Module 06 : Input / Output

In this Module, we have four lectures, viz.

Click the proper link on the left side for the lectures

Input/Output Organization

- The computer system's *input/output* (I/O) architecture is its interface to the outside world.
- Till now we have discussed the two important modules of the computer system -
  - **The processor** and
  - **The memory** module.
- The third key component of a computer system is a set of **I/O modules**
- Each I/O module interfaces to the system bus and controls one or more peripheral devices.

Introduction to I/O

There are several reasons why an I/O device or peripheral device is not directly connected to the system bus. Some of them are as follows -

- There are a wide variety of peripherals with various methods of operation. It would be impractical to include the necessary logic within the processor to control several devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.

Thus, an I/O module is required.

Input/Output Modules

The major functions of an I/O module are categorized as follows –

- Control and timing
- Processor Communication
- Device Communication
- Data Buffering
- Error Detection

During any period of time, the processor may communicate with one or more external devices in unpredictable manner, depending on the program's need for I/O.

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O.

Control & timings:

The I/O function includes a control and timing requirement to co-ordinate the flow of traffic between internal resources and external devices.

For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps –

a. The processor interacts with the I/O module to check the status of the attached device.
b. The I/O module returns the device status.
c. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
d. The I/O module obtains a unit of data from external device.
e. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

Processor & Device Communication

During the I/O operation, the I/O module must communicate with the processor and with the external device.

Processor communication involves the following -

Command decoding :

The I/O module accepts command from the processor, typically sent as signals on control bus.

Data :

Data are exchanged betweeen the processor and the I/O module over the data bus.

Status Reporting :

Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor(read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are ***BUSY*** and ***READY***.

Address Recognition :

Just as each word of memory has an address, so thus each of the I/O devices. Thus an I/O module must recognize one unique address for each peripheral it controls.

One the other hand, the I/O must be able to perform device communication. This communication involves command, status information and data.

Introduction to I/O

Data Buffering:

An essential task of an I/O module is data buffering. The data buffering is required due to the mismatch of the speed of CPU, memory and other peripheral devices. In general, the speed of CPU is higher than the speed of the other peripheral devices. So, the I/O modules store the data in a data buffer and regulate the transfer of data as per the speed of the devices.

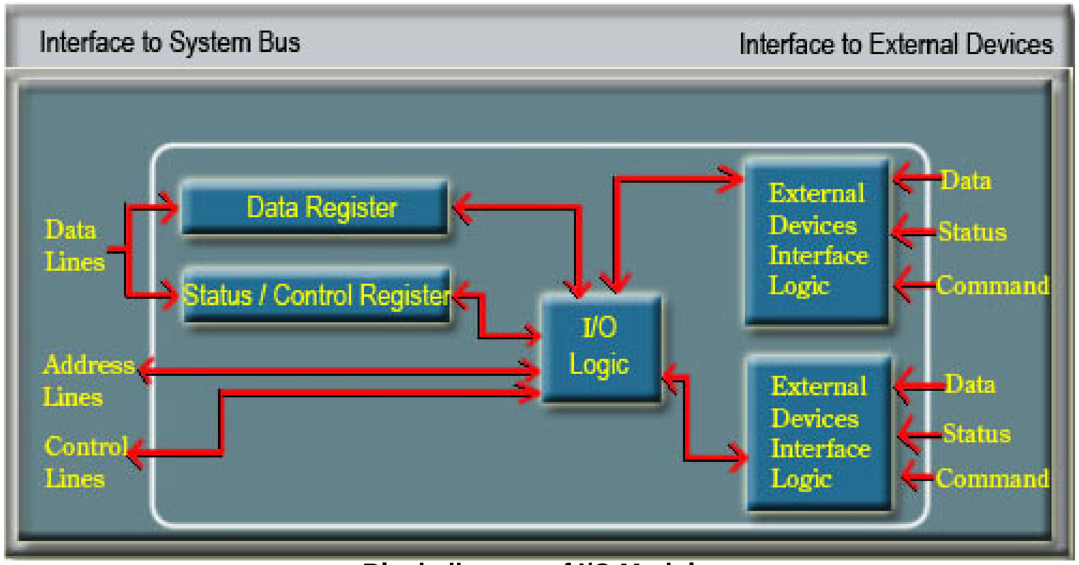
In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus the I/O module must be able to operate at both device and memory speed.

Error Detection:

Another task of I/O module is error detection and for subsequently reporting error to the processor. One class or error includes mechanical and electrical malfunctions reported by the device (e.g. paper jam). Another class consists of unintentional changes to the bit pattern as it is transmitted from devices to the I/O module.

Introduction to I/O

Block diagram of I/O Module is shown in the figure.


Interface to System Bus
Interface to External Devices
Data Lines
Address Lines
Control Lines
Data Register
Status / Control Register
I/O Logic
External Devices Interface Logic
Data
Status
Command
External Devices Interface Logic
Data
Status
Command

Block diagram of I/O Module.

Introduction to I/O

Print this page

<< Previous | First | Last | Next >>

There will be many I/O devices connected through I/O modules to the system. Each device will be indentified by a unique address.

When the processor issues an I/O command, the command contains the address of the device that is used by the command. The I/O module must interpret the addres lines to check if the command is for itself.

Generally in most of the processors, the processor, main memory and I/O share a common bus(data address and control bus).

Two types of addressing are possible -

- Memory-mapped I/O
- Isolated or I/O mapped I/O

<< Previous | First | Last | Next >>

Memory-mapped I/O:

There is a single address space for memory locations and I/O devices.

The processor treats the status and address register of the I/O modules as memory location.

For example, if the size of address bus of a processor is 16, then there are 2^{16} combinations and all together 2^{16} address locations can be addressed with these 16 address lines.

Out of these 2^{16} address locations, some address locations can be used to address I/O devices and other locations are used to address memory locations.

Since I/O devices are included in the same memory address space, so the status and address registers of I/O modules are treated as memory location by the processor. Therefore, the same machine instructions are used to access both memory and I/O devices.

Isolated or I/O -mapped I/O:

In this scheme, the full range of addresses may be available for both.

The address refers to a memory location or an I/O device is specified with the help of a command line.

In general $IO/\overline{M}$ command line is used to identify a memory location or an I/O device.

if $IO/\overline{M}$ =1, it indicates that the address present in address bus is the address of an I/O device.

if $IO/\overline{M}$ =0, it indicates that the address present in address bus is the address of a memory location.

Since full range of address is available for both memory and I/O devices, so, with 16 address lines, the system may now support both 2^{16} memory locations and 2^{16} I/O addresses.

Input / Output Subsystem

There are three basic forms of input and output systems –

- **Programmed I/O**
- **Interrupt driven I/O**
- **Direct Memory Access(DMA)**

With programmed I/O, the processor executes a program that gives its direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.

With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the I/O module completes its work.

In Direct Memory Access (DMA), the I/O module and main memory exchange data directly without processor involvement.

Program controlled I/O

With both programmed I/O and Interrupt driven I/O, the processor is responsible for extracting data from main memory for output operation and storing data in main memory for input operation.

To send data to an output device, the CPU simply moves that data to a *special memory location* in the I/O address space if I/O mapped input/output is used or to an address in the memory address space if memory mapped I/O is used.

Data	I/O Address Space (in memory)	if I/O mapped input/output is used
	memory address space	if memory mapped I/O is used

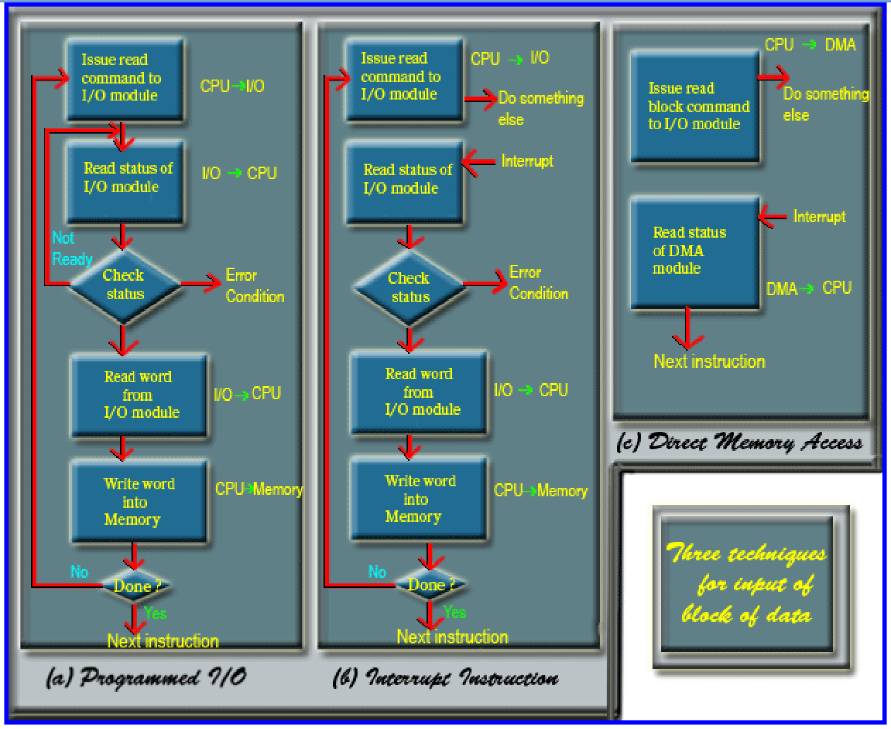
To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU.

