Unit 10 Input – Output Architecture

Structure:

10.1 Introduction

In the previous unit, we have discussed file system interface and its implementation. In our earlier discussion about memory hierarchy (Unit 7), it was implicitly assumed that memory in the computer system would be “fast enough'' to match the speed of the processor (at least for the highest elements in the memory hierarchy) and that no special consideration need be given about how long it would take for a word to be transferred from memory to the processor – an address would be generated by the processor, and after some fixed time interval, the memory system would provide the required information. (In the case of a cache miss, the time interval would be longer, but generally still fixed. For a page fault, the processor would be interrupted; and the page fault handling software invoked.)

Although input-output devices are “mapped’’ to appear like memory devices in many computer systems, ***Input-Output (I/O)*** devices have characteristics quite different from memory devices, and often pose special problems for computer systems. This is principally for two reasons:

- I/O devices span a wide range of speeds. (E.g. terminals accepting input at a few characters per second; disks reading data at over 10 million characters / second).
- Unlike memory operations, I/O operations and the CPU are not generally synchronized with each other.

This unit will give you an overview of various Input / Output features.

Objectives:

After studying this unit, you should be able to:

- explain the significance of I/O operations
- describe the I/O structure for a medium-scale processor system
- discuss I/O control strategies
- explain the I/O address space

10.2 I/O structure

You can see figure 10.1 which shows the general I/O structure associated with many medium-scale processors. Note that the I/O controllers and main memory are connected to the main system bus. The cache memory (usually found on-chip with the CPU) has a direct connection to the processor, as well as to the system bus.

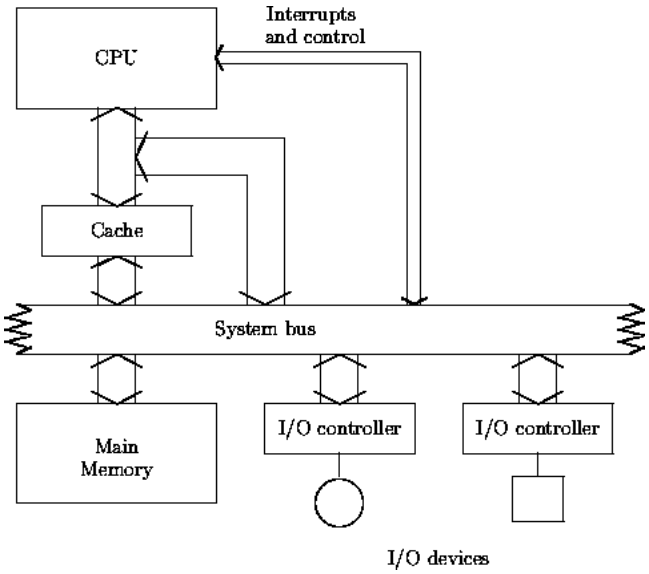

Fig. 10.1: A general I/O structure for a medium-scale processor system

Note that the I/O devices shown here are not connected directly to the system bus; they interface with another device called an I/O controller. In simpler systems, the CPU may also serve as the I/O controller, but in systems where throughput and performance are important, I/O operations are generally handled outside the processor.

Until relatively recently, the I/O performance of a system was somewhat of an afterthought for systems designers. The reduced cost of high-performance disks, permitting the proliferation of virtual memory systems, and the dramatic reduction in the cost of high-quality video display devices, have meant that designers must pay much more attention to this aspect to ensure adequate performance in the overall system.

Because of the different speeds and data requirements of I/O devices, different I/O strategies may be useful, depending on the type of I/O device which is connected to the computer. Because the I/O devices are not synchronized with the CPU, some information must be exchanged between the CPU and the device to ensure that the data is received reliably. This interaction between the CPU and an I/O device is usually referred to as “handshaking". For a complete “handshake," four events are important:

- The device providing the data (the *talker*) must indicate that valid data is now available.
- The device accepting the data (the *listener*) must indicate that it has accepted the data. This signal informs the talker that it need not maintain this data word on the data bus any longer.
- The talker indicates that the data on the bus is no longer valid, and removes the data from the bus. The talker may then set up new data on the data bus.
- The listener indicates that it is not now accepting any data on the data bus. The listener may use data previously accepted during this time, while it is waiting for more data to become valid on the bus.

