UNIT-4

Peripheral Devices and Their Characteristics

I/o Device Interface:

Introduction: Input-output interface ***provides a method for transferring information between internal storage and external I/0 devices.*** **Peripherals**(I/O Devices) connected to a computer ***need special communication links for interfacing them with the central processing unit.*** The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral.

The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a ***conversion of signal values*** may be required.

2. The ***data transfer rate*** of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

3. ***Data codes and formats*** in peripherals ***differ*** from the word format in the CPU and memory.

4. The ***operating modes of peripherals*** are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include ***special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers.*** These components are called ***interface units*** because they interface ***between the processor bus and the peripheral device***. In addition, ***each device may have its own controller*** that supervises the operations of the particular mechanism in the peripheral.

I/O Bus and Interface Modules:

A typical communication link between the processor and several peripherals is shown below:

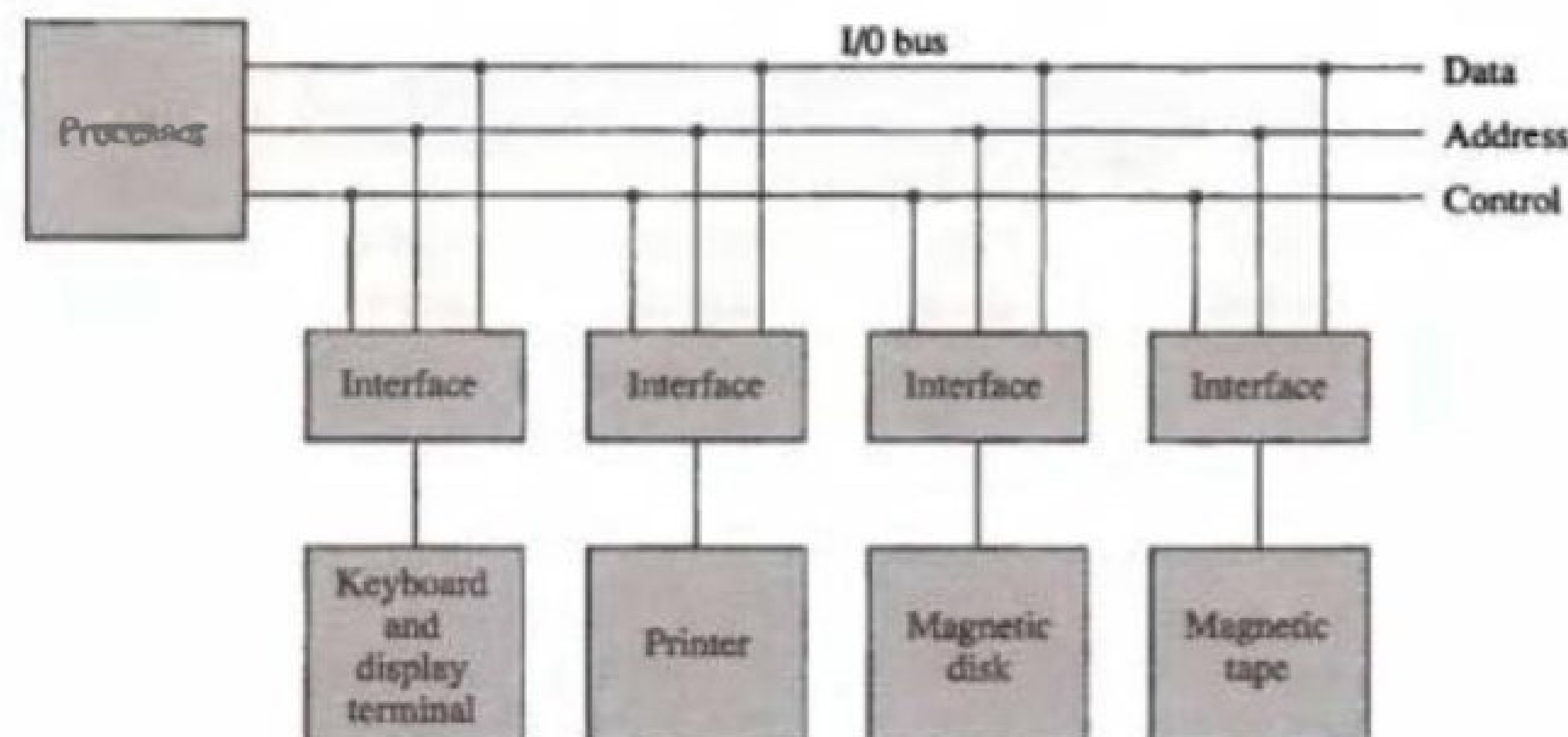


Fig: Connection of I/O bus to input-output devices.

Scanned with OKEN Scanner

The I/0 bus consists of data lines, address lines, and control lines. ***Each peripheral device has associated with it an interface unit.***

Each ***interface decodes the address and control received from the I/O bus***, interprets them for the peripheral, and ***provides signals for the peripheral controller***. It also ***synchronizes the data flow and supervises the transfer*** between peripheral and processor.

Each ***peripheral has its own controller that*** operates the particular electromechanical device. For example, the ***printer controller controls the paper motion, the print timing, and the selection of printing characters***. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. ***To communicate with a particular device***, the processor places a device address on the address lines. ***Each interface attached to the I/0 bus contains an address decoder that monitors the address lines***. ***When the interface detects its own address, it activates the path between the bus lines and the device that it controls***. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the ***processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute*** it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. There are **four types of commands that an interface may receive**. They are classified as control, status, data output, and data input.

- A ***control command*** is issued to ***activate the peripheral*** and to inform it what to do.
- A ***status command*** is used to test various status conditions in the interface and the peripheral. For Example, the computer may wish to check the status of the peripheral before a transfer is initiated.
- A ***data output command*** causes the interface to respond by transferring ***data from the bus into one of its registers.***
- The ***data input command*** is the opposite of the data output. In this case the ***interface receives an item of data from the peripheral and places it in its buffer register.*** The processor checks if data are available by means of a status command and then issues a data input command. ***The interface places the data on the data lines, where they are accepted by the processor.***

There are ***three ways that computer buses can be used to communicate with memory and I/O:***

1. Use two separate buses, one for memory and the other for I/O. (This method uses a separate I/O Processor alongside CPU to provide an independent pathway for the transfer of information between external devices and internal memory.)

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

Isolated I/O: Many computers use one common bus to transfer information between memory or I/0 and the CPU. The distinction between a memory transfer and I/0 transfer is made through ***separate read and write lines***.

Scanned with OKEN Scanner

The ***CPU specifies whether the address on the address lines*** is for a memory word or for an interface register by ***enabling one of two possible read or write lines***. ***The I/0 read and I/0 write control lines are enabled during an I/0 transfer. The memory read and memory write control lines are enabled during a memory transfer***. This configuration isolates all I/0 interface addresses from the addresses assigned to memory and is referred to as the isolated I/0 method for assigning addresses in a common bus. In the isolated I/0 configuration, the CPU has distinct ***input and output instructions***, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/0 read (for input) or I/0 write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word.

Memory-mapped I/O: **Memory mapped I/O** uses the same address space for both memory and I/O. This is the case in computers that ***employ only one set of read and write signals and do not distinguish between memory and I/O addresses***. This configuration is referred to as memory-mapped I/O. In a memory-mapped l/O organization ***there are no specific input or output instructions.*** The ***CPU can manipulate l/O data residing in interface registers with the same instructions that are used to manipulate memory words***. Each interface is organized as a set of registers that respond to read and write requests in the normal address space.

Computers with memory-mapped l/O can ***use memory-type instructions to access l/0 data***. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The ***advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from l/O registers.*** In a typical computer, there are more memory-reference instructions than l/O instructions. With memory-mapped l/O all instructions that refer to memory are also available for l/O.

DATA TRANSFER MODES:

The transfer of data between two units may be done in parallel or serial. In ***parallel data transmission, each bit of the message has its own path*** and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. ***This method requires the use of one pair of conductors or one conductor and a common ground.*** Parallel transmission is faster but requires many wires. It is **used for short distances and where speed is important**. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be **synchronous or asynchronous**.

