

Embedded Systems Design: A Unified Hardware/Software Introduction

Chapter 6 Interfacing

Outline

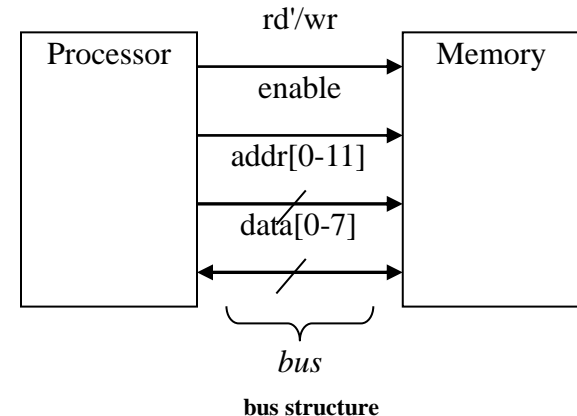
- Interfacing basics
- Microprocessor interfacing
 - I/O Addressing
 - Interrupts
 - Direct memory access
- Arbitration
- Hierarchical buses
- Protocols
 - Serial
 - Parallel
 - Wireless

Introduction

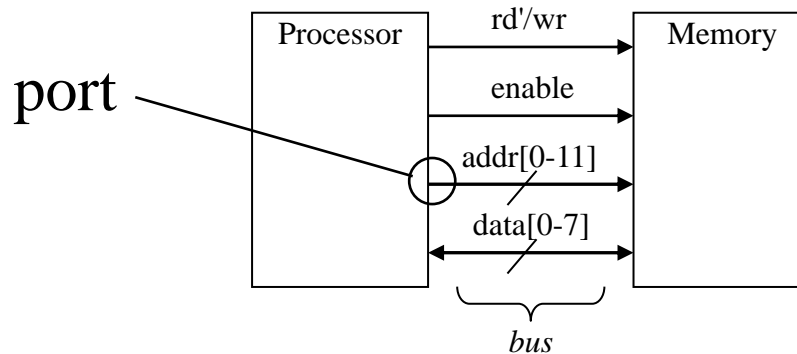
- Embedded system functionality aspects
 - Processing
 - Transformation of data
 - Implemented using processors
 - Storage
 - Retention of data
 - Implemented using memory
 - Communication
 - Transfer of data between processors and memories
 - Implemented using buses
 - Called *interfacing*

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

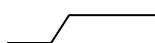
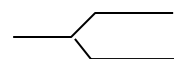


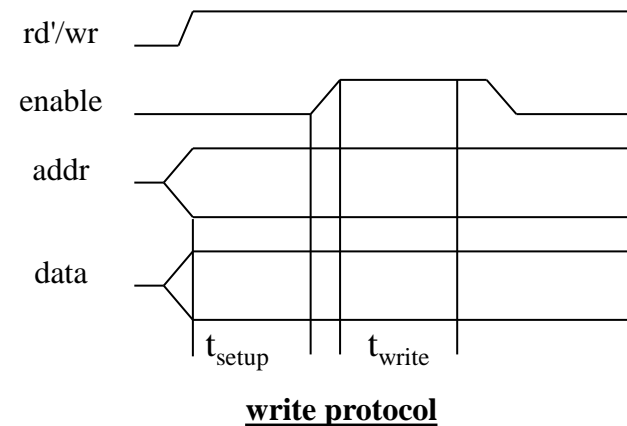
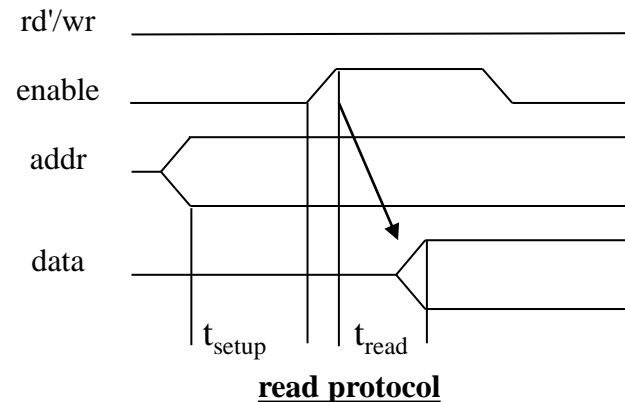
Ports



- 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

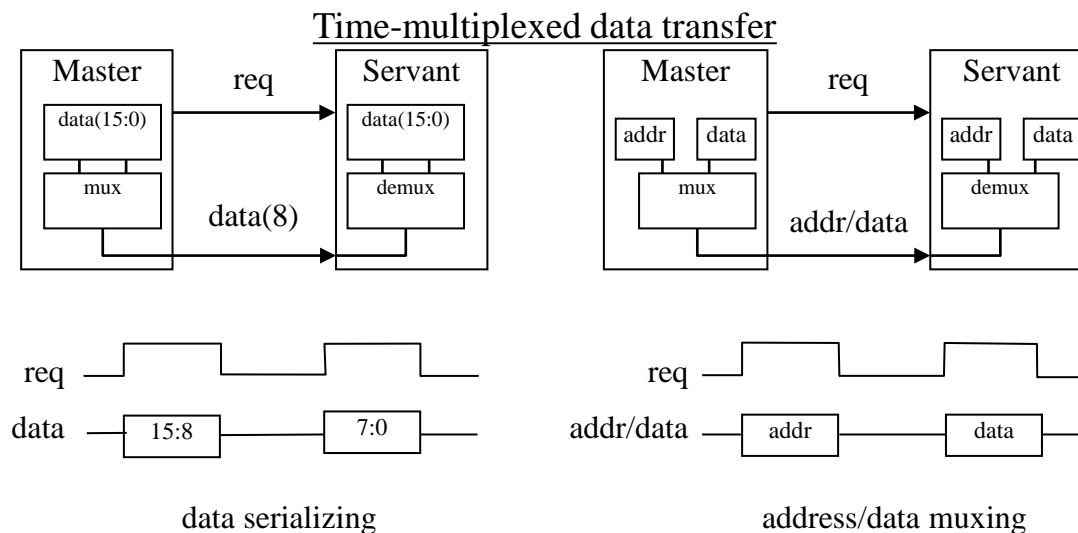
Timing Diagrams

- 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 (e.g., *go'*, */go*, or *go_L*)
 - Use terms *assert* (active) and *deassert*
 - Asserting *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}

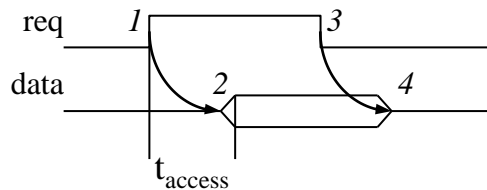
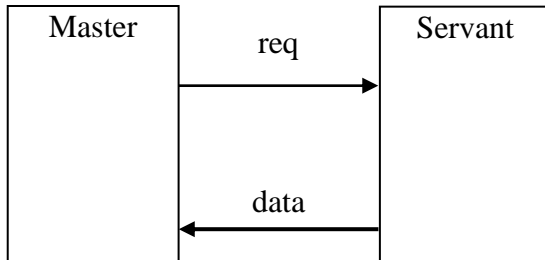


Basic protocol concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses: special kind of data
 - 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

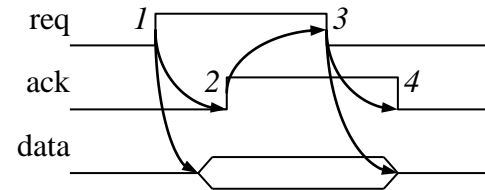
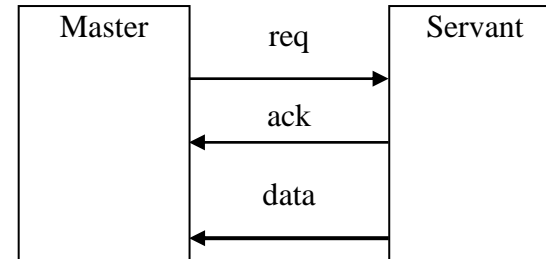


Basic protocol concepts: control methods



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

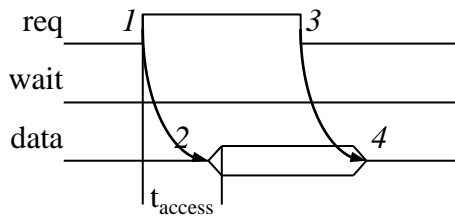
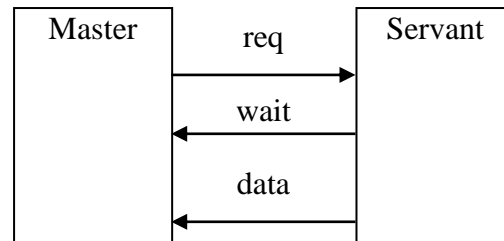
Strobe protocol



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

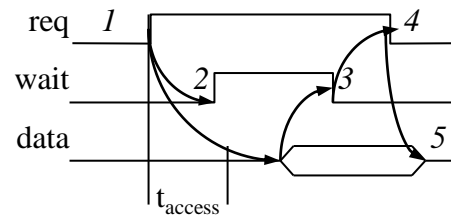
Handshake protocol

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

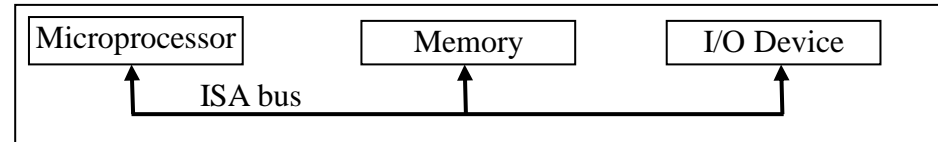


1. Master asserts *req* to receive data
2. Servant can't put data within t_{access} , **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

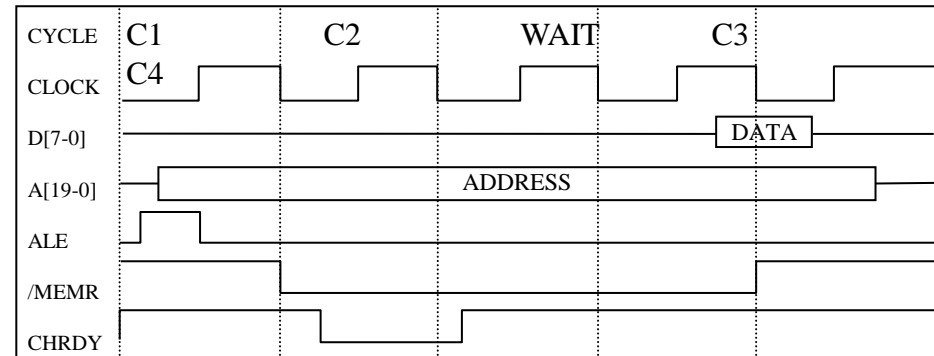
Slow-response case

ISA bus protocol – memory access

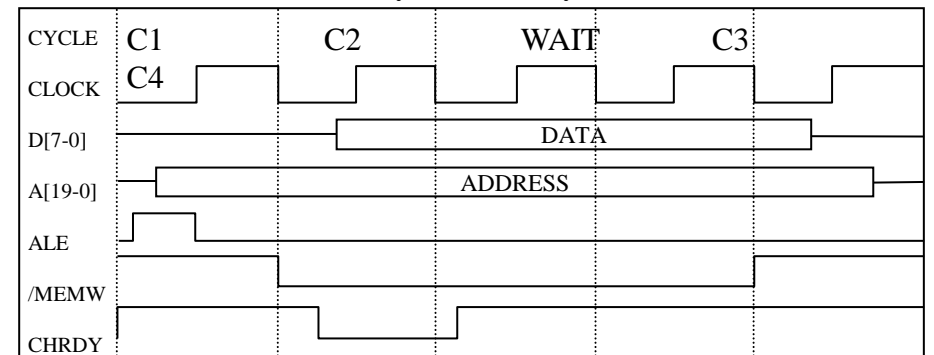
- ISA: Industry Standard Architecture
 - Common in 80x86's
- Features
 - 20-bit address
 - Compromise strobe/handshake control
 - 4 cycles default
 - Unless CHRDY deasserted – resulting in additional wait cycles (up to 6)



memory-read bus cycle



memory-write bus cycle

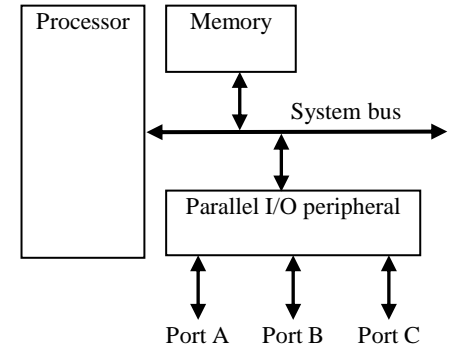


Microprocessor interfacing: I/O addressing

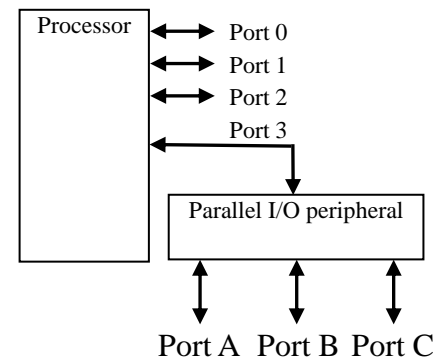
- A microprocessor communicates with other devices using some of its pins
 - Port-based I/O (parallel I/O)
 - Processor has one or more N-bit ports
 - Processor's software reads and writes a port just like a register
 - E.g., $P0 = 0xFF$; $v = P1.2$; -- P0 and P1 are 8-bit ports
 - Bus-based I/O
 - Processor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

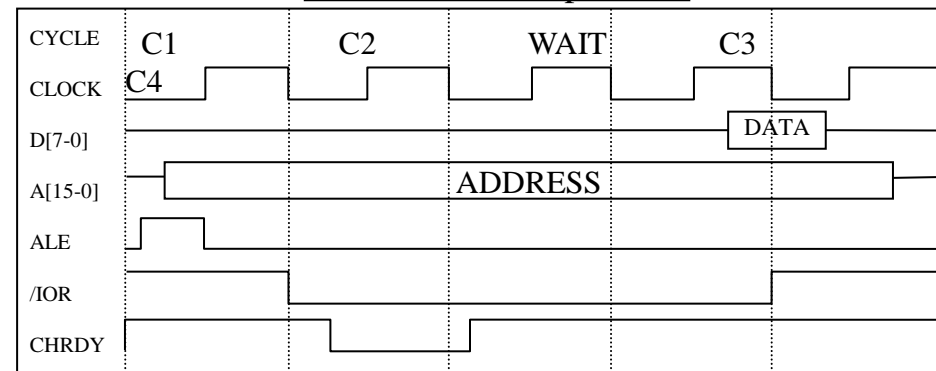
Memory-mapped I/O vs. Standard I/O

- Memory-mapped I/O
 - Requires no special instructions
 - Assembly instructions involving memory like MOV 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

