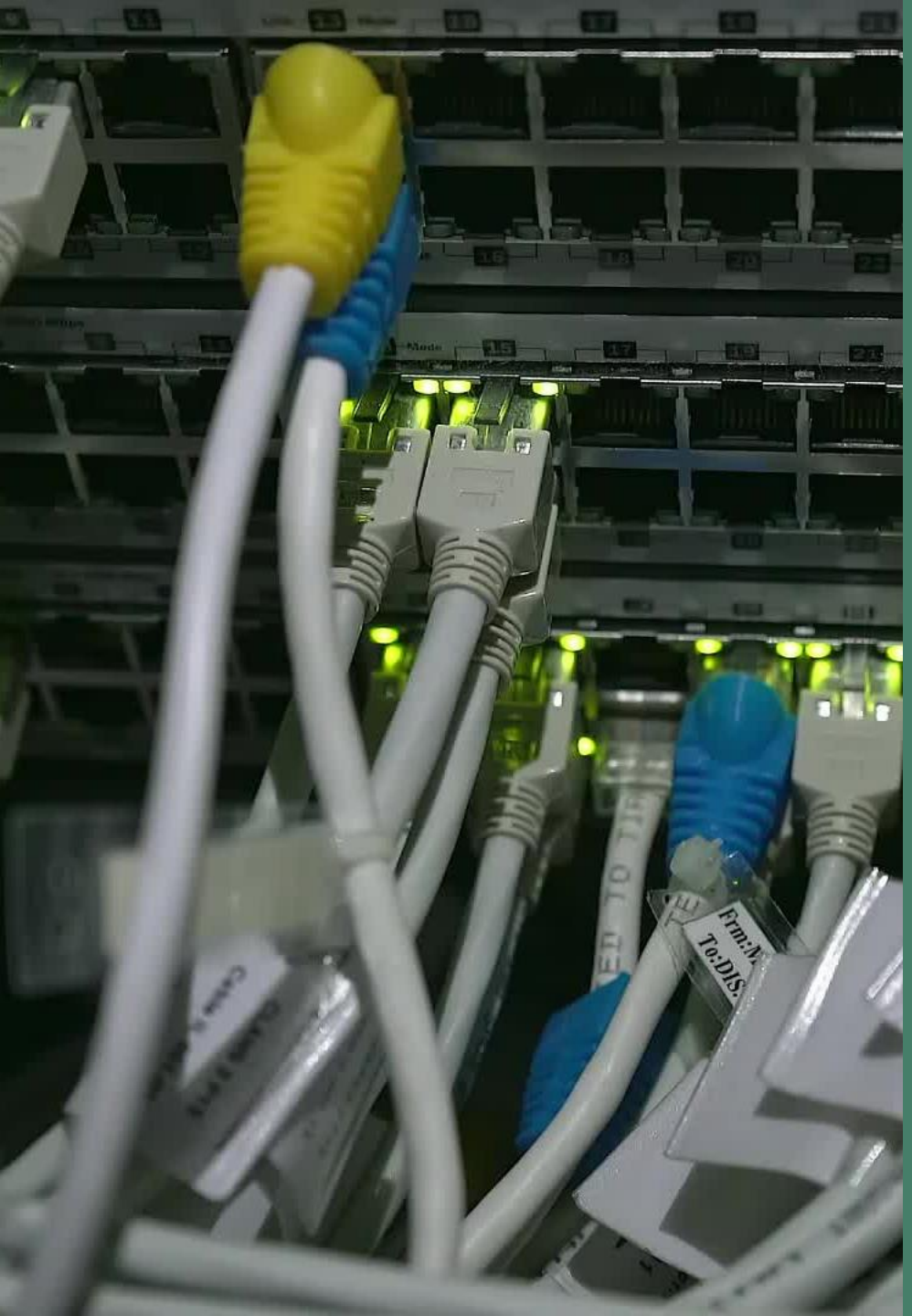


INPUT/OUTPUT (I/O) ORGANIZATION

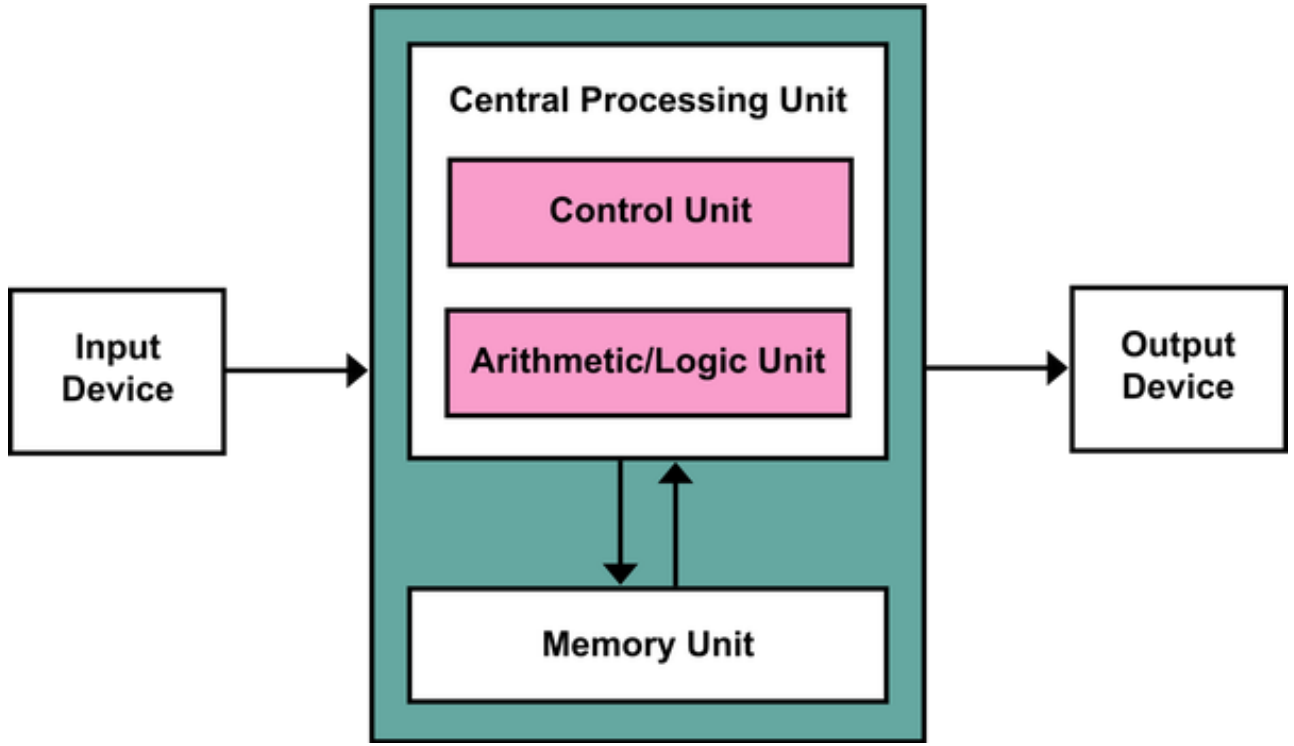
UNIT-3

TOPICS TO BE COVERED

- Accessing I/O Devices
- Interrupts
- Interrupt Hardware
- Enabling and Disabling Interrupts
- Handling Multiple Devices
- Controlling Device request
- Exceptions
- Direct Memory Access
- Bus Interface Circuit
- Standard I/O Interfaces : PCI BUS ; SCSI BUS ; USB

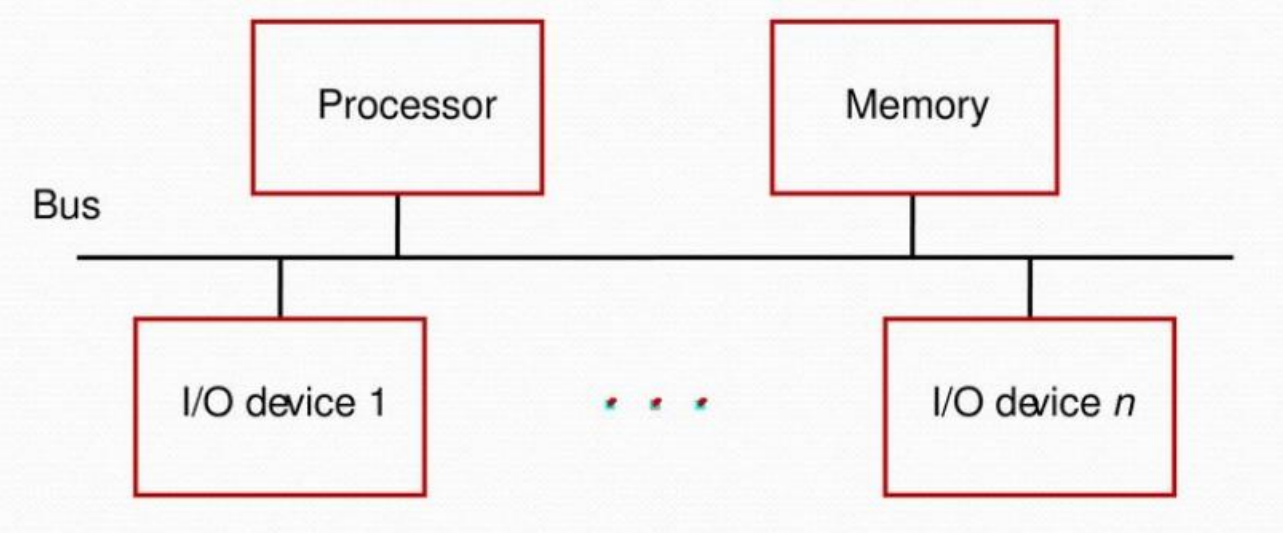
WHAT IS I/O ORGANIZATION ?

I/O Organization provides a systematic means of controlling interaction with the outside world.



ACCESSING I/O DEVICES

- A Simple arrangement to connect I/O devices to a computer is to use a single bus arrangement as shown below

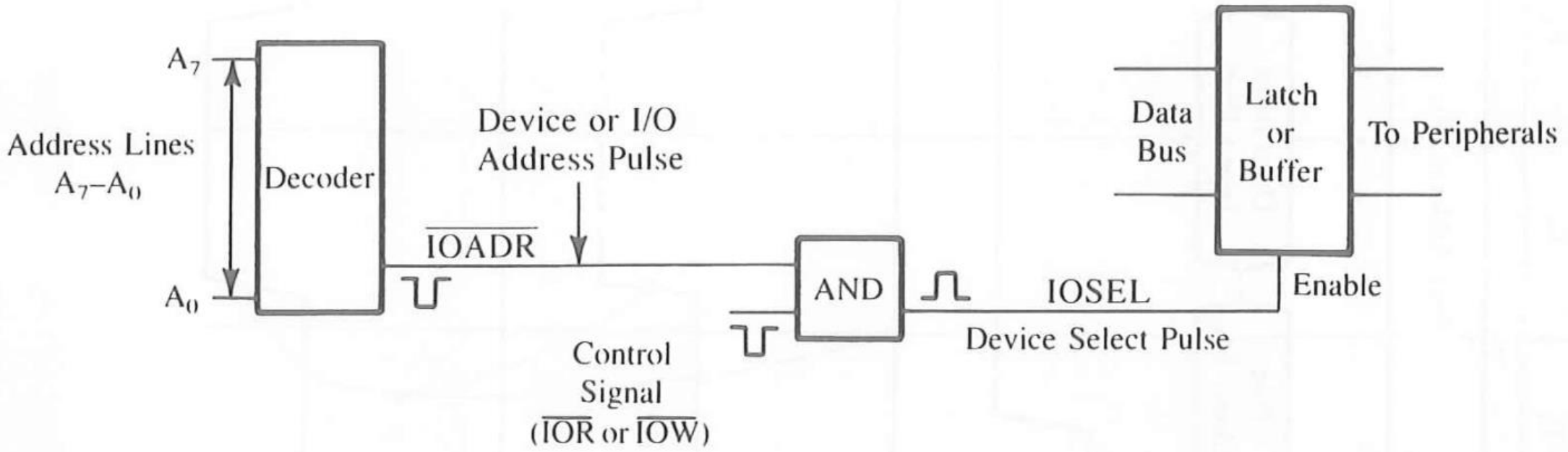


- Multiple I/O devices may be connected to the processor and the memory via a bus.
- Bus consists of three sets of lines to carry address, data and control signals.
- Each I/O device is assigned a unique address.
- To access an I/O device, the processor places the address on the address lines.
- The device recognizes the address and responds to the control signals.

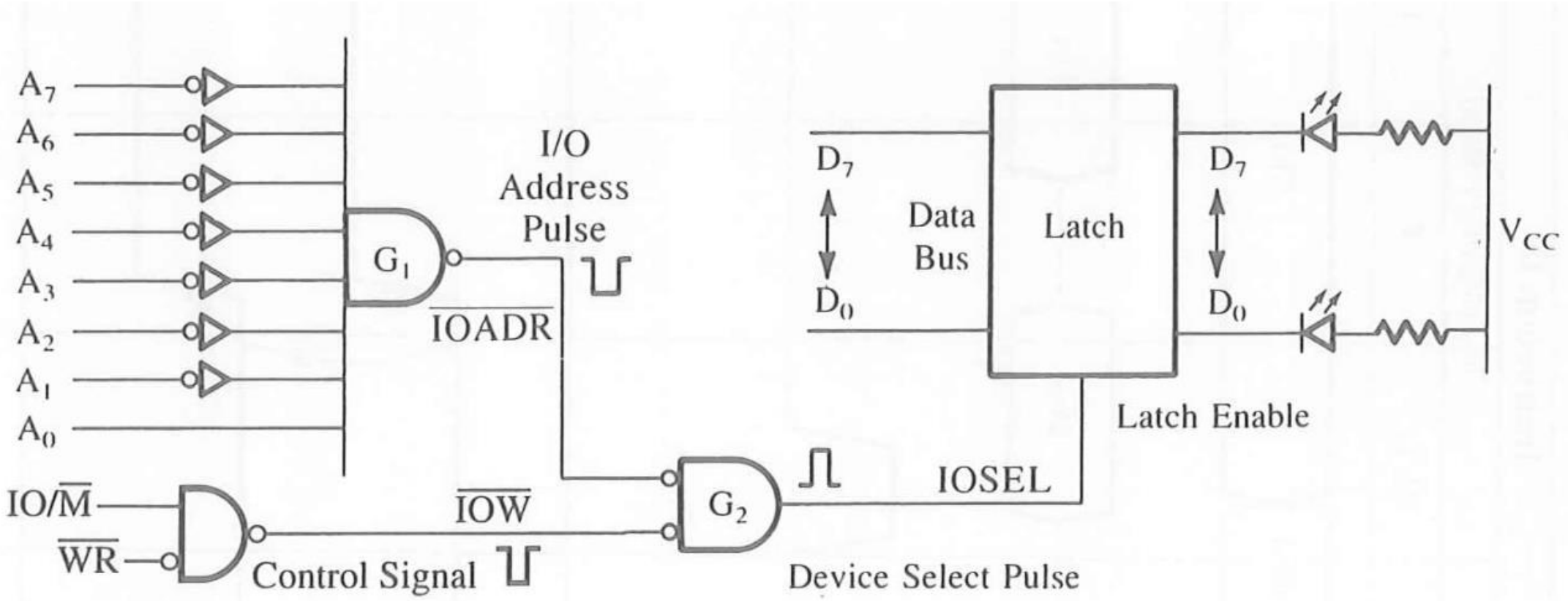
ACCESSING I/O DEVICES

- **I/O devices and the memory may share the same address space:**
  - Memory-mapped I/O.
  - Any machine instruction that can access memory can be used to transfer data to or from an I/O device.
  - Simpler software.
- **I/O devices and the memory may have different address spaces:**
  - Peripheral-mapped I/O or I/O – mapped I/O
  - Special instructions to transfer data to and from I/O devices.
  - I/O devices may have to deal with fewer address lines.
  - I/O address lines need not be physically separate from memory address lines. In fact, address lines may be shared between I/O devices and memory, with a control signal to indicate whether it is a memory address or an I/O address.

