# Introduction to Embedded System I/O Architectures

# I/O terminology

- Synchronous / Iso-synchronous / Asynchronous
- Serial vs. Parallel
- Input/Output/Input-Output devices
- Full-duplex/ Half-duplex

# Synchronous / Iso-synchronous / Asynchronous

- Synchronous: Separation by a constant interval or phase difference (usually clock)
- Iso-synchronous: a special case of Synchronous when maximum time interval can be varied
- Asynchronous: data is transmitted and received at variable time intervals.

# Example

Synchronous: Frames data over LAN, inter-processor communication

Iso-Synchronous: special mode of USB for voice/video transfer

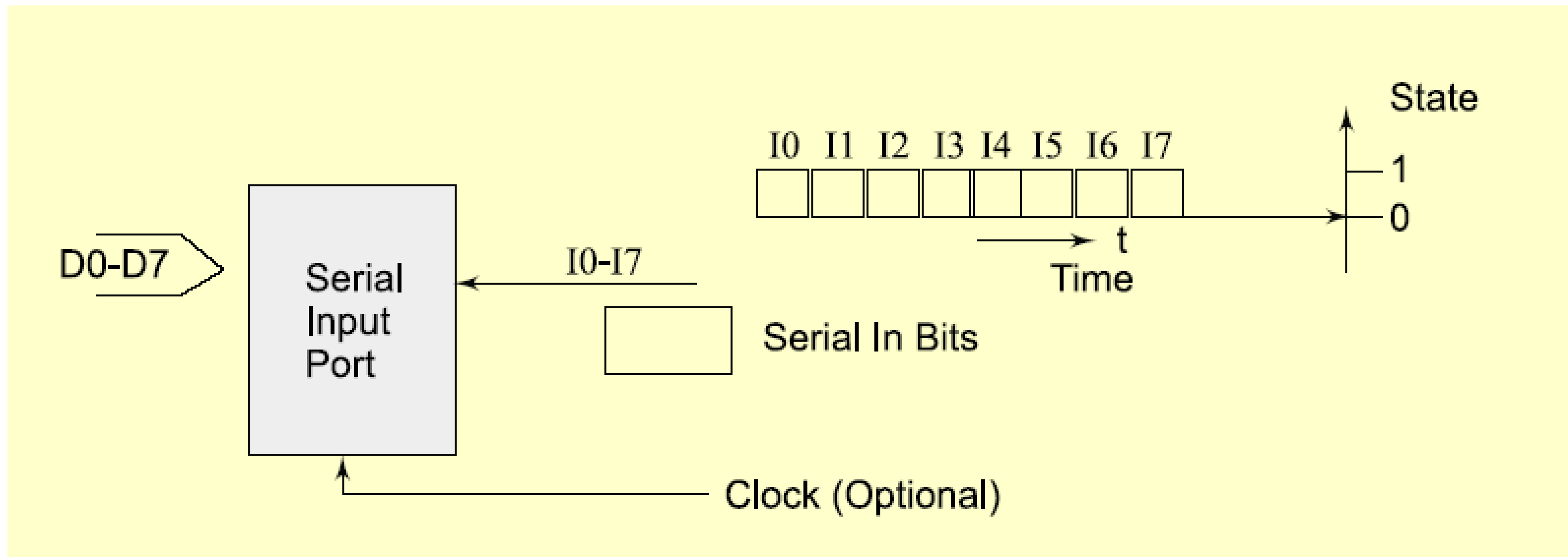
Asynchronous: UART Serial, modem

# Serial/Parallel

- Serial communication: Send one bit at a time e.g. UART serial port

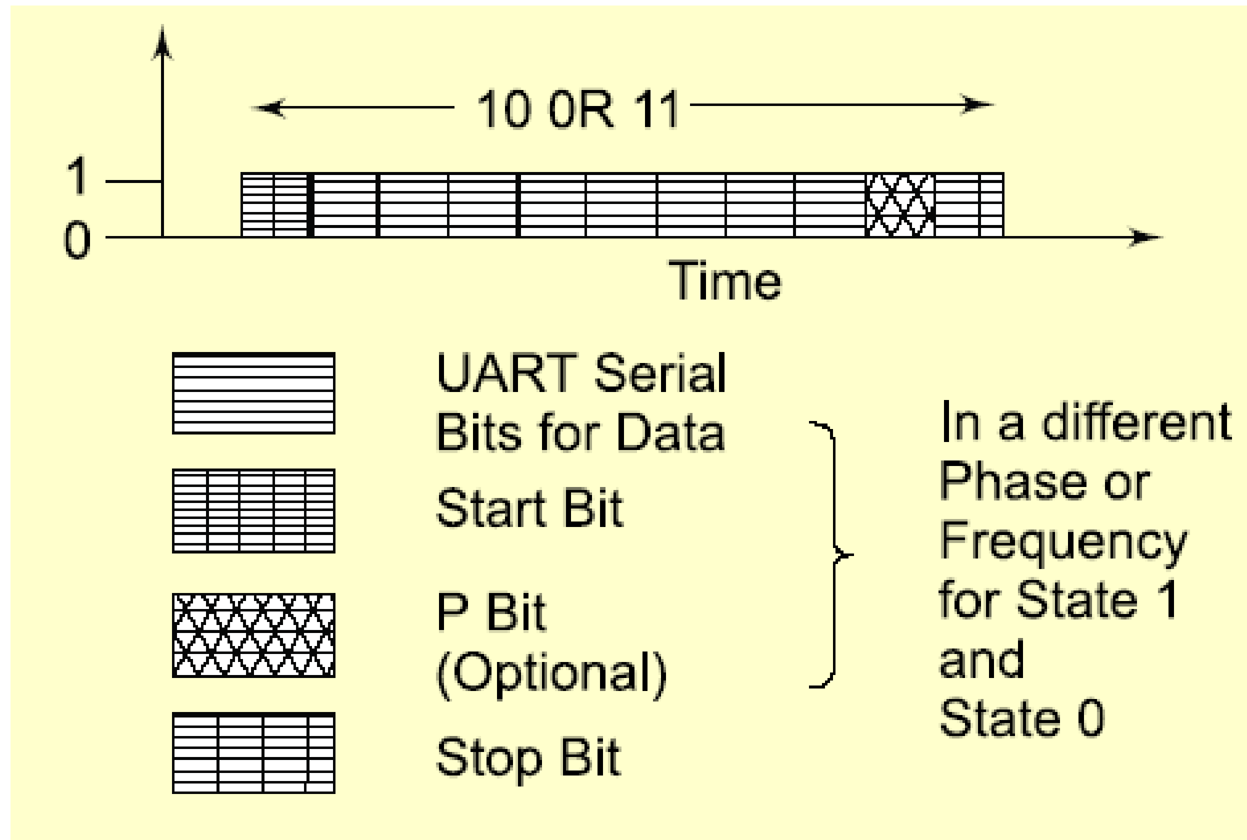
- Parallel communication: Send multiple bits of data e.g. LCD connection, PCI, ISA bus

# Synchronous Serial transfer



# Typical Data Format Asynchronous Communication

# Asynchronous Serial Transfer



# Full-Duplex and Half-Duplex

- Full-duplex: the communication can be both ways simultaneously on a bi-directional line e.g., telephone

- Half-duplex the communication can be only one-way on a bi-directional line e.g., a walkie-talkie

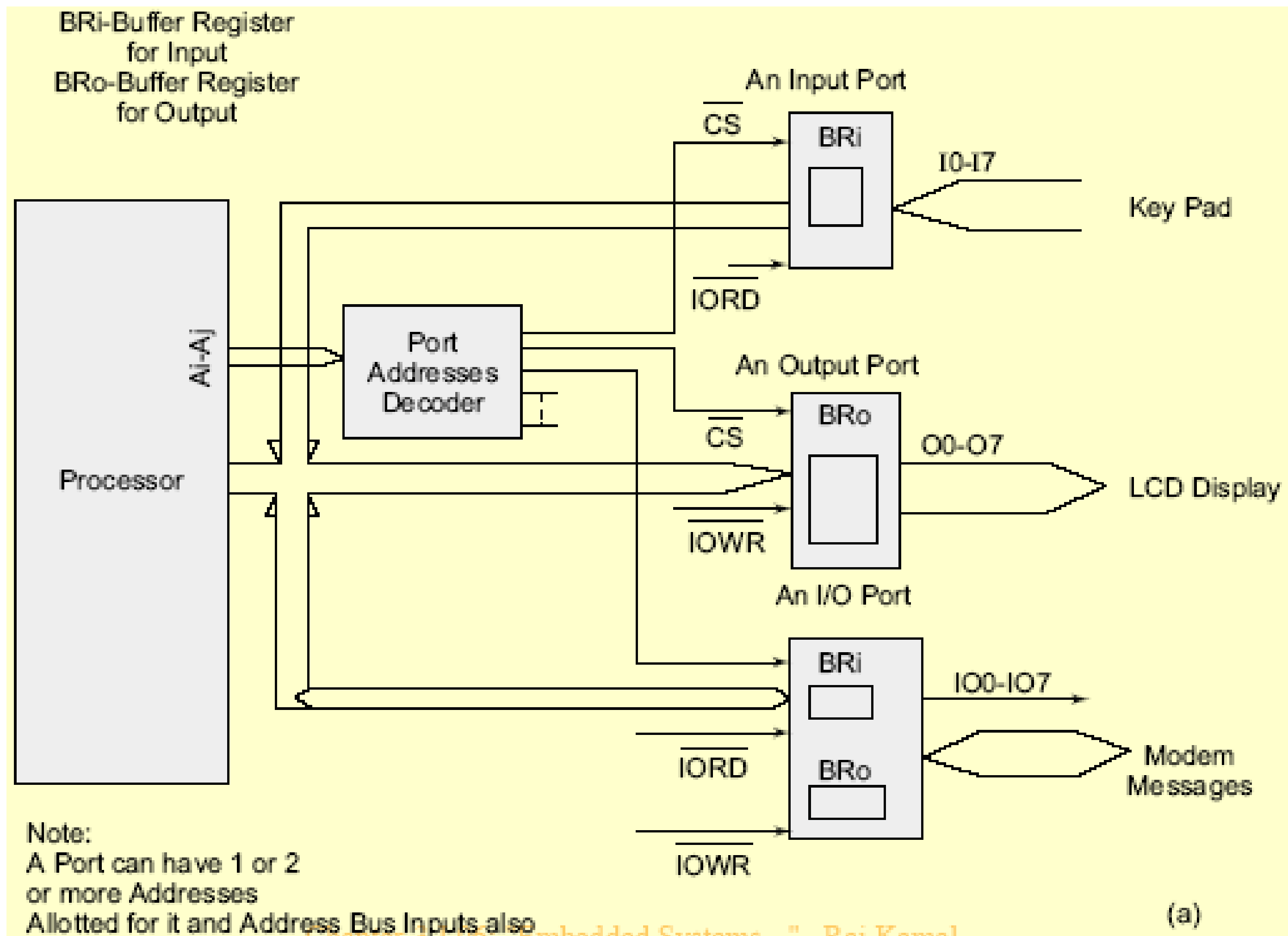
# Parallel Data Communication

- Typically used when the distance between processor and other components is small, less than a few meters
- Require high volume of data transfer
- Doesn't mean faster transfer comparing with serial data communication