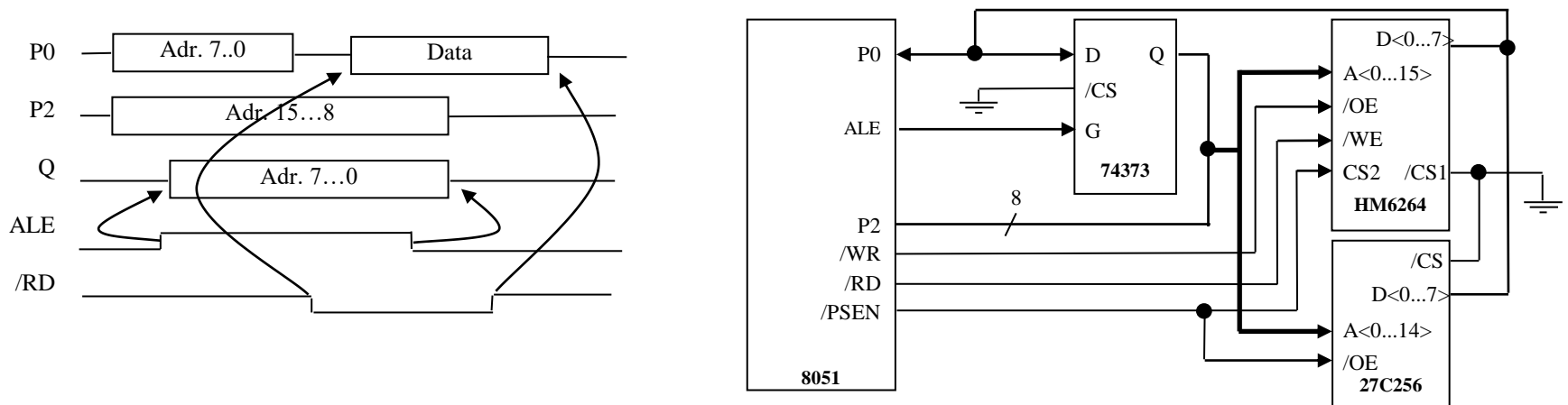
ISA bus

- 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

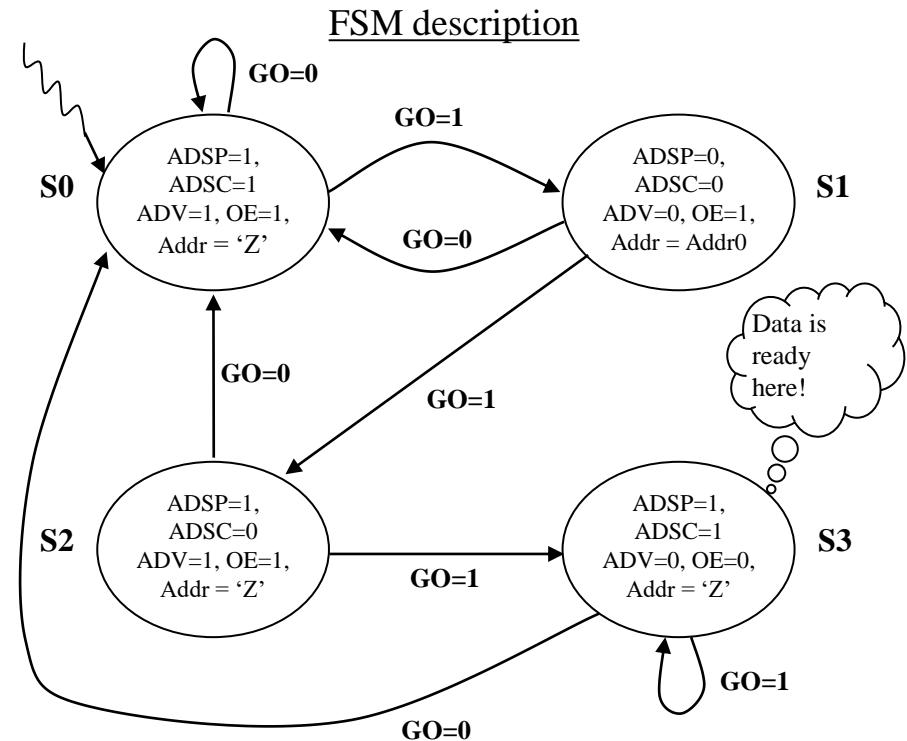
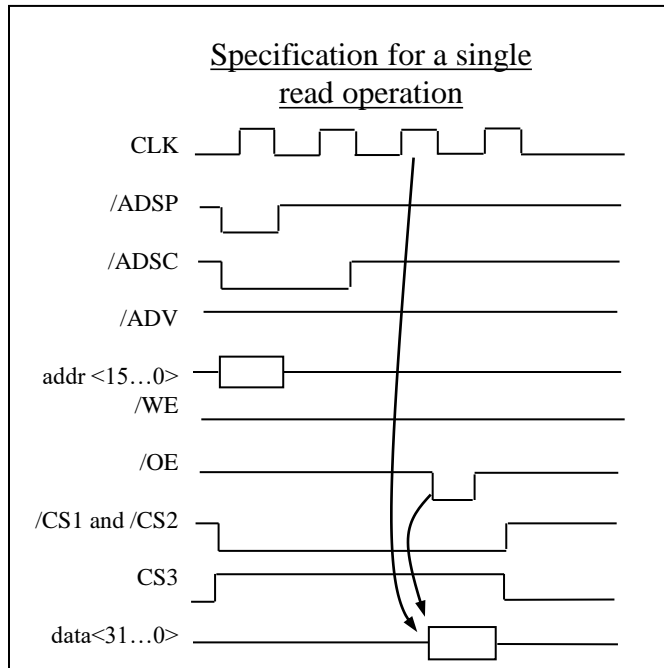


A basic memory protocol



- Interfacing an 8051 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

A more complex memory protocol



- Generates control signals to drive the TC55V2325FF memory chip in burst mode
 - Addr0* is the starting address input to device
 - GO* is enable/disable input to device

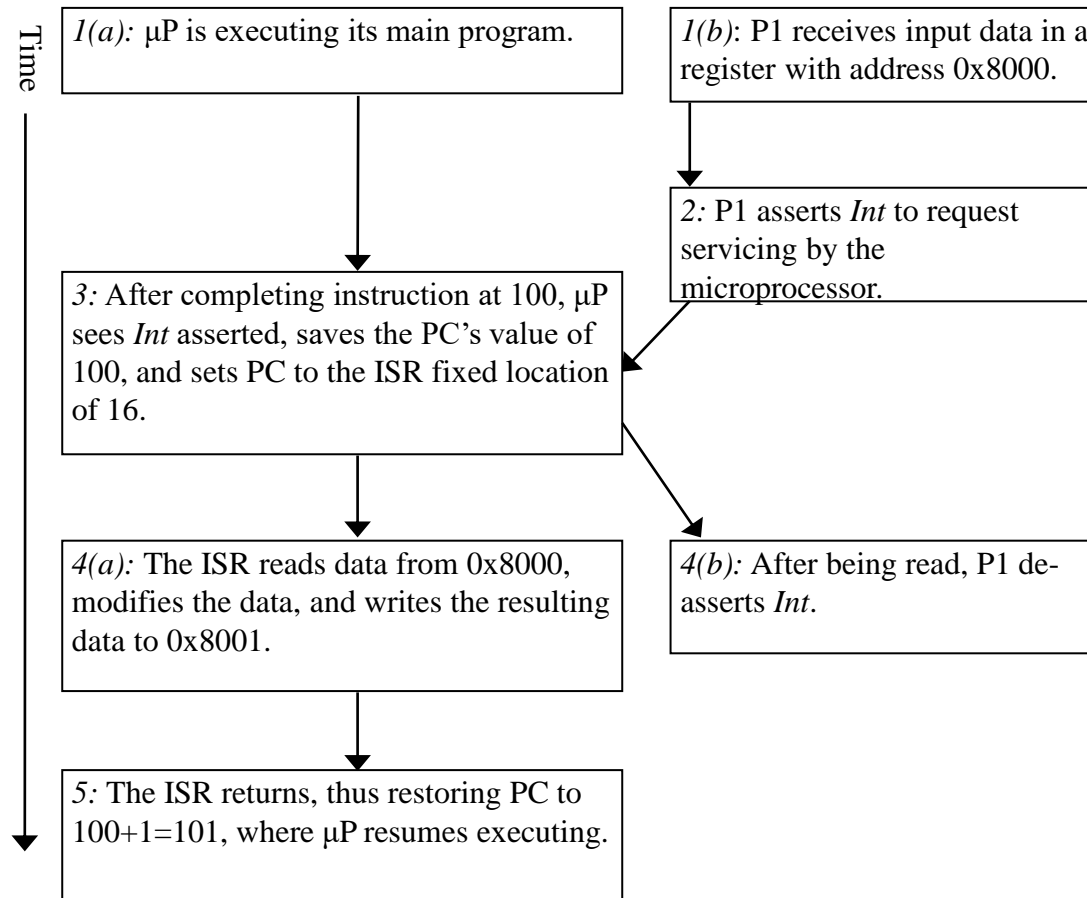
Microprocessor interfacing: interrupts

- 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

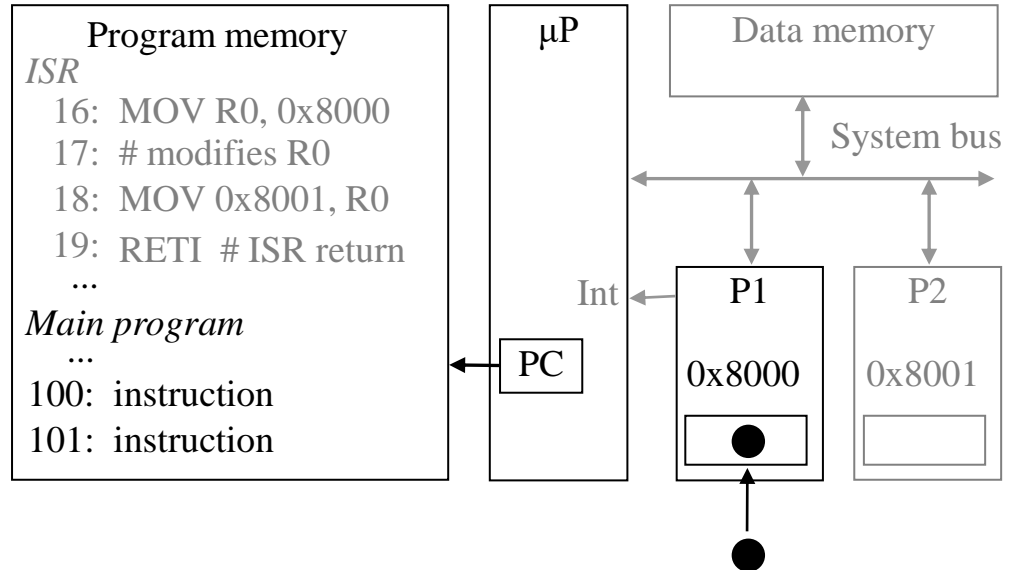
Interrupt-driven I/O using fixed ISR location



Interrupt-driven I/O using fixed ISR location

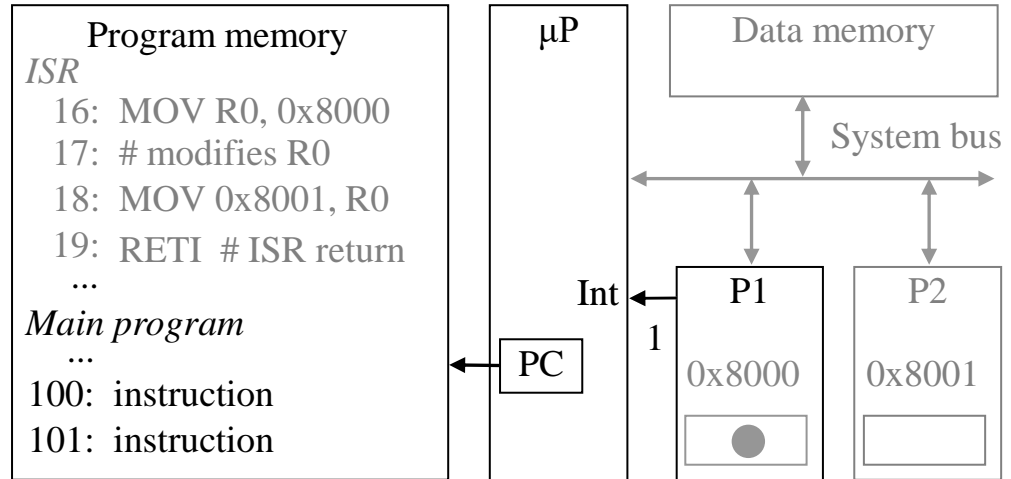
1(a): μP is executing its main program

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



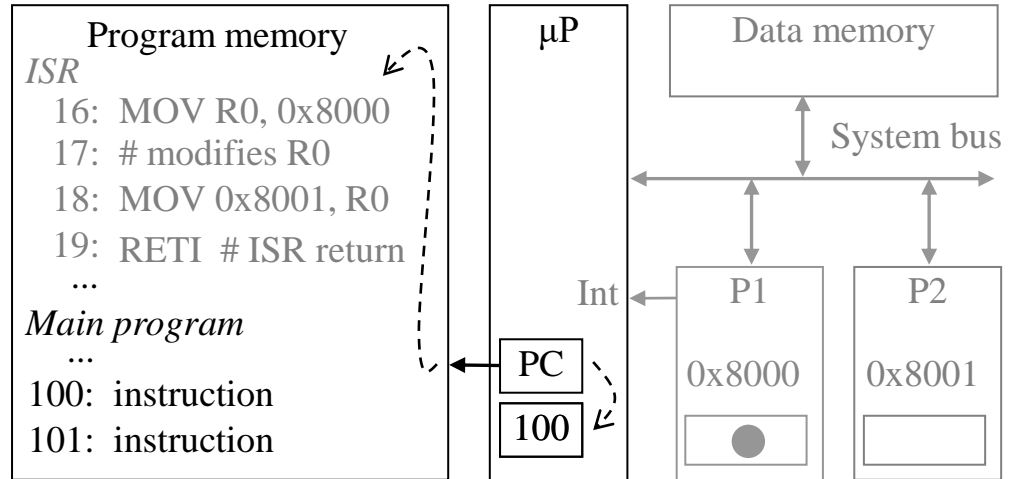
Interrupt-driven I/O using fixed ISR location

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



Interrupt-driven I/O using fixed ISR location

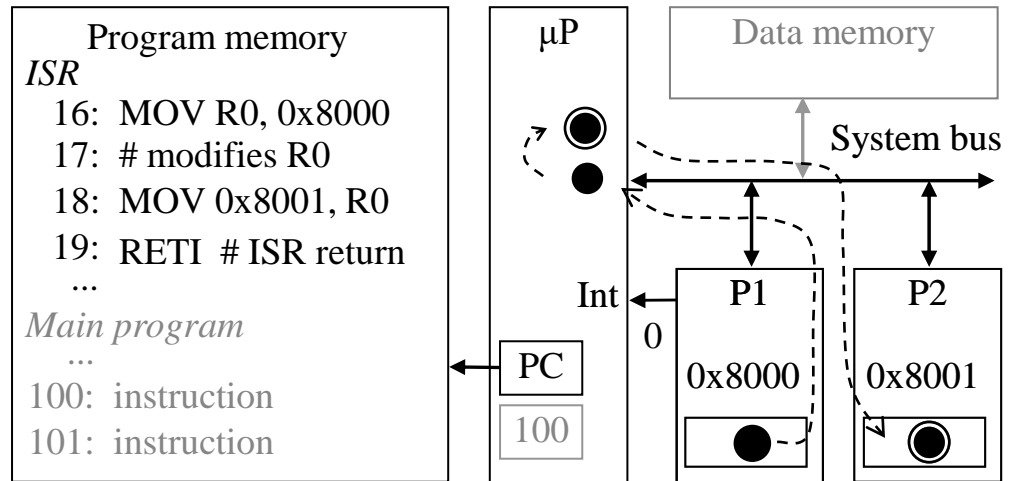
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location

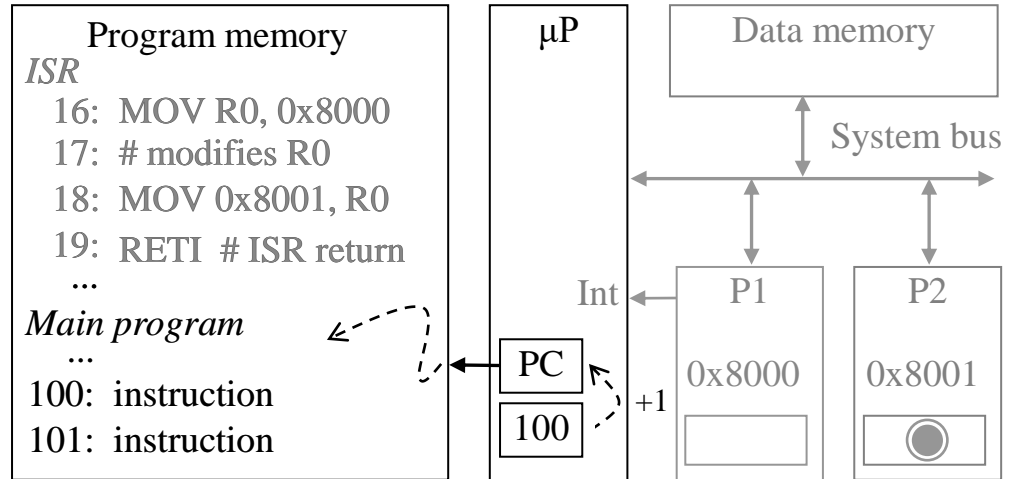
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

