First Edition : 2009

Microprocessors and Interfacing

A. P. Godse
D. A. Godse

Technical Publications Pune®

Strictly According to the Revised Syllabus of

JNTU - 2005 Course

Microprocessors & Interfacing

[EC 05400] Third Year, B. Tech., Semester-II (ECE / Elex. & Comm. / Elex. & Telematics / I.T. / Bio. Medical)
Third Year, B. Tech., Semester - I (Instru. & Control)

Atul P. Godse
M. S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Mrs. Deepali A. Godse
B.E. Industrial Electronics, M. E. (Computer)
Assistant Professor in Bharati Vidyapeeth's
Women's College of Engineering
Pune

Price Rs. 285/-

Visit us at : **www.vtubooks.com**

Technical Publications Pune®

Syllabus (Microprocessors & Interfacing)

Unit-I (Chapter-1, 2)

An overview of 8085, Architecture of 8086, Microprocessor, Special functions of general purpose registers, 8086 flag register and function of 8086 flags.

UNIT-II (Chapter-3)

Addressing modes of 8086, Instruction set of 8086, Assembler directives simple programs, Procedures, and Macros.

UNIT-III (Chapter-4)

Assembly language programs involving logical, Branch and Call instructions, Sorting, Evaluation of arithmetic expressions, String manipulation.

UNIT-IV (Chapter-5, 6)

Pin diagram of 8086-Minimum mode and maximum mode of operation, Timing diagram, Memory interfacing to 8086 (Static RAM and EPROM), Need for DMA, DMA data transfer method, Interfacing with 8237/8257.

UNIT-V (Chapter-7)

8255 PPI-Various modes of operation and interfacing to 8086, Interfacing keyboard, Displays, Stepper motor and actuators, D/A and A/D converter interfacing.

UNIT-VI (Chapter-8, 9)

Interrupt structure of 8086, Vector interrupt table, Interrupt service routines, Introduction to DOS and BIOS interrupts, 8259 PIC architecture and interfacing cascading of interrupt controller and its importance.

UNIT-VII (Chapter-10)

Serial data transfer schemes, Asynchronous and synchronous data transfer schemes, 8251 USART architecture and interfacing, TTL to RS 232C and RS232C to TTL conversion, Sample program of serial data transfer, Introduction to High-speed serial communications standards, USB.

UNIT-VIII (Chapter-11)

8051 Microcontroller architecture, Register set of 8051, Modes of timer operation, Serial port operation, Interrupt structure of 8051, Memory and I/O interfacing 8051.

Table of Contents :

Features of Book

- 8085, 8086/88 Architecture, programming and interfacing.
- Free download 8086 programs on www.vtubooks.com.
- 8051 Microcontroller architecture.
- Large number of programming examples.
- Programs using modular programming approach.
- Practical interfacing design examples.

Microprocessors & Interfacing

Atul P. Godse
M. S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Mrs. Deepali A. Godse
B.E. Industrial Electronics, M. E. (Computer)
Assistant Professor in Bharati Vidyapeeth's
Women's College of Engineering
Pune

Technical Publications Pune

Microprocessors & Interfacing

ISBN 9788184311259

Published by :

Technical Publications Pune®

#1, Amit Residency, 412, Shaniwar Peth, Pune - 411 030, India.

Printers :

Vikram Printers

34, Parvati Industrial Estate
Pune-Satara Road,
Pune - 411009.

Preface

Thanks to professors, students and authors of various technical books for their overwhelming response to our books. Looking at the feedback and the response we received from previous books, we are very pleased to release a text book on **Microprocessors and its Applications.**

The purpose of this book is to fulfil a need for text stating in plain, lucid and simple everyday language. This book provides a logical method for explaining and it prepares a background of the topic with essential illustrations. This text is provided with number of solved design examples which helps students to understand the application of microprocessor and microcontroller based systems.

The rapid spread of microprocessors in society has both simplified and complicated our lives. To get the conceptual view of the microprocessors, it is better to study them from the popular family like Intel. So we felt it is necessary to introduce a book which covers microprocessors and microcontroller with their features, internal architecture, internal organization and design details.

