



Embedded Systems: Communication

Embedded Systems: bus and interfacing

Lecturer:

Ing. Luca Pizzamiglio

Politecnico di Milano, DEI

pizzamiglio@gmail.com

Outline



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

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, memories and peripherals
 - Implemented using **buses**
 - Called *interfacing*

A simple bus

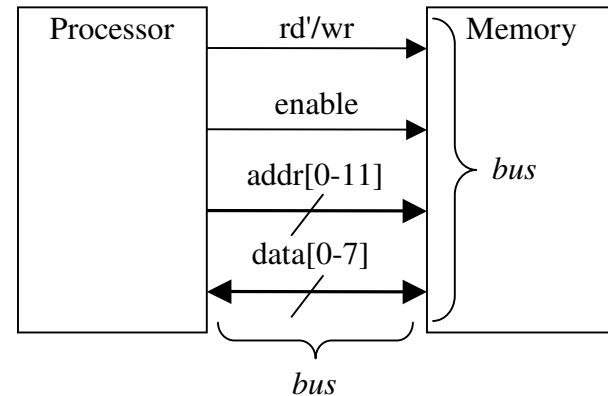


- Wires

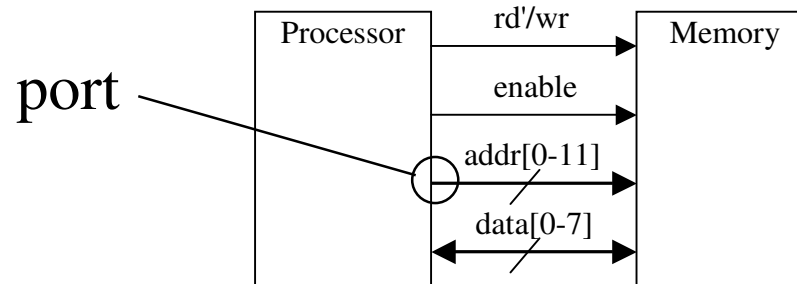
- ▶ Uni-directional or bi-directional
- ▶ One line may represent **multiple** wires

- Bus

- ▶ **Set** of wires with a **single** function
 - Address bus (12 bits), data bus (8 bits)
- ▶ Or, **entire** collection of wires
 - Address, data and control
 - Associated **protocol**: rules for communication



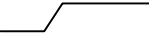

Ports

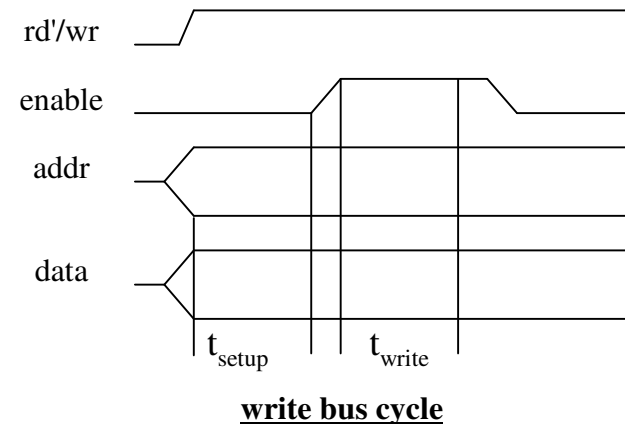
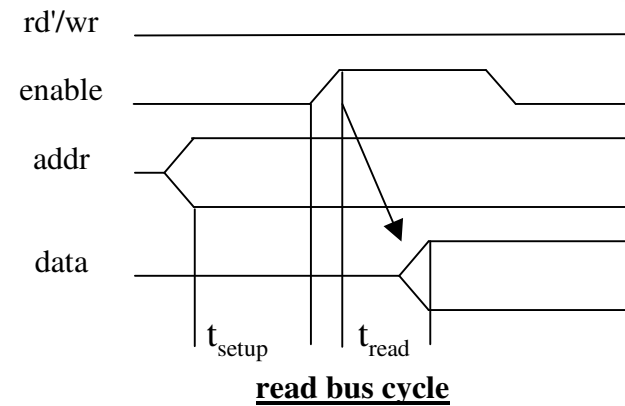


- Connection between chips and physical links
- Often referred to as a *pin*
 - ▶ Actual pins on periphery of IC package that plug into socket on printed-circuit board
 - ▶ Sometimes metallic balls instead of pins
 - ▶ Today, metal “pads” connecting processors and memories within single IC (SOC technology)
- Single wire or set of wires with single function
 - ▶ enable port
 - ▶ 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 (e.g., \overline{go} , $\overline{go_L}$ or \overline{go})
 - ▶ Use terms *assert* (active) and *deassert*
 - ▶ Asserting \overline{go} means $go=0$
- **Data** signal: not valid or valid 
- Protocol may have subprotocols
 - ▶ Called **bus cycle**, e.g., read and write
 - ▶ Each may be several **clock** cycles
- 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



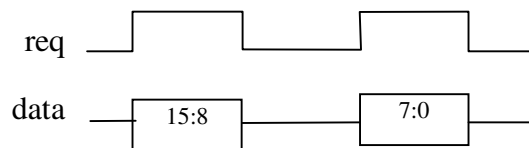
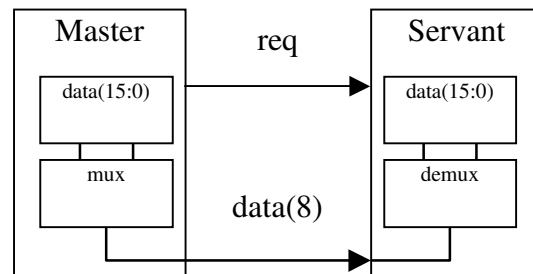
- **Actor**: entity involved in communication task
 - ▶ master initiates the communication
 - ▶ servant (slave) respond
- **Data direction**: denotes the data transfer direction
 - ▶ sender and receiver are detected
 - ▶ different from master and servant
- **Addresses**: special kind of data
 - ▶ Specifies a location in memory, a peripheral, or a register within a peripheral

	Read	Write
Processor	Master/receiver	Master/sender
Memory	Servant/sender	Servant/receiver

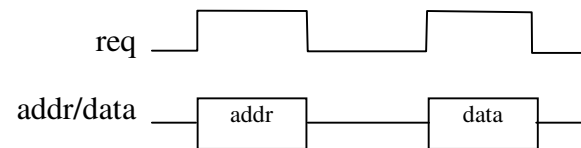
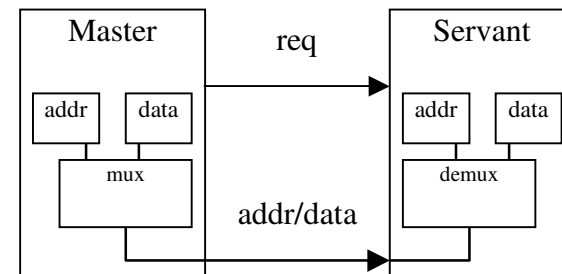
Basic protocol concepts



- Time **multiplexing**
 - ▶ Share a single set of wires for multiple pieces of data
 - Same data; e.g. serializing of data bus
 - Different data; e.g. multiplexing of data and address buses
 - ▶ Saves wires at expense of time
 - Led by design constraints

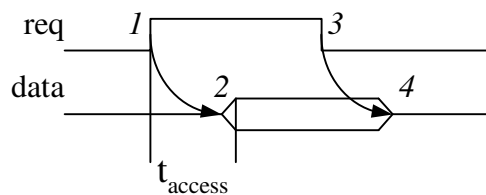
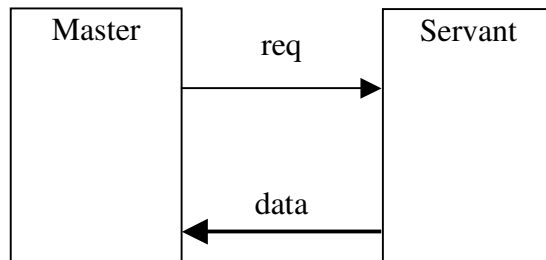


data serializing



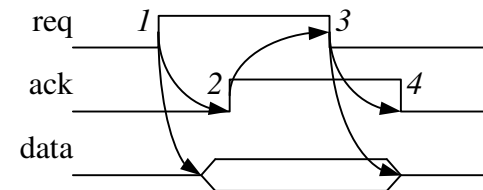
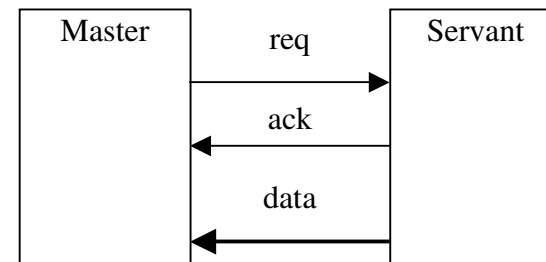
address/data muxing

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

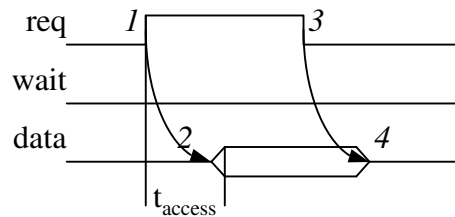
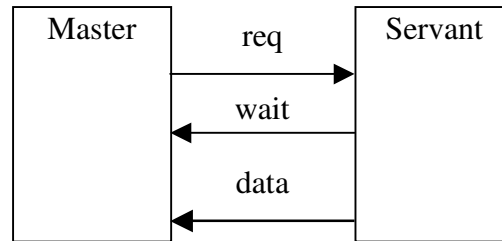
Strobe protocol