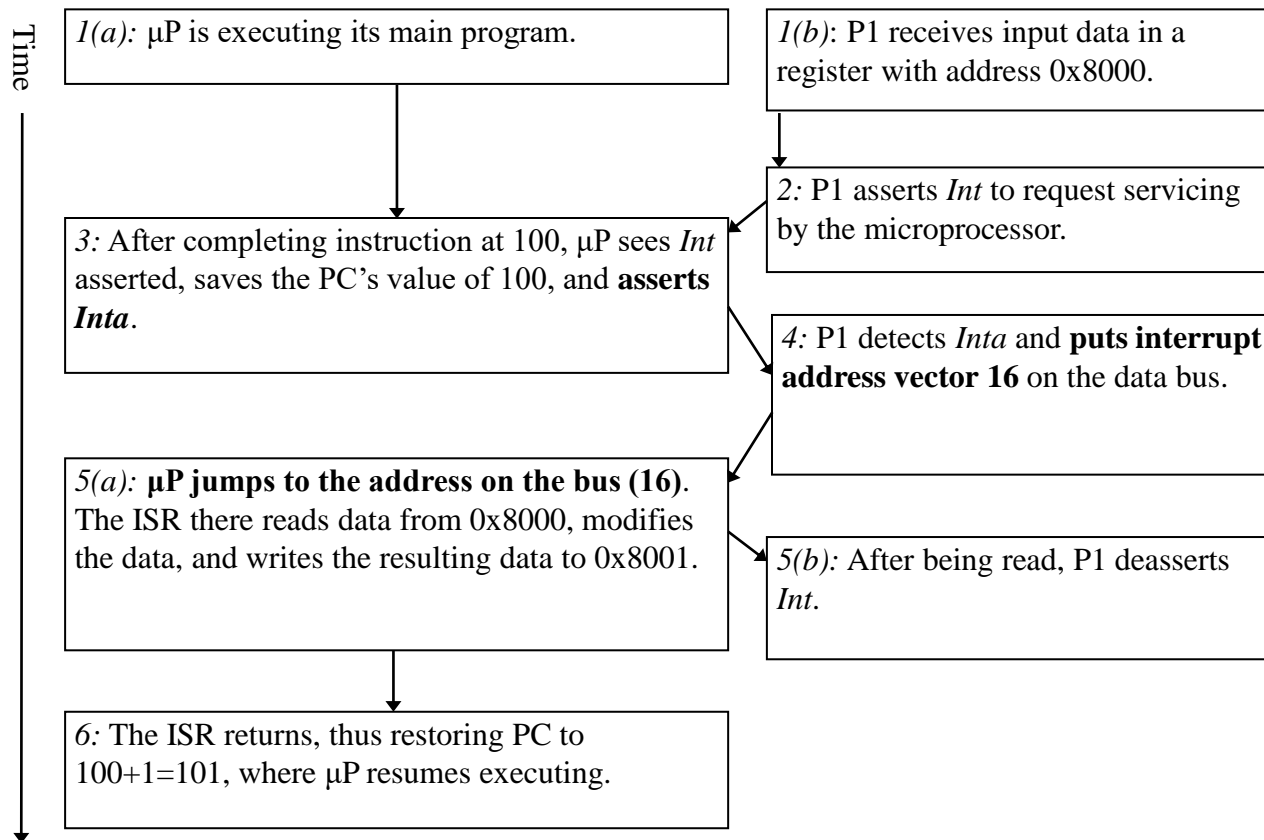


Interrupt-driven I/O using fixed ISR location

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



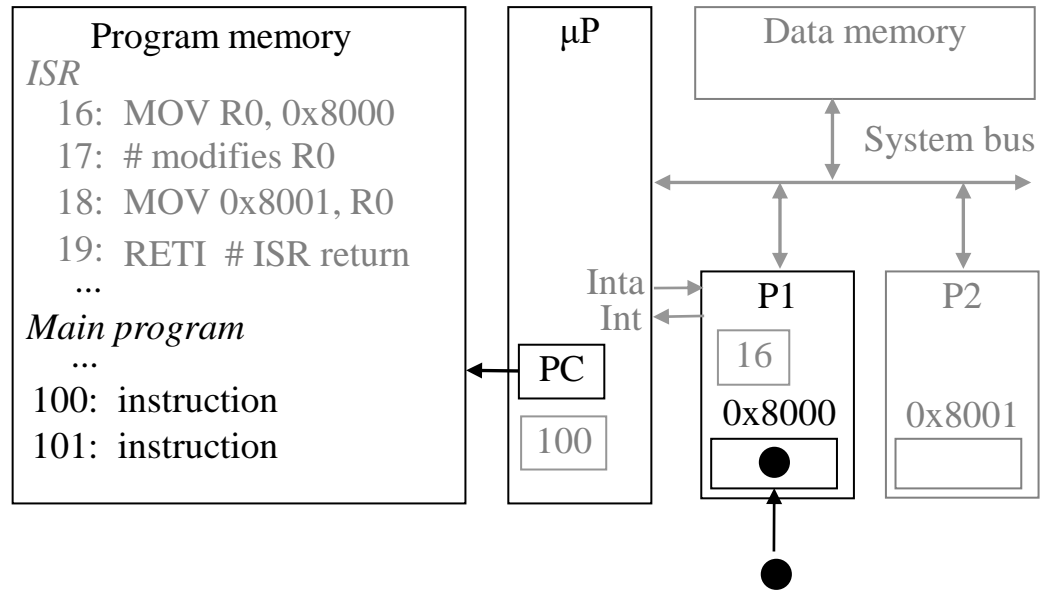
Interrupt-driven I/O using vectored interrupt



Interrupt-driven I/O using vectored interrupt

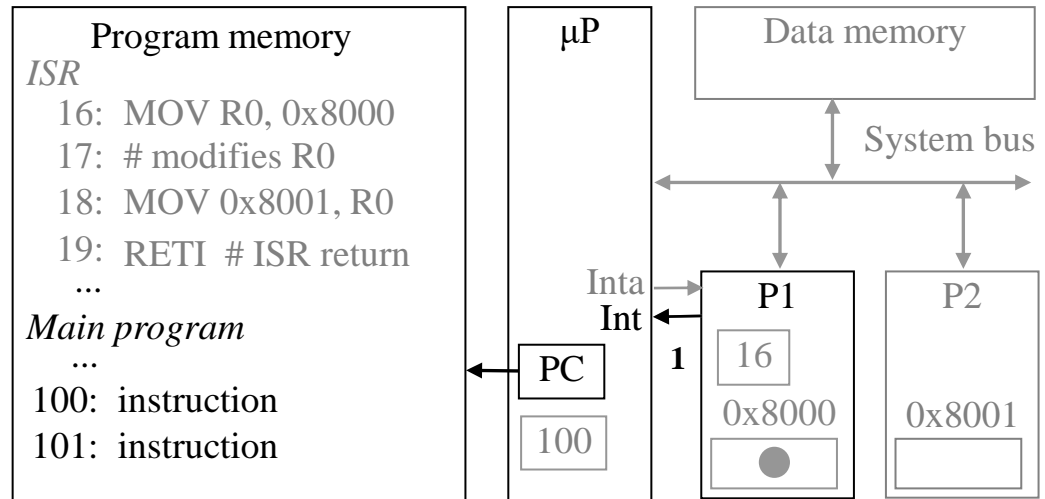
1(a): P is executing its main program

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



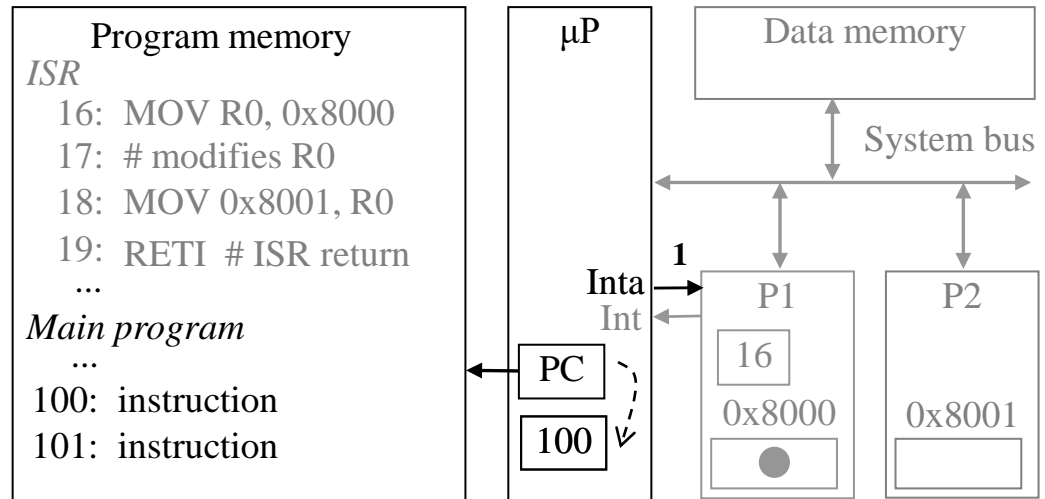
Interrupt-driven I/O using vectored interrupt

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



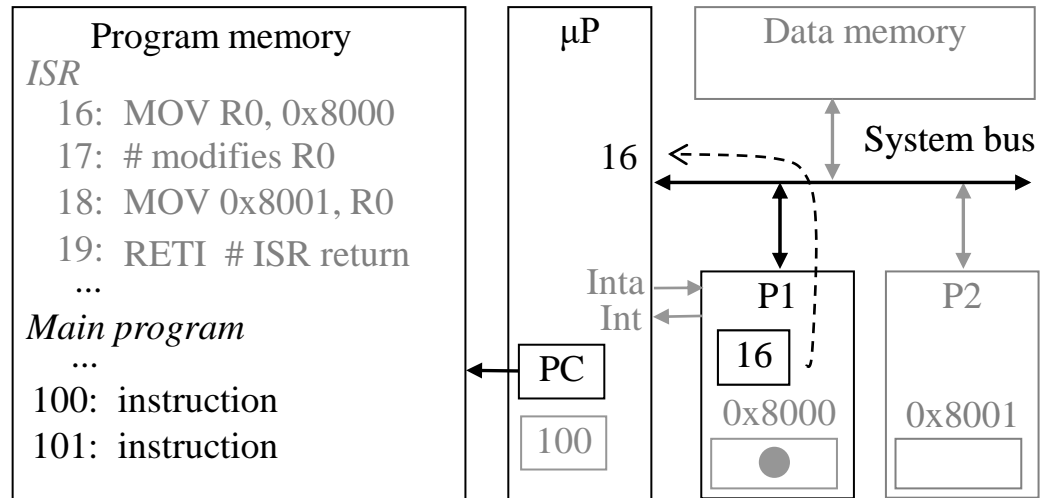
Interrupt-driven I/O using vectored interrupt

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

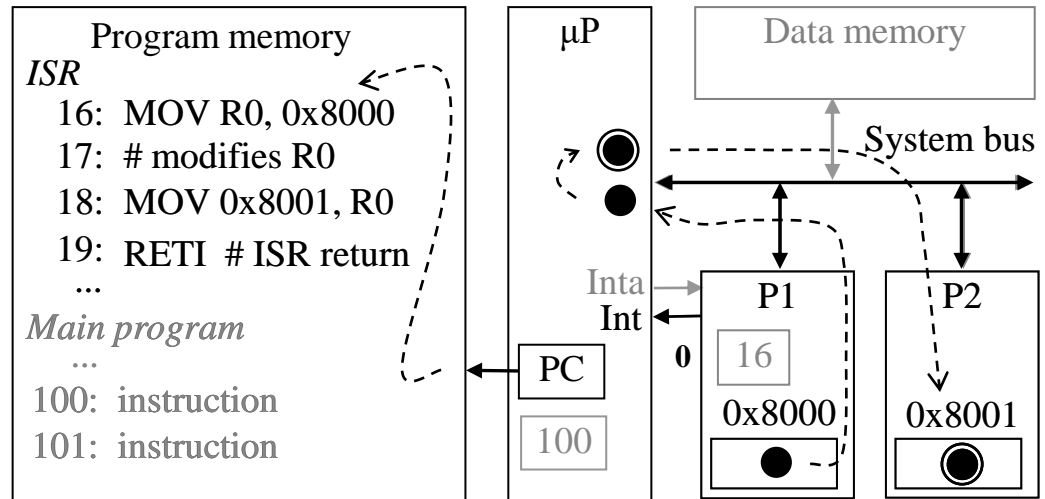
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-driven I/O using vectored interrupt

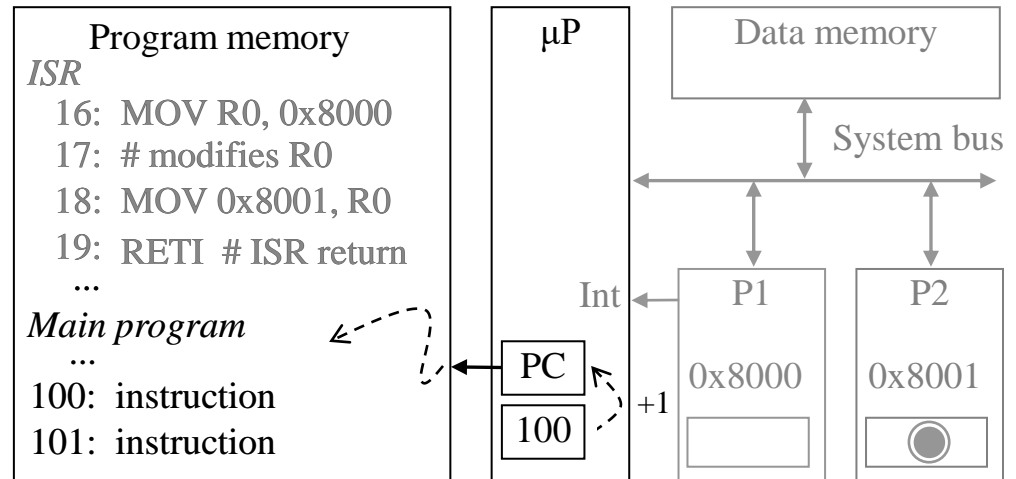
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

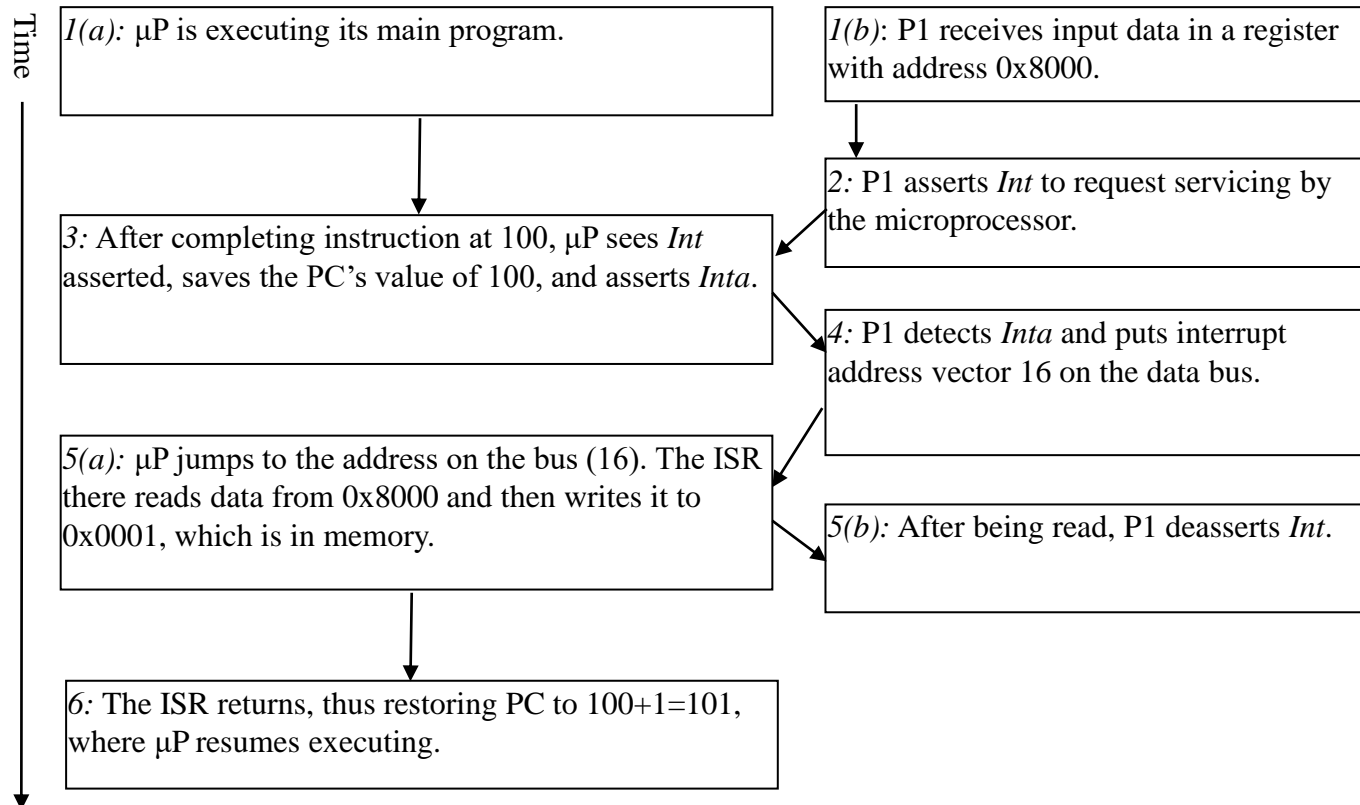
Additional interrupt issues

- 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

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

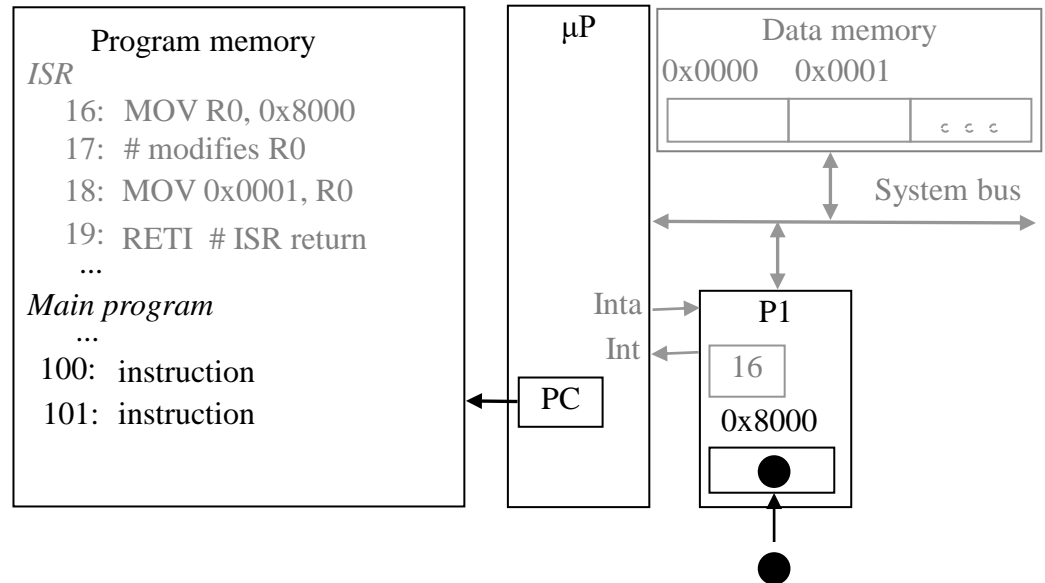
Peripheral to memory transfer *without* DMA, using vectored interrupt



