# Module II
# INPUT/OUTPUT ORGANIZATION

## ACCESSING I/O DEVICES

There are 2 ways to deal with I/O devices (Figure 4.1).

***1) Memory mapped I/O***

When I/O devices and memory share the same address space, the arrangement is called *memory mapped I/O.*

- Memory and I/O devices share a common address-space.
- Any data-transfer instruction (like Move, Load) can be used to exchange information.

For example,

Move DATAIN, R0;

This instruction reads data from DATAIN (input-buffer associated with keyboard) & stores them into processor-register R0.

Advantage:

1.No need of separate address space.
2.No special instructions are needed.
3.Easy to implement.

Disadvantage:

Slow compare to other technique.

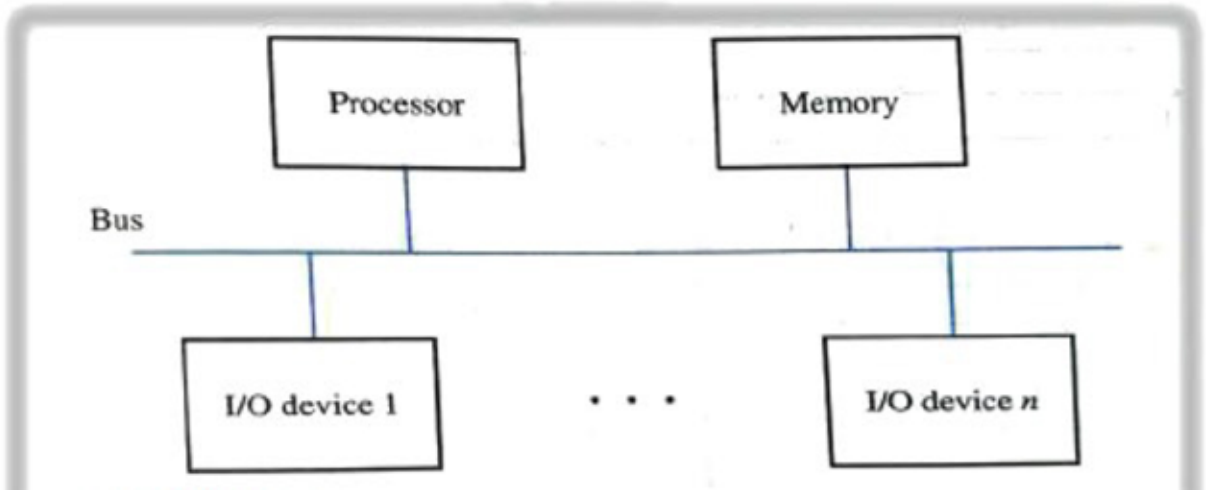



FIGURE 4.1
A single bus structure.

2) **In *I/O mapped I/O***

- Memory and I/O address-spaces are different.
- A special instruction named IN and OUT is used for data transfer.

Advantage:

I/O devices deal with fewer address-lines.

1. Faster compare to other technique.
2. Dedicated instructions for I/O operation.

Disadvantage:

Complex to design

## I/O Interface for an Input Device

- *Address decoder:* decodes address sent on bus, so as to enable input-device (Figure 4.2).
- D*ata register:* holds data being transferred to or from the processor.
- *Status register:* contains information relevant to operation of I/O device.
- Address decoder, data- and status-registers, and control-circuitry required to coordinate I/O transfers constitute device's interface-circuit.

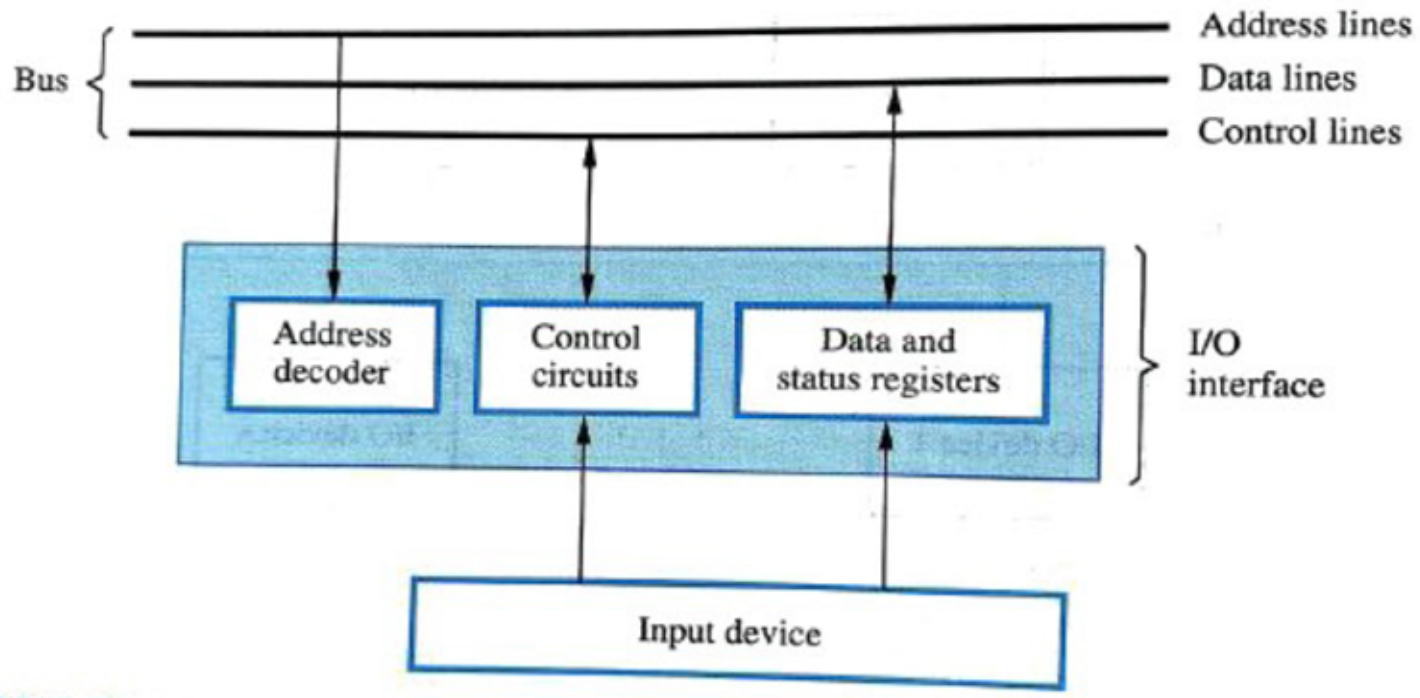



FIGURE 4.2
I/O interface for an input device.

## MECHANISMS USED FOR INTERFACING I/O DEVICES

### 1) Program Controlled I/O

- Processor repeatedly checks a status-flag to achieve required synchronization between processor & input/output device. (We say that the processor polls the device).
- Main drawback: The processor wastes its time in checking the status of the device before actual data transfer takes place.

In figure 4.3 the status register are explained based on the program in figure 4.4. The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. For example, bits KIRQ and DIRQ are the interrupt request bits for the keyboard and the display, respectively. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is called to provide the requested service.

### 2) Interrupt I/O

- Synchronization is achieved by having I/O device send a special signal over bus whenever it is ready for a data transfer operation.

**3) Direct Memory Access (DMA)**

- This involves having the device-interface transfer data directly to or from the memory without continuous involvement by the processor.

## INTERRUPTS

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.

- I/O device initiates the action instead of the processor. This is done by sending a special hardware signal to the processor called as *interrupt* (INTR), on the interrupt-request line.
- The processor can be performing its own task without the need to continuously check the I/O device.

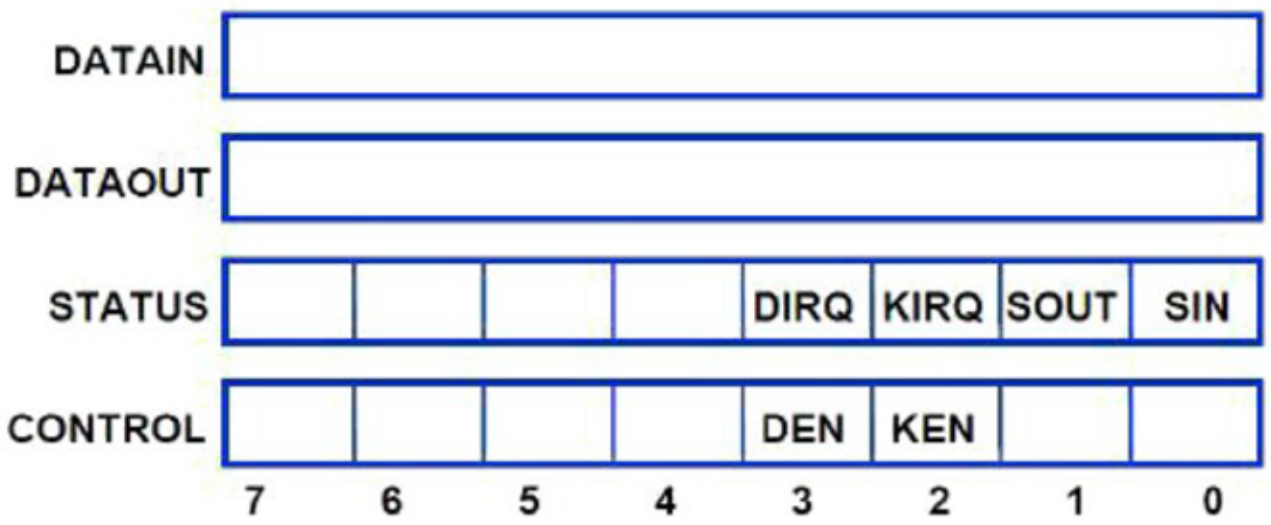



FIGURE 4.3
Registers in an I/O interface.

```
        Move        #LINE,R1        Initialize memory pointer.
WAITK   TestBit     #0,STATUS       Test SIN.
        Branch=0    WAITK           Wait for character to be entered.
        Move        DATAIN,R0       Read character.
WAITD   TestBit     #1,STATUS       Test SOUT.
        Branch=0    WAITD           Wait for display to become ready.
        Move        R0,DATAOUT      Send character to display.
        Move        R0,(R1)+        Store character and advance pointer.
        Compare     #$0D,R0         Check if carriage return.
        Branch≠0    WAITK           If not, get another character.
        Call        PROCESS         Call a subroutine to process the input
                                      line.
```

FIGURE 4.4
A program that reads one line from the keyboard and sends it to the display.

- When device gets ready, it will "alert" the processor by sending an interrupt-signal (Figure 4.5).
- The routine executed in response to an interrupt-request is called ISR (Interrupt Service Routine).
- Once the interrupt-request signal comes from the device, the processor has to inform the device that its request has been recognized and will be serviced soon. This is indicated by a special control signal on the bus called *interrupt-acknowledge* (INTA).

**Difference between subroutine & ISR**

- A subroutine performs a function required by the program from which it is called.
- However, the ISR may not have anything in common with the program being executed at the time the interrupt-request is received. Before starting execution of ISR, any information that may be altered during the execution of that routine must be saved. This information must be restored before the interrupted-program resumed.
- Another difference is that an interrupt is a mechanism for coordinating I/O transfers whereas a subroutine is just a linkage of 2 or more function related to each other.

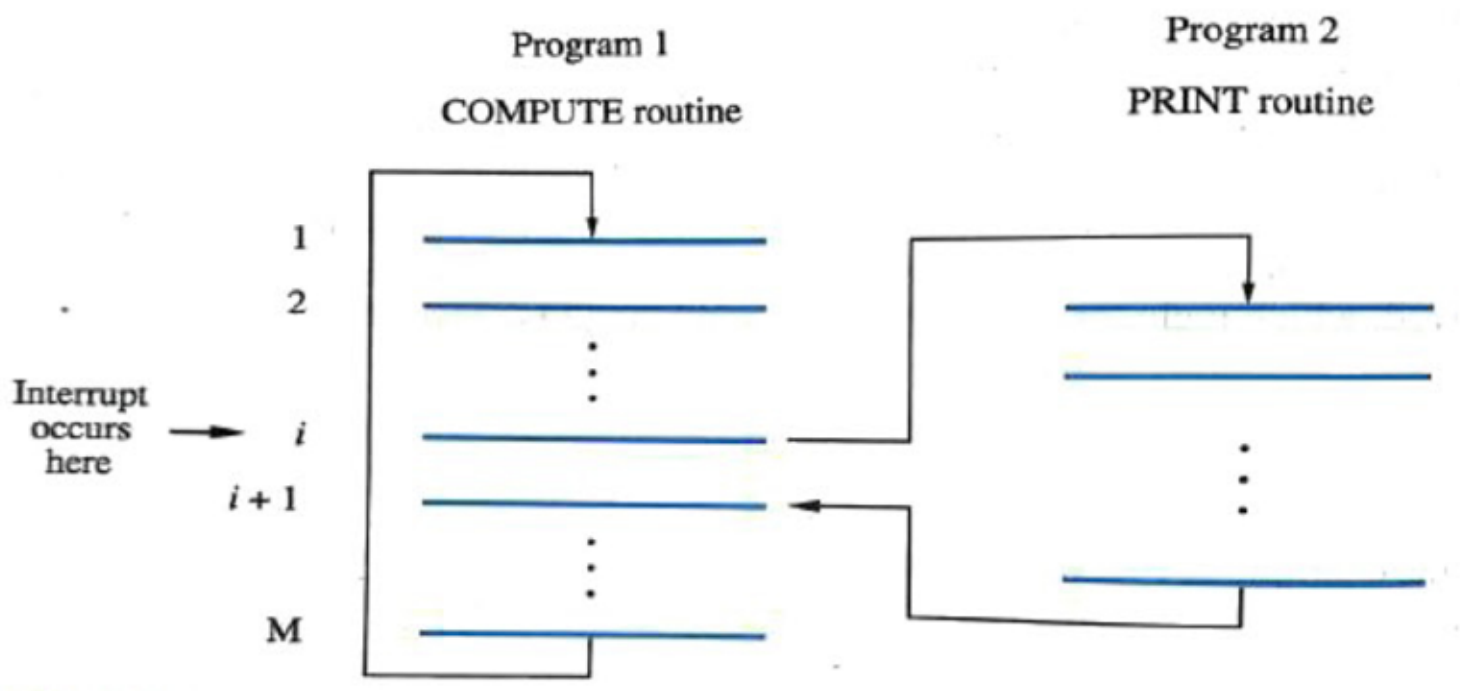



**FIGURE 4.5**
**Transfer of control through the use of interrupts.**

The speed of operation of the processor and I/O devices differ greatly. Also, since I/O devices are manually operated in many cases (like pressing a key on keyboard); there may not be synchronization between the CPU operations and I/O operations with reference to CPU clock. To cater to the different needs of I/O operations, three mechanisms have been developed for interfacing I/O devices.

1. Program controlled I/O
2. Interrupt I/O
3. Direct memory access (DMA).

- Saving registers increases the delay between the time an interrupt request is received and the start of execution of the ISR. This delay is called interrupt latency.
- Since interrupts can arrive at any time, they may alter the sequence of events. Hence, facility must be provided to enable and disable interrupts as desired.

- Consider the case of a single interrupt request from one device. The device keeps the interrupt request signal activated until it is informed that the processor has accepted its request. This activated signal, if not deactivated may lead to successive interruptions, causing the system to enter into an infinite loop.

## INTERRUPT HARDWARE

- An I/O device requests an interrupt by activating a bus-line called interrupt-request (IR).
- A single IR line can be used to serve "n" devices (Figure 4.6).
- All devices are connected to IR line via switches to ground.
- To request an interrupt, a device closes its associated switch. Thus, if all IR signals are inactive (i.e. if all switches are open), the voltage on the IR line will be equal to $V_{dd}$.
- When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the INTR received by the processor to goto 1.
- The value of INTR is the logical OR of the requests from individual devices

$$INTR = INTR_1 + INTR_2 + \ldots\ldots + INTR_n$$

- A special gate known as open-collector or open-drain are used to drive the INTR line.
- Resistor R is called a *pull-up resistor* because it pulls the line voltage up to the high-voltage state when the switches are open.