Synchronous Data Transfer: In synchronous transmission, the two units (***Sending Unit and Receiving unit) share a common clock frequency*** and bits are transmitted continuously at the rate dictated by the clock pulses. In long distant serial transmission, each unit is driven by a separate clock of the same frequency. *Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other*.

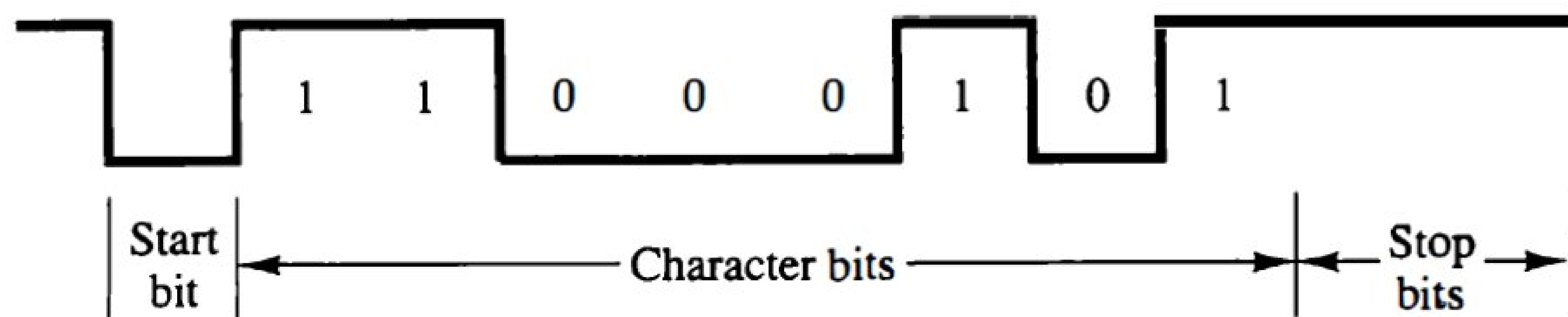
Scanned with OKEN Scanner

In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, ***where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other.***

Eg:Any two units of a digital system are designed independently, such as CPU and I/O interface. If the registers in the I/O interface share a common clock with CPU registers, then transfer between the two units is said to be synchronous.

Asynchronous Serial Transfer: A serial asynchronous data transmission technique used in many interactive terminals ***employs special bits that are inserted at both ends of the character code***. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the ***start bit, is always a 0 and is used to indicate the beginning of a character.*** The last bit called the stop bit is always a 1.

An example of this format is shown below:



A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1 . When a character is not being sent, the line is kept in the 1-state.

2. The initiation of a character transmission is detected from the start bit, which is always 0.

3. The character bits always follow the start bit.

4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Asynchronous way of data transfer can be achieved using two methods:

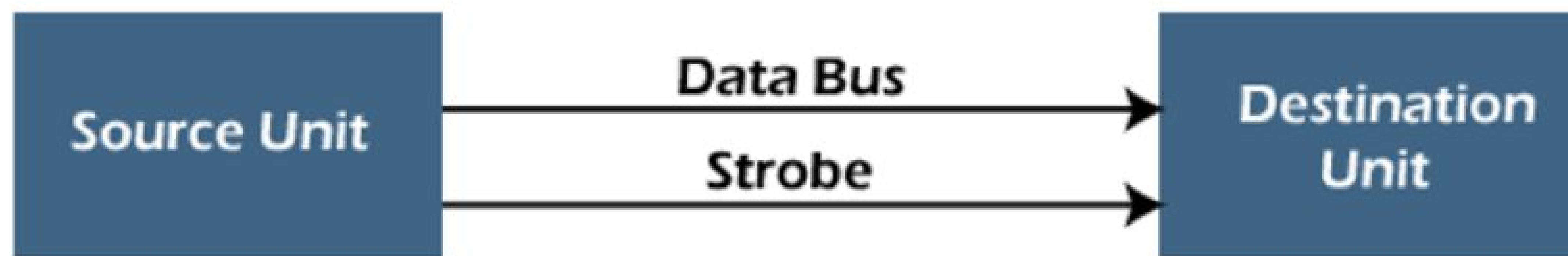
1) Strobe control

2) Handshaking

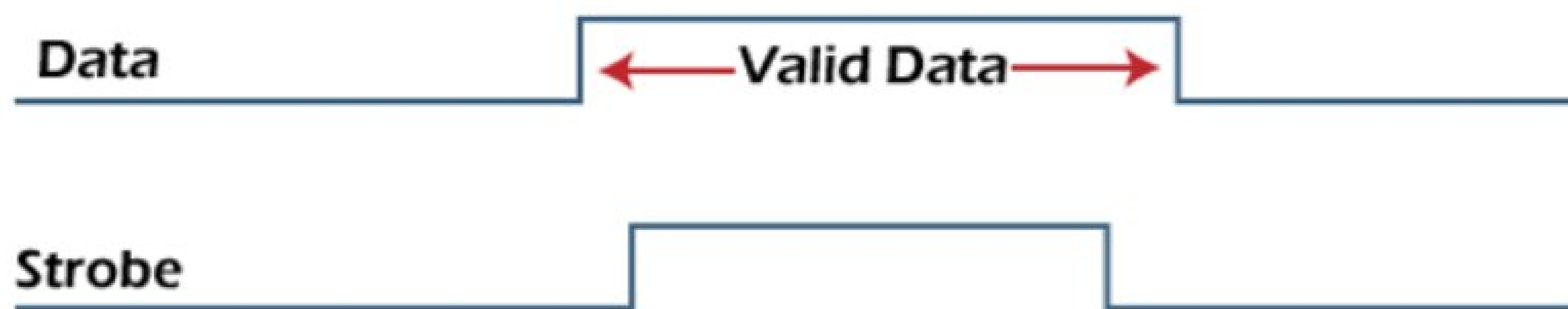
Strobe Control Method: The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as a strobe, and it may be achieved either by source or destination, depending on which initiate the transfer.

Source initiated strobe: In the below block diagram, strobe is initiated by source, and as shown in the timing diagram, the source unit first places the data on the data bus.

Scanned with OKEN Scanner



(a) Block Diagram

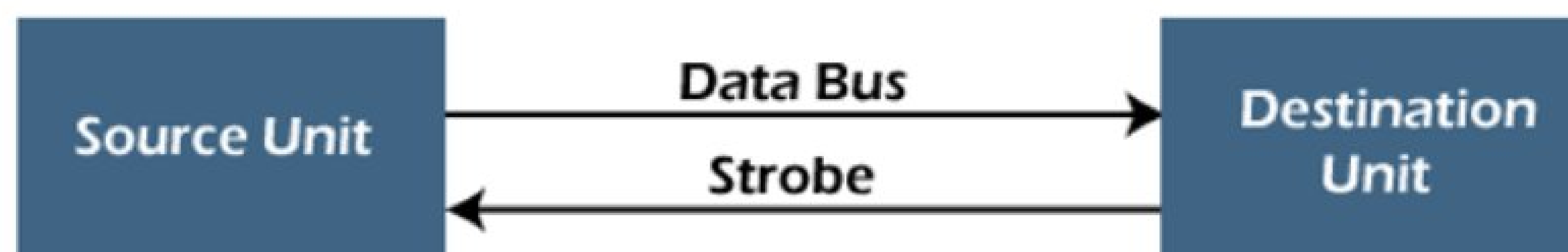


(b) Timing Diagram

After a brief delay, the source activates a strobe pulse. The information on the data bus and strobe control signal remains in the active state for a sufficient time to allow the destination unit to receive the data. **The destination unit uses a falling edge of strobe control to transfer the contents of a data bus to one of its internal registers.** The source removes the data from the data bus after it disables its strobe pulse. Thus, new valid data will be available only after the strobe is enabled again.

Example: The strobe may be a memory-write control signal from the CPU to a memory unit.

Destination initiated strobe: In the below block diagram, the strobe initiated by destination, and in the timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data.