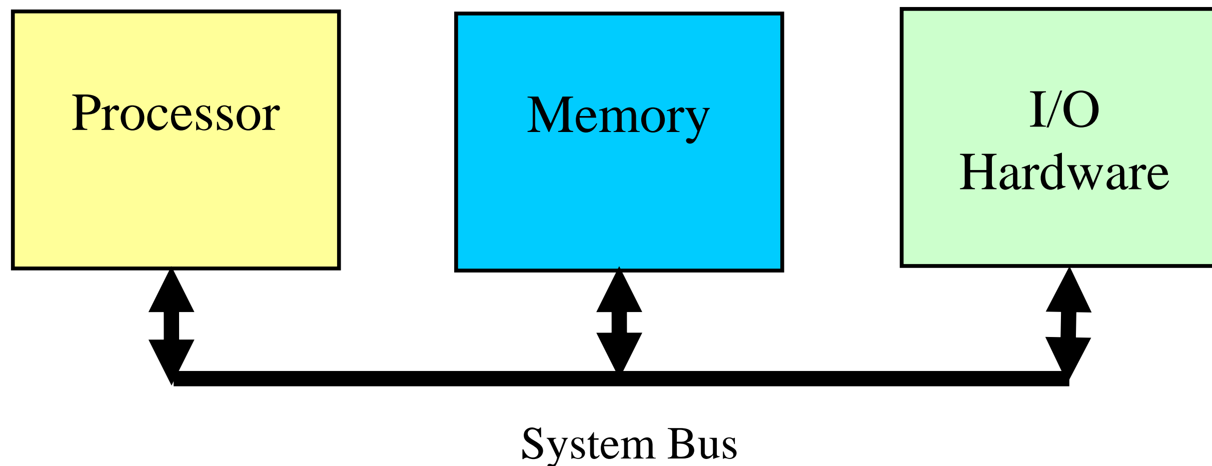
# Parallel communication

- Advantage
  High data transfer rate
- Disadvantage
  - More number of wires
  - Capacitive effects among wires make it applicable only with short wires
  - High Capacitor results in longer delay
  - Noise and cross talk effects

