### 11.4.8.6 Processor Control Instructions

These instructions do not perform computations. They are used to do tasks such as initializing the 8087, enabling interrupts, writing the status word to memory, etc.

**FINIT/FNINT :** Initializes 8087. Disables interrupt output, sets stack pointer to register 7, sets default status.

**FDISI/FNDISI :** Disables the 8087 interrupt output pin so that it cannot cause an interrupt when an exception (error) occurs.

**FENI/FNENI :** Enables 8087 interrupt output so it can cause an interrupt when an exception occurs.

**FLDCW source :** Loads a status word from a memory location into the 8087 status register. This instruction should be preceded by the FCLEX instruction to prevent a possible exception response if an exception bit in the status word is set.

**FSTCW/FNSTCW destination :** Copies the 8087 control word to a memory location. You can determine its current value with 8086 instructions.

**FSTSW/FNSTW destination :** Copies the 8087 status word to a memory location. You can check various status bits with 8086 instructions and take further action on the state of these bits.

**FCLEX/FNCLEX :** Clears all of the 8087 exception flag bits in the status register. Unasserts BUSY and INT outputs.

**FSAVE/FNSAVE destination :** Copies the 8087 control word, status word, pointers, and entire register stack to 94-byte area of memory. After copying all of this the FSAVE/FNSAVE instruction initializes the 8087.

**FRSTOR source :** Copies a 94-byte area of memory into the 8087 control register, status register, pointer registers, and stack registers.

**FSTENV/FNSTENV destination :** Copies the 8087 control register, status register, tag words, and exception pointers to a series of memory locations. This instruction does not copy the 8087 register stack to memory as the FSAVE/FNSAVE instruction does.

**FLDENV source :** Loads the 8087 control register, status register, tag word, and exception pointers from a named area in memory.

**FINCSTP :** Increment the 8087 stack pointer by one.

**FDECSTP :** Decrement stack pointer by one.

**FFREE destination :** Changes the tag for the specified destination register to empty.

**FNOP :** Performs no operation. Actually copies ST to ST.

**FWAIT :** This instruction is actually an 8086 instruction which makes the 8086 wait until it receives a not busy signal from the 8087 to its $\overline{TEST}$ pin.

### 11.4.9 Programming using 8087 Coprocessor

**Program 33 :** It shows how the assembler automatically adjusts the FLD, FILD, and FBLD instructions for different size of operands.

```
.MODEL SMALL
.DATA
    DATA1    DD    30.0     ; Single- precision
    DATA2    DQ    30.0     ; Double-precision
    DATA3    DT    30.0     ; Extended-precision
    DATA4    DW    30       ; 16-bit integer
    DATA5    DD    30       ; 32-bit integer
    DATA6    DQ    30       ; 64-bit integer
    DATA7    DT    30H      ; BCD 30
.CODE
START:   MOV  AX,@DATA          ; [ Initialise
         MOV  DS,AX             ;     data segment]
         FINIT                  ; Initialise 8087
         FLD  DATA1
         FLD  DATA2
         FLD  DATA3
         FILD      DATA4
         FILD      DATA5
         FILD      DATA6
         FBLD      DATA7
         END  START
```

**Program 34 :** It calculates the Area of the circle. Area $A = \pi R^2$ where R is the radius of the circle.

```
; Program calculates the area of a circle.
; The radius must be stored at memory location RADIUS.
; The result is found  in memory location AREA after the program execution.
.MODEL SMALL
.DATA
         RADIUS    DD   3.4
         AREA      DD   ?
.CODE
START:   MOV  AX,@DATA ; [ Initialise
         MOV  DS,AX         ;    data segment]
         FINIT              ; Initialise 8087
         FLD  RADIUS        ; Loads radius (R) in ST
         FMUL      ST,ST(0) ; Square radius (R× R)
```

```
        FLDPI              ; π to ST
        FMUL               ; Multiply ST=ST × ST(1)i.e. (π × R× R)
        FSTP     AREA      ; Save area
        END  START
```

**Program 35 :** Procedure to find the roots of the quadratic equations. The roots of quadratic equations $ax^2 + bx + c = 0$ are, $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. The procedure finds the roots R1 and R2 for constants stored in memory locations A, B, and C. The .286 and .287 directives identify the processor as an 80286 and the coprocessor as an 80287.

```
    .286
    .287
.MODEL SMALL
.DATA
      TWO    DD    2.0
      FOUR   DD    4.0
      A      DD    1.0
      B      DD    - 16.0
      C      DD    + 39.0
      R1     DD    ?
      R2     DD    ?
.CODE
START: MOV AX,@DATA            ; [Initialise
       MOV DS,AX               ;  data segment]
       FINIT                   ; Initialise 8087
       FLD        TWO
       FMUL       A            ; form 2a
       FLD        FOUR
       FMUL       A            ; form 4a
       FMUL       C            ; form 4ac
       FLD        B
       FMUL       B            ; form b^2
       FSUBR                   ; form b^2 - 4ac
       FSQRT                   ; form square root of b^2-4ac
       FLD        B
       FSUB       ST, ST(1)
       FDIV       ST, ST(2)
       FSTP       R1           ; save root 1
       FLD        B
       FADD
       FDIVR
       FSTP       R2           ; save root 2
       END        START
       END
```

**Program 36 :** It determines the resonant frequency of an LC circuit. Equation of resonant frequency $f_r = \frac{1}{2\pi\sqrt{LC}}$

```
.MODEL SMALL
.DATA

      RESOF  DD   ?            ; resonant frequency
      L      DD   .000001      ; inductance
      C      DD   .000002      ; capacitance
      TWO         2.0          ; constant

.CODE
START: MOV AX,@DATA            ; [Initialise
       MOV DS,AX               ;    data segment]
       FINIT                   ; Initialise 8087
       FLD         L           ; Get L
       FMUL        C           ; Find LC
       FSQRT                   ; Find √LC
       FMUL        TWO         ; Find 2√LC
       FLDPI                   ; Get π
       FMUL                    ; Get 2π√LC
       FLD1                    ; Get 1
       FDIVR                   ; Form 1/(2π √LC)
       FSTP        RESOF       ; Save frequency
       END         START
       END
```

**Program 37 :** Calculates hypotenuse of a right angle triangle, whose sides A and B are stored in memory named SIDEA and SIDEB

```
   .287     ; tells assembler that coprocessor is 80287
   .286     ; tells assembler that processor is 80286
   .MODEL SMALL
   .DATA
      SIDEA       DD 3.0
      SIDEB       DD 4.0
      HYPOTENUSE  DD 0
      CW          DW 0         ; space for control word
      SW          DW 0         ; space for status word
   .CODE
  START: MOV AX,@DATA          ; [Initialise
         MOV DS,AX             ;  data segment ]
         FINIT                 ; Initialise 8087
         MOV CW, 03FFH         ; Put the CW in memory which
                               ; masks the interrupts and
                               ; sets round to even
         FLDCW CW              ; Load control word in 8087
         FLD SIDEA             ; Put value of SIDEA on top of stack
         FMUL ST, ST(0)        ; Square SIDEA
```

```
        FLD SIDEB           ; Put value of SIDEB on top of stack
        FMUL ST, ST(0)      ; Square SIDEB
        FADD ST, ST(0)      ; A^2 + B^2 result at the top of stack.
        FSQRT               ; √(A^2 + B^2) = hypotenuse in top of
stack
        FSTSW SW            ; Copy status word to memory
                            ; so 8086 can access it
        MOV AX, SW          ; Bring status to AX to check
                            ; for errors
        AND AL, 0BFH        ; Mask unneeded bits
        JNZ LAST            ; handle error if found
        FSTP HYPOTENUSE     ; No error, copy result from 8087 to
                            ; memory.
LAST:   NOP
        END START
```

## Review Questions

1. *Define multiprocessor systems.*
2. *List the advantages of multiprocessor systems.*
3. *Write short notes on*
   a. *Closely coupled multiprocessor configuration.*
   b. *Loosely coupled multiprocessor configuration.*
4. *List the advantages of loosely coupled systems.*
5. *Explain the need for 8087.*
6. *Explain the features of 8087.*
7. *Explain the functions of following pins of 8087.*
   *a) BUSY b) $\overline{RQ}/\overline{GT}$ c) INT*
8. *Draw and explain the circuit connection between 8087 and 8086.*
9. *Explain the interaction between 8086 and 8087.*
10. *Draw and explain the block diagram of 8087.*
11. *Draw the bit pattern of the status register of 8087 and explain the significance of each bit.*
12. *Draw the bit pattern of the control register of 8087 and explain the significance of each bit.*
13. *Give the data formats used by 8087.*
14. *Write a short on stack of 8087.*
15. *Solve the following :*
    *a) Convert $1632.125_{10}$ in short real, long real and temporary real format*
    *b) Convert $-4312.125_{10}$ in short real, long real and temporary real format.*

❑❑❑

# 12 Bus Interface

## 12.1 Introduction

There are many alternatives for the design of bus of a computer. One I/O device interfaced with a particular interface circuitry for one computer may not be suitable with other computers. Therefore, it may be required to design separate interface for every combination of I/O device and computer, resulting many different interfaces.

In personal computers processor is mounted on the motherboard. The devices which require high speed connection to the processor, such as the main memory, may be connected directly to the processor bus. The motherboard usually provides another bus that can support more devices. The two buses are connected by a circuit called **bridge**. The bridge translates the signals and protocols of one bus into those of the other. The devices connected to the expansion bus appear to be connected to the processor bus directly. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and these devices.

A number of standards have been developed for expansion bus. In this section we discuss most widely used bus standards such as PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is shown in Fig. 12.1. The PCI standard is used for expansion bus on the motherboard. The SCSI bus is a high speed parallel bus intended for devices which need the large amount of data transfer. USB bus is used for serial transmission to fulfil the needs of devices like keyboard, mouse etc. The ISA interface is used to interface IDE (Integrated Device Electronics) disk. The Ethernet interface is used for local area network, providing high speed connection among computer in the same primises.

## 12.2 The Peripheral Component Interconnect (PCI) Bus

In early 1992, Intel formed an another industrial group in relation to be PC bus. The main intension behind the formation of group was to overcome the weaknesses in the ISA and EISA buses. They designed PCI (Peripheral Component Interface) specifications in June 1992, and updated in April 1993.

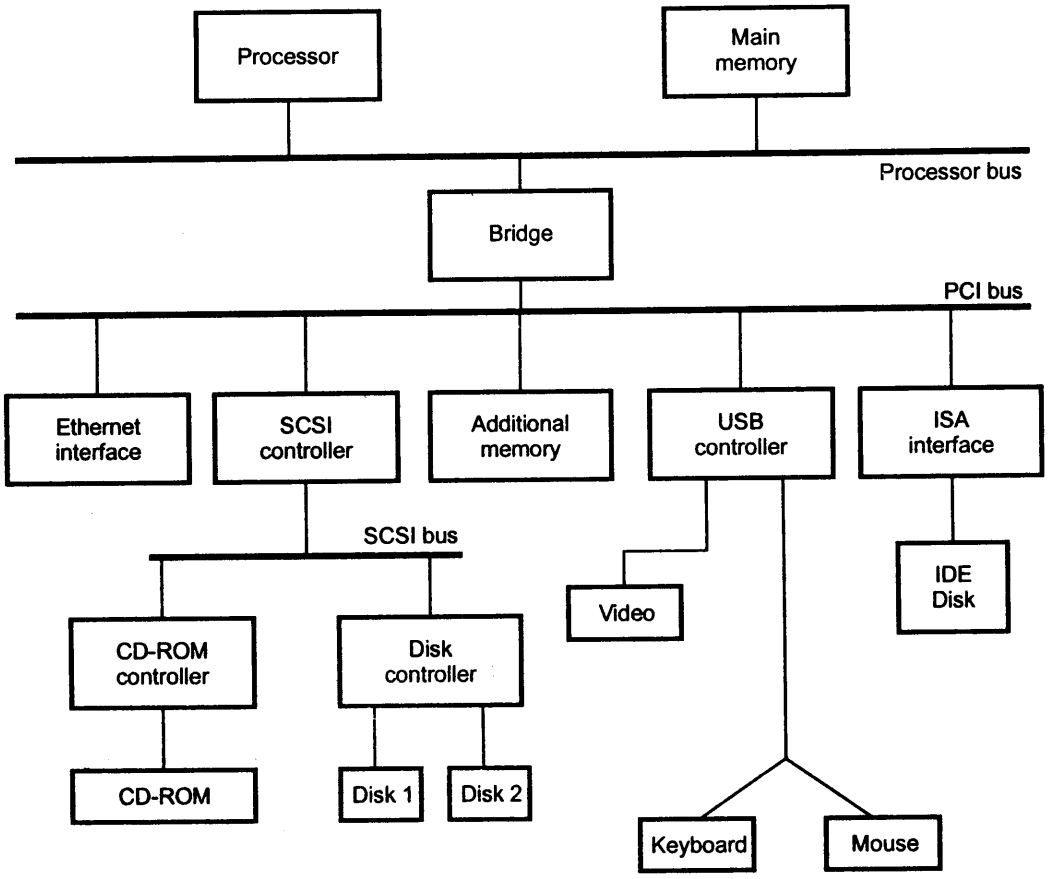



**Fig. 12.1 Bus standards used in typical computer system**

## 12.2.1 Features

1. It is designed to economically meet the I/O requirements of modern systems. It requires very few chips to implement and support other buses attached to the PCI bus.
2. It bypasses the standard I/O bus, uses the system bus to increase the bus clock speed and take full advantage of the CPU's data path.
3. It has an ability to function with a 64-bit data bus.
4. It has high bandwidth. The information is transferred across the PCI bus at 33 MHz, at the full data width of the CPU. When the bus is used in conjunction with a 32-bit CPU, the bandwidth is 132 Mbytes/sec. It is calculated as follows :

   33 MHz × 32-bit = 1,056 Mbits/sec

   1,056 Mbits/sec ÷ 8 = 132 Mbytes/sec

5. PCI bus is designed to support a variety of microprocessor based configurations including both single and multiprocessor systems.

6. The PCI bus can operate concurrently with the processor bus. The CPU can be processing data in a external cache while the PCI bus is busy transferring information between other parts of the system.
7. The PCI bus is processor - independent bus that can function as a mezzanine or peripheral bus.
8. It makes use of synchronous timings and centralized arbitration scheme.
9. It delivers better system performance for high - speed I/O subsystems (e.g. graphic display adapters, network interface controllers, disk controllers and so on).
10. The PCI interface contains a 256 bytes configuration memory which allows the computer to interrogate the PCI interface. This feature allows the system to automatically configure itself for the PCI plug-board and hence it is referred to as plug-and-play.

## 12.2.2 PCI Configurations

As mentioned earlier, PCI bus is designed to support single processor as well as multiprocessor systems. Fig. 12.2 (a) shows a typical use of PCI in a single processor system.