Input/Output Operation: The input and output operation looks very similar to a memory read or write operation except it usually takes *more time* since peripheral devices are slow in speed than main memory modules.

Program controlled I/O

Print this page

<< Previous | First | Last | Next >>



Click on Image To View Large Image

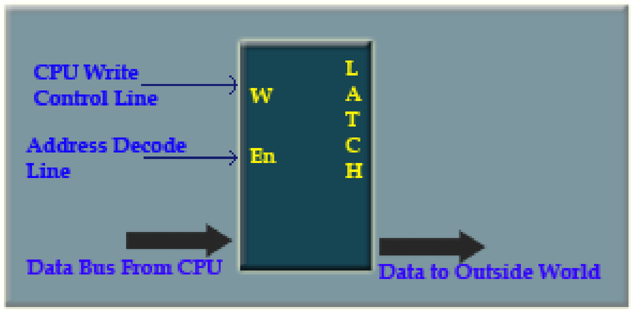
<< Previous | First | Last | Next >>

Input/Output Port

An *I/O port* is a device that looks like a memory cell to the computer but contains connection to the outside world.

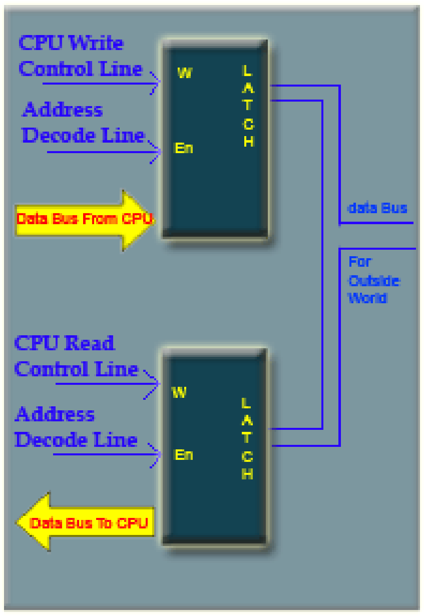
An *I/O port* typically uses a *latch*. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system.

The *I/O ports* can be *read-only*, *write-only*, or *read/write*. The *write-only* port is shown in the figure.



Program controlled I/O

First, the CPU will place the address of the device on the *I/O address bus* and with the help of *address decoder* a signal is generated which will enable the latch.



Next, the CPU will indicate the operation is a write operation by putting the appropriate signal in CPU write control line.

Then the data to be transferred will be placed in the CPU bus, which will be stored in the latch for the onward transmission to the device.

Both the address decode and write control lines must be active for the latch to operate.

The *read/write* or *input/output* port is shown in the figure.

The device is identified by putting the appropriate address in the I/O address lines. The address decoder will generate the signal for the address decode lines. According to the operation, *read* or *write*, it will select either of the latch.

If it is a write operation, then data will be placed in the latch from CPU for onward transmission to the output device.

Program controlled I/O

Print this page

<< Previous | First | Last | Next >>

If it is in a read operation, the data that are already stored in the latch will be transferred to the CPU.

A *read only* (input) port is simply the lower half of the figure.

In case of I/O mapped I/O, a different address space is used for I/O devices. The address space for memory is different. In case of memory mapped I/O, same address space is used for both memory and I/O devices. Some of the memory address space are kept reserved for I/O devices.

To the programmer, the difference between I/O-mapped and memory-mapped input/output operation is the instruction to be used.

For memory-mapped I/O, any instruction that accessed memory can access a memory-mapped I/O port.

I/O-mapped input/output uses special instruction to access I/O port.

<< Previous | First | Last | Next >>

Program controlled I/O

Generally, a given peripheral device will use more than a single I/O port. A typical *PC* parallel printer interface, for example, uses three ports, a *read/write port*, and *input port* and an *output port*.

The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port).

The input port returns control signals from the printer.
- These signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc.

The output port transmits control information to the printer such as
- whether data is available to print.

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory.

For example, to input a sequence of 20 bytes from an input port and store these bytes into memory, the CPU must send each value and store it into memory.

Programmed I/O:

In programmed I/O, the data transfer between CPU and I/O device is carried out with the help of a software routine.

When a processor is executing a program and encounters an instruction relating to I/O, it executes that I/O instruction by issuing a command to the appropriate I/O module.

The I/O module will perform the requested action and then set the appropriate bits in the I/O status register.

The I/O module takes no further action to alert the processor.

It is the responsibility of the processor to check periodically the status of the I/O module until it finds that the operation is complete.

In programmed I/O, when the processor issuses a command to a I/O module, it must wait until the I/O operation is complete.

Generally, the I/O devices are slower than the processor, so in this scheme CPU time is wasted. CPU is checking the status of the I/O module periodically without doing any other work.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module will receive when it is addressed by a processor –

- **Control :** Used to activate a peripheral device and instruct it what to do. For example, a magnetic tape unit may be instructed to rewind or to move forward one record. These commands are specific to a particular type of peripheral device.
- **Test :** Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know if the most recent I/O operation is completed or any error has occurred.
- **Read :** Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer.
- **Write :** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit the data item to the peripheral.

Interrupt driven I/O

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module.

This type of I/O operation, where the CPU constantly tests a part to see if data is available, is polling, that is, the CPU Polls (asks) the port if it has data available or if it is capable of accepting data. Polled I/O is inherently inefficient.

The solution to this problem is to provide an interrupt mechanism. In this approach the processor issues an I/O command to a module and then go on to do some other useful work. The I/O module then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer. Once the data transfer is over, the processor then resumes its former processing.

Let us consider how it works

A. From the point of view of the I/O module:

- For input, the I/O module services a READ command from the processor.
- The I/O module then proceeds to read data from an associated peripheral device.
- Once the data are in the modules data register, the module issues an interrupt to the processor over a control line.
- The module then waits until its data are requested by the processor.
- When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

B. From the processor point of view; the action for an input is as follows: :

- The processor issues a READ command.
- It then does something else
  (e.g. the processor may be working on several different programs at the same time)
- At the end of each instruction cycle, the processor checks for interrupts
- When the interrupt from an I/O module occurs, the processor saves the context
  (e.g. program counter & processor registers) of the current program and processes the interrupt.
- In this case, the processor reads the word of data from the I/O module and stores it in memory.
- It then restores the context of the program it was working on and resumes execution.

Interrupt Processing

The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.

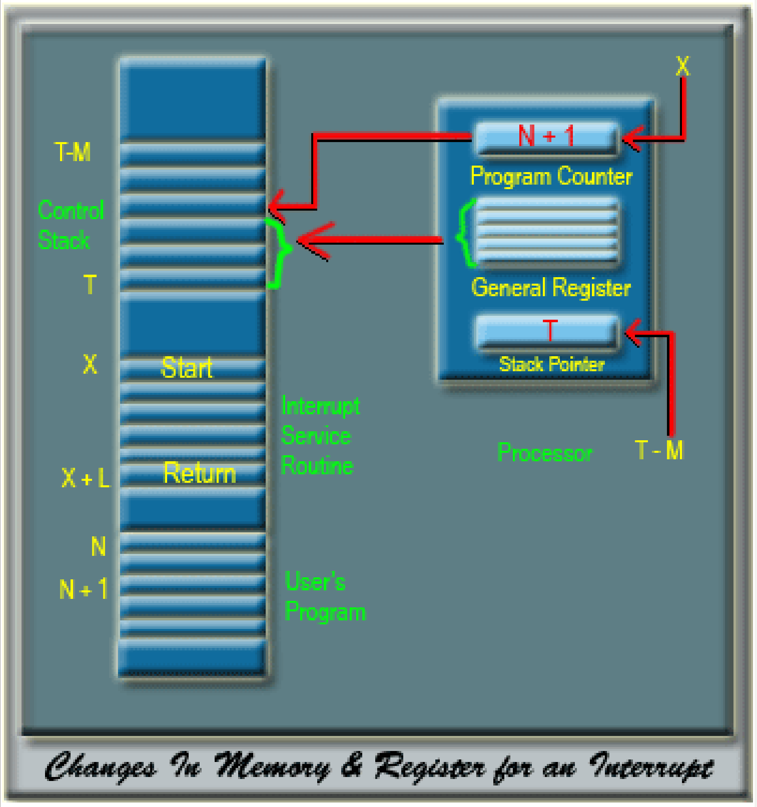
When an I/O device completes an I/O operation, the following sequences of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for the interrupt; if there is one interrupt pending, then the processor sends an acknowledgement signal to the device which issued the interrupt. After getting acknowledgement, the device removes its interrupt signals.
4. The processor now needs to prepare to transfer control to the interrupt routine. It needs to save the information needed to resume the current program at the point of interrupt. The minimum information required to save is the processor status word (PSW) and the location of the next instruction to be executed which is nothing but the contents of program counter. These can be pushed into the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt handling program that will respond to the interrupt.