# Parallel port communication

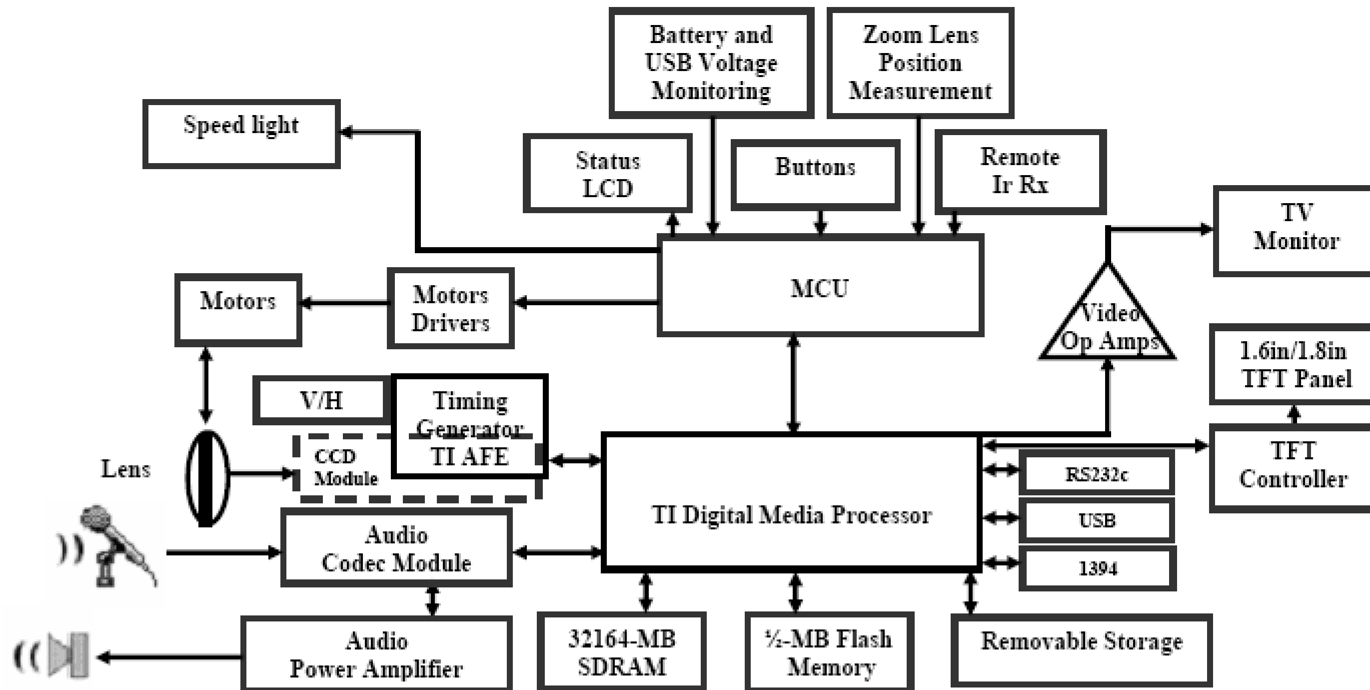


# A single bus architecture was used on early computers.



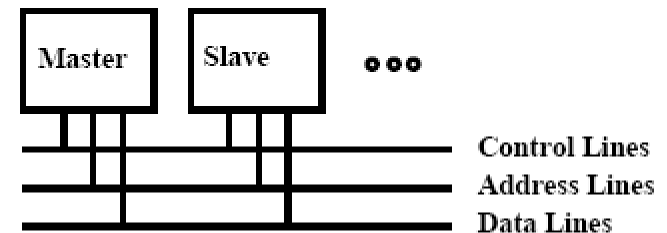

**But Modern systems are more complex and actually contain a hierarchy of different busses!**

# An Embedded System

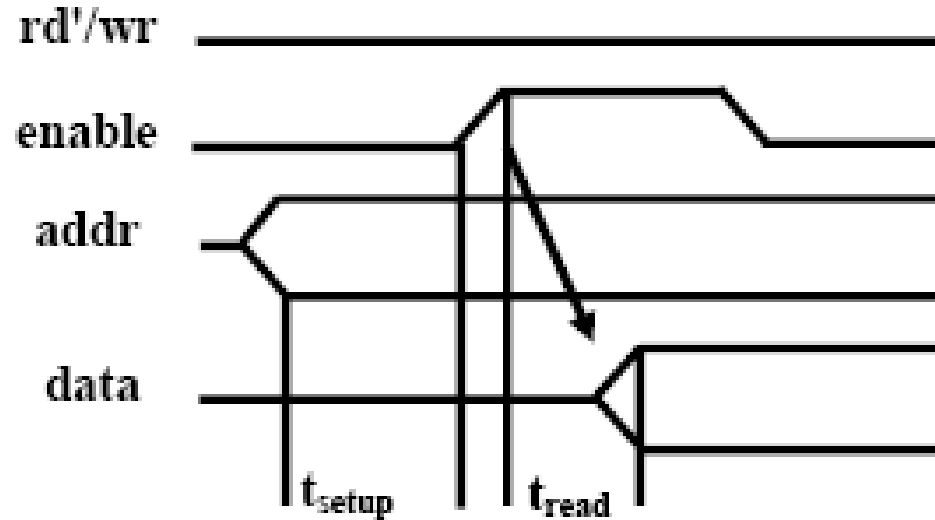


# Basic interfacing mechanism

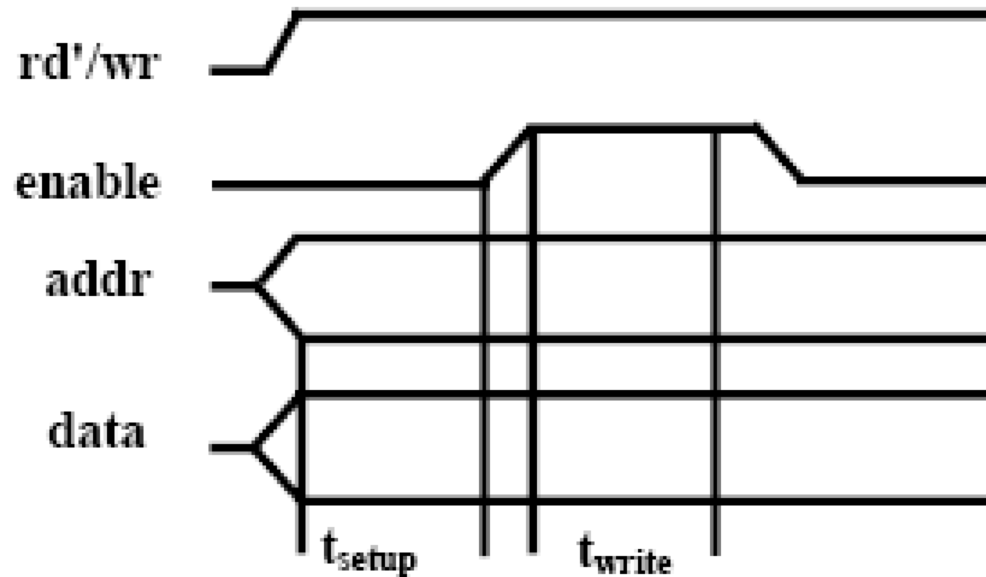
- Protocols
- Arbitrations
- Addressing



# Example of read protocol



# Example of write protocol



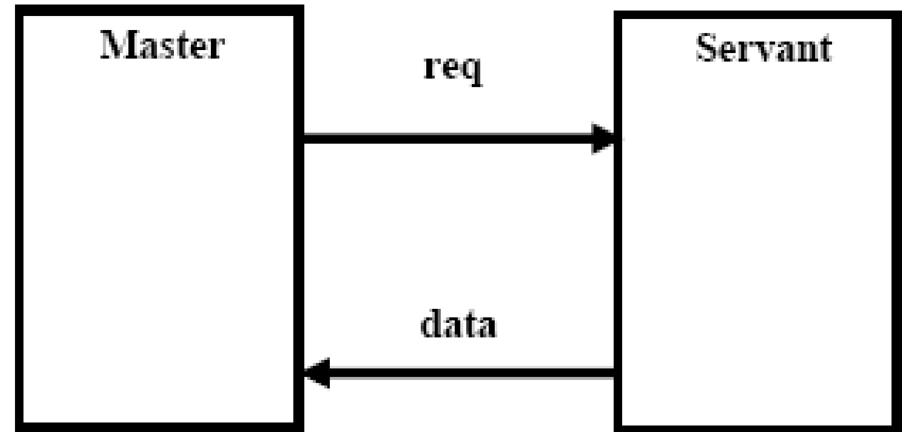
# Basic communication protocol

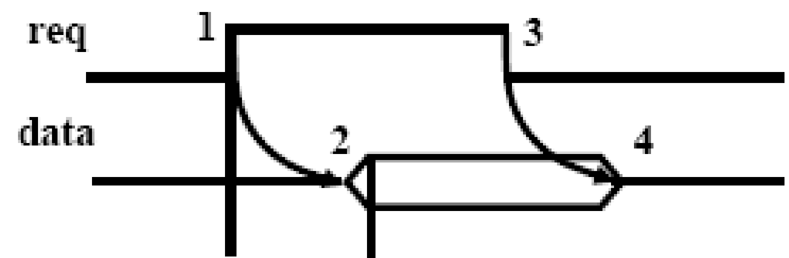
- Strobe

- Handshake

# Strobe protocol

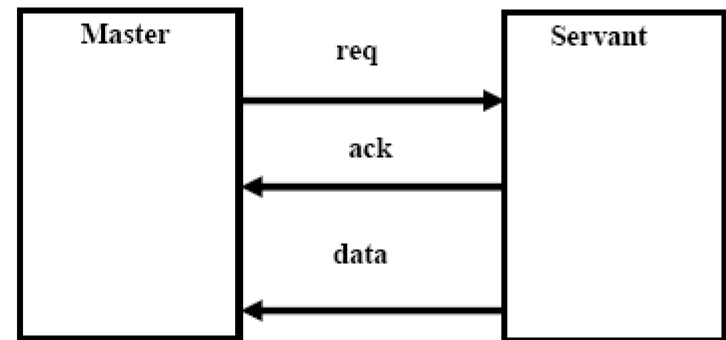
- Master assert req signal
- Servant put data on bus with time $T_{access}$
- Master receive data and deassert req
- Servant ready for next request

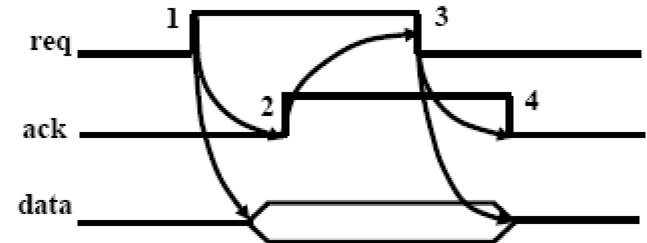


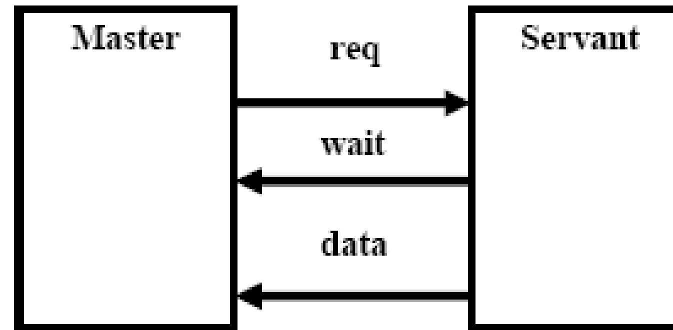
# Hand-shake protocol

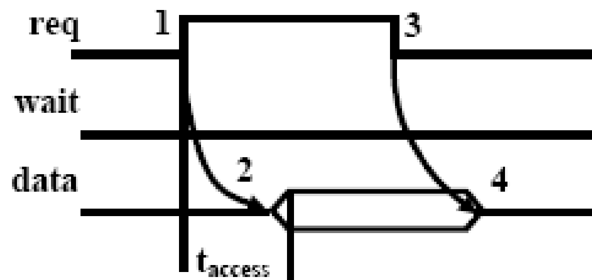
- Master assert req signal
- Servant put data on bus and assert ack
- Master receive data and deassert req
- Servant deassert ack after data completes
- Servant ready for next request

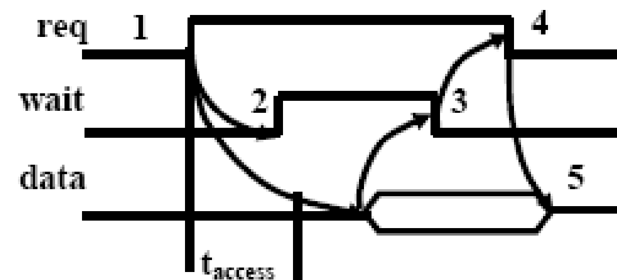


# Strobe and hand-shake combined




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**

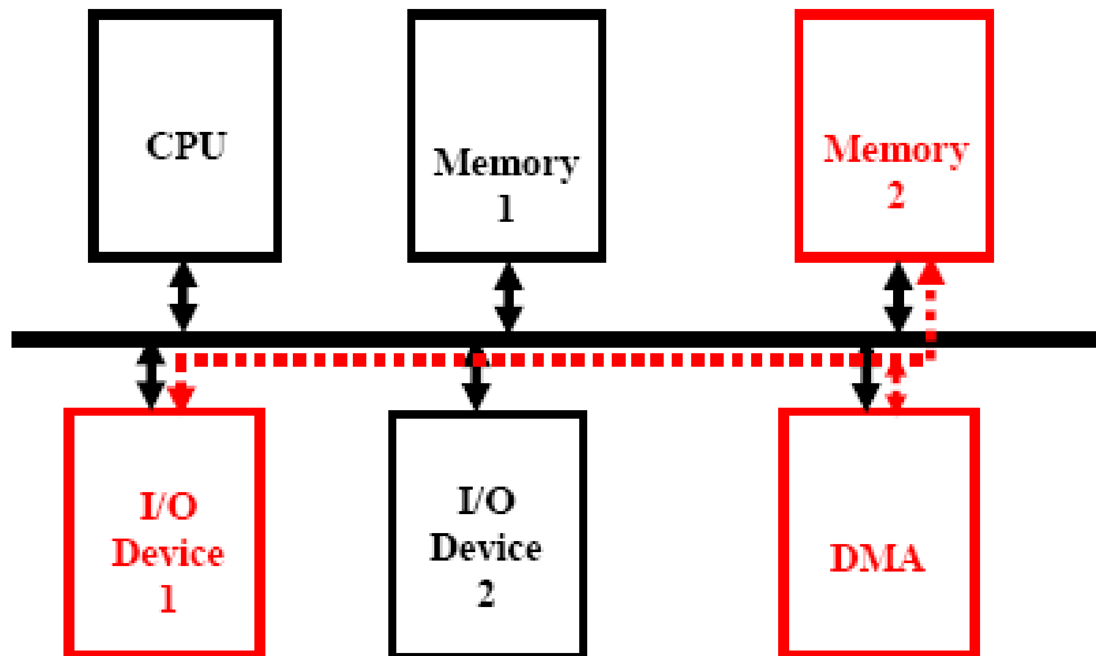
1. Master asserts *req* to receive data
2. Servant can't put data within $t_{access}$, **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