Note that each of the talker and listener supply two signals. The talker supplies a signal (say, *data valid,* or *DAV*) at step (1). It supplies another signal (say, *data not valid,* or $\overline{DAV}$) at step (3). Both these signals can be coded as a single binary value (*DAV*) which takes the value 1 at step (1) and 0 at step (3). The listener supplies a signal (say, *data accepted*, or *DAC*) at step (2). It supplies a signal (say, *data not now accepted*, or $\overline{DAC}$) at step (4). It, too, can be coded as a single binary variable, *DAC*. Because only two binary variables are required, the handshaking information can be communicated over two wires, and the form of handshaking described above is called a *two wire Handshake*. Other forms of handshaking are used in more complex situations; for example, where there may be more than one controller on the bus, or where the communication is among several devices.

Figure 10.2 shows a timing diagram for the signals *DAV* and *DAC* which identifies the timing of the four events described previously.

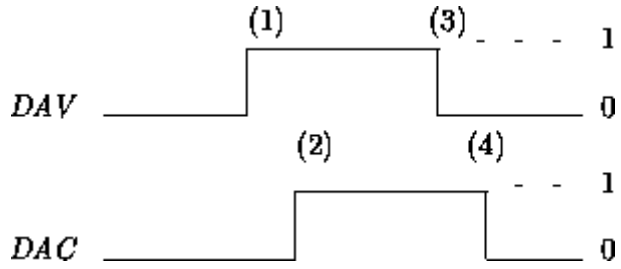

Fig. 10.2: Timing diagram for two-wire handshake

Either the CPU or the I/O device can act as the talker or the listener. In fact, the CPU may act as a talker at one time and a listener at another. For example, when communicating with a terminal screen (an output device) the CPU acts as a talker, but when communicating with a terminal keyboard (an input device) the CPU acts as a listener.

Self Assessment Questions

1. All I/O devices operate at same speed. (True / False)
2. The signal DAV stands for _____.
3. The interaction between the CPU and an I/O device is usually referred to as "_____. (Pick the right option)
   a) Handshaking
   b) Communication
   c) Talking
   d) Listening

10.3 I/O Control Strategies

Several I/O strategies are used between the computer system and I/O devices, depending on the relative speeds of the computer system and the I/O devices. The simplest strategy is to use the processor itself as the I/O controller, and to require that the device follow a strict order of events under direct program control, with the processor waiting for the I/O device at each step.

Another strategy is to allow the processor to be "interrupted" by the I/O devices and to have a (possibly different) "interrupt handling routine" for

each device. This allows for more flexible scheduling of I/O events, as well as more efficient use of the processor. (Interrupt handling is an important component of the operating system.)

A third general I/O strategy is to allow the I/O device, or the controller for the device, access to the main memory. The device would write a block of information in main memory, without intervention from the CPU, and then inform the CPU in some way that a block of memory had been overwritten or read. This might be done by leaving a message in memory, or by interrupting the processor. (This is generally the I/O strategy used by the highest speed devices – hard disks and the video controller.)

10.3.1 Program-controlled I/O

One common I/O strategy is *program-controlled I/O*, (often called *polled I/O*). Here all I/O is performed under control of an “I/O handling procedure," and input or output is initiated by this procedure.