(a) Block Diagram

(b) Timing Diagram

Scanned with OKEN Scanner

The falling edge of the strobe pulse can use again to trigger a destination register. The destination unit then disables the strobe. Finally, and source removes the data from the data bus after a determined time interval.

Example: the strobe may be a memory read control from the memory unit to CPU.

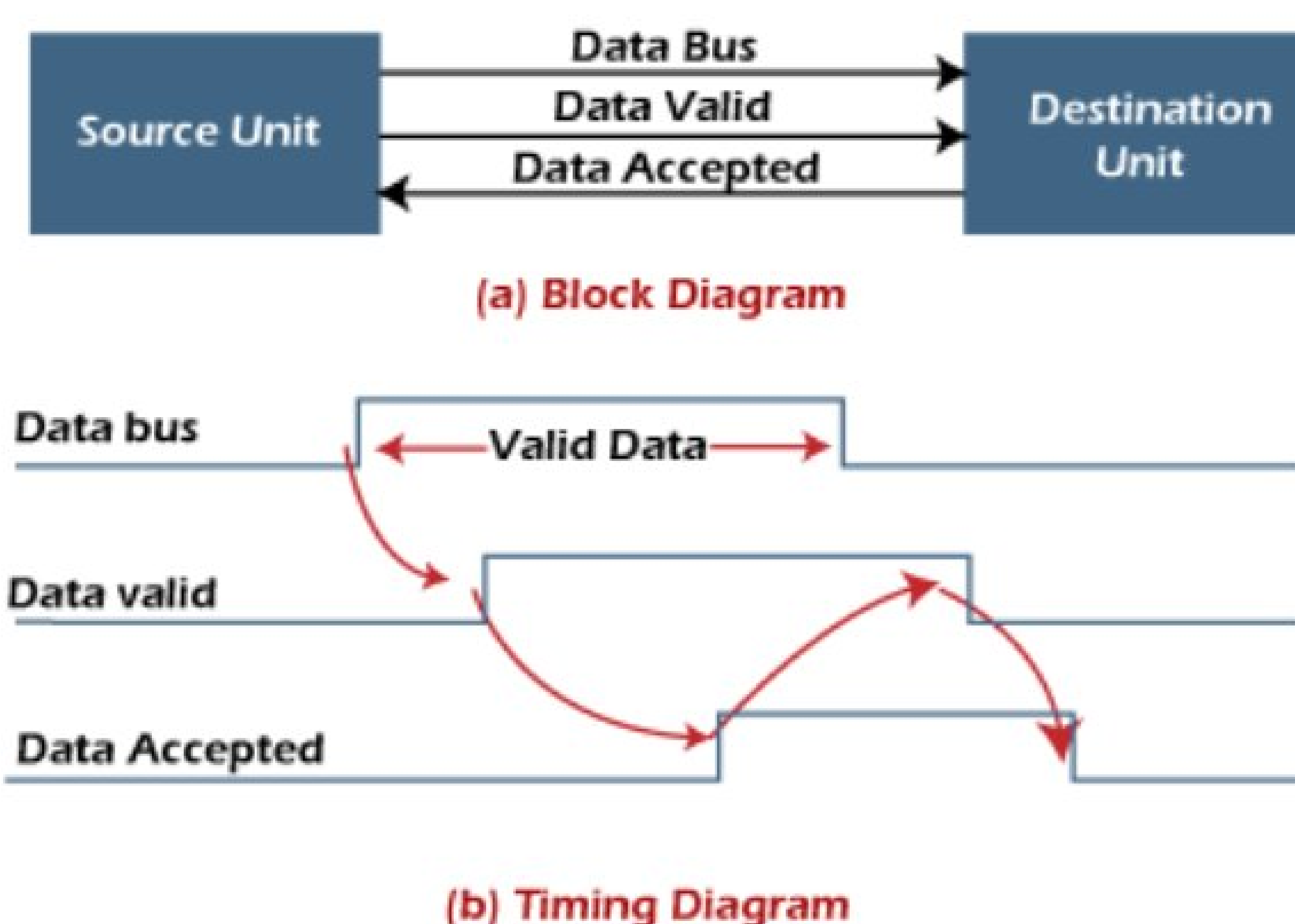
Handshaking Method: The strobe method has the disadvantage that the source unit that initiates the transfer has no way of knowing whether the destination has received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has placed data on the bus.

So this problem is solved by the handshaking method. The handshaking method introduces a second control signal line.

In this method, one control line is in the same direction as the data flow in the bus from the source to the destination. The source unit uses it to inform the destination unit whether there are valid data in the bus.

The other control line is in the other direction from the destination to the source. This is because the destination unit uses it to inform the source whether it can accept data. And in it also, the sequence of control depends on the unit that initiates the transfer. So it means the sequence of control depends on whether the transfer is initiated by source and destination.

Source initiated handshaking: In the below block diagram, two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.



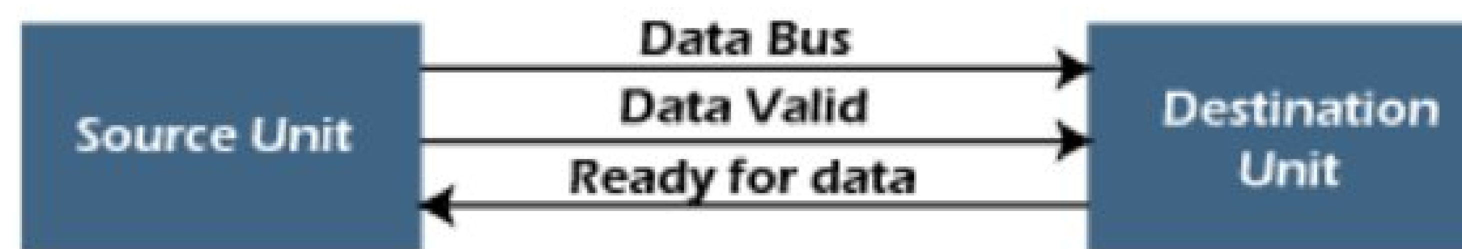
The timing diagram shows the timing relationship of the exchange of signals between the two units. The source initiates a transfer by placing data on the bus and enabling its data valid signal. The destination unit then activates the data accepted signal after it accepts the data from the bus.

The source unit then disables its valid data signal, which invalidates the data on the bus. After this, the destination unit disables its data accepted signal, and the system goes into its initial state. The source unit does not send the next data item until after the destination unit shows

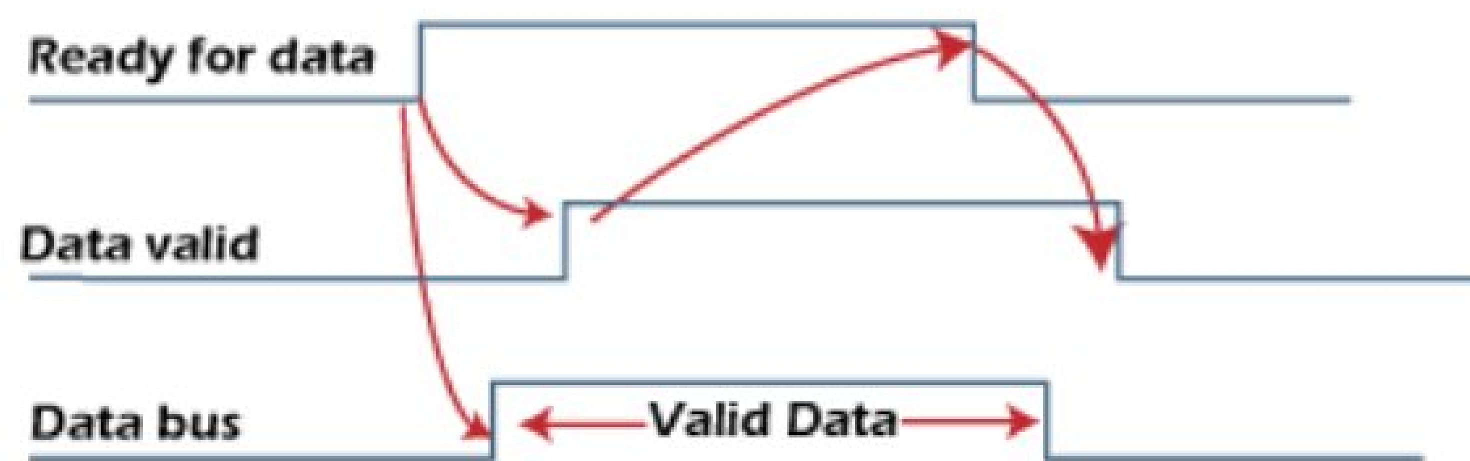
Scanned with OKEN Scanner

readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present at any given time.