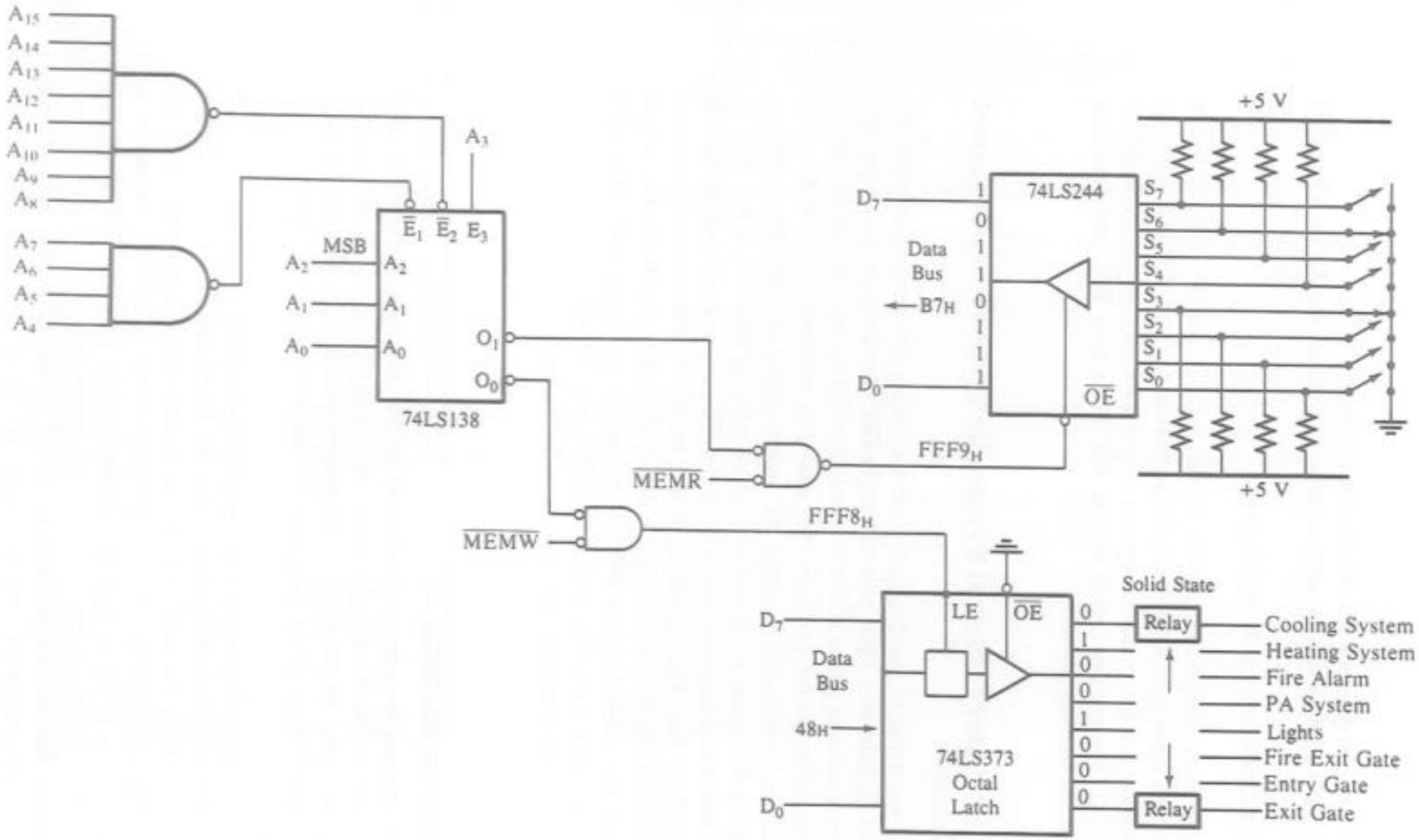
Memory mapped I/O	I/O mapped I/O
1. In this device address is 16 bit. Thus A_0 to A_{15} lines are used to generate device address.	1. In this I/O device address is 8 bit. Thus A_0 to A_7 or A_8 to A_{15} lines are used to generate device address.
2. $\overline{MEMR}$ and $\overline{MEMW}$ control signals are used to control read and write I/O operations.	2. $\overline{IOR}$ and $\overline{IOW}$ control signals are used to control read and write I/O operations.
3. Instructions available are LDA addr, STA addr, LDAX rp, STAX rp, MOV M,R, MOV R,M ADD M, CMP M etc.	3. Instructions available are IN and OUT.
4. Data transfer is between any register and I/O device.	4. Data transfer is between accumulator and I/O device.
5. Maximum number of I/O devices are 65536 (theoretically).	5. Maximum number of I/O devices are 256.
6. Execution speed using LDA addr, STA addr is 13 T-state and 7 T-states for MOV M, r and MOV r, M instructions.	6. Execution speed is 10 T-states.
7. Decoding 16 bit address may require more hardware.	7. Decoding 8 bit address will require less hardware.



Example of Peripheral-mapped I/O interface for 8085 based system



Example of Peripheral-mapped I/O interface for 8085 based LED output port



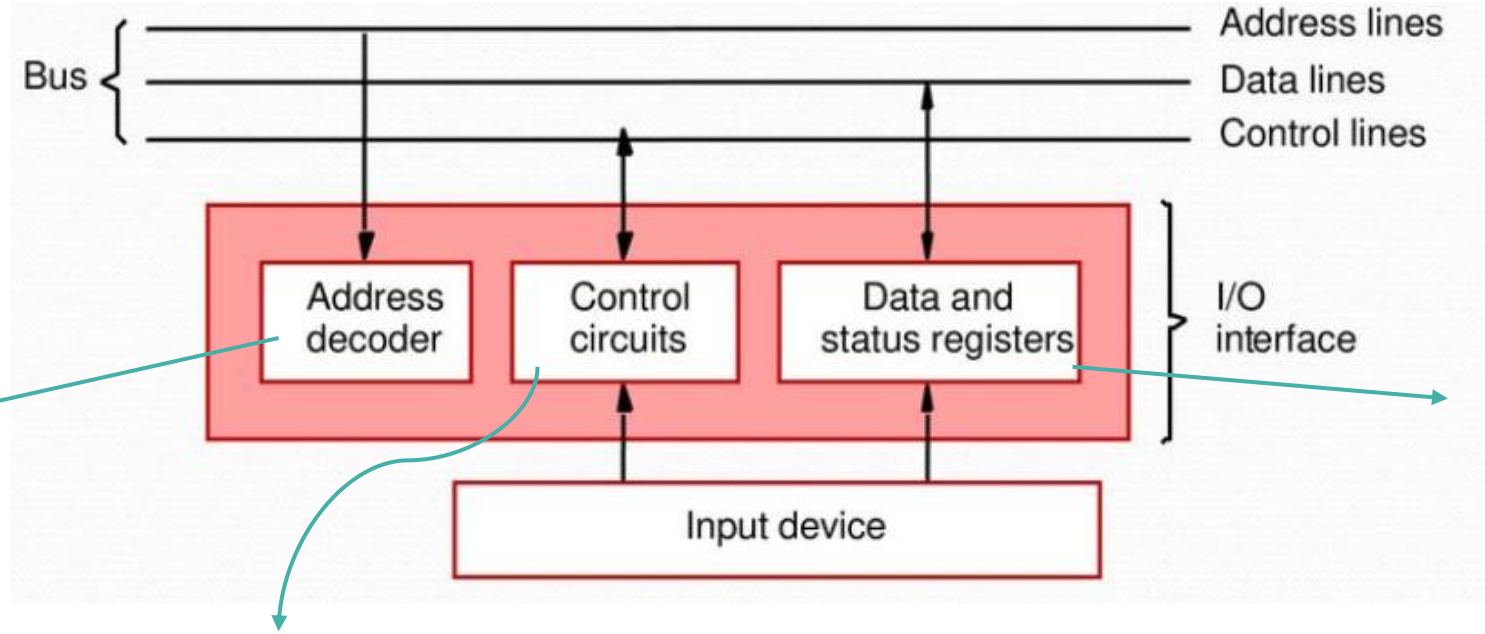
Example of Memory-mapped I/O interface for 8085 based system

ACCESSING I/O DEVICES

- I/O device is connected to the bus using an I/O interface circuit which has:-

  Address decoder, control circuit, data and status registers.

Address decoder decodes the address placed on the address lines thus enabling the device to recognize its address



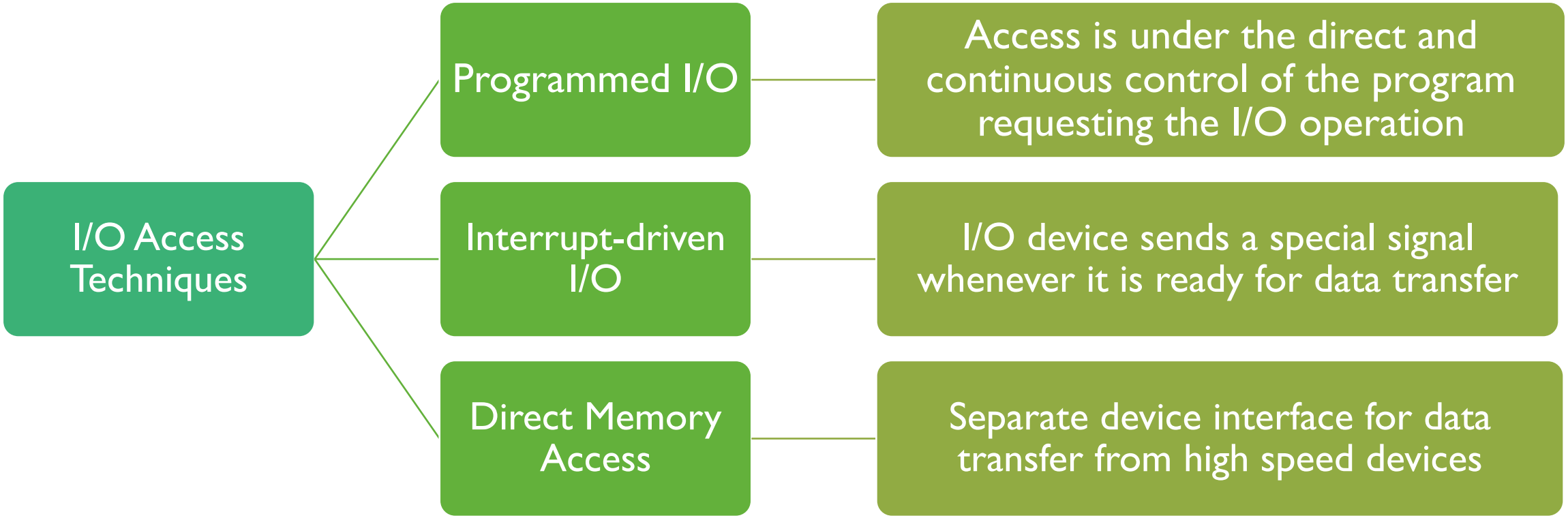
Control circuit coordinates I/O transfers.

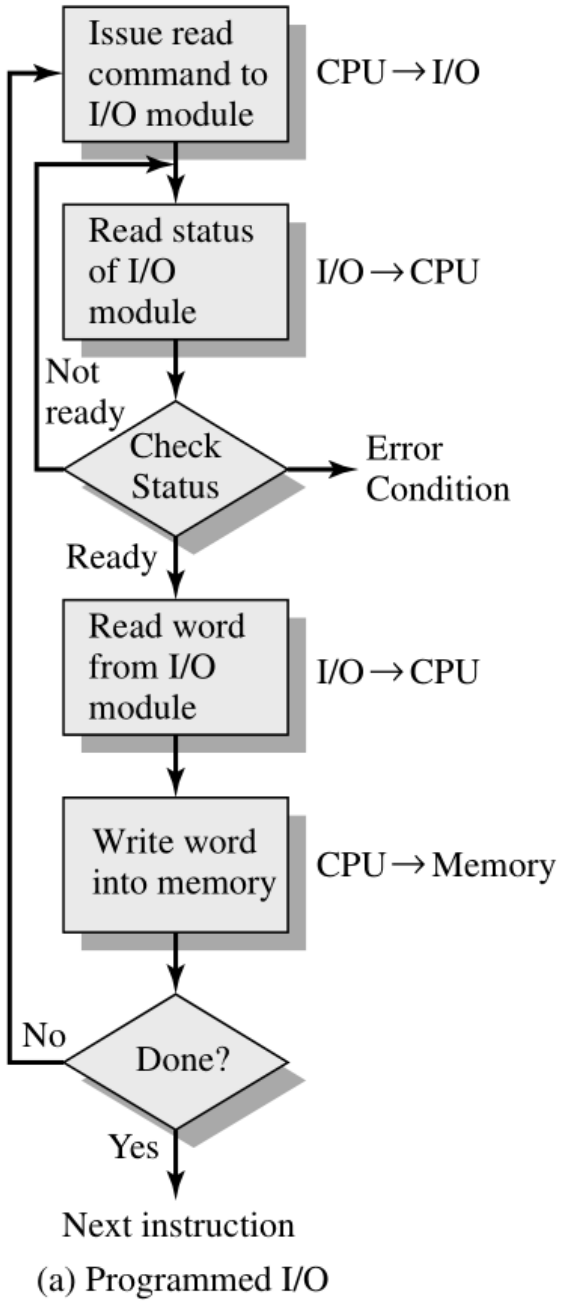
Data register holds the data being transferred to or from the processor.

Status register holds information necessary for the operation of the I/O device.

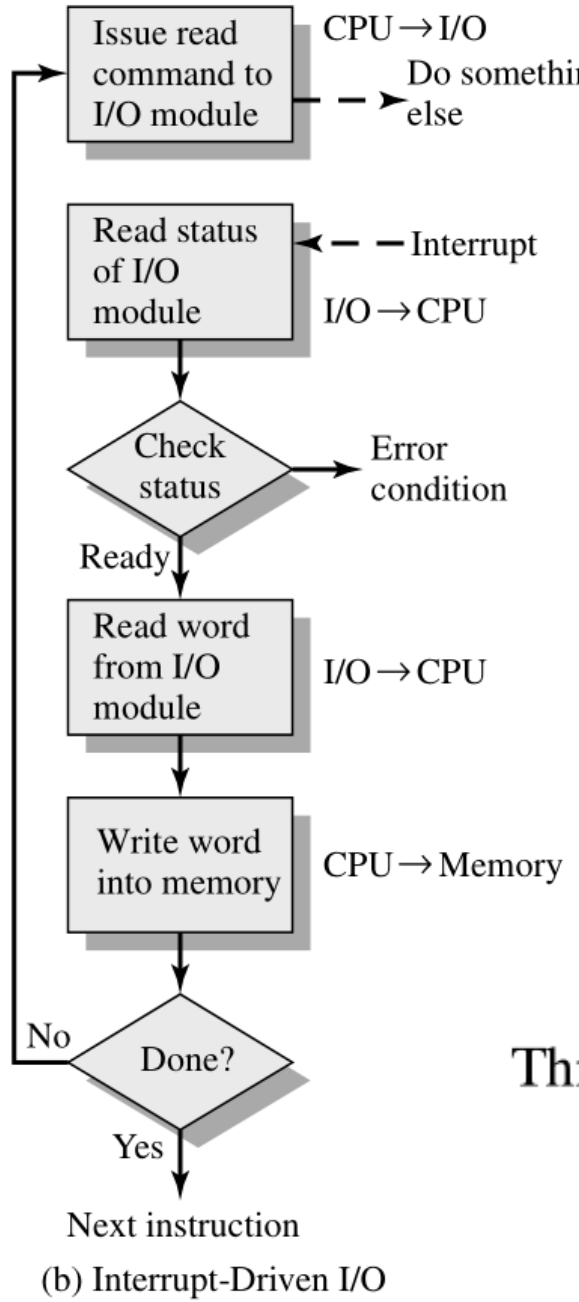
ACCESSING I/O DEVICES

- The rate of transfer to and from I/O devices is slower than the speed of the processor. Then, When and How data must be accessed from I/O devices??