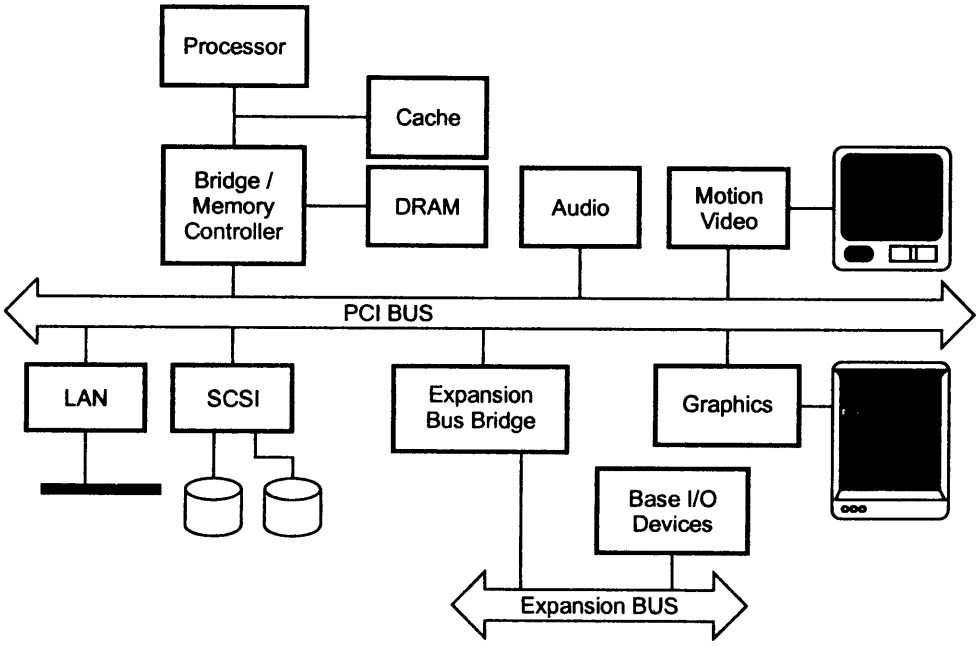



**Fig. 12.2 (a) Conceptual diagram of PCI bus for single processor system**

Notice that the processor bus is separate and independent of the PCI bus. The processor connects to the PCI bus through an integrated circuit called a PCI bridge. The memory controller and PCI bridge provides tight coupling with the processor and delivers data at high speeds.

Fig. 12.2 (b) shows a typical use of PCI in a multiprocessor system. As shown in the Fig. 12.2(b) in multiprocessor systems one or more PCI configurations may be connected by bridges to the processor's system bus. Again, the use of bridges keeps the PCI independent of the processor speed yet provides the ability to receive and deliver data rapidly.

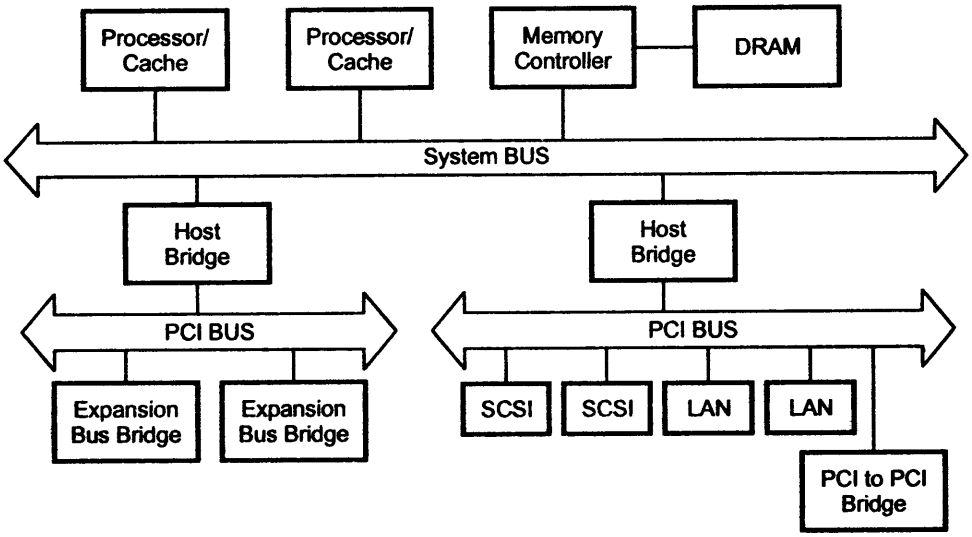



**Fig. 12.2 (b) Conceptual diagram of PCI bus for multiprocessor system**

## 12.2.3 PCI Bus Signals

Fig. 12.3 (See on next page) shows pinouts for 5 V PCI bus. The PCI bus may be configured as a 32 or 64 - bit bus. Fig. 12.4 shows 32-bit and 64-bit connectors for PCI bus.

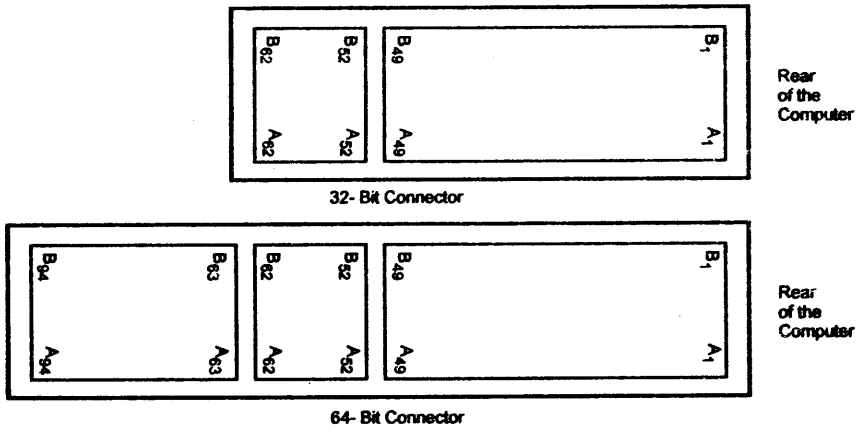



**Fig. 12.4 The 5 V PCI slot and card configuration**

Back of computer

| Pin # | SOLDER SIDE | COMPONENT SIDE |
|---|---|---|
| 1 | −12 V | TRST |
| 2 | TCK | +12 V |
| 3 | GND | TM5 |
| 4 | TD0 | TD1 |
| 5 | +5 V | +5 V |
| 6 | +5 V | INTA |
| 7 | INTB | INTC |
| 8 | INTD | +5 V |
| 9 | PRSNT 1 | |
| 10 | | +VI/O |
| 11 | PRSNT 2 | |
| 12 | KEY | KEY |
| 13 | KEY | KEY |
| 14 | | |
| 15 | GND | RST |
| 16 | CLK | VI/O |
| 17 | GND | VNT |
| 18 | REQ | GND |
| 19 | +V IO | |
| 20 | AD31 | AD30 |
| 21 | AD29 | + 3.3 V |
| 22 | GND | AD28 |
| 23 | AD27 | AD26 |
| 24 | AD25 | GND |
| 25 | + 3.3 V | AD24 |
| 26 | D/BE 3 | IDSEL |
| 27 | AD23 | + 3.3 V |
| 28 | GND | AD22 |
| 29 | AD21 | AD20 |
| 30 | AD19 | GND |
| 31 | + 3.3 V | AD18 |
| 32 | AD17 | AD16 |
| 33 | C/BE 2 | + 3.3 V |
| 34 | GND | FRAME |
| 35 | IRDY | GND |
| 36 | + 3.3 V | TRDY |
| 37 | DEVSEL | GND |
| 38 | GND | STOP |
| 39 | LOCK | + 3.3 V |
| 40 | PERR | SDONE |

| Pin # | SOLDER SIDE | COMPONENT SIDE |
|---|---|---|
| 41 | +3.3 V | SBO |
| 42 | SERR | GND |
| 43 | +3.3 V | PAR |
| 44 | C/BE1 | AD15 |
| 45 | AD14 | +3.3 V |
| 46 | GND | AD13 |
| 47 | AD12 | AD11 |
| 48 | AD10 | GND |
| 49 | GND | AD9 |
| 50 | KEY | KEY |
| 51 | KEY | KEY |
| 52 | AD8 | C/BE0 |
| 53 | AD7 | +3.3 V |
| 54 | +3.3 V | AD6 |
| 55 | AD5 | AD4 |
| 56 | AD3 | GND |
| 57 | GND | AD2 |
| 58 | AD1 | AD0 |
| 59 | +V IO | +V IO |
| 60 | ACK64F | REQ64 |
| 61 | +5 V | +5 V |
| 62 | +5 V | +5 V |
| 63 | | GND |
| 64 | GND | C/BE 7 |
| 65 | C/BE 6 | C/BE 5 |
| 66 | C/BE 4 | +V IO |
| 67 | GND | PAR64 |
| 68 | AD63 | AD 62 |
| 69 | AD 61 | GND |
| 70 | +V IO | AD60 |
| 71 | AD59 | AD58 |
| 72 | AD57 | GND |
| 73 | GND | AD56 |
| 74 | AD55 | AD54 |
| 75 | AD53 | +V IO |
| 76 | GND | AD52 |
| 77 | AD51 | AD50 |
| 78 | AD49 | GND |
| 79 | +V IO | AD48 |
| 80 | AD47 | AD46 |
| 81 | AD45 | GND |
| 82 | GND | AD44 |
| 83 | AD43 | AD42 |
| 84 | AD41 | +V I/O |
| 85 | GND | AD40 |
| 86 | AD39 | AD38 |
| 87 | AD37 | GND |
| 88 | +V I/O | AD36 |
| 89 | AD35 | AD34 |
| 90 | AD33 | GND |
| 91 | GND | AD32 |
| 92 | | |
| 93 | | GND |
| 94 | GND | |

Note : (1) Pins 63-94 exist only on the 64-bit PCI card.
(2) +V I/O is 3.3 V on a 3.3 V board and +5 V on a 5 V board.
(3) Blank pins are reserved.

**Fig. 12.3 The pin-out of the PCI bus**

**PCI signals are functionally divided into the following groups :**

- **System Signals :** System signals include clock and reset signals (Test clock, clock, Test Reset, Reset).
- **Address and Data Signals :** Address and data signals include 64 lines that are time - multiplexed for addresses and data lines (AD0 - AD63). The other signals from this group such as parity, command and byte enable signals are used to interpret and validate the signal lines that carry the addresses and data (PAR, C/BE).

**Interface Control Signals :** Interface control signals control the timing of transactions and provide co-ordination among initiators and targets. These are as follows :

$\overline{\text{FRAME}}$ : The current master uses this signal to indicate the start and duration of a transaction. The signal is asserted at the start and deasserted when the initiator is ready to begin the final data phase.

$\overline{\text{IRDY}}$ : The current master (initiator) uses this signal to indicate that it is ready to read or write valid data.

$\overline{\text{TRDY}}$ : The target device (selected device) uses this signal to indicate that it is ready to read or write valid data.

$\overline{\text{STOP}}$ : This signal indicates that the current target wishes the initiator to stop the current transaction.

$\overline{\text{LOCK}}$ : The signal indicates an automatic operation that may require multiple transactions.

**IDSEL** : This initialization device select signal is used as a chip select during configuration read and write transactions.

$\overline{\text{DEVSEL}}$ : This device select signal is used by the target to indicate device has selected.

- **Arbitration Signals :** Unlike other PCI signals, these are not shared lines. Rather, each PCI master has its own Request and Grant arbitration signals that connect it directly to the PCI bus arbiter.
- **Error Reporting Signals :** These are used to report parity and system errors ($\overline{\text{PERR}}$, $\overline{\text{SERR}}$).
- **Interrupt Signals :** Like bus arbitration signals these are not shared signals. Rather, each PCI device has its own interrupt signals ($\overline{\text{INTA}}$, $\overline{\text{INTB}}$, $\overline{\text{INTC}}$, $\overline{\text{INTD}}$).

- **Cache Support Signals :** These signals support snoopy cache protocol ($\overline{SBO}$, SDONE).
- **JTAG/Boundary Scan Signals :** These signals support testing procedures defined in IEEE standard 149.1 (TCK - Test clock, TDI - Test Input, TDO - Test Output, TMS - Test Mode Select, TRST - Test Reset).

### 12.2.4 PCI Bus Commands

The PCI bus commands are used to carryout transactions between an initiator, or master, and a target. These commands are :

- **INTA Sequence :** INTA sequence is a read command intended for the interrupt controller on the PCI bus. The byte - sized interrupt vector is returned during a byte read operation.
- **Special Cycle :** This command is used by the initiator to send a message to one or more targets.
- **I/O Read Cycle :** I/O read cycle command is used to transfer data from an I/O device to the initiator.
- **I/O Write Cycle :** I/O write cycle command is used to transfer data from an initiator to the I/O device.
- **Memory Read Cycle :** Memory read cycle command is used to transfer data from an memory device on the PCI bus to the initiator
- **Memory Write Cycle :** Memory write cycle command is used to transfer data from an initiator to the memory device on the PCI bus.
- **Configuration Read :** This command is used to read configuration information from the PCI device.
- **Configuration Write :** This command is used to write configuration information in the PCI device.
- **Memory Multiple Access :** This is similar to the memory read cycle command, except that it is usually used to access many data instead of one.
- **Dual Addressing Cycle :** This command is used for transferring address information to a 64 - bit PCI device, which only contains a 32 - bit data path.
- **Line Memory Access :** This command is used to read more than two 32 - bit numbers from the PCI bus.
- **Memory Write with Invalidation :** This command is used to transfer data in one or more cycles to memory.

### 12.2.5 Data Transfer

Every data transfer on the PCI bus is a single transaction consisting of one address phase and one or more data phases. Let us see the typical read cycle on the PCI bus. All the events during read cycle are synchronized with the falling edge of the clock cycle. The devices connected to the bus sample the bus lines on the rising edge at the beginning of a bus cycle. The events during the bus cycle are explained below and they are also labeled on the diagram.

1. The master asserts $\overline{\text{FRAME}}$ signal to indicate the start of a transaction. This signal remains asserted until initiator is ready to begin the data phase. The initiator puts the start address on the address bus, and initiates read cycle by activating C/$\overline{\text{BE}}$ lines.
2. At the beginning of clock2, the target device recognizes its address on the address bus.
3. The initiator ceases the address lines and it changes the information on the C/$\overline{\text{BE}}$ lines to designate which address lines are to be used for transfer for the currently addressed data from 1 to 4 bytes. The initiator also asserts IRDY to indicate that it is ready for the first data byte.
4. The selected target asserts $\overline{\text{DEVSEL}}$ to indicate that it has recognized its address and it responds by placing the requested data on the address bus. It then asserts $\overline{\text{TRDY}}$ to indicate that valid data is present on the bus.
5. The initiator reads the data at the beginning of clock 4 and changes the byte enable lines as needed in preparation for the next read. If target needs some more time to put next byte of data on the address lines, it informs this by deasserting $\overline{\text{TRDY}}$ line. Accordingly, the initiator does not read the data lines at the beginning of the next clock cycle and does not change byte enable during that cycle. This is illustrated in the fifth clock cycle.
6. It may happen that, the target places the data byte but the initiator is not ready to accept that data byte. The initiator indicates this by deasserting $\overline{\text{IRDY}}$ signal.
7. The initiator deasserts $\overline{\text{FRAME}}$ signal indicating that it is last data byte transfer.
8. The initiator deasserts $\overline{\text{IRDY}}$ and the target deasserts $\overline{\text{TRDY}}$ and $\overline{\text{DEVSEL}}$, returning bus to the idle state.

Similar kind of handshaking signals are activated during PCI write operation. Here, the data from initiator is copied into the targeted devices.

