UNIT IV

INPUT/OUTPUT ORGANIOZATION

ACCESSING I/O DEVICES:

In computing, input/output, or I/O, refers to the communication between an information processing system (computer), and the outside world. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. I/O devices are used by a person (or other system) to communicate with a computer.

Some of the input devices are keyboard, mouse, track ball, joy stick, touch screen, digital camera, webcam, image scanner, fingerprint scanner, barcode reader, and microphone and so on. Some of the output devices are speakers, headphones, monitors and printers. Devices for communication between computers, such as modems and network cards, typically serve for both input and output. I/O devices can be connected to a computer through a single bus which enables the exchange of information. The bus consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. Figure 5.1 shows the simple arrangement of I/O devices to processor and memory with single bus.

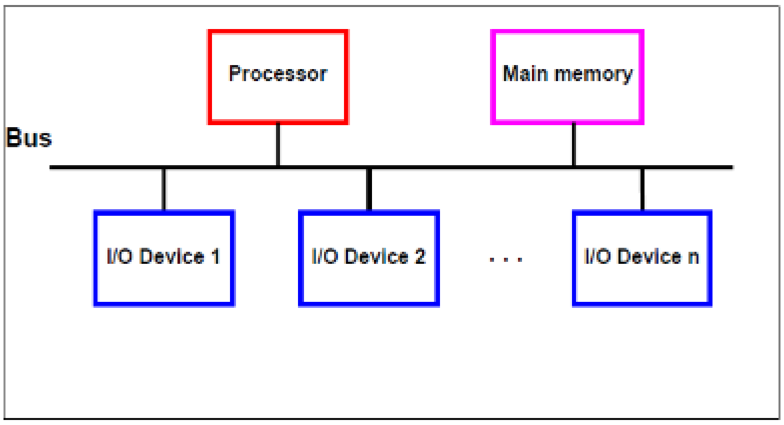


Fig: A Single bus structure

Memory-mapped I/O: The arrangement of I/O devices and the memory share the same address space is called memory-mapped I/O. With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of the input buffer associated with the keyboard, the instruction

Move DATAIN, R0
reads the data from DATAIN and stores them into processor register RO. Similarly, the instruction

Move R0, DATAOUT
sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer. Most computer systems use memory-mapped I/O. Some processors have special in and out instructions to perform I/O transfers.

Figure illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

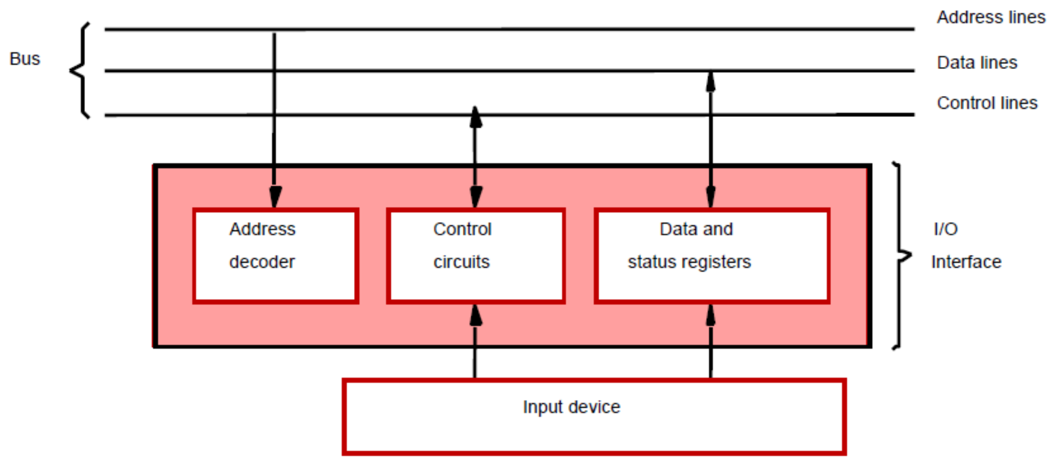


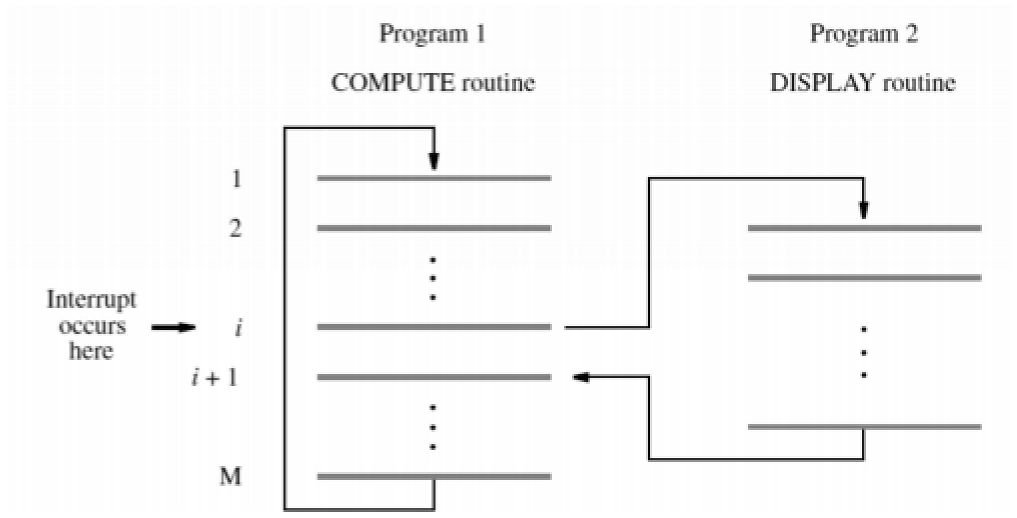
Figure 5.2 I/O interface for an input device