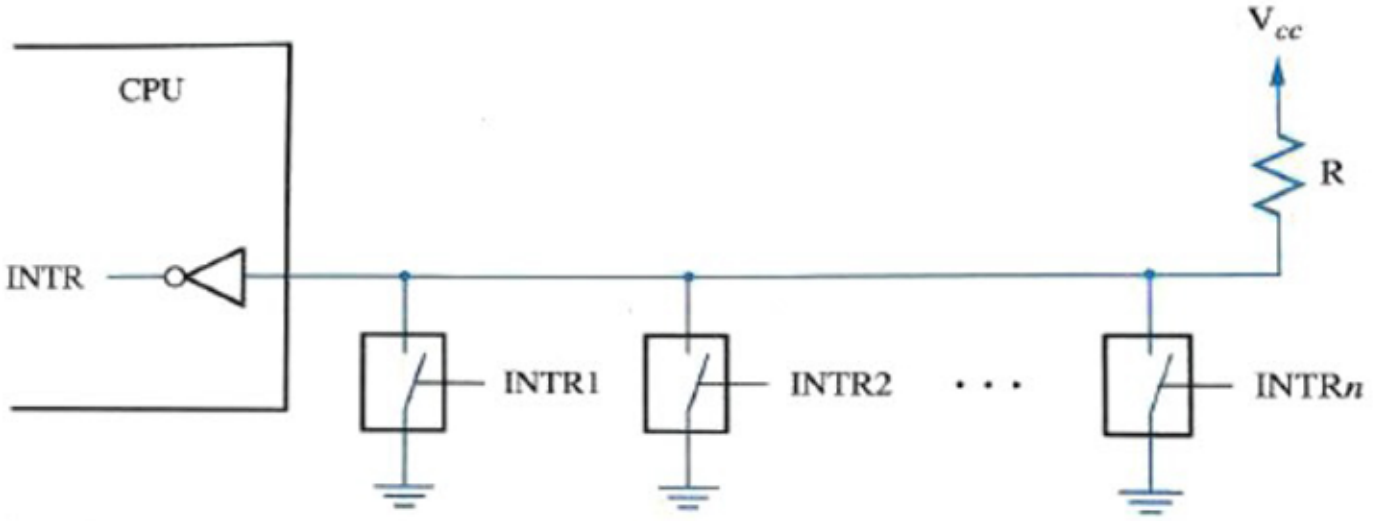



**FIGURE 4.6**
An equivalent circuit for an open-collector bus used to implement a common interrupt-request line.

## ENABLING & DISABLING INTERRUPTS

To prevent the system from entering into an infinite-loop because of interrupt, there are three possibilities:

1. The first possibility is to have the processor-hardware ignore the interrupt-request line until the execution of the first instruction of the ISR has been completed.
2. The second option is to have the processor automatically disable interrupts before starting the execution of the ISR.
3. In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be edge-triggered.

Sequence of events involved in handling an interrupt-request from a single device is as follows:

1. The device raises an interrupt-request.
2. The program currently being executed is interrupted.
3. All interrupts are disabled (by changing the control bits in the Process Status (PS) register).
4. The device is informed that its request has been recognized, and in response, the device deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the ISR.
6. Interrupts are enabled again and execution of the interrupted program is resumed.

## HANDLING MULTIPLE DEVICES

The interrupt from multiple devices are handled by the

1. Polling
2. Vectored Interrupt

### Polling

- Information needed to determine whether a device is requesting an interrupt is available in its status-register.
- When a device raises an interrupt-request, it sets IRQ bit to 1 in its status-register (Figure 4.3).
- KIRQ and DIRQ are the interrupt-request bits for keyboard & display.
- Simplest way to identify interrupting device is to have ISR poll all I/O devices connected to bus.
- The first device encountered with its IRQ bit set is the device that should be serviced. After servicing this device, next requests may be serviced.

Advantage:

1. Simple
2. Easy to implement.

Disadvantage:

More time spent polling IRQ bits of all devices (that may not be requesting any service).

### Vectored Interrupts

- In the program shown in figure 4.4, a device requesting an interrupt identifies itself by sending a special-code to processor over bus. (This enables processor to identify individual devices even if they share a single interrupt-request line).
- The code represents starting-address of ISR for that device.
- ISR for a given device must always start at same location.
- The address stored at the location pointed to by interrupting-device is called the interrupt-vector.
- Processor
  → loads interrupt-vector into PC &
  → executes appropriate ISR

- Interrupting-device must wait to put data on bus only when processor is ready to receive it.
- When processor is ready to receive interrupt-vector code, it activates INTA line.
- I/O device responds by sending its interrupt-vector code & turning off the INTR signal.

## CONTROLLING DEVICE REQUESTS

There are 2 independent mechanisms for controlling interrupt requests.

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

## INTERRUPT NESTING

A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device. Each of the INTR lines is assigned a different priority-level (Figure 4.7).

- Priority-level of processor is the priority of program that is currently being executed.
- During execution of an ISR, interrupt-requests will be accepted from some devices but not from others depending upon device's priority.
- Processor accepts interrupts only from devices that have priority higher than its own.
- At the time of execution of an ISR for some device is started, priority of processor is raised to that of the device
- Processor's priority is encoded in a few bits of Processor-Status (PS) register. This can be changed by program instructions that write into PS. These are called ***privileged instructions.***
- Privileged-instructions can be executed only while processor is running in *super-user* mode.
- Processor is in supervisor-mode only when executing operating-system routines. (An attempt to execute a privileged-instruction while in the user-mode leads to a special type of interrupt called a *privileged exception*).

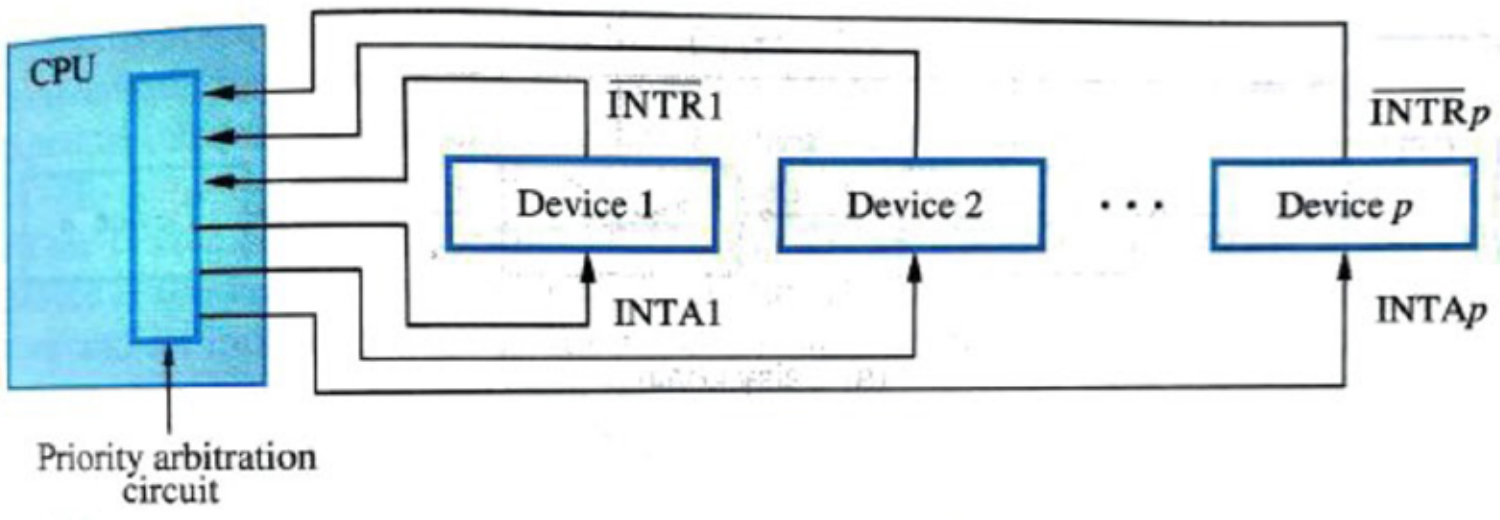
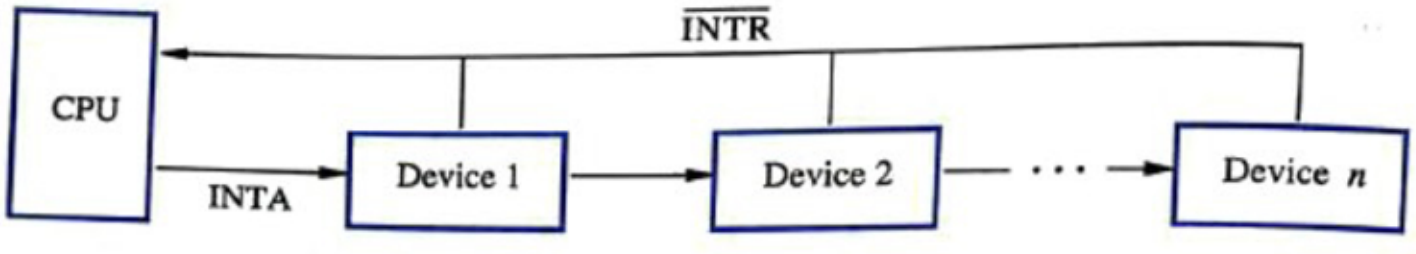


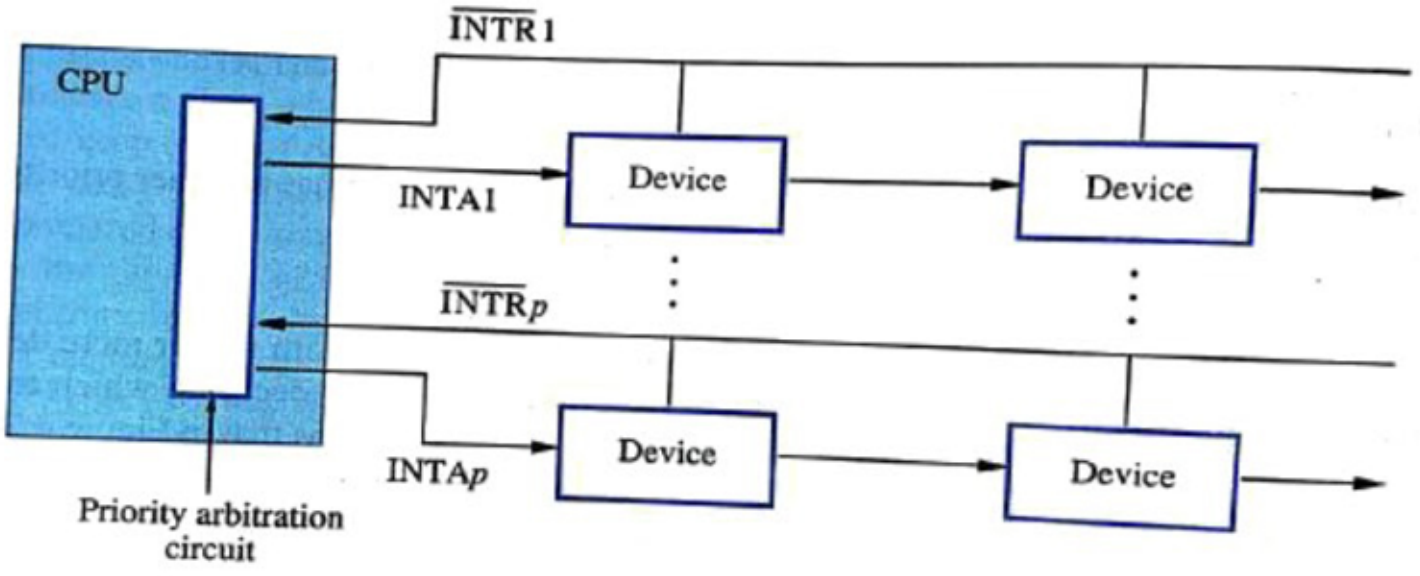

**FIGURE 4.7**
Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

## SIMULTANEOUS REQUESTS

- INTR line is common to all devices Figure 4.8.
- INTA line is connected in a daisy-chain fashion such that INTA signal propagates serially through devices.
- When several devices raise an interrupt-request and INTR line is activated, processor responds by setting INTA line to 1. This signal is received by device 1.
- Device 1 passes signal on to device 2 only if it does not require any service.
- If device 1 has a pending-request for interrupt, it blocks INTA signal and proceeds to put its identifying code on data lines.
- Device that is electrically closest to processor has highest priority.
- Main advantage: This allows the processor to accept interrupt-requests from some devices but not from others depending upon their priorities.




(a) Daisy chain




(b) Arrangement of priority groups

**FIGURE 4.8**
Interrupt priority schemes.

## EXCEPTIONS

*Exception* refers to any event that causes an interruption. I/O interrupts are one example of an exception.

- An interrupt is an event that causes
  → Execution of one program to be suspended &
  → Execution of another program to begin.

### Recovery from Errors

Computers use a variety of techniques to ensure that all hardware-components are operating properly. For e.g. many computers include an error-checking code in main-memory which allows detection of errors in stored-data.

- If an error occurs, control-hardware detects it & informs processor by raising an interrupt.
- When exception processing is initiated (as a result of errors), processor
  → suspends program being executed &
  → Starts an ESR (Exception Service Routine). This routine takes appropriate action to recover from the error to inform user about it.

### Debugging

Debugger helps programmer find errors in a program and uses exceptions to provide two important facilities:

1. Trace &
2. Breakpoints

- When a processor is operating in ***trace-mode***, an exception occurs after execution of every instruction (using debugging-program as ESR).
- Debugging-program enables user to examine contents of registers (AX, BX), memory-locations and so on.
- On return from debugging-program, next instruction in program being debugged is executed, and then debugging-program is activated again.
- ***Breakpoints*** provide a similar facility except that program being debugged is interrupted only at specific points selected by user. An instruction called Trap (or Software interrupt) is usually provided for this purpose.

### Privilege Exception

- To protect OS of computer from being corrupted by user-programs, certain instructions can be executed only while processor is in supervisor-mode. These are called *privileged instructions.*
- For example when the processor is running in user-mode, it will not execute an instruction that changes priority-level of processor.
- An attempt to execute such an instruction will produce a privilege-exception. As a result, processor switches to supervisor-mode & begins to execute an appropriate routine in OS.

## DIRECT MEMORY ACCESS (DMA)

The transfer of a block of data directly between an external device & main memory without continuous involvement by processor is called as ***DMA***.

- DMA transfers are performed by a control-circuit that is part of I/O device interface. This circuit is called as a *DMA controller* (Figure 4.19).

- DMA controller performs the functions that would normally be carried out by processor, In controller, three registers are accessed by processor to initiate transfer operations (Figure 4.18):
  1. A register is used for storing *starting-address*.
  2. A register is used for storing word-*count*.
  3. Third register contains status- & control-flags
- The ***R/W*** bit determines *direction* of transfer.
  When R/W=1, controller performs a *read operation* (i.e. it transfers data from memory to I/O), Otherwise it performs a *write operation* (i.e. it transfers data from I/O device to memory).
- When done=1, controller
  - ✓ Completes transferring a block of data &
  - ✓ Is ready to receive another command.
- When IE=1, controller raises an interrupt after it has completed transferring a block of data (IE=Interrupt Enable).
- Finally, when IRQ=1, controller requests an interrupt. (Requests by DMA devices for using the bus are always given higher priority than processor requests).