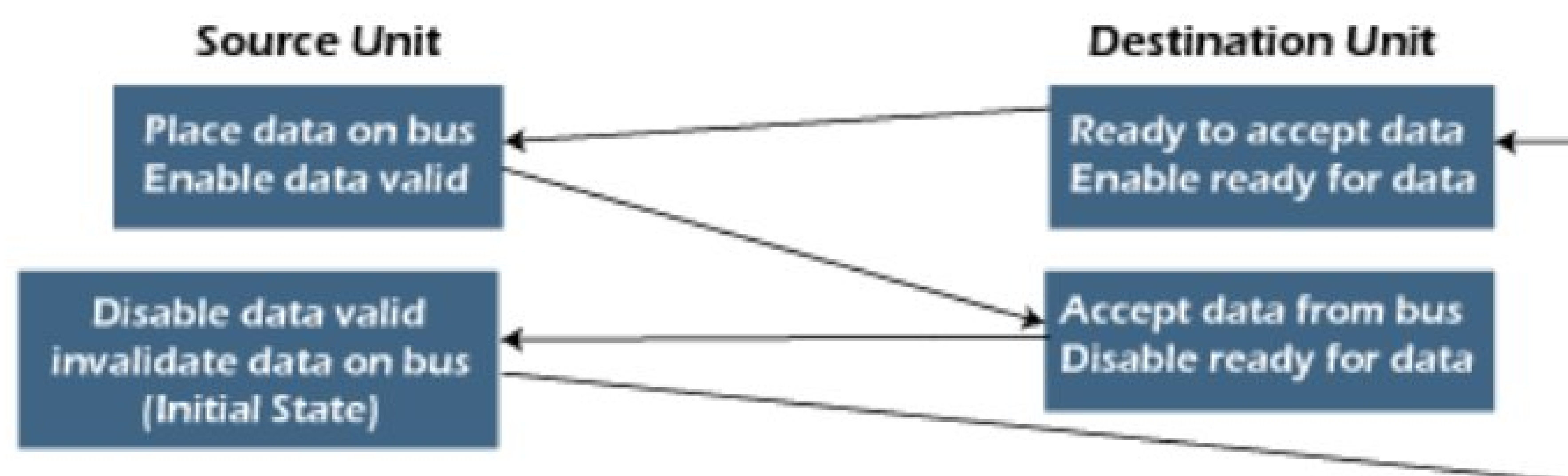
Destination initiated handshaking: In the below block diagram, the two handshaking lines are "**data valid**", generated by the source unit, and "**ready for data**" generated by the destination unit.



(a) Block Diagram



(b) Timing Diagram



(c) Sequence Diagram (Sequence of events)

I/O Transfers:

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/0 instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/0 devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1) **Program Controlled I/O**
2) **Interrupt-initiated I/0**
3) **Direct memory access (DMA)**

1. ***Program Controlled I/O:*** Programmed I/0 operations are the result of I/0 instructions written in the computer program. ***CPU executes a program that transfers data between I/O device and memory.*** Each data item transfer is initiated by an instruction in the program. **Usually, the transfer is to and from a CPU register and peripheral.** *Other*

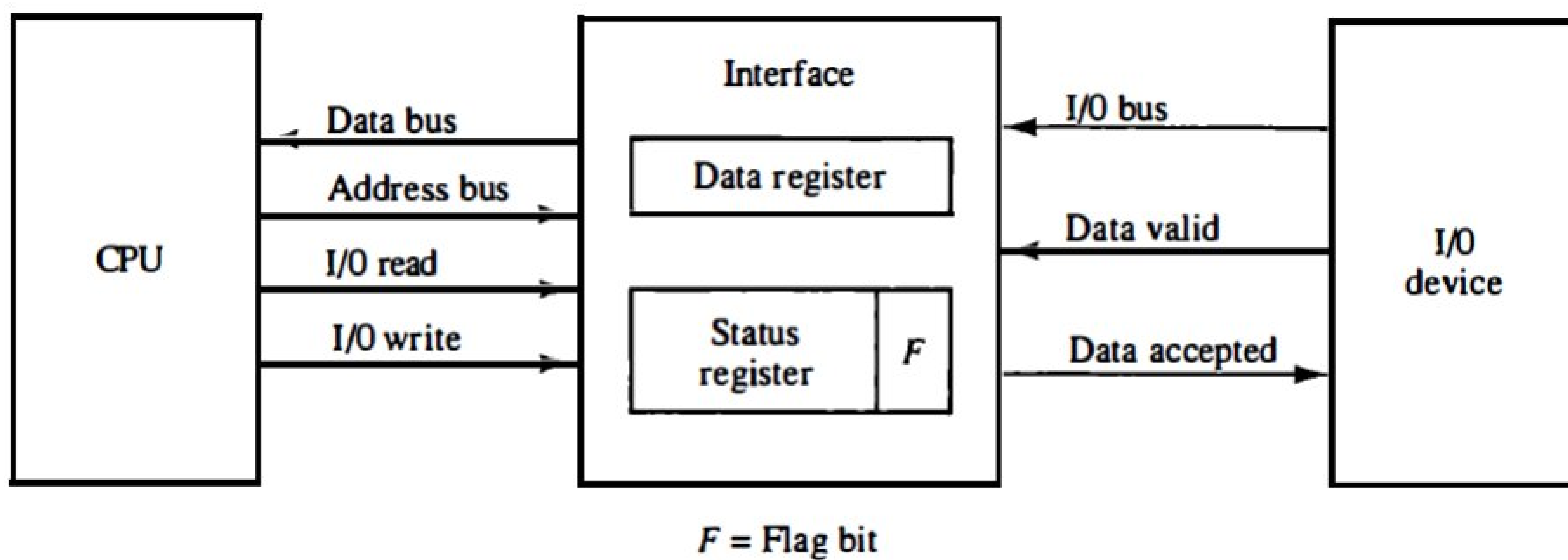
Scanned with OKEN Scanner

instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. In the programmed I/0 method, the CPU stays in a program loop until the I/0 unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly.

Example of Programmed I/0:

In the programmed I/0 method, the I/0 device does not have direct access to memory. A transfer from an I/0 device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig below:



F = Flag bit

The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.

