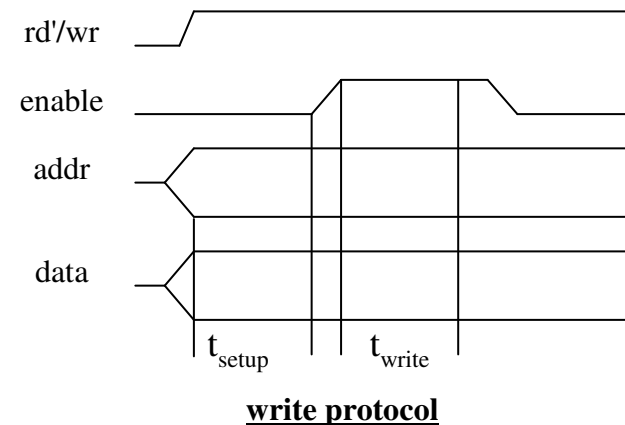
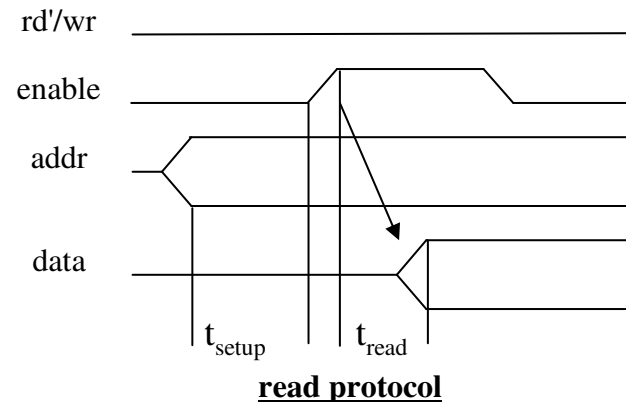
Ports



- Conducting device on periphery
- Connects 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

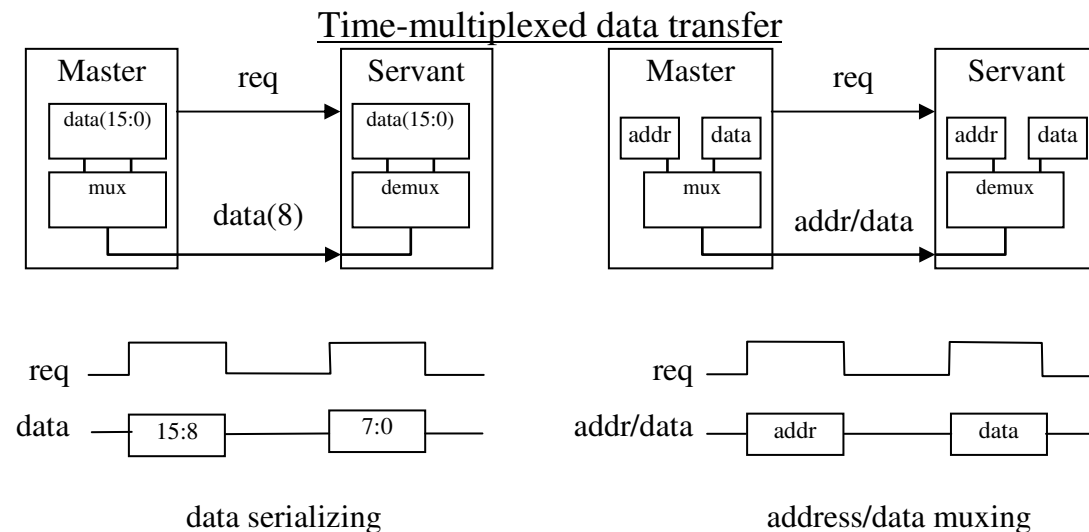
Timing Diagrams

- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis
- Control signal: low or high 
 - May be active low (indicated by: go_n , go' , $/go$, or go_L)
 - Use terms *assert* (active) and *deassert*
 - Asserting go' means $go=0$ 
- Data signal: not valid or valid
- Read example
 - rd'/wr set low, address placed on *addr* for at least t_{setup} time before *enable* asserted, *enable* triggers memory to place data on *data* wires by time t_{read}

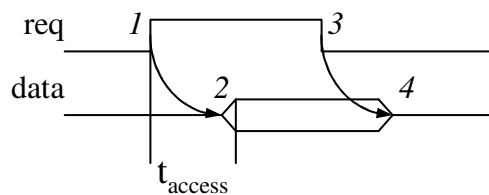
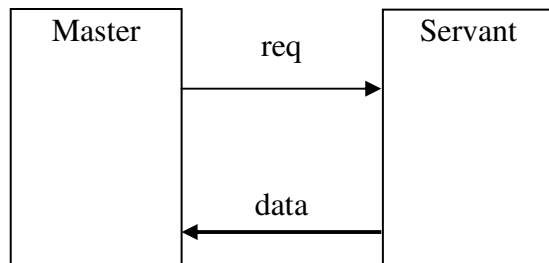