There are two ways in which the DMA operation can be carried out:

1. In one method, processor originates most memory-access cycles. DMA controller is said to "steal" memory cycles from processor. Hence, this technique is usually called ***cycle stealing***.
2. In second method, DMA controller is given exclusive access to main-memory to transfer a block of data without any interruption. This is known as ***block mode (or burst mode)***.

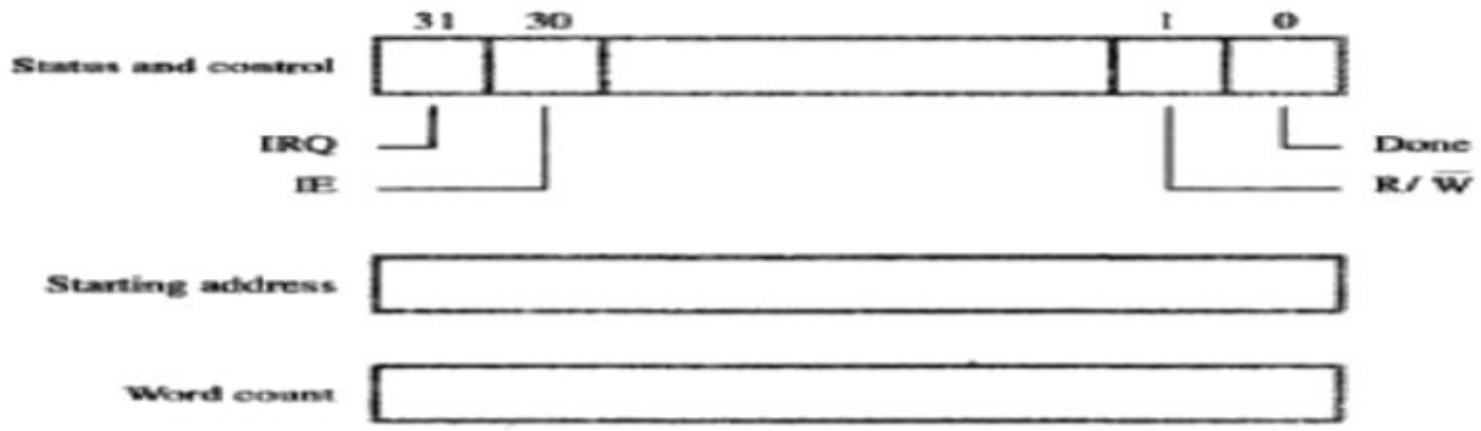



**Figure 4.18** Registers in a DMA interface.

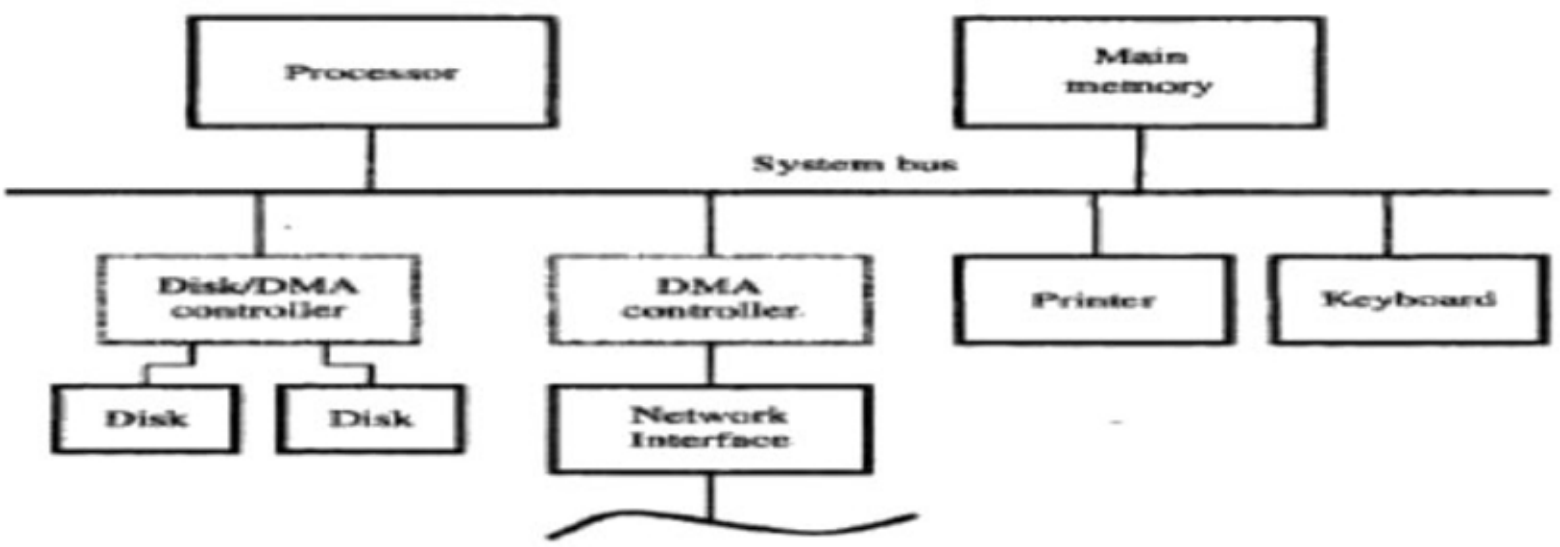



**Figure 4.19** Use of DMA controllers in a computer system.

## BUS ARBITRATION

The device that is allowed to initiate data transfers on bus at any given time is called ***bus-master.***

There can be only one bus master at any given time.

***Bus arbitration*** is the process by which next device to become the bus-master is selected and bus-mastership is transferred to it.

There are two approaches to bus arbitration:

1. In ***centralized arbitration***, a single bus-arbiter performs the required arbitration.
2. In ***distributed arbitration***, all devices participate in selection of next bus-master.

## CENTRALIZED ARBITRATION

A single bus-arbiter performs the required arbitration (Figure: 4.16 & 4.17). Normally, *processor* is the bus master unless it grants bus mastership to one of the DMA controllers.

- A DMA controller indicates that it needs to become bus master by activating Bus-Request line (BR).
- The signal on the BR line is the logical OR of bus-requests from all devices connected to it.
- When BR is activated, processor activates Bus-Grant signal (BG1) indicating to DMA controllers that they may use bus when it becomes free. (This signal is connected to all DMA controllers using a ***daisy-chain arrangement***).
- If DMA controller-1 is requesting the bus, it blocks propagation of grant-signal to other devices.
  → Otherwise, it passes the grant downstream by asserting BG2.
- Current bus-master indicates to all devices that it is using bus by activating Bus-Busy line (BBSY).
- Arbiter circuit ensures that only one request is granted at any given time according to a predefined priority scheme

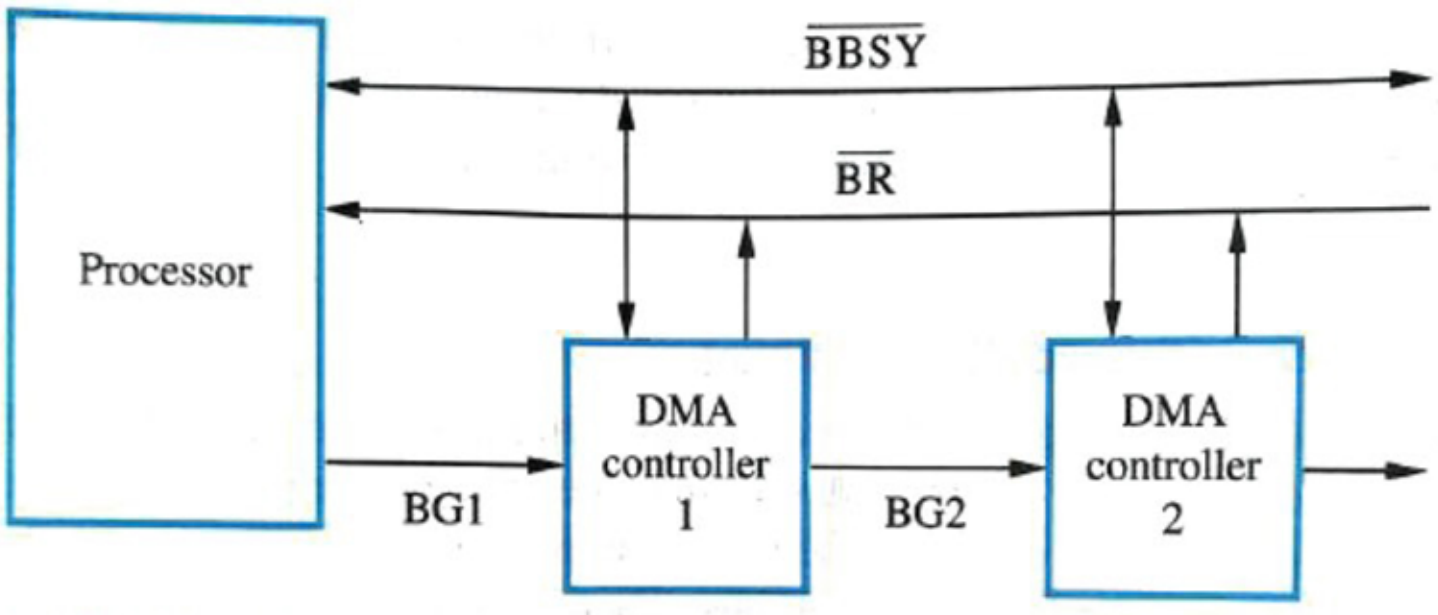



**FIGURE 4.16**
A simple arrangement for bus arbitration using a daisy chain.

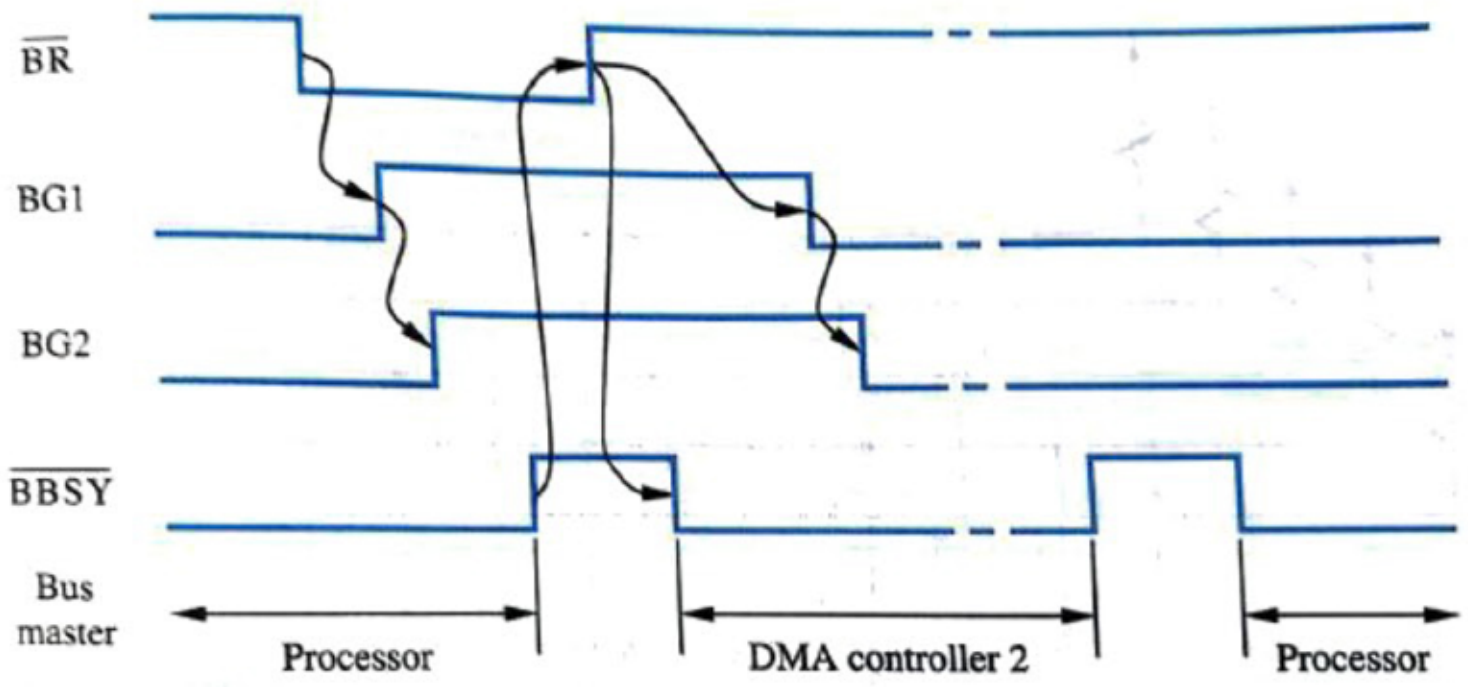



FIGURE 4.17
Sequence of signals during transfer of bus mastership for the devices in Figure 4.16.

**Note:**

A ***conflict*** may arise if both the processor and a DMA controller try to use the bus at the same time to access the main memory. To resolve these conflicts, a special circuit called ***the bus arbiter*** is provided to coordinate the activities of all devices requesting memory transfers

## DISTRIBUTED ARBITRATION

- All device participate in the selection of next bus-master (Figure 4.18)
- Each device on bus is assigned a 4-bit identification number (ID).
- When 1 or more devices request bus, they
  - → Assert Start-Arbitration signal &
  - → Place their 4-bit ID numbers on four open-collector lines $\overline{ARB}0$ through $\overline{ARB}3$.
- A winner is selected as a result of interaction among signals transmitted over these lines by all contenders.
- Net outcome is that the code on 4 lines represents request that has the highest ID number.

Advantage:

This approach offers higher reliability since operation of bus is not dependent on any single device.