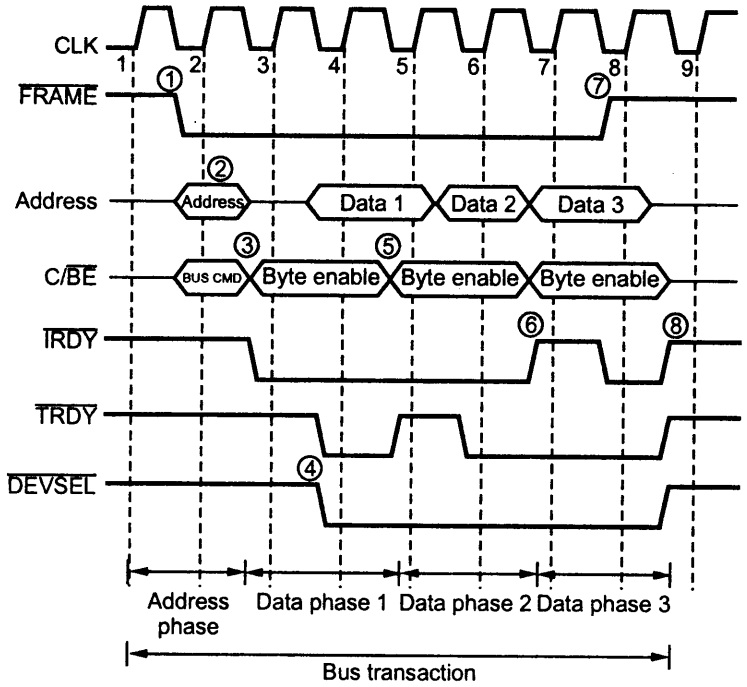



**Fig. 12.5 Timing diagram for PCI read operation**

## 12.2.6 PCI Arbitration

As mentioned earlier PCI bus uses centralized, synchronous arbitration in which each master has a unique request (REQ) and grant (GNT) signals. These signals are connected to the central bus arbiter, as shown in the Fig. 12.6.

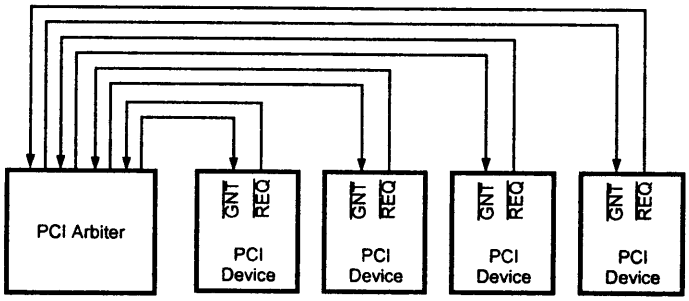



**Fig. 12.6 PCI bus arbiter**

The PCI bus does not specify a particular arbitration algorithm. Therefore, it allows to use a first in first serve approach, a round-robin approach or priority approach. Let us see the operation of PCI arbiter with two bus masters in the system. Refer Fig. 12.7.

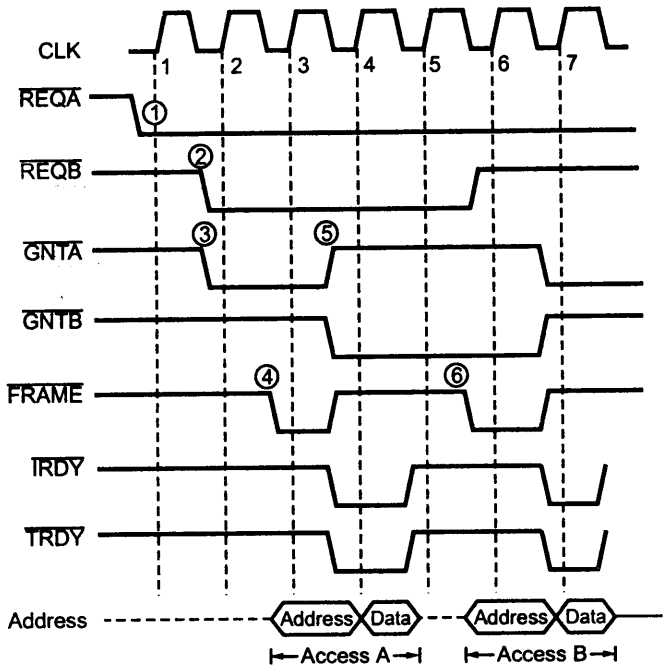



**Fig. 12.7 PCI bus arbitration between two masters**

1. Let us assume that Master A has asserted its $\overline{\text{REQ}}$ ($\overline{\text{REQA}}$) signal before the start of clock 1. The arbiter samples that signal at the beginning of clock 1.
2. The Master B request for bus by asserting its $\overline{\text{REQ}}$ ($\overline{\text{REQB}}$) signal during clock 1.
3. At the same time, the arbiter asserts $\overline{\text{GNTA}}$ signal to grant bus access to Master A.
4. The Master A samples $\overline{\text{GNTA}}$ signal at the beginning of clock 2 and recognizes that it has been granted bus access. At this time $\overline{\text{IRDY}}$ and $\overline{\text{TRDY}}$ are deasserted, indicating that the bus is idle. Therefore, Master A asserts $\overline{\text{FRAME}}$ and places the address information on the address bus and the command on the C/$\overline{\text{BE}}$ bus (it is not shown in the Fig. 12.5). The MASTER A continues to assert $\overline{\text{REQA}}$ signal if it has to perform another transaction as shown in the timing diagram.
5. At the beginning of clock 3, the bus arbiter samples all $\overline{\text{GNT}}$ lines and makes an arbitration decision according to arbitration algorithm used. In this timing diagram it grants the bus for master B for the next transaction. It then deasserts $\overline{\text{GNTA}}$, $\overline{\text{IRDY}}$, $\overline{\text{TRDY}}$ and $\overline{\text{FRAME}}$ signals and asserts $\overline{\text{GNTB}}$. However, Master B will not be able to use the bus until it returns to an idle state.
6. At the beginning of clock 5, Master B finds $\overline{\text{IRDY}}$ and $\overline{\text{FRAME}}$ deasserted and so it is allowed to take control of the bus by asserting $\overline{\text{FRAME}}$. It also deasserts the $\overline{\text{REQB}}$ line because Master B has to perform only one transaction.

### 12.2.7 Configuration Space

As mentioned earlier the PCI interface contains a 256 byte configuration memory that allows the computer to interrogate the PCI interface. This allows the system to automatically configure itself for the PCI plug board.

The Fig. 12.8 shows the map of 256 byte configuration memory. As shown in the Fig. 12.8 the first 64 bytes of the configuration memory contains the header that holds information about the PCI interface. The first 32-bit double word contains the unit ID code and the vendor ID code. When unit is not installed, the unit ID code is FFFFH. It is a 16-bit number $(D_{31} - D_{16})$. When unit is installed, the unit ID code can be between 0000H and FFFEH that identifies the unit. The vendor ID $(D_{15} - D_0)$ is allocated by PCI SIG, which is the governing body for the PCI bus interface standard.

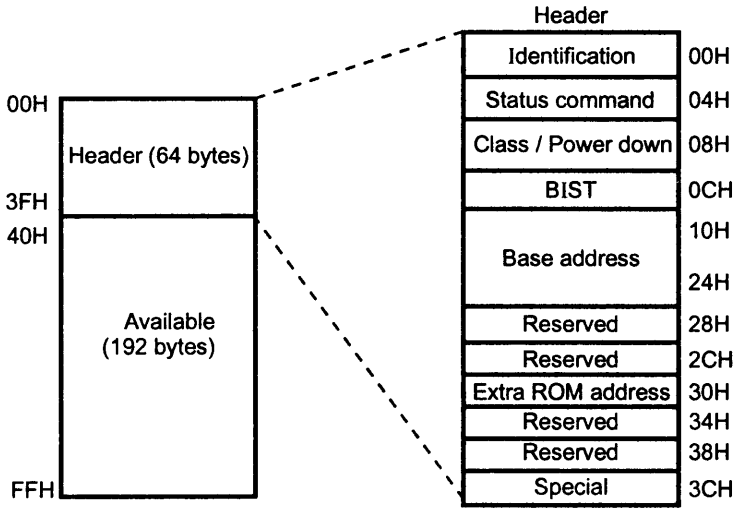



**Fig. 12.8 The map of configuration memory on a PCI expansion board**

The class of the PCI interface is identified by class code. The Table 12.1 shows the current class codes which are assigned by the PCI SIG. The class code is stored in bits $D_{31} - D_{16}$ of the configuration memory at location 08H.

| Class code | Function |
|---|---|
| 0000H | Older non-VGA devices (not plug and play) |
| 0001H | Older VGA devices (not plug and play) |
| 0100H | SCSI controller |
| 0101H | IDE controller |
| 0102H | Floopy disk controller |
| 0103H | IPI controller |
| 0180H | Other hard / floppy controller |
| 0200H | Ethernet controller |
| 0201H | Token ring controller |
| 0202H | FDDI |
| 0280H | Other network controller |
| 0300H | VGA controller |
| 0301H | XGA controller |
| 0380H | Other video controller |
| 0400H | Video multimedia |
| 0401H | Audio multimedia |
| 0480H | Other multimedia controller |
| 0500H | RAM controller |
| 0501H | Flash memory controller |
| 0580H | Other memory controller |
| 0600H | Host bridge |
| 0601H | ISA bridge |
| 0602H | EISA bridge |
| 0603H | MCA bridge |
| 0604H | PCI-PCI bridge |
| 0605H | PCMIA bridge |
| 0680H | Other bridge |
| 0700H-FFFEH | Reserved |
| FFFFH | Unit not in any of the above classes |

**Table 12.1 : Class codes assigned by PCI SIG**

The formats for status word and command words are shown in the Fig. 12.9. These words are loaded in the location 04 of the configuration memory such that status word occupies bits $D_{31}$-$D_{16}$ and command word occupies bits $D_{15}$-$D_0$.

The base address space consists of base addresses for memory, I/O and expansion ROM. The first two double words of the base address space contain either the 32 or 64-bit base address for the memory present on the PCI interface. The next double word contains the base address of the I/O space. It is important to note that, even though the Intel microprocessor uses 16-bit I/O address, there is a provision for expanding the I/O address to 32-bits.

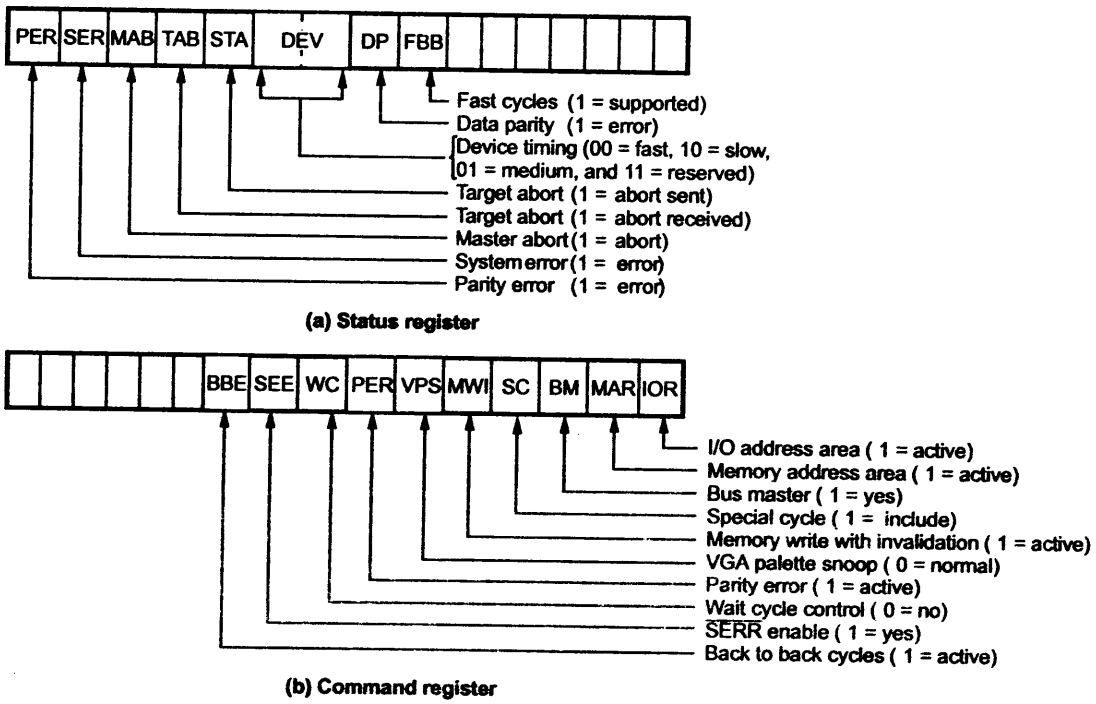



**Fig. 12.9 Bit pattern of the status and control words in the configuration memory**

### BIOS for PCI

In modern computer system BIOS supports the PCI bus utilities using interrupt vector 1AH with AH = B1H. This utilities are summaried as follows :

---

**Function 01H : BIOS Available ?**

---

```
Call with :          AH = 0B1H
                     AL = 01H
Returns :            AH = 00H if PCI BIOS extension
                          is available
```

BX = Version number
EDX = ASCII string PCI
Carry = 1 if no PCI extension present

## Function 02H : PCI Unit Search

Call with :
AH = 0B1H
AL = 02H
CX = Unit
DX = Manufacturer
SI = index

Returns :
AH = result code (see notes)
BX = bus and unit number
Carry = 1 for error

Notes :
The result codes are :
00H = successful search
81H = function not supported
83H = invalid manufacturerID code
86H = Unit not found
87H = invalid register number

## Function 03H : PCI Class Code Search

Call with :
AH = 0B1H
AL = 03H
ECX = class code
SI = index

Returns :
AH = result code (see notes for function 02H)
BX = bus and unit number
Carry = 1 for an error

## Function 06H : Start Special Cycle

Call with :
AH = 0B1H
AL = 06H
BX = bus and unit number
EDX = data

Returns :
AH = result code (see notes for function 02H)
Carry = 1 for error

Notes :
The value passed in EDX is sent to the PCI bus during the address phase

| Function 08H | Configuration Byte-Sized Read |
|---|---|
| Call with : | AH = 0B1H |
| | AL = 08H |
| | BX = bus and unit number |
| | DI = register number |
| Returns : | AH = result code (see notes for function 02H) |
| | CL = data from configuration register |
| | Carry = 1 for error |

| Function 09H | Configuration WOrd-Sized Read |
|---|---|
| Call with : | AH = 0B1H |
| | AL = 08H |
| | BX = bus and unit number |
| | DI = register number |
| Returns : | AH = result code (see notes for function 02H) |
| | CX = data from configuration register |
| | Carry = 1 for error |

| Function 0AH | Configuration Doubleword-Sized Read |
|---|---|
| Call with : | AH = 0B1H |
| | AL = 08H |
| | BX = bus and unit number |
| Returns : | DI = register number |
| | AH = result code (see notes for register 02H) |
| | ECX = data from configuration register |
| | Carry = 1 for error |

**Function 0BH Configuration Byte-Sized Write**

```
Call with :     AH = 0B1H
                AL = 08H
                BX = bus and unit number
                CL = data to be written to
                     configuration register
                DI = register number
Returns :       AH = result code (see notes for
                     function 02H)
              Carry = 1 for error
```

**Function 0CH Configuration Word-Sized Write**

```
Call with :     AH = 0B1H
                AL = 08H
                BX = bus and unit number
                CX = data to be written to
                     configuration register
                DI = register number
Returns :       AH = result code (see notes for
                     function 02H)
              Carry = 1 for error
```

**Function 0DH Configuration Doubleword-Sized Write**

```
Call with :     AH = 0B1H
                AL = 08H
                BX = bus and unit number
               ECX = data to be written to
                     configuration register
                DI = register number
Returns :       AH = result code (see notes for
                     function 02H)
              Carry = 1 for error
```

Let us see the program to determine whether the PCI bus extension BIOS is available.