The I/O handling procedure will require some status information (handshaking information) from the I/O device (*e.g.*, whether the device is ready to receive data). This information is usually obtained through a second input from the device; a single bit is usually sufficient, so one input “port" can be used to collect status, or handshake, information from several I/O devices. (A *port* is the name given to a connection to an I/O device; *e.g.*, to the memory location into which an I/O device is mapped). An I/O port is usually implemented as a register (possibly a set of D flip flops) which also acts as a buffer between the CPU and the actual I/O device. The word *port* is often used to refer to the buffer itself.

Typically, there will be several I/O devices connected to the processor; the processor checks the “status" input port periodically, under program control by the I/O handling procedure. If an I/O device requires service, it will signal this need by altering its input to the “status" port. When the I/O control program detects that this has occurred (by reading the status port) then the appropriate operation will be performed on the I/O device which requested the service. A typical configuration might look somewhat as shown in figure 10.3. The outputs labeled “handshake out" would be connected to bits in the “status" port. The input labeled “`handshake in" would typically be generated by the appropriate decode logic when the I/O port corresponding to the device was addressed.

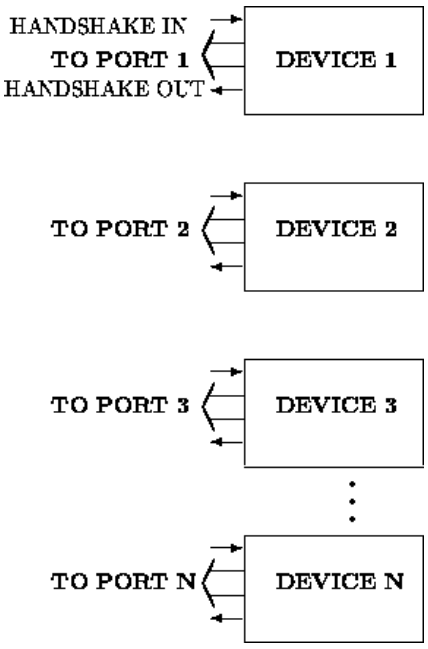

Fig. 10.3: Program controlled I/O

Program-controlled I/O has a number of advantages:

- All control is directly under the control of the program, so changes can be readily implemented.
- The order in which devices are serviced is determined by the program, this order is not necessarily fixed but can be altered by the program, as necessary. This means that the “priority" of a device can be varied under program control. (The “priority" determines which of a set of devices which are simultaneously ready for servicing will actually be serviced first).
- It is relatively easy to add or delete devices.

  Perhaps the chief disadvantage of program-controlled I/O is that a great deal of time may be spent testing the status inputs of the I/O devices, when the devices do not need servicing. This “busy wait" or “wait loop" during which the I/O devices are polled but no I/O operations are performed is really time wasted by the processor, if there is other work which could be done at that time. Also, if a particular device has its data available for only a short time, the data may be missed because the input was not tested at the appropriate time.

Program controlled I/O is often used for simple operations which must be performed sequentially. For example, the following may be used to control the temperature in a room:

```
DO forever
    INPUT temperature
    IF (temperature < setpoint) THEN
       turn heat ON
    ELSE
       turn heat OFF
    END IF
```

Note here that the order of events is fixed in time, and that the program loops forever. (It is really waiting for a change in the temperature, but it is a “busy wait.")

10.3.2 Interrupt-controlled I/O

Interrupt-controlled I/O reduces the severity of the two problems mentioned for program-controlled I/O by allowing the I/O device itself to initiate the device service routine in the processor. This is accomplished by having the I/O device generate an interrupt signal which is tested directly by the hardware of the CPU. When the interrupt input to the CPU is found to be active, the CPU itself initiates a subprogram call to somewhere in the memory of the processor; the particular address to which the processor branches on an interrupt depends on the interrupt facilities available in the processor.

The simplest type of interrupt facility is where the processor executes a subprogram branch to some specific address whenever an interrupt input is detected by the CPU. The return address (the location of the next instruction in the program that was interrupted) is saved by the processor as part of the interrupt process.