This text begins with the architecture of 8085 microprocessor. It explains all the details of 8085 microprocessor such as its architecture, pin description instruction set, memory and I/O interfacing and interrupts. The text then introduces a 16-bit microprocessor 8086. It also explains the details of 8086 like 8085. The text also explains various peripherals and their interfacing with microprocessors.

Finally, the text explains the 8051 microcontroller.

Acknowledgement

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family.

We are specially grateful to the great teacher **Prof. A.V. Bakshi** for his time to time, much needed, valuable guidance. Without the full support and cheerful encouragement of **Mr. Uday Bakshi** the book would not have been completed in time.

Finally, we wish to thank **Mr. Avinash Wani, Mr. Ravindra Wani** and the entire team of **Technical Publications** who have taken immense pain to get the quality printing in time.

Any suggestions for the improvement of the book will be acknowledged and appreciated.

Authors

Atul Godse

Deepali Godse

Dedicated to Neha & Ruturaj

Table of Contents

8086 Programs

1 An Overview of 8085

1.1 8085 Microprocessor

In this chapter we will see features, pin diagram, architecture, register structure, bus organisation, timing and control and instruction set of 8085 microprocessor.

The features of 8085 microprocessor are as given below :

1. It is an 8-bit microprocessor i.e. it can accept, process, or provide 8-bit data simultaneously.
2. It operates on a single +5V power supply connected at V_{CC}; power supply ground is connected to V_{ss}.
3. It operates on clock cycle with 50% duty cycle.
4. It has on chip clock generator. This internal clock generator requires tuned circuit like LC, RC or crystal. The internal clock generator divides oscillator frequency by 2 and generates clock signal, which can be used for synchronizing external devices.
5. It can operate with a 3 MHz clock frequency. The 8085A-2 version can operate at the maximum frequency of 5 MHz.
6. It has 16 address lines, hence it can access (2^{16}) 64 Kbytes of memory.
7. It provides 8 bit I/O addresses to access (2^8) 256 I/O ports.
8. In 8085, the lower 8-bit address bus (A_0 - A_7) and data bus (D_0 - D_7) are multiplexed to reduce number of external pins. But due to this, external hardware (latch) is required to separate address lines and data lines.
9. It supports 74 instructions with the following addressing modes :

   a) Immediate b) Register c) Direct d) Indirect e) Implied
10. The Arithmetic Logic Unit (ALU) of 8085 performs :

    a) 8 bit binary addition with or without carry

    b) 16 bit binary addition. c) 2 digit BCD addition.

    d) 8-bit binary subtraction with or without borrow

    e) 8-bit logical AND, OR, EX-OR, complement (NOT), and bit shift operations.

11. It has 8-bit accumulator, flag register, instruction register, six 8-bit general purpose registers (B, C, D, E, H and L) and two 16-bit registers (SP and PC). Getting the operand from the general purpose registers is more faster than from memory. Hence skilled programmers always prefer general purpose registers to store program variables than memory.
12. It provides five hardware interrupts : TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR.
13. It has serial I/O control which allows serial communication.
14. It provides control signals (IO/$\overline{M}$, $\overline{RD}$, $\overline{WR}$) to control the bus cycles, and hence external bus controller is not required.
15. The external hardware (another microprocessor or equivalent master) can detect which machine cycle microprocessor is executing using status signals (IO/$\overline{M}$, S_0, S_1). This feature is very useful when more than one processors are using common system resources (memory and I/O devices).
16. It has a mechanism by which it is possible to increase its interrupt handling capacity.
17. The 8085 has an ability to share system bus with Direct Memory Access controller. This feature allows to transfer large amount of data from I/O device to memory or from memory to I/O device with high speeds.
18. It can be used to implement three chip microcomputer with supporting I/O devices like IC 8155 and IC 8355.

1.2 Architecture of 8085

Fig. 1.1 (See Fig. on next page) shows the architecture of 8085.

It consists of various functional blocks as listed below :

- Registers
- Arithmetic and Logic Unit
- Instruction decoder and machine cycle encoder
- Address buffer
- Address/Data buffer
- Incrementer/Decrementer Address Latch
- Interrupt control
- Serial I/O control
- Timing and control circuitry.