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

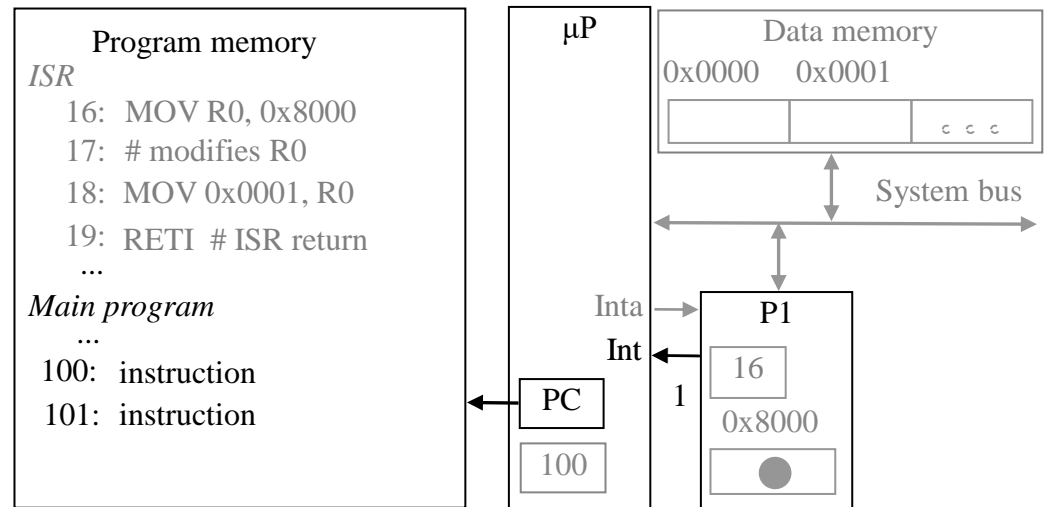
1(a): μP is executing its main program

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



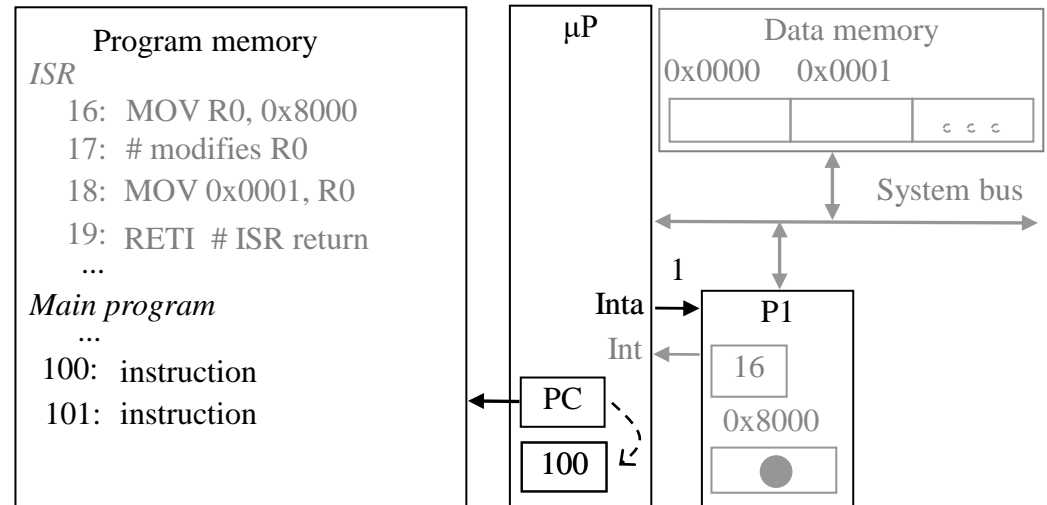
Peripheral to memory transfer *without* DMA, using vectored interrupt

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



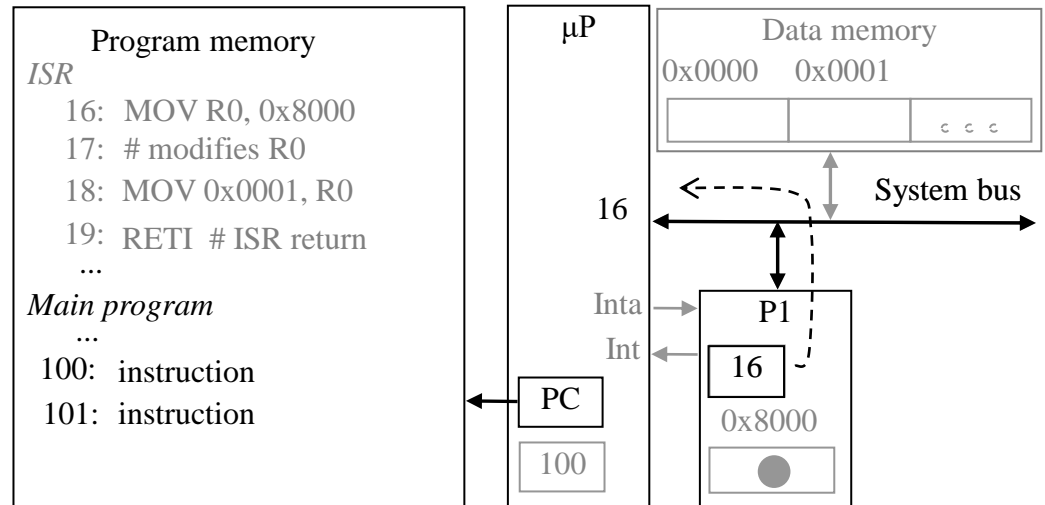
Peripheral to memory transfer *without* DMA, using vectored interrupt

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')

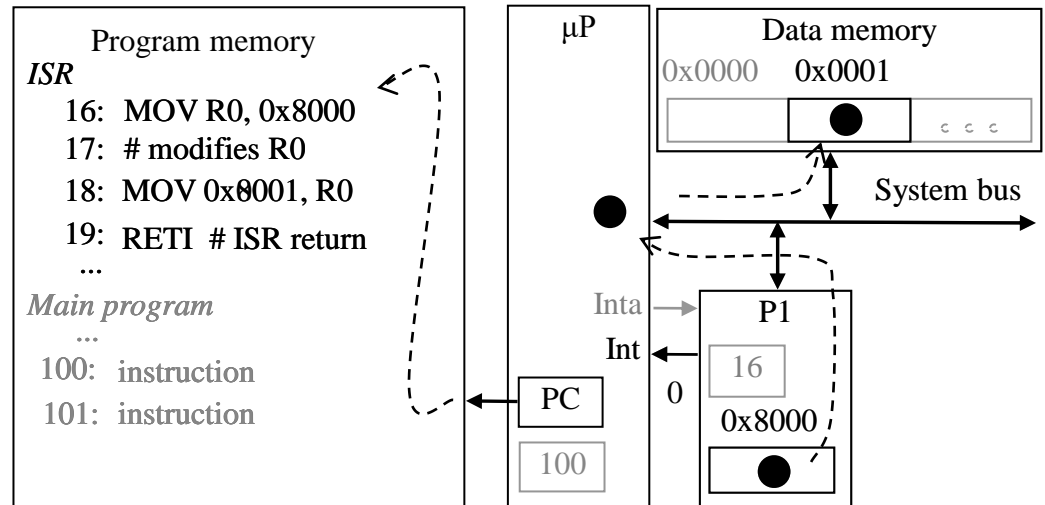
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



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

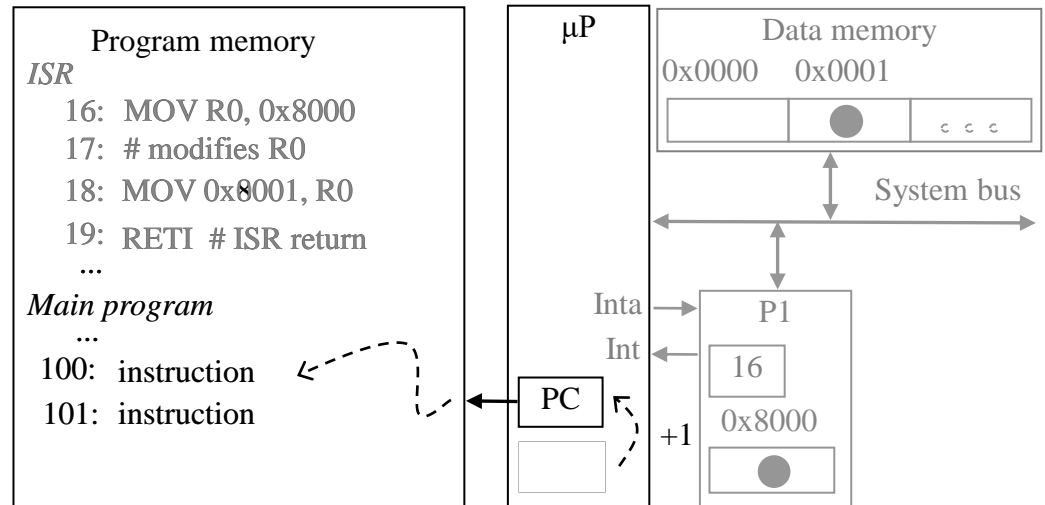
5(a): μ P jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

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

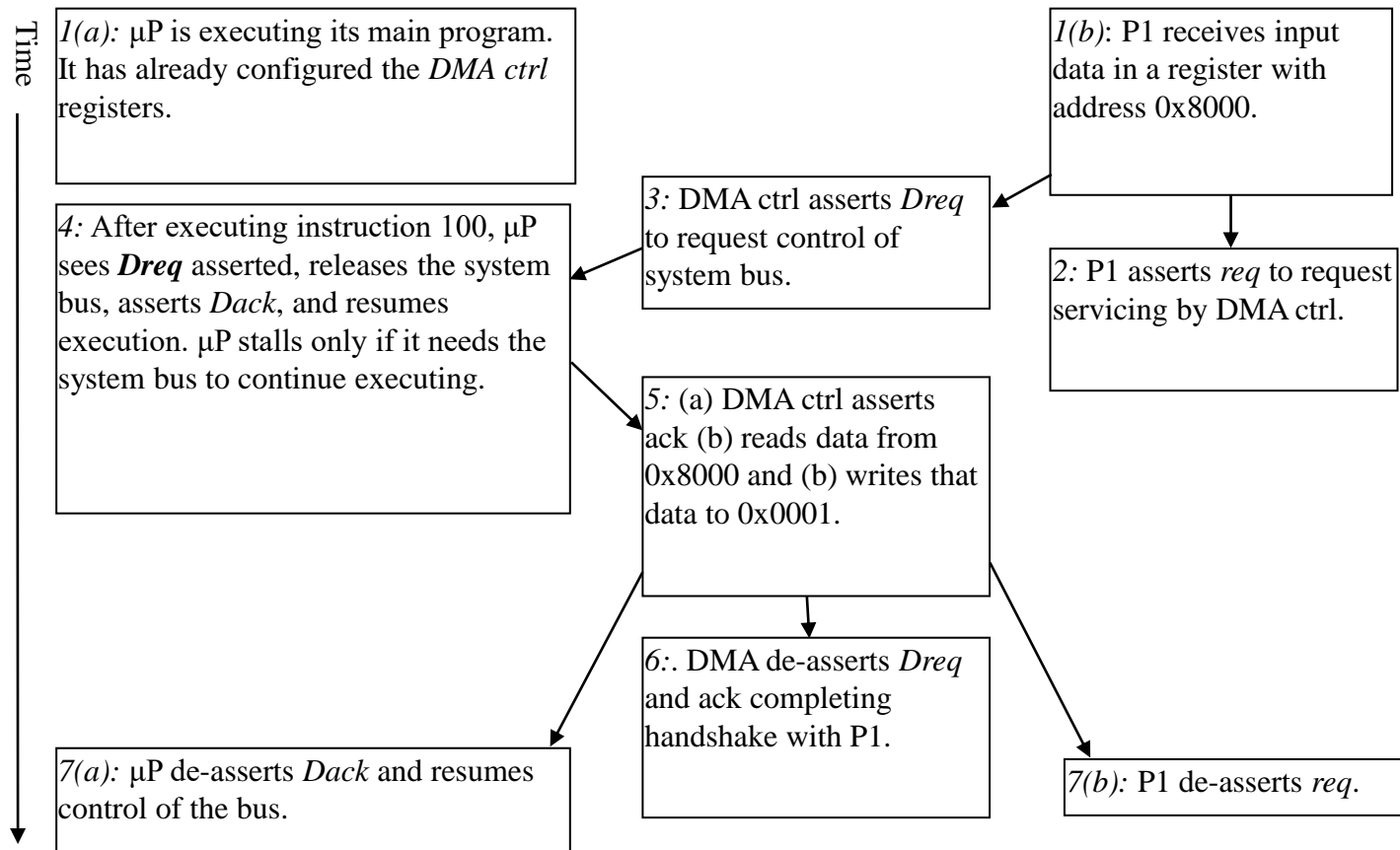


Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

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



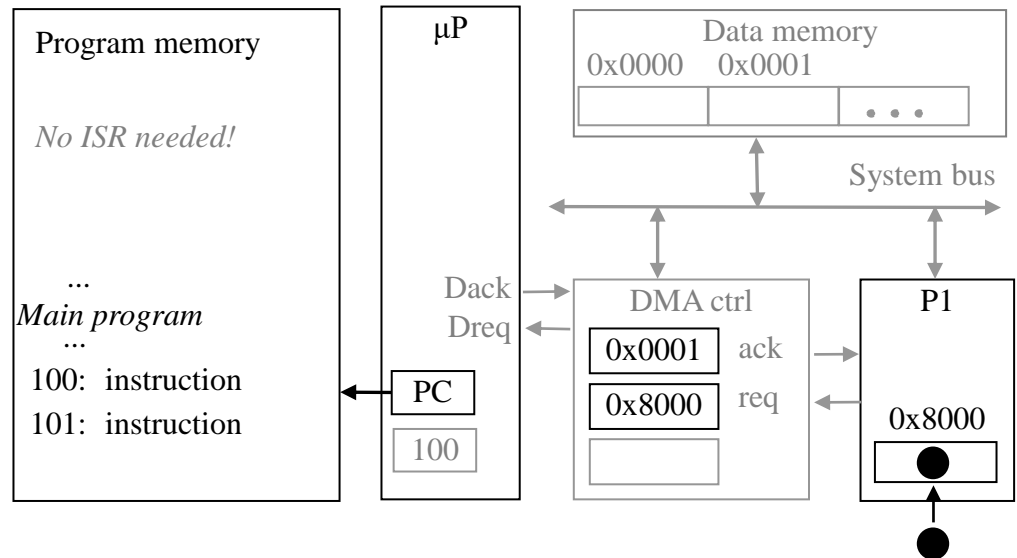
Peripheral to memory transfer with DMA



Peripheral to memory transfer with DMA (cont')

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

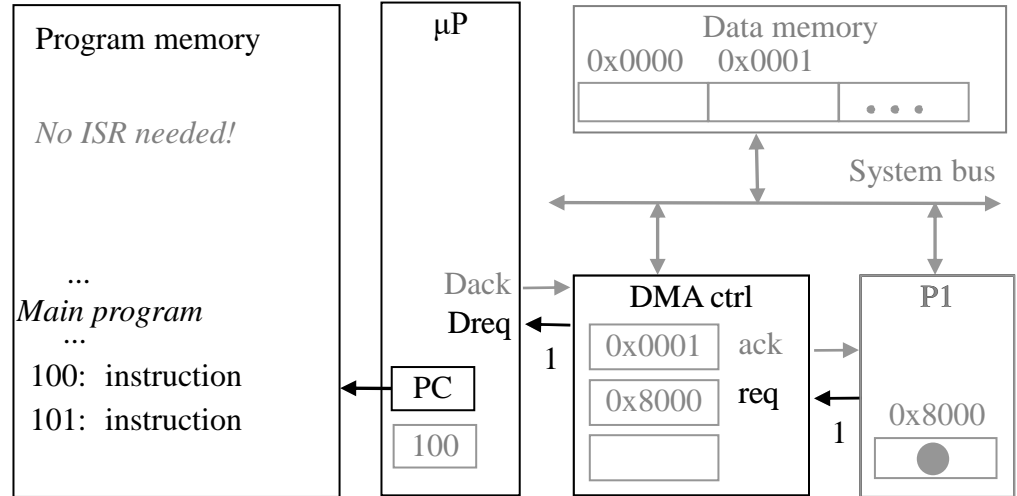
1(b): P1 receives input data in a register with address 0x8000.