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. An input character is read only once.

For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. This is often accomplished in a program loop that repeatedly reads the status register and checks the state of SIN. When SIN becomes equal to 1, the program reads the input data register. A similar procedure can be used to control output operations using an output status flag, SOUT.

INTERRUPTS:

In the programmed I/O transfer, 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. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an interrupt request to the processor. Since the processor is no longer required to continuously poll the status of I/O devices, it can use the waiting period to perform other useful tasks. Indeed, by using interrupts, such waiting periods can ideally be eliminated. The routine executed in response to an interrupt request is called the interrupt-service routine, which is the DISPLAY routine in our example. Interrupts bear considerable resemblance to subroutine calls.



Assume that an interrupt request arrives during execution of instruction i in Figure. The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor returns to instruction i + 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i + 1, must be put in temporary storage in a known location. A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction i + 1. The return address must be saved either in a designated general-purpose register or on the processor stack. The processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This can be accomplished by means of a special control signal, called interrupt acknowledge, which is sent to the device through the interconnection network.

Most modern processors save only the minimum amount of information needed to maintain the integrity of program execution. This is because the process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also 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 some applications, long interrupt latency is unacceptable. For these reasons, the amount of information saved automatically by the processor when an interrupt request is accepted should be kept to a minimum. Typically, the processor saves only the contents of the program counter and the processor status register.

Some computers provide two types of interrupts. One saves all register contents, and the other does not. A particular I/O device may use either type, depending upon its response time requirements. Another interesting approach is to provide duplicate sets of processor registers. In this case, a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers. The duplicate registers are sometimes called the shadow registers.

Enabling and Disabling Interrupts:

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable such interruptions as desired. There are many situations in which the processor should ignore interrupt requests. For instance, the timer circuit should raise interrupt requests only when the COMPUTE routine is being executed. It should be prevented from doing so when some other task is being performed. In another case, it may be necessary to guarantee that a particular sequence of instructions is executed to the end without interruption because the interrupt-service routine may change some of the data used by the instructions in question.

For these reasons, some means for enabling and disabling interrupts must be available to the programmer. It is convenient to be able to enable and disable interrupts at both the processor and I/O device ends. The processor can either accept or ignore interrupt requests. An I/O device can either be allowed to raise interrupt requests or prevented from doing so. A commonly used mechanism to achieve this is to use some control bits in registers that can be accessed by program instructions. The processor has a status register (PS), which contains information about its current state of operation. Let one bit, IE, of this register be assigned for enabling/disabling interrupts. Then, the programmer can set or clear IE to cause the desired action. When IE = 1, interrupt requests from I/O devices are accepted and serviced by the processor. When IE = 0, the processor simply ignores all interrupt requests from I/O devices. The interface of an I/O device includes a control register that contains the information that governs the mode of operation of the device. One bit in this register may be dedicated to interrupt control. The I/O device is allowed to raise interrupt requests only when this bit is set to 1.

Let us now consider 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. A good choice is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. The processor saves the contents of the program counter and the processor status register. After saving the contents of the PS register, with the IE bit equal to 1, the processor clears the IE bit in the PS register, thus disabling further interrupts. Then, it begins execution of the interrupt-service routine. When a Return-from interrupt instruction is executed, the saved contents of the PS register are restored, setting the IE bit back to 1. Hence, interrupts are again enabled.

Handling Multiple Devices:

Let us now consider the situation where a number of devices capable of initiating interrupts 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:

1. How can the processor determine which device is requesting an interrupt? 2
2. . Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

   The means by which these issues are handled vary from one computer to another, and the approach taken is an important consideration in determining the computer's suitability for a given application. When an interrupt request is received it is necessary to identify the particular device that raised the request. Furthermore, if two devices raise interrupt requests at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced. The information needed to determine whether a device is requesting an interrupt is available in its status register. When the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the IRQ bit. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system. The first device encountered with its IRQ bit set to 1 is the device that should be serviced. An appropriate subroutine is then called to provide the requested service. The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we describe next.