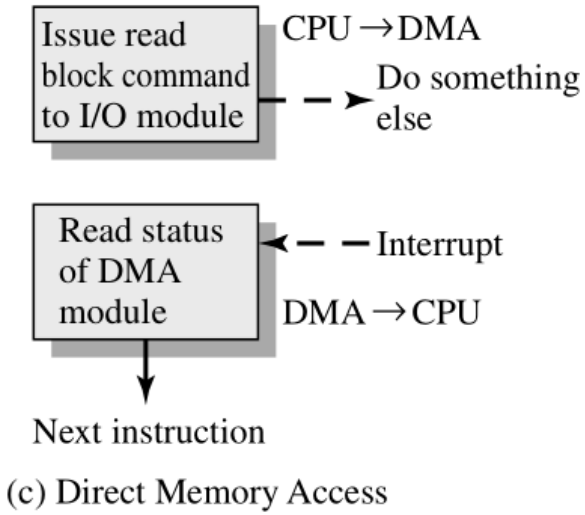




(a) Programmed I/O



(b) Interrupt-Driven I/O

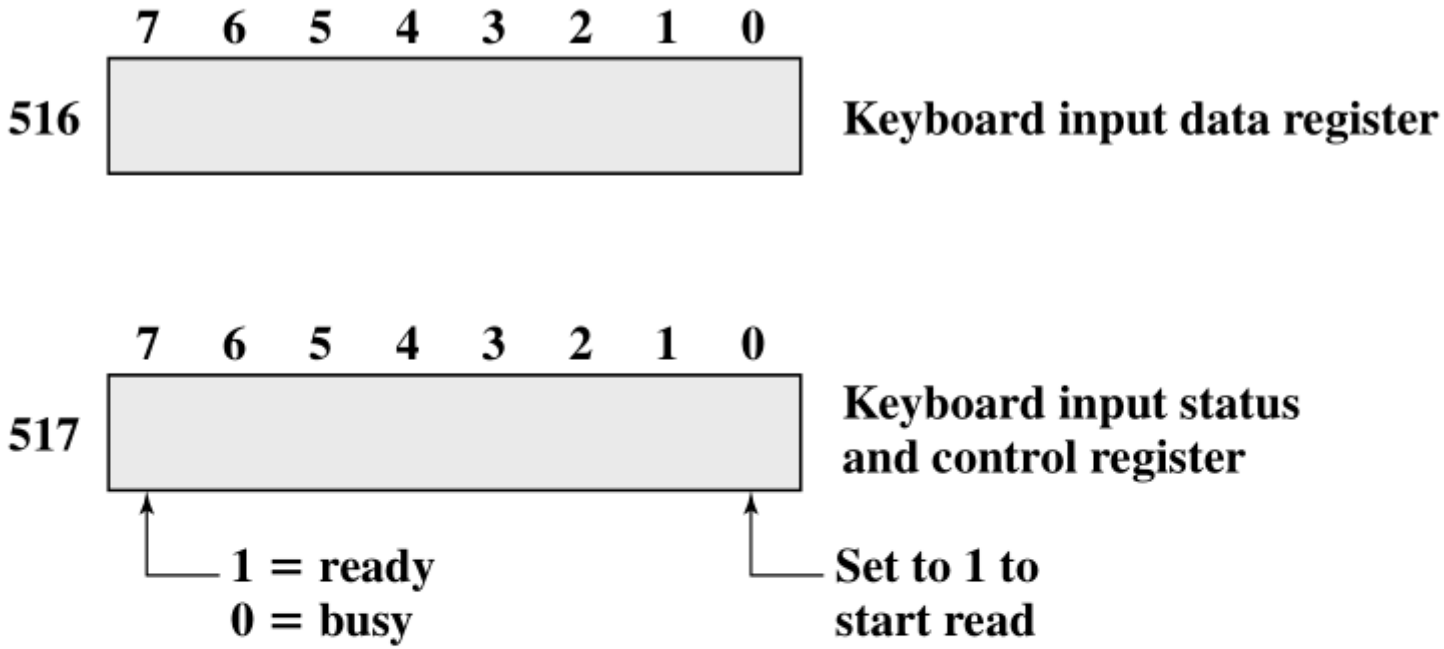


(c) Direct Memory Access

Three Techniques for Input of a Block of Data

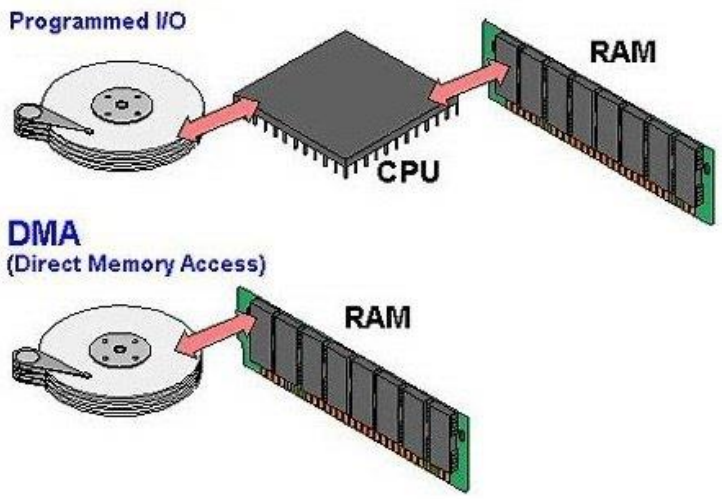
No.	Programmed I/O	Interrupt Driven I/O
1.	In programmed I/O, processor has to check each I/O device in sequence and in effect 'ask' each one if it needs communication with the processor. This checking is achieved by continuous polling cycle and hence processor can not execute other instructions in sequence.	External asynchronous input is used to tell the processor that I/O device needs its service and hence processor does not have to check whether I/O device needs it service or not.
2.	During polling processor is busy and therefore, have serious and decremental effect on system throughput.	In interrupt driven I/O, the processor is allowed to execute its instructions in sequence and only stop to service I/O device when it is told to do so by the device itself. This increases system throughput.
3.	It is implemented without interrupt hardware support.	It is implemented using interrupt hardware support.
4.	It does not depend on interrupt status.	Interrupt must be enabled to process interrupt driven I/O.
5.	It does not need initialization of stack.	It needs initialization of stack.
6.	System throughput decreases as number of I/O devices connected in the system increases.	System throughput does not depend on number of I/O devices connected in the system.

Example of Programmed I/O: Reading Input from Keyboard



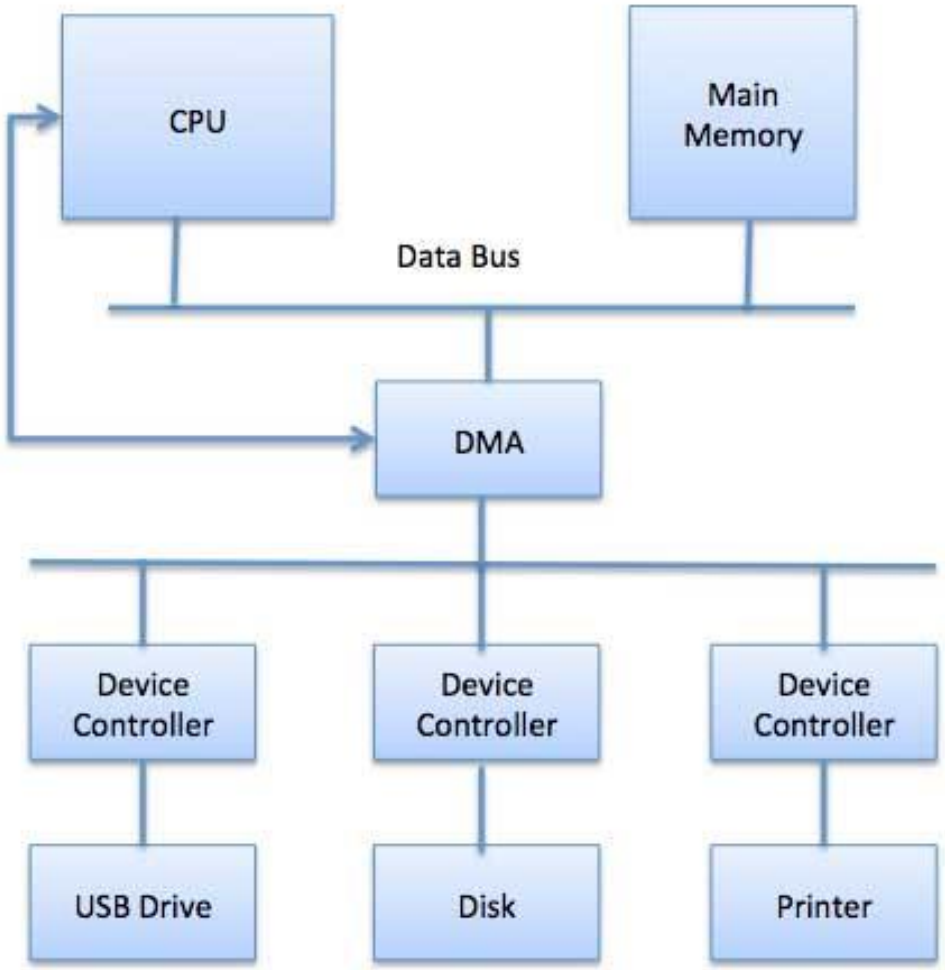
ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

- Programmed I/O is ok for sending commands, receiving status, and communication of a small amount of data
- Inefficient for a large amount of data
  - Keeps CPU busy during the transfer
  - Programmed I/O ≡ memory operations ➲ slow
- Direct Memory Access
  - Device read/write directly from/to memory
  - Transfer from memory to device typically initiated from CPU
  - Transfer from device to memory can be initiated by the device or the CPU



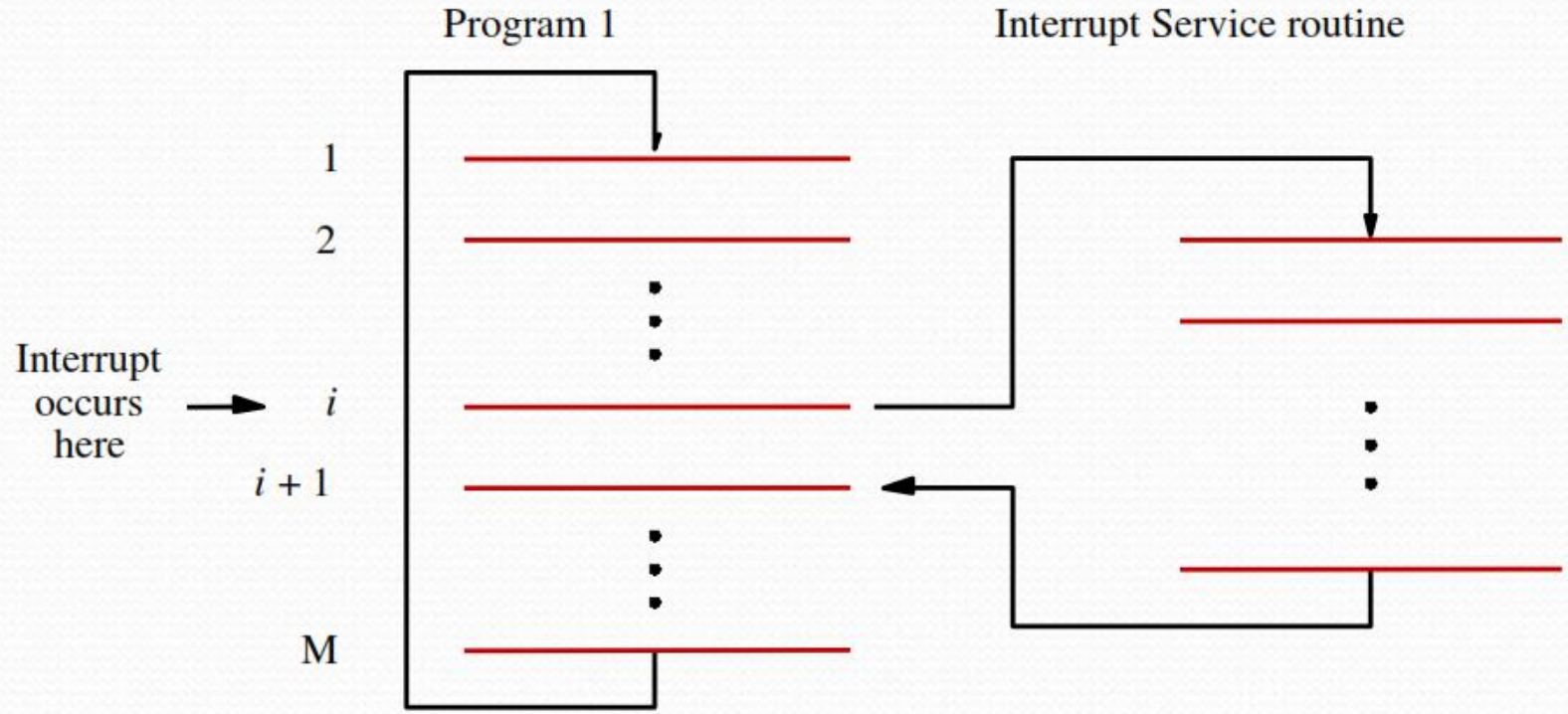
Programmed I/O vs. DMA

Basic Idea about DMA



INTERRUPTS

- In Program Controlled I/O, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation.
- Using INTERRUPT, other tasks can be performed while waiting for the I/O device to become ready.
- I/O device alerts the processor using the **hardware signal called an interrupt** when it becomes ready.
- At least one of the bus control lines, called an **interrupt-request line**, is usually dedicated for this purpose.



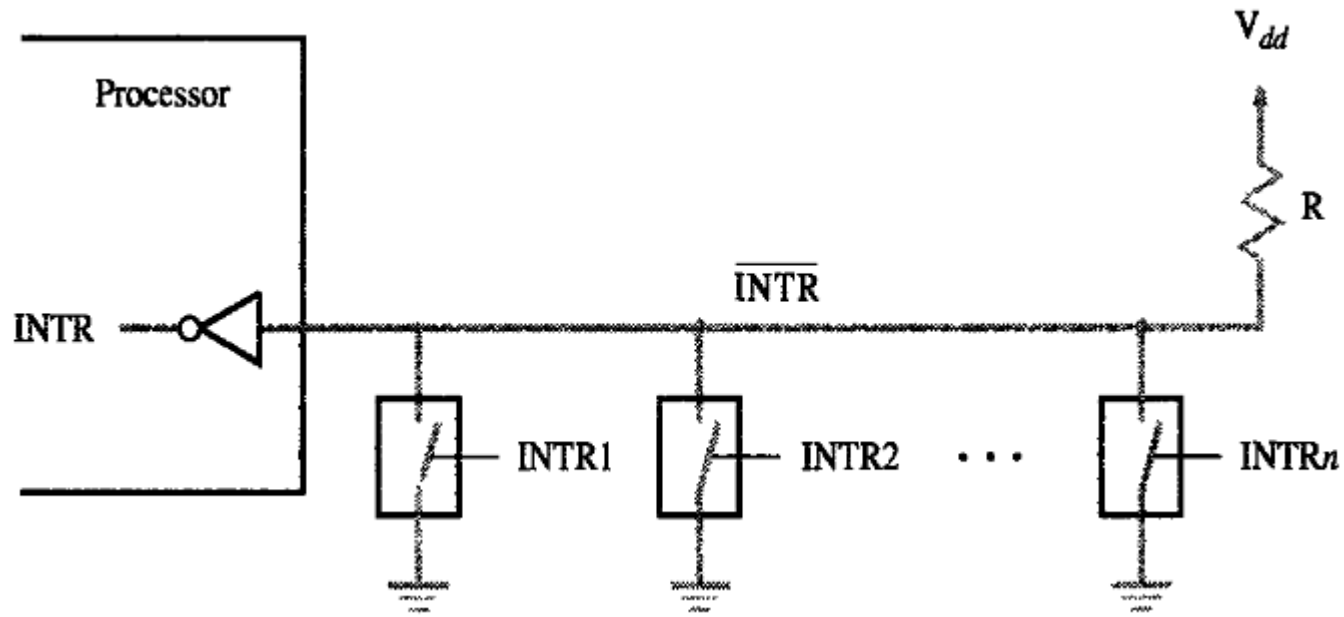
- *Processor is executing the instruction located at address i when an interrupt occurs.*
- *Routine executed in response to an interrupt request is called the interrupt-service routine.*
- *When an interrupt occurs, control must be transferred to the interrupt service routine.*
- *But before transferring control, the current contents of the PC (i+1), must be saved in a known location.*
- *This will enable the return-from-interrupt instruction to resume execution at i+1.*
- *Return address, or the contents of the PC are usually stored on the processor stack.*

- Treatment of an interrupt-service routine is very similar to that of a subroutine.
- However there are significant differences:
  - A subroutine performs a task that is required by the calling program.
  - Interrupt-service routine may not have anything in common with the program it interrupts.
  - Interrupt-service routine and the program that it interrupts may belong to different users.
  - As a result, before branching to the interrupt-service routine, not only the PC, but other information such as condition code flags, and processor registers used by both the interrupted program and the interrupt service routine must be stored.
  - This will enable the interrupted program to resume execution upon return from interrupt service routine.

- Saving and restoring information can be done automatically by the processor or explicitly by program instructions.
- Saving and restoring registers involves memory transfers:
  - Increases the total execution time.
  - Increases the delay between the time an interrupt request is received, and the start of execution of the interrupt-service routine. This delay is called interrupt latency.
- In order to reduce the interrupt latency, most processors save only the minimal amount of information:
  - This minimal amount of information includes Program Counter and processor status registers.
- Any additional information that must be saved, must be saved explicitly by the program instructions at the beginning of the interrupt service routine.

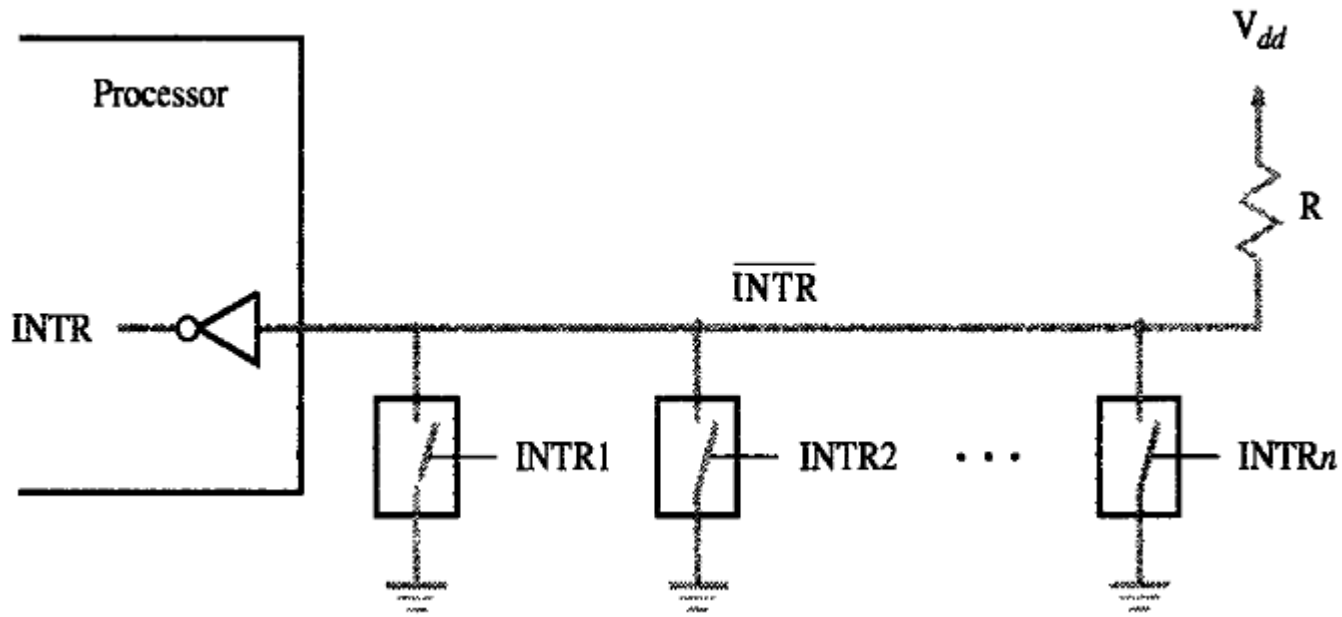
- When a processor receives an interrupt-request, it must branch to the interrupt service routine.
- It must also inform the device that it has recognized the interrupt request.
- This can be accomplished in two ways:
  - Some processors have an explicit interrupt-acknowledge control signal for this purpose.
  - In other cases, the data transfer that takes place between the device and the processor can be used to inform the device.