Peripheral to memory transfer with DMA (cont')

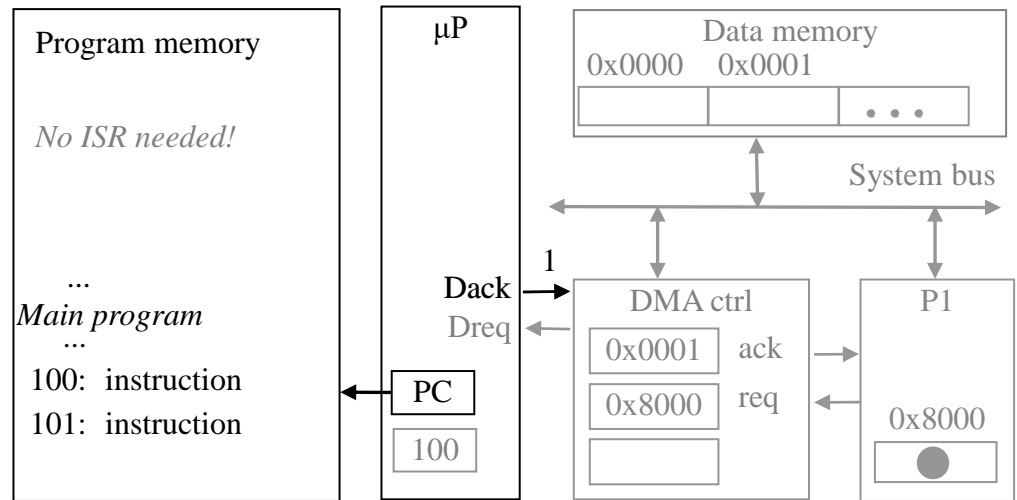
2: P1 asserts *req* to request servicing by DMA ctrl.

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



Peripheral to memory transfer with DMA (cont')

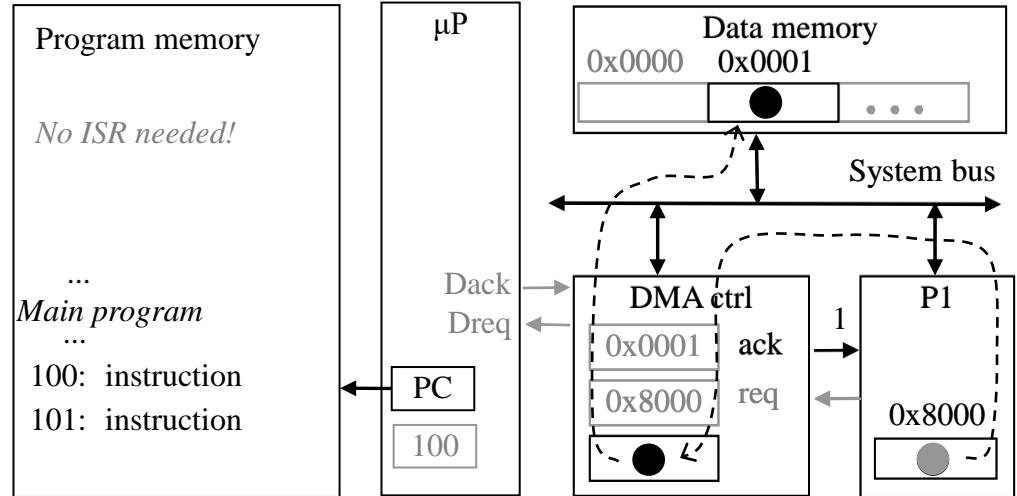
4: After executing instruction 100, μP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, μP stalls only if it needs the system bus to continue executing.



Peripheral to memory transfer with DMA (cont')

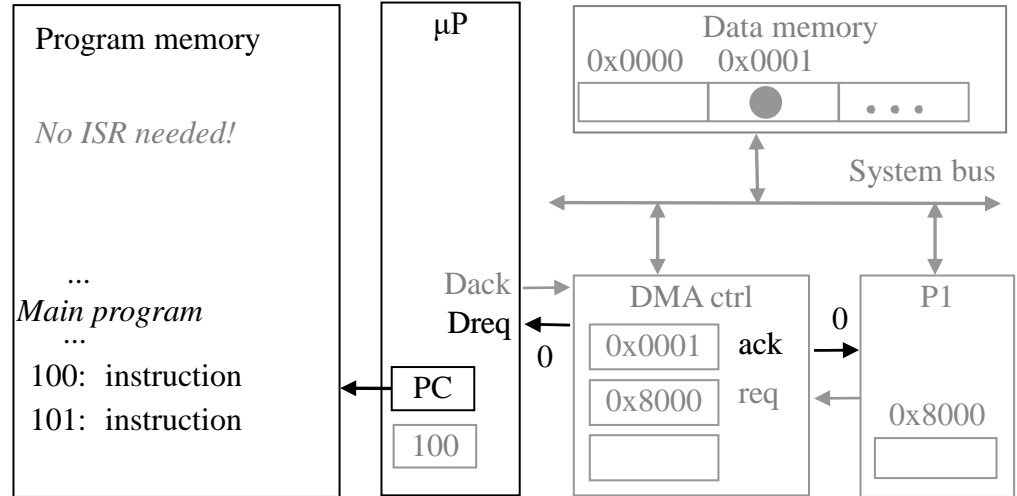
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)

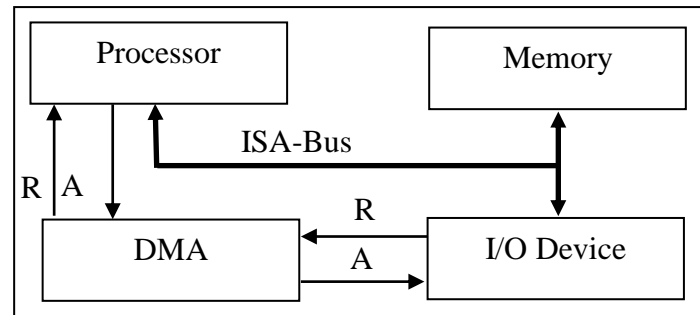


Peripheral to memory transfer with DMA (cont')

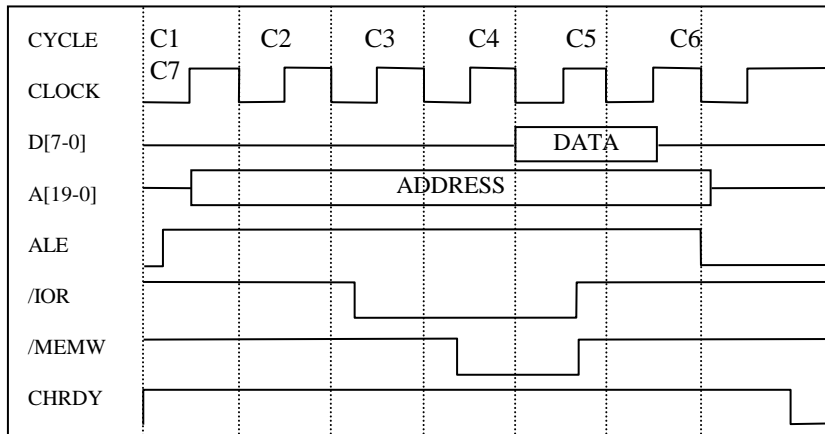
6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



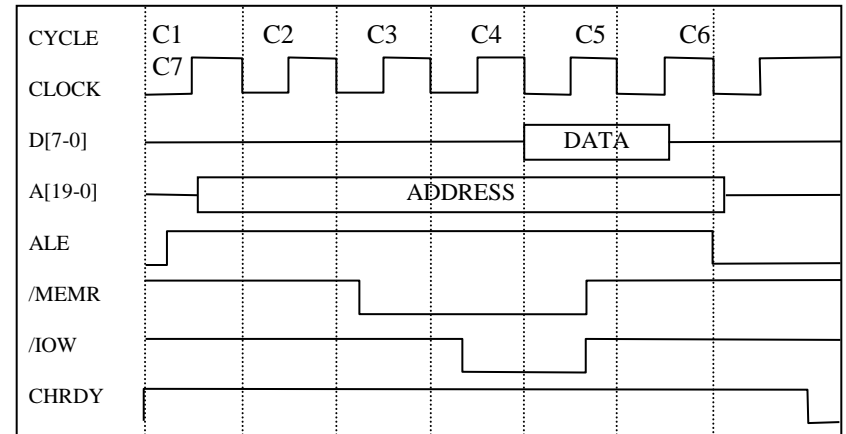
ISA bus DMA cycles



DMA Memory-Write Bus Cycle

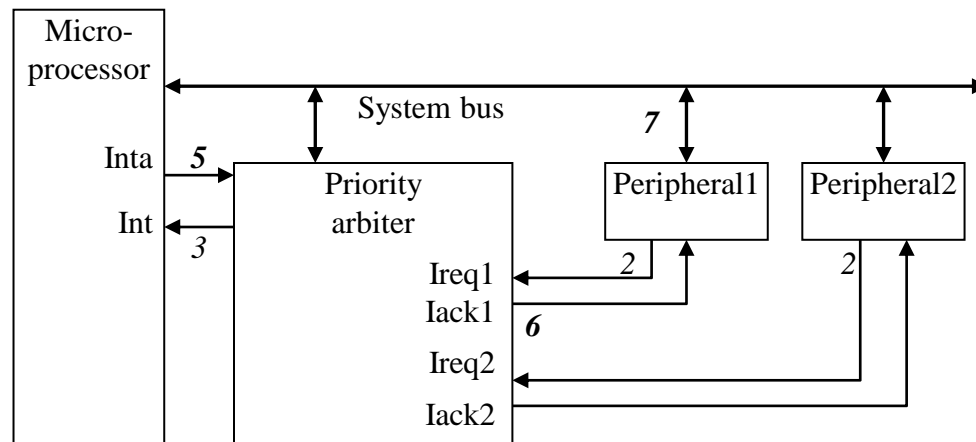


DMA Memory-Read Bus Cycle

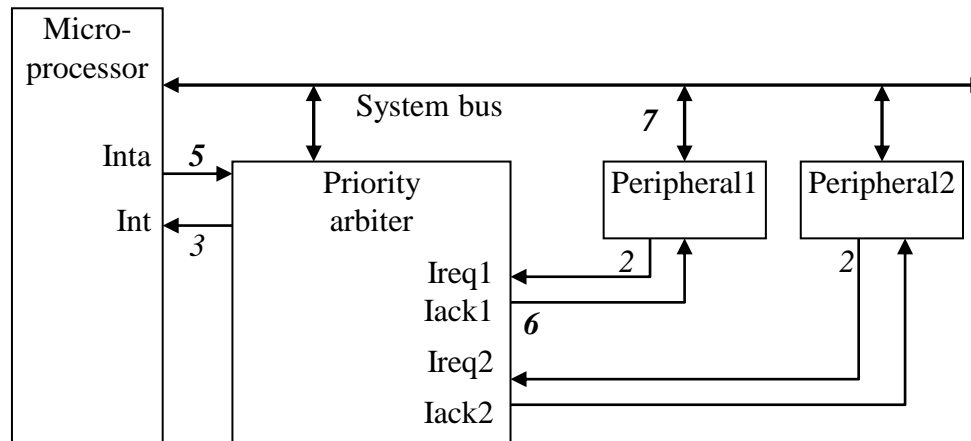


Arbitration: Priority arbiter

- 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



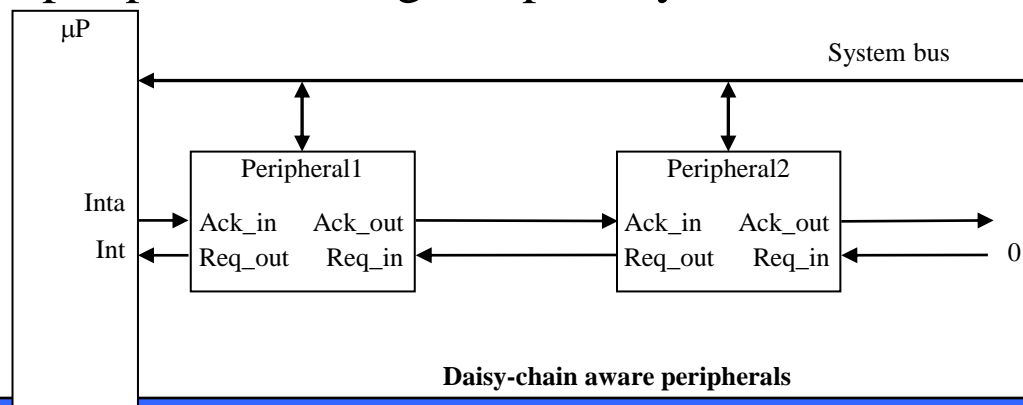
1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor 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
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

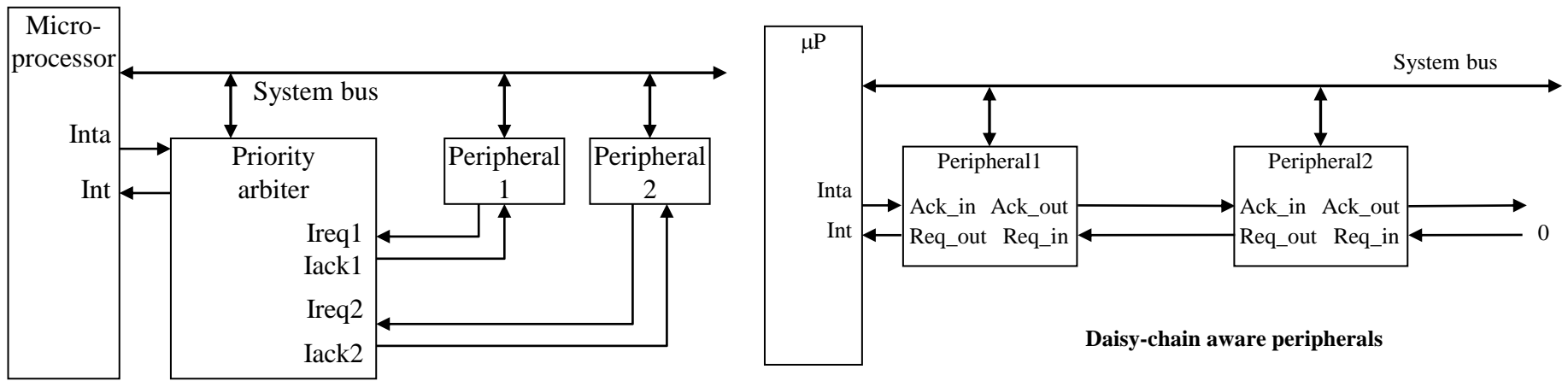
Arbitration: Daisy-chain arbitration

- 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



Arbitration: Daisy-chain arbitration

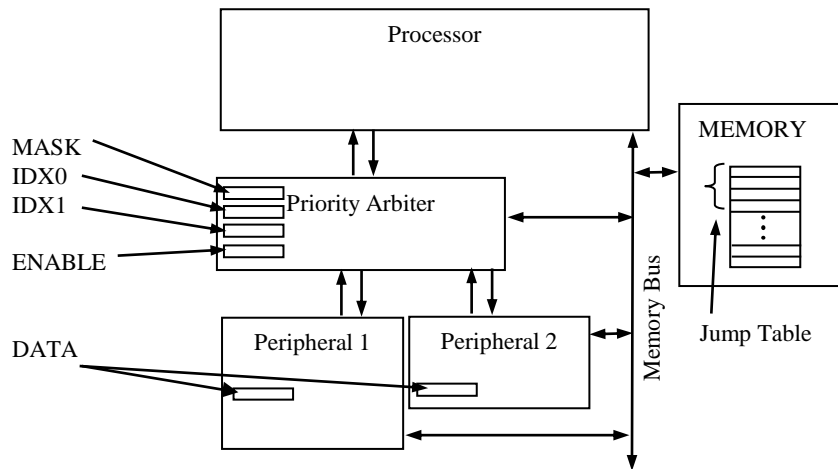
- Pros/cons
 - Easy to add/remove peripheral - no system redesign needed
 - Does not support rotating priority
 - One broken peripheral can cause loss of access to other peripherals