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 interrupt handling schemes based on this approach. A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network. The processor's circuits determine the memory address of the required interrupt-service routine. A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as interrupt vectors, and they are said to constitute the interrupt-vector table. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors. Typically, the interrupt vector table is in the lowest-address range. The interrupt-service routines may be located anywhere in the memory. When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter.

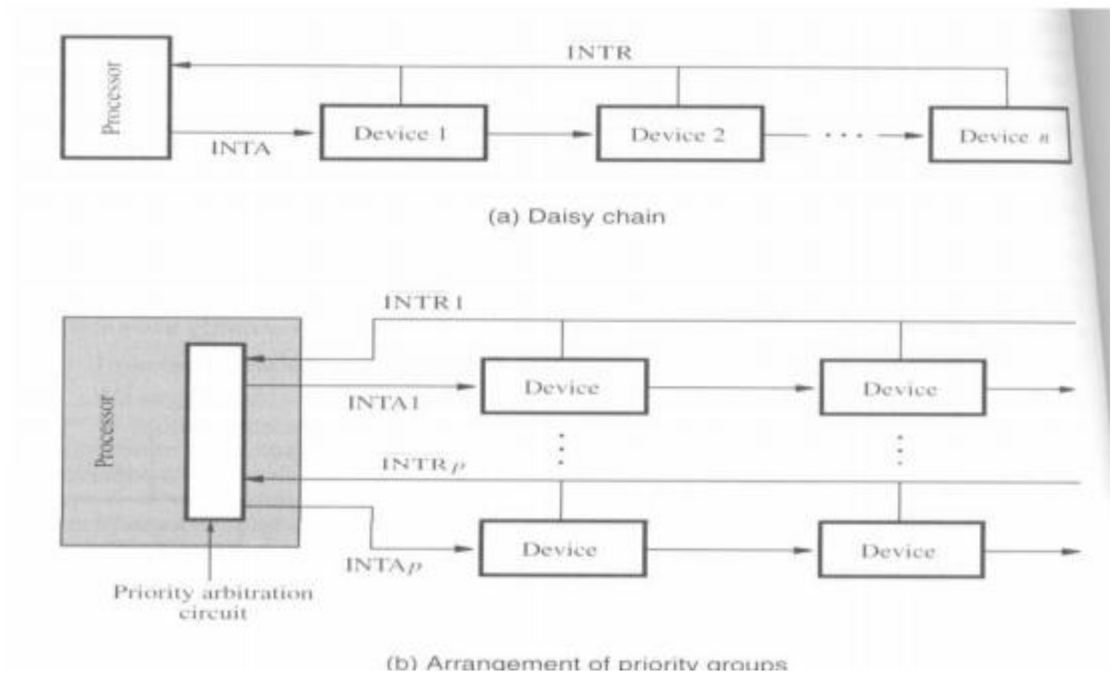
Interrupt Nesting:

The 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 a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices. For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation.

An interrupt request from a high-priority device should be accepted while the processor is servicing a request from a lower-priority device. A multiple-level priority organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own. At the time that execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device either automatically or with special instructions. This action disables interrupts from devices that have the same or lower level of priority. However, interrupt requests from higher-priority devices will continue to be accepted. The processor's priority can be encoded in a few bits of the processor status register.

<u>Simultaneous Requests</u>:

We also need to consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. This is done in hardware, by using arbitration circuits.



Exceptions:

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. So far, we have dealt only with interrupts caused by events associated with I/O data transfers. However, the interrupt mechanism is used in a number of other situations. The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception. We now describe a few other kinds of exceptions.

Recovery from Errors:

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt. The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero. When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine, which takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, we assumed that the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error associated with the current instruction, that instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging:

Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities: trace mode and breakpoints.

Trace Mode:

When processor is in trace mode , an exception occurs after execution of every instance using the debugging program as the exception service routine. The debugging program examine the contents of registers, memory location etc. On return from the debugging program the next instance in the program being debugged is executed The trace exception is disabled during the execution of the debugging program.

Break point:

Here the program being debugged is interrupted only at specific points selected by the user. An instance called the Trap (or) software interrupt is usually provided for this purpose. While debugging the user may interrupt the program execution after instance ‘I’ When the program is executed and reaches that point it examine the memory and register contents.

Privileged Exception:

To protect the OS of a computer from being corrupted by user program certain instance can be executed only when the processor is in supervisor mode. These are called privileged exceptions. When the processor is in user mode, it will not execute instance (ie) when the processor is in supervisor mode , it will execute instance.