**Program :**

```
.MODEL SMALL
.DATA
MES1    DB      'PCI BIOS IS NOT PRESENT$'
MES2    DB      'PCI BIOS IS PRESENT$'
.CODE   MOV     AX,@DATA ; [Initialize data
        MOV     DS,AX    ; segment]
        MOV     AH,0B1H  ; access PCI extension
        MOV     AL, 01H  ; load function number
        INT     1AH      ; call interrupt
        MOV     DX, OFFSET MES2
        .IF CARRY?
            MOV DX,OFFSET MES1
        .ENDIF
        MOV AH,9         ; DISPLAY STRING
        INT 21H
        MOV AH,4CH       ; [Exit to
        INT 21H          ;  DOS]
        END
```

If the BIOS is present, the contents of the configuration memory can be read using the BIOS functions. It is important to note that the BIOS functions does not support data transfers between the computer and the PCI interface, the drivers provided with the interface does the data transfer. These drivers control the flow of data between the microprocessor and the components found on the PCI interface.

**PCI interface**

The Fig. 12.10 shows the block diagram of PCI interface. The block diagram shows the basic components required for PCI interface. The registers, parity block, initiator, target and vendor ID EPROM are required components of any PCI interface.

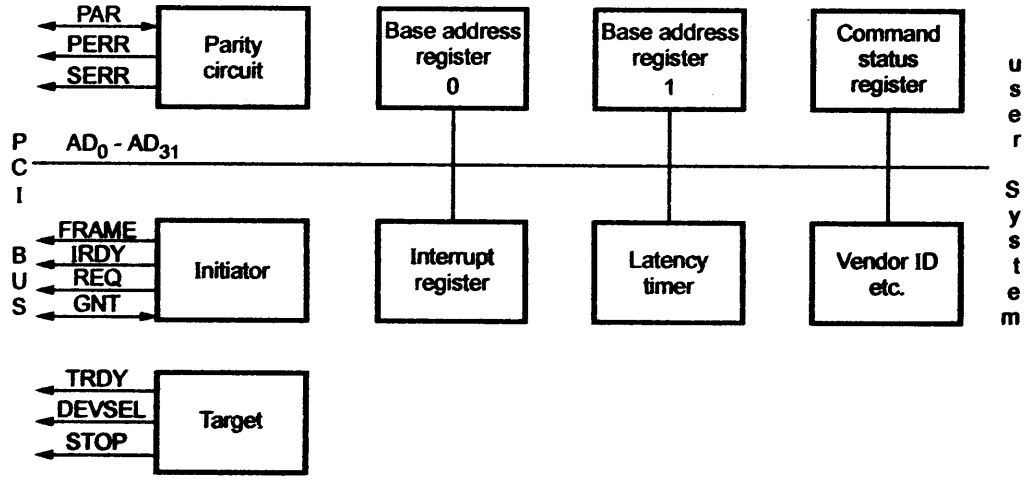



**Fig. 12.10 The block diagram of PCI interface**

## 12.3 Parallel Printer Interface(LPT)

The Parallel Printer Interface (LPT) is located on the rear panel of the personal computer. The LPT stands for line printer. This printer interface gives eight data lines to transfer data alongwith the control signal usually called hand shaking signals to control the flow of data. The printer interface can be programmed to receive or send data. This allows devices rather than printer, such as CD-ROMs, to be connected to and used by the PC through the parallel port. The **centronics protocol** is a printer protocol which specifies the standards for printer interface.

### Interface details

The computer can have two parallel printer interface, commonly known as **parallel ports** (LPT1 and LPT2). The LPT1 is normally at I/O port addresses 378H, 379H and 37AH. The LPT2 port, if present, is located at I/O port addresses 278H, 279H and 27AH. Let us see the details of both ports, but we use LPT1 port addresses. The Table 12.2 shows the pins and signals for parallel/centronics interface and the Fig. 12.11 shows the connectors used for parallel/centronics interface.

| Signal | Description | Pin no. for 25 pin connector | Pin no. for 36 pin connector |
|---|---|---|---|
| $\overline{STR}$ | Strobe to printer | 1 | 1 |
| D0 | Data bit 0 | 2 | 2 |
| D1 | Data bit 1 | 3 | 3 |
| D2 | Data bit 2 | 4 | 4 |
| D3 | Data bit 3 | 5 | 5 |
| D4 | Data bit 4 | 6 | 6 |
| D5 | Data bit 5 | 7 | 7 |
| D6 | Data bit 6 | 8 | 8 |
| D7 | Data bit 7 | 9 | 9 |
| $\overline{ACK}$ | Acknowledge from printer | 10 | 10 |
| BUSY | Busy from printer | 11 | 11 |
| PAPER | Out of paper | 12 | 12 |
| ONLINE | Printer is online | 13 | 13 |
| $\overline{ALF}$ | Low if printer issues a LF after a CR | 14 | 14 |
| $\overline{ERROR}$ | Printer error | 15 | 32 |
| $\overline{RESET}$ | Resets the printer | 16 | 31 |
| $\overline{SEL}$ | Selects the printer | 17 | 36 |
| + 5 V | 5 V from printer | – | 18 |
| Protective ground | Earth ground | – | 17 |
| Signal ground | Signal ground | All other pins | All other pins |

Note : Bar indicates an active low signal.

**Table 12.2 Parallel centronics port pins and signals**

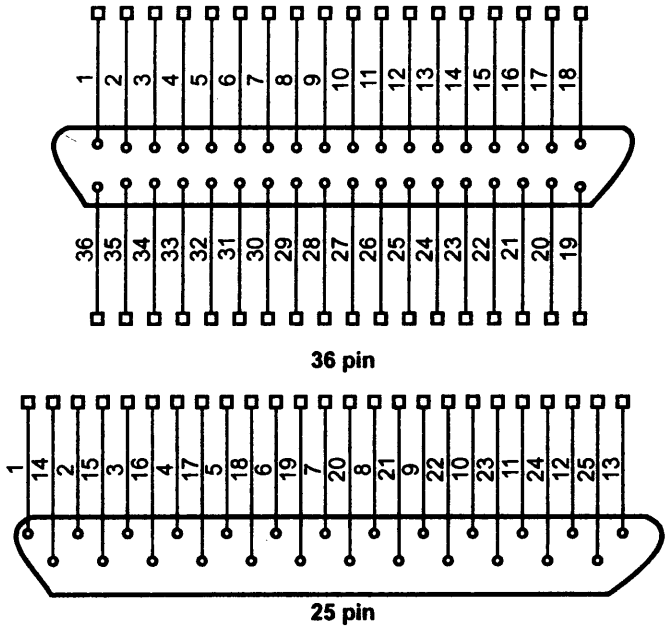



**Fig. 12.11 Connectors used for parallel/centronics port**

**Control/Handshaking Signals for Printer Interface**

**a) Input signals for printer :**

1. INIT : This signal when activated tells the printer to perform its internal initialization sequence.
2. STROBE ($\overline{STB}$) : This signal when activated tells the printer that valid data is available on the data bus.

**b) Status signals output from printer :**

1. $\overline{ACK}$ : This signal when low indicates that the data character has been accepted and the printer is ready for the next data.
2. BUSY : This is active high signal. It goes high when printer is not ready to receive a character.
3. PE : This active high signal goes high when printer is out of paper.
4. SLCT : This signal goes high if the printer is selected for receiving data.
5. ERROR : This active low signal goes low for variety of problem conditions in the printer.

Fig. 12.12 shows the timing waveforms for transfer of data characters to an IBM printer using the basic handshake signals.

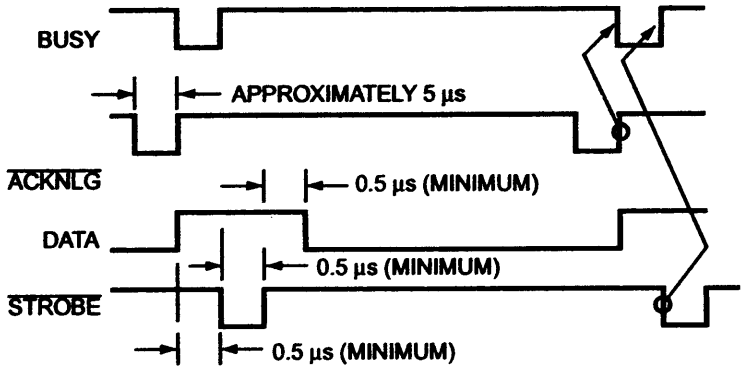



**Fig. 12.12 Timing waveforms for transfer of data characters to an IBM printer**

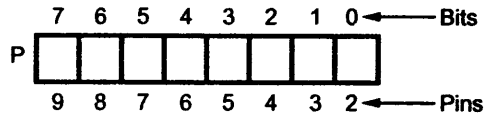
Computer sends the INIT pulse for at least 50 µs, to initialize the printer. Computer then checks for BUSY low to confirm whether the printer is ready to receive data or not. If BUSY signal is low (not busy), computer sends an ASCII code on eight parallel data lines and after atleast 0.5 µs, it also sends $\overline{STB}$ signal to indicate, valid data is available on the data bus. Computer activates this $\overline{STB}$ signal for at least 0.5 µs and it also ensures that valid data is present on the data bus for at least 0.5 µs after the $\overline{STB}$ signal is disabled. When the printer is ready to receive the next character, it asserts its $\overline{ACK}$ signal low for about 5 µs. The rising edge of the $\overline{ACK}$ signal tells the computer that it can send the next character. The rising edge of the $\overline{ACK}$ signal also resets the BUSY signal from the printer. When computer finds busy low, it sends the next character along with strobe and the sequence is repeated till the last character transfer.

The system program written to carry out data transfer through parallel port uses data port (378H), the status register (379H) and an additional status port (37AH). The bits patterns for these ports are as shown in Fig. 12.13. Refer Fig. 12.13 on next page.

### 12.3.1 Accepting 16-Bit Input using Parallel Port

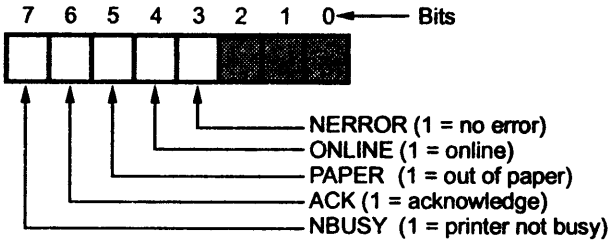
The Fig. 12.14 shows the circuit to accept 16-bit input through unidirectional parallel port. It transfers 4-bit information at a time through the adapter pins 11, 10, 12 and 13 to $S_7$-$S_4$ bits of status register. It is important to note that the signal level on pin 11 is inverted and stored in $S_7$ bit position.

The data port that connects to bits $D_0$ - $D_7$ (pins 2 - 9)

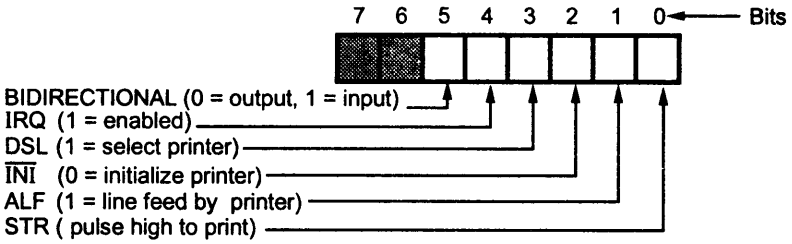



**(a) Data port (378H)**

This is a read-only port that returns the information from the printer through signals such as BUSY, $\overline{\text{ERROR}}$, and so forth.