Basic protocol concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses
 - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
 - Share a single set of wires for multiple pieces of data
 - Saves wires at expense of time

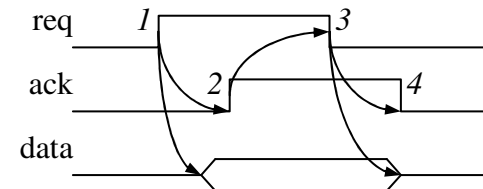
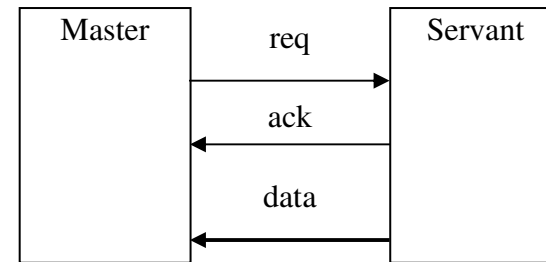


Basic protocol concepts: control methods



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time** t_{access}
3. Master receives data and deasserts *req*
4. Servant ready for next request

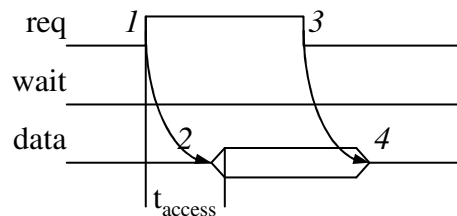
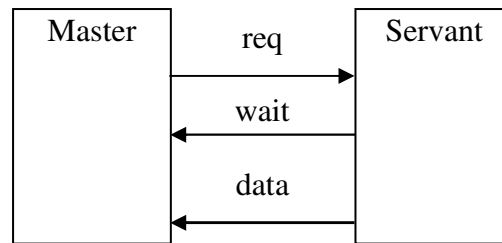
Strobe protocol



1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts** *ack*
3. Master receives data and deasserts *req*
4. Servant ready for next request

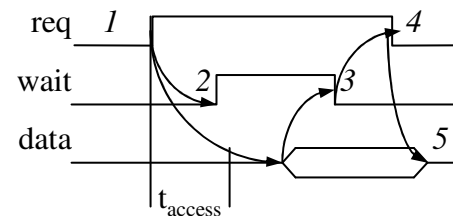
Handshake protocol

A strobe/handshake compromise



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}** (*wait* line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

Fast-response case



1. Master asserts *req* to receive data
2. Servant can't put data within t_{access} , **asserts *wait* ack**
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

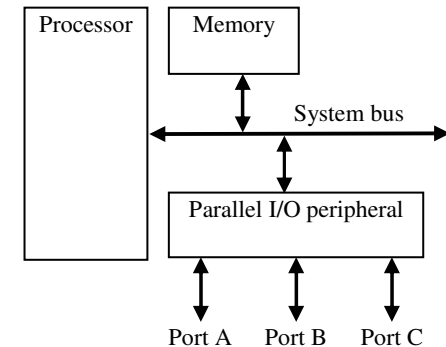
Slow-response case

Microprocessor interfacing: I/O addressing

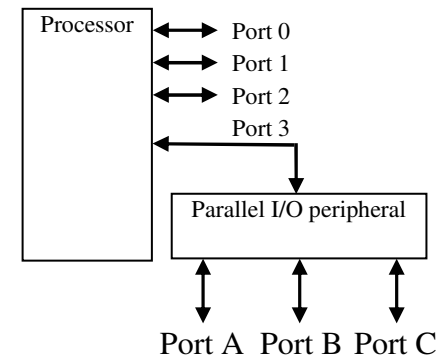
- A microprocessor communicates with other devices using some of its pins
 - Port-based I/O (parallel I/O)
 - Processor has one or more N-bit ports
 - Processor's software reads and writes a port just like a register
 - E.g., P0 = 0xFF; var = P1.2; -- P0 and P1 are 8-bit ports
 - Bus-based I/O
 - Processor 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