INTERRUPT HARDWARE



- A single interrupt-request line may be used to serve *n* devices
- All devices are connected to the line via switches to ground.
- To request an interrupt, a device closes its associated switch.
- if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, interrupt-request line will be equal to V_{dd}. → inactive state of the line.

INTERRUPT HARDWARE



- a device requests an interrupt by closing its switch, the voltage on the line drops to 0,
- causing the interrupt-request signal, INTR, received by the processor to go to 1.
- INTR is the logical OR of the requests from individual devices, that is,

$$INTR = INTR_1 + \cdots + INTR_n$$

ENABLING AND DISABLING OF INTERRUPTS

- Interrupt-requests interrupt the execution of a program, and may alter the intended sequence of events:
  - Sometimes such alterations may be undesirable, and must not be allowed.
  - For example, the processor may not want to be interrupted by the same device while executing its interrupt-service routine.
- Processors generally provide the ability to enable and disable such interruptions as desired.
- One simple way is to provide machine instructions such as *Interrupt-enable* and *Interrupt-disable* for this purpose.
- To avoid interruption by the same device during the execution of an interrupt service routine:
  - First instruction of an interrupt service routine can be Interrupt-disable.
  - Last instruction of an interrupt service routine can be Interrupt-enable.

MORE CLARIFICATION ON WHY ENABLING AND DISABLING OF INTERRUPTS IS DONE

Let us consider in detail the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover.

POSSIBILITITES TO AVOID THIS PROBLEM

The first possibility is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur.

POSSIBILITITES TO AVOID THIS PROBLEM

The second option, which is suitable for a simple processor with only one interrupt-request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called *Interrupt-enable*, indicates whether interrupts are enabled. An interrupt request received while this bit is equal to 1 will be accepted. After saving the contents of the PS on the stack, with the Interrupt-enable bit equal to 1, the processor clears the Interrupt-enable bit in its PS register, thus disabling further interrupts. When a Return-from-interrupt instruction is executed, the contents of the PS are restored from the stack, setting the Interrupt-enable bit back to 1. Hence, interrupts are again enabled.

POSSIBILITITES TO AVOID THIS PROBLEM

In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be *edge-triggered*. In this case, the processor will receive only one request, regardless of how long the line is activated. Hence, there is no danger of multiple interruptions and no need to explicitly disable interrupt requests from this line.

Before proceeding to study more complex aspects of interrupts, let us summarize the sequence of events involved in handling an interrupt request from a single device. Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

INTERRUPTS: HANDLING MULTIPLE DEVICES

- Let us consider a situation where a number of devices Capable of initiating interrupt Are connected to the processor. Because these devices are operationally independent there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced or several devices may request interrupts at exactly the same time. This gives rise to a number of questions??

  - How does the processor know which device has generated an interrupt?
  - How does the processor know which interrupt service routine needs to be executed?
  - When the processor is executing an interrupt service routine for one device, can other device interrupt the processor?
  - If two interrupt-requests are received simultaneously, then how to break the tie?

INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Polling scheme**
  - When a request is received over the common interrupt request line additional information is needed to identify the particular device that activated the line.
  - The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request it makes one of the bit in its status register equal to one which is known as IRQ bit.
  - To identify the interrupting device one way is to check all the devices connected to the common interrupt request line. The first device having its IRQ bit equal to one will be selected for service.
  - This scheme referred to as polling is easy to implement but its main disadvantage is the time spent checking the IRQ bits of all the devices that may not be required. A better alternative is using vectored interrupts.

INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Vectored Interrupts**
  - To reduce the time involved in the polling process a device requesting an interrupt may identify itself directly to the processor. Then the processor can immediately start executing the corresponding interrupt service routine. The term vectored interrupts refers to all interrupt handling scheme based on this approach.
  - A device requesting an interrupt can identify itself by sending a special code to the processor over the bus.
  - This enables the processor to identify individual devices even if they share a single interrupt request line.
  - The code supplied by the device may represent the starting address of the interrupt service routine for that device, the code length is typically in the range of 4 to 8 bits.
  - The remainder of the address is supplied by the processor based on the area in its memory where the addresses for the interrupt service routines are located.

INTERRUPTS: HANDLING MULTIPLE DEVICES

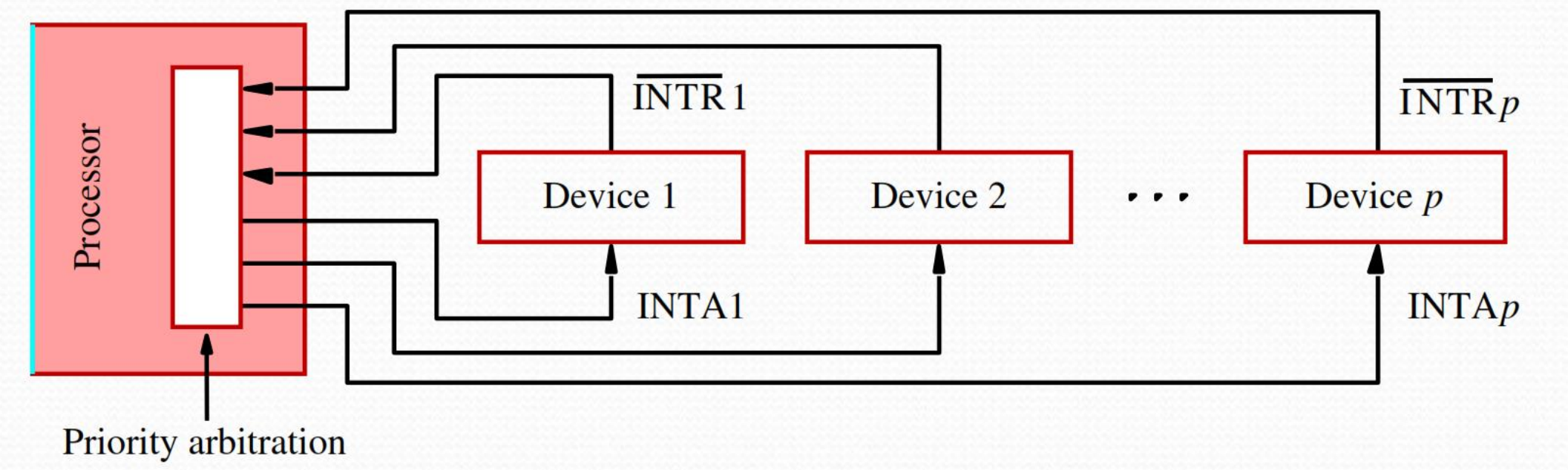
- **Vectored Interrupts**
  - In this arrangement the interrupt service routine for a given device must always start at the same location known as interrupt vector address.
  - In most computers, I/O devices sends the interrupt vector code over the data bus using bus control signal.
  - When a device sends an interrupt request the processor may not be ready to receive the interrupt vector code immediately, the interrupting device must wait to put the data on the bus only when the processor is ready to receive it.
  - When the processor is ready to receive the interrupt vector code it activates the interrupt acknowledgment line INTA.
  - The I/O device response by sending its interrupt vector code and turning off the INTR signal.

INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Interrupt Nesting**
  - Interrupts should be disabled during the execution of an interrupt service routine to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved in which case execution of a given interrupt service routine once started always continues to completion before the processor accepts an interrupt request from the second device. However, delay in responding to an interrupt request may lead to erroneous operation in some cases.
  - To avoid such problem, I/O devices can be organised in a priority structure. An interrupt request from a high priority device should be accepted while the processor is servicing another request from a lower priority device.

INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Interrupt Nesting**
  - A multi level priority organization means that during execution of an interrupt service routine, interrupt request will be accepted from some devices but not from other depending upon the device's priority.
  - A multiple priority scheme can be implemented by using separate interrupt request and interrupt acknowledgment lines for each device as shown below.



INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Interrupt Nesting**
  - Each of the interrupt request lines is assigned a different priority level.
  - Interrupt request received over these lines are sent to a priority arbitration circuit in the processor.
  - A request is accepted only if it has a higher priority level than the currently assigned the processor.
  - When interrupt request is received simultaneously from tw or more devices the processor simply accept the request having the highest priority in case of arrangement where separate lines are used for interrupt request and interrupt acknowledgement.
  - However, if several devices share one interrupt request line then some other mechanism is required to handle simultaneous requests.

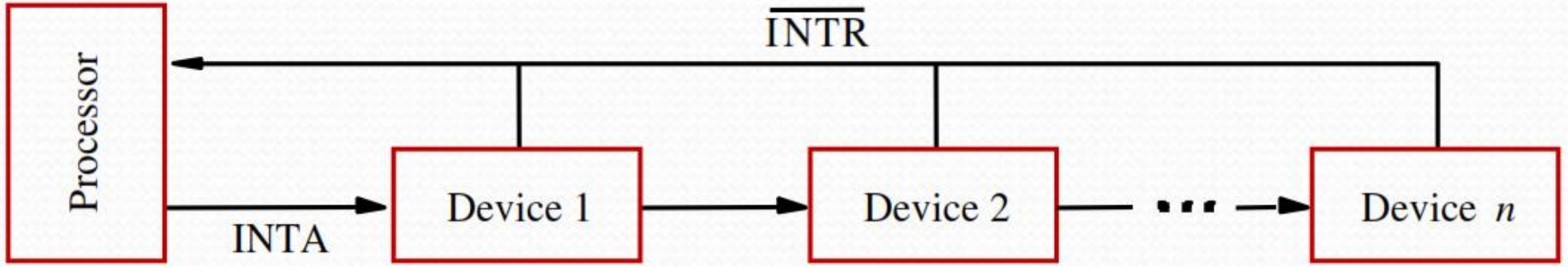
INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Simultaneous Interrupt Request**
  - When the interrupt request line is common to all the devices A widely used scheme to decide the priority of the interrupt is by connecting the interrupt acknowledgement line INTA in a Daisy chain fashion.
  - In this scheme, INTA signal propagates serially through the devices when several devices raises interrupt request and the INTR line is activated.
  - The processor responds by setting the INTA line equal to 1.
  - The signal is received by device 1 and it is passed onto device 2 only if device 1 does not require any service.
  - If device 1 has a pending request for interrupt it blocks the signal and proceeds to put its interrupt vector code on the data line.
  - In the Daisy chain arrangement the device that is electrically closest to the processor has the highest priority the second device along the chain has the second highest priority and so on.

INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Simultaneous Interrupt Request**

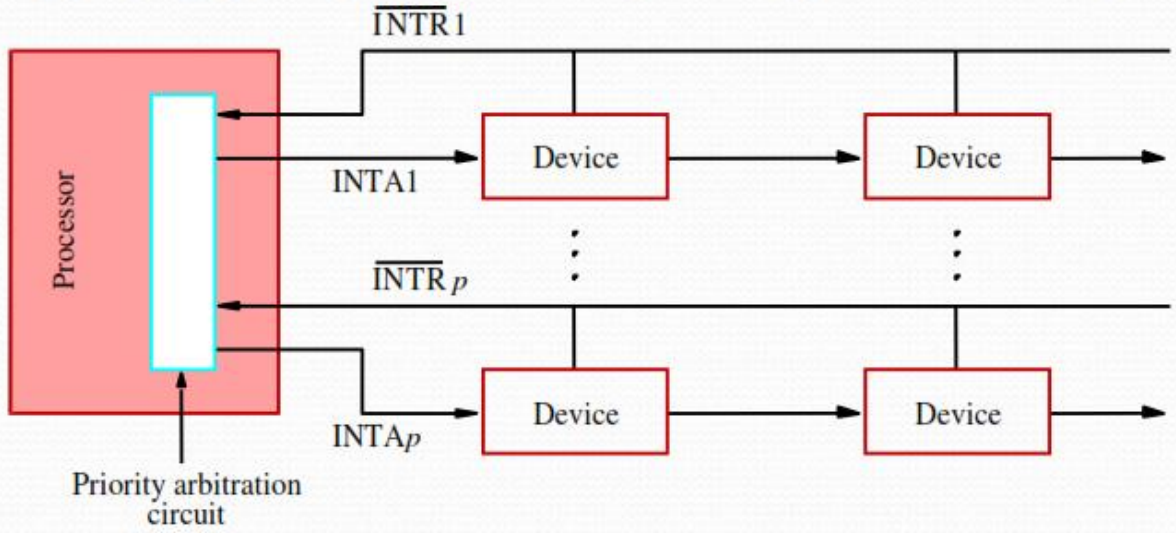
Daisy chain scheme:



INTERRUPTS: HANDLING MULTIPLE DEVICES

- **Simultaneous Interrupt Request**

•When I/O devices were organized into a priority structure, each device had its own interrupt-request and interrupt-acknowledge line.
•When I/O devices were organized in a daisy chain fashion, the devices shared an interrupt-request line, and the interrupt-acknowledge propagated through the devices.
•A combination of priority structure and daisy chain scheme can also used.



•Devices are organized into groups.
•Each group is assigned a different priority level.
•All the devices within a single group share an interrupt-request line, and are connected to form a daisy chain.

INTERRUPTS: CONTROLLING DEVICE REQUESTS

- Until now we have assumed that I/O device interface generates an interrupt request whenever it is ready for I/O transfer.
- But it is important to ensure that interrupt request are generated only by those I/O devices that are being used by a given program. Idle devices must not be allowed to generate interrupt request even though they may be ready to participate in I/O transfer operation.
- Hence, we need a mechanism in the interface circuit of individual devices to control whether I/O device is allowed to generate an interrupt request.
- The control is provided in the form of interrupt enable bit in the device interface circuit.

INTERRUPTS: CONTROLLING DEVICE REQUESTS

- There are 2 independent mechanism for controlling interrupt request:-

1. At the device end, an interrupt enable bit in the control register determines whether the device is allowed to generate an interrupt request.
2. At the processor end, either an interrupt enable bit in the program status register or a priority structure determines whether a given interrupt request will be accepted.

INTERRUPTS: EXCEPTIONS

- The term **exception** is often used to refer to any event that causes the execution of one program to be suspended and the execution of another program to begin. Hence, I/O device interrupts are one example of an exception. This means the interrupt mechanism is used in a number of other situation also.
- **Recovery from errors**
  - Computer use a variety of techniques to ensure that all hardware components are operating properly.
    - For example: error checking code in the main memory which allows detection of error in the stored data if an error occurs the control hardware detects it and inform the processor by raising an interrupt.
    - Another example: error in opcode field of an instruction, arithmetic instruction may attempt a division by zero.

INTERRUPTS: EXCEPTIONS

- **Debugging**
  - System software usually include a program called a debugger which helps the programmer find error in a program.
  - The debugger uses exceptions to provide 2 important facilities called trace and breakpoint.
  - When a processor is operating in trace mode an exception occurs after execution of every instruction.
  - Breakpoint provide a similar facility except that the program being debugged is interrupted only add specific points selected by the user.
  - When trace or breakpoint exception occurs the user can examine the contents of register memory location and so on.

INTERRUPTS: EXCEPTIONS

- **Privilege Exception**
  - To protect the operating system of a computer from being corrupted by user programs certain instructions can be executed only while the processor is in the supervisor mode these are called privileged instructions.
  - An attempt to execute Privileged instruction in the user mode will produce a privilege exception which will cause the processor to switch from user mode to the supervisor mode and begin executing an appropriate routine in the operating system.

INTERRUPTS: EXCEPTIONS

- Difference between handling I/O interrupt-request and handling exceptions due to errors:
  - In case of I/O interrupt-request, the processor usually completes the execution of an instruction in progress before branching to the interrupt-service routine.
  - In case of exception processing however, the execution of an instruction in progress usually cannot be completed.