If there are several devices which are capable of interrupting the processor, then with this simple interrupt scheme the interrupt handling routine must examine each device to determine which one caused the interrupt. Also, since only one interrupt can be handled at a time, there is usually a hardware “priority encoder" which allows the device with the highest priority to interrupt the processor, if several devices attempt to interrupt the

processor simultaneously. In figure 10.3, the “handshake out” outputs would be connected to a priority encoder to implement this type of I/O. The other connections remain the same. (Some systems use a “daisy chain'' priority system to determine which of the interrupting devices is serviced first. “Daisy chain'' priority resolution is discussed later.)

In most modern processors, interrupt return points are saved on a “stack” in memory, in the same way as return addresses for subprogram calls are saved. In fact, an interrupt can often be thought of as a subprogram which is invoked by an external device. If a stack is used to save the return address for interrupts, it is then possible to allow one interrupt handling routine for another interrupt. In modern computer systems, there are often several “priority levels” of interrupts, each of which can be disabled, or “masked.” There is usually one type of interrupt input which cannot be disabled (a non-maskable interrupt) which has priority over all other interrupts. This interrupt input is used for warning the processor of potentially catastrophic events such as an imminent power failure, to allow the processor to shut down in an orderly way and to save as much information as possible.

Most modern computers make use of “vectored interrupts.” With vectored interrupts, it is the responsibility of the interrupting device to provide the address in main memory of the interrupt servicing routine for that device. This means, of course, that the I/O device itself must have sufficient “intelligence” to provide this address when requested by the CPU, and also to be initially “programmed” with this address information by the processor. Although somewhat more complex than the simple interrupt system described earlier, vectored interrupts provide such a significant advantage in interrupt handling speed and ease of implementation (i.e., a separate routine for each device) that this method is almost universally used on modern computer systems.

Some processors have a number of special inputs for vectored interrupts. Others require that the interrupting device itself provide the interrupt address as part of the process of interrupting the processor.

10.3.3 Direct memory access

In most mini and mainframe computer systems, a great deal of input and output occurs between the disk system and the processor. It would be very inefficient to perform these operations directly through the processor; it is

much more efficient if such devices, which can transfer data at a very high rate, place the data directly into the memory, or take the data directly from the processor *without* direct intervention from the processor. I/O performed in this way is usually called ***direct memory access, or DMA***. The controller for a device employing DMA must have the capability of generating address signals for the memory, as well as all of the memory control signals. The processor informs the DMA controller that data is available (or is to be placed into) a block of memory locations starting at a certain address in memory. The controller is also informed of the length of the data block. Figure 10.4 shows the block diagram of DMA controller.