**Slow-response case**

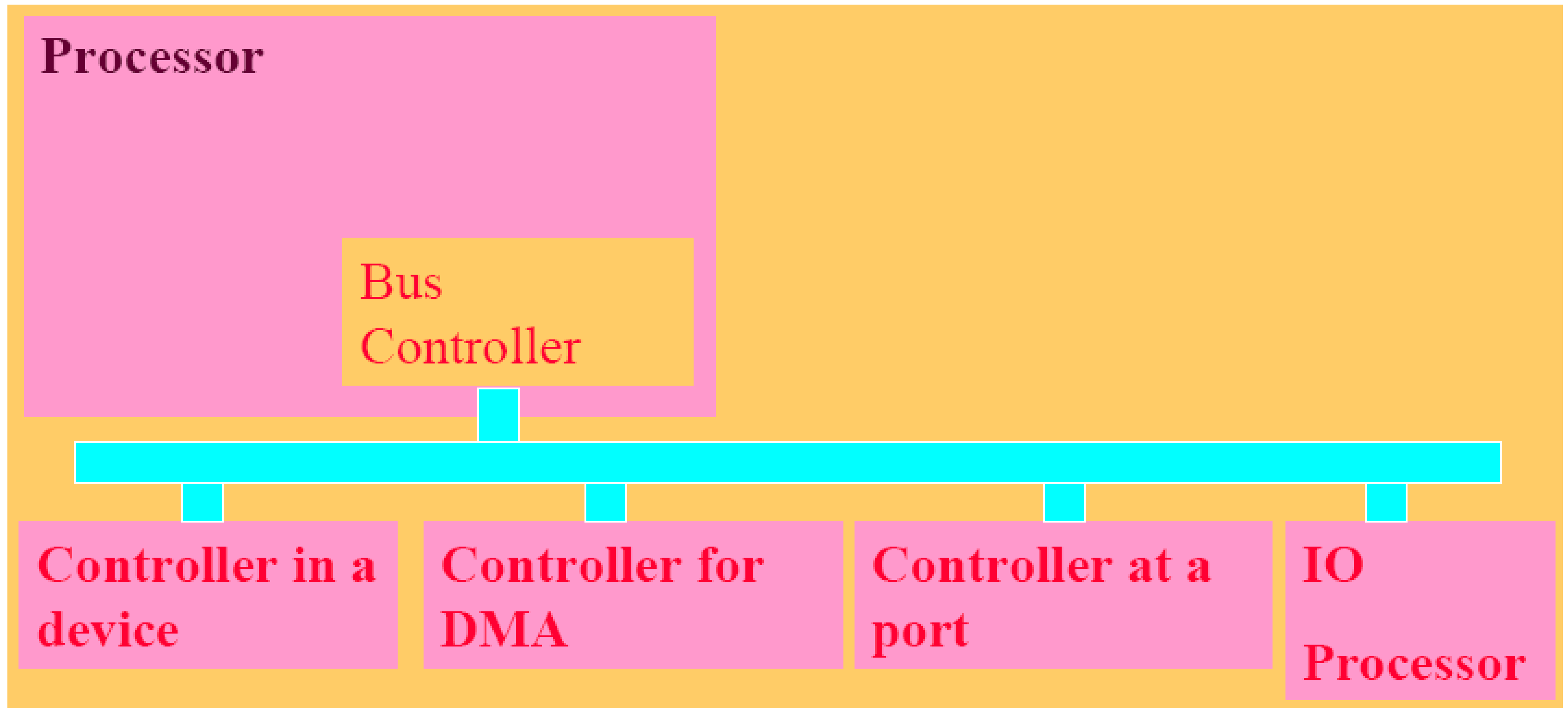
# Arbitration

- When the same set of address/control/data lines are shared by the different units, access to a bus is arbitrated by a bus master
- A bus request is sent to all nodes on the bus, the node that currently has accessed to the bus responds with either a bus grant or a bus busy.

# Arbitration



# System bus shared between controllers and I/O



# Bus signals

- BR signal is a bus request signal
- BG signal is a bus grant signal
- Busy signal is a signal showing that Bus is being used

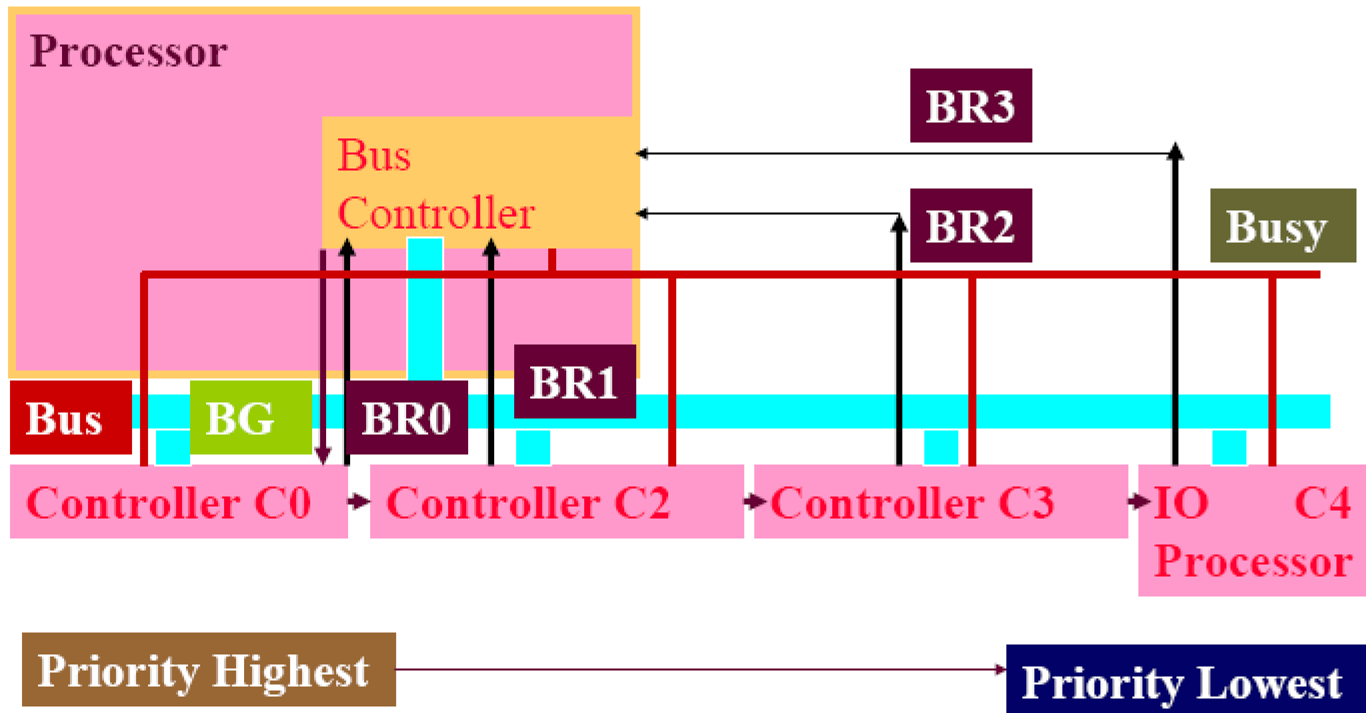
# Methods in bus arbitration

- Daisy Chain method
- Independent Bus request and grant method
- Polling method

# Daisy chain method

- Centralized bus arbitration process
- Bus control passes from one to the other based on priority
- BG signal will function likes a token and pass from high priority controller to lower priority controller
- At each instance of bus access, the $i^{th}$ controller gets the higher priority compared to bus $(i+1)^{th}$

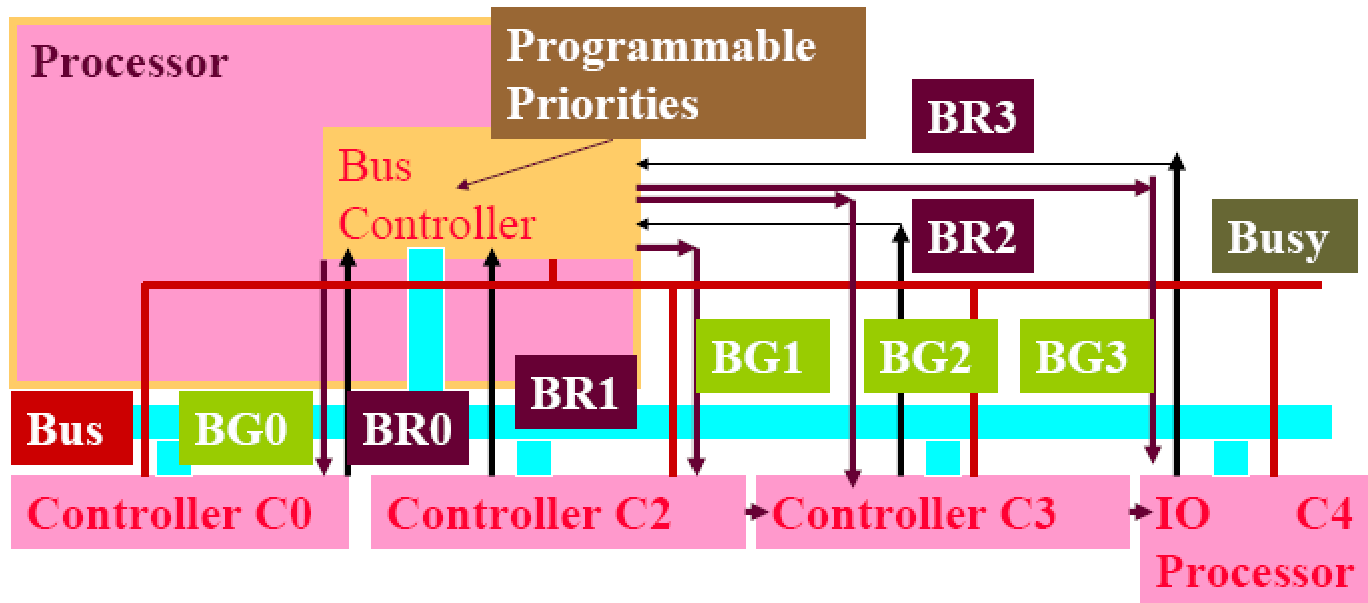
# Daisy Chain Method



# Independent request and grant

- Separate BR and BG signals
- Priority of the bus can be controlled dynamically
- Controller sends BRi and once it recieves BGi, it issues Busy signal
- Any controller which find Busy signal will not issue a bus request