Interrupt Control I/O

Print this page

<< Previous | First | Last | Next >>



Interrupt Processing:

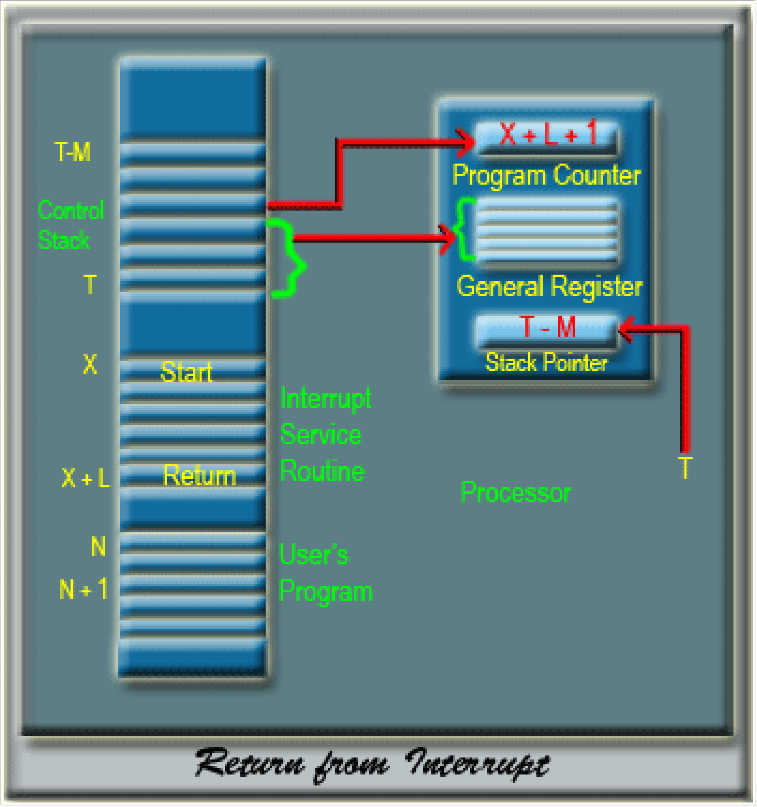
- An interrupt occurs when the processor is executing the instruction of location *N*.
- At that point, the value of program counter is *N+1* .
- Processor services the interrupt after completion of current instruction execution.
- First, it moves the content of general registers to system stack.
- Then it moves the program counter value to the system stack.
- Top of the system stack is maintained by stack pointer.
- The value of stack pointer is modified to point to the top of the stack.
- If M elsments are moved to the system stack, the value of stack pointer is changed from *T* to *T-M*.

<< Previous | First | Last | Next >>

Interrupt Control I/O

- Next, the program counter is loaded with the starting address of the interrupt service routine.
- Processor starts executing the interrupt service routine.

Interrupt Control I/O



Return from Interrupt :

- Interrupt service routine starts at location *X* and the return instruction is in location *X* + *L*.
- After fetching the return instruction, the value of program counter becomes *X* + *L* + *1*.
- While returning to user's program, processor must restore the earlier values.
- From control stack, it restores the value of program counter and the general registers.
- Accordingly it sets the value of the top of the stack and accordingly stack pointer is updated.
- Now the processor starts execution of the user's program (interrupted program) from memory location *N* + *1*.

Interrupt Control I/O

Print this page

<< Previous | First | Last | Next >>

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an interrupt fetch. The control will transfer to interrupt handler routine for the current interrupt.

The following operations are performed at this point.

6. At the point, the program counter and PSW relating to the interrupted program have been saved on the system stack. In addition to that some more information must be saved related to the current processor state which includes the control of the processor registers, because these registers may be used by the interrupt handler. Typically, the interrupt handler will begin by saving the contents of all registers on stack.

7. The interrupt handles next processes the interrupt. This includes an examination of status information relating to the I/O operation or, other event that caused an interrupt.

8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.

9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

<< Previous | First | Last | Next >>

Design Issues for Interrupt

Two design issues arise in implementing interrupt I/O.

- There will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt?
- If multiple interrupts have occurred how the processor does decide which one to process?

Device Identification

Four general categories of techniques are in common use:

- **Multiple interrupt lines**
- **Software poll**
- **Daisy chain (hardware poll, vectored)**
- **Bus arbitration (vectored)**

Multiple Interrupts Lines:

The most straight forward approach is to provide multiple interrupt lines between the processor and the I/O modules.

It is impractical to dedicate more than a few bus lines or processor pins to interrupt lines.

Thus, though multiple interrupt lines are used, it is most likely that each line will have multiple I/O modules attached to it. Thus one of the other three techniques must be used on each line.

Software Poll :

When the processor detects an interrupt, it branches to an interrupt service routine whose job is to poll each I/O module to determine which module caused the interrupt.

The poll could be implemented with the help of a separate command line (e.g. TEST I/O). In this case, the processor raises TEST I/O and place the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt.

Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module.

Once the correct module is identified, the processor branches to a device service routine specific to that device.

The main disadvantage of software poll is that it is time consuming. Processor has to check the status of each I/O module and in the worst case it is equal to the number of I/O modules.

Daisy Chain :

In this method for interrupts all I/O modules share a common interrupt request lines. However the interrupt acknowledge line is connected in a daisy chain fashion. When the processor senses an interrupt, it sends out an interrupt acknowledgement.

The interrupt acknowledge signal propagates through a series of I/O module until it gets to a requesting module.

The requesting module typically responds by placing a word on the data lines. This word is referred to as a vector and is either the address of the I/O module or some other unique identification.

In either case, the processor uses the vector as a pointer to the appropriate device service routine. This avoids the need to execute a general interrupt service routine first. This technique is referred to as a *vectored interrupt*.

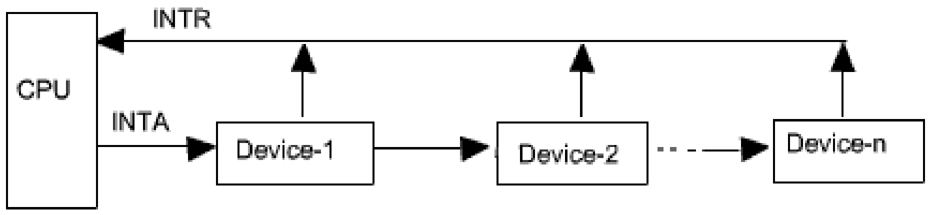


Fig : Daisy chain arrangement

Bus Arbitration :

In bus arbitration method, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the interrupt line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places it vector on the data line.

Handling multiple interrupts

There are several techniques to identify the requesting I/O module. These techniques also provide a way of assigning priorities when more than one device is requesting interrupt service.

With multiple lines, the processor just picks the interrupt line with highest priority. During the processor design phase itself priorities may be assigned to each interrupt lines.

With software polling, the order in which modules are polled determines their priority.

In case of daisy chain configuration, the priority of a module is determined by the position of the module in the daisy chain. The module nearer to the processor in the chain has got higher priority, because this is the first module to receive the acknowledge signal that is generated by the processor.

In case of bus arbitration method, more than one module may need control of the bus. Since only one module at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be classified into two group – centralized and distributed.

Interrupt Control I/O

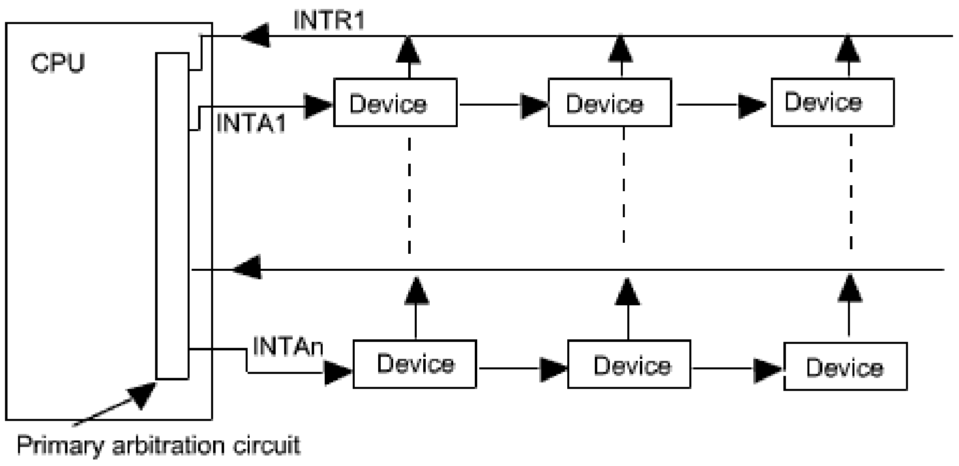
In a centralized scheme, a single hardware device, referred to as a bus controller or arbiter is responsible for allocating time on the bus. The device may be a separate module or part of the processor.

In distributed scheme, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus.