DMA: DIRECT MEMORY ACCESS

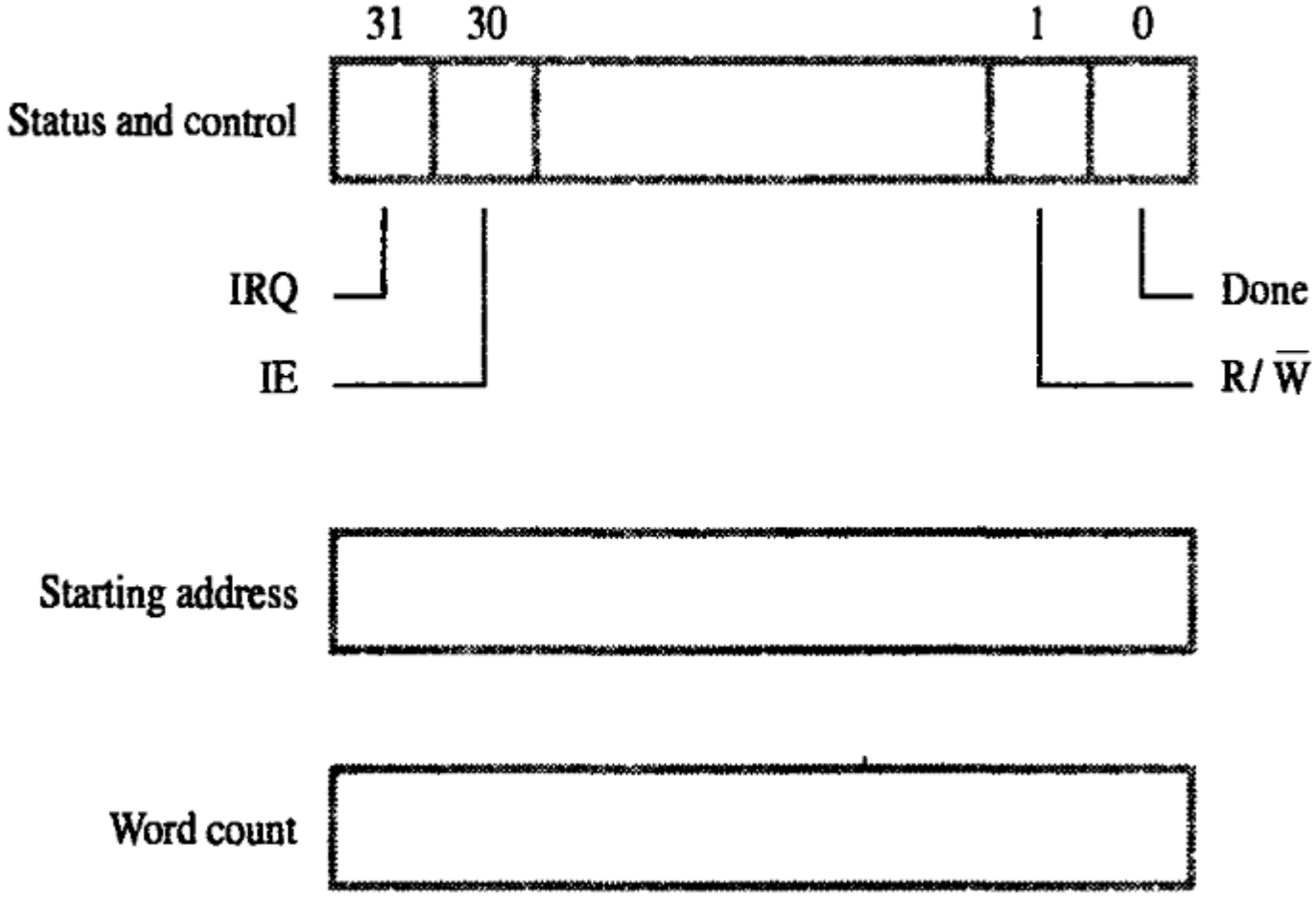
- A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory without continuous intervention by the processor this approach is called direct memory access or DMA.
- DMA transfers are performed by a control circuit that is part of the I/O device interface referred as a DMA controller.
- The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory.
- For each word transferred it provide the memory address and all the bus signals that control data transfer.
- Since it has to transfer blocks of data, DMA controller must increment the memory address for successive words and keep track of the number of transfers.

DMA: DIRECT MEMORY ACCESS

- Although DMA controller can transfer data without intervention by the processor its operation must be under the control of a program executed by the processor.
- To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block and the direction of the transfer.
- On receiving this information, the DMA controller proceeds to perform the requested operation.
- When the entire block has been transferred the controller informs the processor by raising an interrupt signal.
- While a DMA transfer is taking place the program that requested the transfer cannot continue and the processor can be used to execute another program. After the DMA transfer is completed the processor can return to the program that requested the transfer.

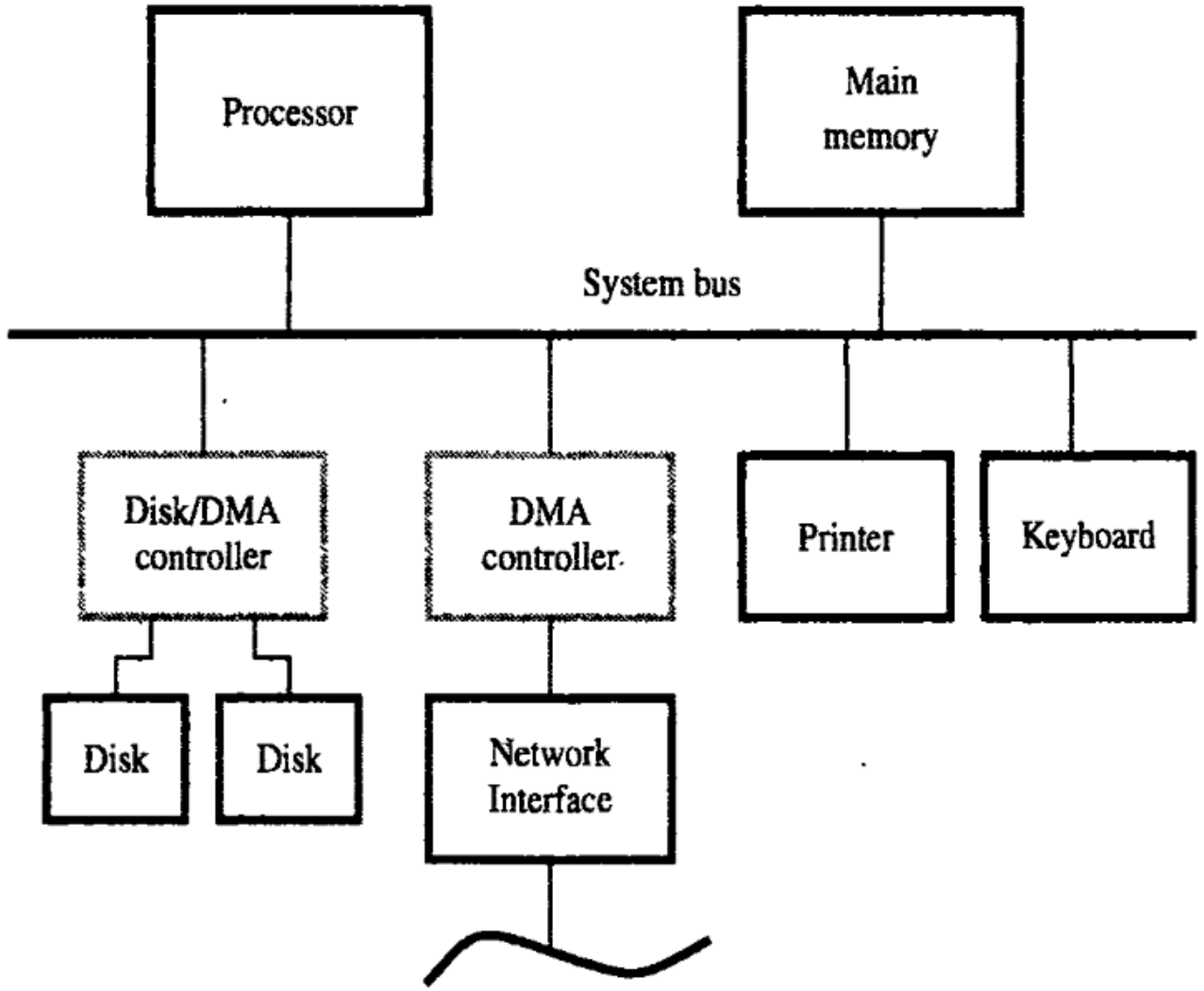
DMA: DIRECT MEMORY ACCESS

- An example of the DMA controller registered that are accessed by the processor to initiate transfer operations



DMA: DIRECT MEMORY ACCESS

- Use of DMA controllers in a computer system



DMA: DIRECT MEMORY ACCESS

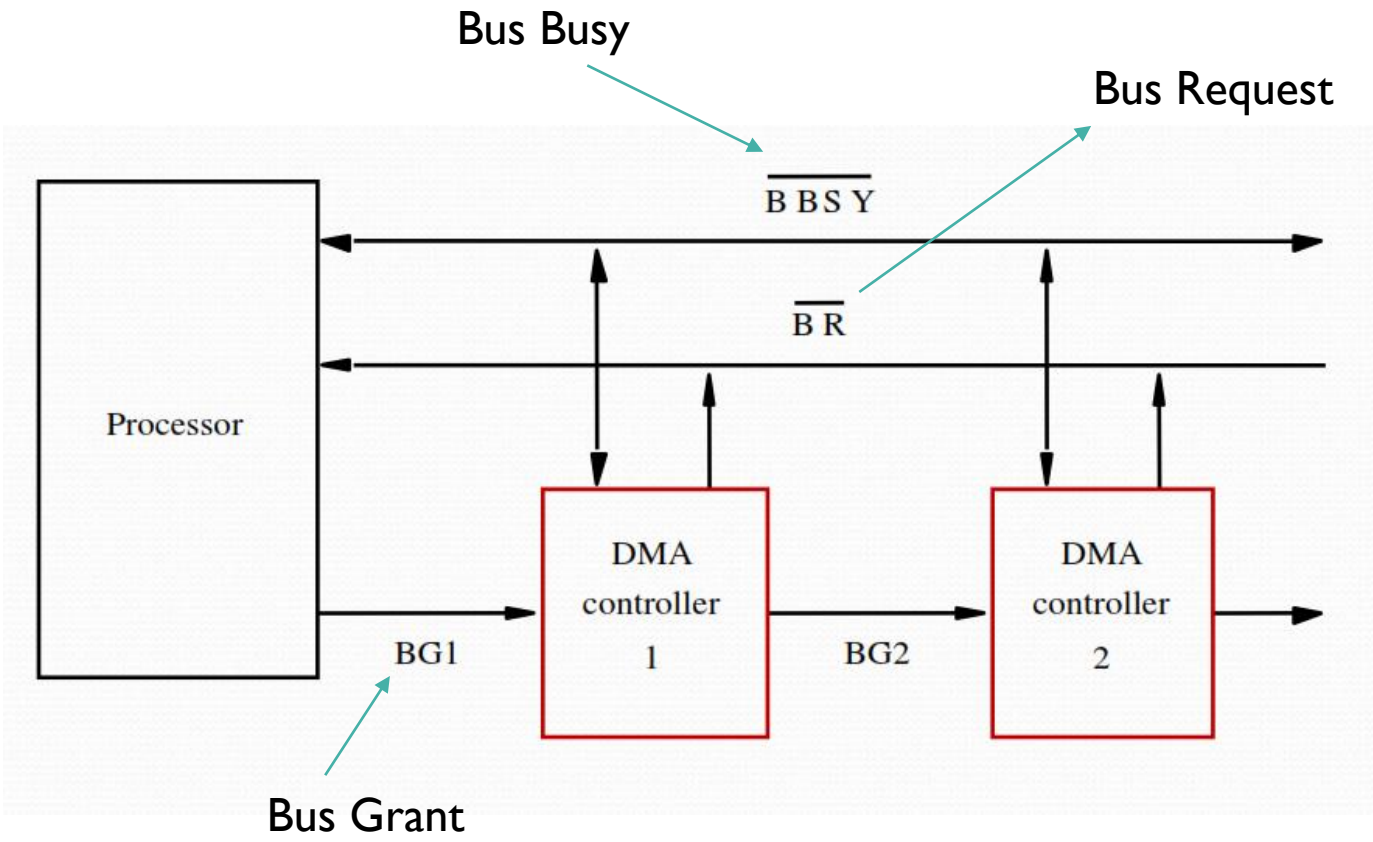
- Memory accesses by the processor and the DMA controller are intermixed.
- Request by DMA devices for using the bus are always given higher priority than the processor request.
- Among different devices top priority is given to high speed peripherals such as disk, high speed network interface or graphics display device.
- Since the processor originates most memory access cycles, the DMA controller can be said to steal memory cycle from the processor. Hence, this intermixing technique is usually called **cycle stealing**.
- Also, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption this is known as **block or burst mode**.
- Conflict may arise if both the processor and DMA controller or 2 DMA controller try to use the bus at the same time to access the main memory. To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers

DMA: BUS ARBITRATION

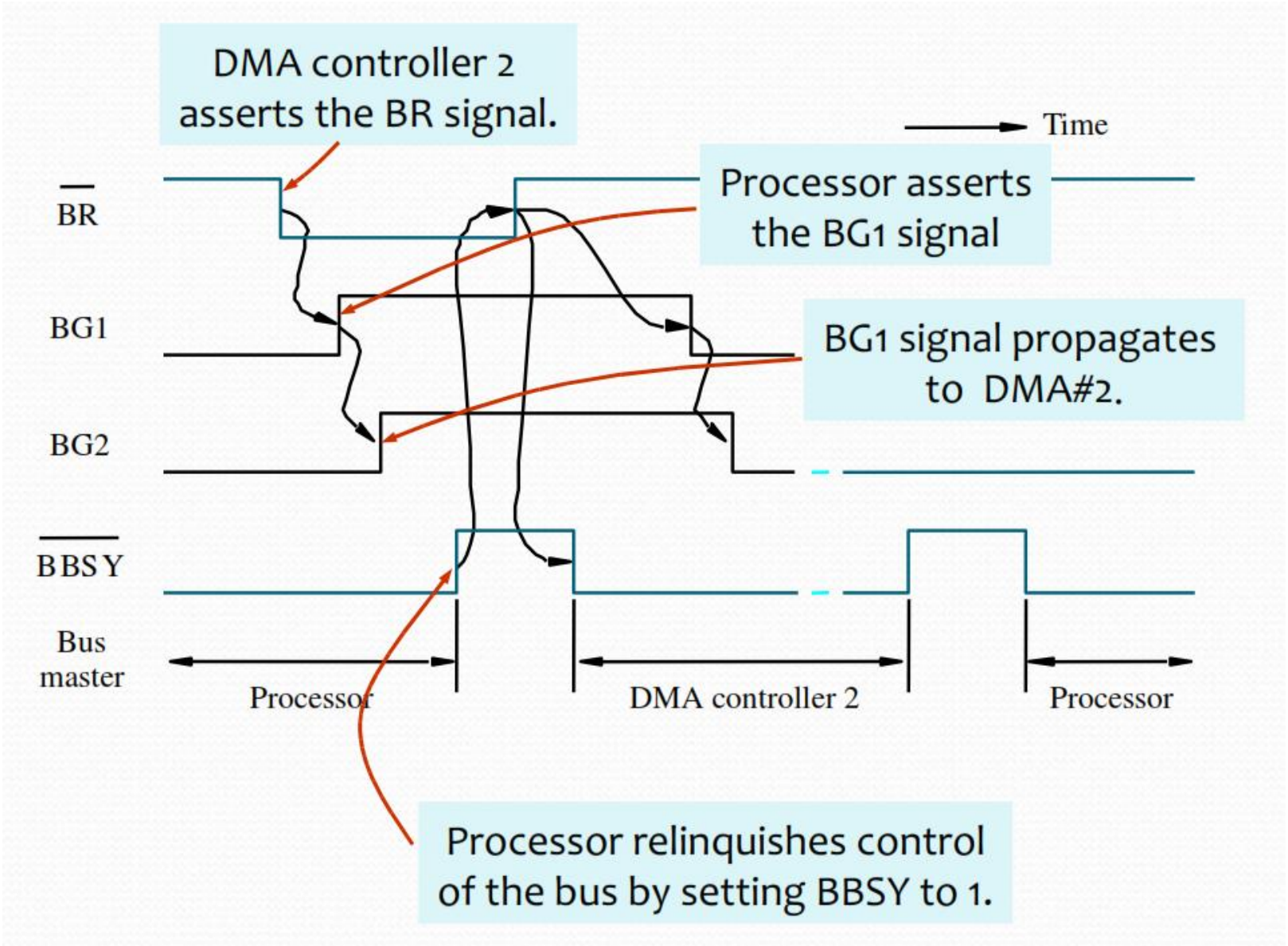
- The device that is allowed to initiate data transfers on the bus at any given time is called **bus master**.
- Bus arbitration is the process by which the next device to become bus master is selected and bus master-ship is transferred to it.
- There are 2 approaches to bus arbitration:-
  1. Centralised Arbitration
  2. Distributed Arbitration

DMA: CENTRALISED ARBITRATION

- Normally processor is the bus master
- DMA controller indicates that it needs to become the bus master by activating the bus request line
- Processor activates the bus grant signal indicating to the DMA controller that they may use the bus when it becomes free
- Grant signal is connected to all the DMA controller in Daisy chain arrangement
- Bus busy signal is activated by the bus master to prevent other devices from using the bus
- DMA controller waits for bus busy to become inactive then assumes mastership of the bus.