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

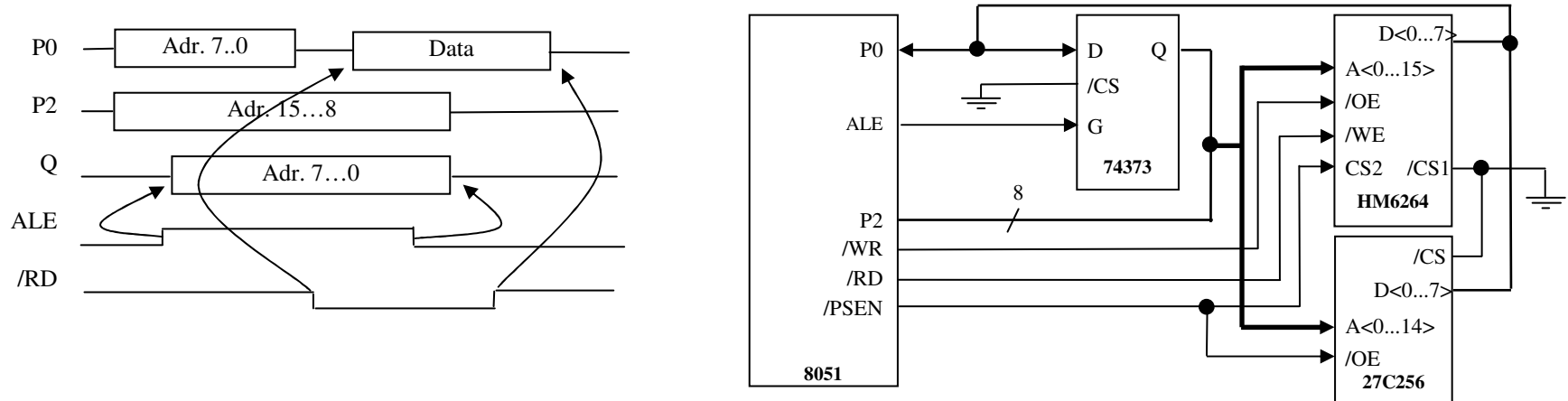
Types of bus-based I/O: memory-mapped I/O and standard I/O

- 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 work with peripherals as well
- 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

A basic memory protocol



- Interfacing an 8051 to external memory
 - Ports P0 and P2 support port-based I/O when 8051 internal memory being used
 - Those ports serve as data/address buses when external memory is being used
 - 16-bit address and 8-bit data are time multiplexed; low 8-bits of address must therefore be latched with aid of ALE signal

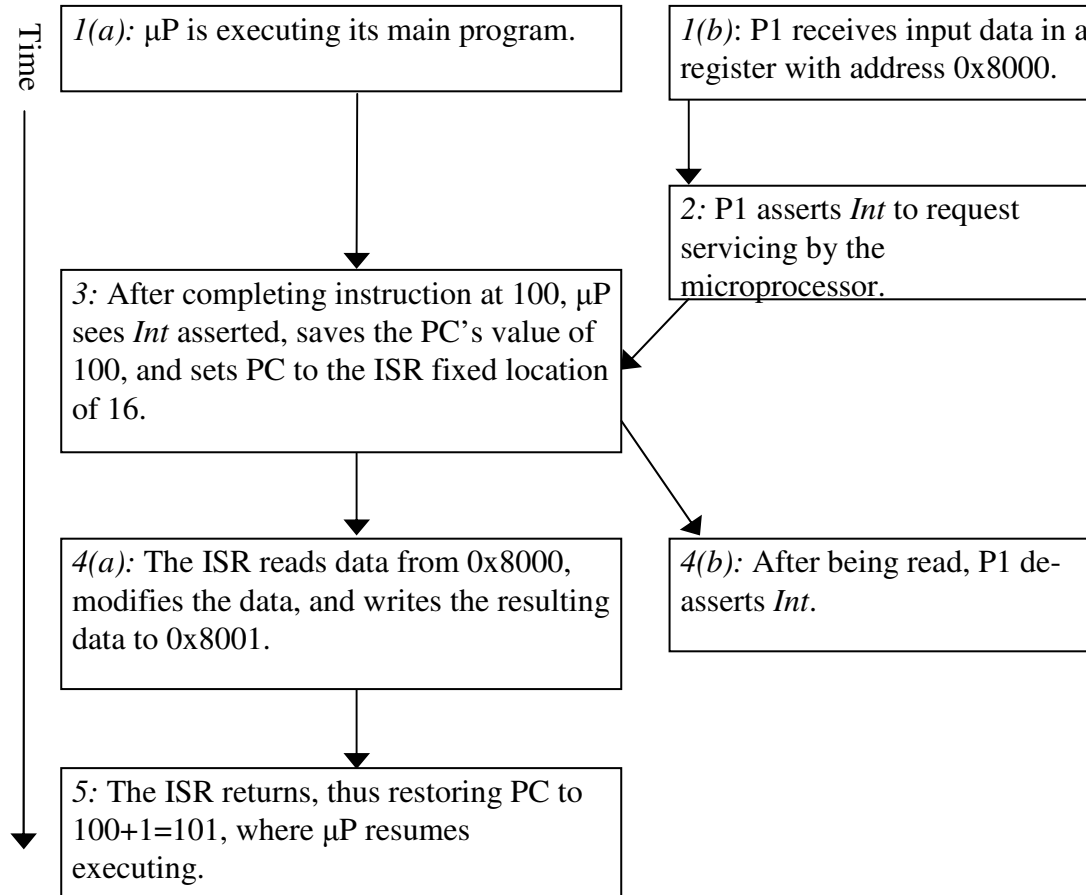
Microprocessor interfacing: interrupts

- 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 (Interrupt Service Routine)?
 - 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