## BUSES

Bus

- ✓ is used to inter-connect main-memory, processor & I/O devices
- ✓ includes lines needed to support interrupts & arbitration

- Primary function: To provide a communication-path for transfer of data.
- *Bus protocol* is set of rules that govern the behavior of various devices connected to the buses. and specifies parameters such as:
  - ✓ asserting control-signals
  - ✓ timing of placing information on bus

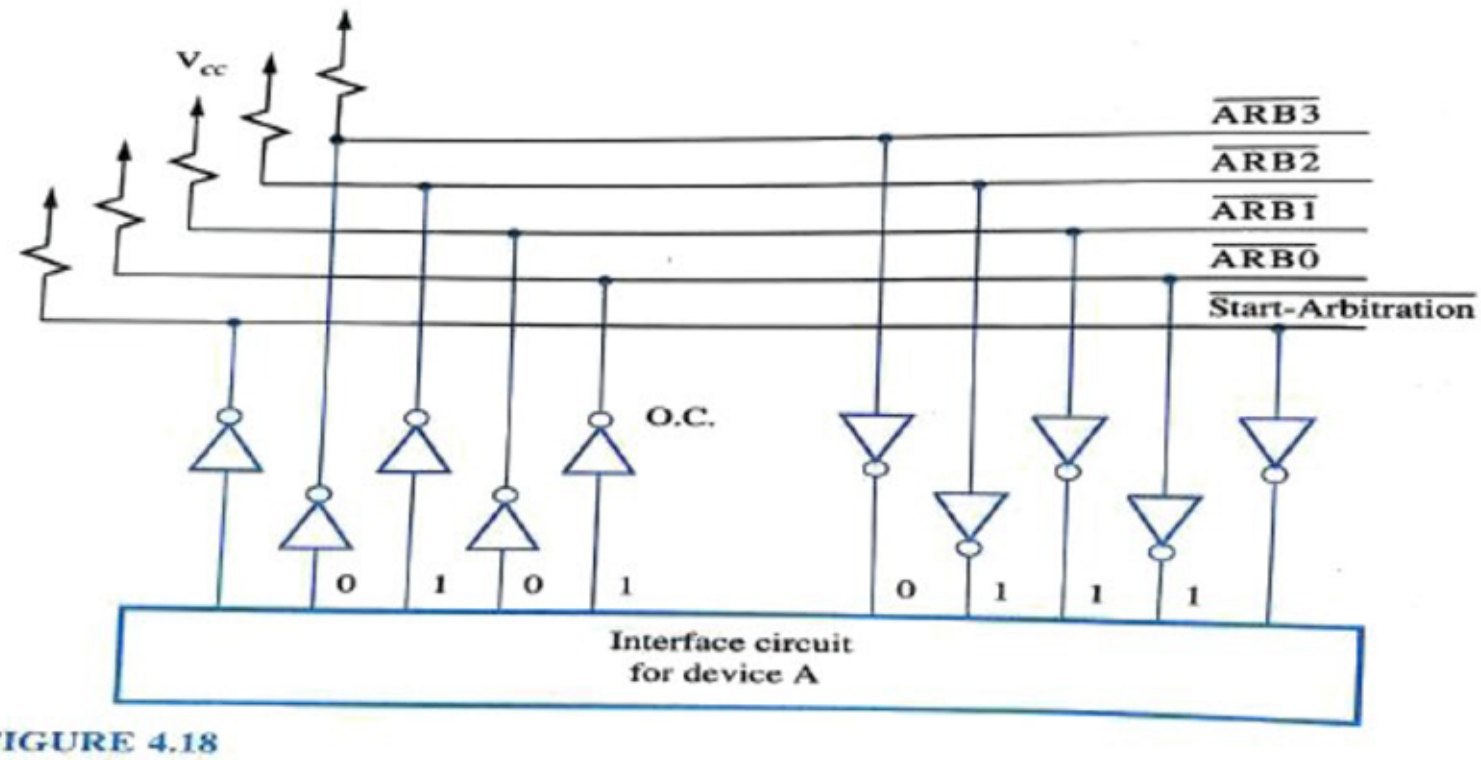



**FIGURE 4.18**
Connection of device interface circuit to priority arbitration lines.

- ✓ rate of data-transfer

A typical bus consists of three sets of lines:

1. Address,
2. Data and
3. Control lines.

- ➢ Control-signals specify whether a read or a write operation is to be performed.
- ➢ R/W line specifies
  - → read operation when R/W=1
  - → write operation when R/W=0
- ➢ In data-transfer operation, one device plays the role of a ***bus-master (initiator)*** which initiates data transfers by issuing Read or Write commands on.
- ➢ Device addressed by master is referred to as a ***slave (target).***

Timing of data transfers over a bus is classified into two types:

1. Synchronous and
2. Asynchronous

## SYNCHRONOUS BUS

- ➢ All devices derive timing-information from a common clock-line.
- ➢ Equally spaced pulses on this line define equal time intervals.
- ➢ Each of these intervals constitutes a bus-cycle during which one data transfer can take place.

**A sequence of events during a read operation:**

- ➢ At time $t_0$, the master (processor)
  - ✓ places the device-address on address-lines &
  - ✓ Sends an appropriate command on control-lines (Figure 4.19).
- ➢ Information travels over bus at a speed determined by its physical & electrical characteristics.

- Clock pulse width ($t_1$-$t_0$) must be longer than the maximum propagation-delay between 2 devices connected to bus.
- Information on bus is unreliable during the period $t_0$ to $t_1$ because signals are changing state.
- Slave places requested input-data on data-lines at time $t_1$.
- At end of clock cycle(at time $t_2$), master strobes(captures) data on data-lines into its input-buffer
- For data to be loaded correctly into any storage device (such as a register built with flip-flops), data must be available at input of that device for a period greater than setup-time of device.

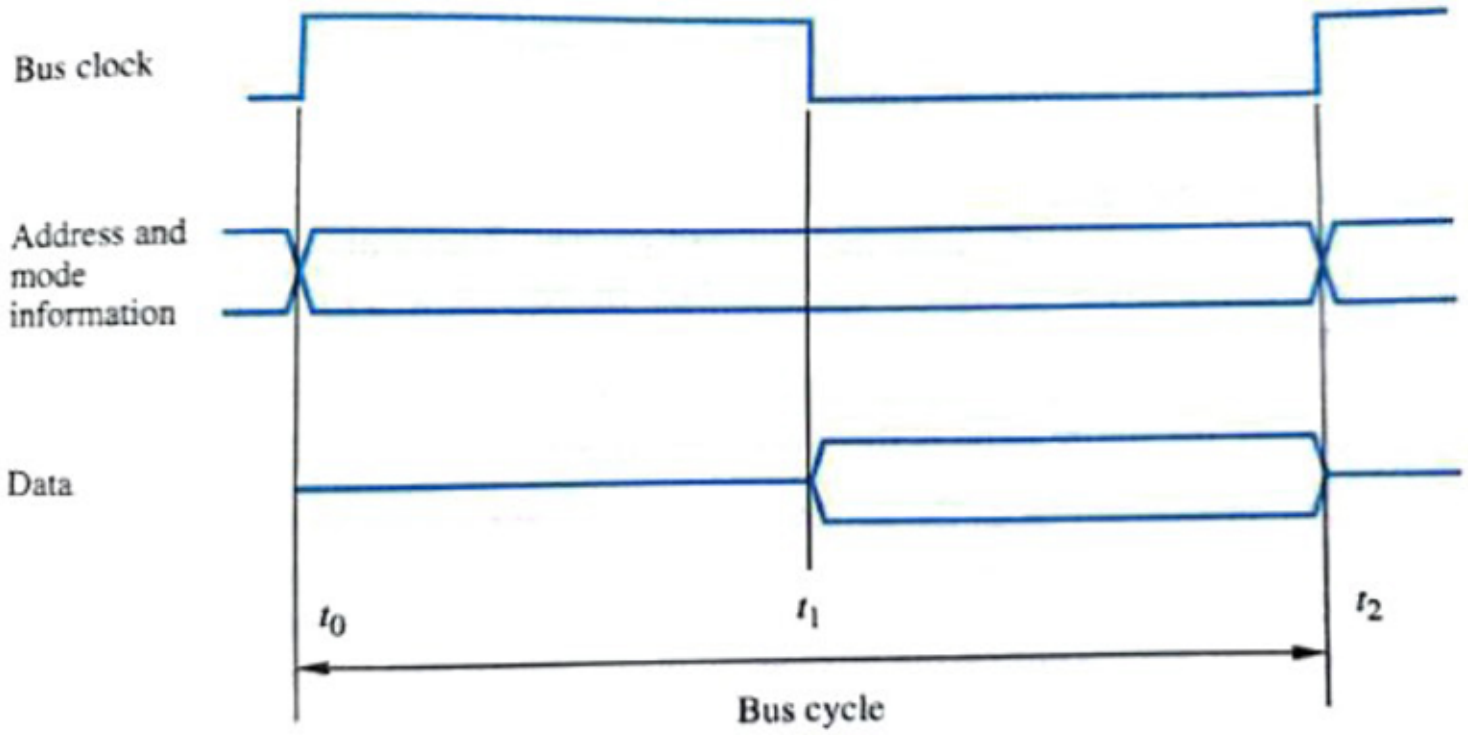



FIGURE 4.19
Timing of an input transfer on a synchronous bus.

## ASYNCHRONOUS BUS

This method uses handshake-signals between master and slave for coordinating data transfers.

- There are 2 control-lines:
  1. Master-ready (MR) to indicate that master is ready for a transaction
  2. Slave-ready (SR) to indicate that slave is ready to respond

**The read operation proceeds as follows:**

- At $t_0$, master places address- & command-information on bus. All devices on bus begin to decode this information.
- At $t_1$, master sets MR-signal to 1 to inform all devices that the address- & command-information is ready.
- At $t_2$, selected slave performs required input-operation & sets SR signal to 1 (Figure 4.20).
- At $t_3$, SR signal arrives at master indicating that the input-data are available on bus skew.

- At $t_4$, master removes address- & command-information from bus.
- At $t_5$, when the device-interface receives the 1-to-0 transition of MR signal, it removes data and SR signal from the bus. This completes the input transfer

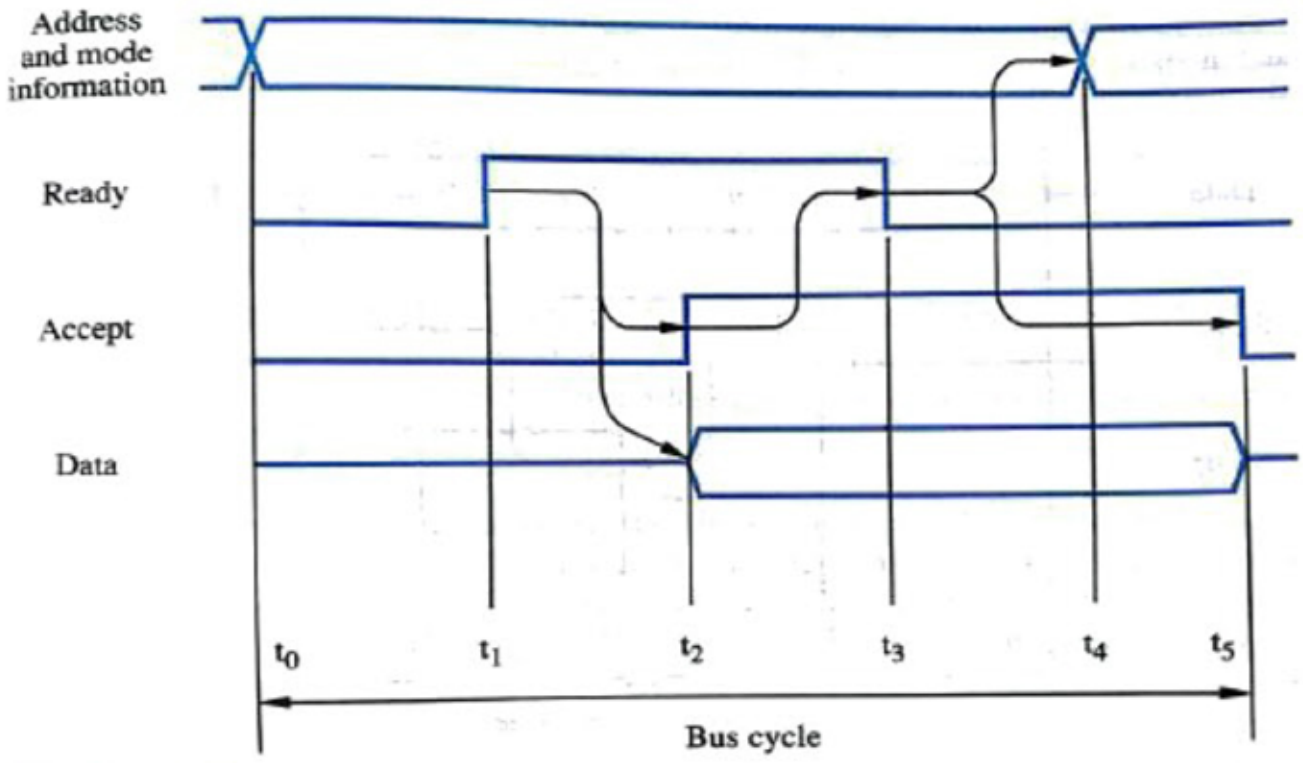



**FIGURE 4.20**
Handshake control of data transfer during an input operation.

The timing diagram for output operation is given in the following figure.

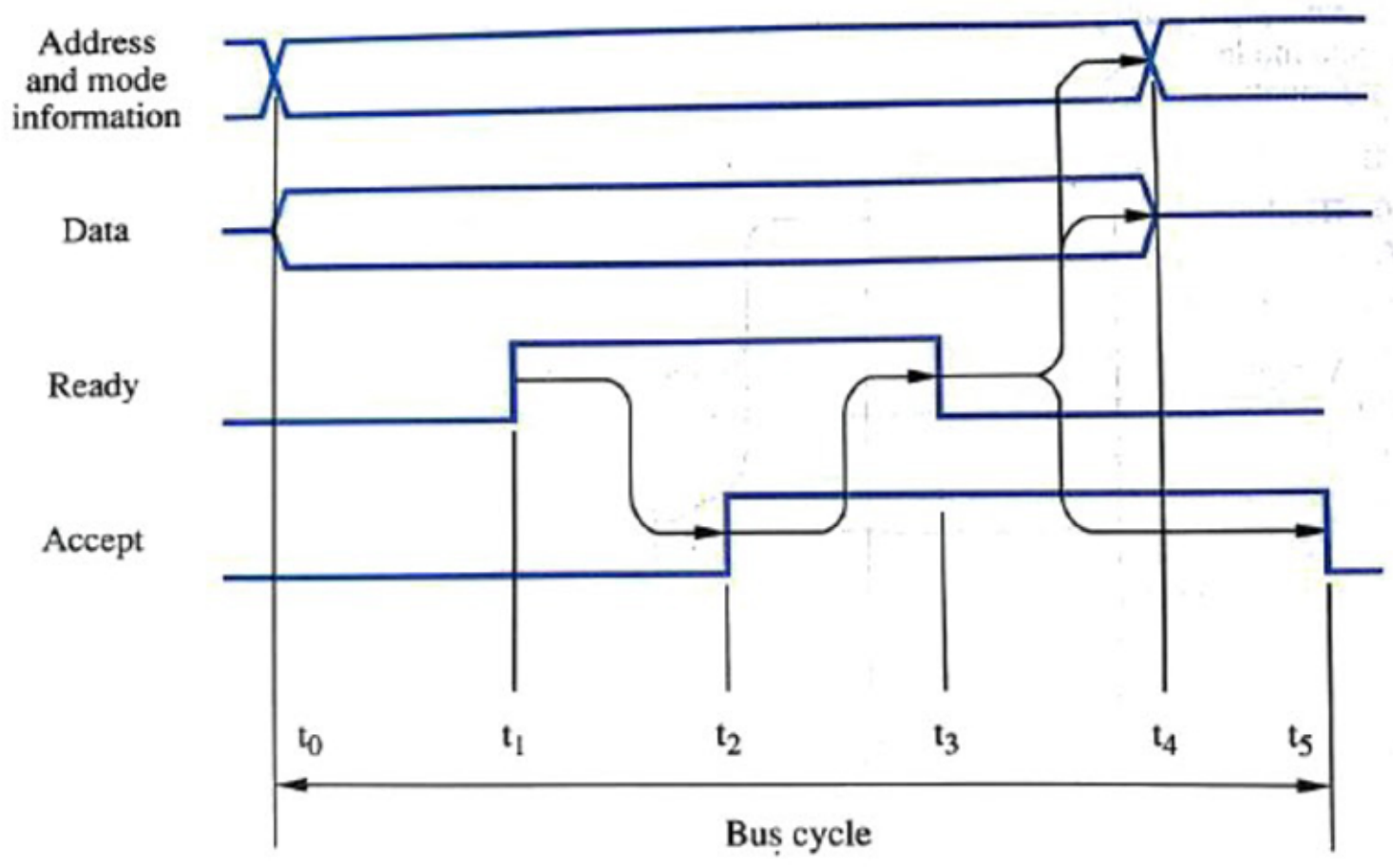



**FIGURE 4.21**
Handshake control of data transfer during an output operation.

Output operation is same as input operation, In this case, the master places the output data on the data lines at the same time that it transmits the address and command information. The selected slave strobes the data into its output buffer when into output buffer when it

receives the Master-ready signal and indicates that it has done so by setting the Slave-ready signal to 1. The remainder of the signal is identical to the input operation.

## INTERFACE CIRCUITS

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a *port,* and it can be either a parallel or a serial port.

1. A ***parallel*** port transfers multiple bits of data simultaneously to or from the device.
2. A ***serial*** port sends and receives data one bit at a time.