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
}
```

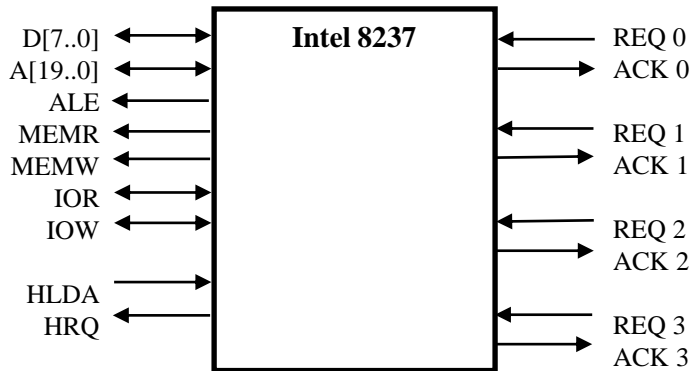
- Fixed priority: i.e., Peripheral1 has highest priority
- Keyword “_at_” followed by memory address forces compiler to place variables in specific memory locations
 - e.g., memory-mapped registers in arbiter, peripherals
- A peripheral’s index into interrupt table is sent to memory-mapped register in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

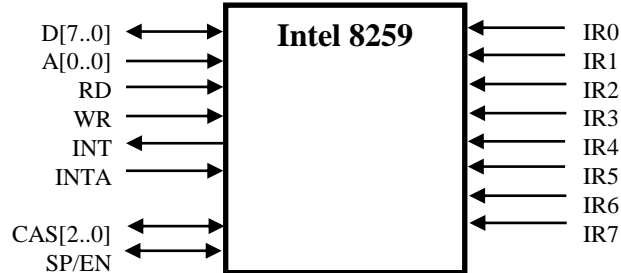
void InitializePeripherals(void) {
    ARBITER_MASK_REG = 0x03; // enable both channels
    ARBITER_CH0_INDEX_REG = 13;
    ARBITER_CH1_INDEX_REG = 17;
    INTERRUPT_LOOKUP_TABLE[13] = (void*)Peripheral1_ISR;
    INTERRUPT_LOOKUP_TABLE[17] = (void*)Peripheral2_ISR;
    ARBITER_ENABLE_REG = 1;
}
```


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.	

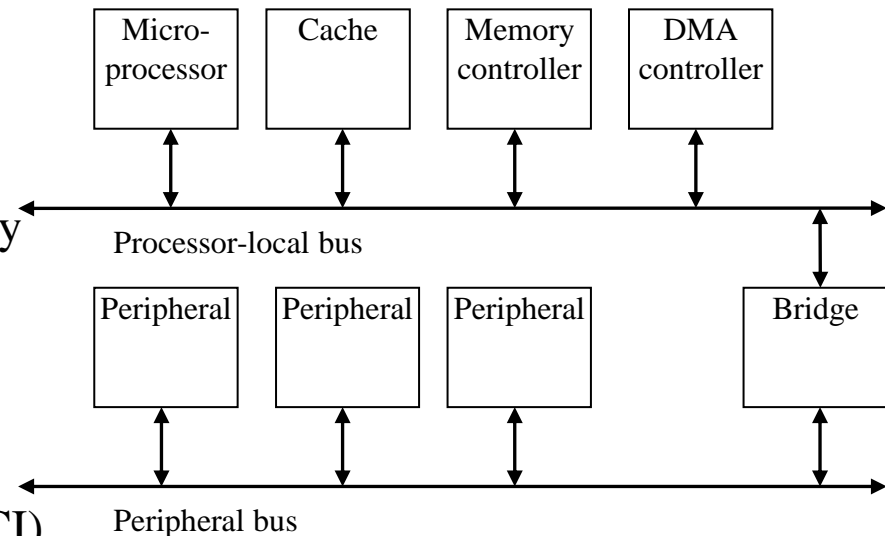
Intel 8259 programmable priority controller



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.

Multilevel bus architectures

- Don't want one bus for all communication
 - 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 standard bus (ISA, PCI) for portability
- Bridge
 - Single-purpose processor converts communication between busses



Advanced communication principles

- Layering
 - Break complexity of communication protocol into pieces easier to design and understand
 - Lower levels provide services to higher level
 - Lower level might work with bits while higher level might work with packets of data
 - Physical layer
 - Lowest level in hierarchy
 - Medium to carry data from one actor (device or node) to another
 - Parallel communication
 - Physical layer capable of transporting multiple bits of data
 - Serial communication
 - Physical layer transports one bit of data at a time
 - Wireless communication
 - No physical connection needed for transport at physical layer
-

Parallel communication

- Multiple data, control, and possibly power wires
 - One bit per wire
- High data throughput with short distances
- Typically used when connecting devices on same IC or same circuit board
 - Bus must be kept short
 - long parallel wires result in high capacitance values which requires more time to charge/discharge
 - Data misalignment between wires increases as length increases
- Higher cost, bulky

Serial communication

- Single data wire, possibly also control and power wires
- Words transmitted one bit at a time
- Higher data throughput with long distances
 - Less average capacitance, so more bits per unit of time
- Cheaper, less bulky
- More complex interfacing logic and communication protocol
 - Sender needs to decompose word into bits
 - Receiver needs to recompose bits into word
 - Control signals often sent on same wire as data increasing protocol complexity

Wireless communication

- Infrared (IR)
 - Electronic wave frequencies just below visible light spectrum
 - Diode emits infrared light to generate signal
 - Infrared transistor detects signal, conducts when exposed to infrared light
 - Cheap to build
 - Need line of sight, limited range
- Radio frequency (RF)
 - Electromagnetic wave frequencies in radio spectrum
 - Analog circuitry and antenna needed on both sides of transmission
 - Line of sight not needed, transmitter power determines range

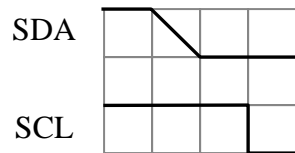
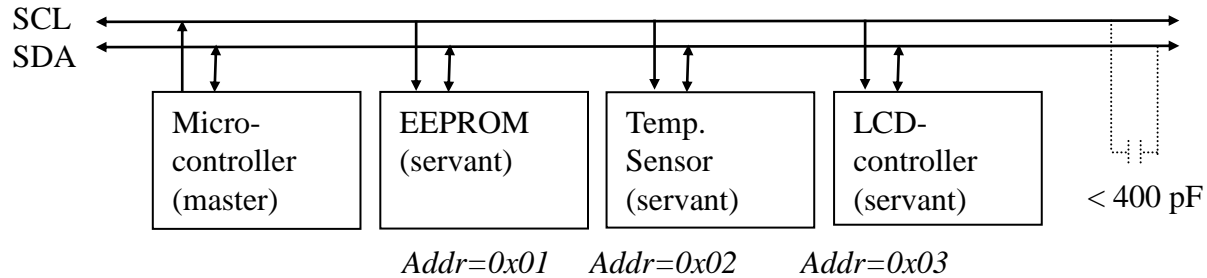
Error detection and correction

- Often part of bus protocol
- Error detection: ability of receiver to detect errors during transmission
- Error correction: ability of receiver and transmitter to cooperate to correct problem
 - Typically done by acknowledgement/retransmission protocol
- Bit error: single bit is inverted
- Burst of bit error: consecutive bits received incorrectly
- Parity: extra bit sent with word used for error detection
 - Odd parity: data word plus parity bit contains odd number of 1's
 - Even parity: data word plus parity bit contains even number of 1's
 - Always detects single bit errors, but not all burst bit errors
- Checksum: extra word sent with data packet of multiple words
 - e.g., extra word contains XOR sum of all data words in packet

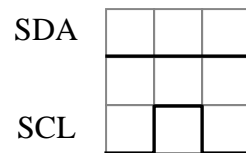
Serial protocols: I²C

- I²C (Inter-IC)
 - Two-wire serial bus protocol developed by Philips Semiconductors nearly 20 years ago
 - Enables peripheral ICs to communicate using simple communication hardware
 - Data transfer rates up to 100 kbits/s and 7-bit addressing possible in normal mode
 - 3.4 Mbits/s and 10-bit addressing in fast-mode
 - Common devices capable of interfacing to I²C bus:
 - EPROMS, Flash, and some RAM memory, real-time clocks, watchdog timers, and microcontrollers

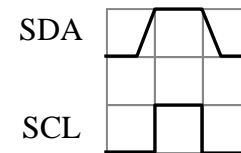
I2C bus structure



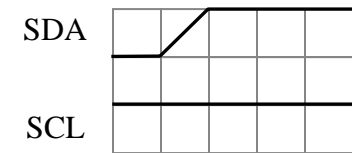
Start condition



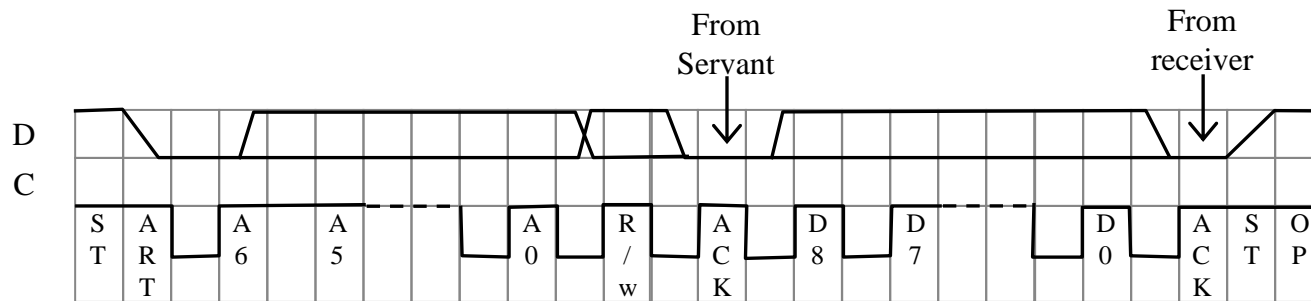
Sending 0



Sending 1



Stop condition



Typical read/write cycle

Serial protocols: CAN

- CAN (Controller area network)
 - Protocol for real-time applications
 - Developed by Robert Bosch GmbH
 - Originally for communication among components of cars
 - Applications now using CAN include:
 - elevator controllers, copiers, telescopes, production-line control systems, and medical instruments
 - Data transfer rates up to 1 Mbit/s and 11-bit addressing
 - Common devices interfacing with CAN:
 - 8051-compatible 8592 processor and standalone CAN controllers
 - Actual physical design of CAN bus not specified in protocol
 - Requires devices to transmit/detect dominant and recessive signals to/from bus
 - e.g., '1' = dominant, '0' = recessive if single data wire used
 - Bus guarantees dominant signal prevails over recessive signal if asserted simultaneously

Serial protocols: FireWire

- FireWire (a.k.a. I-Link, Lynx, IEEE 1394)
 - High-performance serial bus developed by Apple Computer Inc.
 - Designed for interfacing independent electronic components
 - e.g., Desktop, scanner
 - Data transfer rates from 12.5 to 400 Mbits/s, 64-bit addressing
 - Plug-and-play capabilities
 - Packet-based layered design structure
 - Applications using FireWire include:
 - disk drives, printers, scanners, cameras
 - Capable of supporting a LAN similar to Ethernet
 - 64-bit address:
 - 10 bits for network ids, 1023 subnetworks
 - 6 bits for node ids, each subnetwork can have 63 nodes
 - 48 bits for memory address, each node can have 281 terabytes of distinct locations

Serial protocols: USB

- USB (Universal Serial Bus)
 - Easier connection between PC and monitors, printers, digital speakers, modems, scanners, digital cameras, joysticks, multimedia game equipment
 - 2 data rates:
 - 12 Mbps for increased bandwidth devices
 - 1.5 Mbps for lower-speed devices (joysticks, game pads)
 - Tiered star topology can be used
 - One USB device (hub) connected to PC
 - hub can be embedded in devices like monitor, printer, or keyboard or can be standalone
 - Multiple USB devices can be connected to hub
 - Up to 127 devices can be connected like this
 - USB host controller
 - Manages and controls bandwidth and driver software required by each peripheral
 - Dynamically allocates power downstream according to devices connected/disconnected

Parallel protocols: PCI Bus

- PCI Bus (Peripheral Component Interconnect)
 - High performance bus originated at Intel in the early 1990's
 - Standard adopted by industry and administered by PCISIG (PCI Special Interest Group)
 - Interconnects chips, expansion boards, processor memory subsystems
 - Data transfer rates of 127.2 to 508.6 Mbits/s and 32-bit addressing
 - Later extended to 64-bit while maintaining compatibility with 32-bit schemes
 - Synchronous bus architecture
 - Multiplexed data/address lines

Parallel protocols: ARM Bus

- ARM Bus
 - Designed and used internally by ARM Corporation
 - Interfaces with ARM line of processors
 - Many IC design companies have own bus protocol
 - Data transfer rate is a function of clock speed
 - If clock speed of bus is X, transfer rate = $16 \times X$ bits/s
 - 32-bit addressing

Wireless protocols: IrDA

- IrDA
 - Protocol suite that supports short-range point-to-point infrared data transmission
 - Created and promoted by the Infrared Data Association (IrDA)
 - Data transfer rate of 9.6 kbps and 4 Mbps
 - IrDA hardware deployed in notebook computers, printers, PDAs, digital cameras, public phones, cell phones
 - Lack of suitable drivers has slowed use by applications
 - Windows 2000/98 now include support
 - Becoming available on popular embedded OS's

Wireless protocols: Bluetooth

- Bluetooth
 - New, global standard for wireless connectivity
 - Based on low-cost, short-range radio link
 - Connection established when within 10 meters of each other
 - No line-of-sight required
 - e.g., Connect to printer in another room

Wireless Protocols: IEEE 802.11

- IEEE 802.11
 - Proposed standard for wireless LANs
 - Specifies parameters for PHY and MAC layers of network
 - PHY layer
 - physical layer
 - handles transmission of data between nodes
 - provisions for data transfer rates of 1 or 2 Mbps
 - operates in 2.4 to 2.4835 GHz frequency band (RF)
 - or 300 to 428,000 GHz (IR)
 - MAC layer
 - medium access control layer
 - protocol responsible for maintaining order in shared medium
 - collision avoidance/detection

Chapter Summary

- Basic protocol concepts
 - Actors, direction, time multiplexing, control methods
- 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
- Advanced communication
 - Parallel vs. serial, wires vs. wireless, error detection/correction, layering
 - Serial protocols: I²C, CAN, FireWire, and USB; Parallel: PCI and ARM.
 - Serial wireless protocols: IrDA, Bluetooth, and IEEE 802.11.