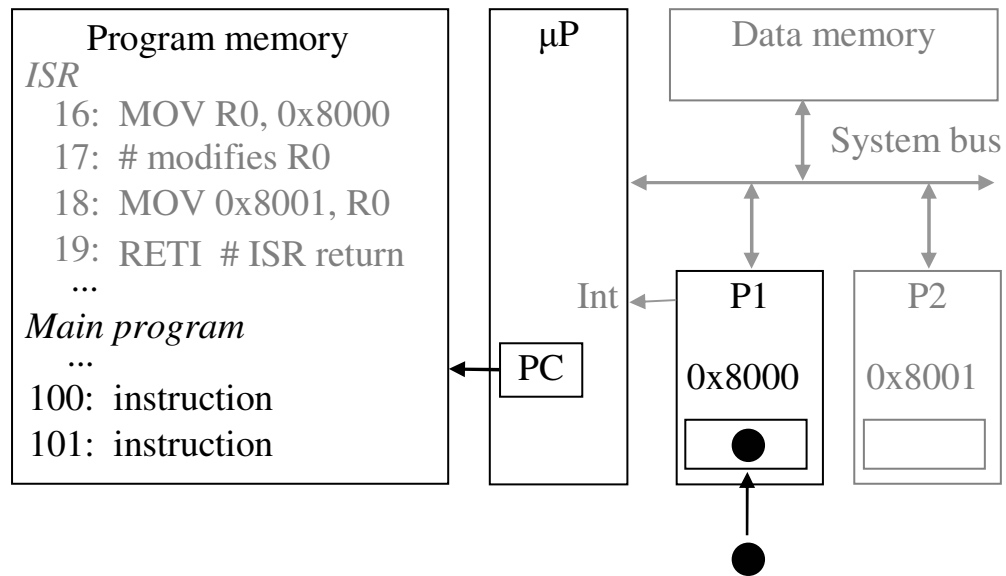
Interrupt-driven I/O using fixed ISR location