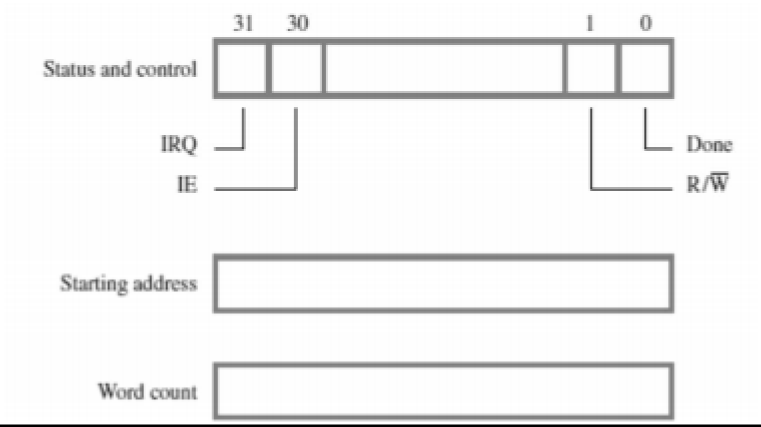
Use of Exceptions in Operating Systems:

The operating system (OS) software coordinates the activities within a computer. It uses exceptions to communicate with and control the execution of user programs. It uses hardware interrupts to perform I/O operations.

DIRECT MEMORY ACCESS:

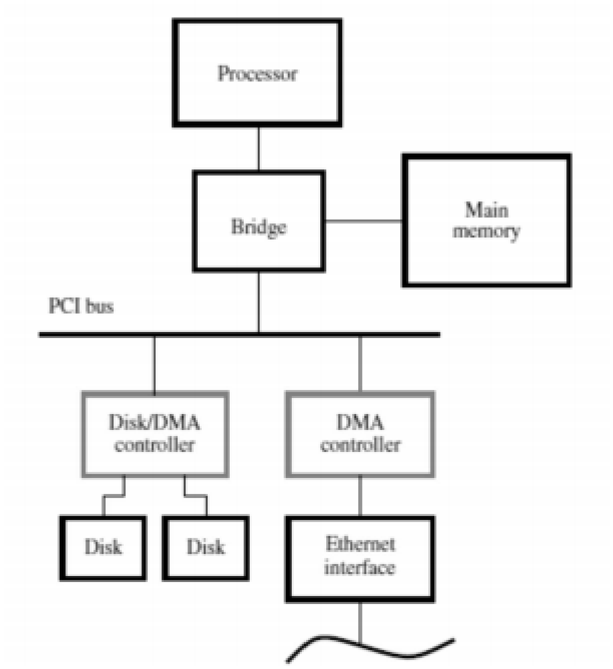
An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks. A special control unit is provided to manage the transfer, without continuous intervention by the processor. This approach is called direct memory access, or DMA. The unit that controls DMA transfers is referred to as a DMA controller. It may be part of the I/O device interface, or it may be a separate unit shared by a number of I/O devices. 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 provides the memory address and generates all the control signals needed. It increments the memory address for successive words and keeps track of the number of transfers.

Although a DMA controller transfer's data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an operating system routine. To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, the number of words in the block, and the direction of the transfer. The DMA controller then proceeds to perform the requested operation. When the entire block has been transferred, it informs the processor by raising an interrupt. Figure shows an example of the DMA controller registers that are accessed by the processor to initiate data transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a Read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a Write operation. Additional information is also transferred as may be required by the I/O device. For example, in the case of a disk, the processor provides the disk controller with information to identify where the data is located on the disk.



When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt. Figure shows how DMA controllers may be used in a computer system such as that in Figure. One DMA controller connects a high-speed Ethernet to the computer's I/O bus (a PCI bus in the case of Figure). The disk controller, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller.

To start a DMA transfer of a block of data from the main memory to one of the disks, an OS routine writes the address and word count information into the registers of the disk controller. The DMA controller proceeds independently to implement the specified operation. When the transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit.



Cycle Stealing

• Requests by DMA devices for using the bus are having higher priority than processor requests

• Top priority is given to high speed peripherals such as, Disk High speed Network Interface and Graphics display device.

• Since the processor originates most memory access cycles, the DMA controller can be said to steal the memory cycles from the processor.

• This interviewing technique is called Cycle stealing.

Burst Mode:

The DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as Burst/Block Mode.

Bus Arbitration:

BUS MASTER: The device which is allowed to initiate the data transfer is called bus master.

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

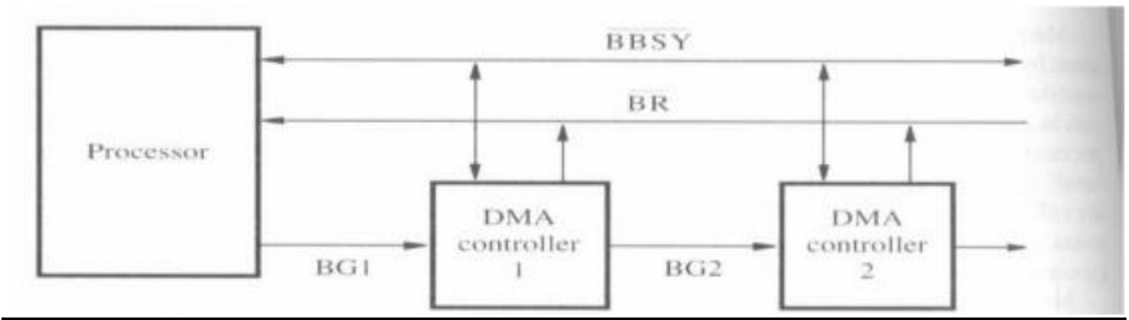
Types: There are 2 approaches to bus arbitration. They are,