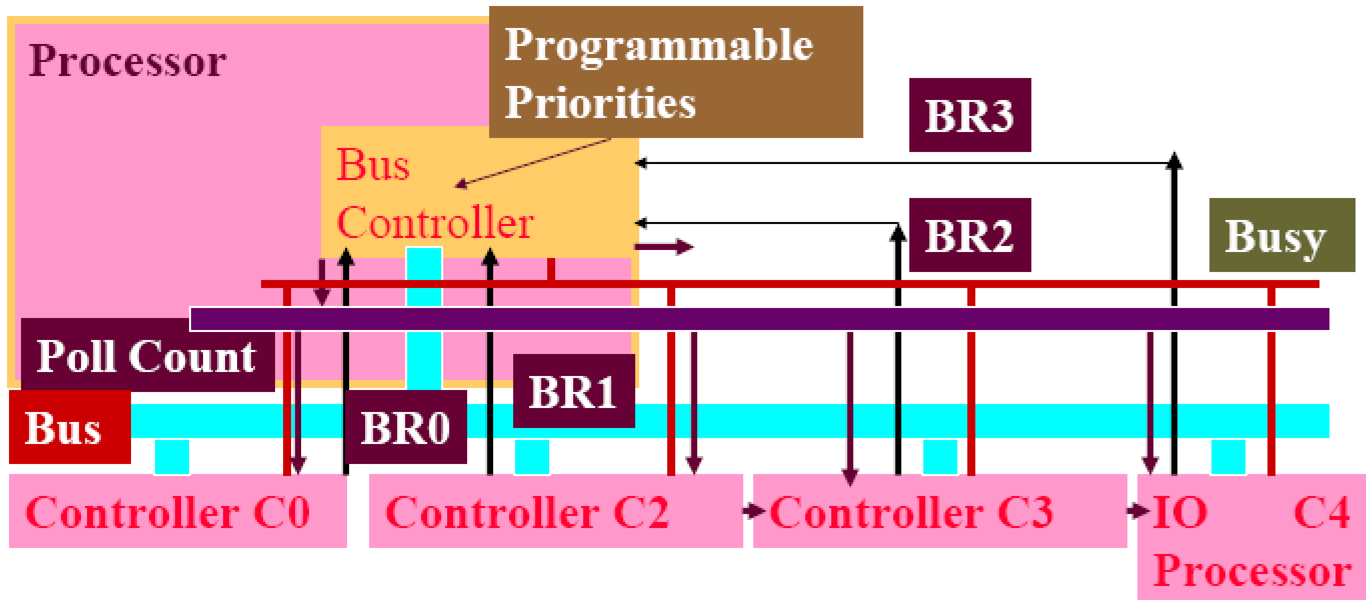
# Independent request and grant



# Polling method

- Each controller has a predefined poll count number
- After finish using the bus, the bus master first sent its current poll count and keep increment.
- On count==i, the controller i can send the request (BR signal) and now it owns the bus

# Polling

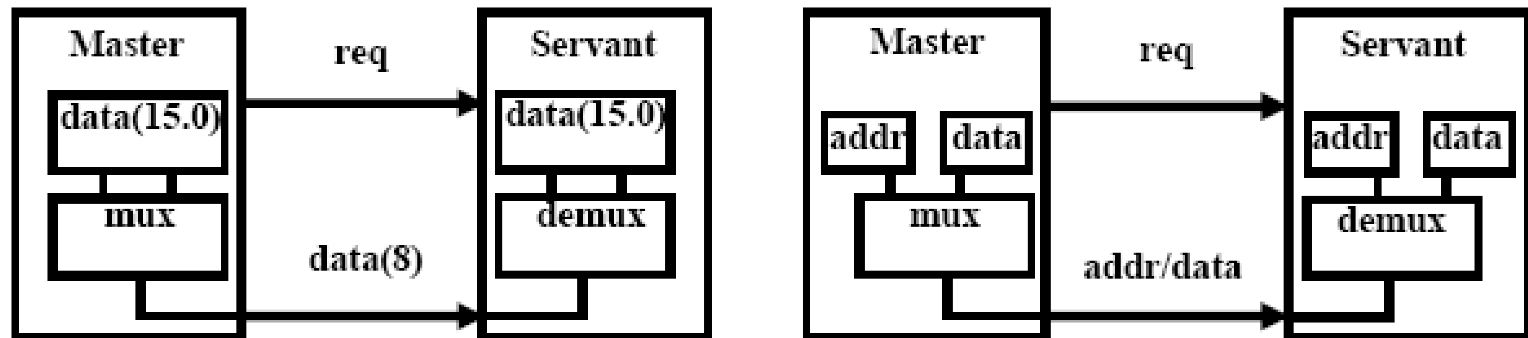


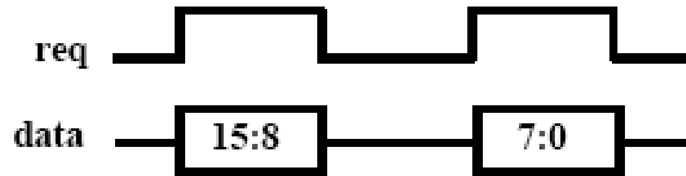
# Advantages/Disadvantages

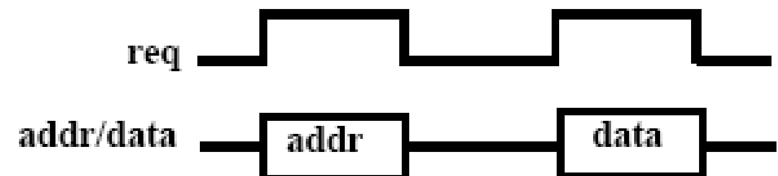
- Daisy Chain Method
  - Simple, control and priorities are fixed
  - Can cause starvation
- Independent and grant
  - Complex, require dynamic arbitration
  - Nearly fair arbitration
- Polling Method
  - Controller next to the bus master get highest priority
  - Require polling (keep processor busy)