It is also possible to combine different device identification techniques to identify the devices and to set the priorities of the devices. As for example multiple interrupt lines and daisy chain technologies can be combined together to give access for more devices.

In one interrupt line, more than one device can be connected in daisy chain fashion. The High priorities devices should be connected to the interrupt lines that has got higher priority.

A possible arrangement is shown in the figure.



Interrupt Nesting

The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and starts the execution of another. The execution of this another program is nothing but the interrupt service routine for that specified device.

Interrupt may arrive at any time. So during the execution of an interrupt service routine, another interrupt may arrive. This kind of interrupts are known as *nesting of interrupt*.

Whether interrupt nesting is allowed or not? This is a design issue. Generally nesting of interrupt is allowed, but with some restrictions. The common notion is that a high priority device may interrupt a low priority device, but not the vice-versa.

To accomodate such type of restrictions, all computer provide the programmer with the ability to enable and disable such interruptions at various time during program execution. The processor provides some instructions to enable the interrupt and disable the interrupt. If interrupt is disabled, the CPU will not respond to any interrupt signal.

On the other hand, when multiple lines are used for interrupt and priorities are assigned to these lines, then the interrupt received in a low priority line will not be served if an interrupt routine is in execution for a high priority device. After completion of the interrupt service routine of high priority devices, processor will respond to the interrupt request of low priority devices

Direct Memory Access

We have discussed the data transfer between the processor and I/O devices. We have discussed two different approaches namely *programmed I/O* and *Interrupt-driven I/O*. Both the methods require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

To transfer large block of data at high speed, 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 associated with the I/O device and this circuit is referred as DMA controller. The DMA controller allows direct data transfer between the device and the main memory without involving the processor.

Direct Memory Access [DMA]

To transfer data between memory and I/O devices, DMA controller takes over the control of the system from the processor and transfer of data take place over the system bus. For this purpose, the DMA controller must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The later technique is more common and is referred to as cycle stealing, because the DMA module in effect steals a bus cycle.

The typical block diagram of a DMA controller is shown in the figure.

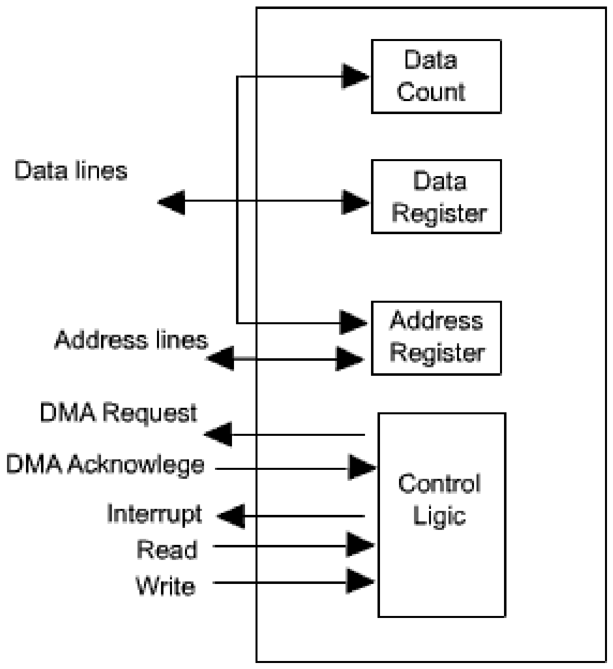


Fig : Typical DMA block diagram

Direct Memory Access [DMA]

Print this page

<< Previous | First | Last | Next >>

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information.

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module.
- The address of the I/O devise involved, communicated on the data lines.
- The starting location in the memory to read from or write to, communicated on data lines and stored by the DMA module in its address register.
- The number of words to be read or written again communicated via the data lines and stored in the data count register.

The processor then continues with other works. It has delegated this I/O operation to the DMA module.

The DMA module checks the status of the I/O devise whose address is communicated to DMA controller by the processor. If the specified I/O devise is ready for data transfer, then DMA module generates the DMA request to the processor. Then the processor indicates the release of the system bus through DMA acknowledge.

The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.

<< Previous | First | Last | Next >>

Direct Memory Access [DMA]

When the transfer is completed, the DMA module sends an interrupt signal to the processor. After receiving the interrupt signal, processor takes over the system bus.

Thus the processor is involved only at the beginning and end of the transfer. During that time the processor is suspended.

It is not required to complete the current instruction to suspend the processor. The processor may be suspended just after the completion of the current bus cycle. On the other hand, the processor can be suspended just before the need of the system bus by the processor, because DMA controller is going to use the system bus, it will not use the processor.

The point where in the instruction cycle the processor may be suspended shown in the figure.

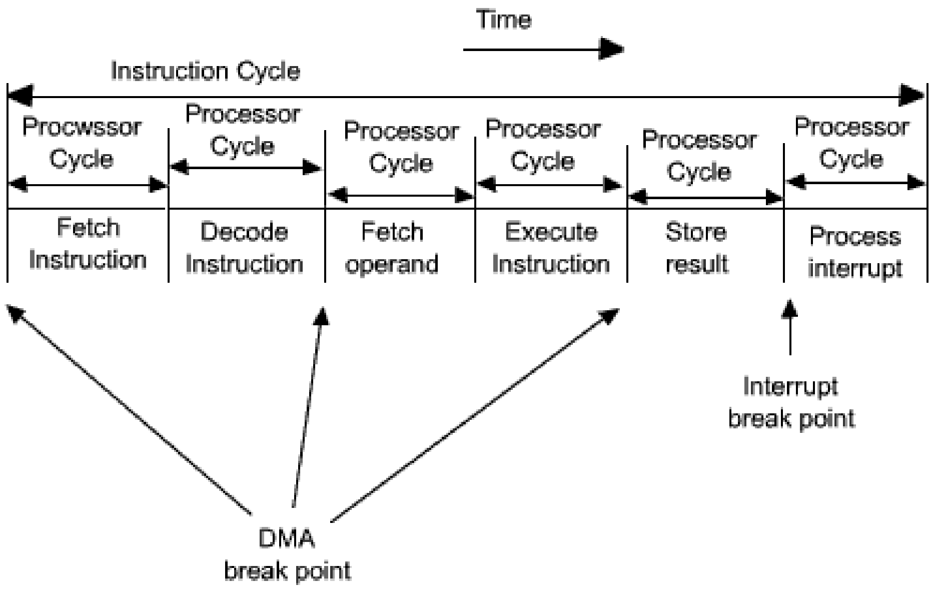


Fig : DMA break point

When the processor is suspended, then the DMA module transfer one word and return control to the processor.

Note that, this is not an interrupt, the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle.

During that time processor may perform some other task which does not involve the system bus. In the worst situation processor will wait for some time, till the DMA releases the bus.

The net effect is that the processor will go slow. But the net effect is the enhancement of performance, because for a multiple word I/O transfer, DMA is far more efficient than interrupt driven or programmed I/O.

Direct Memory Access [DMA]

The DMA mechanism can be configured in different ways. The most common amongst them are:

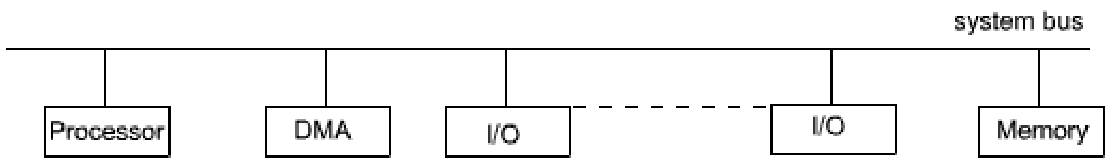
- **Single bus, detached DMA - I/O configuration.**
- **Single bus, Integrated DMA - I/O configuration.**
- **Using separate I/O bus.**

Single bus, detached DMA - I/O configuration

In this organization all modules share the same system bus.The DMA module here acts as a surrogate processor. This method uses programmed I/O to exchange data between memory and an I/O module through the DMA module.

For each transfer it uses the bus twice. The first one is when transferring the data between I/O and DMA and the second one is when transferring the data between DMA and memory. Since the bus is used twice while transferring data, so the bus will be suspended twice. The transfer consumes two bus cycle.

The interconnection organization is shown in the figure.



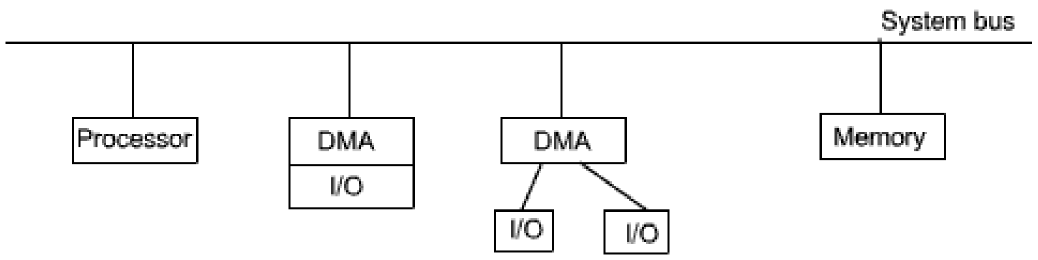
Single bus, Integrated DMA - I/O configuration

By integrating the DMA and I/O function the number of required bus cycle can be reduced. In this configuration, the DMA module and one or more I/O modules are integrated together in such a way that the system bus is not involved. In this case DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.