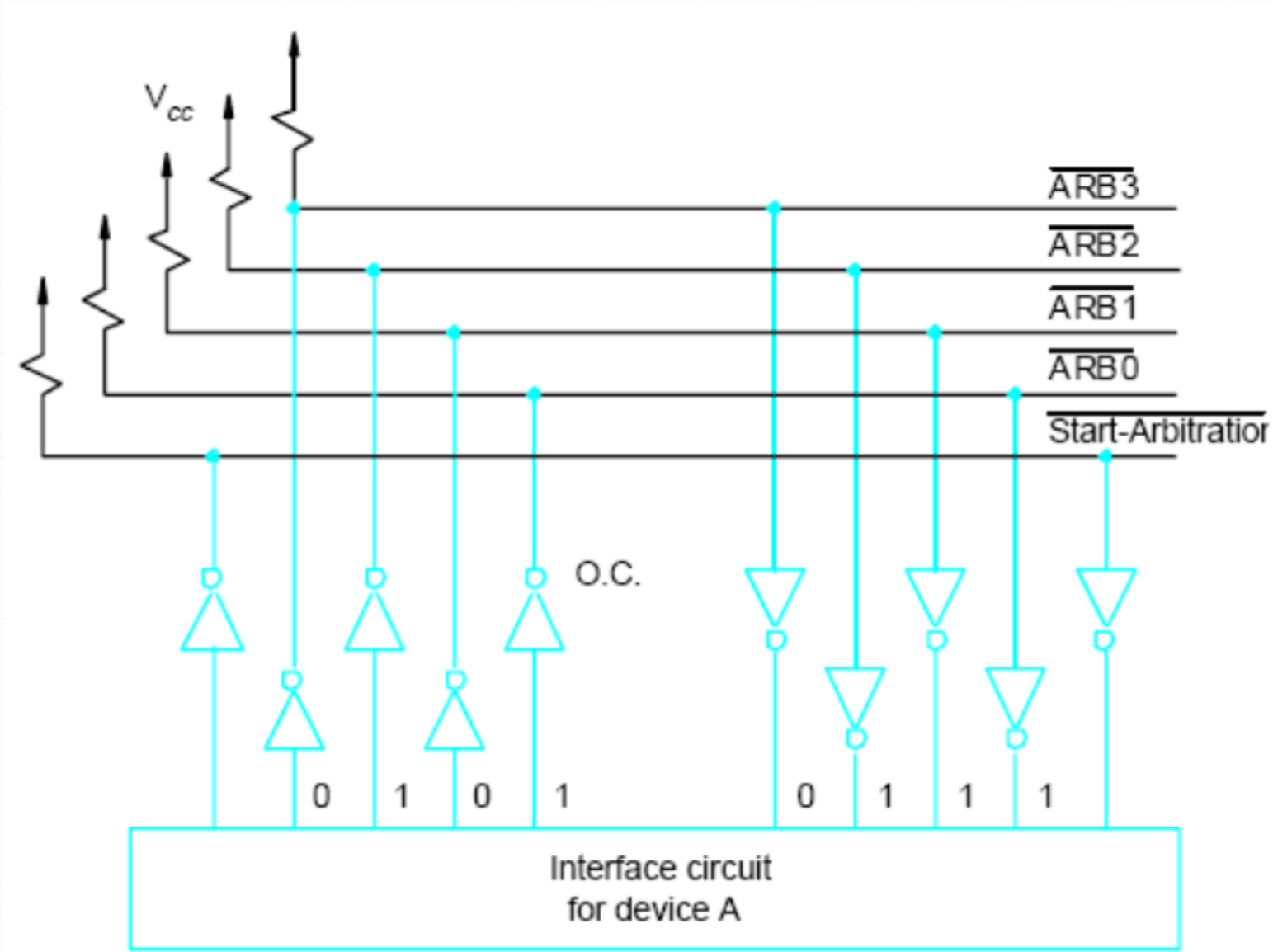
DMA: CENTRALISED ARBITRATION



Distributed arbitration

- All devices waiting to use the bus share the responsibility of carrying out the arbitration process.
  - Arbitration process does not depend on a central arbiter and hence distributed arbitration has higher reliability.
- Each device is assigned a 4-bit ID number.
- All the devices are connected using 5 lines, 4 arbitration lines to transmit the ID, and one line for the Start-Arbitration signal.
- To request the bus a device:
  - Asserts the Start-Arbitration signal.
  - Places its 4-bit ID number on the arbitration lines.
- The pattern that appears on the arbitration lines is the logical-OR of all the 4-bit device IDs placed on the arbitration lines.

Distributed arbitration



Distributed arbitration(Contd.,)

- *Arbitration process:*
  - *Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB.*
  - *If it detects a difference, it transmits os on the arbitration lines for that and all lower bit positions.*
  - *The pattern that appears on the arbitration lines is the logical-OR of all the 4-bit device IDs placed on the arbitration lines.*

Distributed arbitration(Contd.,)

- *Device A has the ID 5 and wants to request the bus:*
  - *Transmits the pattern 0101 on the arbitration lines.*
- *Device B has the ID 6 and wants to request the bus:*
  - *Transmits the pattern 0110 on the arbitration lines.*
- *Pattern that appears on the arbitration lines is the logical OR of the patterns:*
  - *Pattern 0111 appears on the arbitration lines.*

Arbitration process:

- *Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB.*
- *If it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions.*
- *Device A compares its ID 5 with a pattern 0101 to pattern 0111.*
- *It detects a difference at bit position 0, as a result, it transmits a pattern 0100 on the arbitration lines.*
- *The pattern that appears on the arbitration lines is the logical-OR of 0100 and 0110, which is 0110.*
- *This pattern is the same as the device ID of B, and hence B has won the arbitration.*

Buses

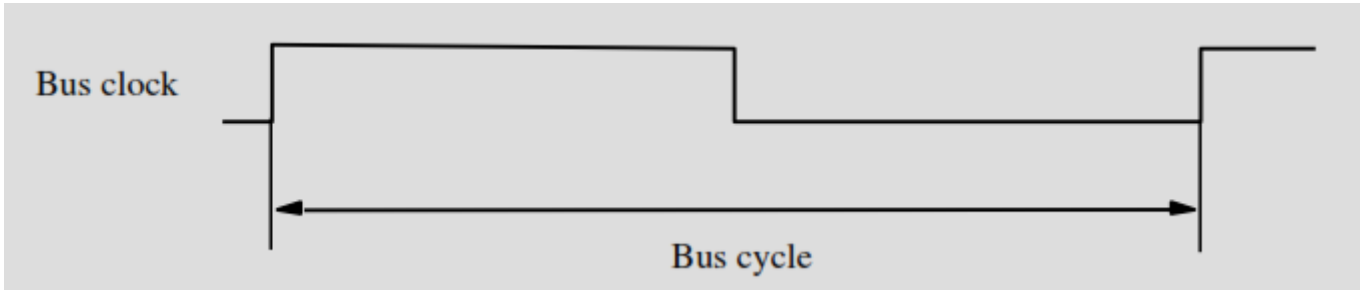
- Processor, main memory, and I/O devices are interconnected by means of a bus.
- Bus provides a communication path for the transfer of data.
  - Bus also includes lines to support interrupts and arbitration.
- A bus protocol is the set of rules that govern the behavior of various devices connected to the bus, as to when to place information on the bus, when to assert control signals, etc.

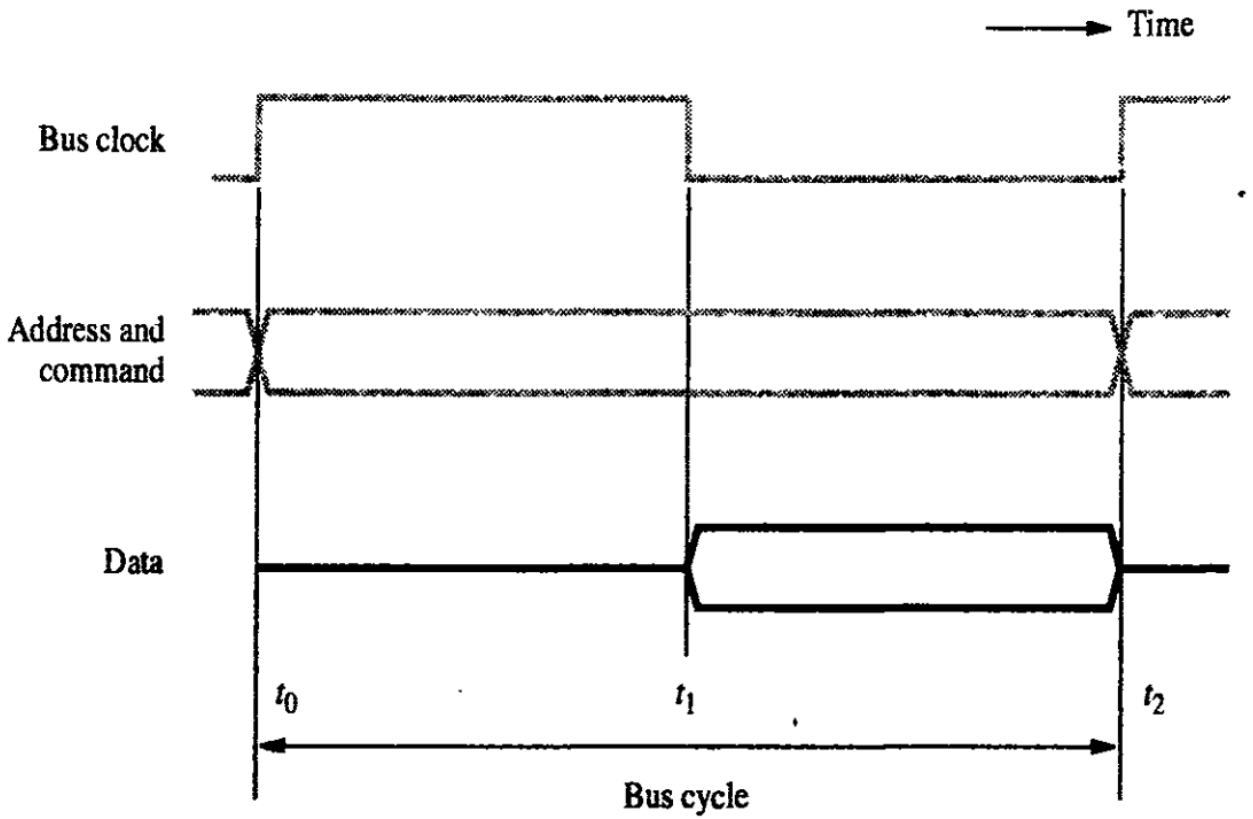
Buses (contd..)

- Bus lines may be grouped into three types:
  - Data
  - Address
  - Control
- Control signals specify:
  - Whether it is a read or a write operation.
  - Required size of the data, when several operand sizes (byte, word, long word) are possible.
  - Timing information to indicate when the processor and I/O devices may place data or receive data from the bus.
- Schemes for timing of data transfers over a bus can be classified into:
  - Synchronous,
  - Asynchronous.

Buses (contd..)

- In any data transfer operation one device plays the role of a **master.**
- **T**his is the device that initiate data transfer by issuing read or write command on the bus hence it may be called an **initiator.**
- **N**ormally the processor act as the master but other devices with DMA capability may also become bus masters.
- The device addressed by the master is referred to as **slave or target**.

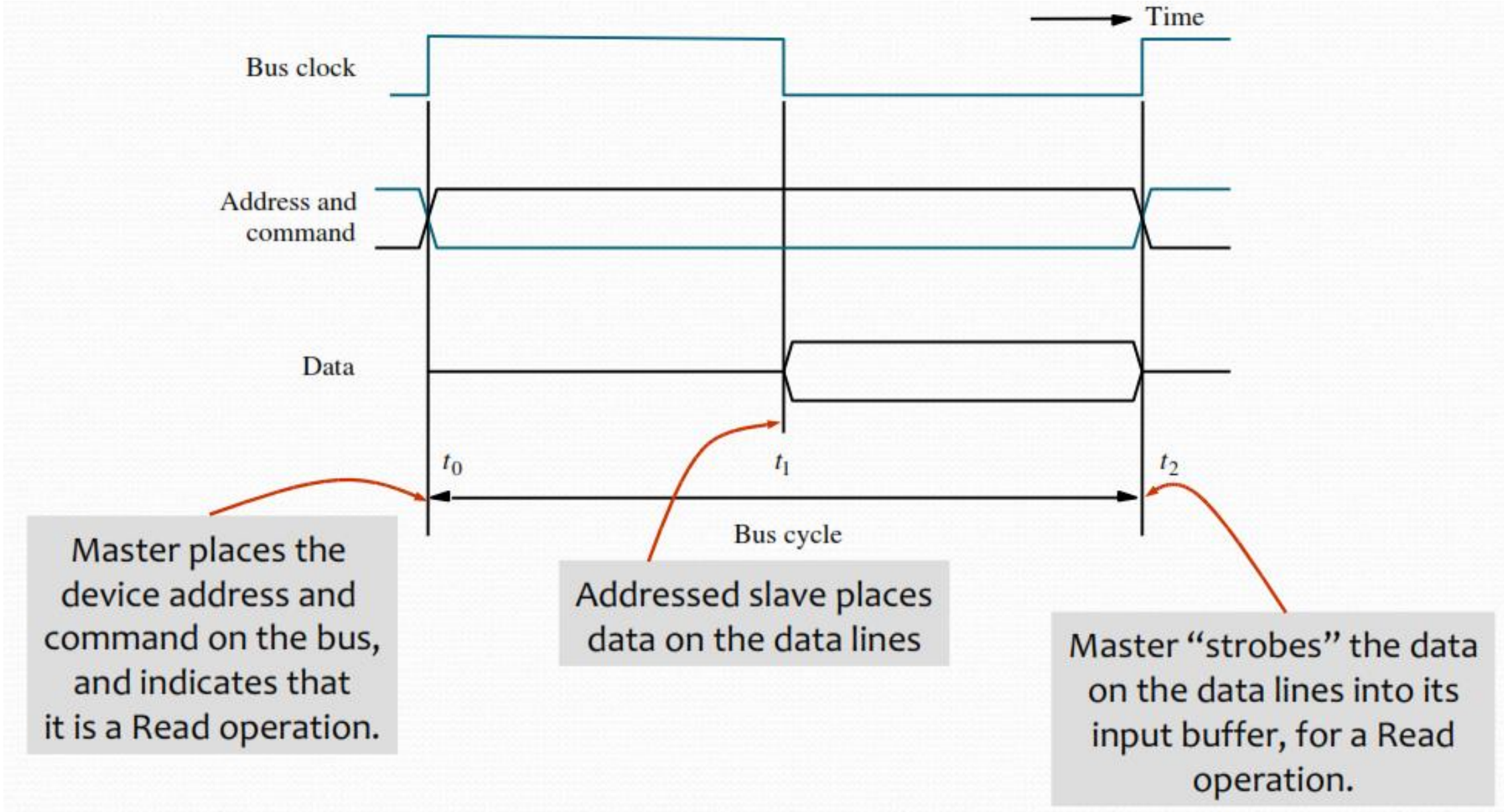




Timing of an input transfer on a synchronous bus.

SYNCHRONOUS BUS

- In synchronous bus all devices drive timing information from a common clock line
- Equally spaced pulses on this line define equal time intervals
- Each of these interval constitute **BUS CYCLE** during which one data transfer can take place


Time
Bus clock
Address and command
Data
t_0
t_1
t_2
Bus cycle
Master places the device address and command on the bus, and indicates that it is a Read operation.
Addressed slave places data on the data lines
Master "strobes" the data on the data lines into its input buffer, for a Read operation.

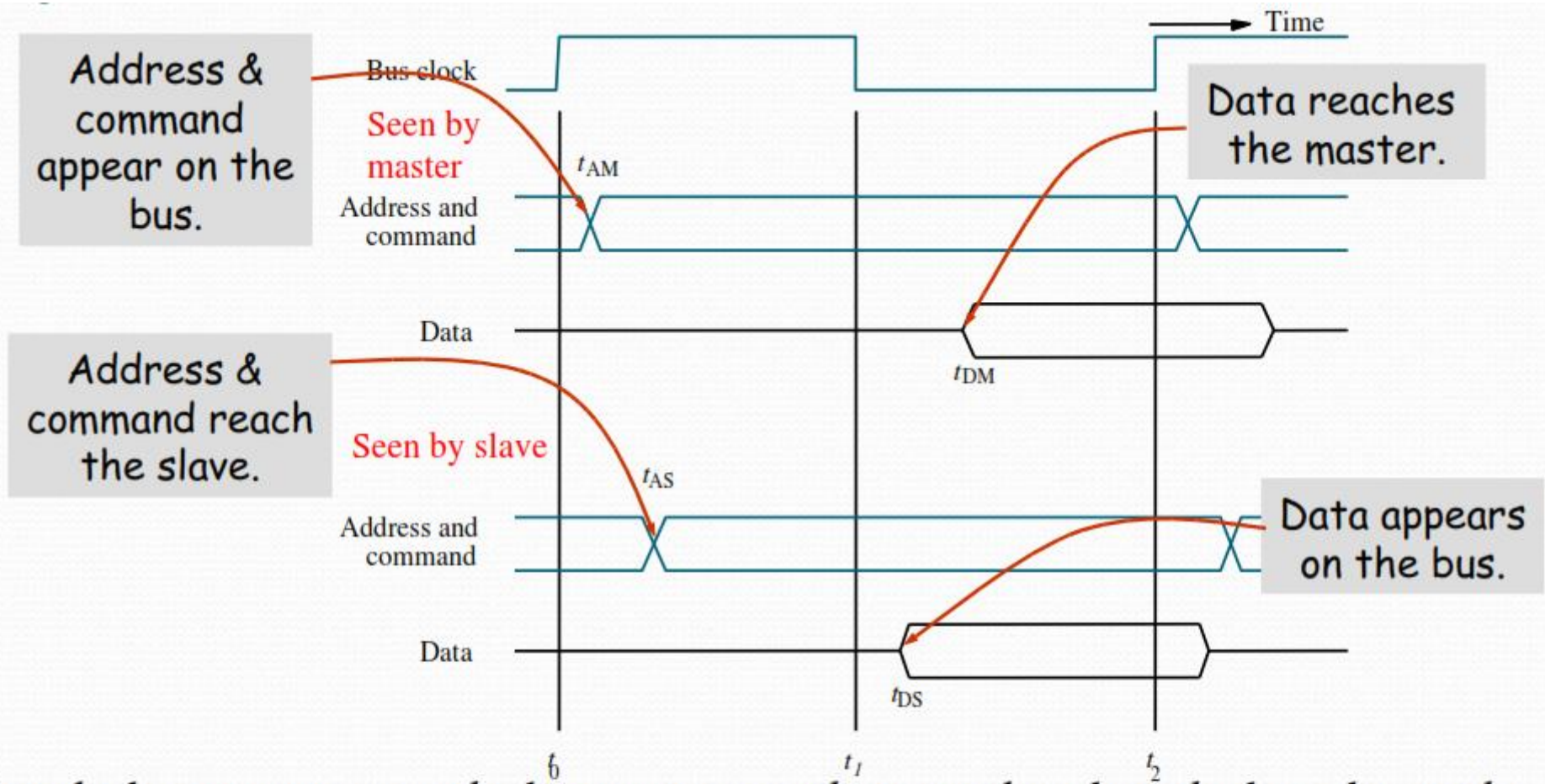
IDEAL CASE

Synchronous bus (contd..)