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

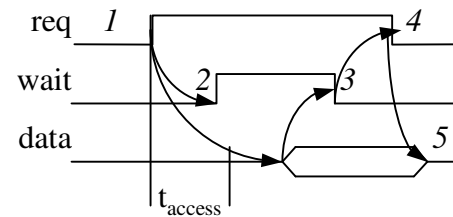
Handshake protocol

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



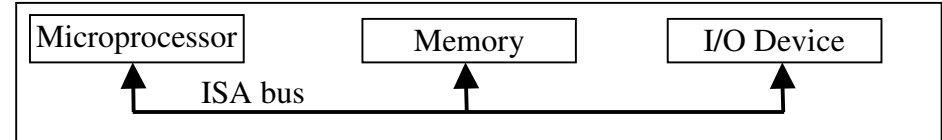
1. Master asserts *req* to receive data
2. Servant can't put data within t_{access} , **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

Slow-response case

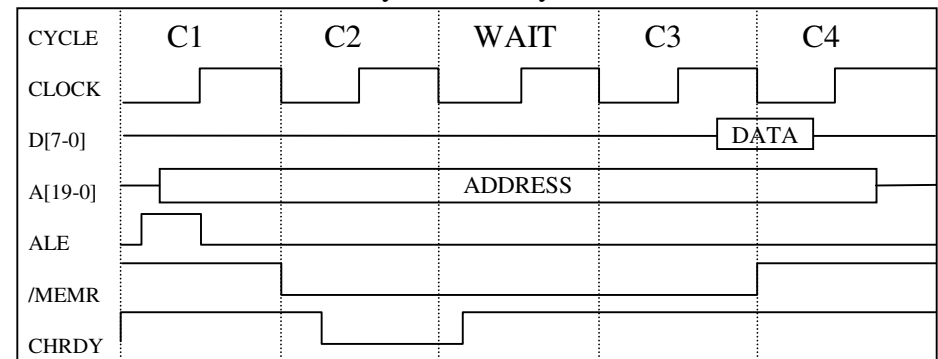
ISA bus protocol - memory access



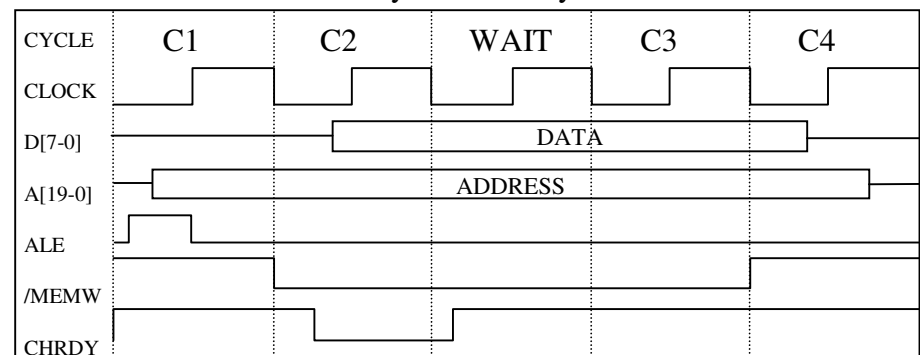
- Real example
- ISA: Industry Standard Architecture
 - ▶ Common in old 80x86's
- Some features
 - ▶ 20-bit address
 - ▶ Compromise strobe/handshake control
 - 4 cycles default
 - Unless CHRDY deasserted - resulting in additional wait cycles (up to 6)



memory-read bus cycle



memory-write bus cycle



Outline



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

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
 - ▶ 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

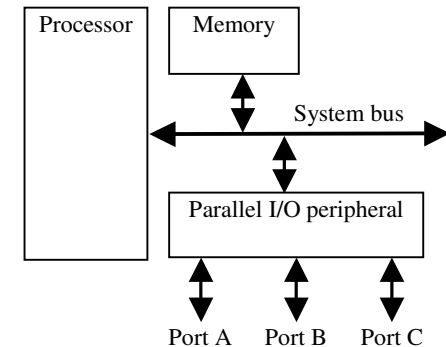


- **Parallel I/O peripheral**

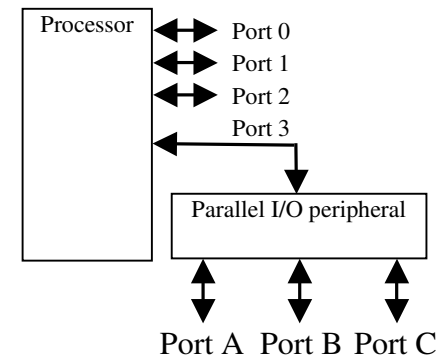
- ▶ When processor only supports bus-based I/O but parallel I/O needed
- ▶ Each port on peripheral connected to a **register** within peripheral that is read/written by the processor

- **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 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

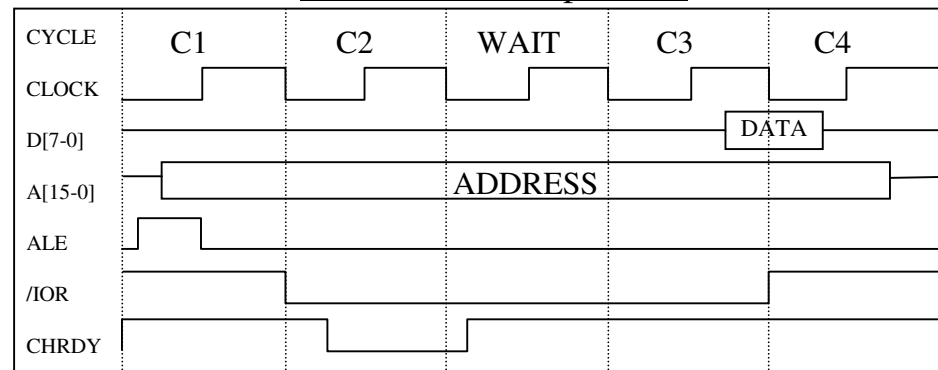
- ▶ No loss of memory addresses to peripherals
- ▶ Simpler address decoding logic in peripherals possible
 - When number of peripherals much smaller than address space then high-order address bits can be ignored
 - smaller and/or faster comparators

ISA bus

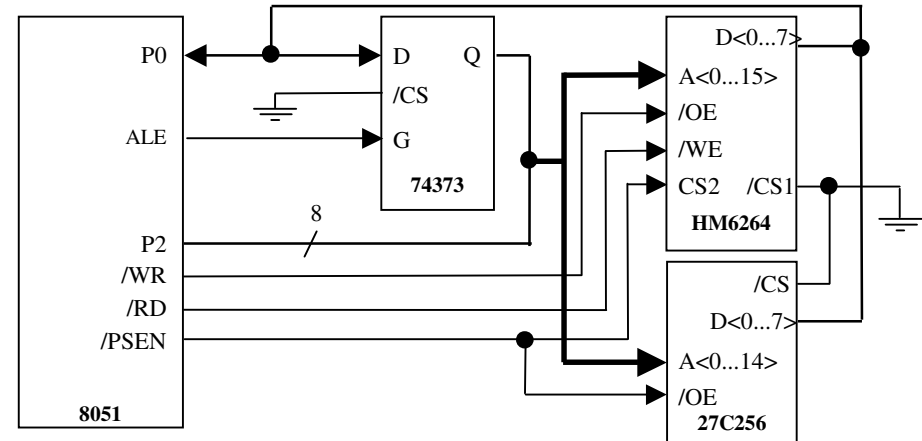
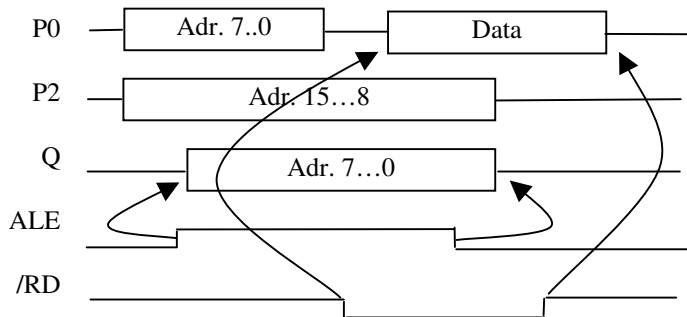


- ISA supports standard I/O
 - ▶ **/IOR** distinct from **/MEMR** for peripheral read
 - **/IOW** used for writes
 - ▶ 16-bit address space for I/O vs. 20-bit address space for memory
 - ▶ Otherwise very similar to memory protocol