Interrupt-driven I/O using fixed ISR location

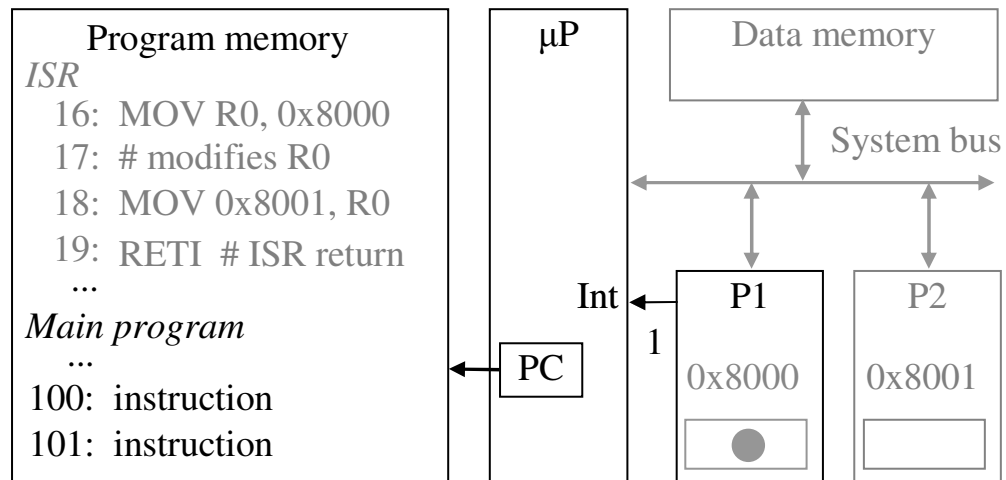
1(a): μP is executing its main program

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



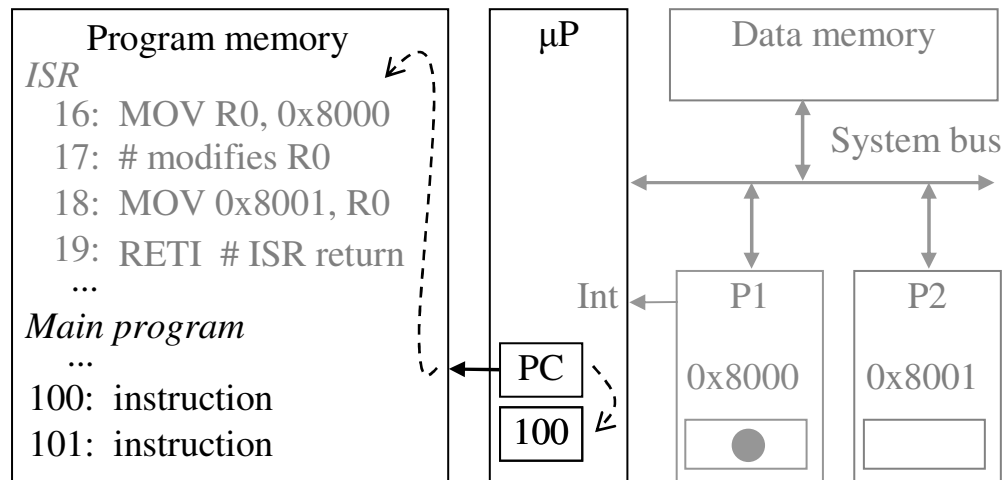
Interrupt-driven I/O using fixed ISR location

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



Interrupt-driven I/O using fixed ISR location

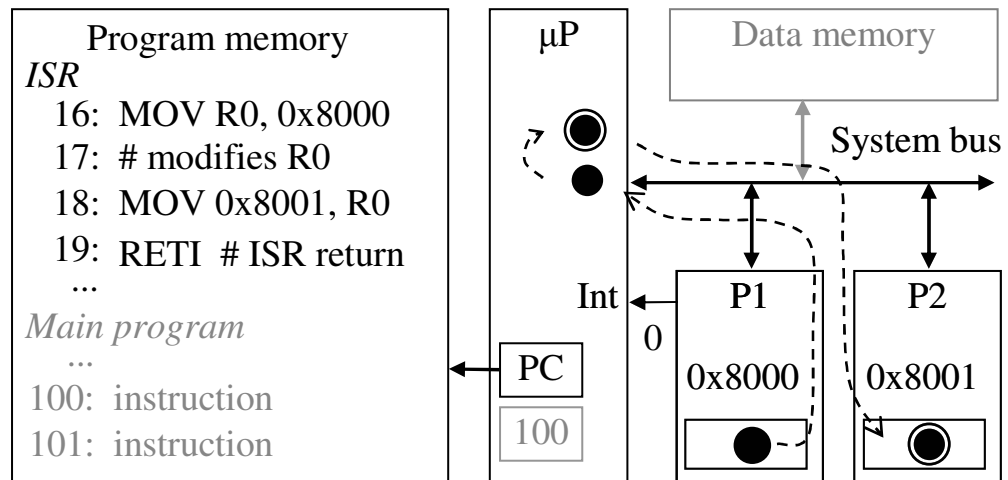
3: After completing instruction at 100, μ P sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location

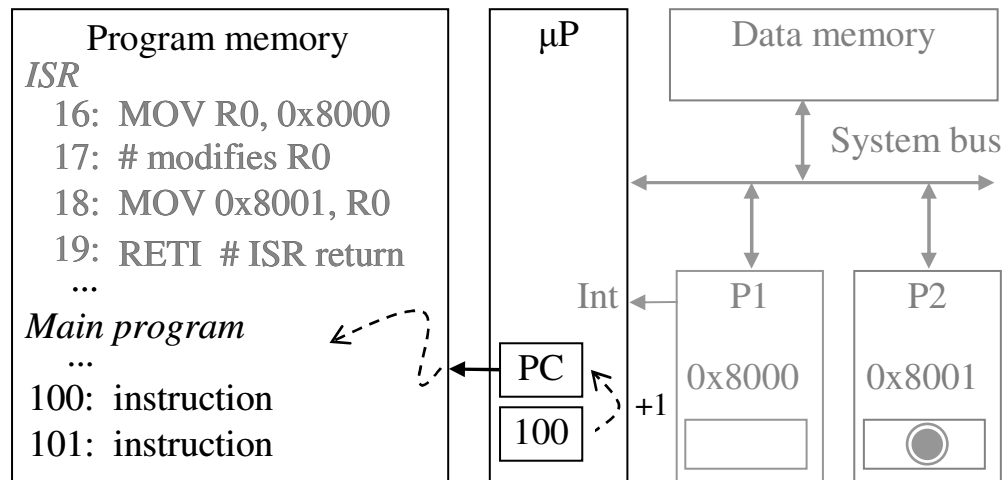
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