**An I/O interface does the following:**

1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control.
5. Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port

**Parallel port**

- The hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open.
- When a key is pressed, its switch closes and establishes a path for an electrical signal.
- This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character.

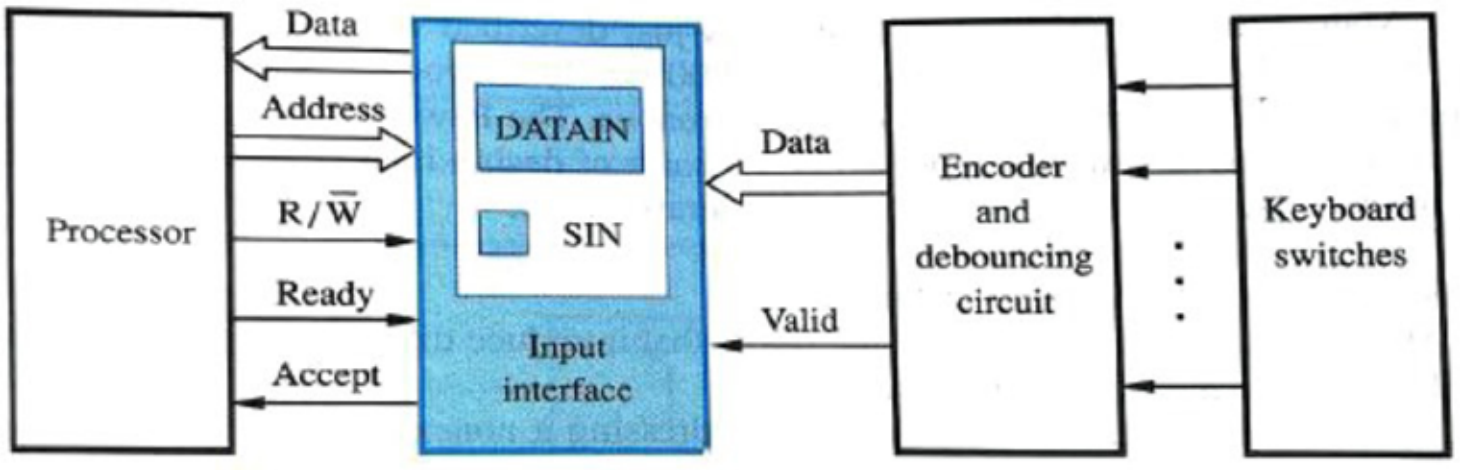



FIGURE 4.22
Keyboard connected to processor.

The output of the encoder consists of the bits that represent the encoded character and one control signal called Valid, which indicates that a key is being pressed. This information is

sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN. When a key is pressed, the valid signal changes from 0 to 1, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1.

The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals Master-ready and Slave-ready, as indicated in figure 4.21. The third control line, R/W distinguishes read and write transfers.

Figure 4.22 shows a suitable circuit for an input interface. The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the address that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit D0, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit A0 determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slave-ready signal when either Read-status or Read-data is equal to 1.

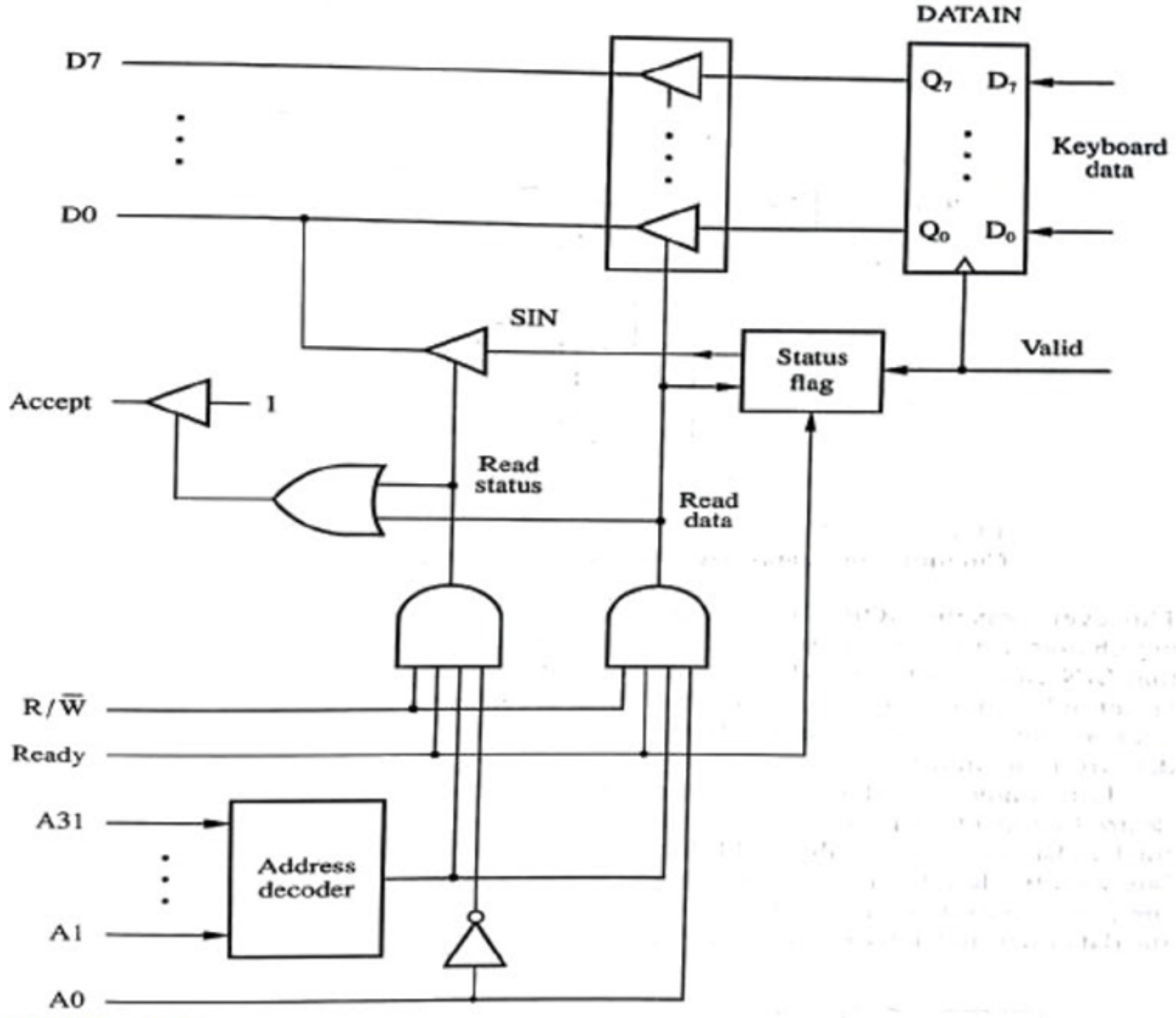



**FIGURE 4.23**
An input interface circuit connecting a keyboard to an asynchronous bus.

A possible implementation of the status flag circuit is given in Figure 4.24. The KIN flag is the output of a NOR latch connected as shown. A flip-flop is set to 1 by the rising edge on the valid signal line. This event changes the state of the NOR latch to set KIN to 1, but only when Master-ready is low. The reason for this additional condition is to ensure that KIN does not change state while being read by the processor. Both the flip-flop and the latch are reset to 0 when Read-data becomes equal to 1, indicating that KBD_DATA is being read.

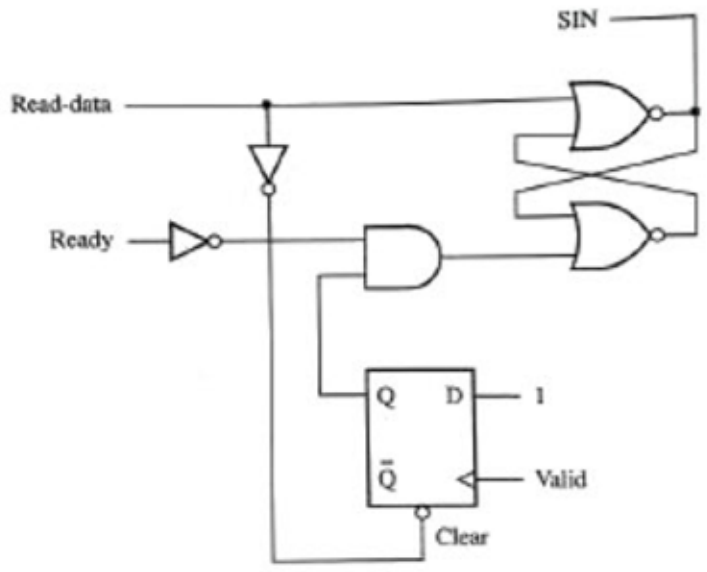



FIGURE 4.24
Circuit for the status flag block in Figure 4.23.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in figure 4.25. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the *Valid* signal. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the Valid signal.

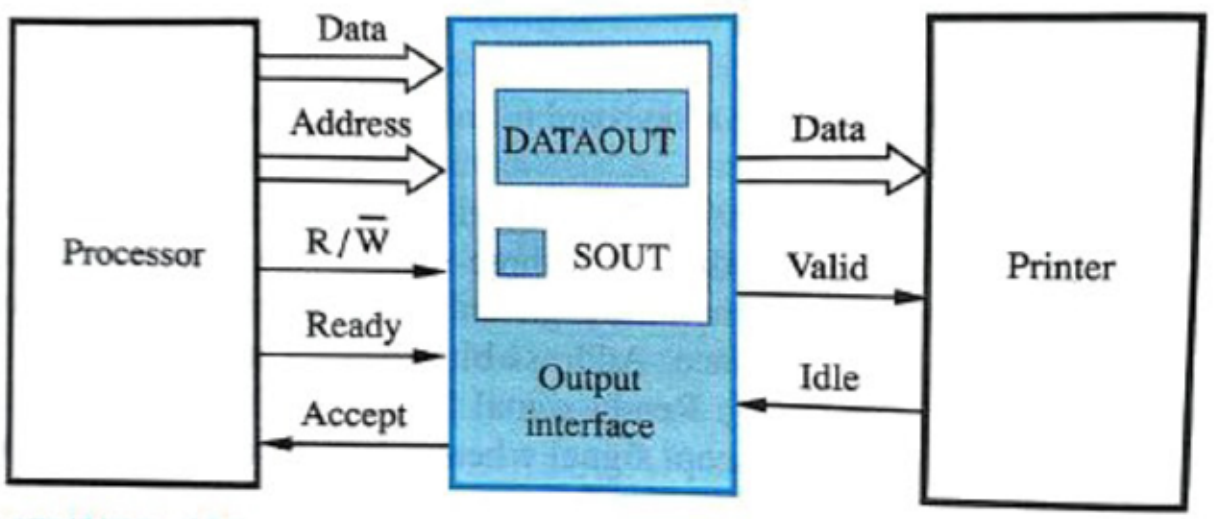



FIGURE 4.25
Printer connected to processor.

The interface contains a data register, DATAOUT, and a status flag, SOUT. The SOUT flag is set to 1 when the printer is ready to accept another character and it is cleared to 0 when a new character is loaded into DATAOUT by the processor.

Figure 4.26 shows an implementation of this interface. Its operation is similar to the input interface of figure 4.26. The only significant difference is the handshake control circuit.

The input and output interfaces just described can be combined into a single interface as shown in figure 4.27.

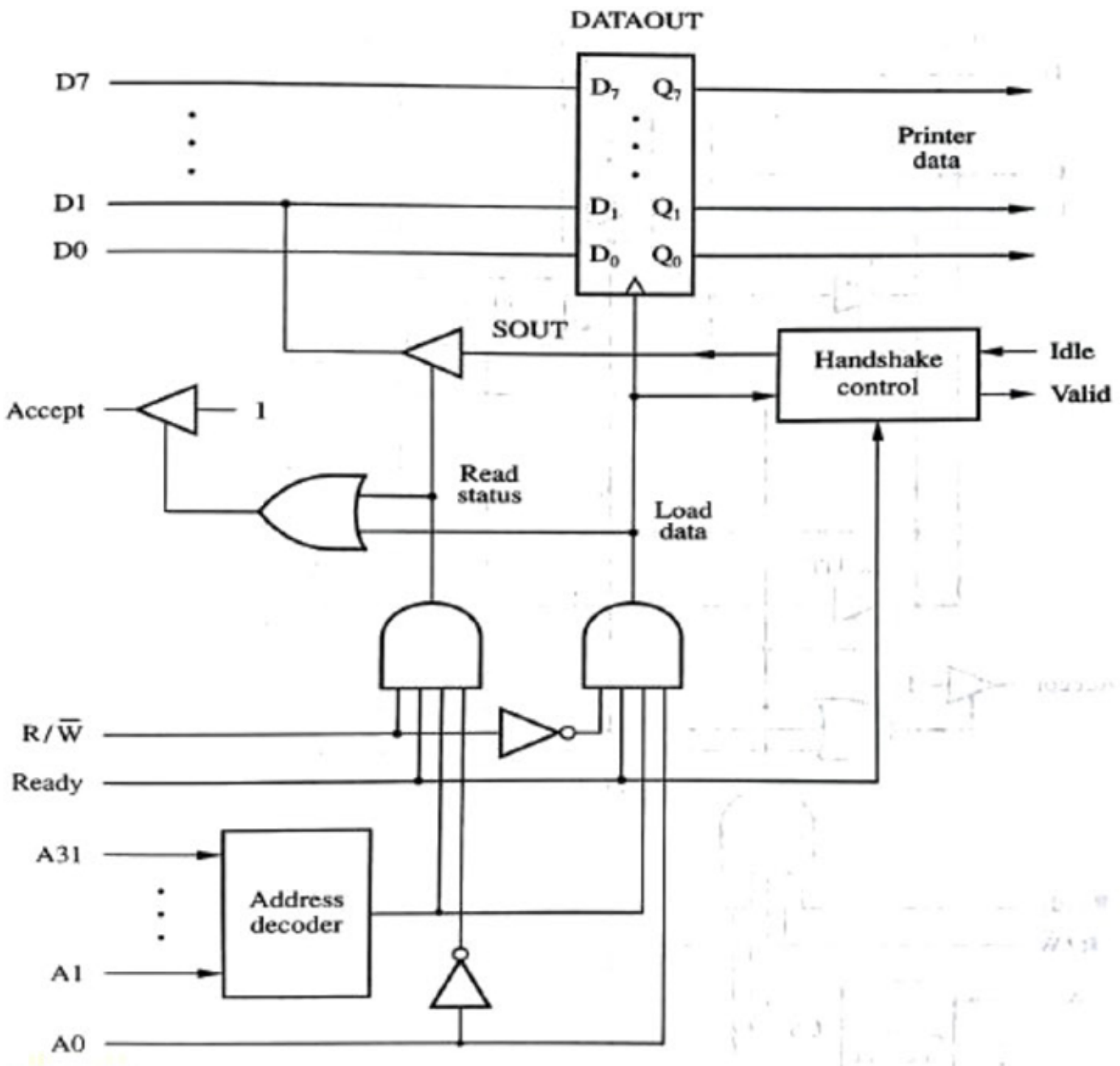



**FIGURE 4.26**
An output interface circuit connecting a printer to an asynchronous bus.

The circuit in figure 4.28 has separate input and output data lines for connection to an I/O device. A more flexible parallel port is created if the data lines to I/O devices are bidirectional. Figure 17 shows a general-purpose parallel interface circuit that can be configured in a variety of ways. Data lines P7 through P0 can be used for either input or output purposes. For increased flexibility, the circuit makes it possible for some lines to serve as inputs and some lines to serve as outputs, under program control. The DATAOUT register is connected to these lines via three-state drivers that are controlled by a data direction register, DDR. The processor can write any 8-bit pattern into DDR. For a given bit, if the DDR value is 1, the corresponding data line acts as an output line; otherwise, it acts as an input line.