ISA I/O bus read protocol



A basic memory protocol



- Interfacing an 8051 uC 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

Outline



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

Microprocessor interfacing: interrupts



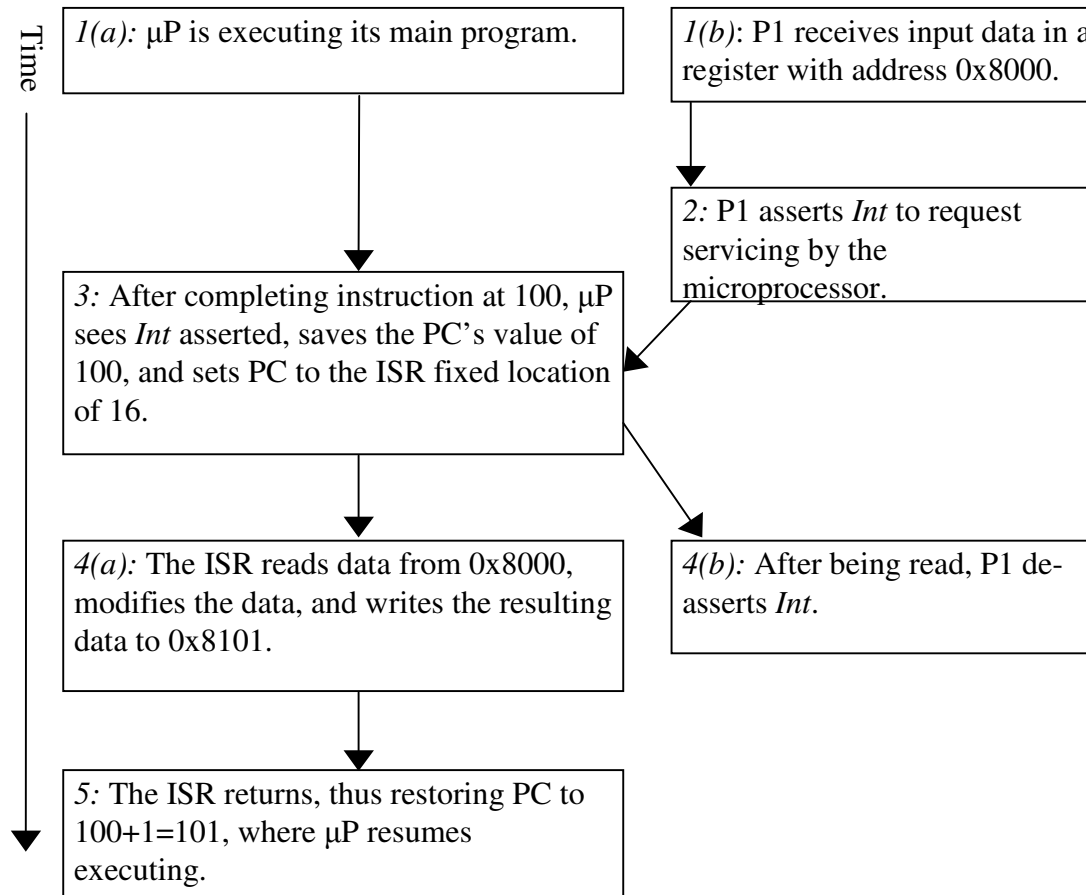
- Suppose a peripheral **intermittently** receives data, which must be serviced by the processor
 - ▶ The processor can **poll** the peripheral regularly to see if data has arrived - **wasteful**
 - ▶ The peripheral can **interrupt** the processor when it has data
- Requires an extra pin or pins: **Int**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

Interrupt-driven I/O using fixed ISR location

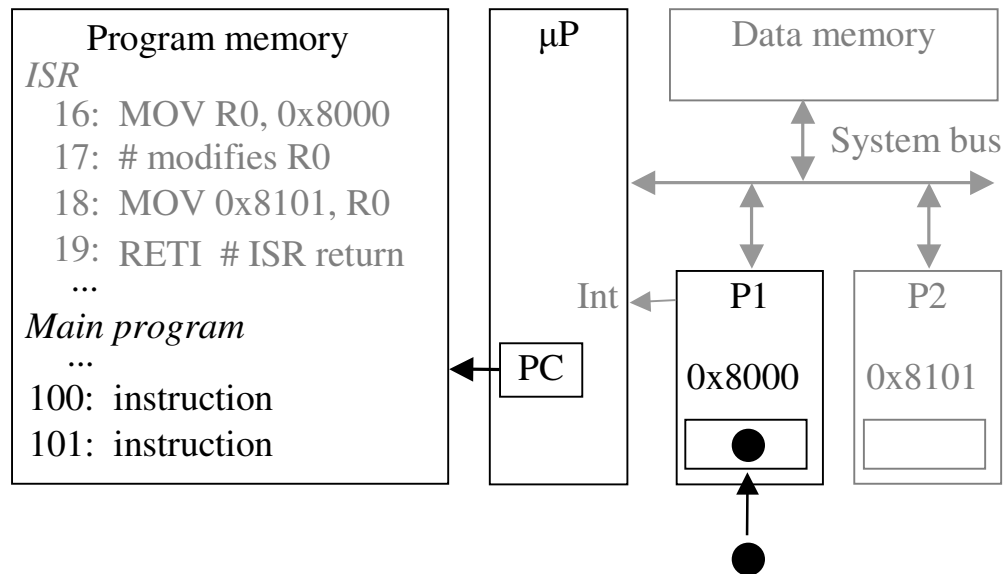


Interrupt-driven I/O using fixed ISR location



1(a): μP is executing its main program

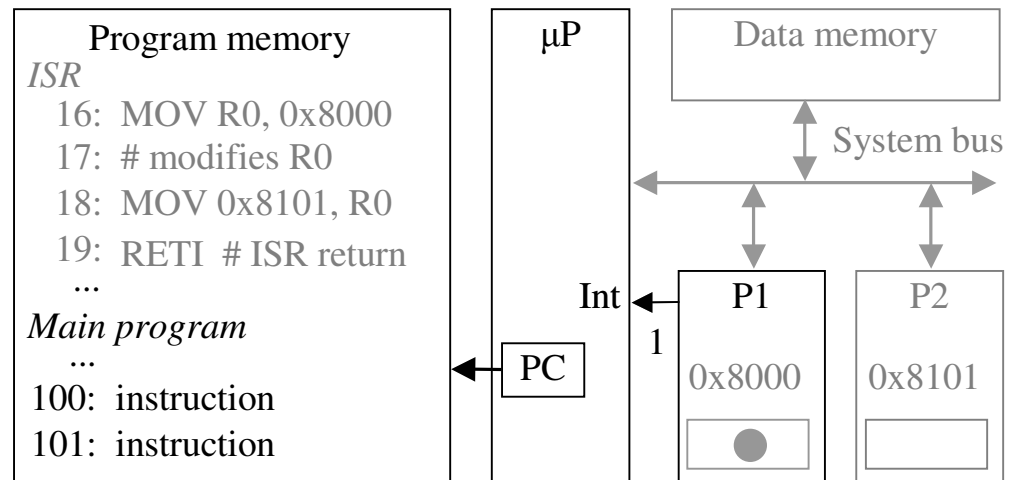
1(b): P1 receives input data in a register with address 0x8000.



Interrupt-driven I/O using fixed ISR location



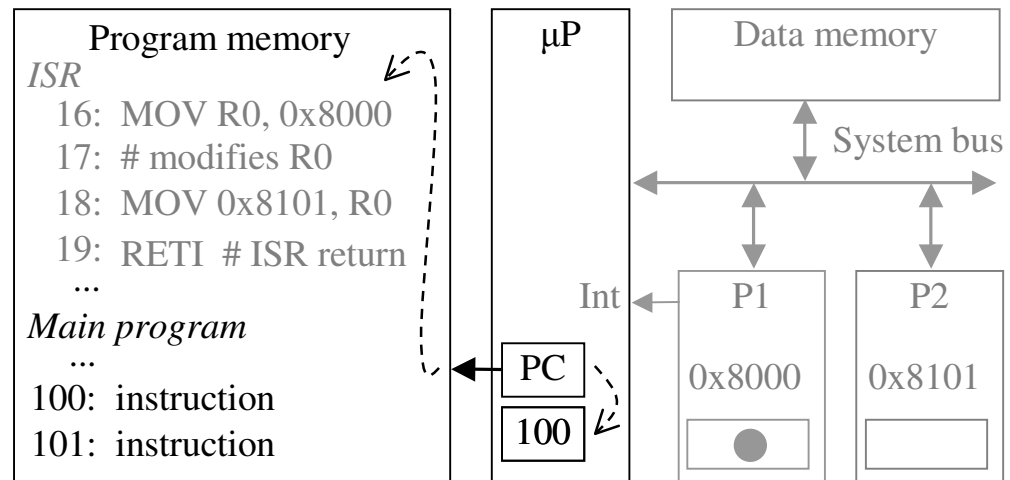
2: P1 asserts *Int* to request servicing by the microprocessor



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.

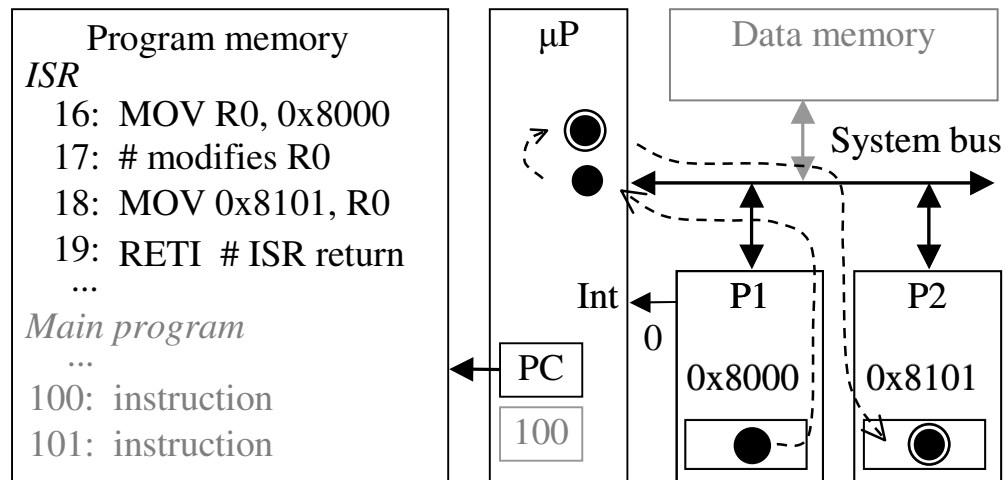


Interrupt-driven I/O using fixed ISR location



4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8101.