**(b) Status register (379H)**




**(c) Additional status port (37AH)**

**Fig. 12.13**

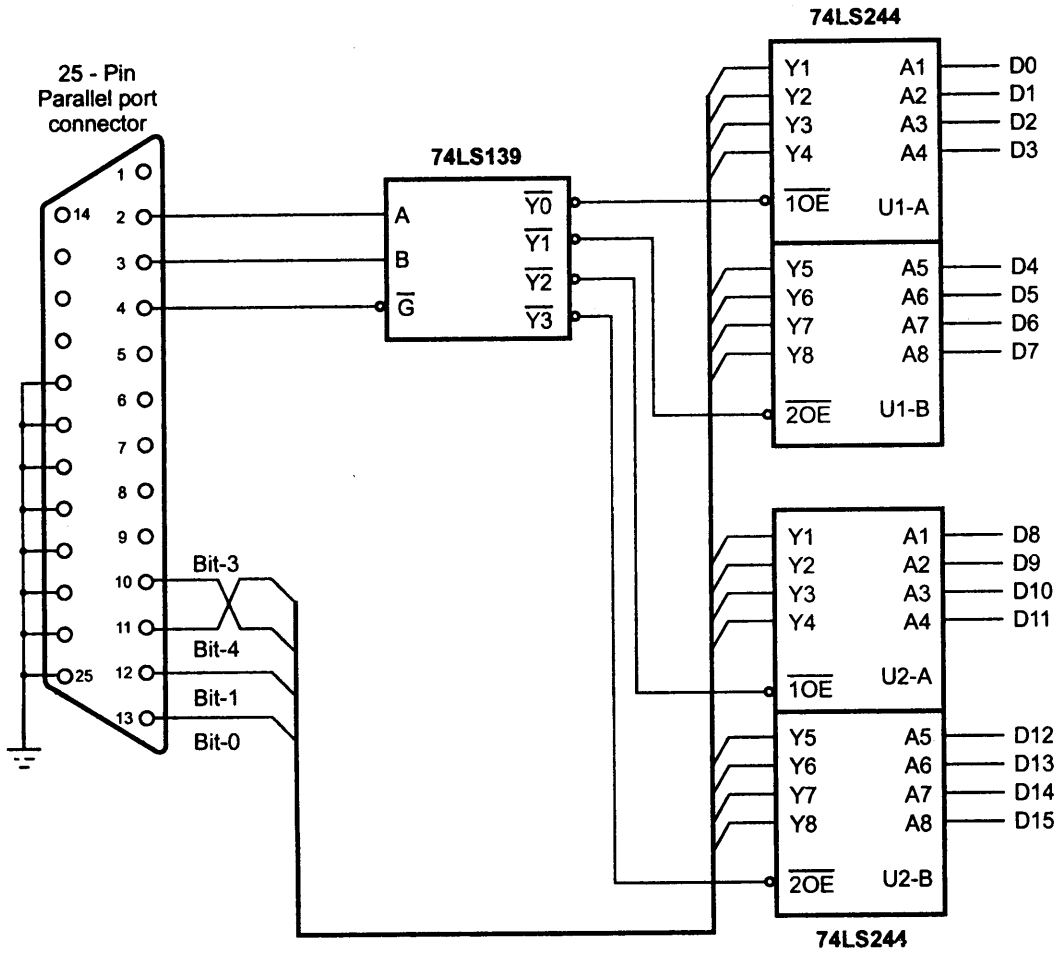



**Fig. 12.14 16-bit input interface for the unidirectional parallel port**

**Program : To read 16-bit data from parallel port**

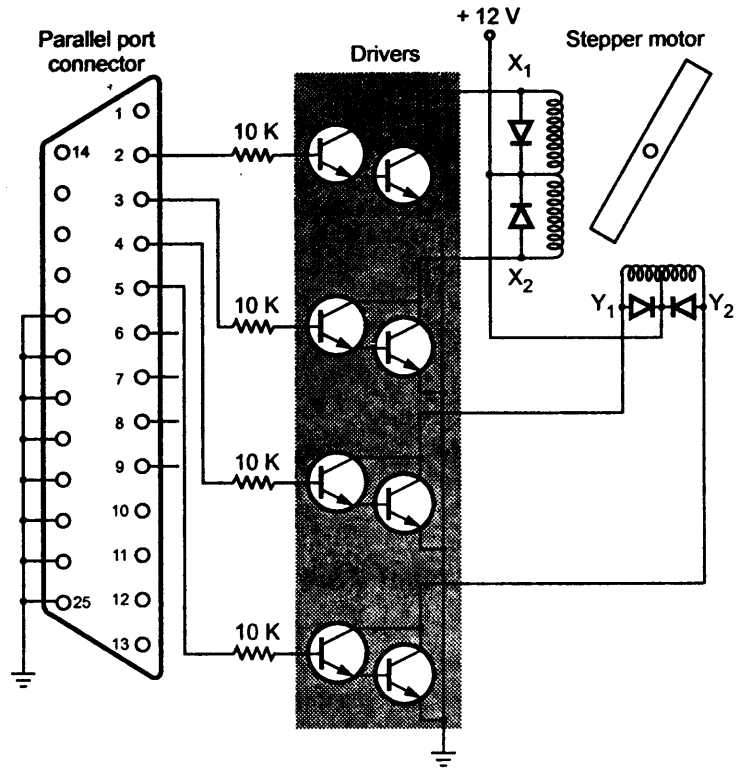
```
            MOV DX, 378H ; Load port address of data register
BACK :      MOV AL, CH   ; Load the control word to enable 4-bit
                         ; section
            OUT DX, AL
            MOV DX, 379H ; Load port address of status register
            IN AL, DX    ; Read 4-bits
            XRA AL, 80H  ; Invert the bit on S7 bit position
            MOV CL, 04H  ; Shift 16-bit data
            SAL BX, CL   ; 4-bit left
            MOV CL, 04H  ; [Adjust the bits to D3-D0 bit
            SHR AL, CL   ; position by shifting them right by
                           4-bits]
            OR BL, AL    ; Save partial data
            INC CH       ; Increment counter
            CMP CH, 04H  ; Check whether counter is 4
            JNZ BACK     ; if not read the next 4-bit of
                         ; data
```

## 12.3.2 Interfacing Stepper Motor through Parallel Port

The Fig. 12.15 shows the circuit to interface stepper motor using parallel port. Bit $D_0$ - $D_3$ are used to excite two ends of two windings. Each winding is center-tapped and the center-tap is connected to the 12 V supply. The excitation provided by $D_0$-$D_3$ lines is buffered using driver transistors. The transistors are selected such that they can source rated current for the winding. Motor is rotated by 1.8° per excitation.

**Program : To rotate stepper motor clockwise by 90°**

```
EX_CODE  DB 06H, 0AH, 09H, 05H ; Code sequence for clockwise
                               ; rotation
            MOV DX, 378H       ; Load port address
            MOV CX, 32H        ; Set repetation count to 5010
BACK :      MOV BL, 04H        ; Counts excitation sequence
            LEA SI, EX_CODE
REP1 :      MOV AL, [SI]       ; Get excitation code
            OUT DX, AL         ; Send excitation code
            CALL Delay         ; Wait for sometime
            INC SI             ; Point to next excitation code
            DEC BL             ; decrement counter
            JNZ REP1           ; if not zero goto REP1
            Loop BACK          ; if CX not zero repeat
```




| Excitation code | | | | In Hex |
|---|---|---|---|---|
| $X_1$ | $X_2$ | $Y_1$ | $Y_2$ | |
| 0 | 1 | 1 | 0 | 06H |
| 1 | 0 | 1 | 0 | 0AH |
| 1 | 0 | 0 | 1 | 09H |
| 0 | 1 | 0 | 1 | 05H |

**Fig. 12.15 Stepper motor interfaced through parallel port**

## 12.3.3 Bidirectional Operation of Parallel Port

The Fig. 12.16 shows the circuit required for bidirectional operation of parallel port. It consists of octal latch (IC74LS374), octal buffer (IC74LS244), and two 2-input OR gates. For data output, the output enable ($\overline{OE}$) pin of 74LS374 is enabled and it is disable for data input operation. Bit C5 of the control register controls the $\overline{OE}$ pin of the latch. In case of write operation, data is written to the data register (at base address of the port) and it is latched into 74LS374 since $\overline{OE}$ is enabled. The output of 74LS374 is connected to pin 2 to 9 COM port and therefore, data appears on data lines of COM port. In case of read operation, the latch (74LS374) is disabled and hence data written to the data register causes the byte to be latched into 74LS374, but not to appear on the data lines. Now the data available on the port lines can be read by enabling the 74LS244, i.e. reading the data register.

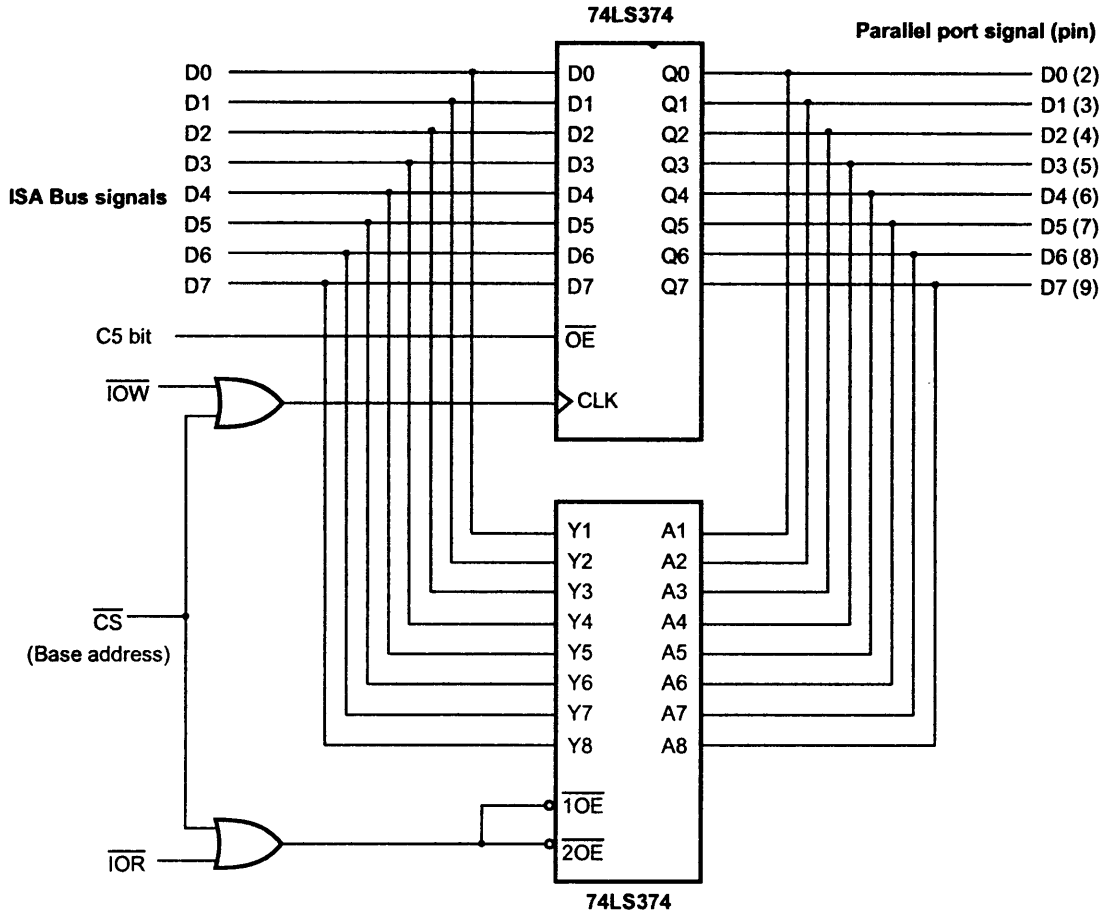



**Fig. 12.16 Bidirectional function of parallel port**

## 12.3.4 Accepting 16-bit Input using Bidirectional Parallel Port

The Fig. 12.17 shows the circuit to read 16-bit data using bidirectional parallel port. The circuit consists of two buffers (IC74LS244). It allows to read 8-bits at a time and uses the data lines (pins 2-9) for data transfer. To carryout data transfer, the C5 bit of the control register is set to disable latch. Then C0 bit of the control register is set to send low on pin 0. This enables lower buffer to tranfser lower 8-bits by reading the data on the data lines through data register. Then C0 bit of the control register is set to send high on pin 0. This enables higher buffer to transfer higher 8-bits by reading the data on the data lines through data register.

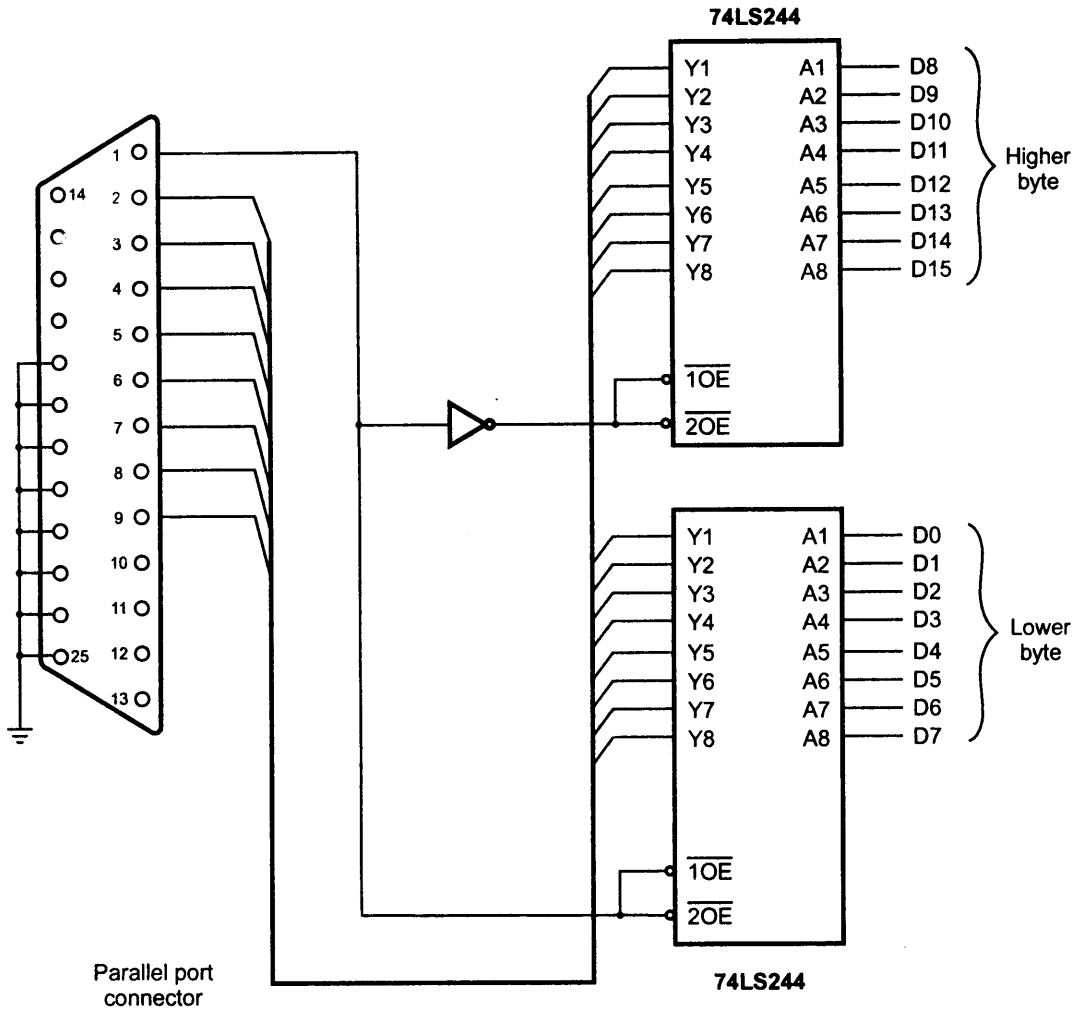



**Fig. 12.17 16-bit input interface for the bidirectional parallel port**

## 12.3.5 Interfacing 8-bit ADC using Parallel Port

The Fig. 12.18 shows the interfacing of 8-bit ADC using parallel port. The circuit interfaces 8-bit successive approximation ADC AD570 to the parallel port. The AD570 uses two control signals : B/$\overline{C}$ (Blank and $\overline{\text{Convert}}$) and $\overline{DR}$ ($\overline{\text{Data Ready}}$). B/$\overline{C}$ signal should remain high during the conversion process. The B/$\overline{C}$ is derived from C0 bit of the control register. However, this signal gives a pulse. Thus D flip-flop is used to generate B/$\overline{C}$ signal. The output of D flip-flop gives high when C0 signal activates the clock input of D flip-flop and it will remain high until the activation of its $\overline{CLR}$ (clear) input.

As the B/$\overline{C}$ input is driven low a conversion starts. Upon completion of the conversion, the $\overline{DR}$ line goes low and the data appears at the output. Putting the B/$\overline{C}$ input high three states the outputs and readies the device for the next conversion. This is illustrated in Fig. 12.18.

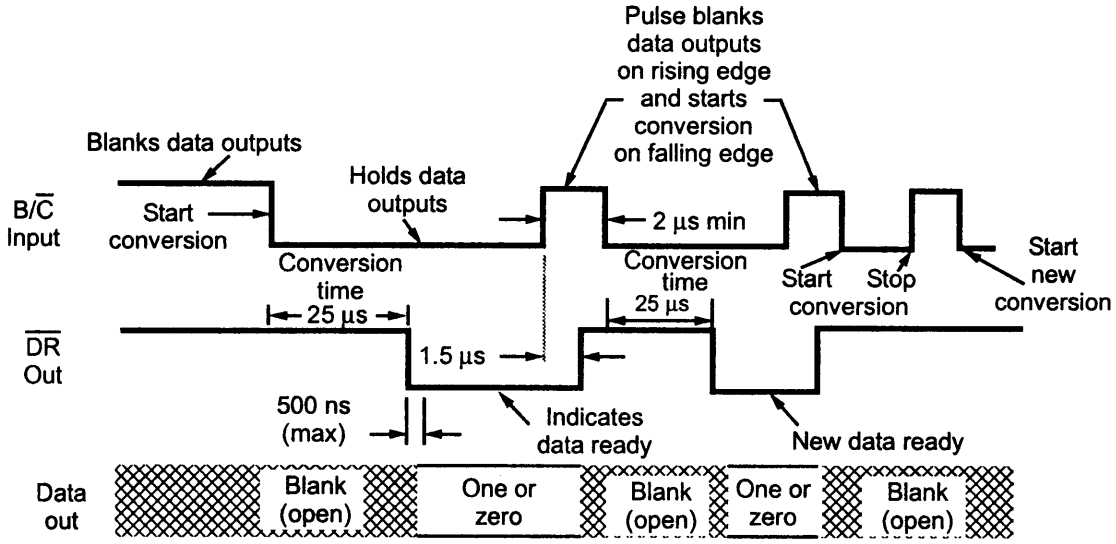



**Fig. 12.18**

The positive edge to B/$\overline{C}$ input is issued through the bit C0 of the control register (pin 1). The end of conversion is determined by polling the status of $\overline{DR}$ signal through the bit 6 of the status register. Thus $\overline{DR}$ signal is connected to the pin-10 of the connector. When the conversion is complete, the output operation of the data register is disabled and converted data is read through data register.