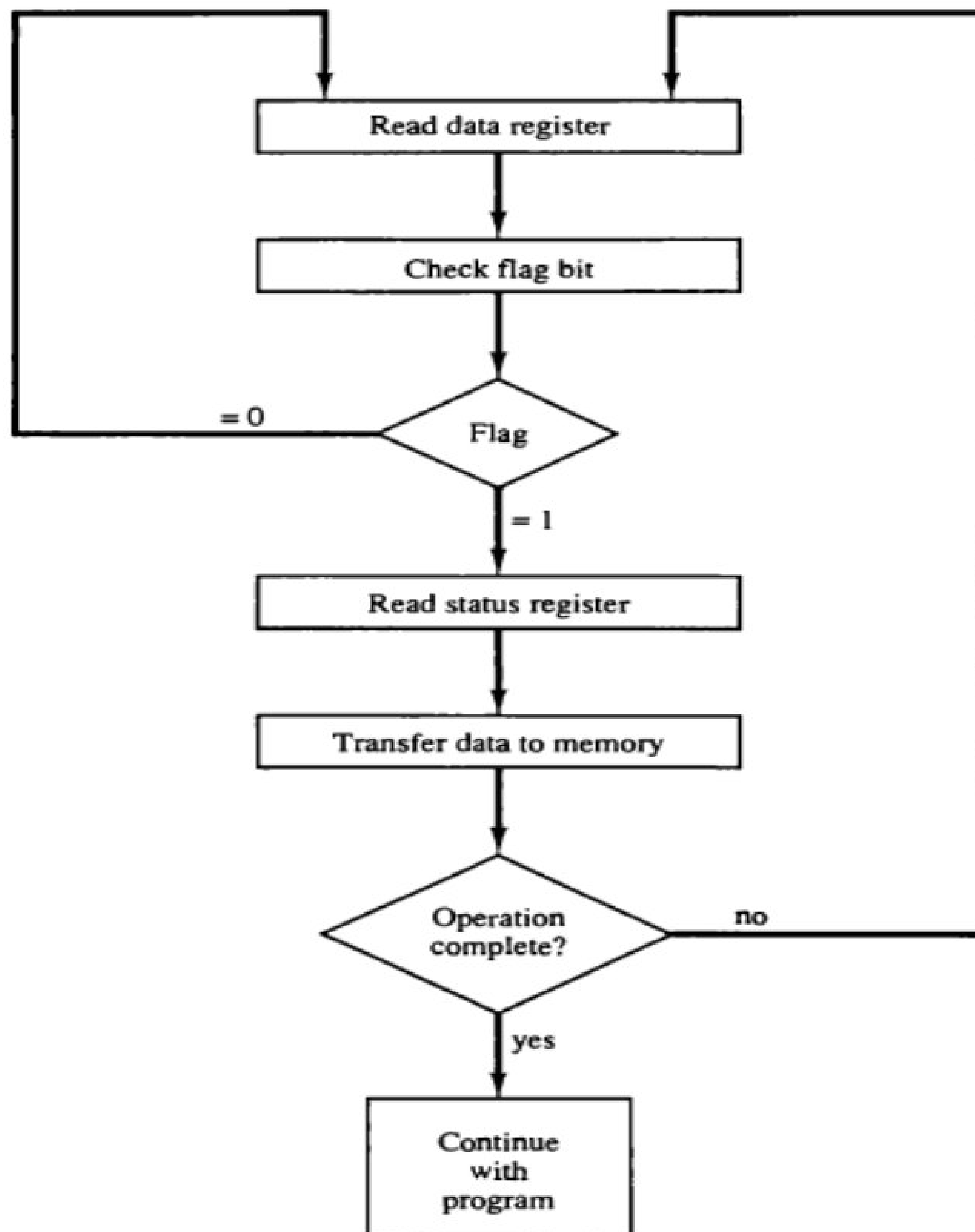
A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the VO device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte. A flowchart of the program that must be written for the CPU is shown in Fig below. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1 Read the status register.

Scanned with OKEN Scanner

2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.

3. Read the data register.



Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. The programmed VO method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously.

<u>2) INTERRUPT DRIVEN I/O:</u>

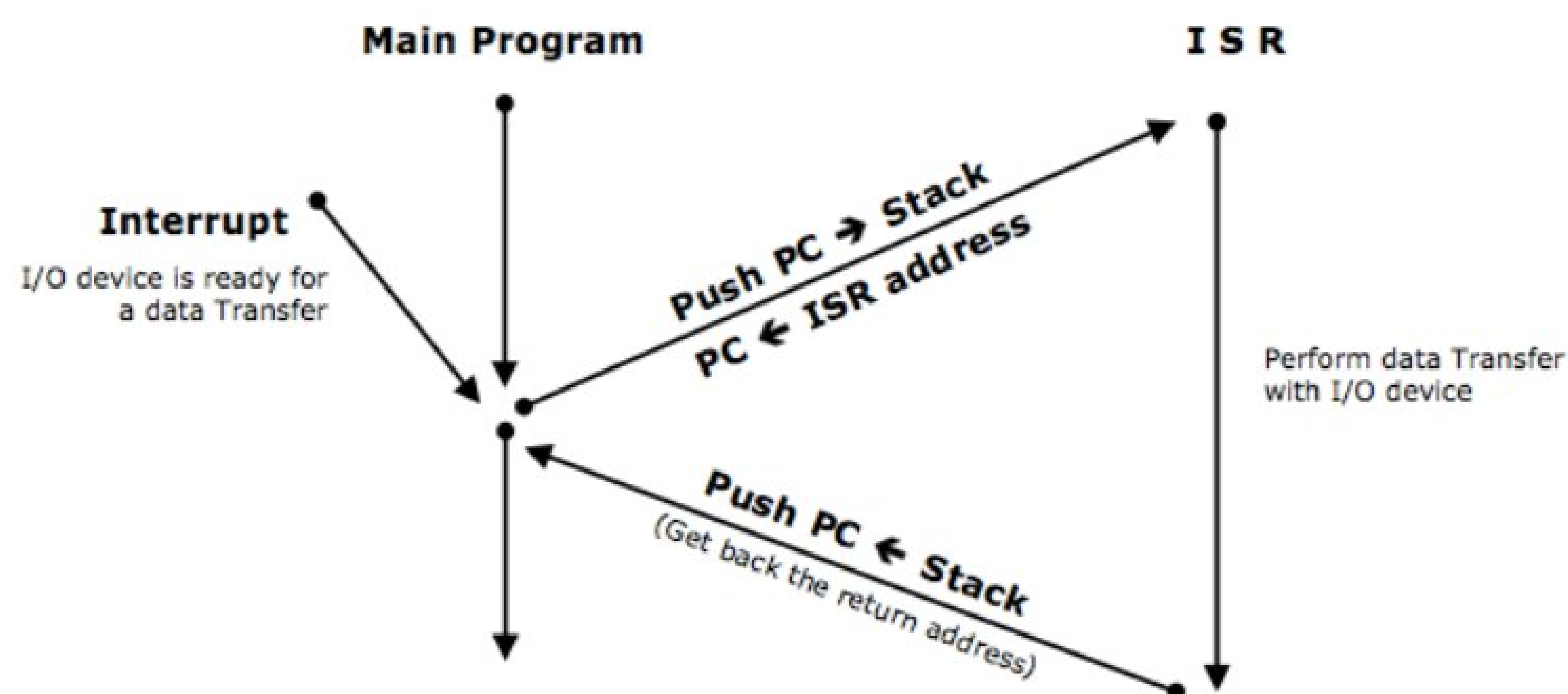
1) In interrupt driven I/O, the transfer is not initiated by the processor.
2) Instead, **an I/O device which wants to perform a data transfer with the processor, must give an interrupt to the processor.**
3) An interrupt is a condition that makes the processor execute an ISR (Interrupt Service Routine).
4) In the ISR, processor will perform data Transfer with the I/O device.
5) This relieves the processor from periodically checking the status of every I/O device thereby saves as
lot of time of the processor.
6) **The processor is free to carry on its own operations.**
7) Whenever a device wants to transfer data, it will interrupt the processor.
8) This is how many I/O devices Transfer data with the processor.

Scanned with OKEN Scanner

9) E.g.:: **Keyboard.** Instead of the processor checking all the time, whether a key is pressed, the
keyboard interrupts the processor as an when we press a key. In the ISR of the keyboard, which is a
part of the keyboard driver software, the processor will read the data from the keyboard.
10) **Hence interrupt driven I/O is much better than Polled I/O (Programmed I/O).**

INTERRUPT HANDLING MECHANISM

1) When an interrupt occurs, processor, firstly, finishes the current instruction.
2) It then suspends the current program and executes an ISR.
3) To do so, it Pushes the value of PC (address of next instruction), into the stack.
4) Now it loads the ISR address into PC and proceeds to execute the ISR.
5) At the end of the ISR, it POPs the return address from the stack and loads it back into PC.
6) This is how the processor return to the very next instruction in the program.