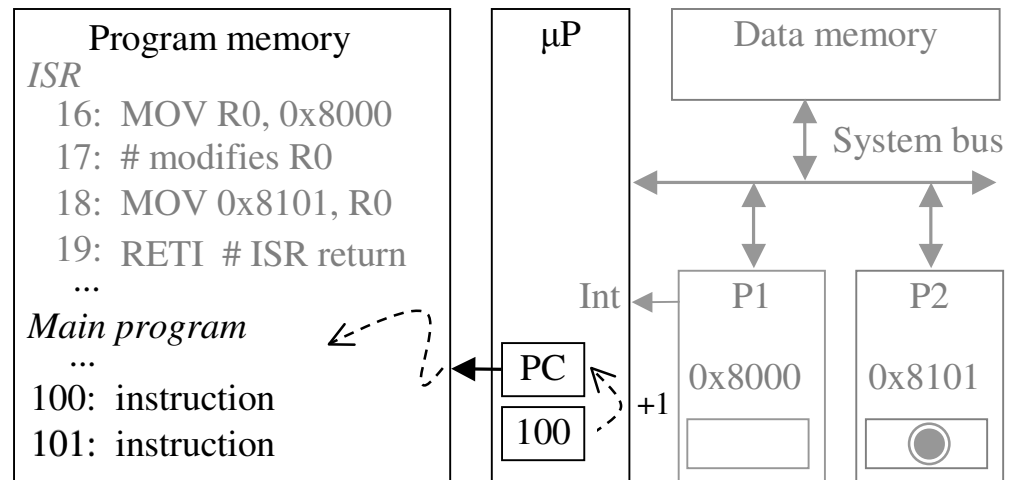
4(b): After being read, P1 deasserts *Int*.



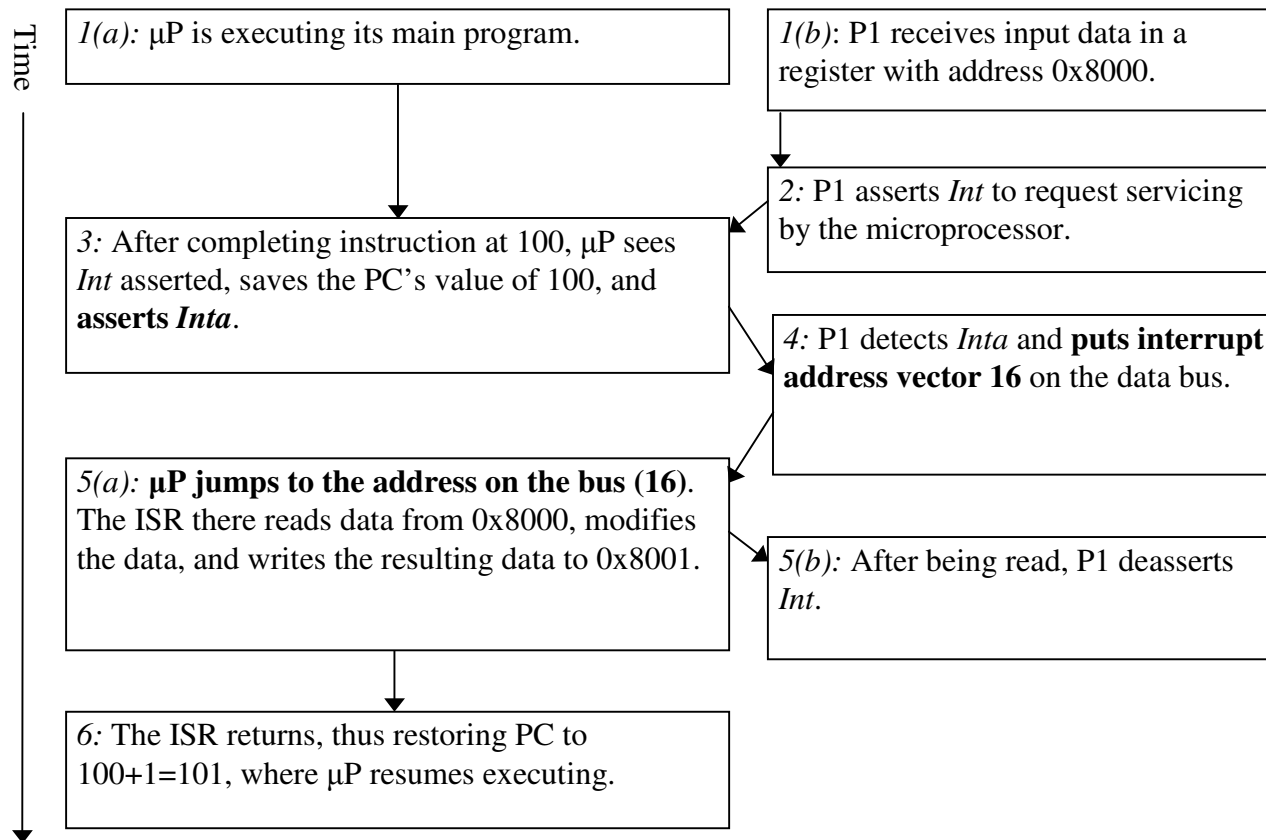
Interrupt-driven I/O using fixed ISR location



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



Interrupt-driven I/O using vectored interrupt

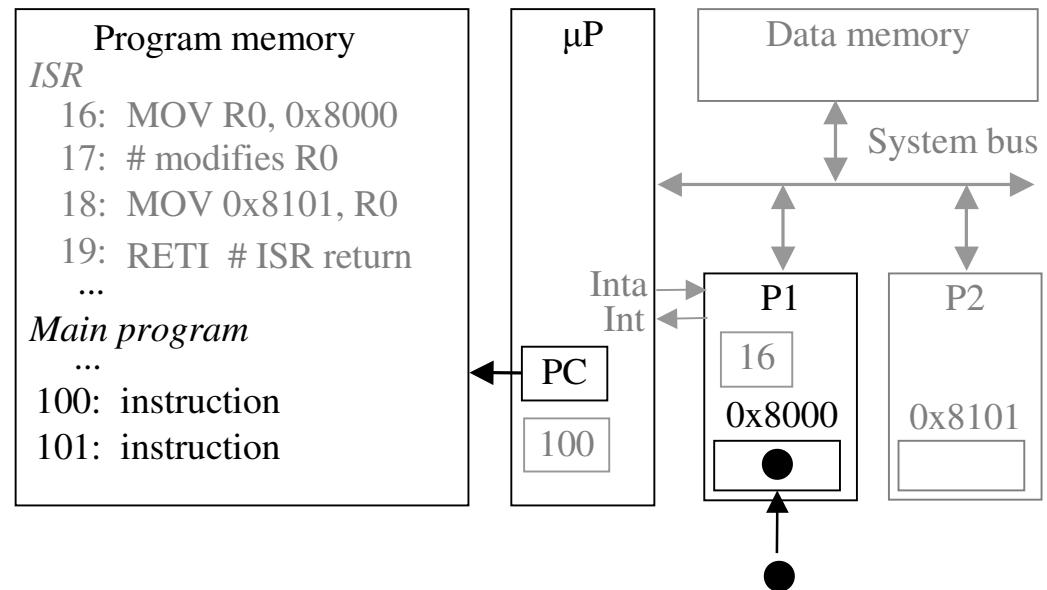


Interrupt-driven I/O using vectored interrupt



1(a): P is executing its main program

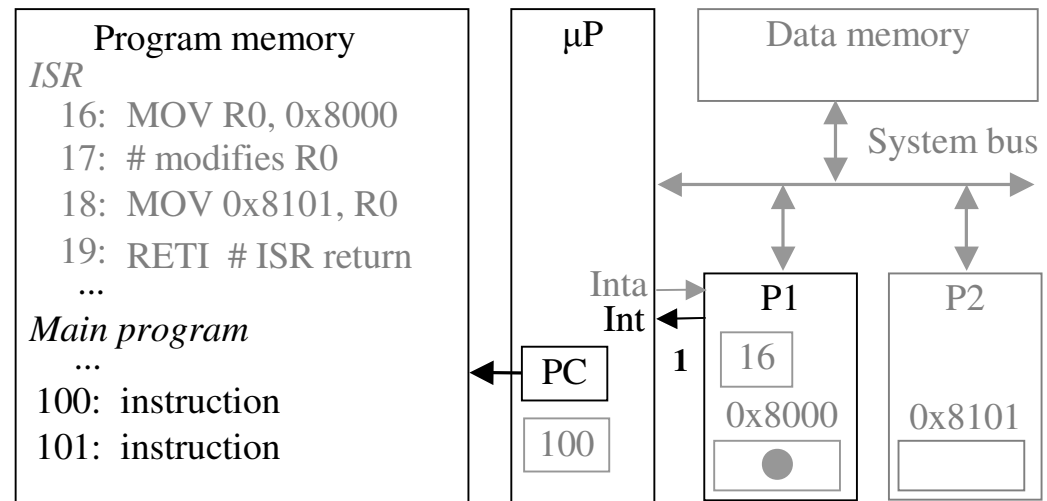
1(b): P1 receives input data in a register with address 0x8000.