# Time multiplexing

Time-multiplexed data transfer




Data serializing

Address/data muxing

# I/O addressing

- Bus-based I/O

  Processor has address, data, and control ports that form a single bus

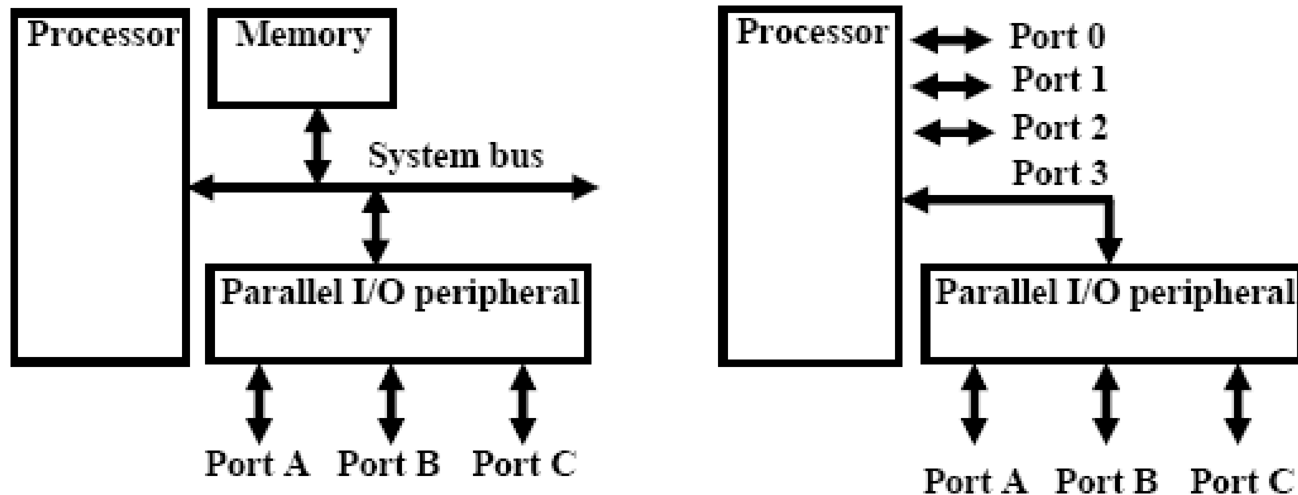
- Port-based I/O

  Processor has one ore more N-bit ports

  Read and Write a port like a register

# Parallel I/O Peripheral

- Extend I/O capabilities



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

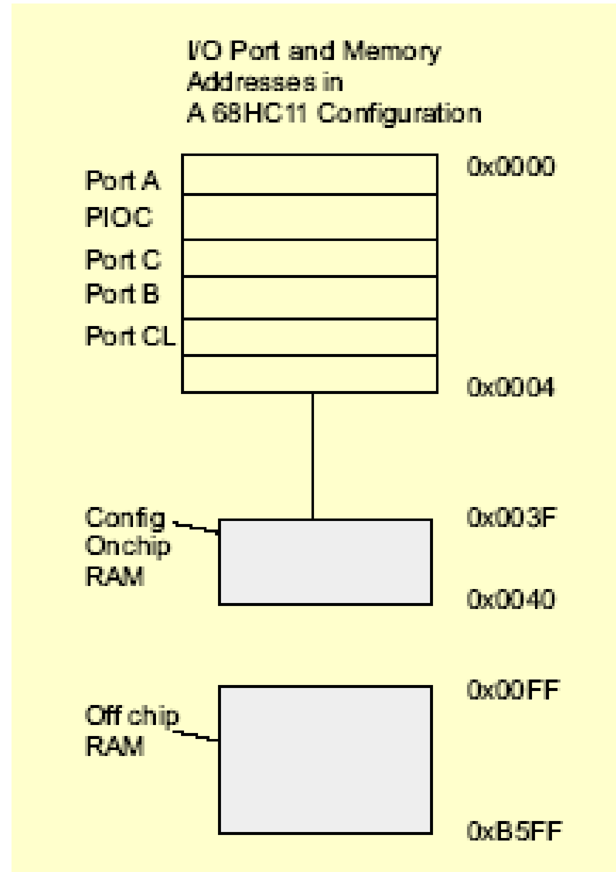
Memory-mapped I/O

- Peripheral registers occupy memory space same as memory

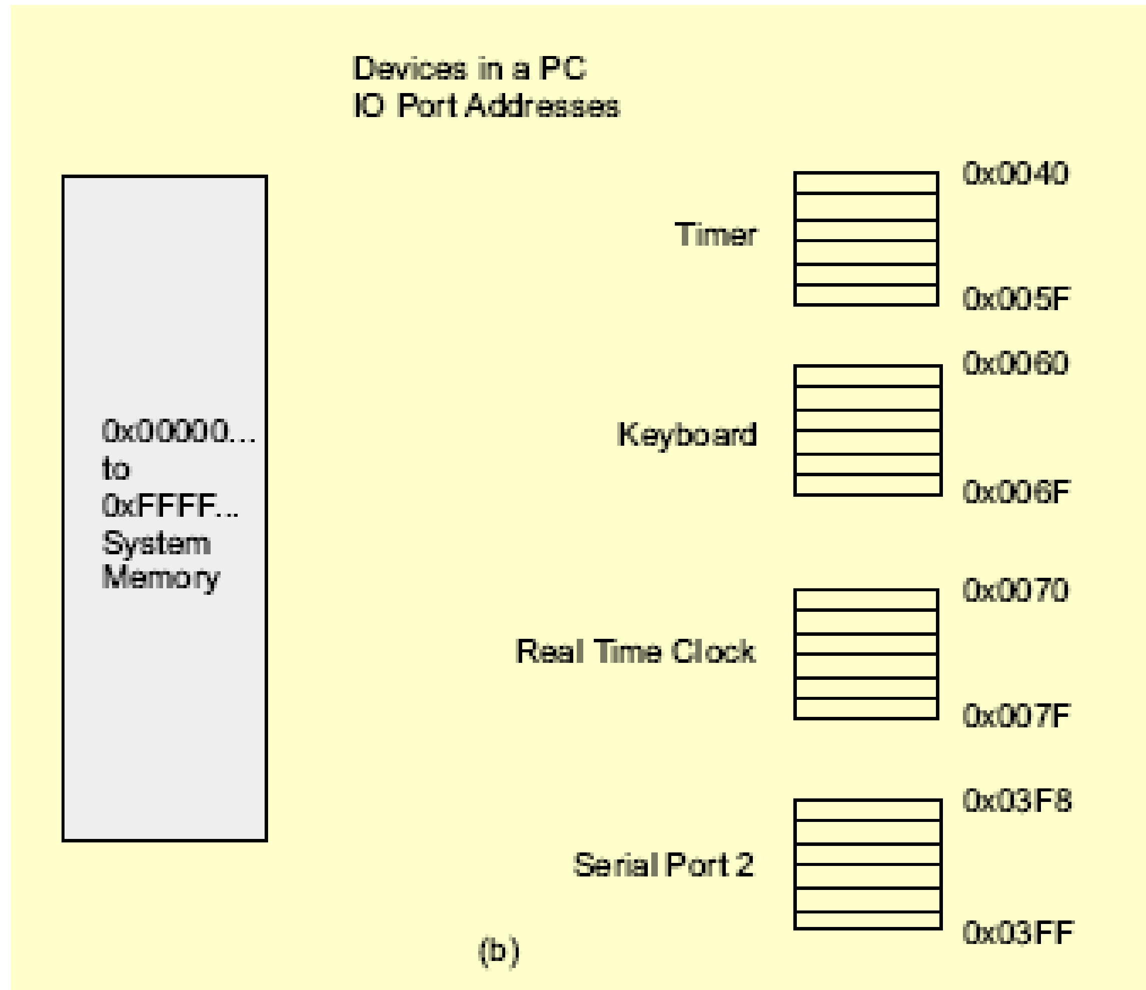
Standard I/O

- Additional pin indicates whether memory or peripheral

# Memory Mapped I/O



# Standard I/O or I/O mapped I/O



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

Memory-mapped I/O

- Request no special instructions

Standard I/O

- No loss of memory space
- Simpler address decoding

# I/O Transfers

- I/O devices are orders of magnitude slower than a Processor. Handshake lines are used to synchronize each transfer.
- I/O device sets a hardware handshake line when it is ready to transfer data. (input ready, output ready)
- Before any data transfer, the Processor must check that the I/O device is ready by reading the device's handshake line.
- Handshake line bits are connected to another I/O port (status port). Typically the next I/O address after the I/O data port.
- When the Processor reads or writes the data port, the handshake line is usually reset in hardware.

# Learn how to communicate

- Programmed I/O (Software Polling)
- Interrupt Driven I/O
- Direct Memory Access (DMA)

# Programmed I/O (Polling)

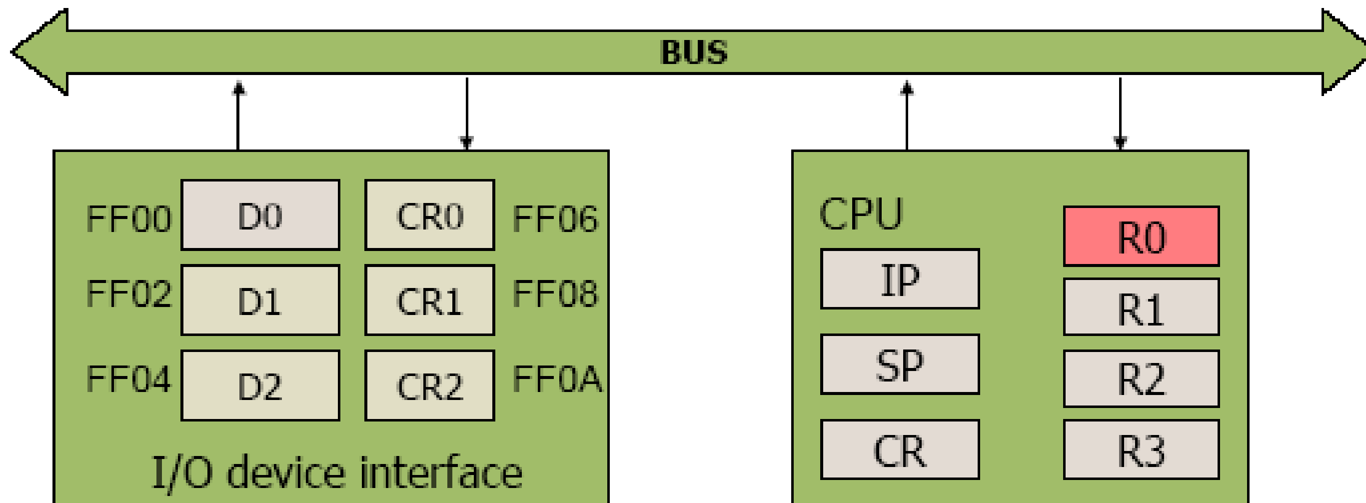
- Processor must read and check I/O ready bits for proper value before a transfer. Requires looping, reading status port, and constantly waiting for correct value.
- Processor then inputs or outputs data value to/from I/O device.
- This approach is limited to systems with just a few I/O devices
  - Processor overhead to loop and check in software

# I/O operation

- Structure of an I/O operation
  - Phase 1: prepare the device for the operation
    - In case of output, data is transferred to the data buffer registers
    - The operation parameters are set with the control registers
    - The operation is triggered
  - Phase 2: wait for the operation to be performed
    - Devices are much slower than the processor
    - It may take a while to get/put the data on the device
  - Phase 3: complete the operation
    - Error checking
    - Clean up the control registers

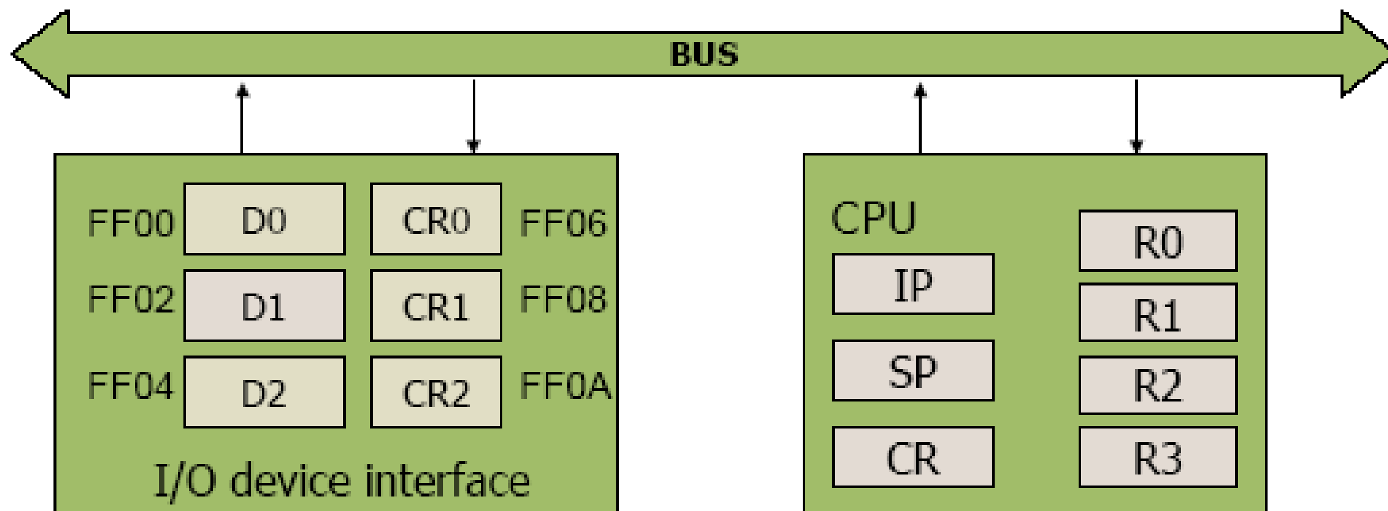
# Example of input operation

- Phase 1: nothing
- Phase 2: wait until bit 0 of CR0 becomes 1
- Phase 3: read data from D0 and reset bit 0 of CR0



# Example of output operation

- Phase 1: write data to D1 and set bit 0 of CR1
- Phase 2: wait for bit 1 of CR1 to become 1
- Phase 3: clean CR1



# Programmed I/O Pseudo Code

```
// loop until output ready=1
do {        // read status and mask off ready bit
   status = read_port(status_port_address);
   output_ready = status & 0x01;
}
while (output_ready == 0);
// output new data value
write_port(data_port_address, data_value);
```

# Polling

- If the device is *very* fast, then polling may be the best approach
  - In fact, an interrupt based mechanism involves at least a pair of process switches
  - It the duration of the operation is less than two process switches, then polling is the most optimized solution