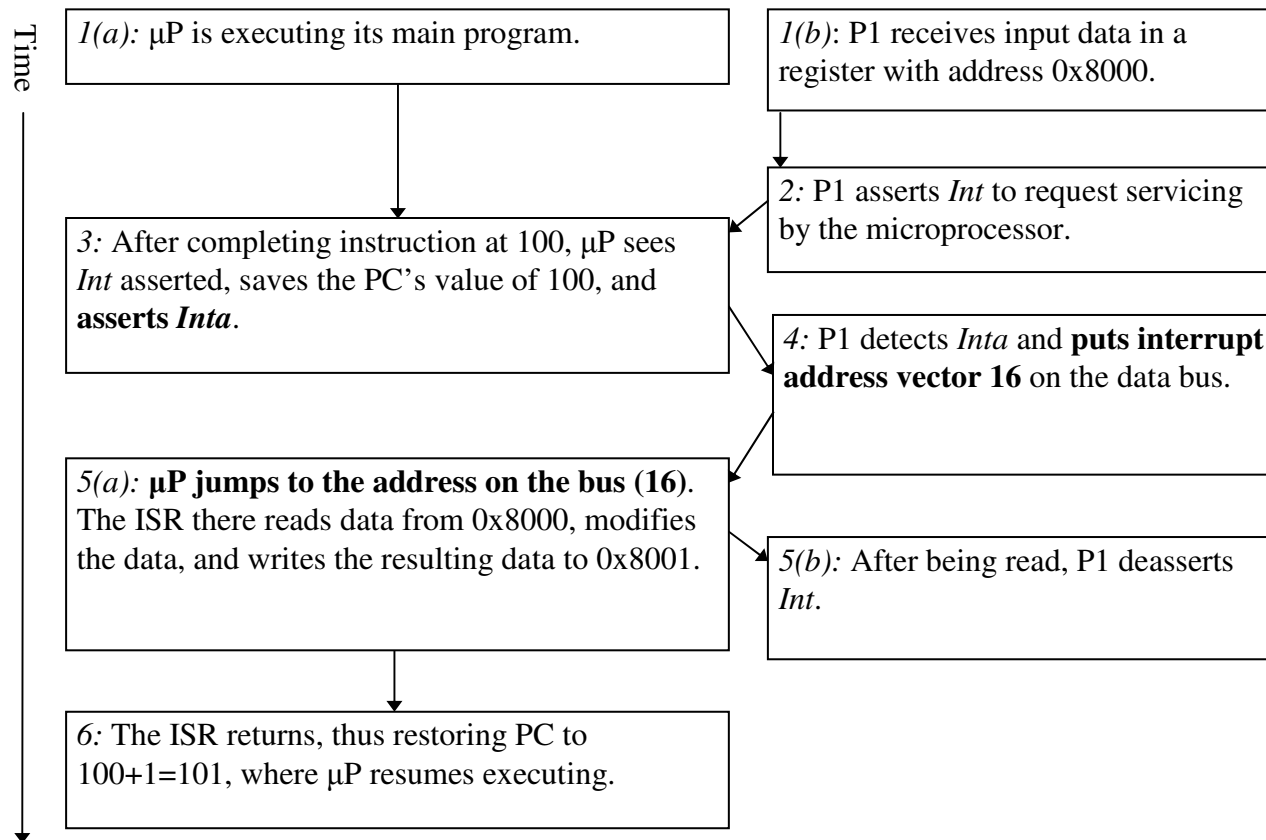


Interrupt-driven I/O using fixed ISR location

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



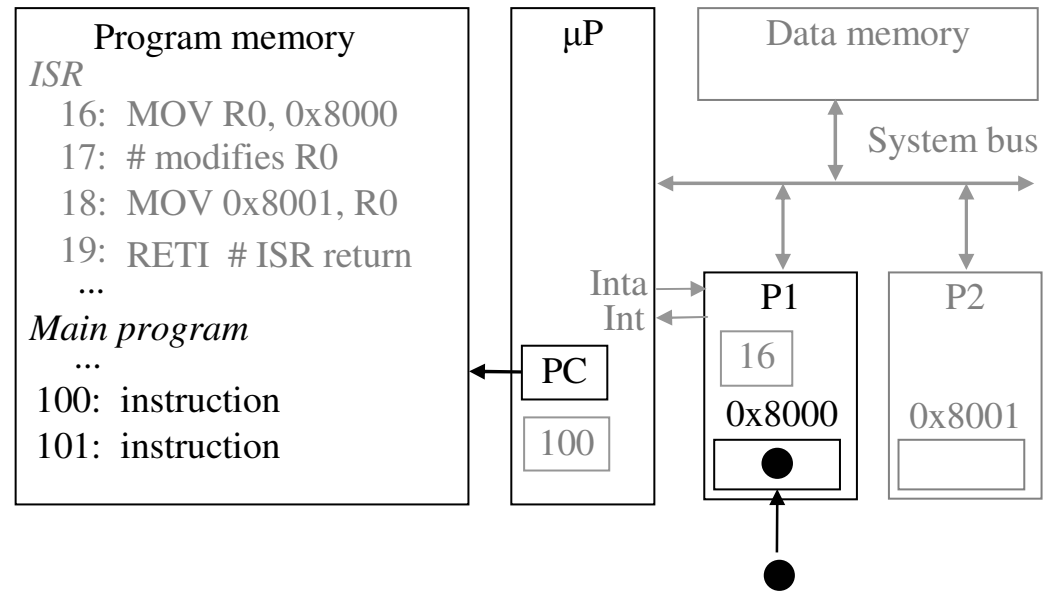
Interrupt-driven I/O using vectored interrupt