Two lines C1 and C2 are provided to control the interaction between the interface circuit and the I/O device is serves. These lines are also programmable. Line C2 is bidirectional to provide several different modes of signaling, including the handshake.

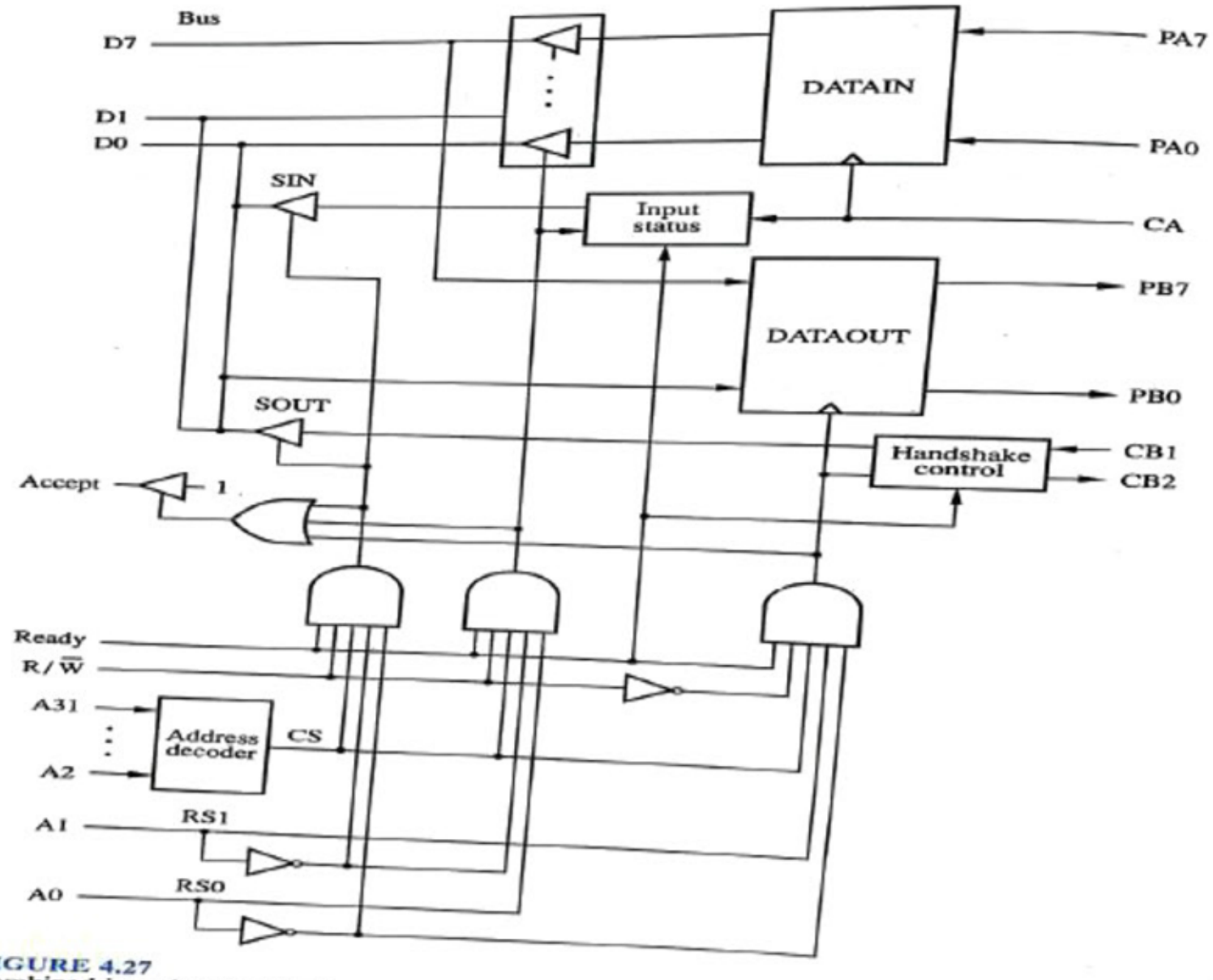



FIGURE 4.27
Combined input/output interface circuit.

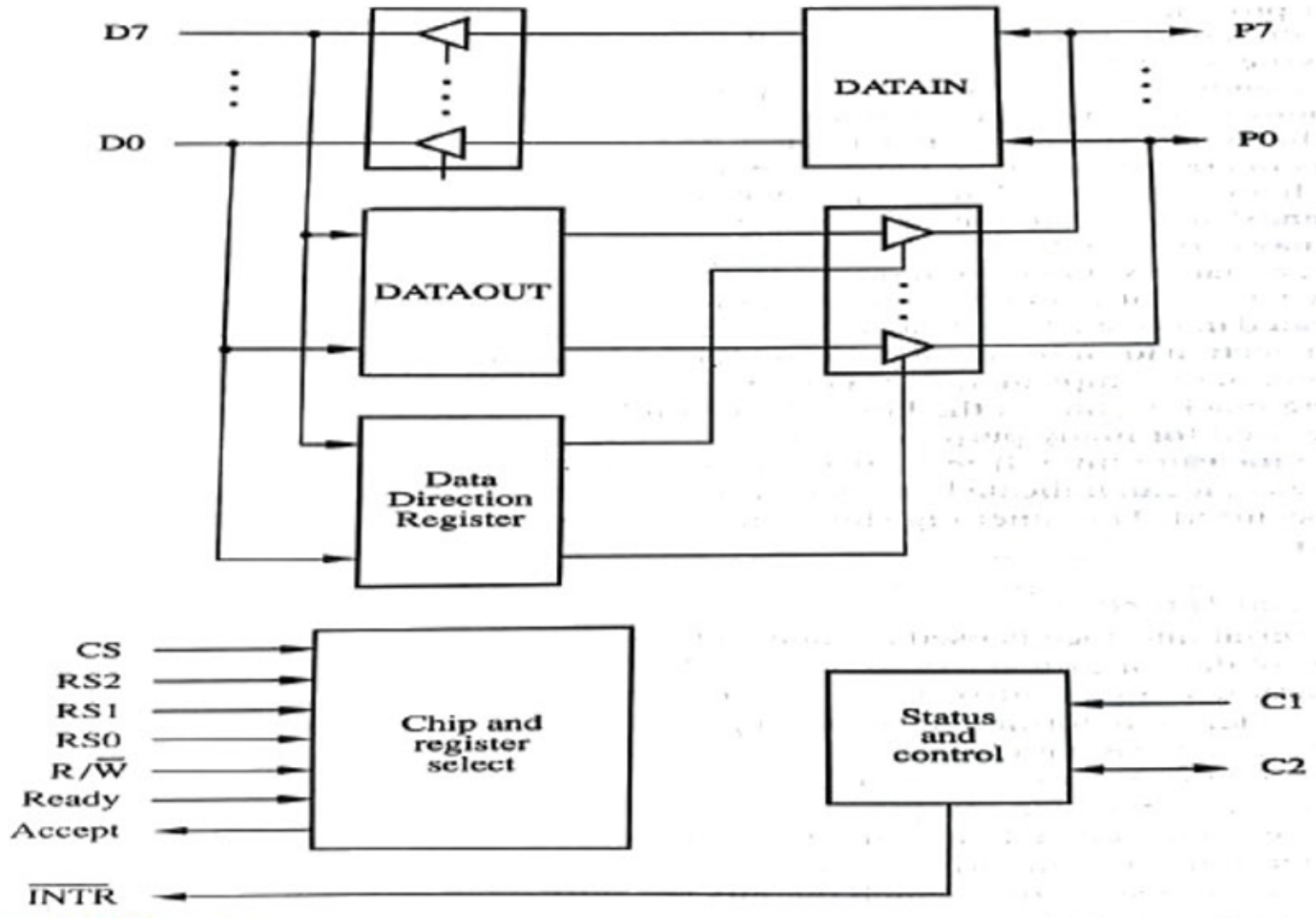



FIGURE 4.28
A bidirectional 8-bit parallel interface.

**Serial Port**

A Serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of an interface circuit for a serial port is that it is capable of communicating in a bit-serial fashion on the device side and in a bit-parallel fashion on the bus side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in figure 4.29. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are loaded into the output register, from which the bits are shifted out and sent to the I/O device.

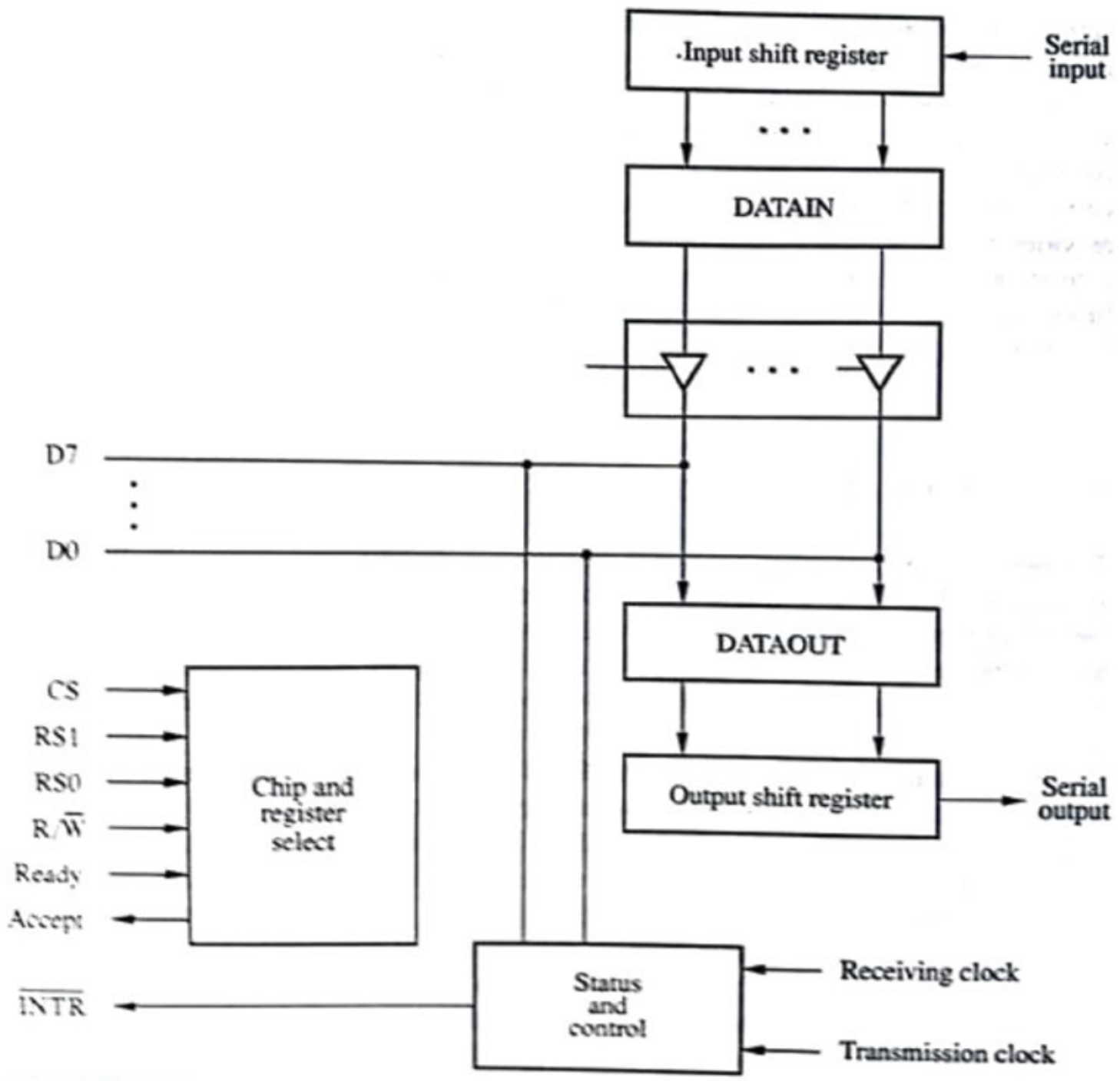



FIGURE 4.29
A serial interface.

The double buffering used in the input and output paths are important. A simpler interface could be implemented by turning DATAIN and DATAOUT into shift registers and eliminating the shift registers in figure 7.15. However, this would impose awkward

restrictions on the operation of the I/O device; after receiving one character from the serial line, the device cannot start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to allow the processor to read the input data. With the double buffer, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of serial data. An analogous situation occurs in the output path of the interface.

Because serial interfaces play a vital role in connecting I/O devices, several widely used standards have been developed. A standard circuit that includes the features of our example in figure 7.15 is known as a Universal Asynchronous Receiver Transmitter (UART). It is intended for use with low-speed serial devices. Data transmission is performed using the asynchronous start-stop format. To facilitate connection to communication links, a popular standard known as RS-232-C was developed.

## STANDARD I/O INTERFACES

The processor bus is the bus defied by the signals on the processor chip itself. Devices that require a very high-speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a bridge that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices. It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM developed a bus they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ***ANSI*** (American National Standards Institute), or international bodies such as ***ISO*** (International Standards Organization) have blessed these standards and given them an official status.

A given computer may use more than one bus standards. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

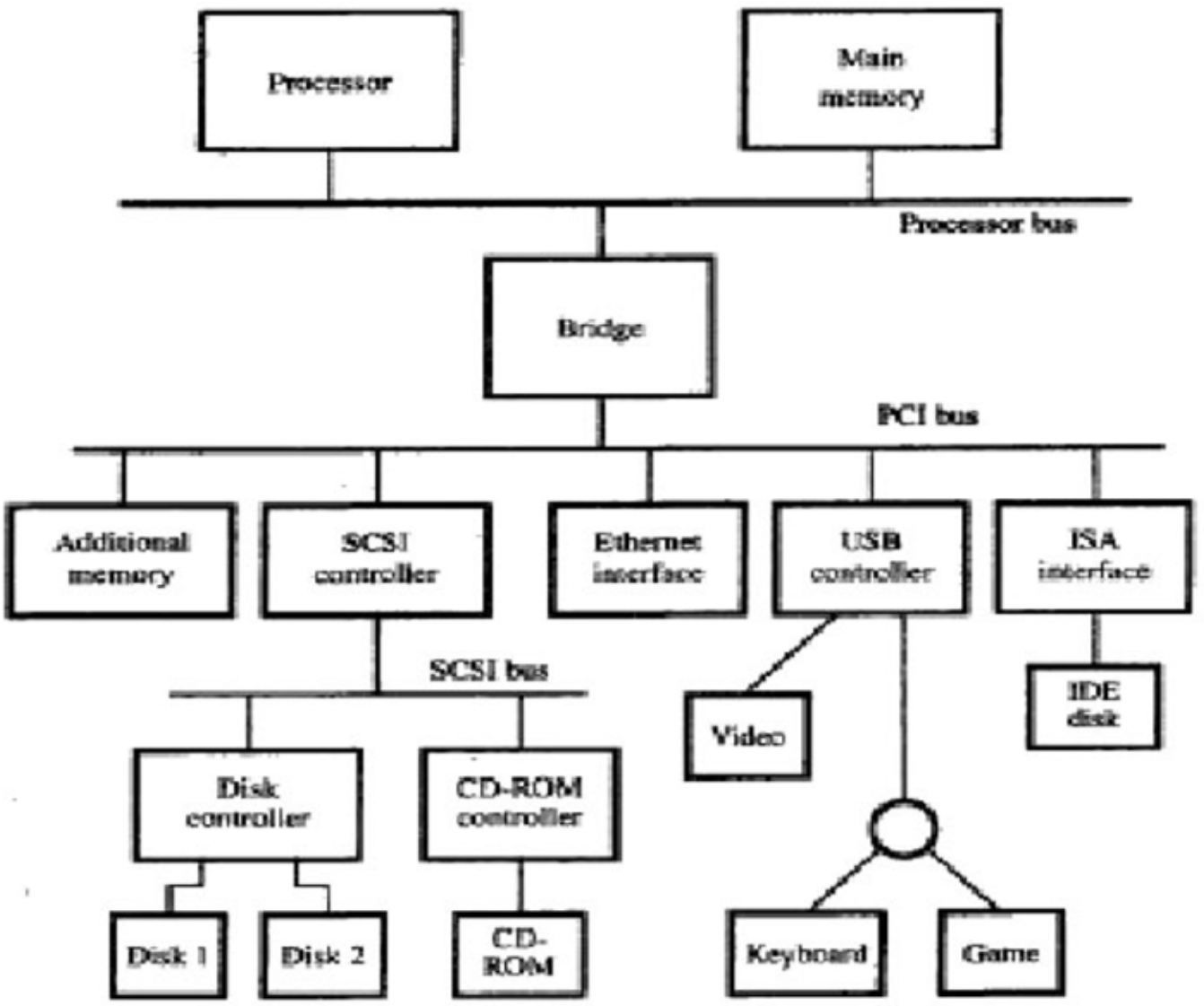



**Figure 4.38** An example of a computer system using different interface standards.

## Peripheral Component Interconnect (PCI) Bus:-

The PCI bus is a good example of a system bus that grew out of the need for standardization. It supports the functions found on a processor bus bit in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor.