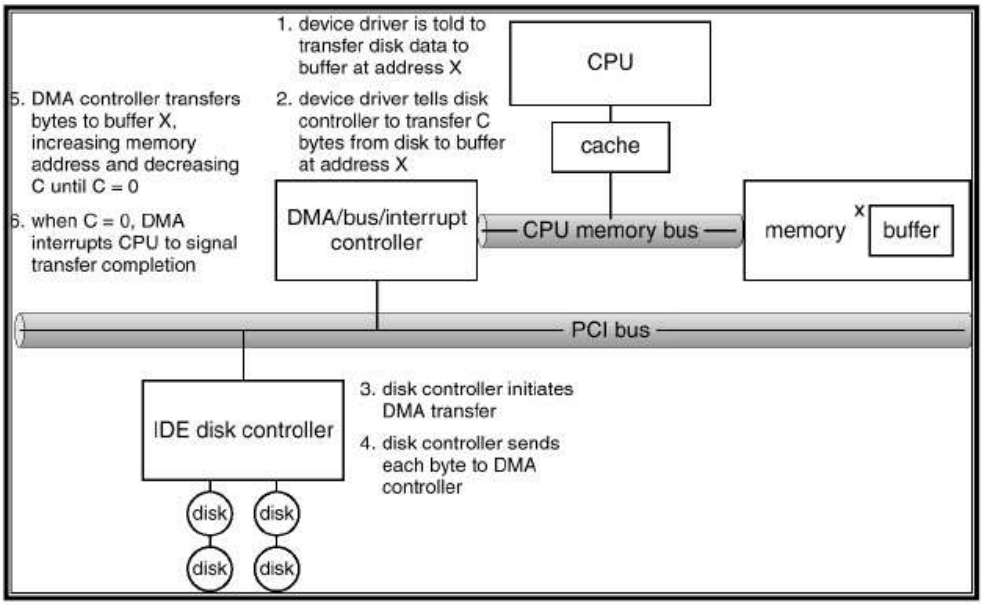

Fig. 10.4: Block diagram of a DMA controller

There are two possibilities for the timing of the data transfer from the DMA controller to memory:

- The controller can cause the processor to halt if it attempts to access data in the same bank of memory into which the controller is writing. This is the fastest option for the I/O device, but may cause the processor to run more slowly because the processor may have to wait until a full block of data is transferred.
- The controller can access memory in memory cycles which are not used by the particular bank of memory into which the DMA controller is writing

data. This approach, called “cycle stealing”, is perhaps the most commonly used approach. (In a processor with a cache that has a high hit rate this approach may not slow the I/O transfer significantly).

DMA is a sensible approach for devices which have the capability of transferring blocks of data at a very high data rate, in short bursts. It is not worthwhile for slow devices, or for devices which do not provide the processor with large quantities of data. Because the controller for a DMA device is quite sophisticated, the DMA devices themselves are usually quite sophisticated (and expensive) compared to other types of I/O devices.

One problem that systems employing several DMA devices have to address is the contention for the single system bus. There must be some method of selecting which device controls the bus (acts as “bus master”) at any given time. There are many ways of addressing the “bus arbitration” problem; three techniques which are often implemented in processor systems are the following: Daisy chain arbitration, priority encoded arbitration and distributed arbitration by self-section. These are also often used to determine the priorities of other events which may occur simultaneously, like interrupts. They rely on the use of at least two signals (**bus_request** and **bus_grant**), used in a manner similar to the two-wire handshake. Let’s see them.

Daisy chain arbitration: Here, the requesting device or devices assert the signal **bus_request**. The bus arbiter returns the **bus_grant** signal, which passes through each of the devices which can have access to the bus, as shown in figure 10.5. Here, the priority of a device depends solely on its position in the daisy chain. If two or more devices request the bus at the same time, the highest priority device is granted the bus first, and then the **bus_grant** signal is passed further down the chain. Generally a third signal (**bus_release**) is used to indicate to the bus arbiter that the first device has finished its use of the bus. Holding **bus_request** asserted indicates that another device wants to use the bus.

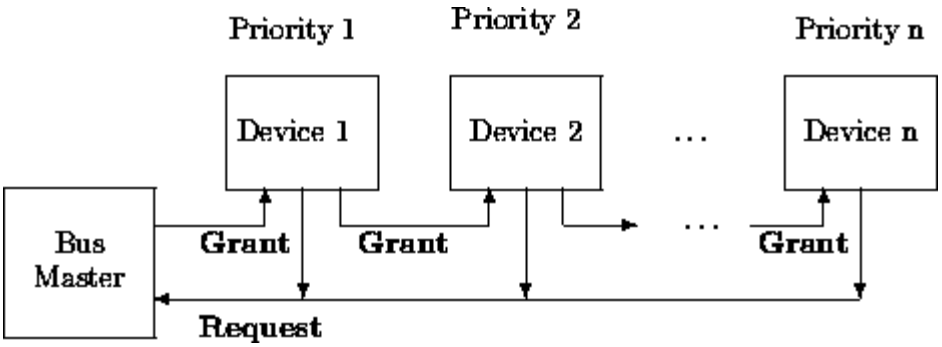

Fig. 10.5: Daisy chain bus arbitration

Priority encoded arbitration: Here, each device has a request line connected to a centralized arbiter that determines which device will be granted access to the bus. The order may be fixed by the order of connection (priority encoded), or it may be determined by some algorithm preloaded into the arbiter. Figure 10.6 shows this type of system. Note that each device has a separate line to the bus arbiter. (The **bus_grant** signals have been omitted for clarity.)