Interrupt-driven I/O using vectored interrupt

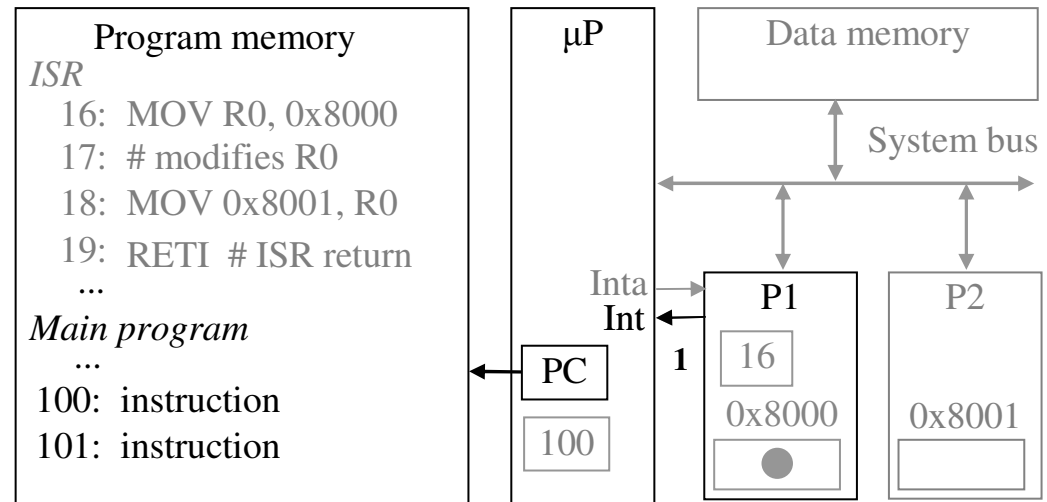
1(a): P is executing its main program

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



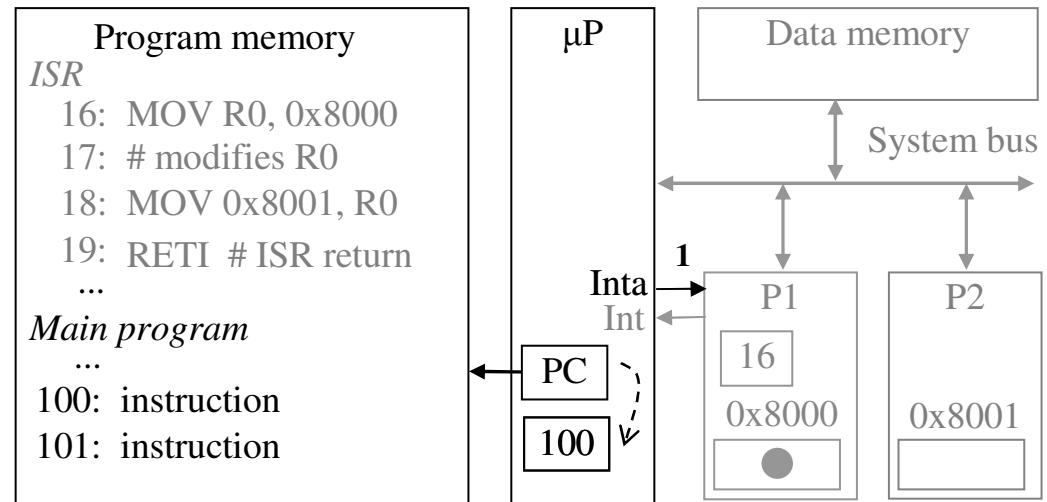
Interrupt-driven I/O using vectored interrupt