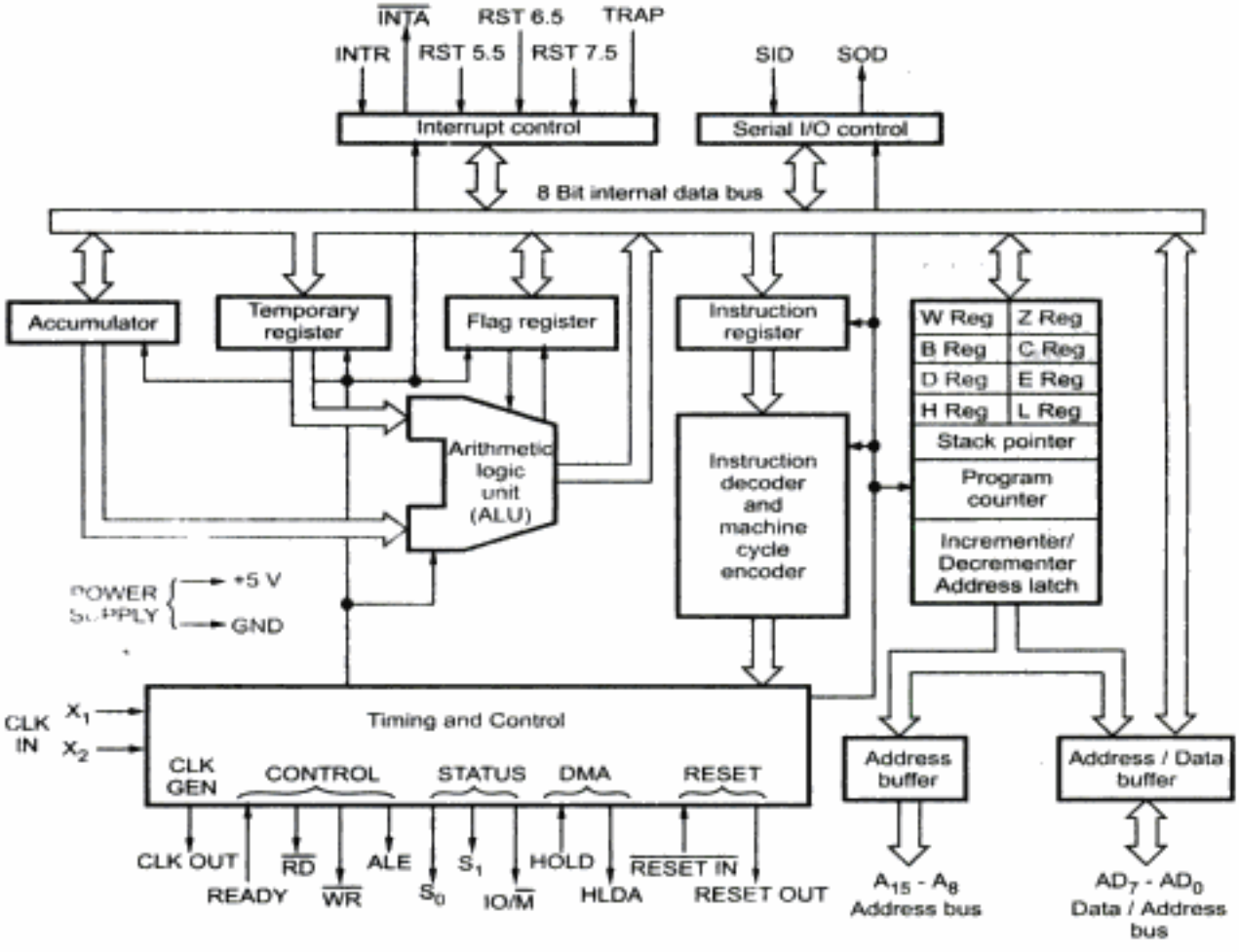


Fig. 1.1 Architecture of 8085

1.2.1 Register Structure

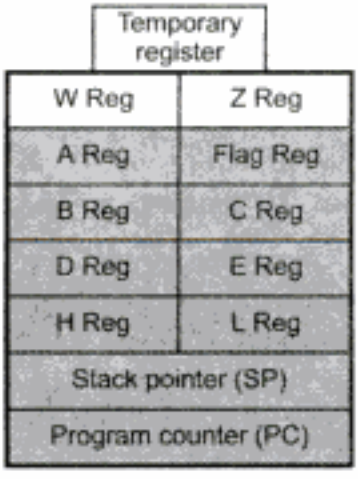


Fig. 1.2 Register structure of 8085

The Fig. 1.2 shows the register structure of 8085. The shaded portion of this register model is called **programmer's model** of 8085. It includes six 8-bit registers- (B, C, D, E, H and L) one accumulator, one flag register and two 16-bit registers (SP and PC). All these registers are accesible to programmer and hence they are included in the programmer's model. The remaining registers - temporary, W and Z are not accessible to the programmers; they are used by microprocessor for internal, intermediate operations.