Centralized arbitration (A single bus arbiter performs arbitration)

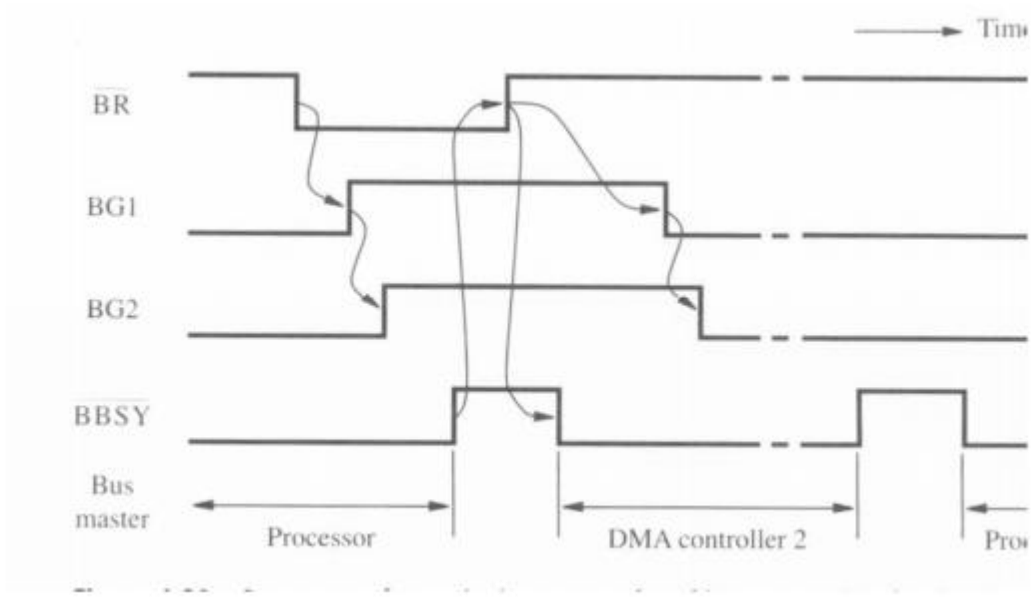
Distributed arbitration (all devices participate in the selection of next bus master).

Centralized Arbitration:

Here the processor is the bus master and it may grants bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line. The signal on BR is the logical OR of the bus request from all devices connected to it. When BR is activated the processor activates the Bus Grant Signal (BGI) and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a daisy chain arrangement. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

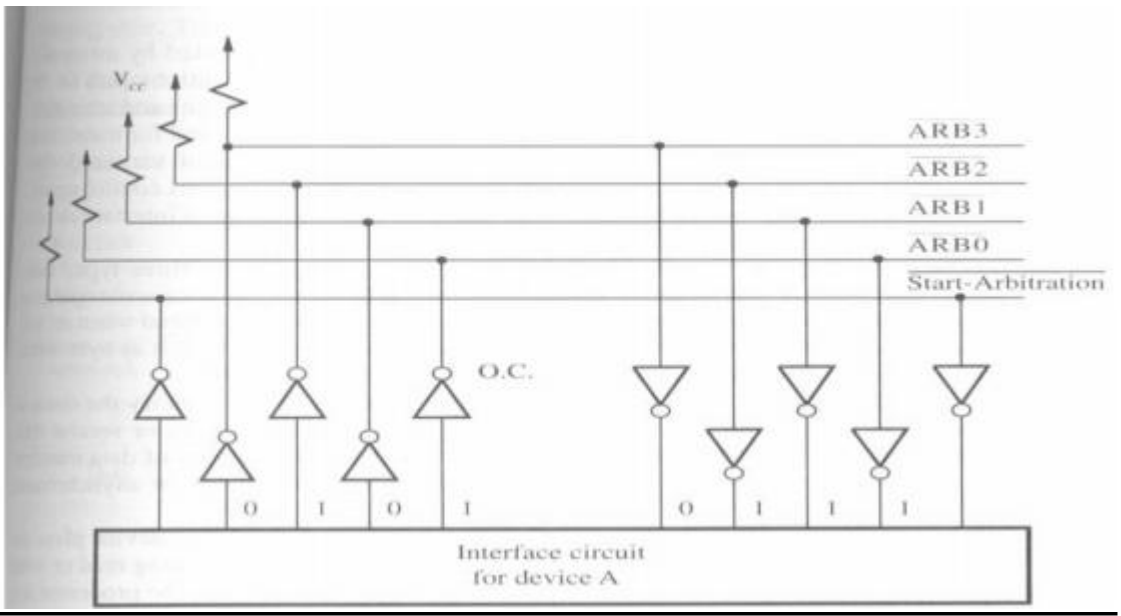


The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as bus master, it may perform one or more data transfer. After it releases the bus, the processor resources bus mastership



Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process. Each device on the bus is assigned a 4 bit id. When one or more devices request the bus, they assert the Start-Arbitration signal & place their 4 bit ID number on four open collector lines, ARB0 to ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type. Hence, if the i/p to one driver is equal to 1, the i/p to another driver connected to the same bus line is equal to „0‟(ie. bus the is in low-voltage state).