The DMA module, processor and the memory module are connected through the system bus. In this configuration each transfer will use the system bus only once and so the processor is suspended only once.

The system bus is not involved when transferring data between DMA and I/O device, so processor is not suspended. Processor is suspended when data is transferred between DMA and memory.

The configuration is shown in the figure.



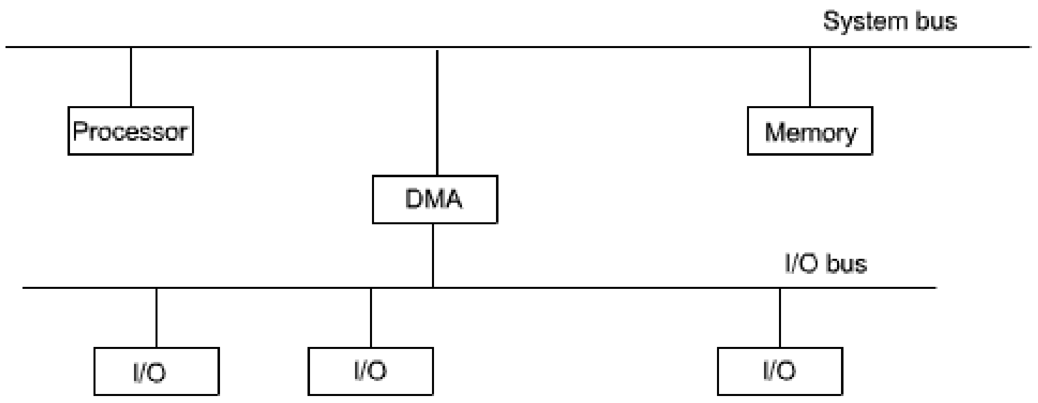
Direct Memory Access [DMA] Print this page

Using separate I/O bus

In this configuration the I/O modules are connected to the DMA through another I/O bus. In this case the DMA module is reduced to one.

Transfer of data between I/O module and DMA module is carried out through this I/O bus. In this transfer, system bus is not in use and so it is not needed to suspend the processor.

There is another transfer phase between DMA module and memory. In this time system bus is needed for transfer and processor will be suspended for one bus cycle. The configuration is shown in the figure.



Module 07 : Connecting I/O Devices

In this Module, we have two lectures, viz.

Click the proper link on the left side for the lectures

Buses

The *processor*, *main memory*, and *I/O devices* can be interconnected through common data communication lines which are termed as *common bus.*

The primary function of a common bus is to provide a communication path between the devices for the transfer of data. The bus includes the control lines needed to support interrupts and arbitration.

The bus lines used for transferring data may be grouped into three categories:

- data,
- address
- control lines.

A single $R/\overline{W}$ line is used to indicate Read or Write operation. When several sizes are possible like byte, word, or long word, control signals are required to indicate the size of data.

The bus control signal also carry timing information to specify the times at which the processor and the I/O devices may place data on the bus or receive data from the bus.

There are several schemes exist for handling the timing of data transfer over a bus. These can be broadly classified as

- Synchronous bus
- Asynchronous bus

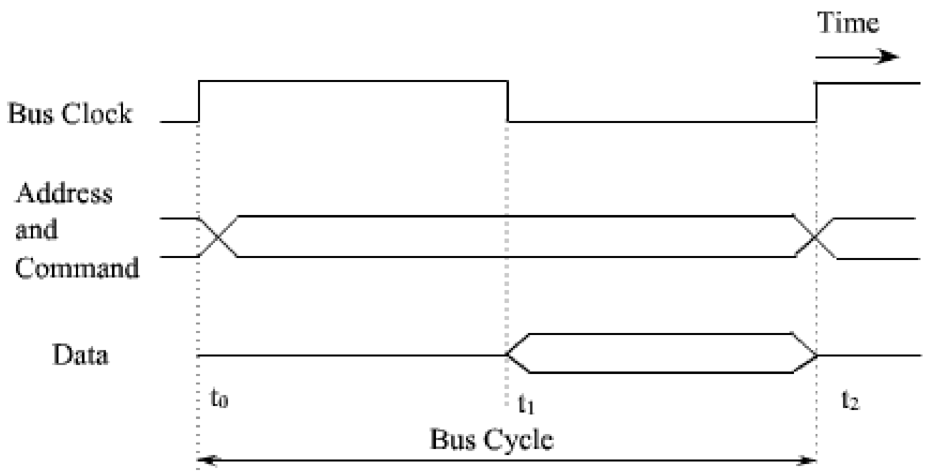
Synchronous Bus :

In a synchronous bus, all the devices are synchronised by a common clock, so all devices derive timing information from a common clock line of the bus. A clock pulse on this common clock line defines equal time intervals.

In the simplest form of a synchronous bus, each of these clock pulse constitutes a bus cycle during which one data transfer can take place.

The timing of an input transfer on a synchronous bus is shown in the figure (Next Page..).

Buses



Timing of an input transfer on a synchronous bus

Buses

Let us consider the sequence of events during an input (read) operation.

At time t_0, the master places the device address on the address lines and sends an appropriate command (read in case of input) on the command lines.

In any data transfer operation, one device plays the role of a master, which initiates data transfer by issuing read or write commands on the bus.

Normally, the processor acts as the master, but other device with DMA capability may also becomes bus master. The device addressed by the master is referred to as a slave or target device.

The command also indicates the length of the operand to be read, if necessary.

The clock pulse width, $t_1 - t_0$, must be longer than the maximum propagation delay between two devices connected to the bus.

After decoding the information on address and control lines by slave, the slave device of that particular address responds at time t_1. The addressed slave device places the required input data on the data line at time t_1.

At the end of the clock cycle, at time t_2, the master strobes the data on the data lines into its input buffer. The period $t_2 - t_1$ must be greater than the maximum propagation delay on the bus plus the set up time of the input buffer register of the master.

A similar procedure is followed for an output operation. The master places the output data on the data lines when it transmits the address and command information. At time t_2, the addressed device strobe the data lines and load the data into its data buffer.

Multiple Cycle Transfer

The simple design of device interface by synchronous bus has some limitations.

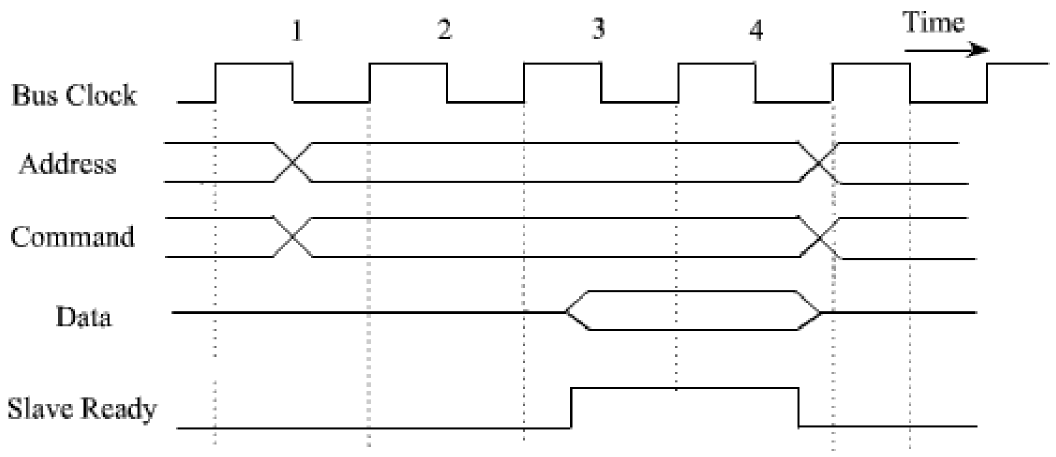
- A transfer has to be completed within one clock cycle. The clock period, must be long enough to accommodate the slowest device to interface. This forces all devices to operate at the speed of slowest device.
- The processor or the master has no way to determine whether the addressed device has actually responded. It simply assumes that, the output data have been received by the device or the input data are available on the data lines.

To solve these problems, most buses incorporate control signals that represent a response from the device. These signals inform the master that the target device has recognized its address and it is ready to participate in the data transfer operation.

They also adjust the duration of the data transfer period to suit the needs of the participating devices.

A high frequency clock pulse is used so that a complete data transfer operation span over several clock cycles. The numbers of clock cycles involved can vary from device to device

An instance of this scheme is shown in the figure.



An input transfer using multiple clock cycle

In *clock cycle 1*, the master sends address and command on the bus requesting a read operation.

The target device responded at *clock cycle 3* by indicating that it is ready to participate in the data transfer by making the slave ready signal high.

Then the target device places the data on the data line.

The target device is a slower device and it needs two clock cycle to transfer the information. After two clock cycle, that is at *clock cycle 5*, it pulls down the slave ready signal down.

When the slave ready signal goes down, the master strobes the data from the data bus into its input buffer.