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



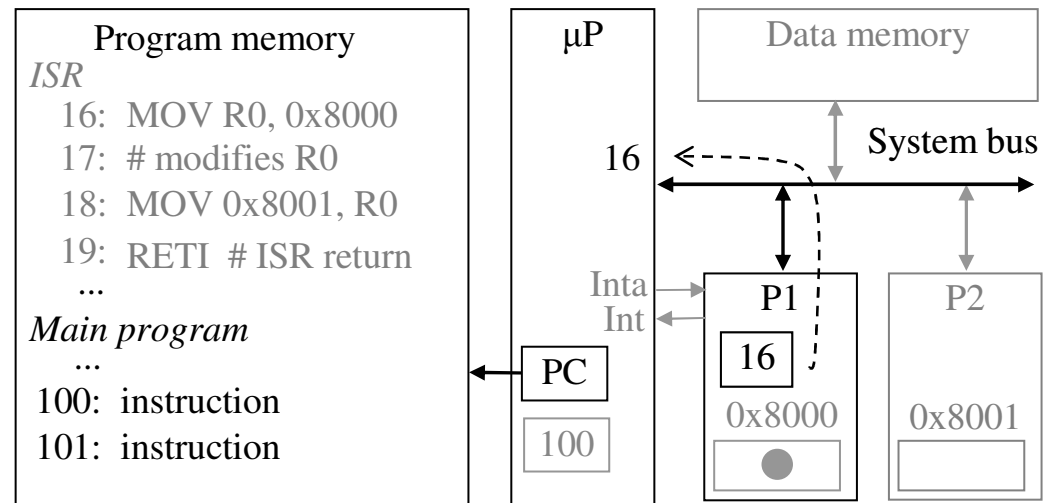
Interrupt-driven I/O using vectored interrupt

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

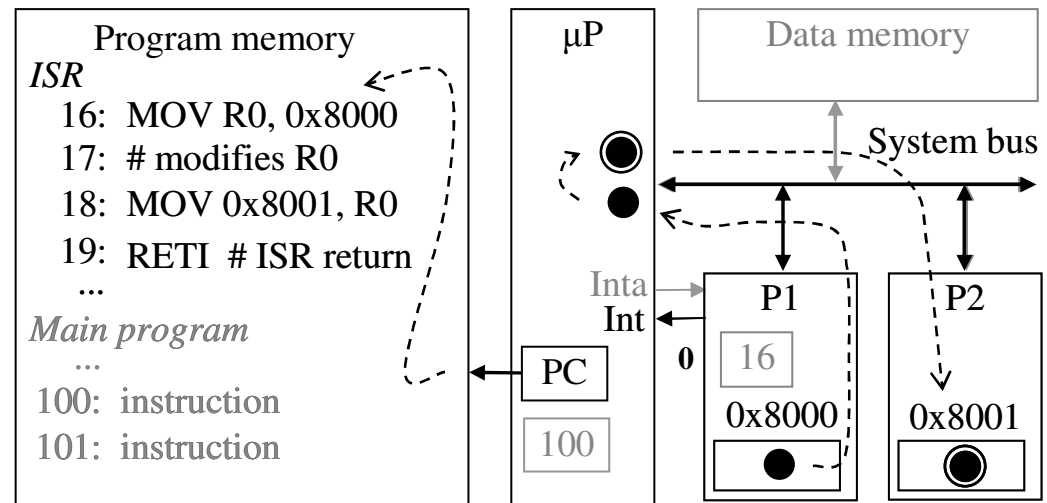
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-driven I/O using vectored interrupt

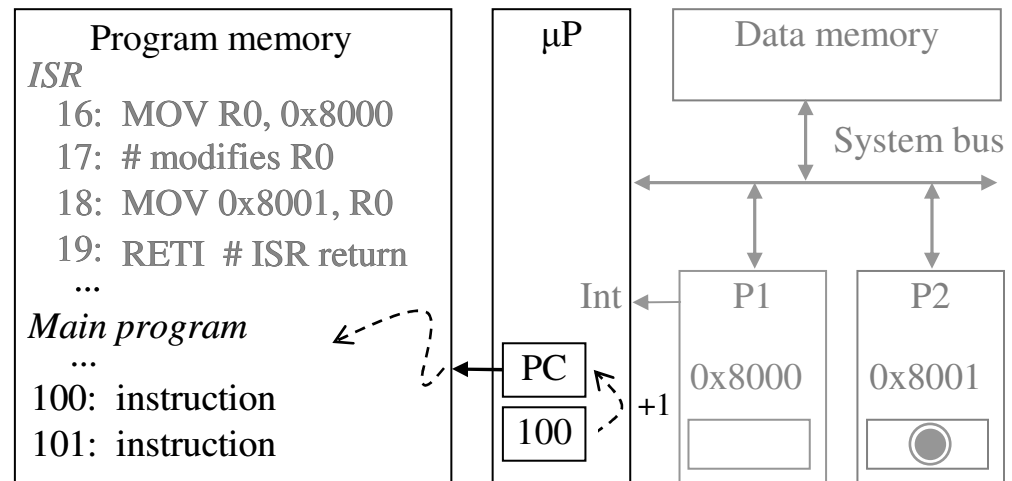
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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
 - 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

Things to do before Thursday...

- Read through Page 166 in ESD