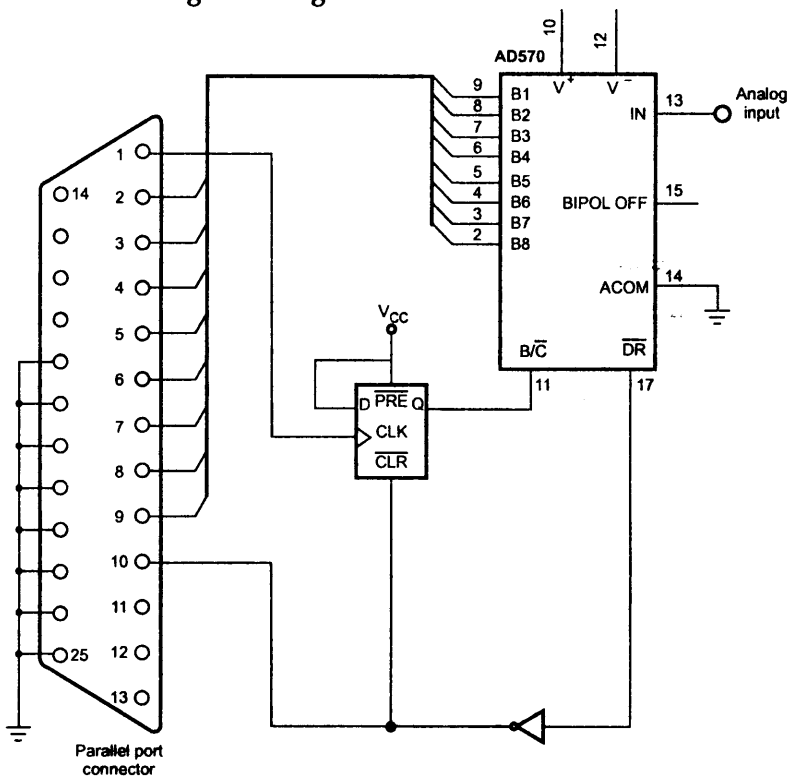



**Fig. 12.19 AD570 interfaced to parallel port**

## 12.4 Universal Serial Bus (USB)

The **Universal Serial Bus (USB)** was born out of the frustration of PC users experience trying to connect an incredibly wide range of peripherals to their computers. This was not possible with the existing centronics parallel interface and the RS-232 serial port interface. These interfaces could not handle increasing computer power and the number of peripherals. They have become bottle-neck of slow communication, with limited options for expansion. This is the situation that prompted the development of USB. The result is versatile interface that can replace existing interfaces to low - to moderate - speed standard and custom peripheral types on computers of all types. USB gives fast and flexible interface for connecting all kinds of peripherals.

USB is playing a **key role** in fast growing consumer areas like digital imaging, PC telephony, and multimedia games, etc. The presence of USB in most new PCs and its plug-n-play capability, means that PCs and peripherals (such as CD ROM drives, tape and floppy drives, scanners, printers, video devices, digital cameras, digital speakers, telephones, modems, keyboards, mice, digital joysticks and others) will automatically configure and work together, with high degree of reliability, in this exciting new application areas. USB opens the door to new levels of innovation and its use for input devices. There are also brand new opportunities of all types of peripherals from printers to scanners to high speed connection such as Ethernet, DSL, cable and satellite communications.

USB has advantages that specifically benefit developers, including the hardware designers who select components and design the circuits, the PC programmers who write the software that communicates with USB peripherals, and peripherals programmers who write the code that resides inside USB peripherals.

### 12.4.1 USB Features

#### 1. Simple connectivity

USB offers simple connectivity. It reduces the proliferation of cables and wall transformers. With USB there is no need to open the computer's enclosure to add an expansion card for each peripheral. A typical PC has two USB ports. You can expand the number of ports by connecting a USB hub to an existing port. Each hub has additional ports for attaching more peripherals or hubs.

#### 2. Simple cables

The USB cable connector are keyed so you cannot plug them in wrong. The USB connectors are slim in contrast to typical RS-232 and parallel connectors. Cable lengths are limited to 5 metres for 12 Mbps connections and 3 metres for 1.5 Mbps.

**3. One interface for many devices**

USB is versatile enough to be usable with many kinds of peripherals with no need of having a different connector and protocols for each peripheral. USB supports all kinds of data, from slow mouse inputs to digitized audio and compresses video.

**4. Automatic configuration**

When a user connects a USB peripheral to a powered system, windows automatically detects the peripheral and loads the appropriate software driver for it. There is no need to locate and run a setup programme or restart the system before using the peripheral.

**5. No user setting**

USB peripherals do not have user selectable settings such as port address and interrupt request (IRQ) lines.

**6. Frees hardware resources for other devices**

Using USB for as many peripherals as possible frees up IRQ lines for the peripherals that do require them. The PC does dedicate a series of port addresses and one interrupt request (IRQ) line to the interface, and also individual peripherals do not require additional resources. In contrast, each non USB peripheral requires dedicated port addresses, often an IRQ line, and sometimes an expansion slot (for example a parallel port card).

**7. Hot pluggable**

You can install or remove a peripheral regardless of the power state i.e. whether or not the system and peripheral are powered, they do not damage the PC or peripheral. The operating system detects when a USB device is attached and readies it for use.

**8. Data transfer rates**

USB supports three data transfer rates, 480 Mb/s (high-speed), 12 Mb/s (full-speed) and 1.5 Mb/s (low-speed).

**9. Coexistence with IEEE 1394**

USB 2.0 and IEEE 1394 offer similar data rate primarily differ in terms of application s. The USB 2.0 is preferred to be connected for most PC peripherals where as IEEE 4's primary target is audio visual consumer electronic devices such as digital ncorders, digital VCRs, DVDs, and digital TVs.

**10. Reliability**

Reliability of USB results from both the hardware design and data transfer protocols. The hardware specification for USB drivers, receivers and cables eliminate most noise that could otherwise cause data errors. In addition, the USB protocol enables detecting of data

errors and notifying the senders so it can retransmit. The detecting, notifying, and retransmitting are typically done in hardware and do not require any programming.

**11. Low cost**

Even though USB is more complex than earlier interfaces, its components and cables are inexpensive. A device with a USB interface is likely to cost the same or less than its equivalent older interfaces.

**12. Low power consumption**

Power circuits and code automatically power down USB peripherals when not in use, yet keep them ready to respond when needed. In addition to the environmental benefits of reduce power consumption, this feature is especially useful on battery powered computers where every milliampere counts.

**13. Flexibility**

USB's four transfer types and three speed make it feasible for many types of peripherals.

**14. Operating system support**

Windows 98 was the first Windows operating system to reliably support USB, and its successors such as Windows 2000 support USB as well. Other computers and operating systems also have USB support. ON apple's iMac, the only peripherals connectors are USB. Other Macintoshes also support USB, and support is in progress for Linux, NetBSD, and FreeBSD.

## 12.4.2 USB System

The Fig. 12.20 shows the basic components of USB system. It consists of USB host, USB device and USB cable. The USB host is a personal computer (PC) and devices are scanner, printer etc. There will be only one host in the USB system, however there can be 127 devices in the USB system.

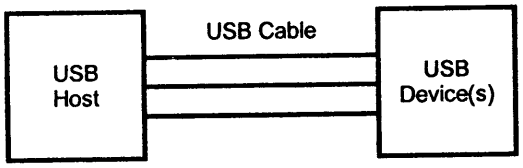



**Fig. 12.20 USB system**

## 12.4.3 Cables

USB cables are designed to ensure correct connections are always made. By having different connectors on host and device, it is imposssible to connect, two hosts or two devices together.

USB requires a shielded cable containing 4 wires. Two of these, D + and D –, form a twisted pair responsible for carrying a differential data signal, as well as some single-ended signal states. (For low speed the data lines may not be twisted). The signals on these two wires are referenced to the (third) GND wire.

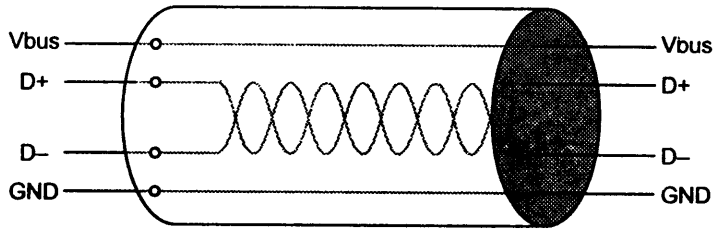



**Fig. 12.21 Makeup of USB cable**

The fourth wire is called VBUS, and carries a nominal 5 V supply, which may be used by a device for power.

### 12.4.4 USB Connector

USB uses different connectors on host and device to enforce correct connections. "A" Type connector point downstream from a Host or Hub, while "B". Type connector point uspstream from a USB device or hub.

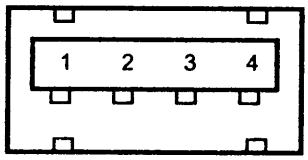



**Fig. 12.22 Type A connector**

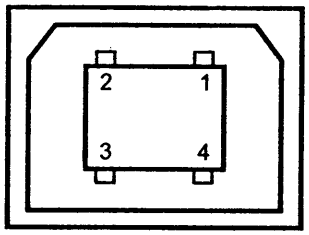



**Fig. 12.23 Type B connector**

**Standard "A" and Standard "B" connector Pin Assignments**

| Contact Number | Signal Name | Typical Cable Colour |
|---|---|---|
| 1 | VBUS | Red |
| 2 | D – | White |
| 3 | D + | Green |
| 4 | GND | Black |
| Shell | Shield | Drain Wire |

A mini-B plug has also been defined as an alternative to the standard B connector on handheld and portable devices. The mini-B connector has a fifth pin, named ID, but it is not connected.

**Mini-B Connector Pin Assignments**

| Contact Number | Signal Name | Typical Cable Colour |
|---|---|---|
| 1 | VBUS | Red |
| 2 | D – | White |
| 3 | D + | Green |
| 4 | ID | No connection |
| 5 | GND | Black |
| Shell | Shield | Drain Wire |

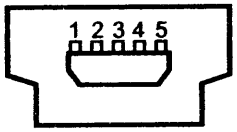



**Fig. 12.24 "Mini B" type connector**

## 12.4.5 USB Data

As mentioned earlier, the USB data signals are biphase signals. They are generated using a circuit shown in Fig. 12.25.

Usually, IC 75773 from Texas instruments is used as both the differential line driver and receiver in the above circuit.

The USB uses NRZI (non-return-to zero, inverted) data encoding method for transmitting data packets. In this method, the signal level does not change for the transmission of logic 1. However, it is inverted for each change to a logic 0. This is

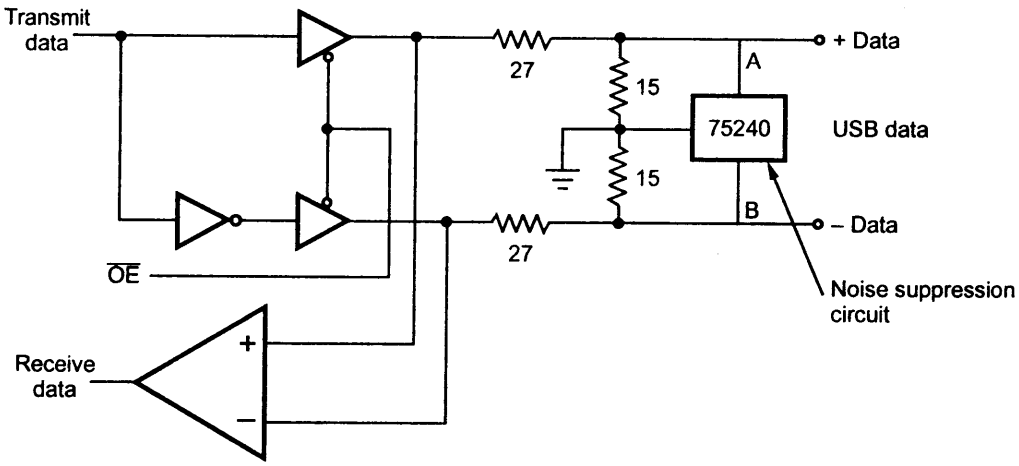



**Fig. 12.25 USB interface using a pair of CMOS buffers**

illustrated in Fig. 12.26. In this method data bits are always transmitted beginning with the least significant bit first, followed by subsequent bits.

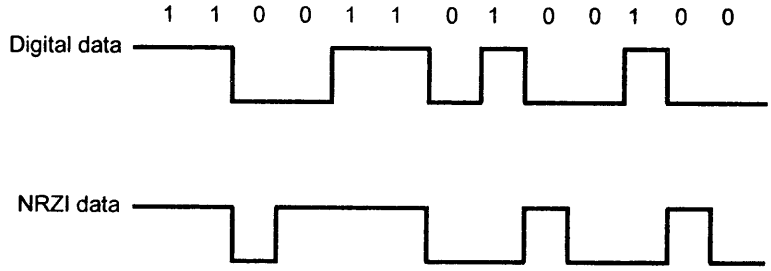



**Fig. 12.26 NRZI encodinging method**

To maintain the signal frequency in the specified range i.e. to achieve synchronization we have to insert a sync bit in the data stream, if a logic 1 is transmitted for more than six bits in a row. The process of inserting sync bit is known as **bit stuffing.** The bit stuffing is illustrated in Fig. 12.27. Bit stuffing ensures that the receiver can maintain synchronization for long strings of logic 1s.

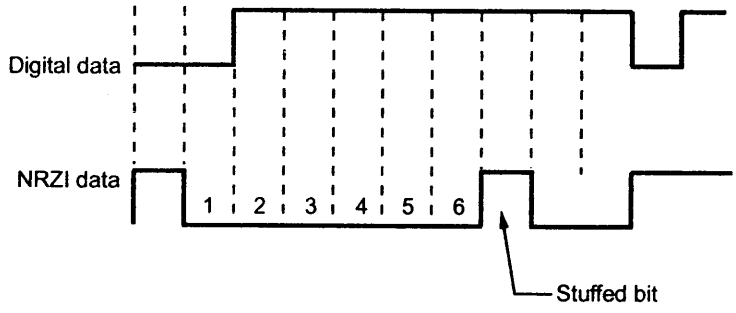



**Fig. 12.27 Bit stuffing**

The Fig. 12.28 shows the flowchart to generate USB data from the raw digital serial data.