If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, then aborts the operation.

Asynchronous Bus

In asynchronous mode of transfer, a *handshake signal* is used between *master* and *slave*.

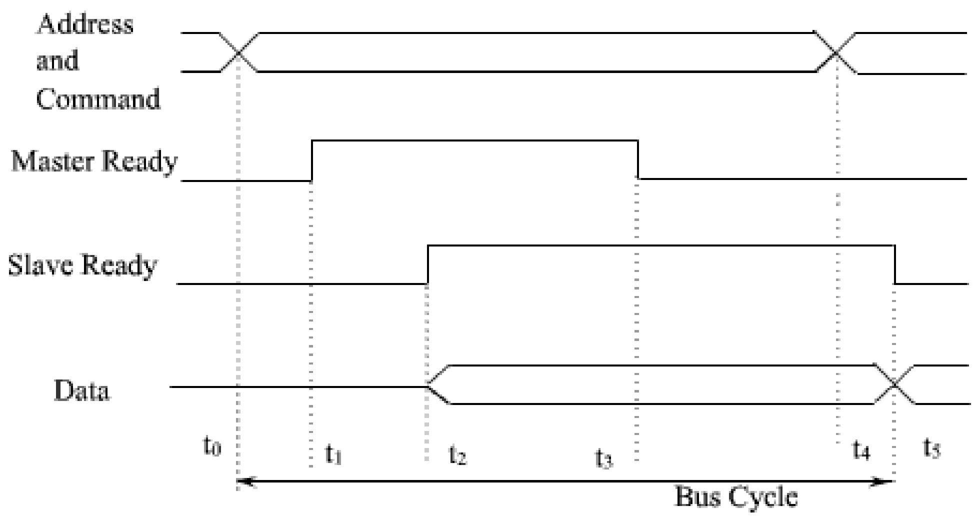
> In *asynchronous bus*, there is no common clock, and the common clock signal is replaced by two timing control signals: *master-ready* and *slave-ready*.
>
> Master-ready signal is assured by the master to indicate that it is ready for a transaction, and slave-ready signal is a response from the slave.

The handshaking protocol proceeds as follows:

- The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the master-ready signal.
- This causes all devices on the bus to decode the address.
- The selected target device performs the required operation and inform the processor (or master) by activating the slave-ready line.
- The master waits for slave-ready to become asserted before it remove its signals from the bus.
- In case of a read operation, it also strobes the data into its input buffer.

Buses

The timing of an input data transfer using the handshake scheme is shown in the figure.



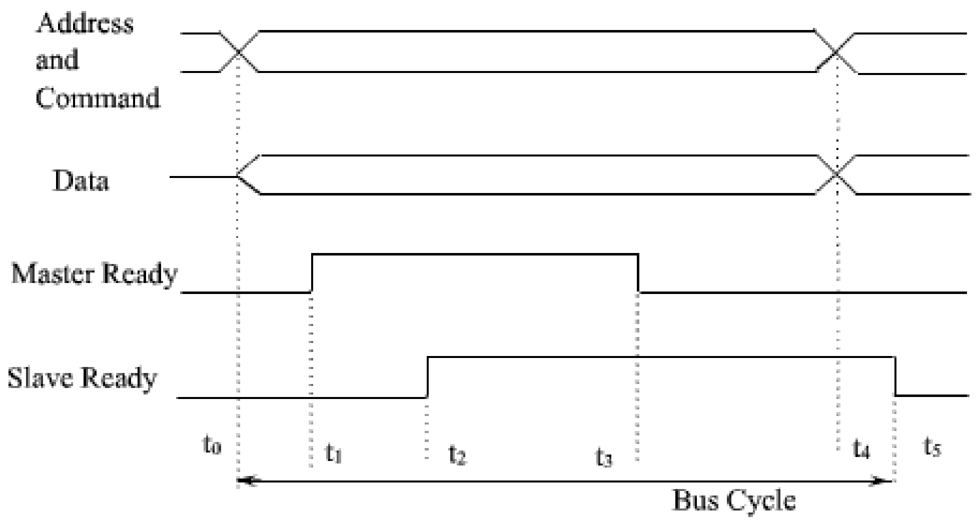
Handshake control of data transfer during an input operation

Buses

Print this page

<< Previous | First | Last | Next >>

The timing of an output operation using handshaking scheme is shown in the figure.



Handshake control of data transfer during an output operation

<< Previous | First | Last | Next >>

External Memory

Main memory is taking an important role in the working of computer. We have seen that computer works on *Von-Neuman* stored program principle. We have to keep the information in main memory and CPU access the information from main memory.

The main memory is made up of semiconductor device and by nature it is volatile. For permanent storage of information we need some non volatile memory. The memory devices need to store information permanently are termed as *external memory*. While working, the information will be transferred from external memory to main memory.

The devices need to store information permanently are either magnetic or optical devices.

Magnetic Devices:

- Magnetic disk (Hard disk)
- Floopy disk
- Magnetic tape

Optical Devices:

- CD- ROM
- CD-Recordable(CD –R)
- CD-R/W
- DVD

Magnetic Disk

A disk is a *circular platter* constructed of metal or of plastic coated with a magnetizable material.

Data are recorded on and later retrieved from the disk via a conducting coil named the *head*.

During a read or write operation, the head is stationary while the platter rotates beneath it.

The *write mechanism* is based on the fact that electricity flowing through a coil produces a magnetic field.Pulses are sent to the head, and magnetic patterns are recorded on the surface below. The pattern depends on the *positive* or *negative* currents. The direction of current depends on the information stored , i.e., positive current depends on the information '1' and negative current for information '0'.

The *read mechanism* is based on the fact that a magnetic field moving relative to a coil produces on electric current in the coil. When the surface of the disk passes under the head, it generates a current of the same polarity as the one already recorded.

Read/ Write head detail is shown in the figure:

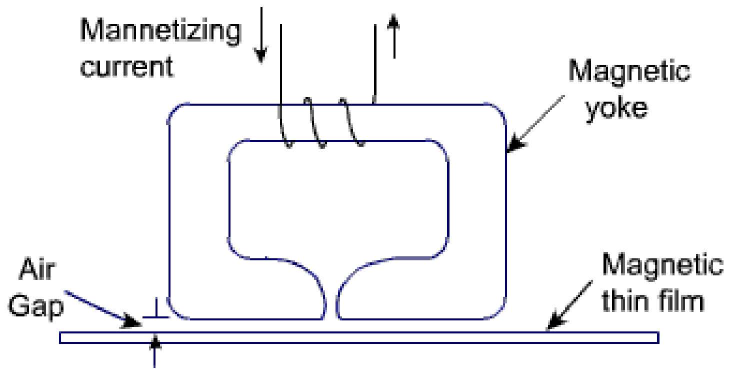


Figure : Read/Write head detail

The head is a relatively small device capable of *reading from* or *writing to* a portion of the platter rotating beneath it.

The data on the disk are organized in a concentric set of rings, called *track*. Each track has the same width as the head. Adjacent tracks are separated by gaps. This prevents error due to misalignment of the head or interference of magnetic fields.

For simplifying the control circuitry, the same number of bits are stored on each track. Thus the density, in bits per linear inch, increases in moving from the outermost track to the innermost track.

Data are transferred to and from the disk in blocks. Usually, the block is smaller than the capacity of the track. Accordingly, data are stored in block-size regions known as sector.

A typical disk layout is shown in the figure:

To avoid, imposition unreasonable precision requirements on the system, adjacent tracks (sectors) are separated by *intratrack* (intersector) gaps.

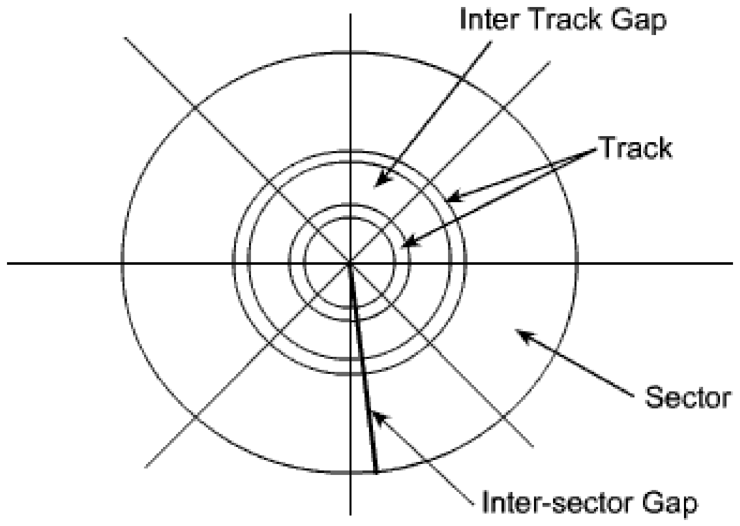
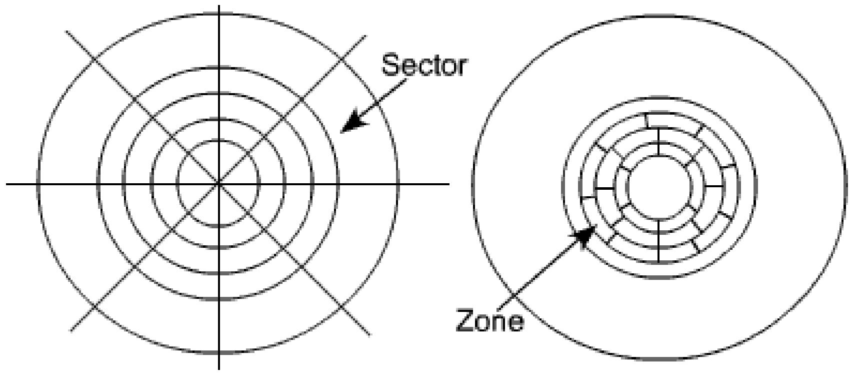


Figure : Disk Data Layout

Some means are needed to locate sector positions within a track. Clearly there must be some starting points on the track and a way of identifying the *start* and *end* of each sector. These requirements are handled by means of a control data recorded on the disk. Thus, the disk is formatted with some *extra data* used only by the disk drive and not accessible to the user.