- Once the master places the device address and command on the bus, it takes time for this information to propagate to the devices:
  - This time depends on the physical and electrical characteristics of the bus.
- Also, all the devices have to be given enough time to decode the address and control signals, so that the addressed slave can place data on the bus.
- Width of the pulse t_1 - t_0 depends on:
  - Maximum propagation delay between two devices connected to the bus.
  - Time taken by all the devices to decode the address and control signals, so that the addressed slave can respond at time t_1.

Synchronous bus (contd..)

- At the end of the clock cycle, at time t_2, the master strobes the data on the data lines into its input buffer if it's a Read operation.
  - "Strobe" means to capture the values of the data and store them into a buffer.
- When data are to be loaded into a storage buffer register, the data should be available for a period longer than the setup time of the device.
- Width of the pulse t_2 - t_1 should be longer than:
  - Maximum propagation time of the bus plus
  - Set up time of the input buffer register of the master.



- *Signals do not appear on the bus as soon as they are placed on the bus, due to the propagation delay in the interface circuits.*
- *Signals reach the devices after a propagation delay which depends on the characteristics of the bus.*
- *Data must remain on the bus for some time after t_2 equal to the hold time of the buffer.*

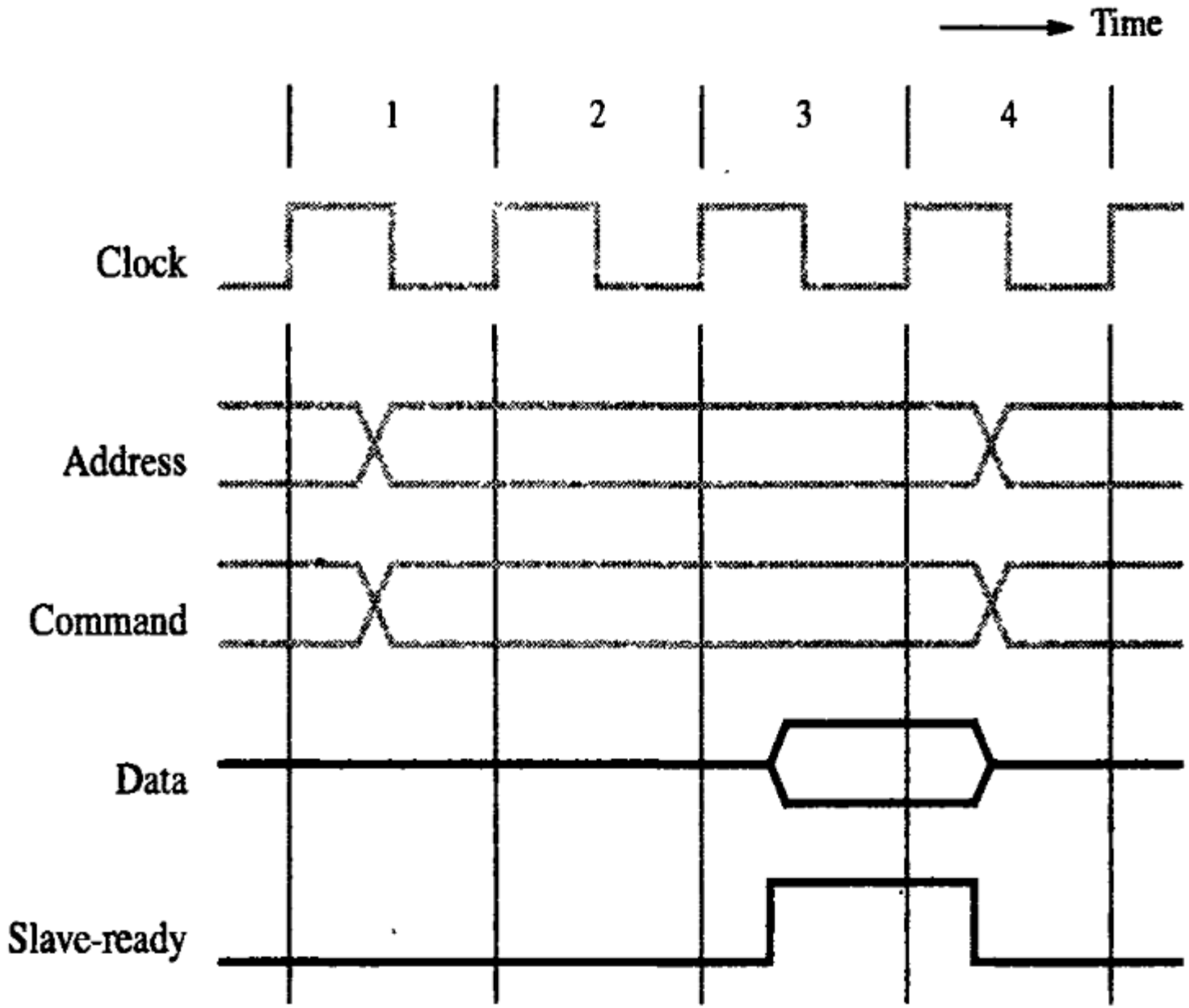
Synchronous bus (contd..)

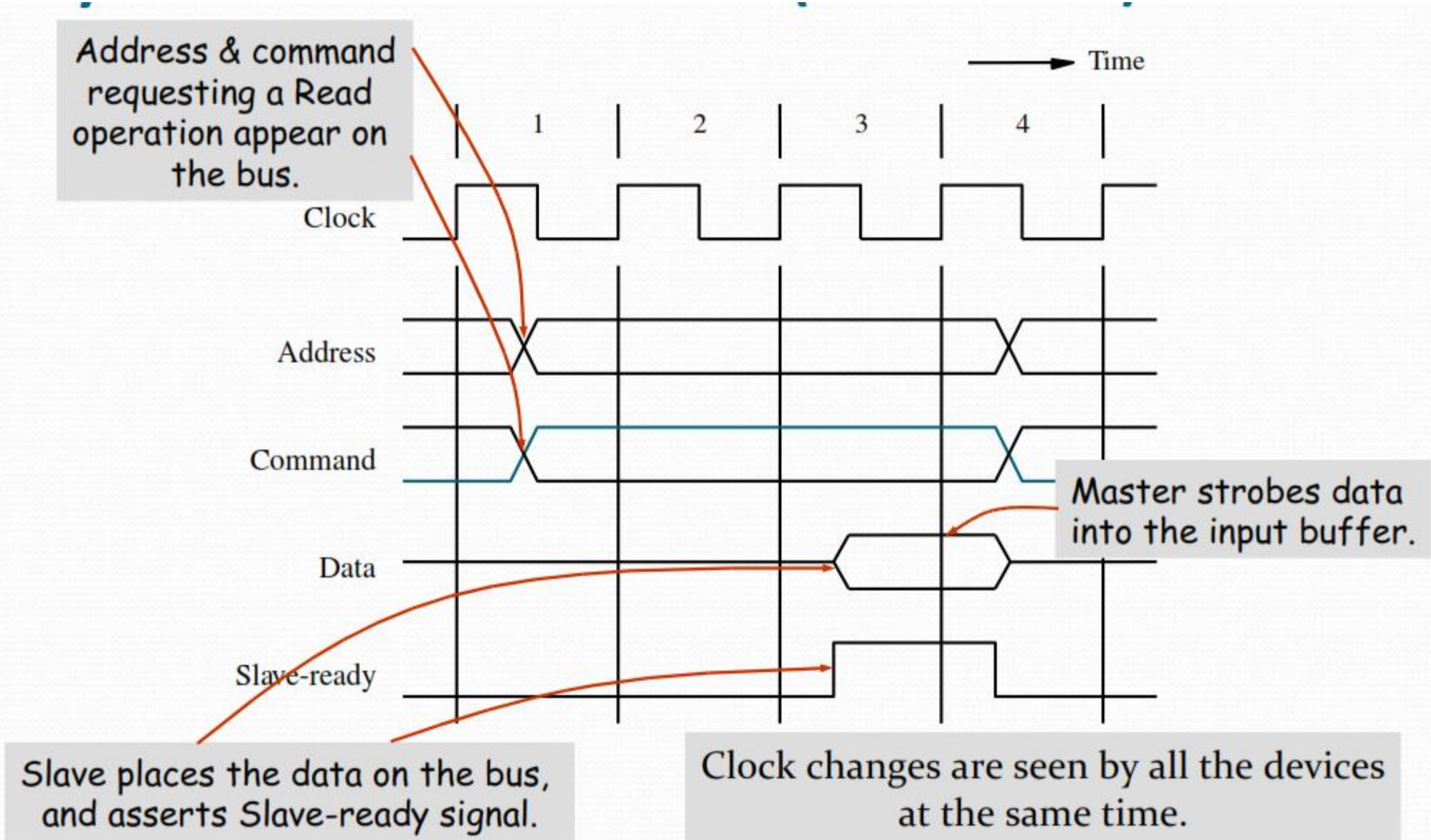
The master sends the address and command signals on the rising edge at the beginning of clock period 1 (t_0). However, these signals do not actually appear on the bus until t_{AM}, largely due to the delay in the bus driver circuit. A while later, at t_{AS}, the signals reach the slave. The slave decodes the address and at t_1 sends the requested data. Here again, the data signals do not appear on the bus until t_{DS}. They travel toward the master and arrive at t_{DM}. At t_2, the master loads the data into its input buffer. Hence the period $t_2 - t_{DM}$ is the setup time for the master's input buffer. The data must continue to be valid after t_2 for a period equal to the hold time of that buffer.

MULTIPLE BUS CYCLES IN SYNCHRONOUS BUSES

- The scheme using one clock cycle for data transfer is simple in design but it has limitations.
- Since transfer has to be completed within one clock cycle, the clock period t_2-t_0 must be chosen to accommodate the longest delay on the bus due to slowest device interface.
- This will force all devices to operate at the speed of the slowest device.
- Also in this scheme processor has no way of determining whether the address device has actually responded. It simply assumes that at t_2 the output data has been received to/from the I/O device. If the device does not respond error will not be detected.
- To overcome this limitation most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognised its address and that it is ready to participate in the data operation.
- To simplify this process a high frequency clock signal is used such that a complete data transfer cycle would require several clock cycles (**MULTIPLE BUS CYCLE**)

An input transfer using multiple clock cycles.



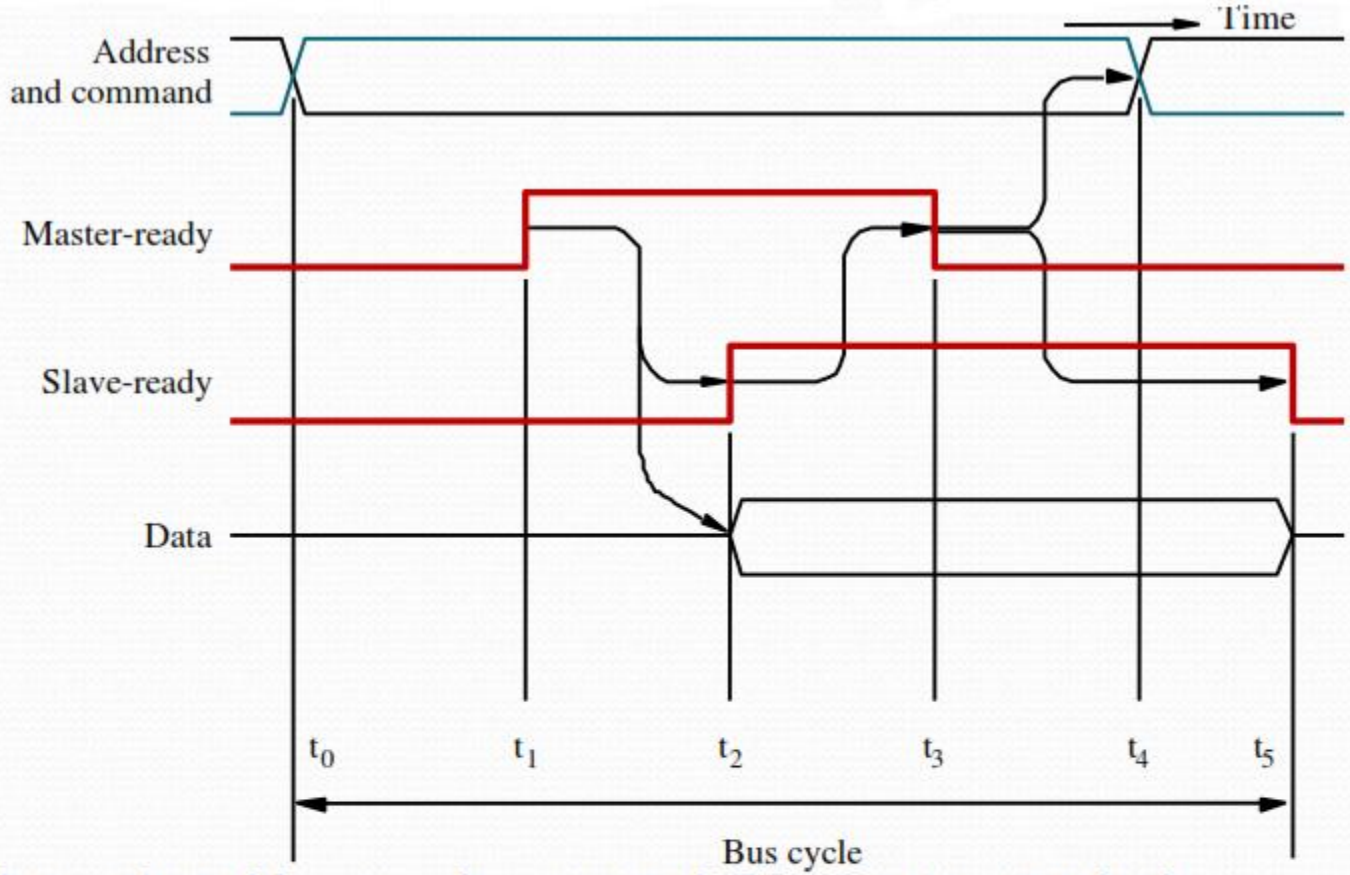

Address & command requesting a Read operation appear on the bus.
Time
1
2
3
4
Clock
Address
Command
Data
Slave-ready
Master strobes data into the input buffer.
Slave places the data on the bus, and asserts Slave-ready signal.
Clock changes are seen by all the devices at the same time.

Asynchronous bus

- Data transfers on the bus is controlled by a handshake between the master and the slave.
- Common clock in the synchronous bus case is replaced by two timing control lines:
  - Master-ready,
  - Slave-ready.
- Master-ready signal is asserted by the master to indicate to the slave that it is ready to participate in a data transfer.
- Slave-ready signal is asserted by the slave in response to the master-ready from the master, and it indicates to the master that the slave is ready to participate in a data transfer.

Asynchronous bus (contd..)

- Data transfer using the handshake protocol:
  - Master places the address and command information on the bus.
  - Asserts the Master-ready signal to indicate to the slaves that the address and command information has been placed on the bus.
  - All devices on the bus decode the address.
  - Address slave performs the required operation, and informs the processor it has done so by asserting the Slave-ready signal.
  - Master removes all the signals from the bus, once Slave-ready is asserted.
  - If the operation is a Read operation, Master also strobes the data into its input buffer.



t_o - Master places the address and command information on the bus.

t_1 - Master asserts the Master-ready signal. Master-ready signal is asserted at t_1 instead of t_o

t_2 - Addressed slave places the data on the bus and asserts the Slave-ready signal.

t_3 - Slave-ready signal arrives at the master.

t_4 - Master removes the address and command information.

t_5 - Slave receives the transition of the Master-ready signal from 1 to 0. It removes the data and the Slave-ready signal from the bus.

Asynchronous vs. Synchronous bus

- Advantages of asynchronous bus:
  - Eliminates the need for synchronization between the sender and the receiver.
  - Can accommodate varying delays automatically, using the Slave-ready signal.
- Disadvantages of asynchronous bus:
  - Data transfer rate with full handshake is limited by two-round trip delays.
  - Data transfers using a synchronous bus involves only one round trip delay, and hence a synchronous bus can achieve faster rates.