Interrupt-driven I/O using vectored interrupt



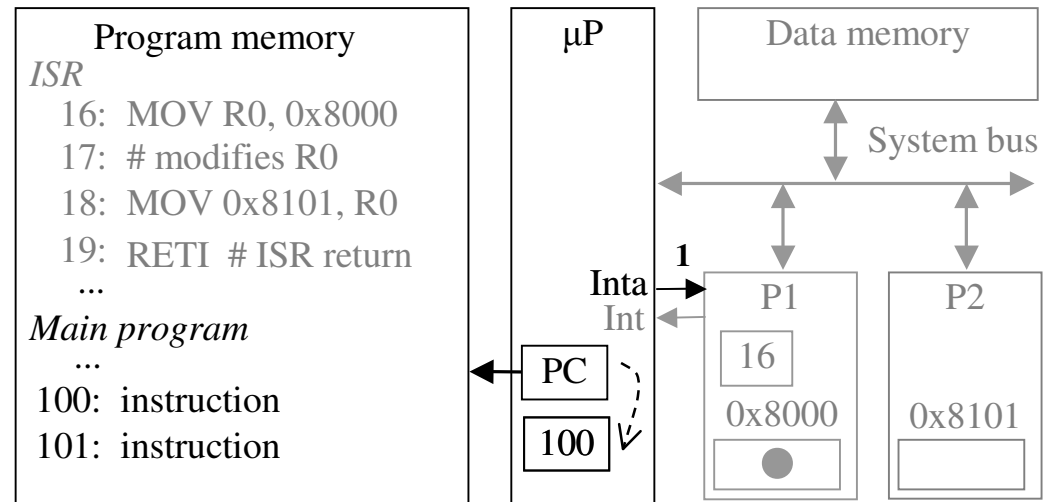
2: P1 asserts *Int* to request servicing by the microprocessor



Interrupt-driven I/O using vectored interrupt



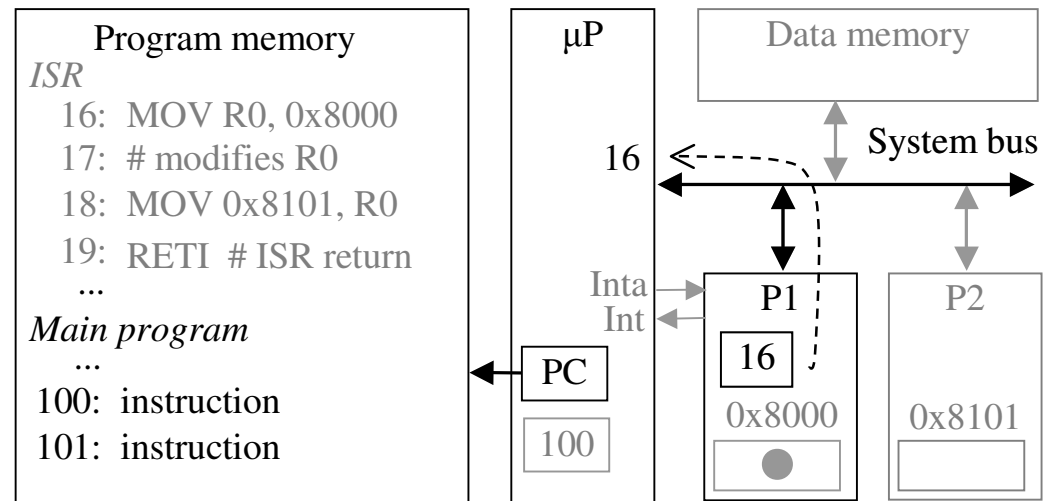
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and **asserts** *Inta*



Interrupt-driven I/O using vectored interrupt



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

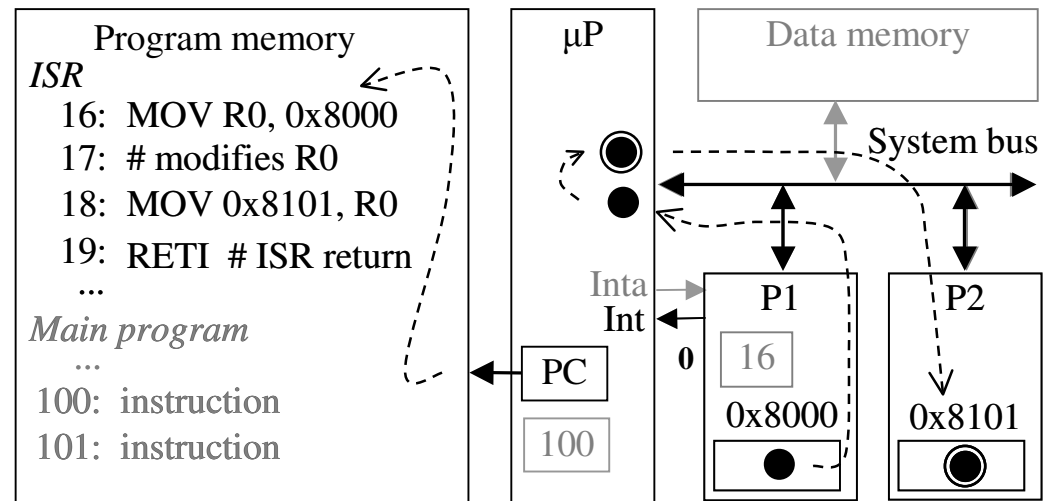


Interrupt-driven I/O using vectored interrupt



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 0x8101.

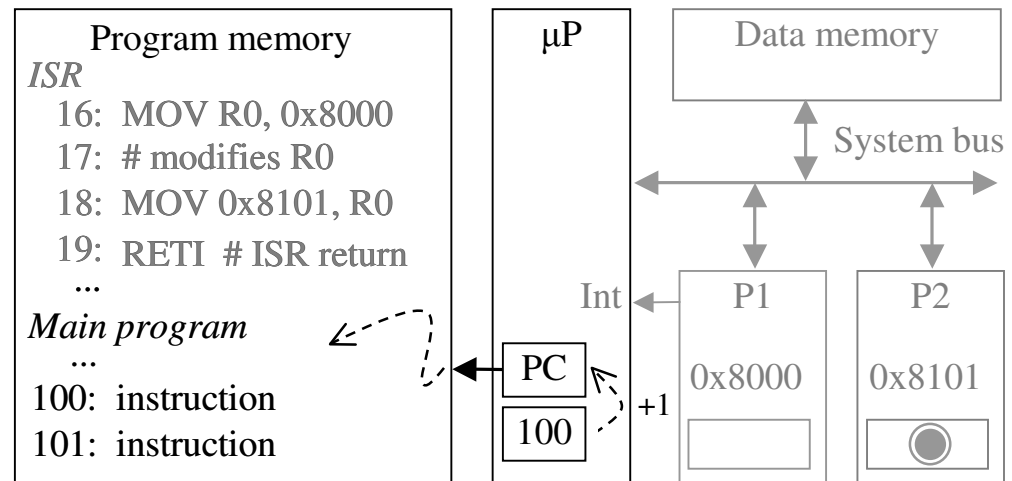
5(b): After being read, P1 deasserts *Int*.



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 and one interrupt acknowledge pin
 - ▶ Table in memory holding ISR addresses (e.g. 256 words)
 - ▶ Peripheral doesn't provide ISR address, but rather **index** into table
 - Fewer bits are sent by the peripheral
 - More suitable for small data bus
 - 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

Interrupt controller



- Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU
- It could provide a priority encoding scheme
- Interrupt controller could be programmable (PIC)
 - ▶ Configured by software
 - ▶ High flexibility
 - ▶ It manages the interrupt vector table
- Simple CPUs use simple schemas
 - ▶ Typically microcontrollers
- Modern CPUs, including embedded ones, integrate a PIC