Since data density in the outermost track is less and data density is more in inner tracks so there are wastage of space on outer tracks.

To increase the capacity, the concept of *zone* is used instead of *sectors*. Each track is divided in zone of equal length and fix amount of data is stored in each zone. So the number of zones are less in innermost track and number of zones are more in the outermost track. Therefore, more number of bits are stored in outermost track. The disk capacity is increasing due to the use of zone, but the complexity of control circuitry is also more.



Physical characteristics of disk

The head may be either fixed or movable with respect to the radial direction of the platter.

In a fixed-head disk, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks.

In a movable-head disk, there is only one read-write head. Again the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, the shaft that rotates the disk, and the electronics circuitry needed for input and output the binary data and to control the mechanism.

A non removable disk is permanently mounted on the disk drive. A removable disk can be removed and replaced with another disk.

For most disks, the magnetizable coating is applied to both sides of the platters, which is then referred to as double sided. If the magnetizable coating is applied to one side only, then it is termed as *single sided disk*.

Some disk drives accommodate multiple platters stacked vertically above one another. Multiple arms are provided for read write head. The platters come as a unit known as a disk pack.

The physical organization of disk is shown in the figure:

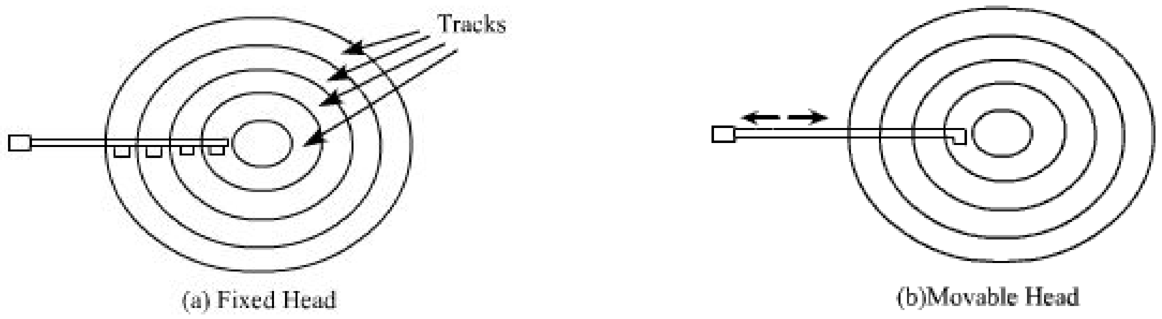


Figure: Fixed and Movable head disk

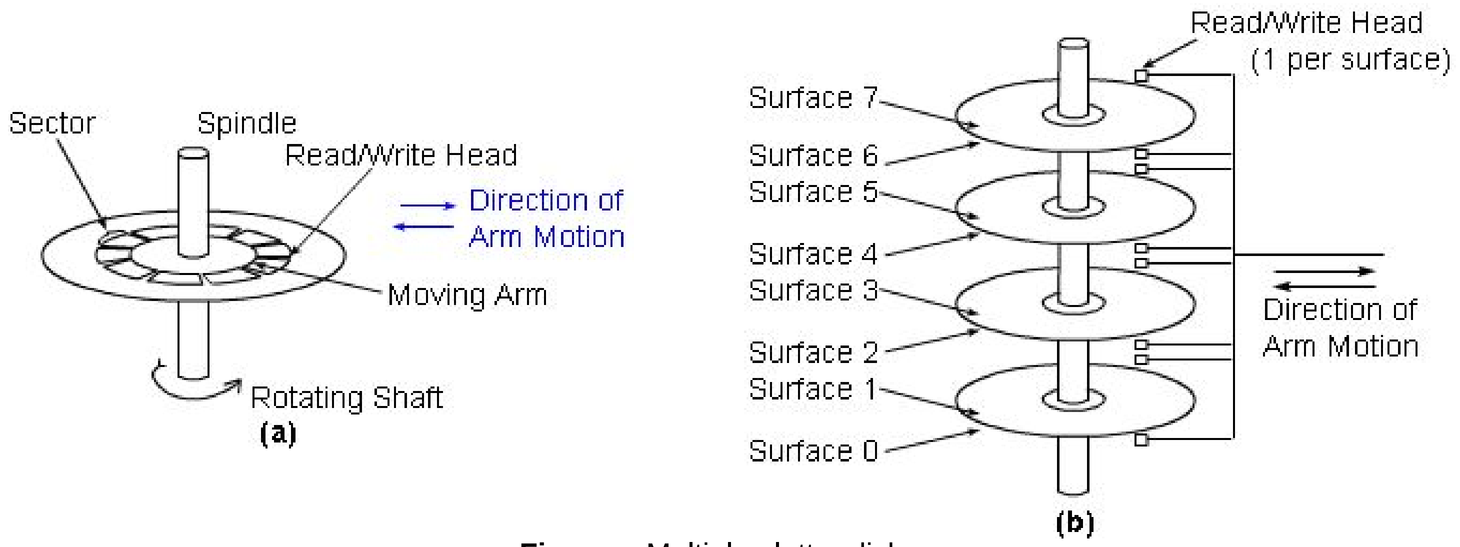
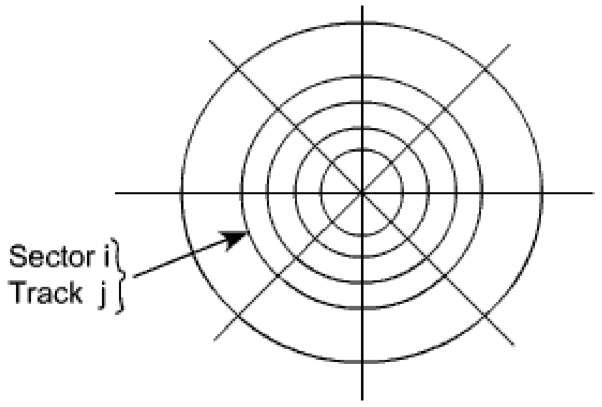


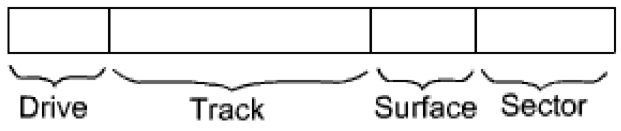
Figure: Multiple platter disk

Organization and accessing of data on a disk

The organization of data on a disk is shown in the figure below.



Organization of one surface of a disk



Each surface is divided into concentric tracks and each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disks form a logical cylinder. Data bits are stored serially on each track.

Data on disks are addressed by specifying the surface number, the track number, and the sector number.

In most disk systems, read and write operations always start at sector boundaries. If the number of words to be written is smaller than that required to fill a sector, the disk controller repeats the last bit of data for the remaining of the sector.

During read and write operation, it is required to specify the starting address of the sector from where the operation will start, that is the *read/write head* must positioned to the correct track, sector and surface. Therefore the address of the disk contains *track no*., *sector no*., and *surface no*. If more than one drive is present, then drive number must also be specified.

The format of the disk address word is shown in the figure. It contains the *drive no*, *track no*., *surface no*. and *sector no*.

The *read/write head* will first positioned to the correct track. In case of *fixed head system*, the correct head is selected by taking the *track no*. from the address. In case of *movable head system*, the head is moved so that it is positioned at the correct track.

External Memory

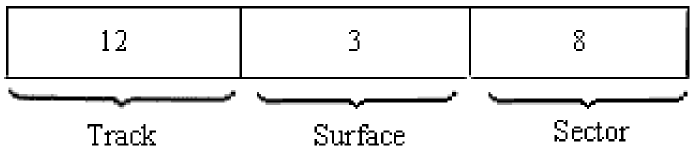
Print this page

<< Previous | First | Last | Next >>

By the surface no, it selects the correct surface.

To get the correct sector below the *read/write head*, the disk is rotated and bring the correct sector with the help of sector number. Once the correct sector, track and surface is decided, the *read/write operation* starts next.