Interface circuits

- I/O interface consists of the circuitry required to connect an I/O device to a computer bus.
- Side of the interface which connects to the computer has bus signals for:
  - Address,
  - Data
  - Control
- Side of the interface which connects to the I/O device has:
  - Datapath and associated controls to transfer data between the interface and the I/O device.
  - This side is called as a "port".
- Ports can be classified into two:
  - Parallel port,
  - Serial port.

Interface circuits (contd..)

- Parallel port transfers data in the form of a number of bits, normally 8 or 16 to or from the device.
- Serial port transfers and receives data one bit at a time.
- Processor communicates with the bus in the same way, whether it is a parallel port or a serial port.
  - Conversion from the parallel to serial and vice versa takes place inside the interface circuit.

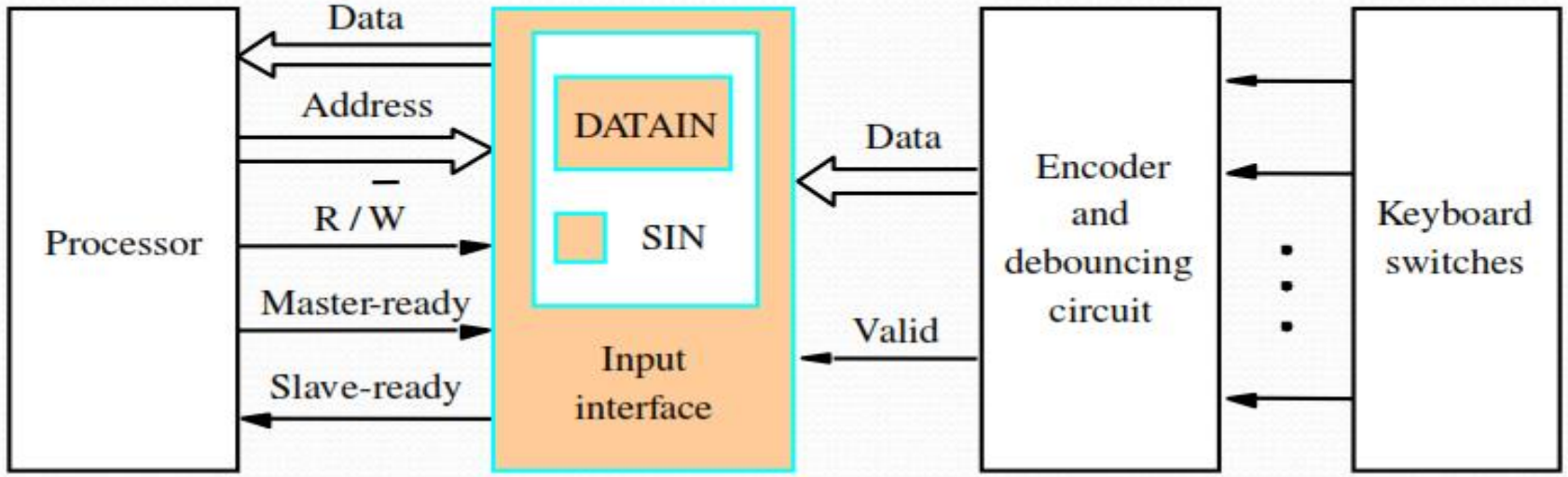
Interface circuits (contd..)

- In case of a parallel port the connection between the device and the computer uses a multi pin connector and a cable with as many wires typically arranged in a flat configuration.
- The circuits at either end are relatively simple as there is no need to convert between parallel and serial formats.
- This arrangement is suitable for devices that are physically close to the computer.
- For long distances the problem of timing limits the data rate.
- The serial format is much more convenient and cost effective where longer cables are needed

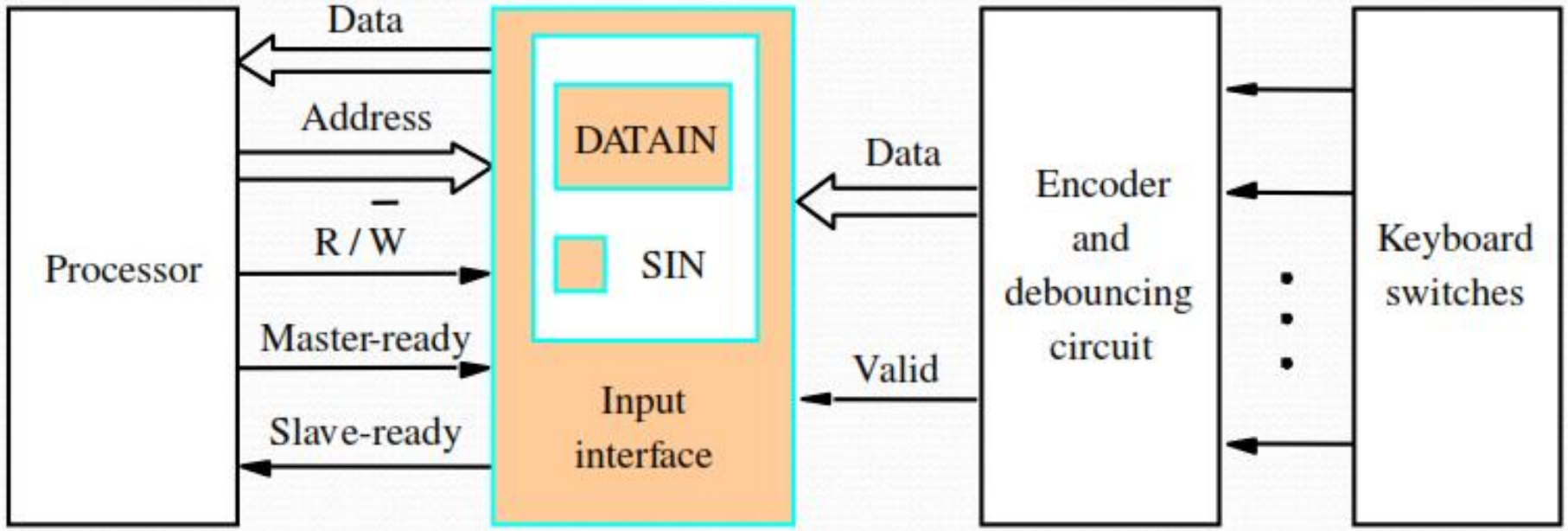
FUNCTIONS OF I/O INTERFACE CIRCUIT

1. Provides a storage buffer for at least one word of data (or one byte, in the case of byte-oriented devices)
2. Contains status flags that can be accessed by the processor to determine whether the buffer is full (for input) or empty (for output)
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control scheme
5. Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

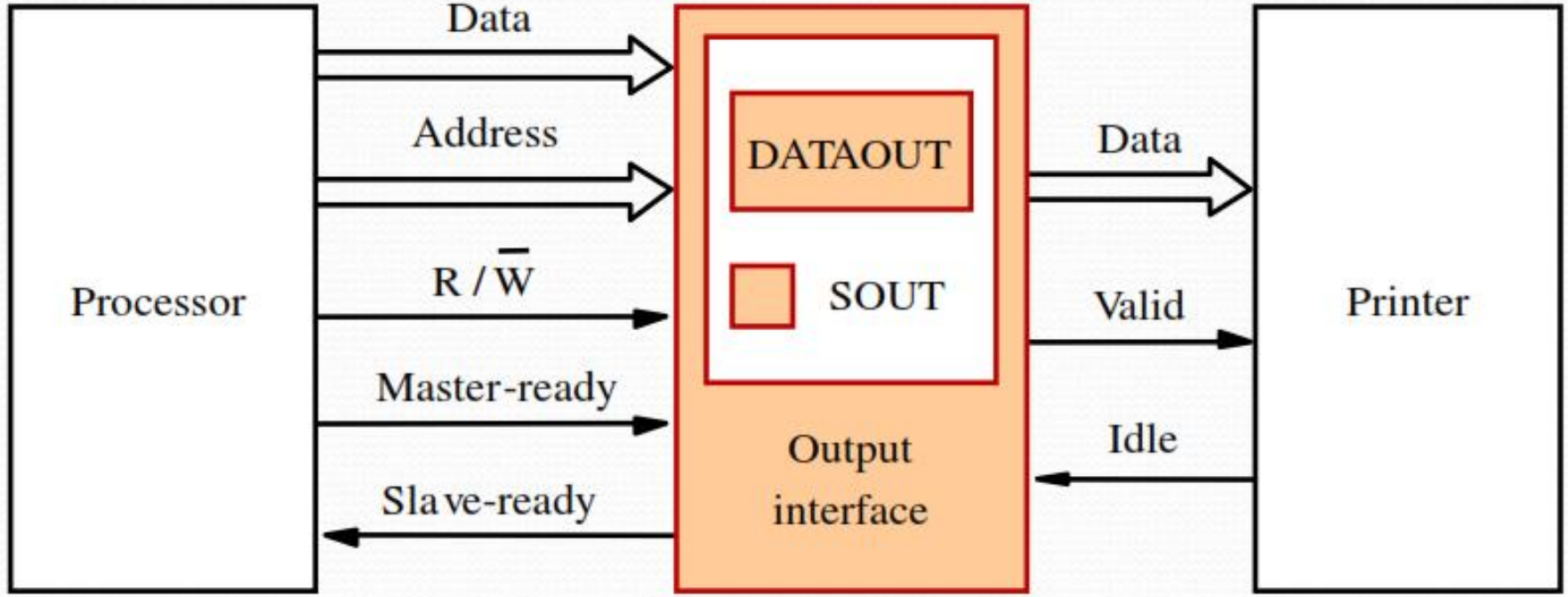
Parallel port



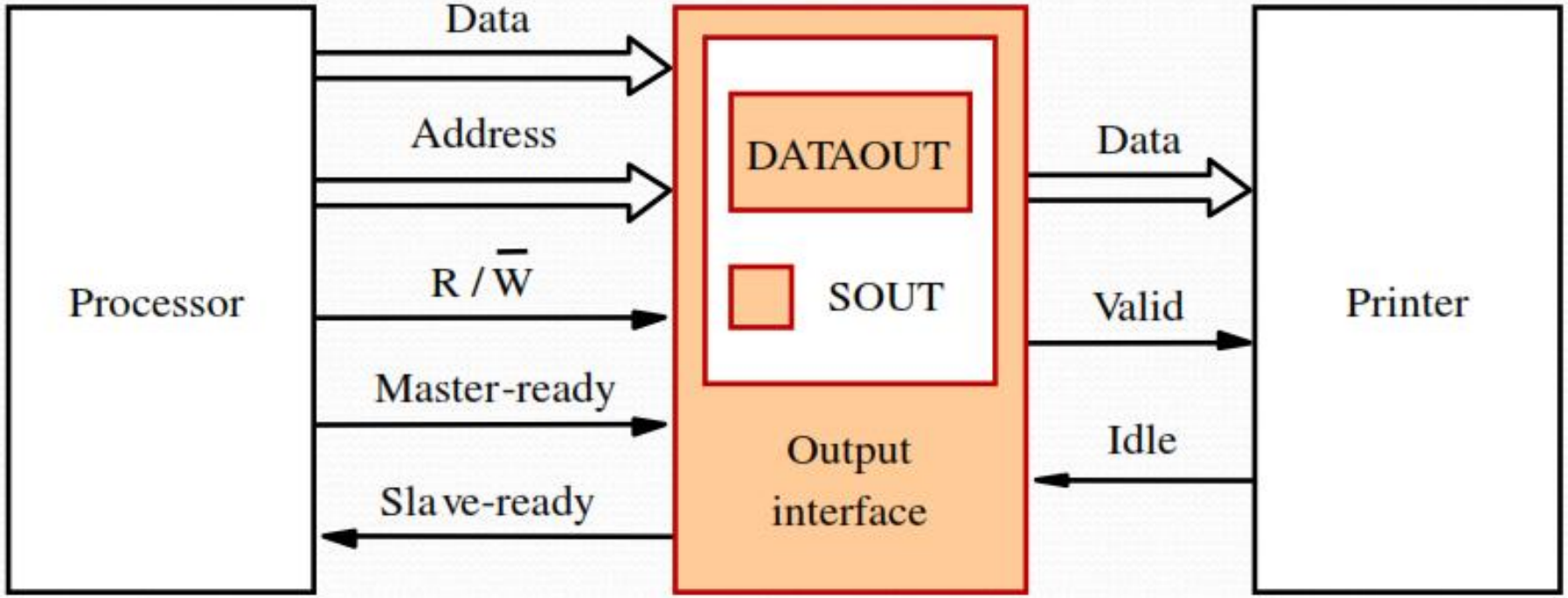
- *Keyboard is connected to a processor using a parallel port.*
- *Processor is 32-bits and uses memory-mapped I/O and the asynchronous bus protocol.*
- *On the processor side of the interface we have:*
  - *Data lines.*
  - *Address lines*
  - *Control or R/W line.*
  - *Master-ready signal and*
  - *Slave-ready signal.*



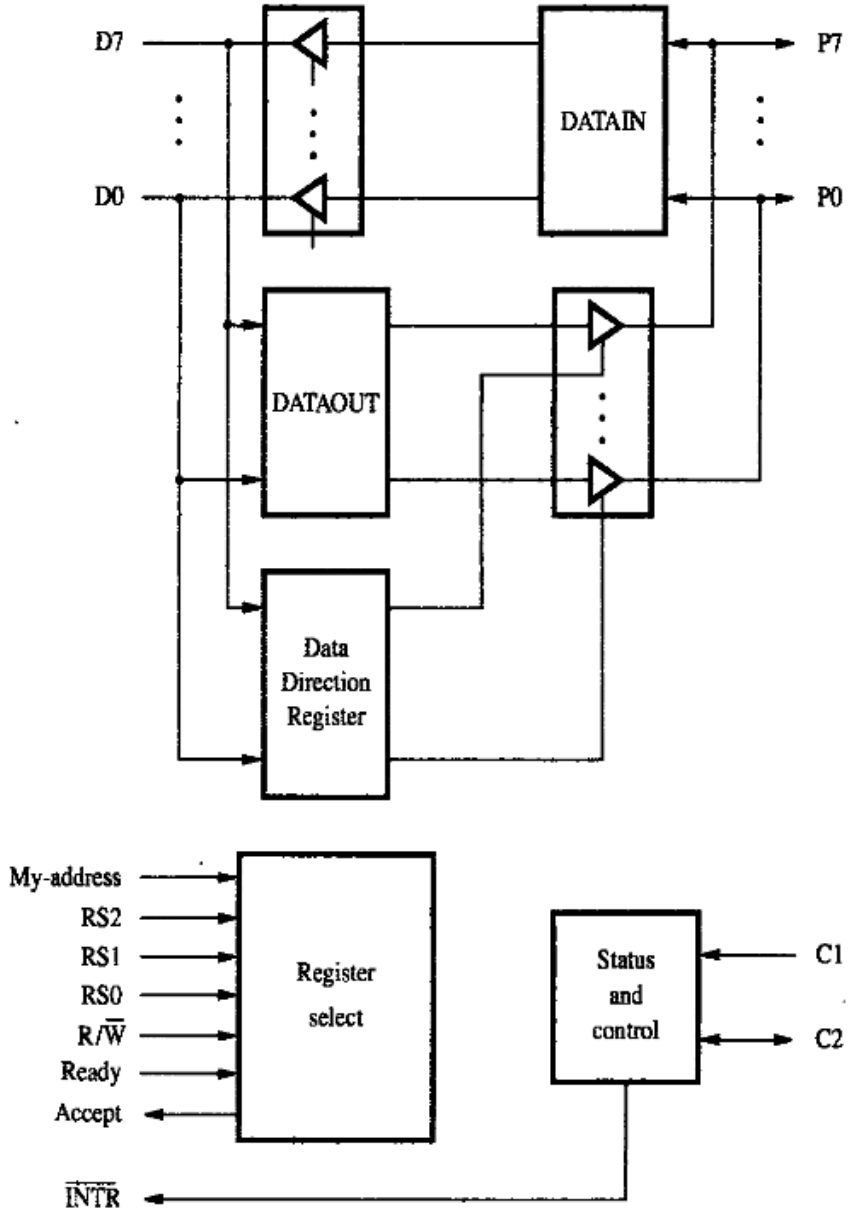
- *On the keyboard side of the interface:*
  - *Encoder circuit which generates a code for the key pressed.*
  - *Debouncing circuit which eliminates the effect of a key bounce (a single key stroke may appear as multiple events to a processor).*
  - *Data lines contain the code for the key.*
  - *Valid line changes from 0 to 1 when the key is pressed. This causes the code to be loaded into DATAIN and SIN to be set to 1.*



- *Printer is connected to a processor using a parallel port.*
- *Processor is 32 bits, uses memory-mapped I/O and asynchronous bus protocol.*
- *On the processor side:*
  - *Data lines.*
  - *Address lines*
  - *Control or R/W line.*
  - *Master-ready signal and*
  - *Slave-ready signal.*



- *On the printer side:*
  - *Idle signal line which the printer asserts when it is ready to accept a character. This causes the SOUT flag to be set to 1.*
  - *Processor places a new character into a DATAOUT register.*
  - *Valid signal, asserted by the interface circuit when it places a new character on the data lines.*



A general 8-bit parallel interface.