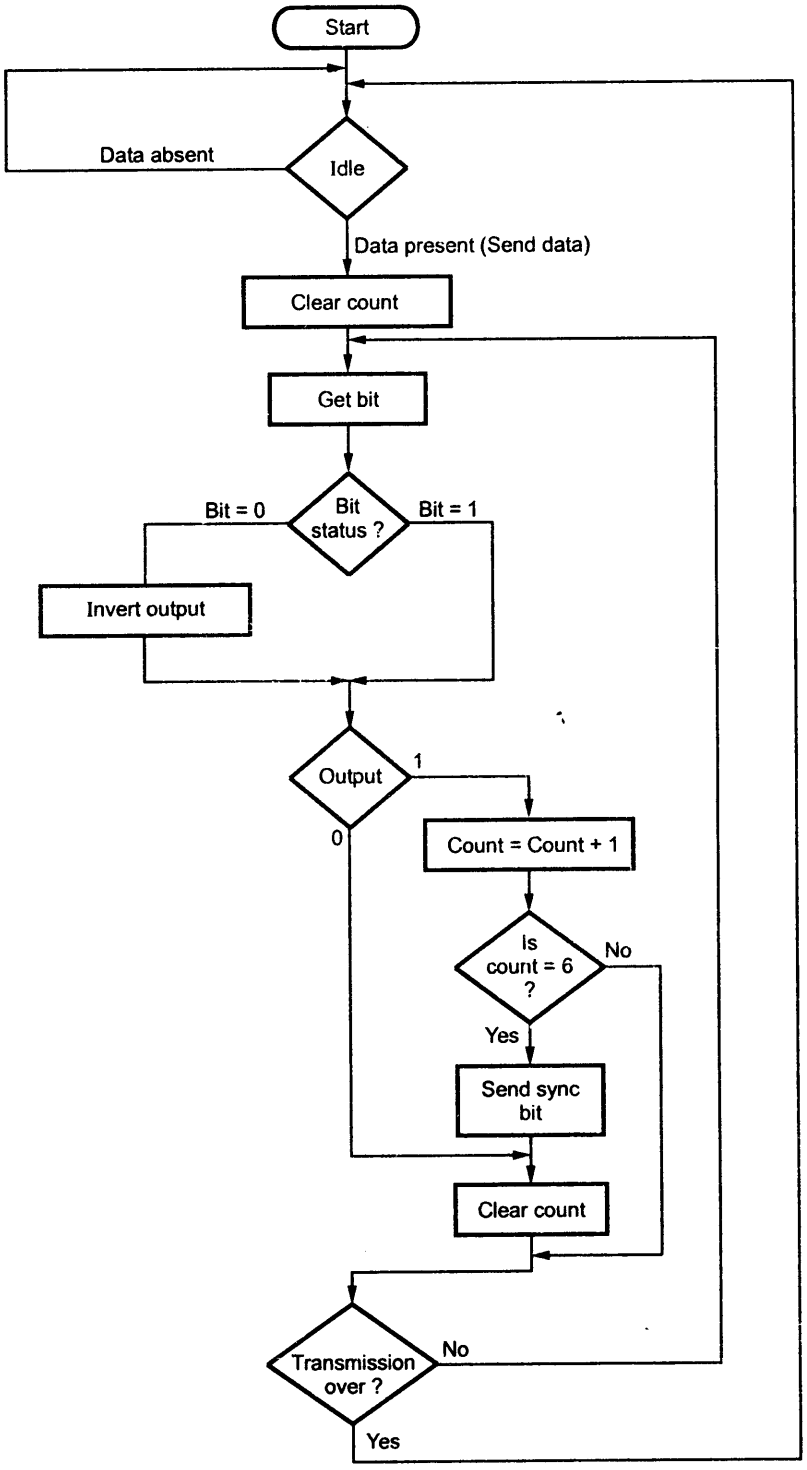



**Fig. 12.28**

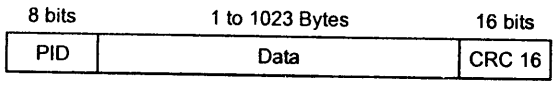
## 12.4.6 USB Commands

We have seen how USB data is generated. This data is transmitted to particular receptor with the use of USB commands. The communication begins with the transmission of the sync byte (80 H). It is followed by the packet identification byte (PID). The PID contains eight bits, but only the rightmost four bits contain the type of packet. The leftmost four bits of the PID are the complement form of four rightmost bits. For example, if a command is 1001, the actual PID byte is 0110 1001. The Table 12.3 shows the available 4-bit PIDs and their 8-bit codes. The PIDs are also used as token indicators, as data indicators, and for handshaking.

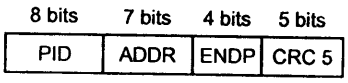
| PID Code | Name | Type | Description |
|---|---|---|---|
| E1H | OUT | Token | Host → function transation |
| D2H | ACK | Handshake | Receiver accepts packet |
| C3H | Data0 | Data | Data packet PID even |
| A5H | SOF | Token | Start of frame |
| 69H | IN | Token | Function → host transation |
| 5AH | NAK | Handshake | Receiver does not accept data |
| 4BH | Data1 | Data | Data packet PID odd |
| 3CH | PRE | Special | Host preamble |
| 2DH | Setup | Token | Setup command |
| 1EH | Stall | Token | Stalled |

**Table 12.3 : PID codes and description**

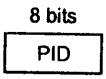
The Fig. 12.29 shows the formats of data, token, handshaking and start-of frame packets used on the USB.



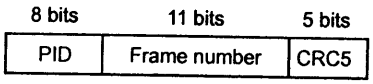

(a) Data packet




(b) Token packet




(c) Handshaking packet




(d) Start of frame packet

**Fig. 12.29 Packet formats**

The data packet begins with PID, then data byte and ends with CRC (cyclic redundancy check) code. Packets use two types of CRC codes : one is a 5-bit CRC and the other (used for data packets) is a 16-bit CRC. The 5-bit CRC is generated with the $X^5 + X^2 + 1$ polynomial. On the other hand, the 16-bit CRC is generated with the $X^{16} + X^{15} + X^2 + 1$ polynomial.

The USB uses the ACK (acknowledge) and NAK (Not acknowledge) tokens to co-ordinate the transfer of data packets between the host system (host is a PC or other computer that contain two components : controller and a root hub). (A hub is device that contains one or more connecters or internal connections to USB devices along with the hardware to enable communicating with each device and the USB device.) Once a data packet is transferred from the host to the USB device, the USB device either transmits and ACK or a NAK token back to the host. If the data and CRC are received without error, the ACK is sent; otherwise, the NAK is sent. If the host receives a NAK token, it retransmits the data packet until the receiver receives it without error. This method of data transfer is known as **stop and wait flow control**. In this method, the host has to wait for the client to send an ACK or NAK before transferring additional data packets.

## 12.4.7 USB Host

The USB host does the following basic functions.

1. It recognises the attachment and removal of USB devices.
2. It installs a device when it is pluged in.
3. It manages flow of control information between the host and USB devices.
4. It manages flow of data between the host and USB devices.
5. It gathers activity and status information of USB devices.
6. It provides power to low power USB devices connected to the host.

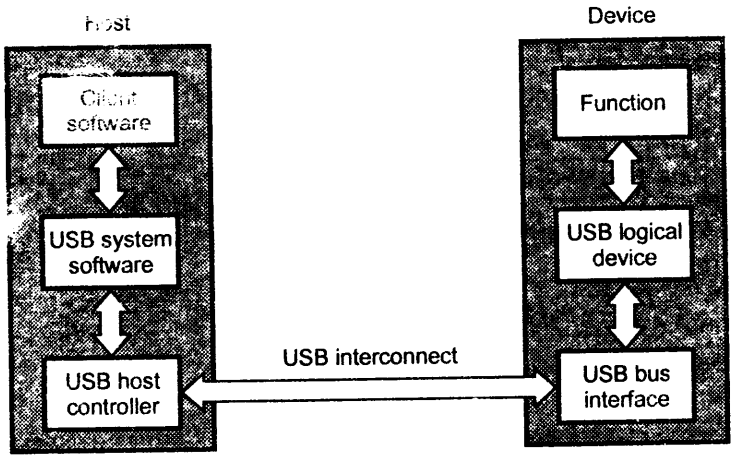



**Fig. 12.30 Implementation of USB interface**

The Fig. 12.30 shows the connection between USB host and USB device. The USB host constitutes USB host controller hardware, USB system software such as operating system, controller driver, and USB driver and the client software. The client software is a device driver for the USB device that remains resident and executes on the host.

### 12.4.8 USB Device

The USB device does the following basic functions.

1. It responds to all requests made by the host.
2. It monitors the device address in each communication and selects itself for comminication if it is the addressed device.
3. It sends the data with error correcting bits and it receives data after checking errors if any. In case of error in data transfer, it requests for retransmission of data.
4. If the USB device is self-powered, it manages its own power supply.

The Fig. 12.31 shows the simple hardware for USB device. It is a composition of USB bus interface hardware, protocol controller and custom I/O device. The bus interface hardware contains a Serial Interface Engine (SIE) and a transceiver.

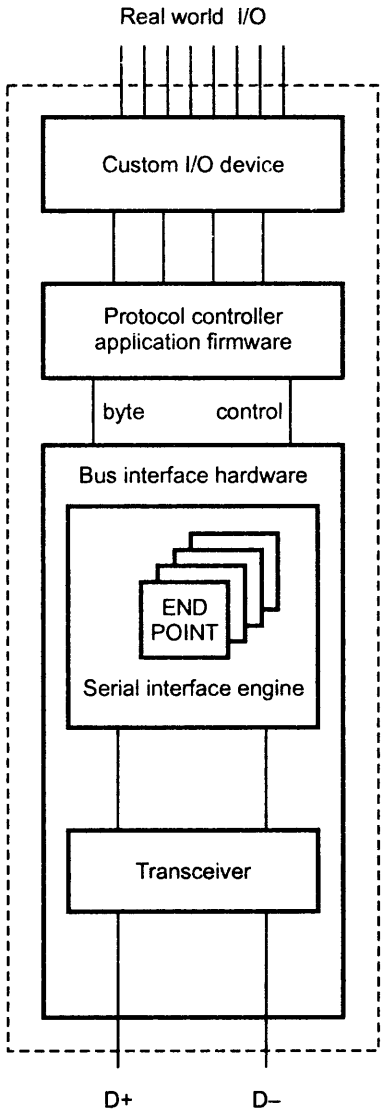



**Fig. 12.31 Hardware for USB device**

## 12.4.9 USB Descriptor

The USB device contains a number of descriptors which help to define what the device is capable of. The descriptor is a data structure which contains information about the device and its properties. Before going to see the details of descriptor we need to have an idea what do we mean by the configurations, interfaces and end points.

### Configurations

The architecture of generic USB device is multi-layered. A device consists of one or more configurations, each of which describes a possible setting the device can be programmed into. Such settings can include the power characteristics of the configuration (for example, the maximum power consumed by the configuration and whether it is self-powered or not) and whether the configuration supports remote wake-up.

A device can have only one configuration at a time. To change configuraion the whole device would have to stop functioning. Different configurations might be used, for example, to specify different current requirements, as the current required is defined in the configuration descriptor.

### Interface

Each configuration contains one or more interfaces that are accessible after the configuration is set. An interface provides the definitions of the functions available within the device and may even contain alternate settings within a single interface. For example, an interface for an audio device may have different settings we can select for different bandwidths.

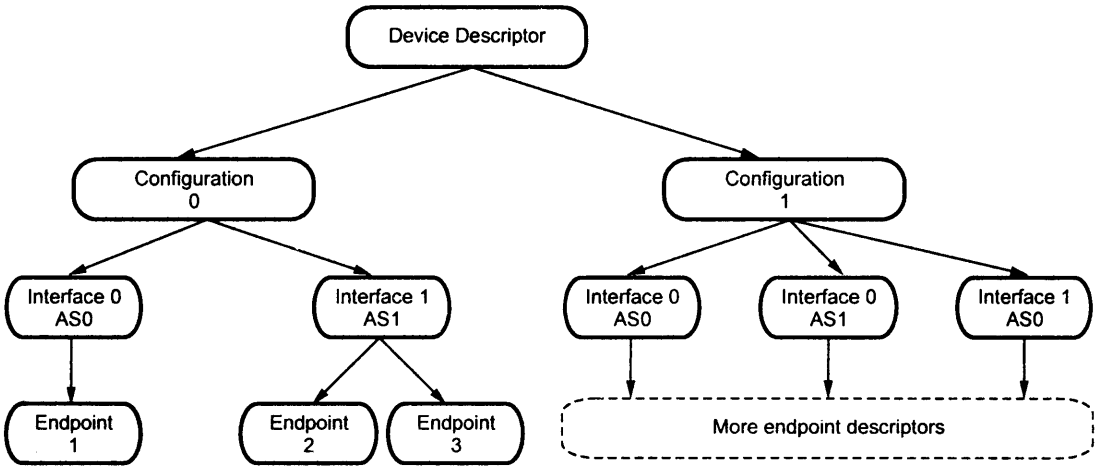



**Fig. 12.32 USB descriptor**

### Endpoint

Endpoints are the unidirectional access points of communicating with a device. They provide buffers to temporarily store incoming or outgoing data from the device. Each endpoint has a unique address within a configuration, the endpoints number plus its direction. The endpoint has characteristics that describe the communication it supports, such as transfer type, maximum packet size, and transfer direction (input or output). There may be multiple endpoints inside a device. Each device has at least one endpoint -- "endpoint 0"-- which is used as a control endpoint. It must be able to both send and receive data, but can only communicate in one direction at a time. Typically, when a device receives data such as an Out or Setup command from the host, this data is stored in the endpoint and the device's microporcessor is interrupted and works on this data. When a device receives an In command that is addressed to it from the host, data for the host that is stored in the endpoint is send to the host.

The data endpoint supports unidirectional flow of data i.e. they can either receive data or send data.

### Standard Descriptors

- **Device descriptor** : It describes general information about the device, like Vendor, Product and Revision ID, supported device class, subclass and protocol if applicable, maximum packet size for the default endpoint, etc. A USB device has only one device descriptor.
- **Configuration descriptors** : They describe the number of interfaces in this configuration, suspend and resume functionalily supported and power requirements.
- **Interface descriptors** : They describe interface class, subclass and protocol if applicable number of alternate settings for the interface and the number of endpoints.
- **Endpoint descriptors** : They describe endpoint address, direction and type, maximum packet size supported and polling frequency if type is interrupt endpoint. There is no descriptor for the default endpoint (endpoint 0 ) and it is never counted in an interface descriptor. Endpoint descriptor contains information required by the host to determine the bandwidth requirements of each endpoint.
- **String descriptors** : They are optional and provide additional information in human readable unicode format. They can be used for vendor and device names or serial numbers.

## 12.4.10 Device Controller

The USB device controller is nothing but a microprocessor, microcontroller or a digital signal processor (DSP) in the USB device. It controls many of the operations of the device. It executes a program to respond various requests from the host.

### 12.4.11 Functions

The USB device provides various capabiltities to the USB host. For example, keyboard device allows to provide input to PC. USB refers these capabilitites as functions. Functions provided by USB I/O device can be classified into human interface device HID such as input device (keyboard, mouse), printer device (printer, plotter), imaging device (scanner), or mass storage device (CDROM, Floppy drive, DVD drive).