Suppose that the disk system has 8 data recording surfaces with 4096 track per surface. Tracks are divided into 256 sectors. Then the format of disk address word is:



Suppose each sector of a track contains 512 bytes of disk recorded serially, then the total capacity of the disk is:

$$\begin{aligned} 8 \times 4096 \times 256 \times 512 &= 2^3 \times 2^{12} \times 2^8 \times 2^9 \\ &= 4 \text{ Gigabyte (GB)} \end{aligned}$$

<< Previous | First | Last | Next >>

External Memory

Print this page

<< Previous | First | Last | Next >>

For moving head system, there are two components involved in the time delay between receiving an address and the beginning of the actual data transfer.

Seek Time:

Seek time is the time required to move the read/write head to the proper track. This depends on the initial position of the head relative to the track specified in the address.

Rotational Delay:

Rotational delay, also called the *latency time* is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector comes under the Read/write head.

<< Previous | First | Last | Next >>

Disk Operation

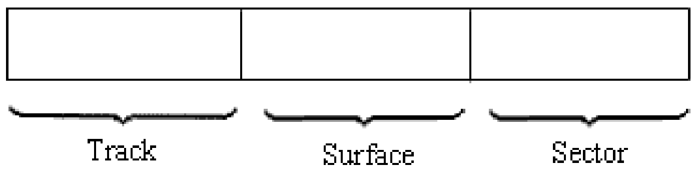
Communication between a disk and the main memory is done through DMA. The following information must be exchanged between the processor and the disk controller in order to specify a transfer.

Main memory address :
The address of the first main memory location of the block of words involved in the transfer.

Disk address :
The location of the sector containing the beginning of the the desired block of words.

Word count :
The number of words in the block to be transferred.

The disk address format is:



The word count may corresponds to fewer or more bytes than that are contained in a sector. When the data block is longer than a track:

The disk address register is incremented as successive sectors are read or written. When one track is completed then the surface count is incremented by 1.

Thus, long data blocks are laid out on cylinder surfaces as opposed to being laid out on successive tracks of a single disk surface.

This is efficient for moving head systems, because successive sector areas of data storage on the disk can be accessed by electrically switching from one *Read/Write head* to the next rather than by mechanically moving the arm from track to track.

The *track-to-track* movement is required only at *cylinder-to-cylinder* boundaries.

Disk Performance Parameter

When a disk drive is operating, the disk is rotating at constant speed.

To *read* or *write*, the head must be positioned at the desired tack and at the beginning of the desired *sector* on the *track*.

Track selection involves moving the head in a *movable-head system* or electronically selecting one head on a fixed head system.

On a *movable-head system*, the time taken to position the head at the track is known a *seek time*.

Once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes to reach the beginning of the desired sector is known as *rotational delay* or *rotational latency*.

The sum of the *seek time*, (for movable head system) and the *rotational delay* is termed as ***access time*** of the disk, the time it takes to get into appropriate position (track & sector) to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head, and the data transfer takes place.

Disk Performance Parameter

Print this page

<< Previous | First | Last | Next >>

Seek Time:

Seek time is the time required to move the disk arm to the required track. The seek time is approximated as

$$T_s = m \times n + s$$

where

T_s =	estimated seek time
n =	number of tracks traversed
m =	constant that depends on the disk drive
s =	startup time

Rotational Delay:

Disk drive generally rotates at 3600 *rpm*, i.e., to make one revolution it takes around 16.7 *ms*. Thus on the average, the rotational delay will be 8.3 *ms*.

<< Previous | First | Last | Next >>

Transfer Time:

The transfer time to or from the disk depends on the rotational speed of the disk and it is estimated as

$$T = \frac{b}{rN}$$

where

T =	Transfer time
b =	Number of bytes to be transferred.
N =	Numbers of bytes on a track
r =	Rotational speed, in revolution per second.

Thus,

the total ***average access time*** can be expressed as

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

where T_s is the average seek time.

Disk Performance Parameter

Print this page

<< Previous | First | Last | Next >>

Issues with disks:

Disks are *potential bottleneck* for system performances and storage system reliability.

The *disk access time* is relatively higher than the time required to access data from main memory and performs CPU operation. Also the disk drive contains some mechanical parts and it involves mechanical movement, so the failure rate is also high.

The disk performance has been improving continuously, microprocessor performance has improved much more rapidly.

A *disk array* is an arrangement of several disk, organized to increase performance and improve reliability of the resulting storage system. Performance is increased through *data striping*. Reliability is improved through *redundancy*.

Disk arrays that implement a combination of data striping and redundancy are called ***Redundant Arrays of Independent Disks*** (RAID).

<< Previous | First | Last | Next >>

Data Striping

In data striping, he data is segmented in equal-size partitions distributed over multiple disks. The size of the partition is called the *striping unit*.

The partitions are usually distributed using a *round-robin algorithm***:**
if the disk array consists of D *disks*, then partition i is written in to disk ($i \bmod D$).

Consider a striping unit equal to a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array.

If many I/O requests of the size of a disk block are made, and the requested blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request.

Since the striping unit are distributed over several disks in the disk array in round robin fashion, large I/O requests of the size of many continuous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate.

<< Previous | First | Last | Next >>

Redundancy

While having more disks increases storage system performance, it also lower overall storage system reliability, because the probability of failure of a disk in disk array is increasing.

Reliability of a disk array can be increased by storing redundant information. If a disk fails, the redundant information is used to reconstruct the data on the failed disk.

One design issue involves here - where to store the redundant information. There are two choices-either store the redundant information on a same number of check disks, or distribute the redundant information uniformly over all disk.

In a RAID system, the disk array is partitioned into reliability group, where a reliability group consists of a set of data disks and a set of check disks. A common redundancy scheme is applied to each group.

<< Previous | First | Last | Next >>

RAID levels

RAID Level 0 : Nonredundant

A **RAID level 0** system is not a true member of the RAID family, because it does not include redundancy, that is, no redundant information is maintained.

It uses data striping to increase the I/O performance.

For **RAID 0**, the user and system data are distributed across all of the disk in the array, i.e. data are striped across the available disk.

If two different I/O requests are there for two different data block, there is a good probability that the requested blocks are in different disks. Thus, the two requests can be issued in parallel, reducing the I/O waiting time.

RAID level 0 is a low cost solution, but the reliability is a problem since there is no redundant information to retrieve in case of disk failure.

RAID level 0 has the best write performance of all RAID levels, because there is no need of updation of redundant information.

RAID Level 1 : Mirrored

RAID level 1 is the most expensive solution to achieve the redundancy. In this system, two identical copies of the data on two different disks are maintained. This type of redundancy is called mirroring.

Data striping is used here similar to **RAID 0**.

Every write of a disk block involves two write due to the mirror image of the disk blocks.

These writes may not be performed simultaneously, since a global system failure may occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, write a block on a disk first and then write the other copy on the mirror disk.

A read of a block can be scheduled to the disk that has the smaller access time. Since we are maintaining the full redundant information, the disk for mirror copy may be less costly one to reduce the overall cost.

RAID Level 2 :

RAID levels 2 and **3** make use of a parallel access technique where all member disks participate in the execution of every I/O requests.

Data striping is used in **RAID levels 2** and **3**, but the size of strips are very small, often a small as a single byte or word.

With **RAID 2**, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the cods are stored in the corresponding bit positions on multiple parity disks.

RAID 2 requires fewer disks than **RAID 1**. The number of redundant disks is proportional to the log of the number of data disks. For error-correcting, it uses Hamming code.

On a single read, all disks are simultaneously accessed. The requested data and the associated error correcting code are delivered to the array controller. If there is a single bit error, the controller can recognize and correct the error instantly, so that read access time is not slowed down.

On a single write, all data disks and parity disks must be accessed for the write operation.

RAID level 3 :

RAID level 3 is organized in a similar fashion to **RAID level 2**. The difference is that **RAID 3** requires only a single redundant disk.

RAID 3 allows parallel access, with data distributed in small strips.

Instead of an error correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

In this event of drive failure, the parity drive is accessed and data is reconstructed from the remaining drives. Once the failed drive is replaced, the missing data can be restored on the new drive.

RAID level 4 :

RAID levels 4 through **6** make use of an independent access technique, where each member disk operates independently, so that separate I/O request can be satisfied in parallel.

Data stripings are used in this scheme also, but the data strips are relatively large for **RAID levels 4** through **6**.

With **RAID 4**, a bit-by-bit parity strip is calculated across corresponding strips on each data disks, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of small size is occurred. Each time a write occurs, update is required both in user data and the corresponding parity bits.

RAID level 5 :

RAID level 5 is similar to **RAID 4**, only the difference is that **RAID 5** distributes the parity strips across all disks.

The distribution of parity strips across all drives avoids the potential I/O bottleneck.

RAID level 6 :

In **RAID level 6**, two different parity calculations are carried out and stored in separate blocks on different disks.

The advantage of **RAID 6** is that it has got a high data availability, because the data can be regenerated even if two disk containing user data fails. It is possible due to the use of Reed-Solomon code for parity calculations.

In **RAID 6**, there is a write penalty, because each write affects two parity blocks.