- Almost all devices are *much* slower than the processor
  - usually, a data transfer takes much more than two process switches
  - For these devices, an interrupt-based mechanism is the best approach

# How much slower for I/O device?

# Interrupts

- Processor can service something else until interrupt came
- I/O ready signals generate a hardware Interrupt signal. Eliminates Processor I/O ready wait loops.
- An interrupt signal stops the Processor at the next instruction and the Processor calls the Interrupt service routine (ISR)
- Interrupt service routine transfers the data
- Interrupt routines must save all registers on the stack for the interrupt return to work (ISRs like subroutines - but “called” by a hardware signal)

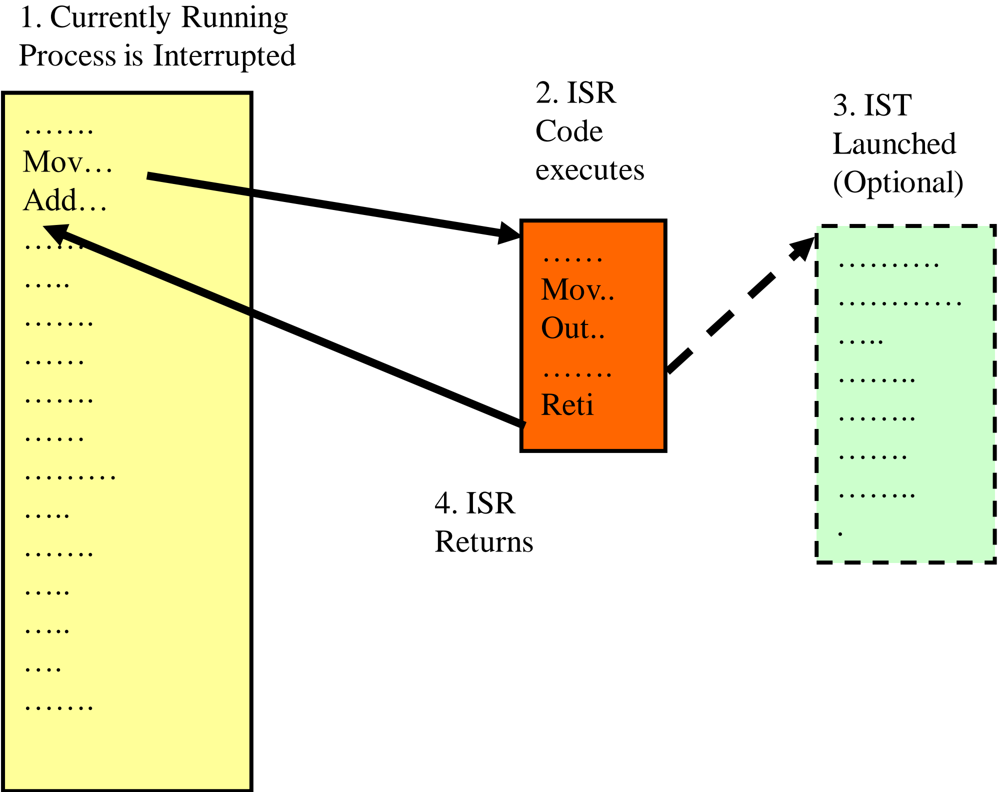
# Interrupts

- Interrupt control hardware is needed to support and enable/disable multiple interrupt signals
- Lower number interrupts typically have a higher priority (can even interrupt a higher number interrupt's ISR) Gives fast I/O devices priority.
- Most modern processors have a vectored interrupt system. Each interrupt jumps to a different ISR address (X86 uses a table lookup of addresses in low memory to jump to ISR)

# Interrupt vector

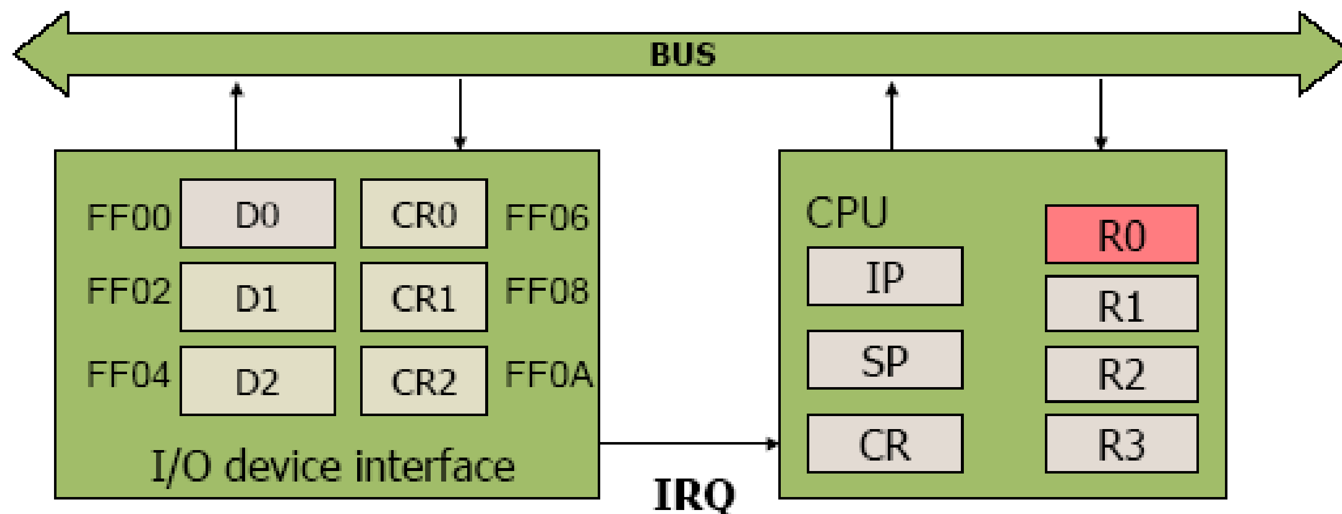
| Interrupt Type | Generating Device |
|---|---|
| 8 | Timer/Counter #0 |
| 17 | Zilog 85230 SCC |
| 18 | Timer/Counter #1 |
| 19 | Timer/Counter #2 |
| 20 | Serial Port Receive |
| 21 | Serial Port Transmit |

# Servicing an Interrupt



# Input with interrupts

- Phase 1: do nothing
- Phase 2: execute other code
- Phase 3: upon reception of the interrupt, read data from D0, clean CR0 and return to the interrupted code



# Interrupt handler

Line follower robot inside the maze

```
Main CPU function()
while(true){
  Map_computation();
}
```

```
Interrupt handler()
If left sensor
    turn right
If right sensor
    turn left
```

# Temporal diagram

User Process

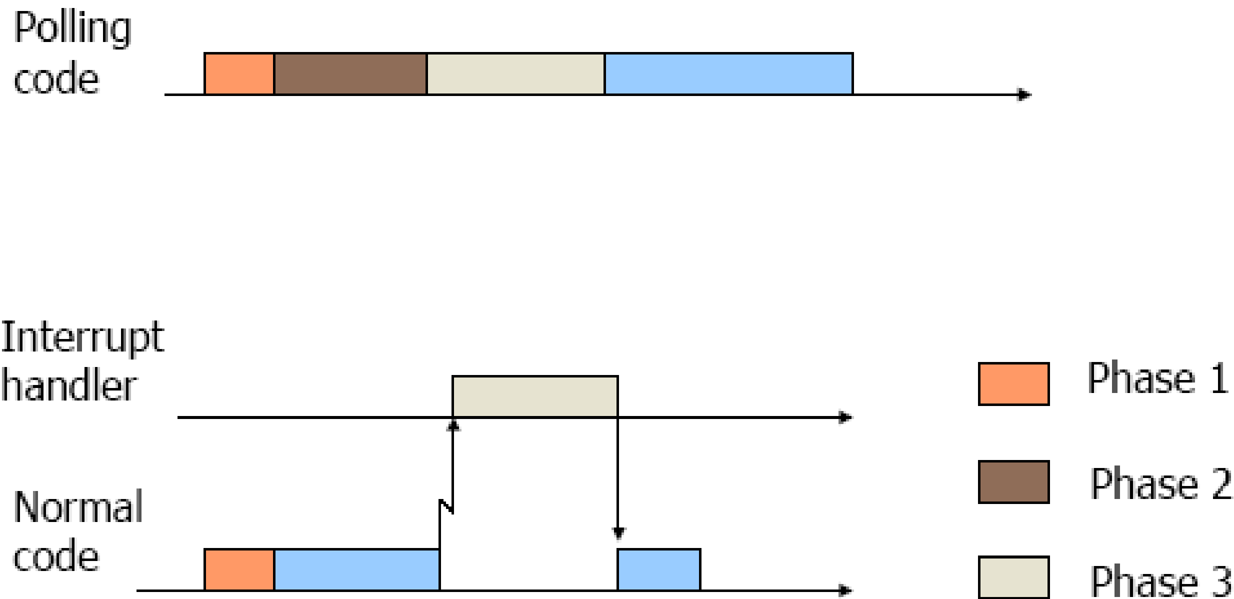
Internal Process

Interrupt Handler

External Process (device)