Embedded Systems Design: A Unified Hardware/Software Introduction

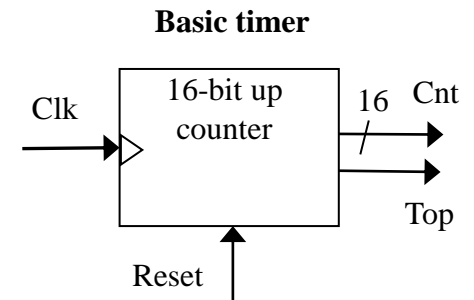
Chapter 4 Standard Single Purpose Processors: Peripherals

Introduction

- Single-purpose processors
 - Performs specific computation task
 - Custom single-purpose processors
 - Designed by us for a unique task
 - *Standard* single-purpose processors
 - “Off-the-shelf” -- pre-designed for a common task
 - a.k.a., peripherals
 - serial transmission
 - analog/digital conversions

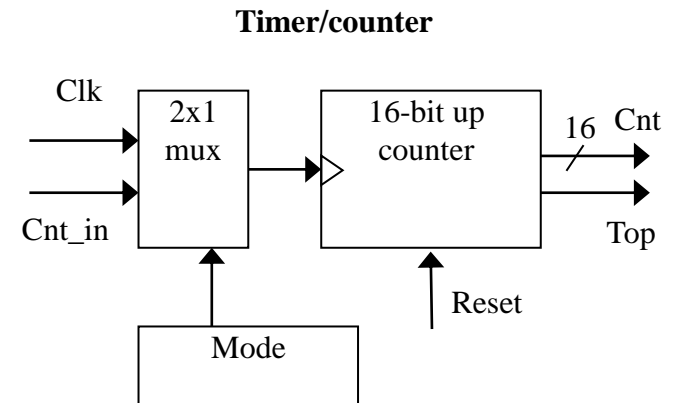
Timers, counters, watchdog timers

- Timer: measures time intervals
 - To generate timed output events
 - e.g., hold traffic light green for 10 s
 - To measure input events
 - e.g., measure a car's speed
- Based on counting clock pulses
 - E.g., let Clk period be 10 ns
 - And we count 20,000 Clk pulses
 - Then 200 microseconds have passed
 - 16-bit counter would count up to $65,535 \times 10 \text{ ns} = 655.35 \text{ microsec.}$, resolution = 10 ns
 - Top: indicates top count reached, wrap-around



Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
 - e.g., count cars passing over a sensor
 - Can often configure device as either a timer or counter



Other timer structures

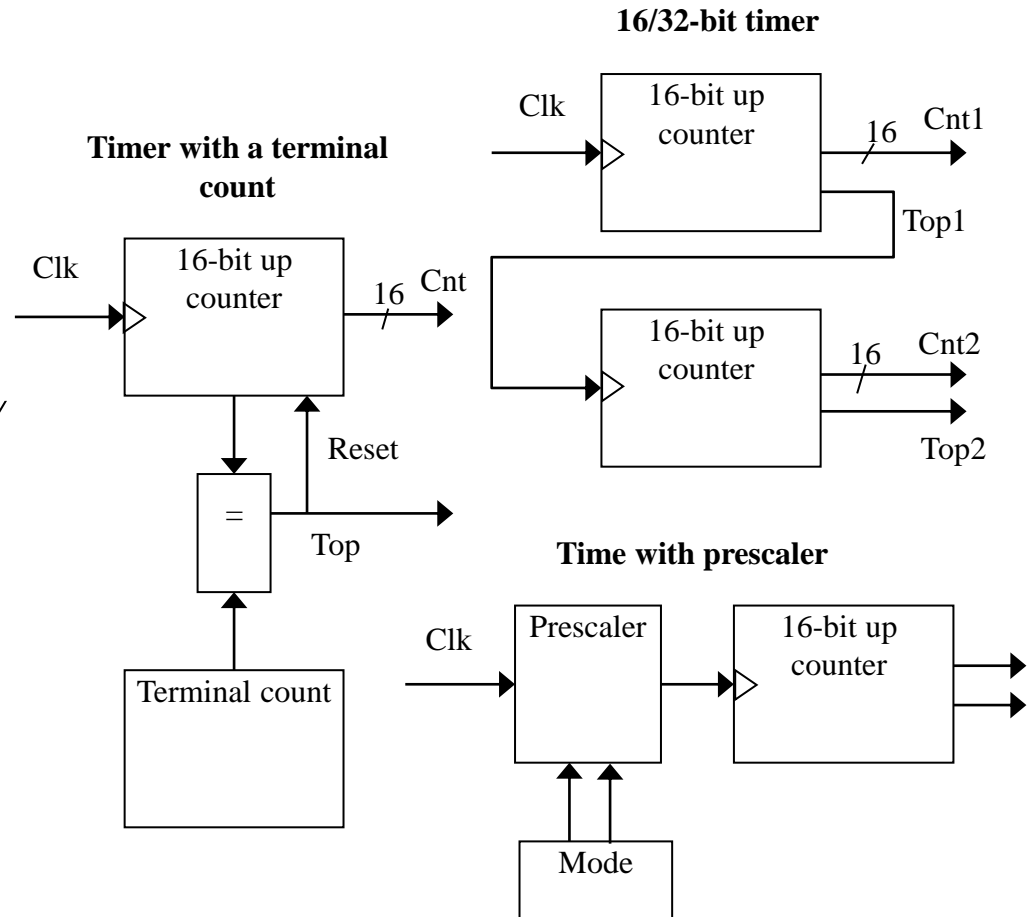
- Interval timer

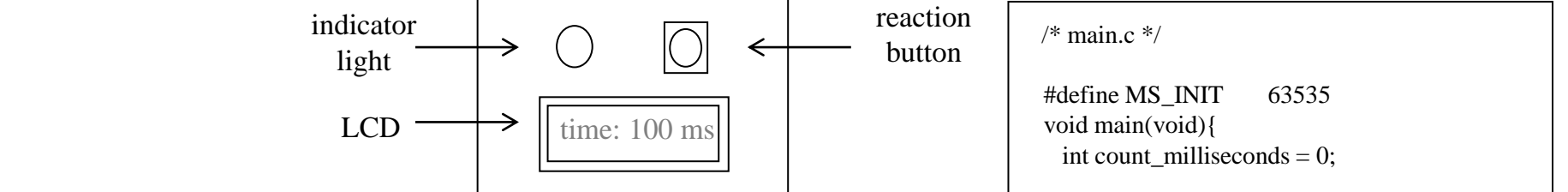
- Indicates when desired time interval has passed
- We set terminal count to desired interval
 - $\text{Number of clock cycles} = \text{Desired time interval} / \text{Clock period}$

- Cascaded counters

- Prescaler

- Divides clock
- Increases range, decreases resolution





- Measure time between turning light on and user pushing button
 - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
 - Resolution = $6 \times 83.33 = 0.5$ microsec.
 - Range = 65535×0.5 microseconds = 32.77 milliseconds
 - Want program to count millisec., so initialize counter to $65535 - 1000/0.5 = 63535$
- ```

configure timer mode
set Cnt to MS_INIT

wait a random amount of time
turn on indicator light
start timer

while (user has not pushed reaction button){
 if(Top) {
 stop timer
 set Cnt to MS_INIT
 start timer
 reset Top
 count_milliseconds++;
 }
}
turn light off
printf("time: %i ms", count_milliseconds);
}

```

```

/* main.c */

#define MS_INIT 63535

void main(void){
 int count_milliseconds = 0;

 configure timer mode
 set Cnt to MS_INIT

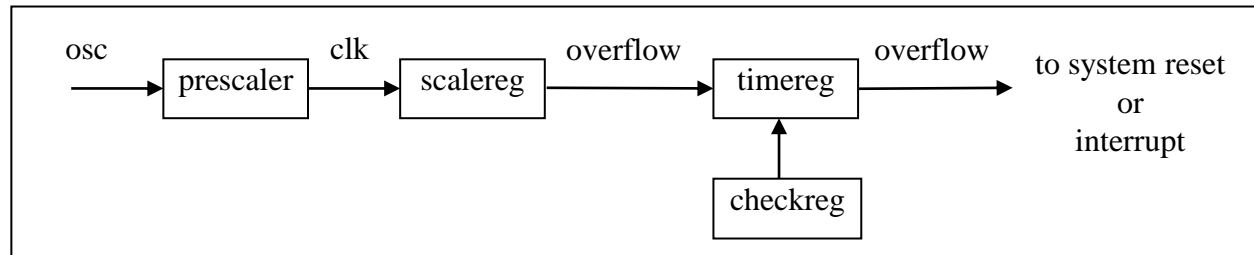
 wait a random amount of time
 turn on indicator light
 start timer

while (user has not pushed reaction button){
 if(Top) {
 stop timer
 set Cnt to MS_INIT
 start timer
 reset Top
 count_milliseconds++;
 }
}
 turn light off
 printf("time: %i ms", count_milliseconds);
}

```

# Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - $timereg\ value = 2 * (2^{16} - 1) - X = 131070 - X$
  - For 2 min.,  $X = 120,000$  microsec.



```
/* main.c */

main(){
 wait until card inserted
 call watchdog_reset_routine

 while(transaction in progress){
 if(button pressed){
 perform corresponding action
 call watchdog_reset_routine
 }
 }

 /* if watchdog_reset_routine not called every
 < 2 minutes, interrupt_service_routine is
 called */
}
```

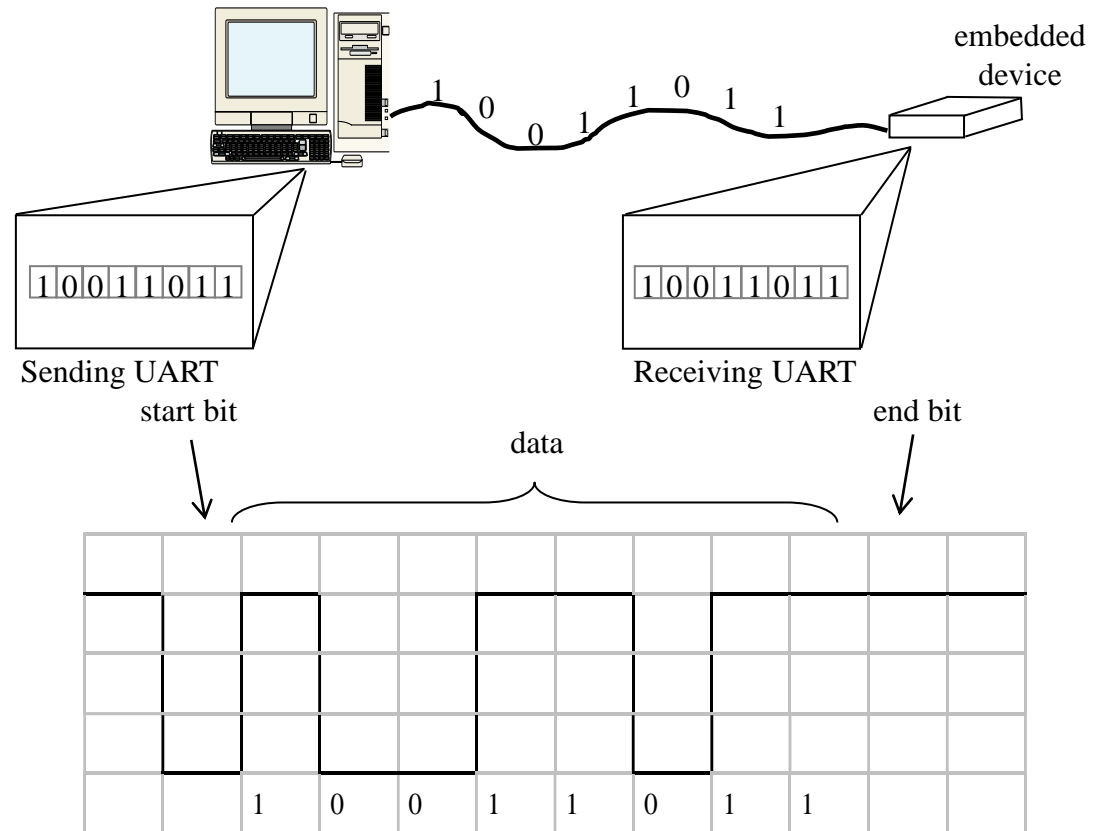
```
watchdog_reset_routine(){
 /* checkreg is set so we can load value into
 timereg. Zero is loaded into scalereg and
 11070 is loaded into timereg */

 checkreg = 1
 scalereg = 0
 timereg = 11070
}

void interrupt_service_routine(){
 eject card
 reset screen
}
```

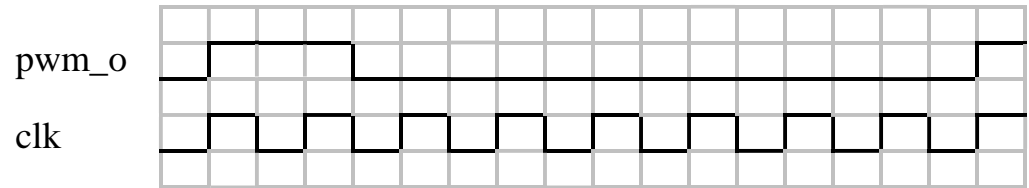
# Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher

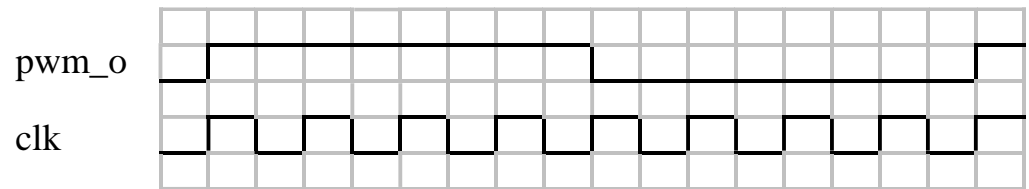


# Pulse width modulator

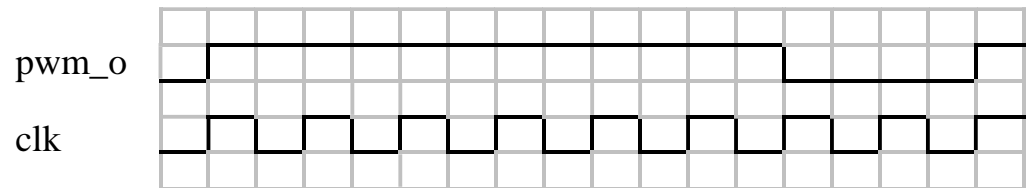
- Generates pulses with specific high/low times
- Duty cycle: % time high
  - Square wave: 50% duty cycle
- Common use: control average voltage to electric device
  - Simpler than DC-DC converter or digital-analog converter
  - DC motor speed, dimmer lights
- Another use: encode commands, receiver uses timer to decode



25% duty cycle – average `pwm_o` is 1.25V

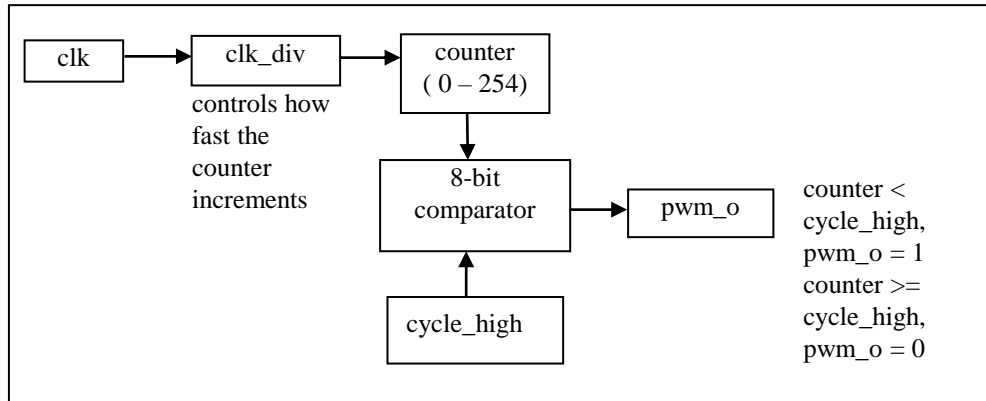


50% duty cycle – average `pwm_o` is 2.5V.



75% duty cycle – average `pwm_o` is 3.75V.