INTERFACE CIRCUITS:

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a port, and it can be either a parallel or a serial port. A parallel port transfers multiple bits of data simultaneously to or from the device. A serial port sends and receives data one bit at a time. An I/O interface does the following:

1. Provides a register for temporary storage of data

2. Includes a status register containing status information that can be accessed by the processor

3. Includes a control register that holds the information governing the behaviour of the interface

4. Contains address-decoding circuitry to determine when it is being addressed by the processor

5. Generates the required timing signals

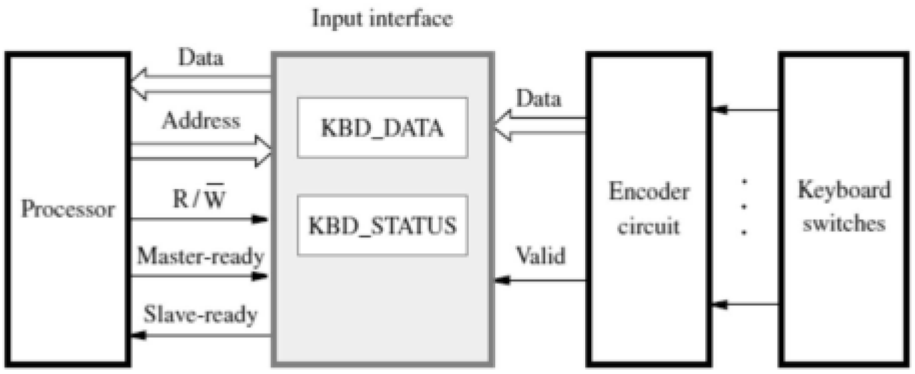
6. Performs any format conversion that may be necessary to transfer data between the processor and the I/O **device, such as parallel-to-serial conversion in the case of a serial port**

Parallel port:

We describe an interface circuit for an 8-bit input port that can be used for connecting a simple input device, such as a keyboard. Then, we describe an interface circuit for an 8-bit output port, which can be used with an output device such as a display. We assume that these interface circuits are connected to a 32-bit processor that uses memory-mapped I/O and the asynchronous bus protocol depicted.

Input Interface:

Figure shows a circuit that can be used to connect a keyboard to a processor. The registers in this circuit correspond to those given in Figure. Assume that interrupts are not used, so there is no need for a control register. There are only two registers: a data register, KBD_DATA, and a status register, KBD_STATUS. The latter contains the keyboard status flag, KIN.



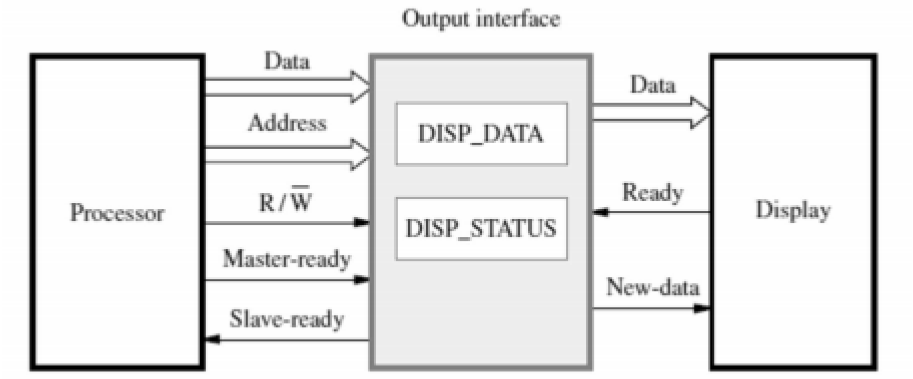
A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such mechanical pushbutton switches is that the contacts bounce when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position. Although bouncing may last only one or two milliseconds, this is long enough for the computer to erroneously interpret a single pressing of a key as the key being pressed and released several times. The effect of bouncing can be eliminated using a simple debouncing circuit, which could be part of the keyboard hardware or may be incorporated in the encoder circuit. Alternatively, switch bouncing can be dealt with in software. The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided. When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1.

The output of the encoder in Figure consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status flag KIN to be set to 1. The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The interface circuit is shown connected to an asynchronous bus on which transfers are controlled by the handshake signals Master-ready and Slave-ready, as in Figure. The bus has one other control line, R/W, which indicates a Read operation when equal to 1.