# Polling vs. Interrupt

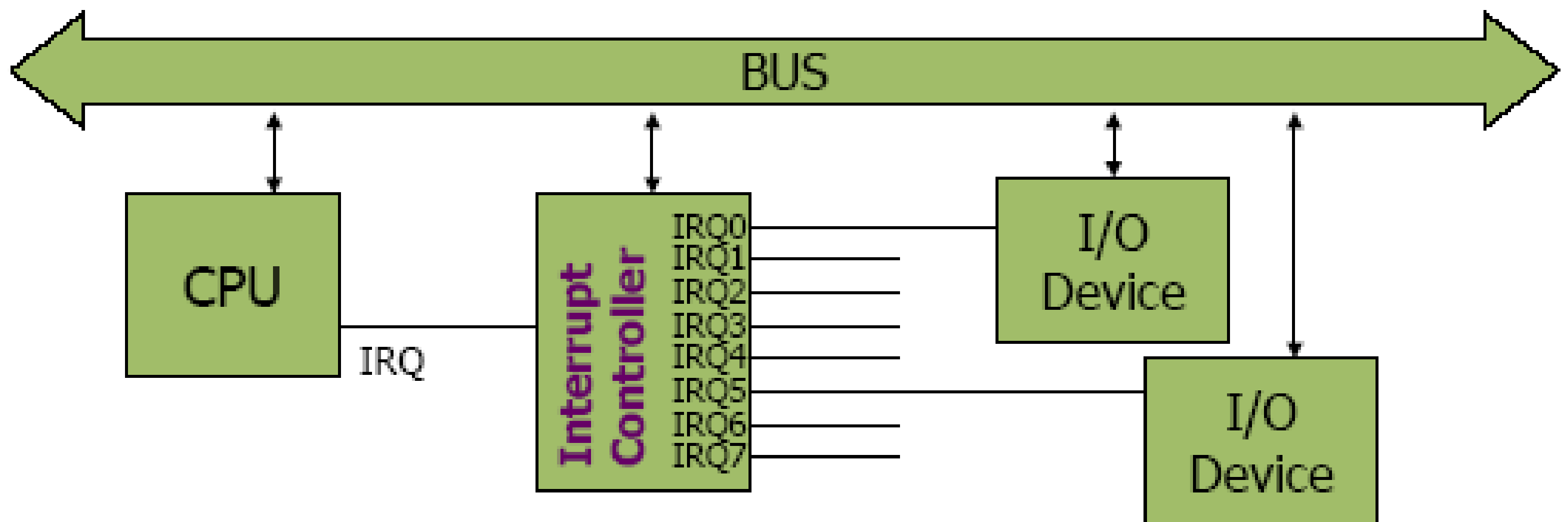
- Let's compare polling and interrupt



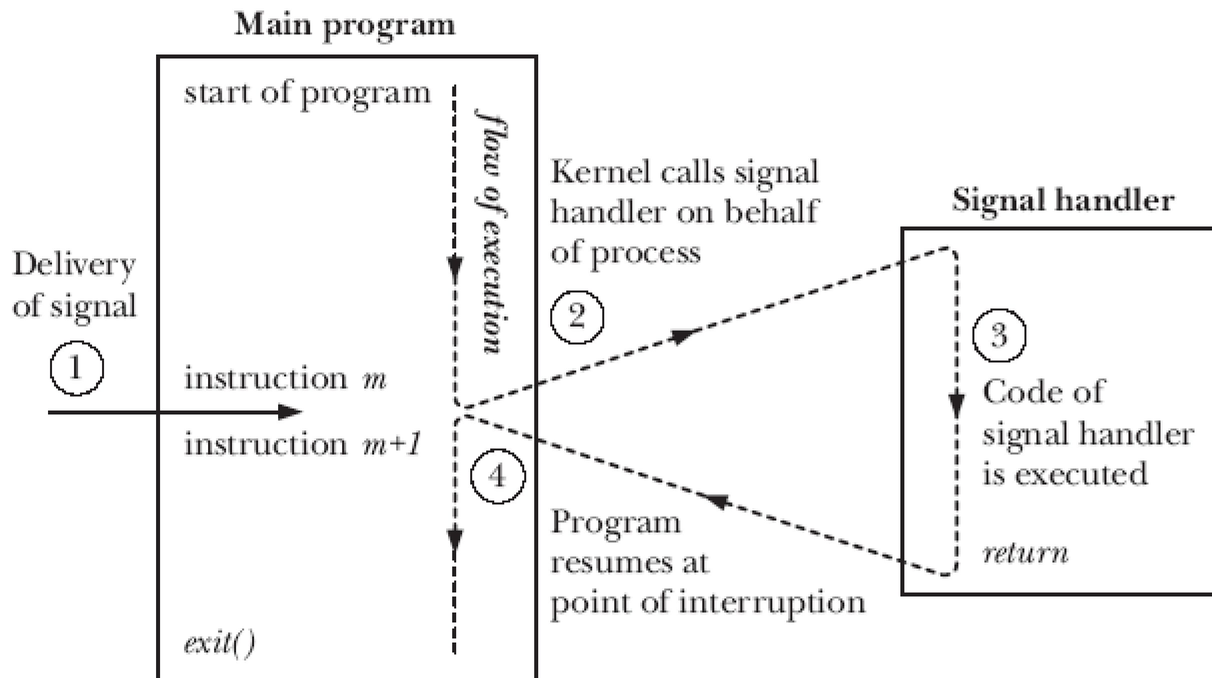

Phase 1: prepare the device for the operation
Phase 2: wait for the operation to be performed
Phase 3: complete the operation

# Many interrupt support

- Usually, processor has one single IRQ pin
  - However, there are several different I/O devices
  - Intel processors use an external Interrupt Controller
- 8 IRQ input lines, one output line



# Signal handler and handler execution



# Interrupts – Other Issues

- Interrupt latency time – set by longest instruction or longest block of code that disables interrupts. Determines the max time to respond to an interrupt. ISR may launch an interrupt service thread (IST) to complete operations so that it can return in less time.
- Some critical code (non reentrant) might require that interrupts be disabled. Most machines have an enable and disable interrupt instruction. But doing this impacts interrupt latency time.

# Interrupts

- Interrupt controller hardware is needed to set priority and enable/disable each interrupt signal

- Additional bus signals needed
  - ISA – IRQx lines
  - PCI – INTx lines

- For security, the OS must control all interrupt hardware, the vector table, and the ISRs.

# What do you need to know before using interrupt?

# Interrupt Timing Example

- Assume you want to sample an input at 1000Hz using interrupts.
- A hardware timer could be used to generate the interrupt request signal at a 1000Hz rate.
- 1/1000Hz is 1ms, so the ISR would need to execute in less than 1ms.
- But this would not leave any time for the OS and other applications to run!

# Interrupt Timing Example

- The ISR should consume only a small percentage of the available CPU time so that it does not impact the OS,applications and other ISRs that are also running.
- Lets assume now that the ISR should use only 5% of the CPU cycles, so we have 0.05 x 1ms = 50 us for the maximum ISR execution time (us = $10^{-6}$ second)

# Interrupt Service Time

- For an RTOS, one would expect an interrupt response time (time between interrupt hardware signal and ISR start) to be 0 – 100 us. (i.e., CE is around 10-20us)
- Service time = Worst case time to respond to interrupt + ISR execution time
- A General Purpose Desktop OS will be a lot slower (i.e., Windows XP can be >5000us)
- The OS interrupt response time will limit the maximum interrupt rate possible

# Ways to Measure Interrupt Times

- Add up ISR instruction execution times
- Measure time on device with test equipment (scope or logic analyzer)
- Use code profiling tools (i.e., CE's kernel tracker remote tool)

**Why Worry about Interrupt Times?**

- Interrupts occurring too fast (i.e., faster than they are being serviced) can eventually crash the OS when stacks overflow or it may also miss interrupts

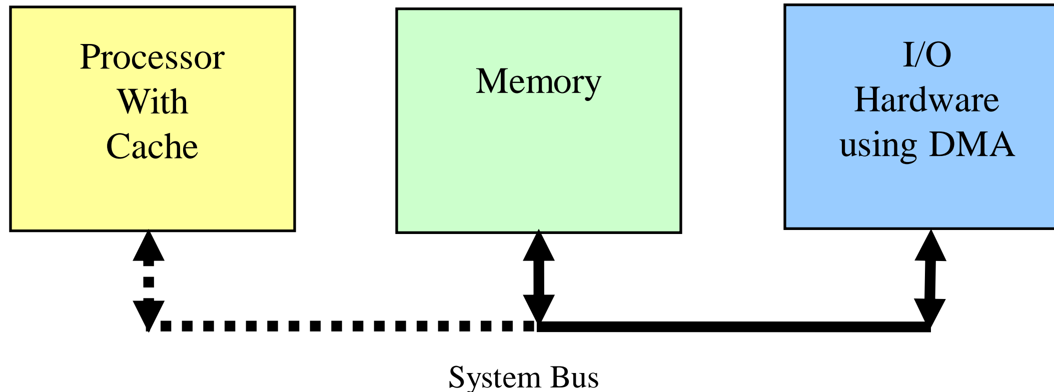
# Direct Memory Access

- A DMA controller transfers blocks of data directly to/from memory and the I/O device.
- DMA controller is a state machine with an address pointer and a counter. Counts down number of memory locations for transfer and drives bus address and control lines (is a Bus Master)
- Processor not involved in transfer once it starts

# Direct Memory Access

- DMA controller and Processor must both share the bus. Need bus arbitration hardware to control bus access (DMAs or Processor).
- DMA controller interrupts Processor when it's block transfer is complete.
- Processor programs DMA controller's address register and counter to start a new transfer.
- Need hardware for each DMA controller and an interrupt system

# DMA Bus Cycle




- Processor does not drive the bus during a DMA bus cycle
- Bus Arbitration hardware is needed to support multiple bus masters

# Direct Memory Access

- Need new bus signals for DMA. DMA controller must request a bus cycle and wait until it is granted by bus arbitration hardware
  - ISA – DRQx and DACKx
  - PCI – REQx and GNTx
- For security, the OS must control all DMA hardware.
- DMA normally used for high-speed block transfer devices like disk drives (disks transfer sectors)
- Processor and DMA controllers can work in parallel on different tasks (overlap of computation and I/O)

# DMA Support

- Phase 1: setup all required control registers for DMA access (OS support)
- Phase 2: do something else
- Phase 3: do something else, OS acknowledge DMA completion

# Comparison

- Polling
  - Faster
- Interrupt
  - More efficient
  - Complex in the case of service many interrupts
  - Limited number of interrupt
  - Lower power
- DMA
  - Less processor overhead
  - Less power (processor can go to sleep)
  - Require hardware support
  - Require interrupt support

# Tradeoffs

| *Transfer Technique* | *Hardware* | *CPU Overhead* |
|---|---|---|
| **Programmed I/O** | | |
| **Interrupt** | | |
| **DMA** | | |

# Questions?