### 12.4.12 Enumeration

Whenever a USB device is attached to the bus it is recognized by the host. The host system software identifies the capabilities of the connected USB device after required handshaking between the host system software and the control endpoint. It then assigns unique device number to the device. This process is known as **enumeration.**

### 12.4.13 USB Hub

Physically there exist one, two or four USB ports at the rear panel of computer. These ports can be used to attach normal devices or a hub. A hub is a USB device which extends the number of ports to connect other USB devices. The maximum number of user devices is reduced by the number of hubs on the bus (i.e. if we attach 50 hubs, then at most (77 = 127 – 50) additional devices can be attached. Hubs can be cascaded upto seven levels deep. Hubs are always full speed devices. If the hub is self powered, then any device can be attached to it. However if the hub is powered, then only low power (100 mA max) devices can be attached to it. A bus powered hub should not be connected to another bus powered hub. The Fig. 12.33 shows the USB hub and device attachments to it through ports. The USB hub monitors USB signals, handles transactions addressed to it, and repeats

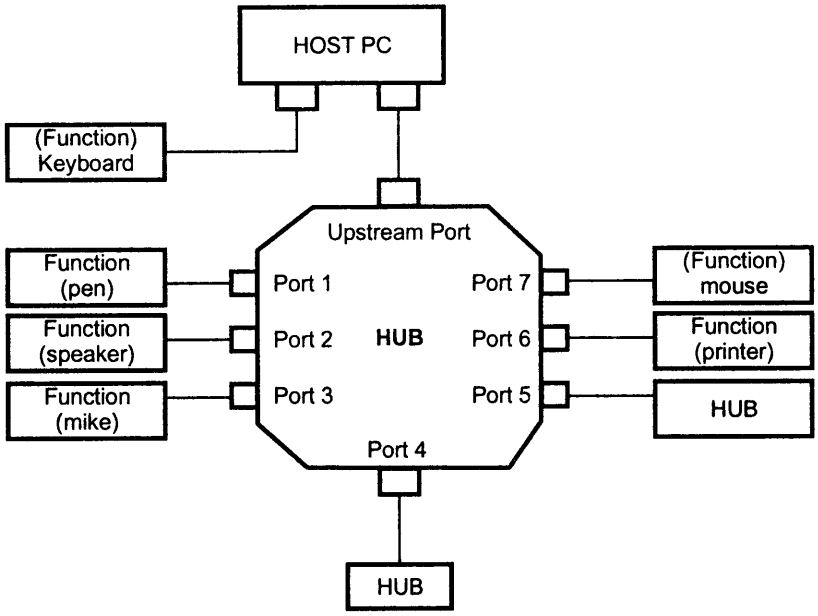



**Fig. 12.33 USB hub**

other transactions to respective device. USB hubs have status bits that are used to inform the host controller the attachment and removal status on one of their ports.

Normally the physical ports of the host controller are handled by a virtual root hub. The hub is simulated by the host controllers device driver and helps to unify the bus topology. So every port can be handled in the same way by the USB subsystem's hub driver. This is illustrated in Fig. 12.34.

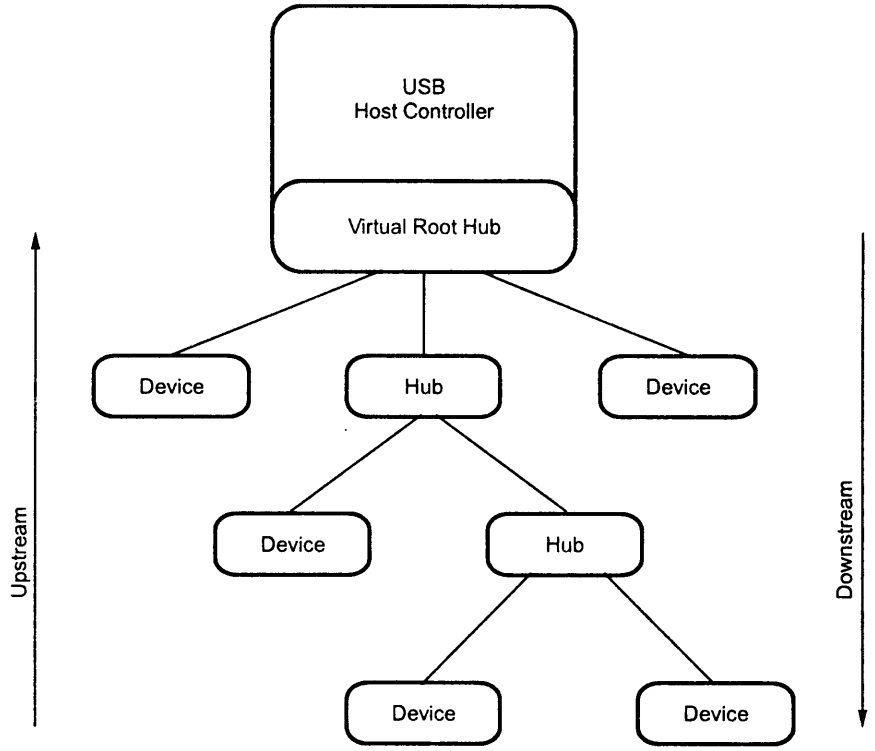



**Fig. 12.34 Hub connections with virtual root hub**

## Review Questions

1. *List the features of PCI bus.*
2. *Write a note on PCI configurations.*
3. *Describe the PCI bus structure.*
4. *Explain the PCI commands.*
5. *Write a note on PCI arbitration.*
6. *Write a short note on configuration space of PCI interface.*
7. *Give the formats of status and control registers in the configuration memory.*
8. *Write a short note on parallel printer interface.*
9. *Explain the control and handshaking signals of printer interface.*
10. *What is centronics interface ?*

11. *Give the formats for status ports of centronics interface.*
12. *Draw the hardware for accepting 16-bit input using parallel port and explain.*
13. *Explain the interfacing of stepper motor through parallel port.*
14. *Explain the bidirectional operation of parallel port.*
15. *Draw the hardware for accepting 16-bit input using bidirectional parallel port and explain.*
16. *Explain the interfacing of 8-bit ADC using parallel port*
17. *Explain the features of USB.*
18. *Give the details of USB connector with the help of diagram.*
19. *How USB data is generated ? Explain the encoding method used by the USB.*
20. *What is bit stuffing ?*
21. *Draw the flow chart explaining the process of generating USB data from the raw digital serial data.*
22. *Write a short note on USB commands.*
23. *What do you mean by stop and wait flow control ?*
24. *Explain the USB system.*
25. *Write a short note on USB cables.*
26. *Explain the functions of USB host.*
27. *Draw and explain the interfacing of host and device.*
28. *Explain the functions of USB device.*
29. *Draw the diagram of hardware for USB device.*
30. *Write a note on USB descriptor.*
31. *Define : configuration, interface and end point.*
32. *Define functions.*
33. *What is enumeration ?*
34. *Write a short note on USB hub.*

❑❑❑

# 13 The 80386, 80486 and Pentium Processor

## 13.1 Introduction to 80386 Microprocessor

The 80386DX is a true 32-bit microprocessor. It has 32-bit internal registers, a 32-bit data bus, and a 32-bit address bus. The 32-bit address bus allows user to design systems with ($2^{32}$) 4 gigabytes of main memory, while 32-bit data bus allows 4-bytes to be read from, written to, or fetched from that memory at a time. Its main features are : Different operating modes, new register set, expanded instruction set, memory management unit, page translation mechanism, and task management functions.

One of the most interesting features of the 80386DX is its ability to operate in three different modes :

1. Real address mode
2. Virtual 8086 mode
3. Protected virtual 8086 mode.

These three modes have their own significance. In real mode it functions basically as a fast 8086 or real mode 80286. The protected virtual address mode (Protected mode) operation provides paging, virtual addressing, multilevel protection and multitasking and debugging capabilities. The protected mode is a very complex environment that requires several segmentation and memory management unit related tables to be set up in memory before it will work at all. The segmentation mechanism used in protected mode is too different from that on the 8086 and 80186.

Before going to study operating modes it is necessary to see the functional units and programming model of 80386DX. This chapter therefore begins with the architecture of 80386DX and it is further devoted to explain the read mode, the protected mode, and the protected virtual 8086 mode of 80386, respectively. It also explains bus interface, memory interface and I/O interface.

### 13.1.1 Features of 80386

1. The 80386 is a 32-bit processor. The 32-bit ALU allows to process 32-bit data.
2. It has 32-bit address bus. So it can access upto 4 Gbyte ($2^{32}$ ) physical memory or 64 Tetrabyte ($2^{46}$ ) of virtual memory (Explained in the later section).
3. The 80386 runs with speed upto 20 MHz instructions per second.
4. The pipelined architecture of the 80386, allows simultaneous instruction fetching, decoding, execution and memory management. Instruction pipelining, a high bus bandwidth and on-chip address translation significantly shorten the average instruction execution time of 80386. These architectural design features enable the 80386 to execute 3 to 4 million instructions per second.
5. It allows programmers to switch between different operating systems such as PC-DOS and UNIX.
6. It can operate on 7 different data types :

   a. Bit b. Byte c. Word d. Double word

   e. Pword f. Quadword g. Tenbyte.

   It has built-in virtual memory management circuitry and protection circuitry required to operate an 80386 in these modes.
7. The 80386 can operate in real mode, protected mode or a variation of protected mode called virtual 8086 mode. In real mode it functions basically as a fast 8086 or real mode 80286. The protection mode operation provides paging, virtual addressing, multilevel protection and multitasking and debugging capabilities.
8. The 80386 microprocessor is compatible with their earlier 8086, 8088, 80186, 80188 and 80286 chips. Virtually anything that runs under these microprocessors will also run under the 80386.

### 13.1.2 Architecture of 80386DX

The internal architecture of the 80386 consists of six functional units :

1. Bus interface unit
2. Code fetch unit
3. Instruction decode unit
4. Execution unit
5. Segmentation unit
6. Paging unit

These units operate in parallel. Fetching, decoding, execution, memory management and bus accesses for several instructions are performed simultaneously. This parallel operation is called **pipelined instructions processing**. Fig. 13.1 shows instruction pipelining in 80386. With pipelining, each instruction is executed in stages.

As shown in the Fig. 13.1, bus unit is ahead of decode unit, decode unit is ahead of exécution unit and execution unit is ahead of memory management unit. Thus the processing of several instructions at different stages may overlap as shown in the Fig.13.1. The pipelining drastically reduces the overall processing time and thus results in high performance processing of 80386 instructions.

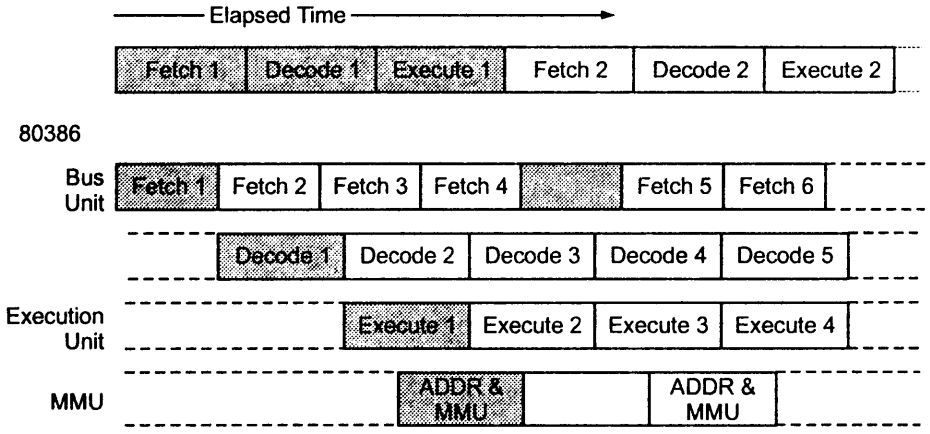



**Fig. 13.1 Instruction pipelining in 80386**

Fig. 13.2 shows the functional units of 80386 and interconnection between these units.

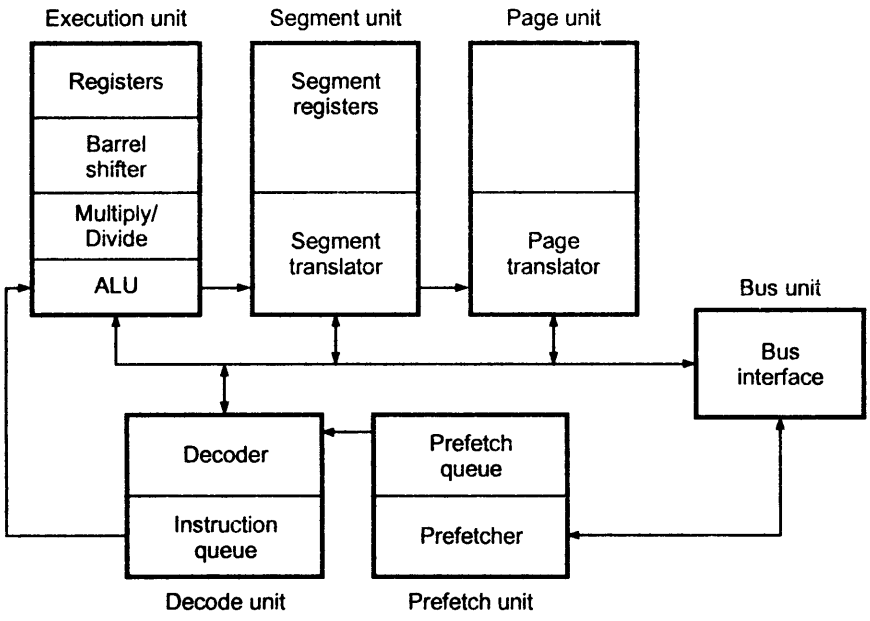



**Fig. 13.2 Functional units of 80386DX**

### Bus interface unit

The Bus Interface Unit is the 80386DX's communication with the outside world. It provides a full 32-bit bi-directional data bus and 32-bit address bus. The bus interface unit is responsible for following operations :

1. It accepts internal requests for code fetch and for data transfers from the code fetch unit and from the execution unit. It then prioritize the request and generates signals to perform bus cycles.
2. It sends address, data and control signals to communicate with memory and I/O devices.
3. It controls the interface to external bus masters and coprocessors.
4. It also provides the address relocation facility.

### Code prefetch unit

The code prefetch unit fetches sequentially the instruction byte stream from the memory. The code prefetch unit uses bus interface unit to fetch instruction bytes when the bus interface unit is not performing bus cycles to execute an instruction. These prefetched instruction bytes are stored in the 16-byte code queue. A 16-byte code queue holds these instructions until the decoder needs them. The prefetcher always fetches instructions in the order in which they appear in the memory. In fact, the prefetcher simply reads code one double word at a time, not caring whether it's bringing in complete instructions or pieces of two instructions with each access. When jump or call instructions are executed, the contents of the prefetched and decode queues are cleared out. In this case, prefetcher again starts filling up its queue. If the first instruction after a jump is a DIV, or some other instruction that takes several cycles without bus cycles, the prefetcher can catch up quickly.

Code prefetches are given a lower priority than data transfers. If memory access is without any wait state, prefetch activity never delays execution. Due to prefetch activity processor spends practically zero time waiting for the next instruction.

### Instruction decode unit

The instruction decode unit takes instruction bytes from the code prefetch queue and translates them into microcode. The decoded instructions are then stored in the instruction queue.

### Execution unit

The execution unit reads the instruction from the instruction queue and executes the instructions. It consists of three subunits : Control unit, Data unit and Protection test unit.

**1. Control unit :** It contains microcode and special hardware. The microcode and special hardware allows 80386DX to reduce time required for execution of multiply and divide instructions. It also speeds the effective address calculation.