# Controlling a DC motor with a PWM



Internal Structure of PWM

| Input Voltage | % of Maximum Voltage Applied | RPM of DC Motor |
|---------------|------------------------------|-----------------|
| 0             | 0                            | 0               |
| 2.5           | 50                           | 1840            |
| 3.75          | 75                           | 6900            |
| 5.0           | 100                          | 9200            |

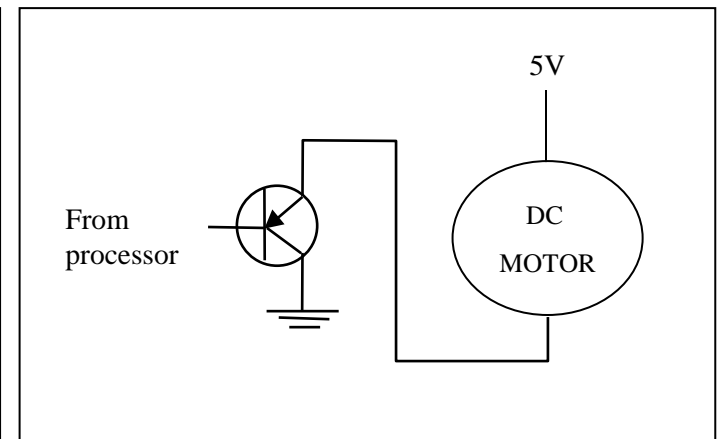
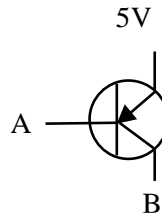
Relationship between applied voltage and speed of the DC Motor

```
void main(void) {

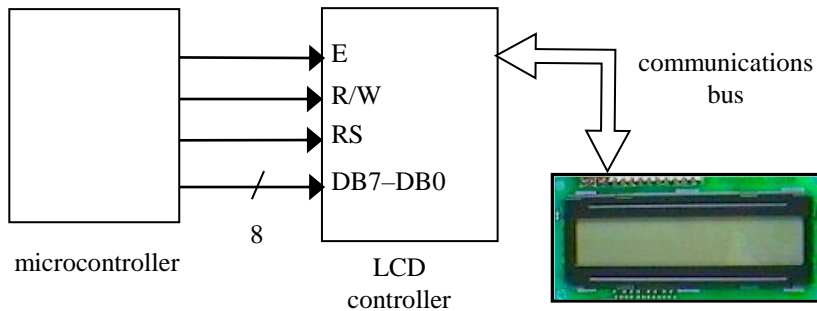
 /* controls period */
 PWMP = 0xff;
 /* controls duty cycle */
 PWM1 = 0x7f;

 while(1) {};
}
```

The PWM alone cannot drive the DC motor, a possible way to implement a driver is shown below using an MJE3055T NPN transistor.



# LCD controller

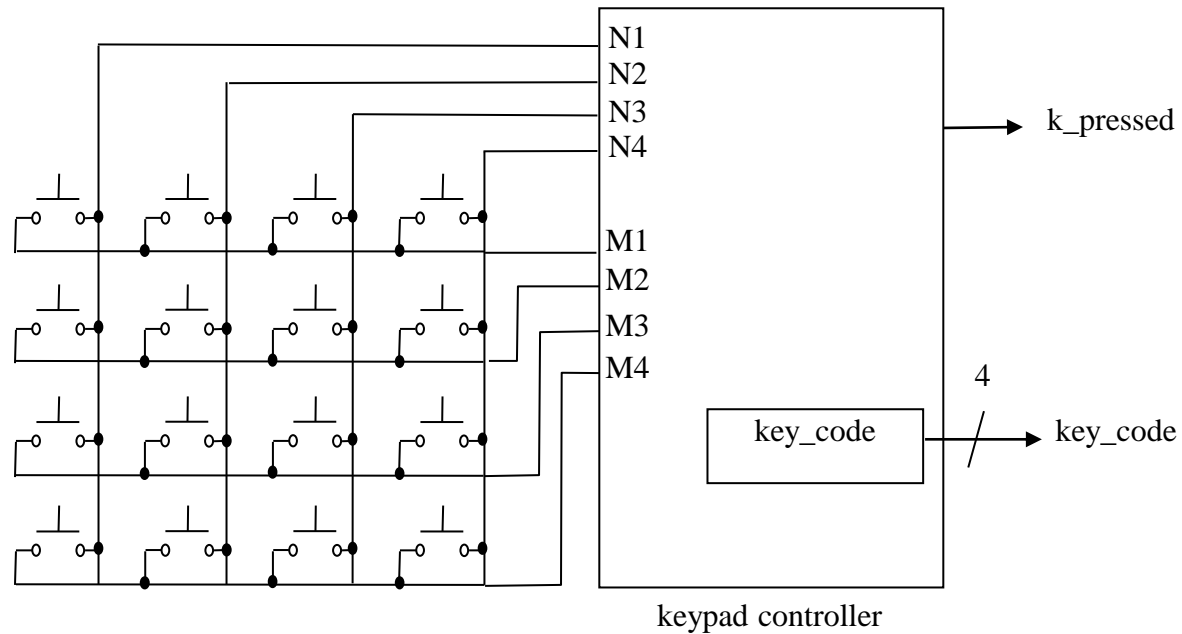


```
void WriteChar(char c){
 RS = 1; /* indicate data being sent */
 DATA_BUS = c; /* send data to LCD */
 EnableLCD(45); /* toggle the LCD with appropriate delay */
}
```

| CODES                      |                 |
|----------------------------|-----------------|
| I/D = 1 cursor moves left  | DL = 1 8-bit    |
| I/D = 0 cursor moves right | DL = 0 4-bit    |
| S = 1 with display shift   | N = 1 2 rows    |
| S/C = 1 display shift      | N = 0 1 row     |
| S/C = 0 cursor movement    | F = 1 5x10 dots |
| R/L = 1 shift to right     | F = 0 5x7 dots  |
| R/L = 0 shift to left      |                 |

| RS | R/W | DB <sub>7</sub> | DB <sub>6</sub> | DB <sub>5</sub> | DB <sub>4</sub> | DB <sub>3</sub> | DB <sub>2</sub> | DB <sub>1</sub> | DB <sub>0</sub> | Description                                                             |
|----|-----|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-------------------------------------------------------------------------|
| 0  | 0   | 0               | 0               | 0               | 0               | 0               | 0               | 0               | 1               | Clears all display, return cursor home                                  |
| 0  | 0   | 0               | 0               | 0               | 0               | 0               | 0               | 1               | *               | Returns cursor home                                                     |
| 0  | 0   | 0               | 0               | 0               | 0               | 0               | 1               | I/D             | S               | Sets cursor move direction and/or specifies not to shift display        |
| 0  | 0   | 0               | 0               | 0               | 0               | 1               | D               | C               | B               | ON/OFF of all display(D), cursor ON/OFF (C), and blink position (B)     |
| 0  | 0   | 0               | 0               | 0               | 1               | S/C             | R/L             | *               | *               | Move cursor and shifts display                                          |
| 0  | 0   | 0               | 0               | 1               | DL              | N               | F               | *               | *               | Sets interface data length, number of display lines, and character font |
| 1  | 0   | WRITE DATA      |                 |                 |                 |                 |                 |                 |                 | Writes Data                                                             |

# Keypad controller

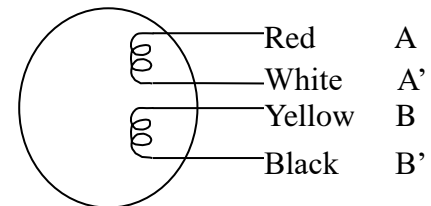
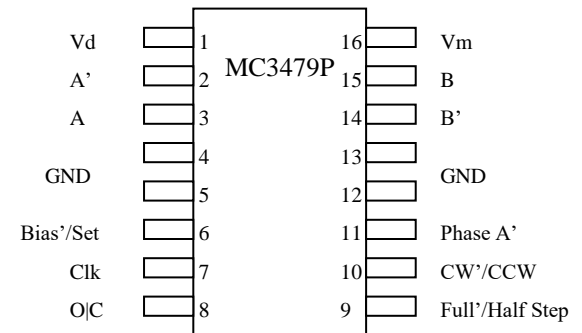


N=4, M=4

# Stepper motor controller

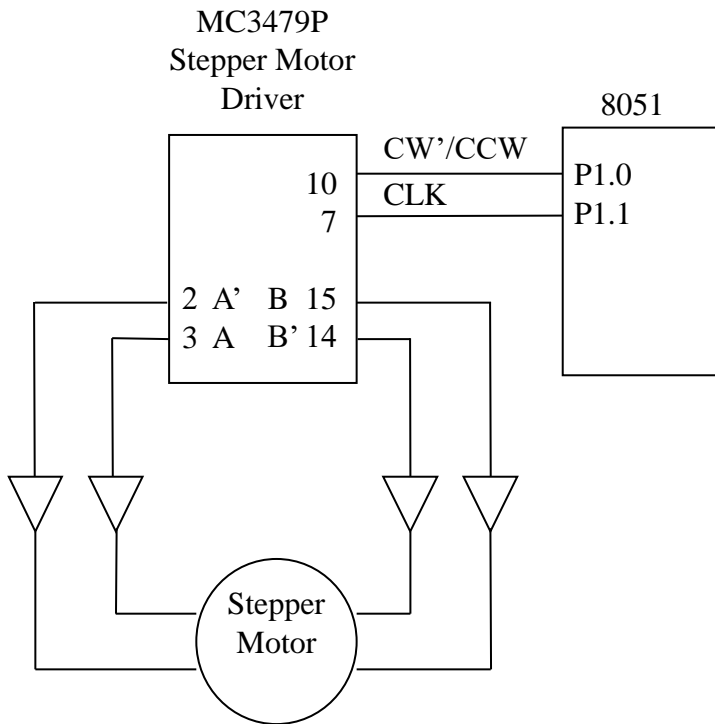
- Stepper motor: rotates fixed number of degrees when given a “step” signal
  - In contrast, DC motor just rotates when power applied, coasts to stop
- Rotation achieved by applying specific voltage sequence to coils
- Controller greatly simplifies this

| Sequence | A | B | A' | B' |
|----------|---|---|----|----|
| 1        | + | + | -  | -  |
| 2        | - | + | +  | -  |
| 3        | - | - | +  | +  |
| 4        | + | - | -  | +  |
| 5        | + | + | -  | -  |





# Stepper motor with controller (driver)



/\* main.c \*/

sbit clk=P1^1;

sbit cw=P1^0;

```
void delay(void){
 int i, j;
 for (i=0; i<1000; i++)
 for (j=0; j<50; j++)
 i = i + 0;
}
```

void main(void){

/\*turn the motor forward \*/

cw=0; /\* set direction \*/

clk=0; /\* pulse clock \*/

delay();

clk=1;

/\*turn the motor backwards \*/

cw=1; /\* set direction \*/

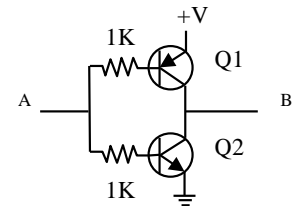
clk=0; /\* pulse clock \*/

delay();

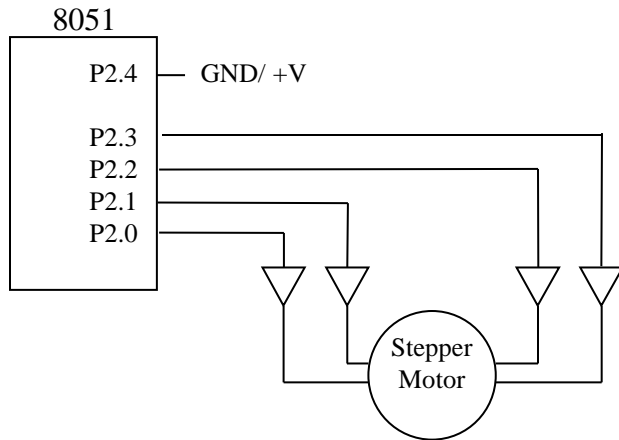
clk=1;

}

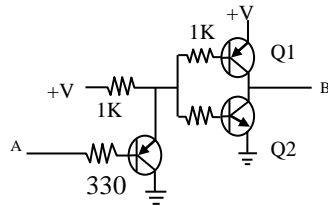
The output pins on the stepper motor driver do not provide enough current to drive the stepper motor. To amplify the current, a buffer is needed. One possible implementation of the buffers is pictured to the left. Q1 is an MJE3055T NPN transistor and Q2 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.



# Stepper motor without controller (driver)



A possible way to implement the buffers is located below. The 8051 alone cannot drive the stepper motor, so several transistors were added to increase the current going to the stepper motor. Q1 are MJE3055T NPN transistors and Q3 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.



```
/*main.c*/
sbit notA=P2^0;
sbit isA=P2^1;
sbit notB=P2^2;
sbit isB=P2^3;
sbit dir=P2^4;

void delay(){
 int a, b;
 for(a=0; a<5000; a++){
 for(b=0; b<10000; b++){
 a=a+0;
 }
 }
}
```

```
void move(int dir, int steps) {
 int y, z;
 /* clockwise movement */
 if(dir == 1){
 for(y=0; y<=steps; y++){
 for(z=0; z<=19; z++){
 isA=lookup[z];
 isB=lookup[z+1];
 notA=lookup[z+2];
 notB=lookup[z+3];
 delay();
 }
 }
 }
}
```

```
/* counter clockwise movement */
if(dir==0){
 for(y=0; y<=step; y++){
 for(z=19; z>=0; z - 4){
 isA=lookup[z];
 isB=lookup[z-1];
 notA=lookup[z - 2];
 notB=lookup[z-3];
 delay();
 }
 }
}

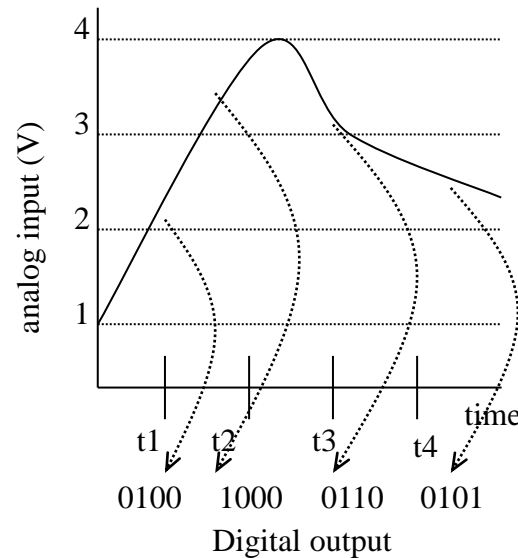
void main(){
 int z;
 int lookup[20] = {
 1, 1, 0, 0,
 0, 1, 1, 0,
 0, 0, 1, 1,
 1, 0, 0, 1,
 1, 1, 0, 0 };
 while(1){
 /*move forward, 15 degrees (2 steps) */
 move(1, 2);
 /* move backwards, 7.5 degrees (1step)*/
 move(0, 1);
 }
}
```

# Analog-to-digital converters

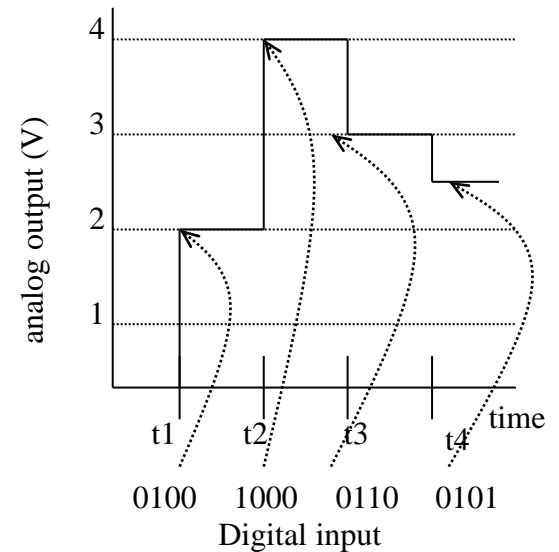
$V_{\max} = 7.5V$

|      |      |
|------|------|
| 7.5V | 1111 |
| 7.0V | 1110 |
| 6.5V | 1101 |
| 6.0V | 1100 |
| 5.5V | 1011 |
| 5.0V | 1010 |
| 4.5V | 1001 |
| 4.0V | 1000 |
| 3.5V | 0111 |
| 3.0V | 0110 |
| 2.5V | 0101 |
| 2.0V | 0100 |
| 1.5V | 0011 |
| 1.0V | 0010 |
| 0.5V | 0001 |
| 0V   | 0000 |

**proportionality**



**analog to digital**



**digital to analog**

# Digital-to-analog conversion using successive approximation

Given an analog input signal whose voltage should range from 0 to 15 volts, and an 8-bit digital encoding, calculate the correct encoding for 5 volts. Then trace the successive-approximation approach to find the correct encoding.

$$5/15 = d/(2^8-1)$$

$$d = 85$$

Encoding: 01010101

## *Successive-approximation method*

$$\frac{1}{2}(V_{\max} - V_{\min}) = 7.5 \text{ volts}$$

$$V_{\max} = 7.5 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(5.63 + 4.69) = 5.16 \text{ volts}$$

$$V_{\max} = 5.16 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(7.5 + 0) = 3.75 \text{ volts}$$

$$V_{\min} = 3.75 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(5.16 + 4.69) = 4.93 \text{ volts}$$

$$V_{\min} = 4.93 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(7.5 + 3.75) = 5.63 \text{ volts}$$

$$V_{\max} = 5.63 \text{ volts}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(5.16 + 4.93) = 5.05 \text{ volts}$$

$$V_{\max} = 5.05 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(5.63 + 3.75) = 4.69 \text{ volts}$$

$$V_{\min} = 4.69 \text{ volts.}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$$\frac{1}{2}(5.05 + 4.93) = 4.99 \text{ volts}$$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|