	INTERRUPT DRIVEN I/O	POLLING (PROGRAMMED I/O)
1	**I/O device interrupts the processor whenever it wants to perform a data Transfer.**	**Processor periodically checks (polls) the status of every I/O device to know if it wants to perform a data Transfer.**
2	**Processor is free** to carry on its own operations, hence **saves system time**.	**Processor is busy** in constantly checking all I/O devices, hence **system time wasted**.
3	**Additional hardware required** to handle interrupts. E.g.:: 8259 Programmable interrupt controller.	Additional hardware **not required**.
4	Increases **cost and complexity** of the system.	System is **cheaper and less complex**.
5	Interrupt **priority** has to be managed through software or through hardware.	No such issue.
6	Interrupt **vector addresses** (ISR Addresses) need to be stored in an Interrupt Vector table - **IVT**.	No such issue.

Scanned with OKEN Scanner

3) DMA BASED I/O

DMA means **transferring data directly between memory and I/O.**

DMA transfers are **very fast** as compared to Processor based transfers due to two reasons.

1. They are **hardware based** so no time is wasted in fetching and decoding instructions.
2. Transfers are **directly between memory and I/O** without data going via the Processor.

To Perform a DMA transfer we need a **DMA Controller like 8237/ 8257.**

It is capable of taking control of the buses from the Processor.

The process is performed as follows.

1) By Default **Processor is the bus master**.

2) The DMA transfer parameters first initialized by the processor.

3) **Processor programs two registers inside the DMAC called CAR and CWCR** giving the starting
address and the number of bytes to be transferred.

4) DMAC now ensures that the I/O device is ready for the transfer by checking the DREQ signal.

5) **If DREQ=1, then DMAC gives HOLD signal** to the Processor requesting control of the system bus.

6) **Processor releases control of the bus** after finishing the current machine (bus) cycle.

7) Processor **gives HLDA** informing DMAC that it is now the bus master.

8) **DMAC issues DACK#** (by default active low, but can be changed) to I/O device indicating that the
transfer is about to begin.

9) Now DMAC **transfers one byte in one cycle**.

10) After every byte is transferred the **Address register and Count register are decremented by 1**.

11) This repeats till Count reaches **"0"** also called **Terminal Count**.

12) Now the **transfer is complete**.

13) DMAC **returns the system bus to Processor by making HOLD = 0**.

14) Processor once again **becomes bus master**.

Advantage of DMA

DMA transfers are very fast.

Drawback of DMA

DMAC becomes the bus master. Hence during DMA cycles, the processor cannot perform any operations
as the bus is already being used for DMA. The processor remains in HOLD state.

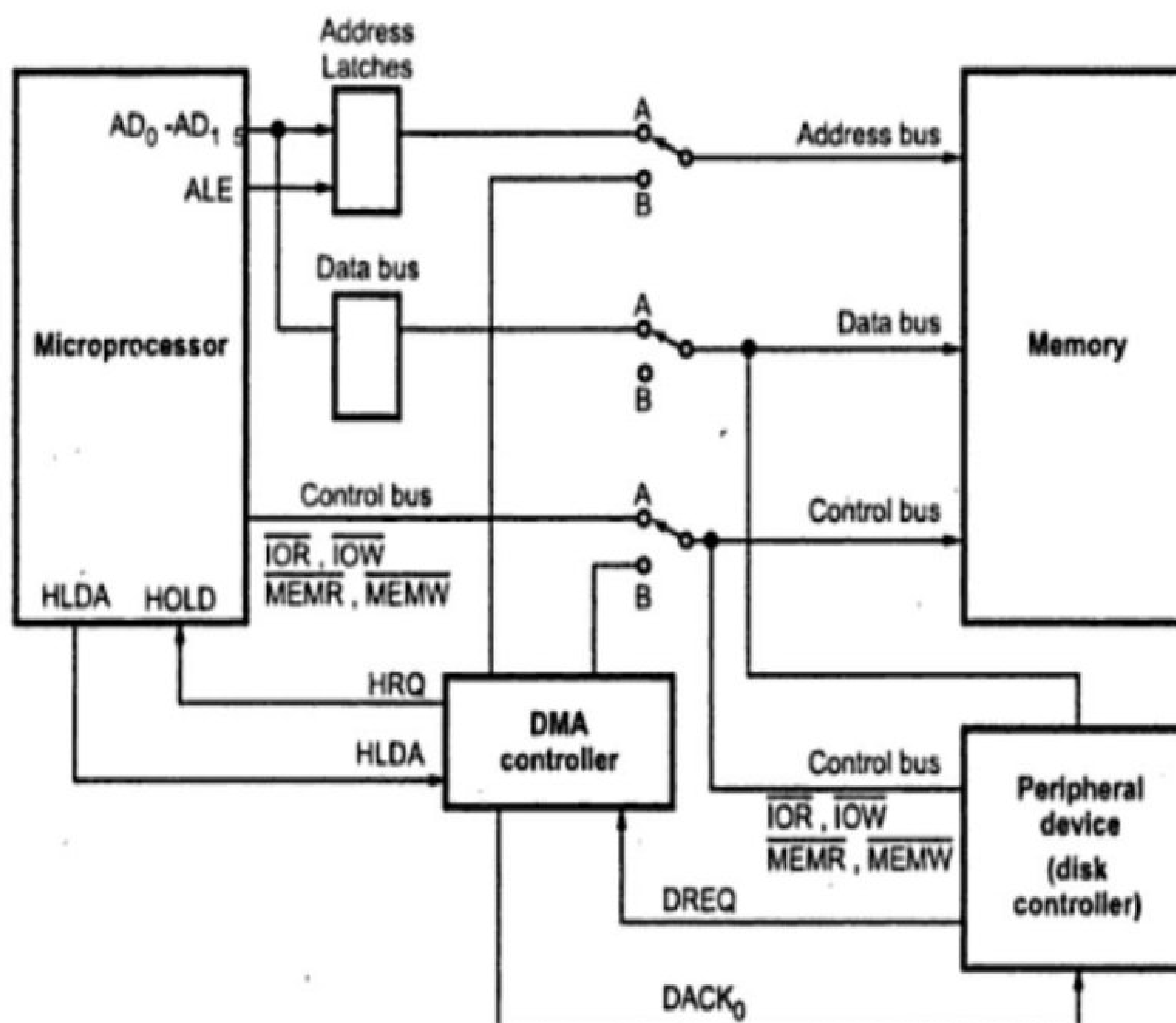
Difference between Interrupt Request and DMA request

When an interrupt occurs, the processor has to suspend the current program, execute the ISR and then
return to the next instruction of the main program. Hence it is necessary that the **processor completes the current instruction** before servicing an interrupt request.

When a DMA request occurs, the processor has to simply relinquish (give away) control of the system bus
and enter hold state. When ever it gets back the bus it can **resume from where it had left**. Hence the processor **need not finish the current instruction** before servicing a DMA request. It simply has to **finish the current machine cycle**. **Hence Instruction cycles are Interrupt Breakpoints and Machine cycles are DMA breakpoints.**

Scanned with OKEN Scanner

DMA Operation



<u>TYPES / METHODS / TECHNIQUES OF DMA TRANSFERS</u>

There are **four modes of data transfer**:

1) <u>BLOCK TRANSFER MODE / BURST MODE</u>.

In this mode, the DMAC is programmed to **transfer ALL THE BYTES** in one complete DMA operation. After a byte is transferred, the **CAR and CWCR are adjusted** accordingly. The **system bus** is **returned** to the **processor, ONLY after all the bytes are transferred.** It is the **fastest** form of DMA but keeps the **processor inactive** for a long time.

2) <u>SINGLE BYTE TRANSFER MODE/ CYCLE STEALING</u>.

Once the DMAC becomes the bus master, it will transfer only **ONE BYTE** and return the bus to the processor. As soon as the processor performs one bus cycle, DMAC will once again take the bus back from the processor. Hence both **DMAC and processor** are **constantly stealing bus cycles** from each other. It is the **most popular** method of DMA, because it keeps the **processor active in the background**. After a byte is transferred, the **CAR** and **CWCR** are **adjusted** accordingly.

3) <u>DEMAND TRANSFER MODE</u>.