Outline



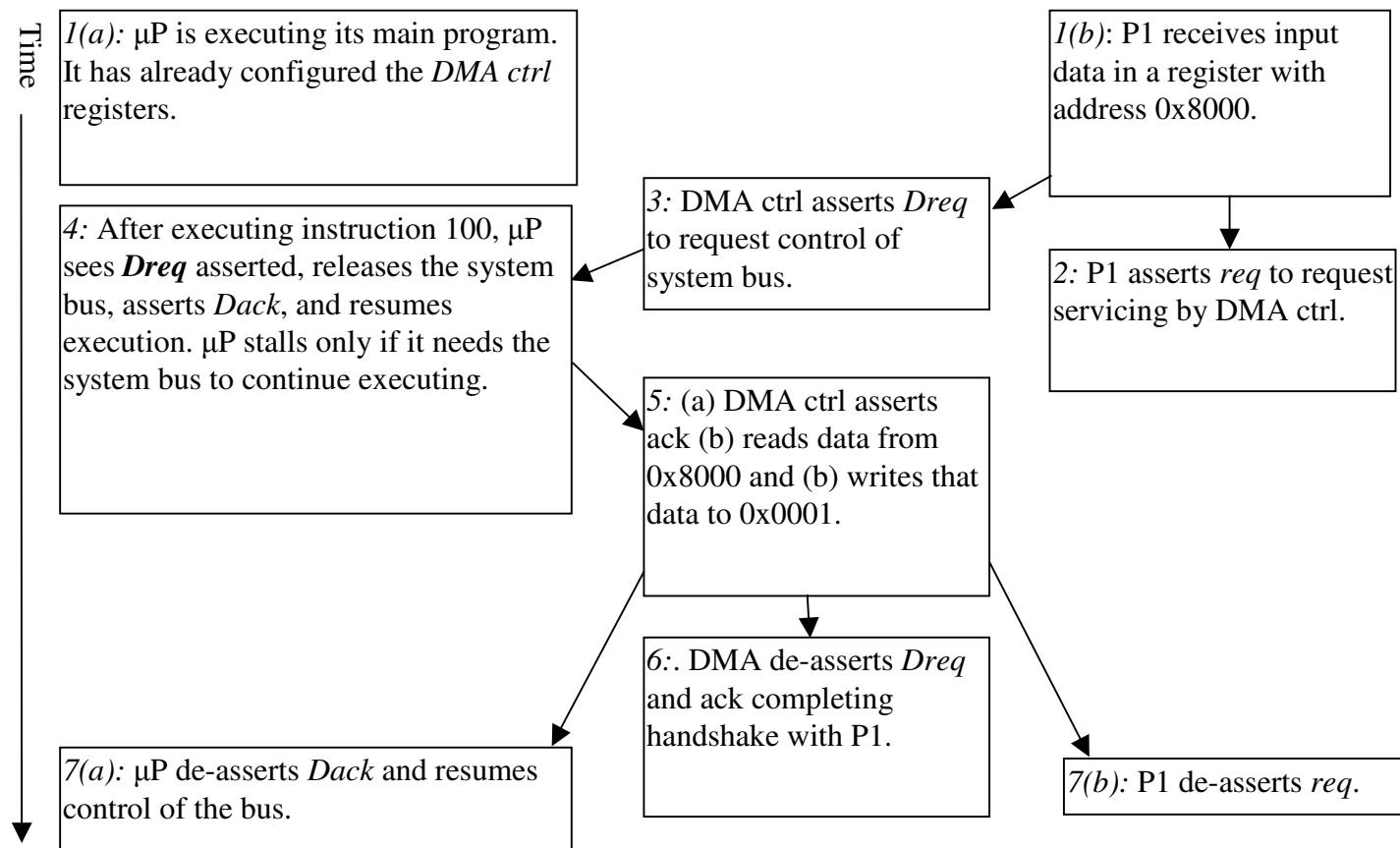
- Interfacing basics
- Microprocessor interfacing
 - ▶ I/O Addressing
 - ▶ Interrupts
 - ▶ Direct memory access
- Arbitration

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 architecture - processor can fetch and execute instructions as long as they don't access data memory - if they do, processor stalls

Peripheral to memory transfer with DMA

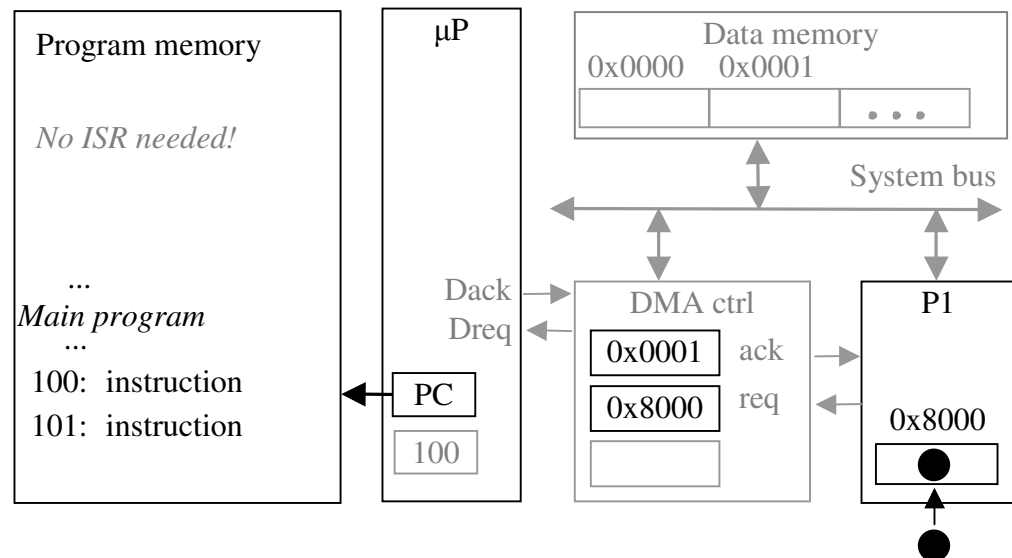


Peripheral to memory transfer with DMA



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.

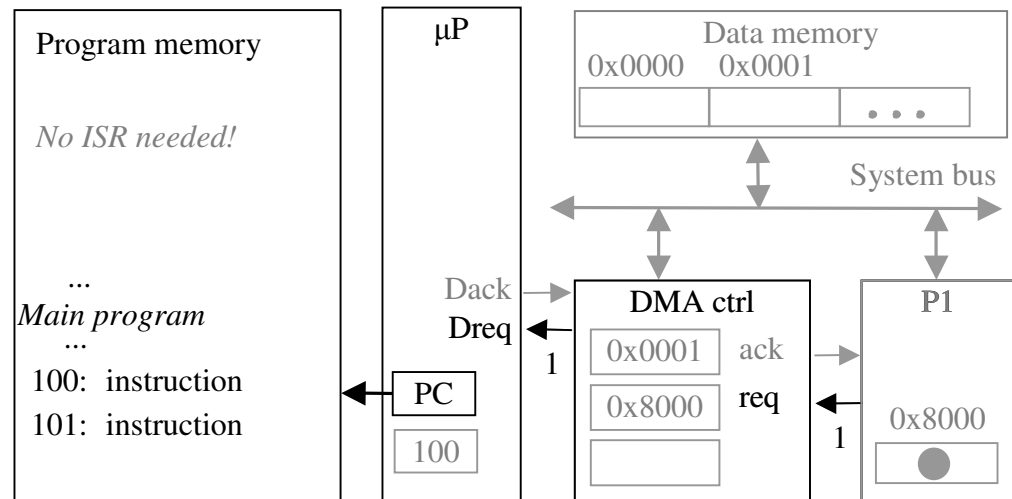


Peripheral to memory transfer with DMA



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

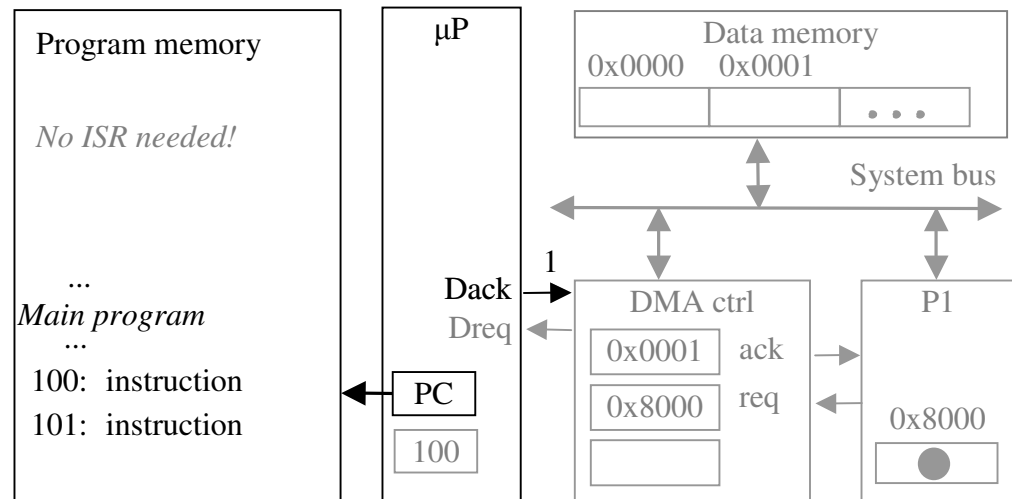
3: DMA ctrl asserts *Dreq* to request control of system bus



Peripheral to memory transfer with DMA



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.