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC At computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Micro-channel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

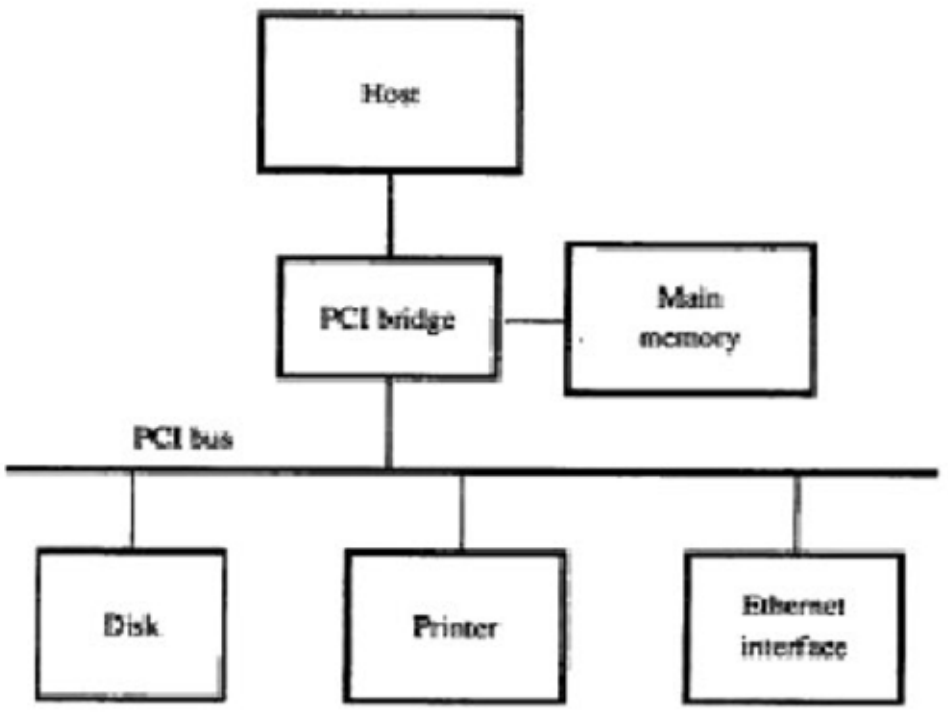



**Figure 4.39** Use of a PCI bus in a computer system.

### Data Transfer

The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. However, as noted, the system designer may choose to use memory-mapped I/O even when a separate I/O address space is available. In fact, this is the approach recommended by the PCI its plug-and-play capability. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

The signaling convention on the PCI bus is similar to the one used, we assumed that the master maintains the address information on the bus until data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected. The slave can store the address in its internal buffer. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction because the number of wires on a bus is an important cost factor. This approach in used in the PCI bus.

**Device Configuration:**

When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it. The PCI simplifies this process by incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices is accessible in the configuration address space. The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it determines whether the device is a printer, a

keyboard, an Ethernet interface, or a disk controller. It can further learn bout various device options and characteristics.

**Table 4.3** Data transfer signals on the PCI bus.

| Name | Function |
|---|---|
| CLK | A 33-MHz or 66-MHz clock. |
| FRAME# | Sent by the initiator to indicate the duration of a transaction. |
| AD | 32 address/data lines, which may be optionally increased to 64. |
| C/BE# | 4 command/byte-enable lines (8 for a 64-bit bus). |
| IRDY#, TRDY# | Initiator-ready and Target-ready signals. |
| DEVSEL# | A response from the device indicating that it has recognized its address and is ready for a data transfer transaction. |
| IDSEL# | Initialization Device Select. |

Devices are assigned addresses during the initialization process. This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one. Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#.

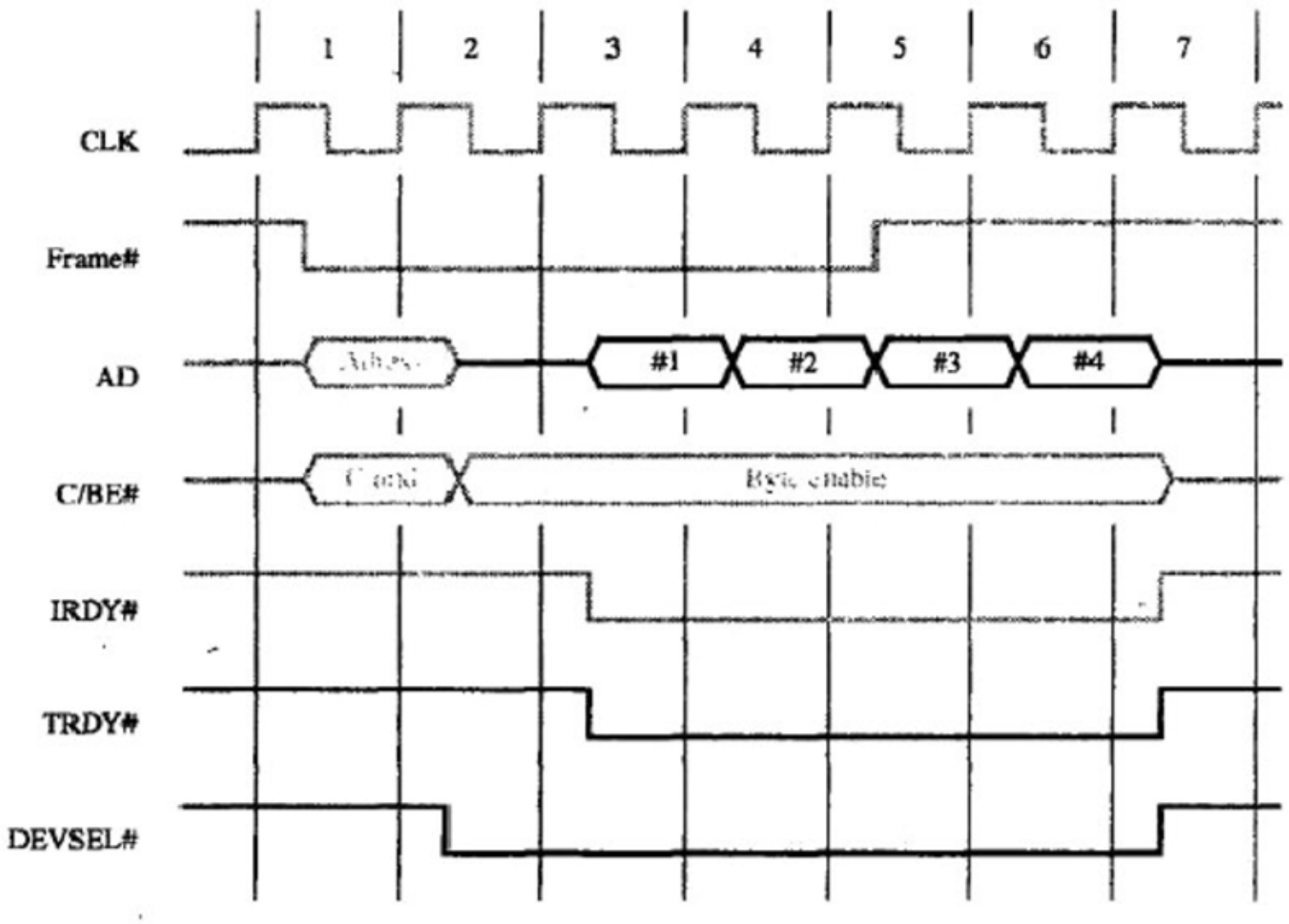



**Figure 4.40** A read operation on the PCI bus.

The PCI bus has gained great popularity in the PC word. It is also used in many other computers, such as SUNs, to benefit from the wide range of I/O devices for which a PCI interface is available. In the case of some processors, such as the Compaq Alpha, the PCI-processor bridge circuit is built on the processor chip itself, further simplifying system design and packaging.

**SCSI Bus:-**

The acronym SCSI stands for Small Computer System Interface.

The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time.

There are also several options for the electrical signaling scheme used. Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory.

A controller connected to a SCSI bus is one of two types – an ***initiator*** or a ***target.***

An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator.

The processor sends a command to the SCSI controller, which causes the following sequence of event to take place:

1. The SCSI controller, acting as an *initiator*, contends for control of the bus.
2. When the initiator wins the *arbitration* process, it selects the target controller and hands over control of the bus to it.

3. The *target* starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, *sends a message* to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a *command* to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
6. The target *transfers the contents* of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a *command* to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of these transfers, the logical connection between the two controllers is terminated.
8. As the initiator controller *receives the data*, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends as *interrupt* to the processor to inform it that the requested operation has been completed

**Bus signal:**

The bus signals are summarized in table 4.1

- All signal names are preceded by minus sign. This indicates that the signals are active, or that a data line is equal to 1, when they are in the low-voltage state.
- The bus has no address lines instead the data lines are used to identify the bus controllers involved during the selection and reselection process and during bus arbitration.

***Arbitration:***

- The bus is free when the –BSY signal is in the inactive (high voltage) state.
- When two or more controller tries to access the bus, SCSI bus uses a simple distributed arbitration scheme. As illustrated in figure 4.42, in which controllers 2 and 6 request the use of the bus simultaneously.

***Selection:***

- Having won arbitration, controller 6 continues to assert –BSY and –DB6 (its address).
- It indicates that it wishes to select controller 5 by asserting the –SEL and then –DB5 lines. Other controllers should stop once –SEL is active.
- The selected target controller responds by asserting –BSY. The selection process is now complete and the target controller ( 5) is asserting –BSY

➢ From this point on, controller 5 has control of the bus, as required for the information transfer phase.

TABLE 4.1
**The SCSI bus signals**

| Category | Name | Function |
|---|---|---|
| Data | −DB(0) through −DB(7) | Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection, and reselection phases |
| | −DB(P) | Parity bit for the data bus |
| Phase | −BSY | Busy: Asserted when the bus is not free |
| | −SEL | Selection: Asserted during selection and reselection |
| Information type | −C/D | Control/Data: Asserted during transfer of control information (command, status, or message) |
| | −MSG | Message: Indicates that the information being transferred is a message |
| Handshake | −REQ | Request: Asserted by a target to request a data transfer cycle |
| | −ACK | Acknowledge: Asserted by the initiator when it has completed a data transfer operation |
| Direction of transfer | −I/O | Input/Output: Asserted to indicate an input operation (relative to the initiator) |
| Other | −ATN | Attention: Asserted by an initiator when it wishes to send a message to a target |
| | −RST | Reset: Causes all device controls to disconnect from the bus and assume their start-up state |

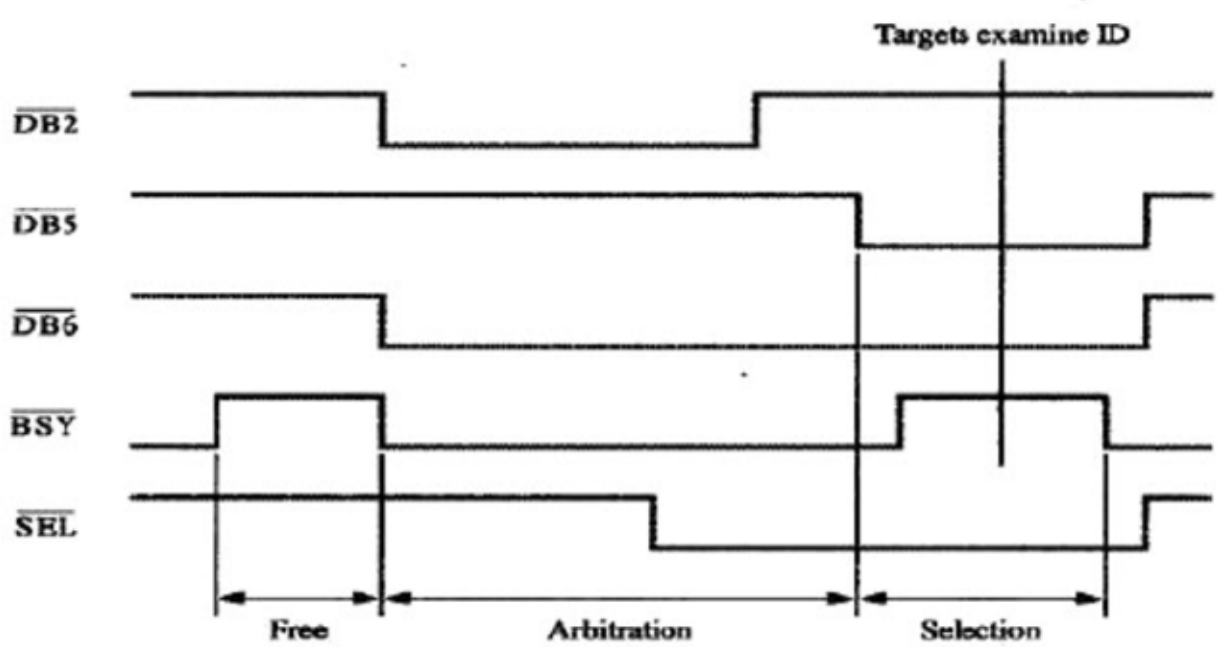



**Figure 4.42** Arbitration and selection on the SCSI bus. Device 6 wins arbitration and selects device 2.

## UNIVERSAL SERIAL BUS (USB)

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and full-speed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices.

The USB has been designed to meet several key objectives:

- Provides a simple, low-cost and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer.
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections.
- Enhance user convenience through a "plug-and-play" mode of operation.

### Port Limitation:

The parallel and serial ports described in previous section provide a general- purpose point of connection through which a variety of low-to medium-speed devices can be connected to a computer. For practical reasons, only a few such ports are provided in a typical computer.

### Device Characteristics:

The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly. A variety of simple devices that may be attached to a computer generate data of a similar nature – low speed and asynchronous. Computer mice and the controls and manipulators used with video games are good examples.

### Plug-and-Play:

The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

### USB Architecture:

The above discussion points to the need for an interconnection system that combines low cost, flexibility, and high data-transfer bandwidth. Also, I/O devices may be located at some distance from the computer to which they are connected. The requirement for high bandwidth would normally suggest a wide bus that carries 8, 16, or more bits in parallel. However, a large number of wires increases cost and complexity and is inconvenient to the user. Also, it is difficult to design a wide bus that carries data for a long distance because of the data skew

problem discussed. The amount of skew increases with distance and limits the data that can be used.

A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements. Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.