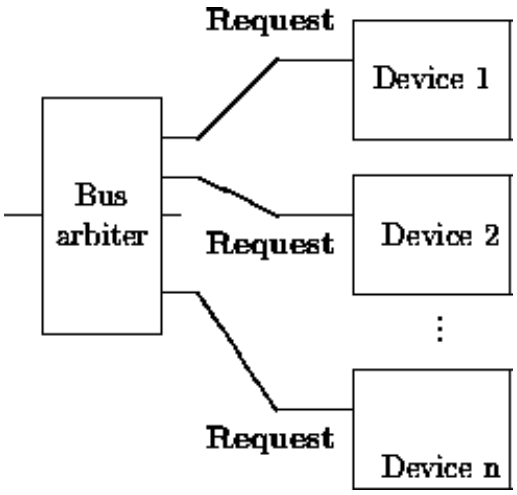

Fig. 10.6: Priority encoded bus arbitration

Distributed arbitration by self-selection: Here, the devices themselves determine which of them has the highest priority. Each device has a **bus_request** line or lines on which it places a code identifying it. Each device examines the codes for all the requesting devices, and determines whether or not it is the highest priority requesting device.

These arbitration schemes may also be used in conjunction with each other. For example, a set of similar devices may be daisy chained together, and this set may be an input to a priority encoded scheme. Using interrupts driven device drivers to transfer data to or from hardware devices works well when the amount of data is reasonably low. For example a 9600 baud modem can transfer approximately one character for every millisecond.

If the interrupt latency, the amount of time that it takes between the hardware device raising the interrupt and the device driver's interrupt handling routine being called, is low (say 2 milliseconds) then the overall system impact of the data transfer is very low. The 9600 baud modem data transfer would only take 0.002% of the CPU's processing time. For high speed devices, such as hard disk controllers or Ethernet devices the data transfer rate is a lot higher. A SCSI device can transfer up to 40 Mbytes of information per second.

Direct Memory Access, or DMA, was invented to solve this problem. A DMA controller allows devices to transfer data to or from the system's memory without the intervention of the processor. A PC's ISA DMA controller has 8 DMA channels of which 7 are available for use by the device drivers. Each DMA channel has associated with it a 16 bit address register and a 16 bit count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. It then tells the device that it may start the DMA when it wishes. When the transfer is complete the device interrupts the PC. Whilst the transfer is taking place the CPU is free to do other things.

Device drivers have to be careful when using DMA. First of all the DMA controller knows nothing of virtual memory, it only has access to the physical memory in the system. Therefore the memory that is being directly memory accessed to or from must be a contiguous block of physical memory. This means that you cannot directly access memory into the virtual address space of a process. You can however lock the processes physical pages into memory, preventing them from being swapped out to the swap device during a DMA operation. Secondly, the DMA controller cannot access the whole of physical memory. The DMA channel's address register represents the first 16 bits of the DMA address; the next 8 bits come from

the page register. This means that DMA requests are limited to the bottom 16 Mbytes of memory.

DMA channels are scares resources, there are only 7 of them, and they cannot be shared between device drivers. Just like interrupts the device driver must be able to work out which DMA channel it should use. Like interrupts, some devices have a fixed DMA channel. The floppy device, for example, always uses DMA channel 2. Sometimes the DMA channel for a device can be set by jumpers, a number of Ethernet devices use this technique. The more flexible devices can be told (via their CSRs) which DMA channels to use and, in this case, the device driver can simple pick a free DMA channel to use.