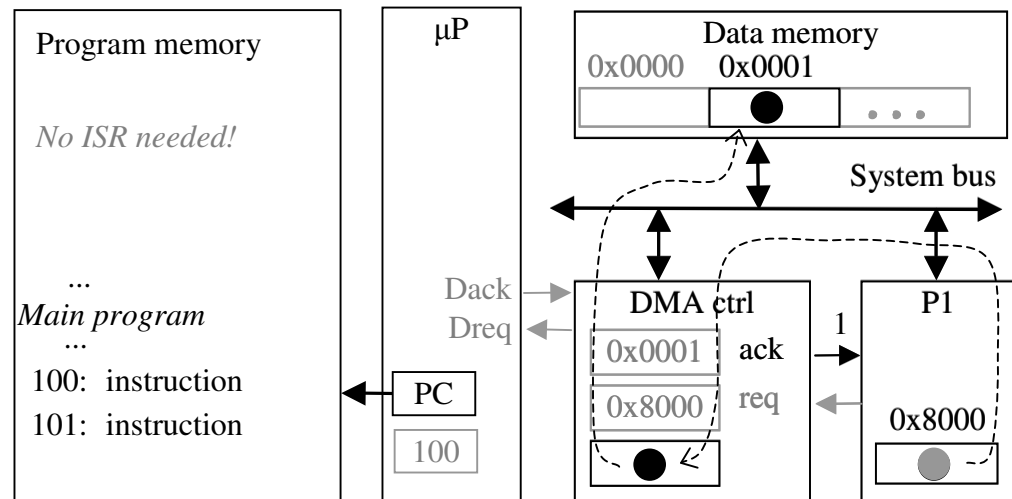


Peripheral to memory transfer with DMA



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

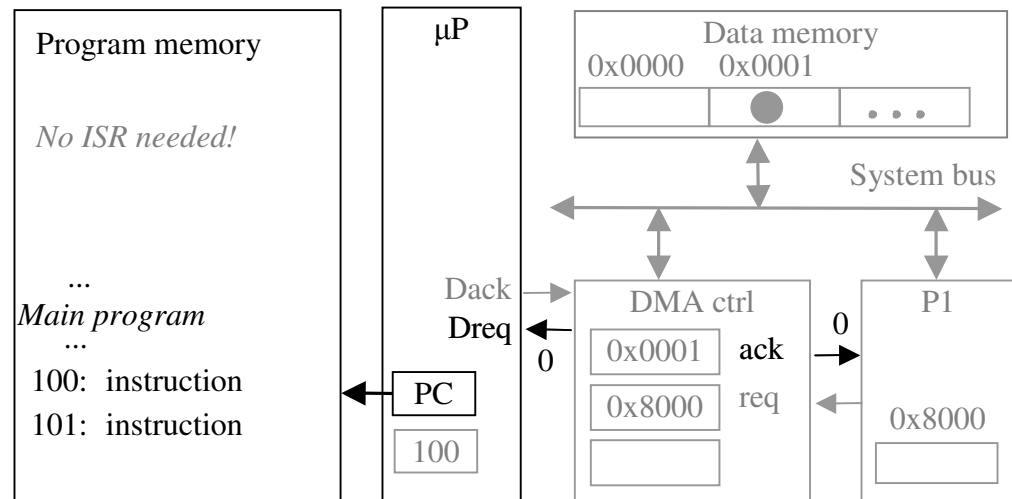
(Meanwhile, processor still executing if not stalled!)



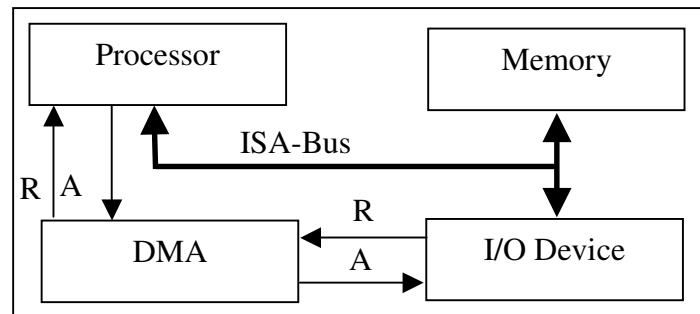
Peripheral to memory transfer with DMA



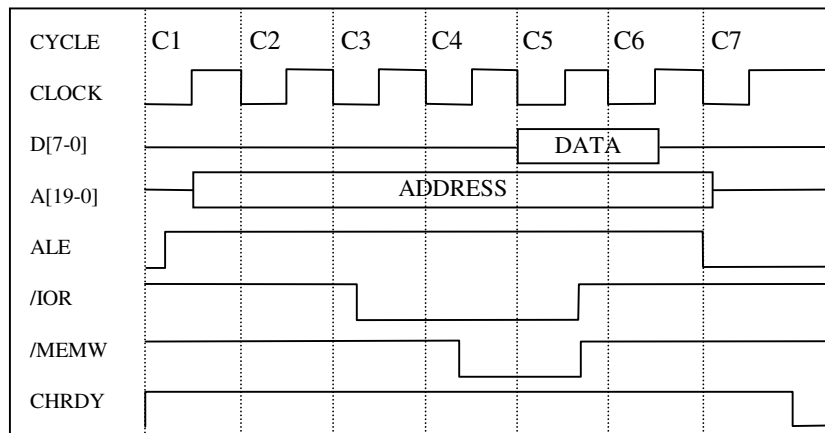
6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



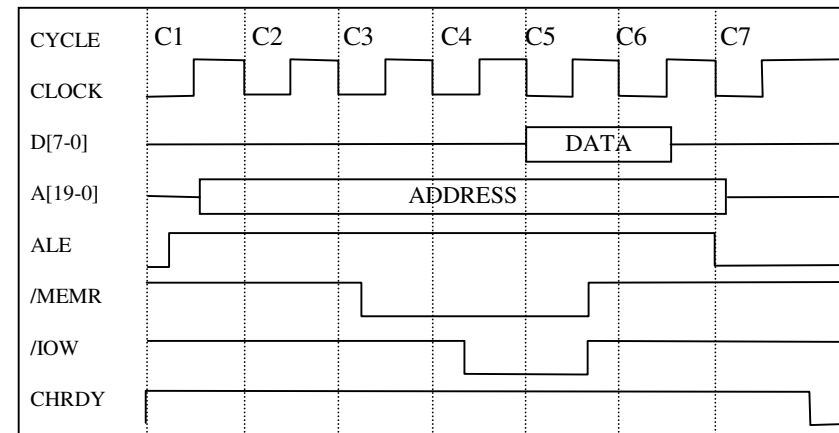
ISA bus DMA cycles



DMA Memory-Write Bus Cycle



DMA Memory-Read Bus Cycle



Outline

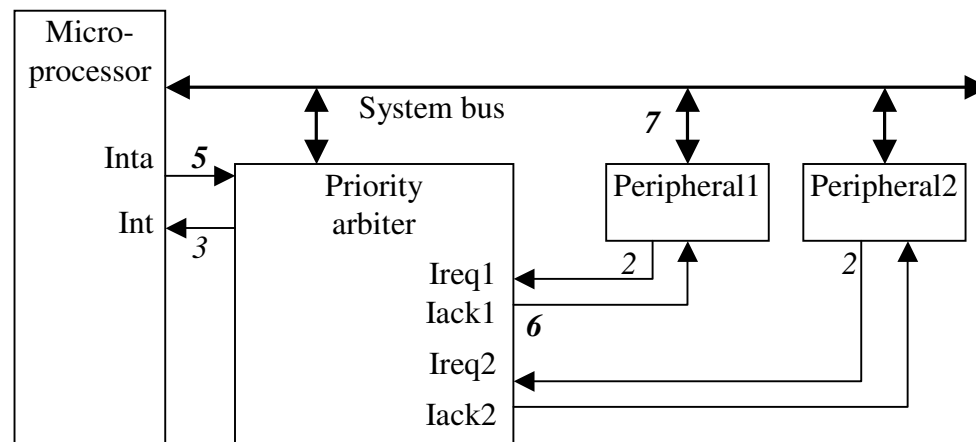


- Interfacing basics
- Microprocessor interfacing
 - ▶ I/O Addressing
 - ▶ Interrupts
 - ▶ Direct memory access
- Arbitration

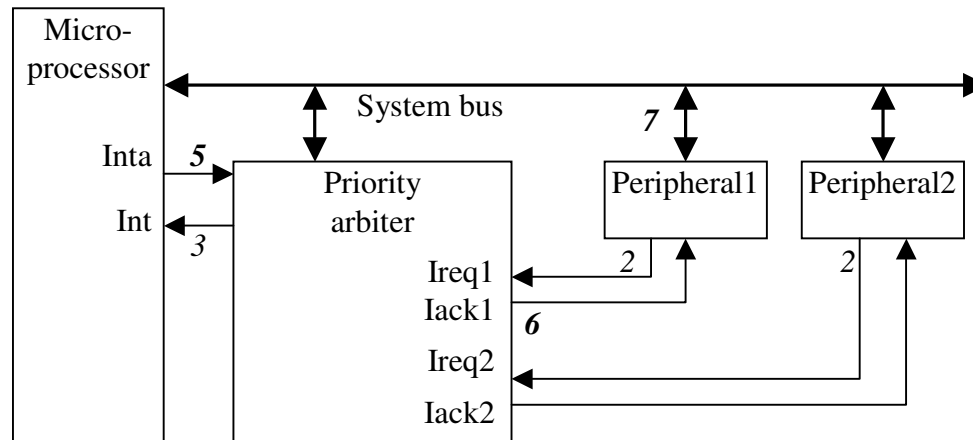
Arbitration: Priority arbiter



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

Arbitration: Priority arbiter

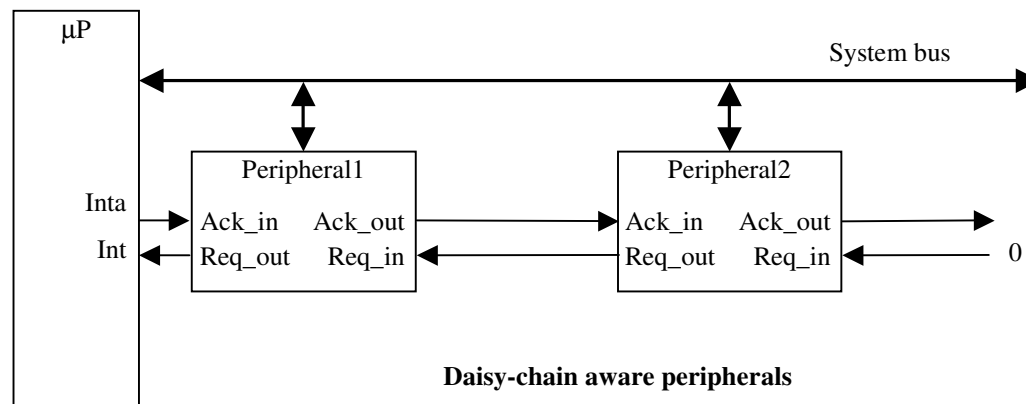


- Types of priority
 - Fixed priority
 - each peripheral has unique rank
 - highest rank chosen first with simultaneous requests
 - preferred when clear difference in rank between peripherals
 - Rotating priority (round-robin)
 - priority changed based on history of servicing
 - better distribution of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