It is very **similar to Block Transfer**, **except** that the **DREQ must remain active throughout the DMA operation**. **If** during the operation **DREQ goes low**, the **DMA operation is stopped** and the **busses** are **returned** to the **processor.**

In the meantime, the **processor** can **continue** with its own operations. **Once DREQ goes high again**, the **DMA operation continues** from where it had stopped. This means, the transfer happens on demand from the I/O device, hence the name.

4) <u>HIDDEN MODE / TRANSPARENT MODE</u>.

Scanned with OKEN Scanner

In this mode, **once the processor programs all parameters inside the DMAC, the DMAC** does not request the processor for the control of the bus. Instead, it observes the processor. **It waits for the processor to enter idle state**. Once the processor is idle, the DMAC will take control of the bus and perform the Transfer. So, the Transfer is **totally transparent to the processor** or hidden from the processor. Hence the name.

Interrupts and Exceptions:

Interrupt

The term Interrupt is usually reserved for hardware interrupts. They are program control interruptions caused by external hardware events. Here, external means external to the CPU. Hardware interrupts usually come from many different sources such as timer chip, peripheral devices (keyboards, mouse, etc.), I/O ports (serial, parallel, etc.), disk drives, CMOS clock, expansion cards (sound card, video card, etc). That means hardware interrupts almost never occur due to some event related to the executing program.

Example –

An event like a key press on the keyboard by the user, or an internal hardware timer timing out can raise this kind of interrupt and can inform the CPU that a certain device needs some attention. In a situation like that the CPU will stop whatever it was doing (i.e. pauses the current program), provides the service required by the device and will get back to the normal program. When hardware interrupts occur and the CPU starts the ISR, other hardware interrupts are disabled (e.g. in 80×86 machines). If you need other hardware interrupts to occur while the ISR is running, you need to do that explicitly by clearing the interrupt flag (with sti instruction). In 80×86 machines, clearing the interrupt flag will only affect hardware interrupts.

Exception

Exception is a software interrupt, which can be identified as a special handler routine. An exception occurs due to an "exceptional" condition that occurs during program execution.

Example –

Division by zero, execution of an illegal opcode or memory related fault could cause exceptions. Whenever an exception is raised, the CPU temporarily suspends the program it was executing and starts the ISR. ISR will contain what to do with the exception. It may correct the problem or if it is not possible it may abort the program gracefully by printing a suitable error message. Although a specific instruction does not cause an exception, an exception will always be caused by an instruction. For example, the division by zero error can only occur during the execution of the division instruction.

Exceptions and interrupts are unexpected events which will disrupt the normal flow of execution of instruction (that is currently executing by processor). An exception is an unexpected event from within the processor. Interrupt is an unexpected event from outside the processor. Whenever an exception or interrupt occurs, the hardware starts executing the code that performs an action in response to the exception. This action may involve killing a process, outputting an error message, communicating with an external device, or horribly crashing the entire computer system by initiating a "Blue Screen of Death" and halting the CPU. The instructions responsible for this action reside in the operating system kernel, and the code that performs this action is called the interrupt handler code. handler code is an operating system subroutine. Then, After the handler code is executed, it may be possible to continue execution after the instruction where the execution or interrupt occurred.

Scanned with OKEN Scanner

Whenever an exception or interrupt occurs, execution transitions from user mode to kernel mode where the exception or interrupt is handled. The following steps must be taken to handle an exception or interrupts:

While entering the kernel, the context (values of all CPU registers) of the currently executing process must first be saved to memory. The kernel is now ready to handle the exception/interrupt.

1) Determine the cause of the exception/interrupt.
2) Handle the exception/interrupt.

When the exception/interrupt have been handled the kernel performs the following steps:

1) Select a process to restore and resume.
2) Restore the context of the selected process.
3) Resume execution of the selected process.

At any point in time, the values of all the registers in the CPU defines the context of the CPU. Another name used for CPU context is CPU state.

Types of interrupts:

1. VECTORED AND NON-VECTORED INTERRUPTS

A key element in interrupt handling is the ISR address.

If an interrupt has a fixed ISR address, it is called a Vectored interrupt.

Such interrupts are **executed faster** as the ISR address is known to the processor.

But such interrupts are **rigid**. Since they have a fixed ISR address they can serve only **one device**. They cannot accept interrupts from multiple devices. So they cannot expand the interrupt structure.

E.g: NMI interrupt of 8086 (Has a fixed vector number i.e. 2)

If an interrupt does not have a fixed ISR address, it is called a Non-Vectored interrupt. Such interrupts are **executed slower**. The ISR address is **obtained from the** interrupting **device**, usually an interrupt controller like **8259**. But such interrupts are **flexible**. Since they don't have a fixed ISR address they **can accept interrupts from multiple devices**. So they can be used to **expand the interrupt structure**.

E.g: INTR interrupt of 8086 (Can service any vector number from 0… 255)

2. MASKABLE AND NON MASKABLE INTERRUPTS

Masking an interrupt means disabling it. A Maskable interrupt is an interrupt that can be disabled. If disabled, whenever this interrupt occurs, the processor will ignore it and simply continue the main program. Such interrupts are generally used to handle low priority, non-critical events like keyboard presses which can be easily disabled (Keypad can be locked)

E.g.:: INTR interrupt of 8086 (is disabled when Interrupt Flag is 0)

A Non-Maskable interrupt is an interrupt that cannot be disabled. Whenever this interrupt occurs, the processor will have to service it. Such interrupts are generally used to handle high priority, critical events like over-heating of the mother board, power failure etc.

E.g.:: NMI interrupt of 8086 (can never be disabled)

3. SOFTWARE AND HARDWARE INTERRUPTS

This is based on how the interrupt occurs.

If an interrupt is caused by **writing an instruction**, it is called a **software interrupt(Exception)**. Software interrupts are predictable events and are given by the programmer.

Scanned with OKEN Scanner

E.g.:: INT n instruction of 8086 (n can be anything between 0… 255)
If an interrupt is caused by **a signal on an external pin**, it is called a **hardware interrupt**.
Hardware interrupts are un-predictable events and are given by external devices.
E.g.:: NMI and INTR pins of 8086

I/O Device Interfaces:

Universal Serial Bus (USB):

The Universal Serial Bus (USB) [1] is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, cameras, and many more. The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s). Later, USB 2, called High-Speed USB, was introduced. It enables data transfers at speeds up to 480 Megabits/s. As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Superspeed) was developed. It supports data transfer rates up to 5 Gigabits/s.

The USB has been designed to meet several key objectives:

• Provide a simple, low-cost, and easy to use interconnection system

• Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications

• Enhance user convenience through a "plug-and-play" mode of operation

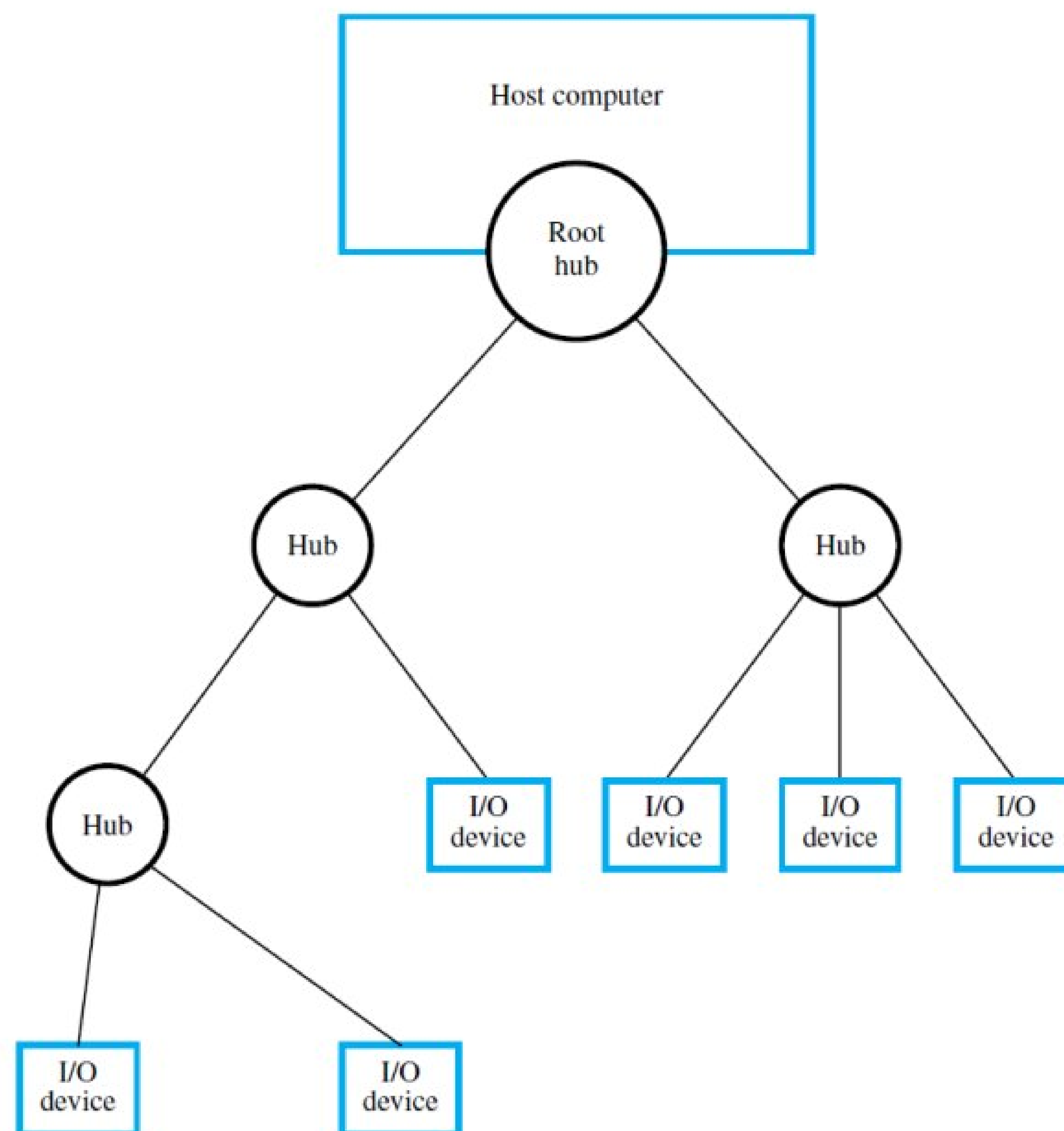
Plug-and-Play

When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device's interface to be able to communicate with it. The USB standard defines both the USB hardware and the software that communicates with it. Its plug-and-play feature means that when a new device is connected, the system detects its existence automatically. The software determines the kind of device and how to communicate with it, as well as any special requirements it might have. As a result, the user simply plugs in a USB device and begins to use it, without having to get involved in any of these details. The USB is also hot-pluggable, which means a device can be plugged into or removed from a USB port while power is turned on.

USB Architecture:

The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure as shown in Figure below:

Scanned with OKEN Scanner



Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker.

If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, the USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host processor. Hence, no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices. Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. The root hub of the USB, which is attached to the processor, appears as a single device. The host software communicates with individual devices by sending information to the root hub, which it forwards to the appropriate device in the USB tree.

When a device is first connected to a hub, or when it is powered on, it has the address 0. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected. When the host is informed that a new device has been connected, it reads the information in a special memory in the device's USB interface to learn about the device's capabilities. It then assigns the device a unique USB address and writes that address in one of the device's interface registers. It is this initial connection procedure that gives the USB its plug-and-play capability.

Isochronous Traffic on USB

An important feature of the USB is its ability to support the transfer of isochronous data in a simple manner. Isochronous data need to be transferred
at precisely timed regular intervals. To accommodate this type of traffic, the root hub transmits a uniquely recognizable sequence of bits over the USB tree every millisecond. This sequence of bits, called a Start of Frame character, acts as a marker indicating the beginning of isochronous data, which are transmitted after this character. Thus, digitized audio and video signals can be transferred in a regular and precisely timed manner.

Electrical Characteristics:

Scanned with OKEN Scanner

USB connections consist of four wires, of which two carry power, +5 V and Ground, and two carry data. Thus, I/O devices that do not have large power requirements can be powered directly from the USB. This obviates the need for a separate power supply for simple devices such as a memory key or a mouse.
Two methods are used to send data over a USB cable. When sending data at low speed, a high voltage relative to Ground is transmitted on one of the two data wires to represent a 0 and on the other to represent a 1. The Ground wire carries the return current in both cases. Such a scheme in which a signal is injected on a wire relative to ground is referred to as single-ended transmission.
The speed at which data can be sent on any cable is limited by the amount of electrical noise present. The term noise refers to any signal that interferes with the desired data signal and hence could cause errors. Single-ended transmission is highly susceptible to noise. The voltage on the ground wire is common to all the devices connected to the computer. Signals sent by one device can cause small variations in the voltage on the ground wire, and hence can interfere with signals sent by another device. Interference can also be caused by one wire picking up noise from nearby wires. The High-Speed USB uses an alternative arrangement known as differential signalling. The data signal is injected between two data wires twisted together. The ground wire is not involved. The receiver senses the voltage difference between the two signal wires directly,
without reference to ground. This arrangement is very effective in reducing the noise seen by the receiver, because any noise injected on one of the two wires of the twisted pair is also injected on the other. Since the receiver is sensitive only to the voltage difference between the two wires, the noise component is cancelled out. The ground wire acts as a shield for the data on the twisted pair against interference from nearby wires. Differential signalling allows much lower voltages and much higher speeds to be used compared to single-ended signalling.

SCSI Bus:

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI). The SCSI bus may be used to connect a variety of devices to a computer. It is particularly well-suited for use with disk drives. It is often found in installations such as institutional databases or email systems where many disks drives are used.
In the original specifications of the SCSI standard, devices are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates of up to 5 Megabytes/s. The standard has undergone many revisions, and its data transfer capability has increased rapidly. SCSI-2 and SCSI-3 have been defined, and each has several options. Data are transferred either 8 bits or 16 bits in parallel, using clock speeds of up to 80 MHz. There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission, where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, with a pair of wires for each signal. wires for each signal.

Data Transfer

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus or to the PCI bus. A SCSI bus may be connected directly to the processor bus, or more likel y to another standard I/O bus such as PCI, through a SCSI controller. Data and commands are transferred in the form of multi-byte messages called packets. To send commands or data to a device, the processor assembles the information in the memory then instructs the SCSI controller to transfer it to the device.

Scanned with OKEN Scanner

Similarly, when data are read from a device, the controller transfers the data to the memory and then informs the processor by raising an interrupt.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. Data are stored on a disk in blocks called sectors, where each sector may contain several hundred bytes. When a data file is written on a disk, it is not always stored in contiguous sectors. Some sectors may already contain previously stored information; others may be defective and must be skipped. Hence, a Read or Write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, on the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data are transferred at high speed. Another delay may ensue to reach the next sector, followed by a burst of data. A single Read or Write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation. Let us examine a complete Read operation as an example. The following is a simplified high-level description, ignoring details and signaling conventions. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller contends for control of the SCSI bus.
2. When it wins the arbitration process, the SCSI controller sends a command to the disk controller, specifying the required Read operation.
3. The disk controller cannot start to transfer data immediately. It must first move the read head of the disk to the required sector. Hence, it sends a message to the SCSI controller indicating that it will temporarily suspend the connection between them. The SCSI bus is now free to be used by other devices.
4. The disk controller sends a command to the disk drive to move the read head to the first sector involved in the requested Read operation. It reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data, it requests control of the bus. After it wins arbitration, it re-establishes the connection with the SCSI controller, sends the contents of the data buffer, then suspends the connection again.
5. The process is repeated to read and transfer the contents of the second disk sector.
6. The SCSI controller transfers the requested data to the main memory and sends an interrupt to the processor indicating that the data are now available.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. Messages refer to more complex operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of the disk's operation and how it moves from one sector to the next. The SCSI bus standard defines a wide range of control messages that can be used to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

Scanned with OKEN Scanner