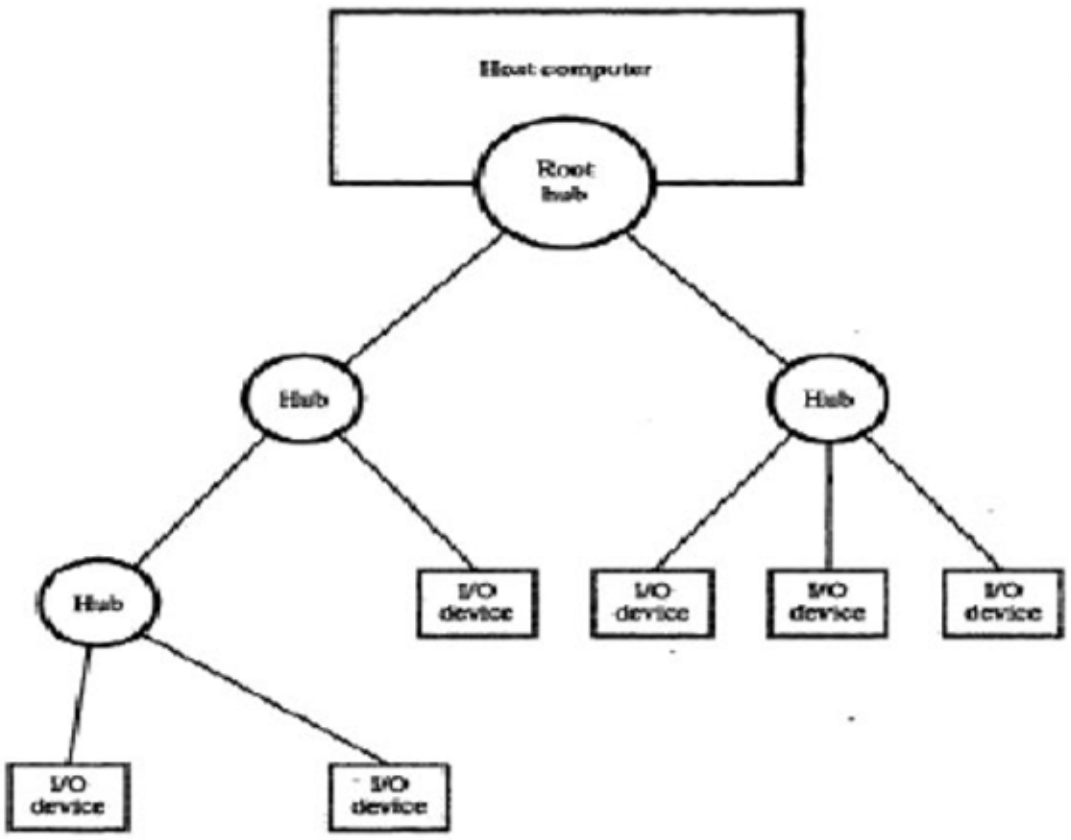



Figure 4.43 Universal Serial Bus tree structure.

To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure shown in figure 4.43. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices.

In the figure 4.44 hub A is connected to the root hub by a high sppe link. The hub serves one high speed device, C and one low speed device D.

The USB standard specifies the hardware details of USB interconnections as well as organization requirements of the host software. The purpose of the USB software is to provide bidirectional communication links between application software and I/O devices. These links are called ***pipes.***

Software that transfers data to or from a given I/O device is called the ***device driver*** for that device.

**Addressing**

A device usually has several addressable locations to enable the software to send and receive control and status information and to transfer data.

When a USB is connected to a computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.

Each device on the USB, whether it is hub or an I/O device, is assigned a 7-bit address; this address is local to the USB tree and is not related in any way to the addresses used on the processor bus.

Location in the device to or from which data transfer can take place, such as status, control, and data registers, are called ***endpoints.***

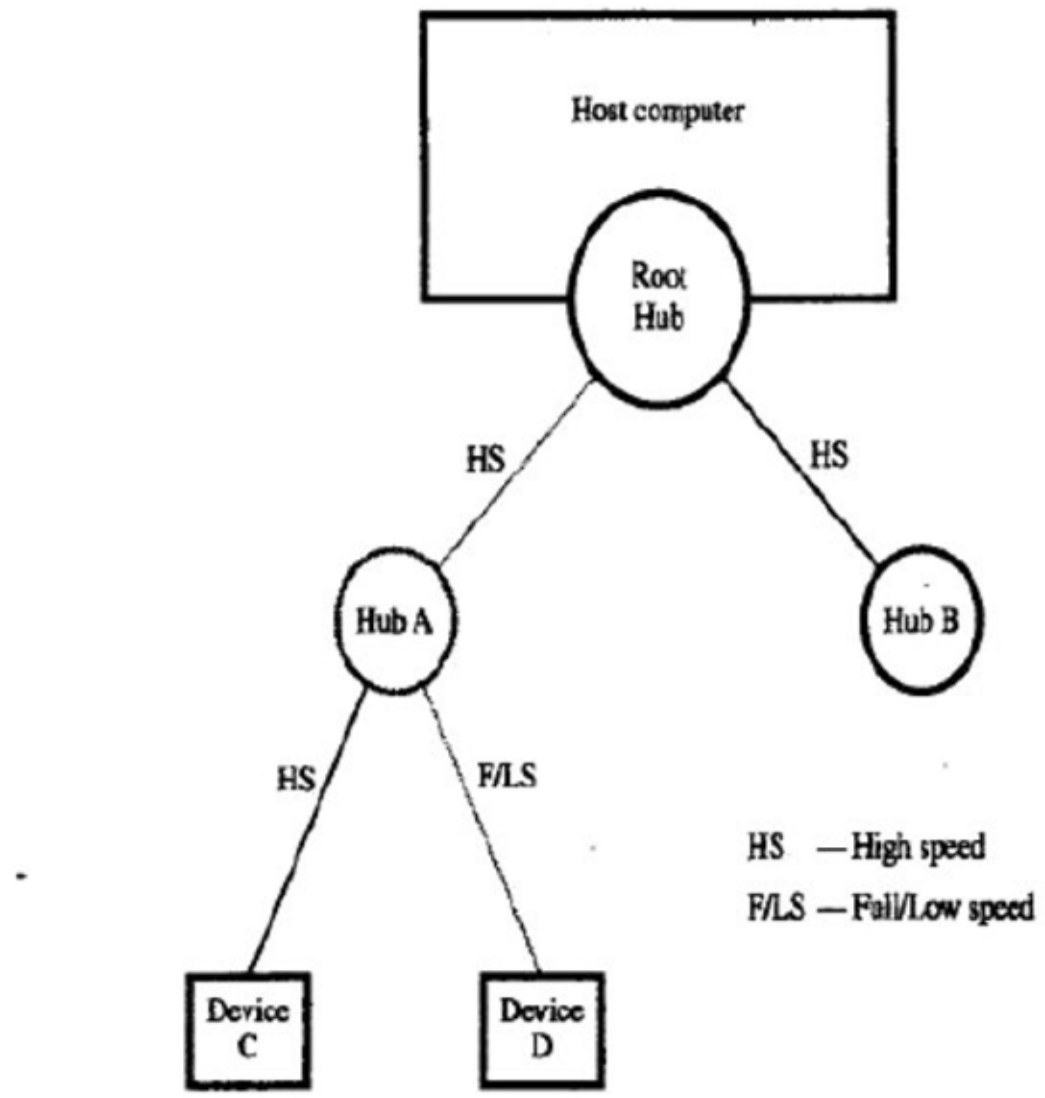



**Figure 4.44** Split bus operation.

## USB Protocols

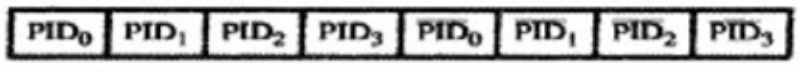
All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information.

- The information transferred on the USB can be divided into two broad categories: control and Data.
- Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error
- Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets

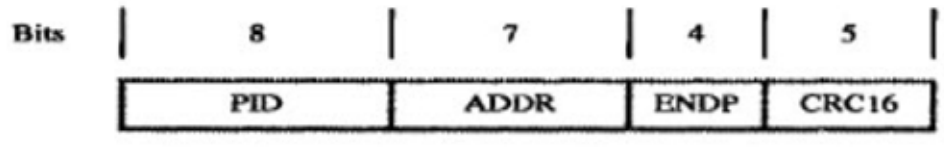
The packet consists of one or more field containing different kinds of information.

- The first field of any packet is called the ***packed identifier (PID)*** which identifies the packet type. There are four bits of information in this field, but they are transmitted twice.
- First time they are sent with their true values, and the second time with each bit complemented as shown in following figure. This enables the receiving device to verify that the PID byte has been received correctly.
- The four PID bits identify one of 16 different packet types like ACK and control packet.
- ***A token packet*** starts with the PID followed by 7 bit *address* of a device and the 4-bit *endpoint* number with in that device. The packet ends with 5 bits for error checking using the method Cyclic Redundancy Check (CRC).
- ***Data packet*** starts with PID followed by up to 8192 bits of data, then 16bit error checking bits.

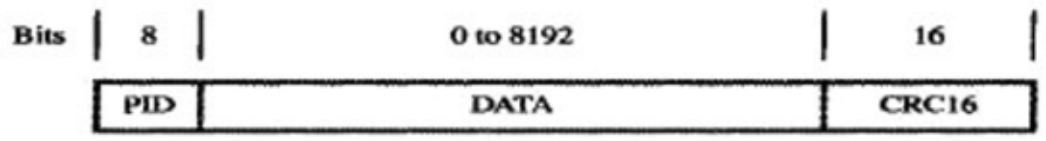
PID's used to identify the data packets are 0, 1 and 2




(a) Packet identifier field




(b) Token packet, IN or OUT




(c) Data packet

**Figure 4.45** USB packet formats.

The figure 4.46 shows the host computer sends a token packet of type OUT to the hub, followed by a data packet containing the output data. The PID field of the data packet identifies it as data packet number 0. The hub verifies the transmission has been error free by checking the error control bits, and then sends an acknowledgement packet (ACK) back to host. Similarly it is conducted between hub and I/O device

### *Isochronous Traffic on USB*

One of the key objectives of the USB is to support the transfer of isochronous data, such as sampled voice in simple manner. Device that generates or receives isochronous data require a time reference to control the sampling process. To provide this reference, transmission over the USB is divided into frames of equal length. A frame is 1ms long for low-and full-speed data. The root hub generates a Start Of Frame control packet (SOF) precisely once every 1ms to mark the beginning of new frame.

Figure 4.47a shows SOF packet format and figure 4.47b shows the frame example of SOF packet.

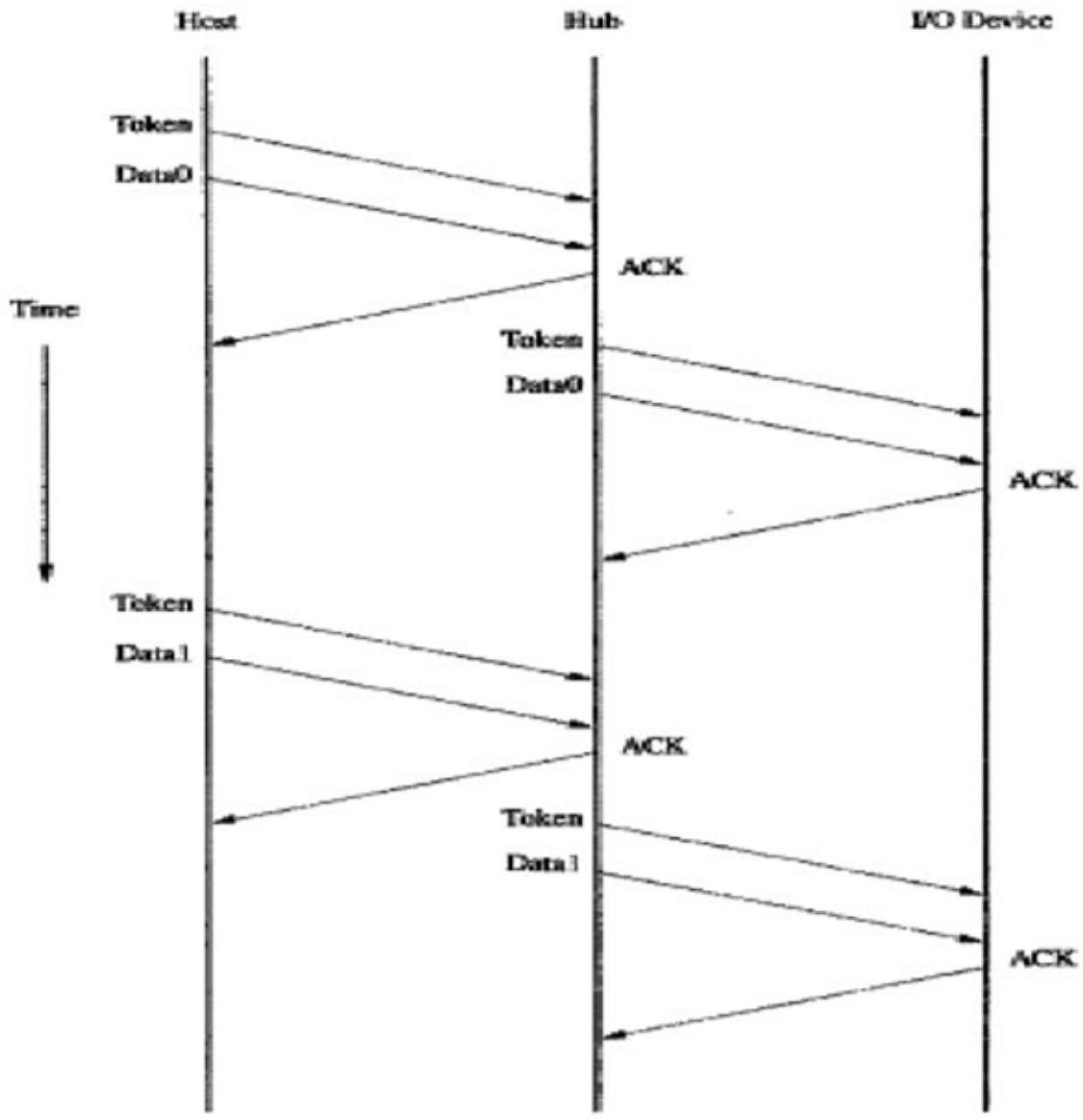
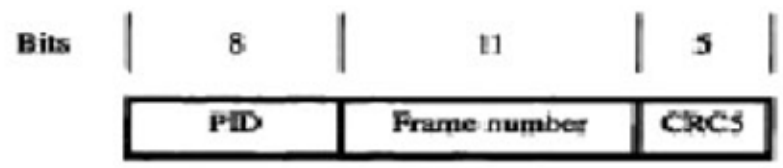
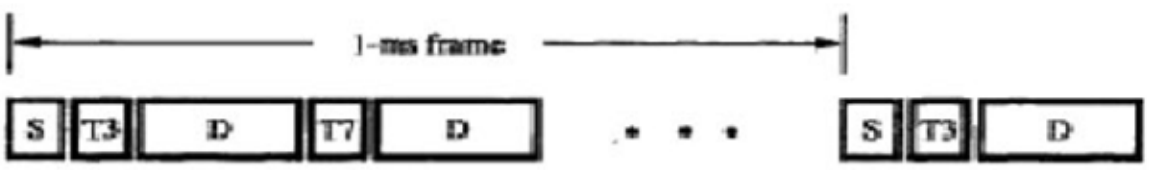



**Figure 4.46** An output transfer.




(a) SOF Packet




S — Start-of-frame packet
Tn — Token packet, address = n
D — Data packet
A — ACK packet

(b) Frame example

**Figure 4.47** USB frames.