Output Interface:

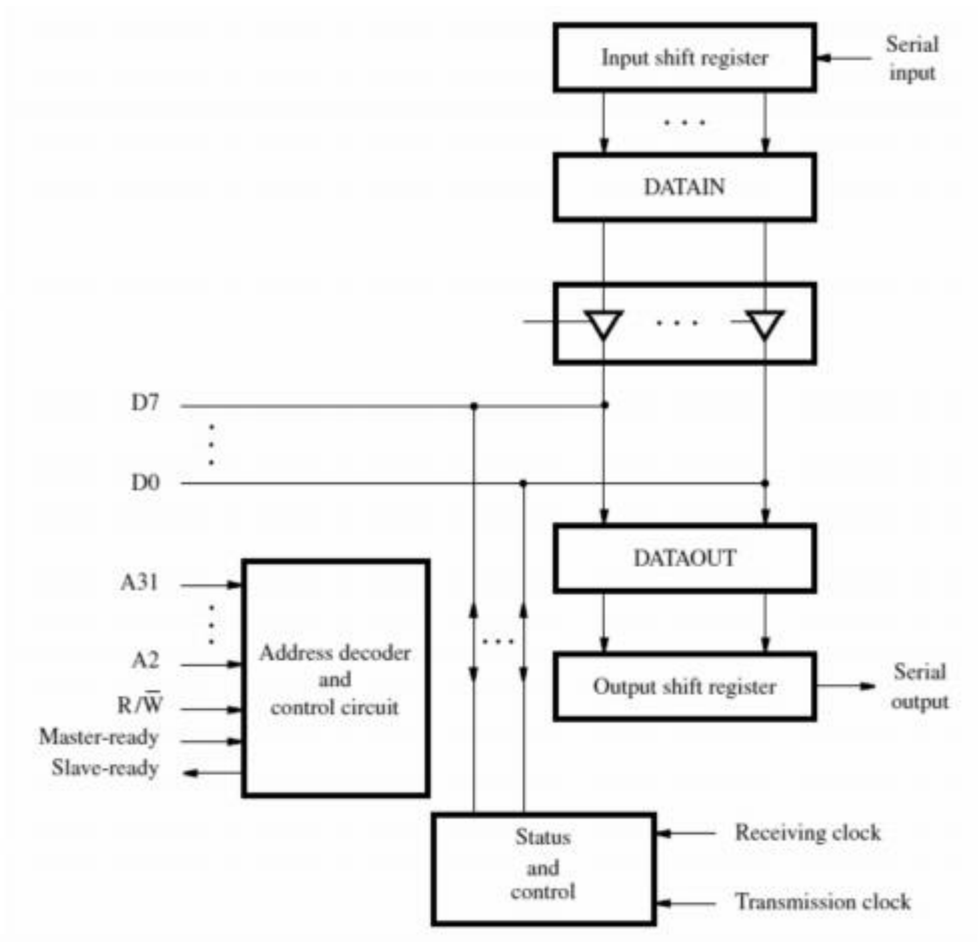
Let us now consider the output interface shown in Figure, which can be used to connect an output device such as a display. We have assumed that the display uses two handshake signals, New-data and Ready, in a manner similar to the handshake between the bus signals Master-ready and Slave-ready. When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1. When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA. When it is ready to receive another character, it asserts Ready again, and the cycle repeats.

Figure shows an implementation of this interface. Its operation is similar to that of the input interface of Figure 7.11, except that it responds to both Read and Write operations. A Write operation in which A2 = 0 loads a byte of data into register DISP_DATA. A Read operation in which A2 = 1 reads the contents of the status register DISP_STATUS. In this case, only the DOUT flag, which is bit b2 of the status register, is sent by the interface. The remaining bits of DISP_STATUS are not used. The state of the status flag is determined by the handshake control circuit.



Serial port:

A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in Figure. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are transferred to the output shift register, from which the bits are shifted out and sent to the I/O device.



The part of the interface that deals with the bus is the same as in the parallel interface described earlier. Two status flags, which we will refer to as SIN and SOUT, are maintained by the Status and control block. The SIN flag is set to 1 when new data are loaded into DATAIN from the shift register, and cleared to 0 when these data are read by the processor. The SOUT flag indicates whether the DATAOUT register is available. It is cleared to 0 when the processor writes new data into DATAOUT and set to 1 when data are transferred from DATAOUT to the output shift register.

During serial transmission, the receiver needs to know when to shift each bit into its input shift register. Since there is no separate line to carry a clock signal from the transmitter to the receiver, the timing information needed must be embedded into the transmitted data using an encoding scheme. There are two basic approaches. The first is known as asynchronous transmission, because the receiver uses a clock that is not synchronized with the transmitter clock. In the second approach, the receiver is able to generate a clock that is synchronized with the transmitter clock. Hence it is called synchronous transmission.

<u>STANDARD I/O INTERFACES:</u>

Standard interfaces have been developed to enable I/O devices to use interfaces that are independent of any particular processor. For example, a memory key that has a USB connector can be used with any computer that has a USB port.