10.4 The I/O Address Space

Some processors map I/O devices in their own, separate, address space; others use memory addresses as addresses of I/O ports. Both approaches have advantages and disadvantages. The advantages of a separate address space for I/O devices are, primarily, that the I/O operations would then be performed by separate I/O instructions, and that all the memory address space could be dedicated to memory.

Typically, however, I/O is only a small fraction of the operations performed by a computer system; generally less than 1 percent of all instructions are I/O instructions in a program. It may not be worthwhile to support such infrequent operations with a rich instruction set, so I/O instructions are often rather restricted.

In processors with memory mapped I/O, any of the instructions which references memory directly can also be used to reference I/O ports, including instructions which modify the contents of the I/O port (*e.g.*, arithmetic instructions.)

Some problems can arise with memory mapped I/O in systems which use cache memory or virtual memory. If a processor uses a virtual memory mapping, and the I/O ports are allowed to be in a virtual address space, the mapping to the physical device may not be consistent if there is a context switch. Moreover the device would have to be capable of performing the virtual-to-physical mapping. If physical addressing is used, mapping across page boundaries may be problematic.

If the memory locations are cached, then the value in cache may not be consistent with the new value loaded in memory. Generally, either there is some method for invalidating cache that may be mapped to I/O addresses, or the I/O addresses are not cached at all.

Self Assessment Questions

4. The simplest I/O strategy is to use the processor itself as the I/O controller. (True / False)
5. DMA stands for _____.
6. A 9600 baud modem can transfer approximately _____ character for every millisecond. (Pick the right option)
   a) Three
   b) One
   c) Two
   d) Four

10.5 Summary

Let's recapitulate the important points discussed in this unit:

- The interaction between the CPU and an I/O device is usually referred to as "handshaking".
- In program-controlled I/O, all I/O operations are performed under control of an "I/O handling procedure," and input or output is initiated by this procedure.
- Interrupt-controlled I/O allows the I/O device itself to initiate the device service routine in the processor.
- Direct Memory Access (DMA) is a sensible approach for devices which have the capability of transferring blocks of data at a very high data rate, in short bursts.

10.6 Terminal Questions

1. Draw a block diagram of an I/O structure in a medium scale processor and explain its working.
2. What are various I/O control strategies? Discuss in brief.
3. Explain programmed I/O and interrupt I/O. How they differ?
4. Discuss the concept of Direct Memory Access. What are its advantages over other methods?

10.7 Answers

Self Assessment Questions

1. False
2. Data valid
3. a) Handshaking
4. True
5. Direct Memory Access
6. b) One

Terminal Questions

1. In the I/O structure of medium sized processors, the I/O controllers and main memory are connected to the main system bus. The cache memory (usually found on-chip with the CPU) has a direct connection to the processor, as well as to the system bus. (Refer Section 10.2 for detail)
2. Several I/O strategies are used between the computer system and I/O devices, depending on the relative speeds of the computer system and the I/O devices. The simplest strategy is to use the processor itself as the I/O controller, and to require that the device follow a strict order of events under direct program control, with the processor waiting for the I/O device at each step. (Refer Section 10.3)
3. In program-controlled I/O, all I/O operations are performed under control of an “I/O handling procedure," and input or output is initiated by this procedure. Interrupt-controlled I/O allows the I/O device itself to initiate the device service routine in the processor. This is accomplished by having the I/O device generate an interrupt signal which is tested directly by the hardware of the CPU. (Refer Sub-sections 10.3.1 and 10.3.2)
4. In most mini and mainframe computer systems, a great deal of input and output occurs between the disk system and the processor. These data can be transferred at a very high rate by placing the data directly into the memory, or by taking the data directly from the processor *without* direct intervention from the processor. I/O performed in this way is usually called *direct memory access*, or DMA. (Refer Sub-section 10.3.3)