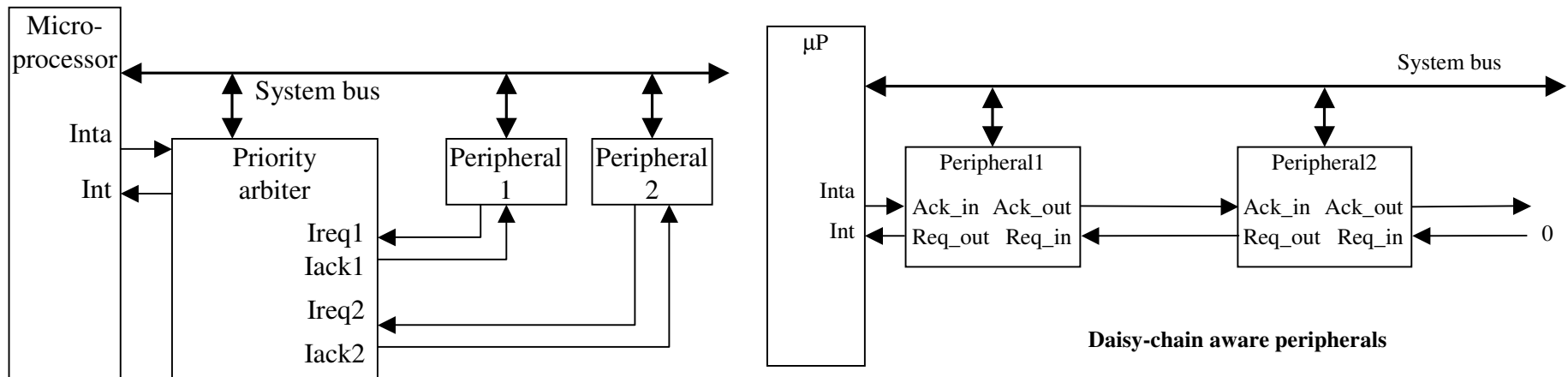


Arbitration: Daisy-chain arbitration



- 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

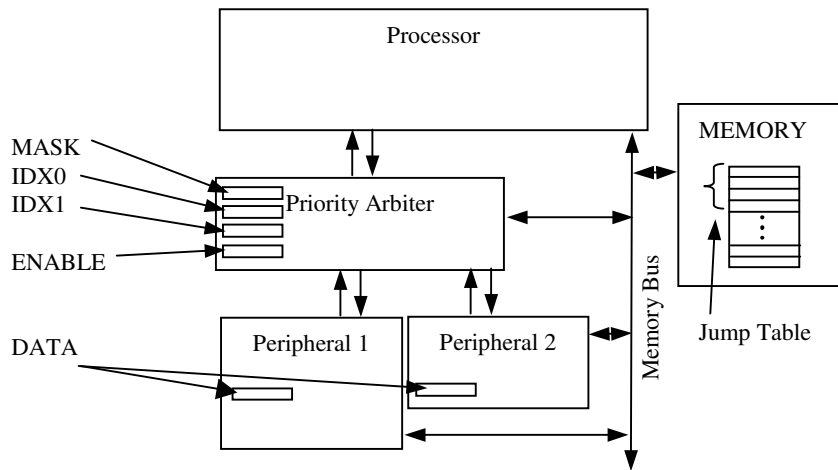


Network-oriented arbitration



- When multiple microprocessors share a bus (sometimes called a network)
 - ▶ Arbitration typically built into bus protocol
 - ▶ Separate processors may try to write simultaneously causing collisions
 - Data must be resent
 - Don't want to start sending again at same time
 - statistical methods can be used to reduce chances
- Typically used for connecting multiple distant chips
 - ▶ Trend - use to connect multiple on-chip processors

Example: Vectored interrupt using an interrupt table



```
unsigned char ARBITER_MASK_REG          _at_ 0xfff0;
unsigned char ARBITER_CH0_INDEX_REG     _at_ 0xfff1;
unsigned char ARBITER_CH1_INDEX_REG     _at_ 0xfff2;
unsigned char ARBITER_ENABLE_REG        _at_ 0xfff3;
unsigned char PERIPHERAL1_DATA_REG      _at_ 0xffe0;
unsigned char PERIPHERAL2_DATA_REG      _at_ 0xffe1;
unsigned void* INTERRUPT_LOOKUP_TABLE[256] _at_ 0x0100;

void main() {
    InitializePeripherals();
    for(;;) {} // main program goes here
}
```

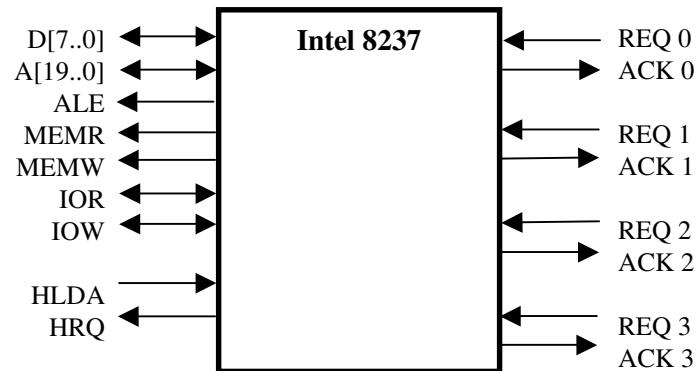
- Fixed priority: i.e., Peripheral1 has highest priority
- Keyword “_at_” followed by memory address forces compiler to place variables in specific memory locations
 - ▶ e.g., memory-mapped registers in arbiter, peripherals
- A peripheral’s index into interrupt table is sent to memory-mapped register in arbiter
- Peripherals receive external data and raise interrupt

```
void Peripheral1_ISR(void) {
    unsigned char data;
    data = PERIPHERAL1_DATA_REG;
    // do something with the data
}

void Peripheral2_ISR(void) {
    unsigned char data;
    data = PERIPHERAL2_DATA_REG;
    // do something with the data
}

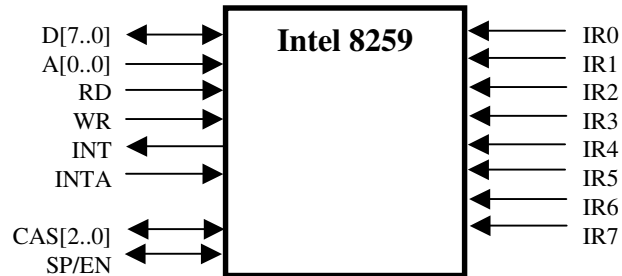
void InitializePeripherals(void) {
    ARBITER_MASK_REG = 0x03; // enable both channels
    ARBITER_CH0_INDEX_REG = 13;
    ARBITER_CH1_INDEX_REG = 17;
    INTERRUPT_LOOKUP_TABLE[13] = (void*)Peripheral1_ISR;
    INTERRUPT_LOOKUP_TABLE[17] = (void*)Peripheral2_ISR;
    ARBITER_ENABLE_REG = 1;
}
```

Intel 8237 DMA controller



Signal	Description
D[7..0]	These wires are connected to the system bus (ISA) and are used by the microprocessor to write to the internal registers of the 8237.
A[19..0]	These wires are connected to the system bus (ISA) and are used by the DMA to issue the memory location where the transferred data is to be written to. The 8237 is
ALE*	This is the address latch enable signal. The 8237 use this signal when driving the system bus (ISA).
MEMR*	This is the memory write signal issued by the 8237 when driving the system bus (ISA).
MEMW*	This is the memory read signal issued by the 8237 when driving the system bus (ISA).
IOR*	This is the I/O device read signal issued by the 8237 when driving the system bus (ISA) in order to read a byte from an I/O device
IOW*	This is the I/O device write signal issued by the 8237 when driving the system bus (ISA) in order to write a byte to an I/O device.
HLDA	This signal (hold acknowledge) is asserted by the microprocessor to signal that it has relinquished the system bus (ISA).
HRQ	This signal (hold request) is asserted by the 8237 to signal to the microprocessor a request to relinquish the system bus (ISA).
REQ 0,1,2,3	An attached device to one of these channels asserts this signal to request a DMA transfer.
ACK 0,1,2,3	The 8237 asserts this signal to grant a DMA transfer to an attached device to one of these channels.
*See the ISA bus description in this chapter for complete details.	

Intel 8259 programmable priority controller



Signal	Description
D[7..0]	These wires are connected to the system bus and are used by the microprocessor to write or read the internal registers of the 8259.
A[0..0]	This pin acts in conjunction with WR/RD signals. It is used by the 8259 to decipher various command words the microprocessor writes and status the microprocessor wishes to read.
WR	When this write signal is asserted, the 8259 accepts the command on the data line, i.e., the microprocessor writes to the 8259 by placing a command on the data lines and asserting this signal.
RD	When this read signal is asserted, the 8259 provides on the data lines its status, i.e., the microprocessor reads the status of the 8259 by asserting this signal and reading the data lines.
INT	This signal is asserted whenever a valid interrupt request is received by the 8259, i.e., it is used to interrupt the microprocessor.
INTA	This signal, is used to enable 8259 interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the microprocessor.
IR 0,1,2,3,4,5,6,7	An interrupt request is executed by a peripheral device when one of these signals is asserted.
CAS[2..0]	These are cascade signals to enable multiple 8259 chips to be chained together.
SP/EN	This function is used in conjunction with the CAS signals for cascading purposes.