UNIVERSAL SERIAL BUS (USB): The Universal Serial Bus (USB) 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 Super speed) 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

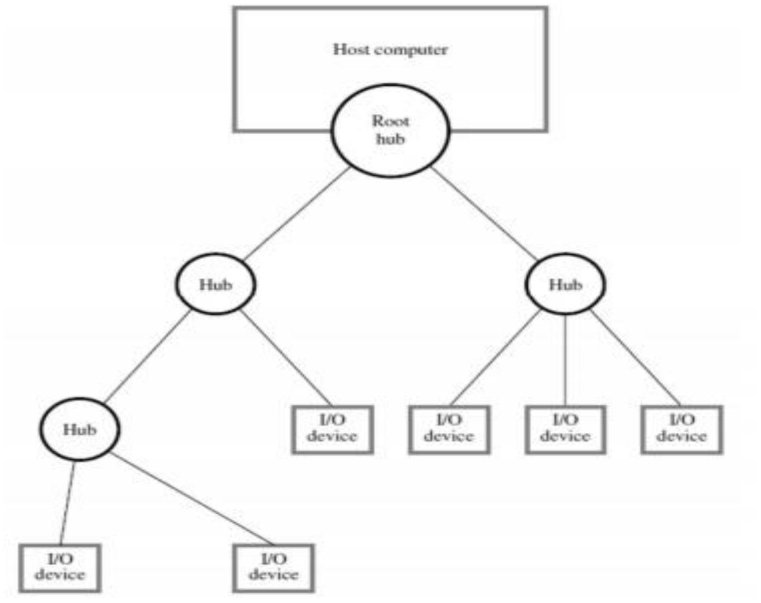
Plug-and-Play

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. 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. The tree structure makes it possible to connect many devices using simple point-to-point serial links.

Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the processor's address space. 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.

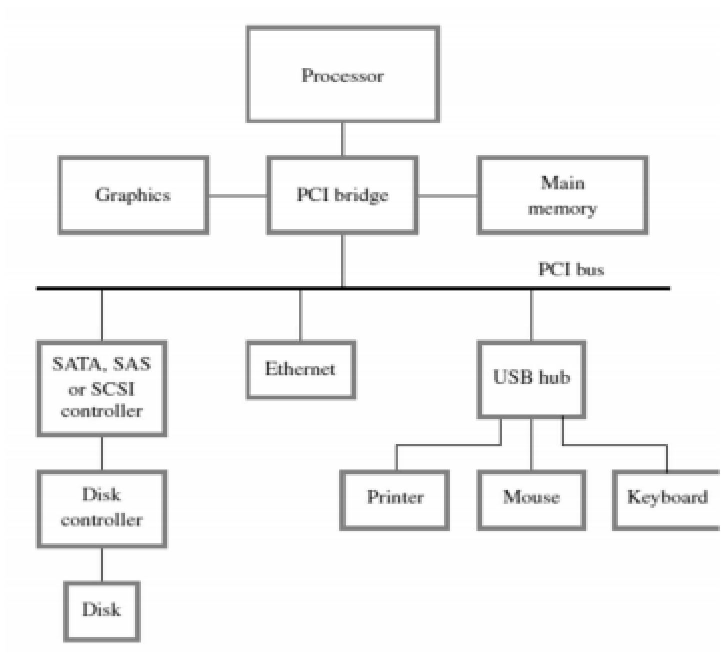


PCI BUS

The PCI (Peripheral Component Interconnect) bus was developed as a low-cost, processor-independent bus. It is housed on the motherboard of a computer and used to connect I/O interfaces for a wide variety of devices. A device connected to the PCI bus appears to the processor as if it is connected directly to the processor bus. Its interface registers are assigned addresses in the address space of the processor. We will start by describing how the PCI bus operates, then discuss some of its features.

Bus Structure:

The PCI bus is connected to the processor bus via a controller called a bridge. The bridge has a special port for connecting the computer's main memory. It may also have another special high speed port for connecting graphics devices. The bridge translates and relays commands and responses from one bus to the other and transfers data between them. For example, when the processor sends a Read request to an I/O device, the bridge forwards the command and address to the PCI bus. When the bridge receives the device's response, it forwards the data to the processor using the processor bus. I/O devices are connected to the PCI bus, possibly through ports that use standards such as Ethernet, USB, SATA, SCSI, or SAS. The PCI bus supports three independent address spaces: memory, I/O, and configuration.



The PCI bus is designed primarily to support multiple-word transfers. The PCI bus uses the same lines to transfer both address and data. In Figure, we assumed that the master maintains the address information on the bus until the data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected, freeing the lines for sending data in subsequent clock cycles. For transfers involving multiple words, the slave can store the address in an internal register and increment it to access successive address locations. A significant cost reduction can be realized in this manner, because the number of bus lines is an important factor affecting the cost of a computer system.

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.