The 8085 registers are classified as :

1. **General Purpose Registers**
2. **Temporary Registers**
   a) Temporary data register b) W and Z registers
3. **Special Purpose Registers**
   a) Accumulator b) Flag registers c) Instruction register
4. **Sixteen Bit Registers**
   a) Program Counter (PC) b) Stack Pointer (SP)

1. General Purpose Registers :

B, C, D, E, H, and L are 8-bit general purpose registers can be used as a separate 8-bit registers or as 16-bit register pairs, BC, DE, and HL. When used in register pair mode, the high order byte resides in the first register (i.e. in B when BC is used as a register pair) and the low order byte in the second (i.e. in C when BC is used as a register pair).

HL pair also functions as a data pointer or memory pointer. These are also called **scratchpad registers,** as user can store data in them. To store and read data from these registers bus access is not required, it is an internal operation. Thus it provides an efficient way to store intermediate results and use them when required. The efficient programmer prefers to use these registers to store intermediate results than the memory locations which require bus access and hence more time to perform the operation.

2. Temporary Registers

a) Temporary Data Register : The ALU has two inputs. One input is supplied by the accumulator and other from temporary data register. The programmer can not access this temporary data register. However, it is internally used for execution of most of the arithmetic and logical instructions.

For example : ADD B is the instruction in the arithmetic group of instructions which adds the contents of register A and register B and stores result in register A. The addition operation is performed by ALU. The ALU takes inputs from register A and temporary data register. The contents of register B are transferred to temporary data register for applying second input to the ALU.

b) W and Z Registers : W and Z registers are temporary registers. These registers are used to hold 8-bit data during execution of some instructions. These registers are not available for programmer, since 8085 uses them internally.

Use of W and Z registers :

The CALL instruction is used to transfer program control to a subprogram or subroutine. This instruction pushes the current PC contents onto the stack and loads the given address into the PC. The given address is temporarily stored in the W and Z registers and placed on the bus for the fetch cycle. Thus the program control is transferred

to the address given in the instruction. XCHG instruction exchanges the contents of H with D and L with E. At the time of exchange W and Z registers are used for temporary storage of data.

3. Special Purpose Registers :

a) Register A (Accumulator) : It is a tri-state eight bit register. It is extensively used in arithmetic, logic, load, and store operations, as well as in, input/output (I/O) operations. Most of the times the result of arithmetic and logical operations is stored in the register A. Hence it is also identified as **accumulator**.

b) Flag Register : It is an 8-bit register, in which five of the bits carry significant information in the form of flags : S (Sign flag), Z (Zero flag), AC (Auxiliary carry flag), P (Parity flag), and CY (carry flag), as shown in Fig. 1.3.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	X	AC	X	P	X	CY

Fig. 1.3 Flag register

S-Sign flag : After the execution of arithmetic or logical operations, if bit D_7 of the result is 1, the sign flag is set. In a given byte if D_7 is 1, the number will be viewed as negative number. If D_7 is 0, the number will be considered as positive number.

Z-Zero flag : The zero flag sets if the result of operation in ALU is zero and flag resets if result is non zero. The zero flag is also set if a certain register content becomes zero following an increment or decrement operation of that register.

AC-Auxiliary Carry flag : This flag is set if there is an overflow out of bit 3 i.e. , carry from lower nibble to higher nibble (D_3 bit to D_4 bit). This flag is used for BCD operations and it is not available for the programmer.

P-Parity flag : Parity is defined by the number of ones present in the accumulator. After an arithmetic or logical operation if the result has an even number of ones, i.e. even parity, the flag is set. If the parity is odd, flag is reset.

CY-Carry flag : This flag is set if there is an overflow out of bit 7. The carry flag also serves as a borrow flag for subtraction. In both the examples shown below, the carry flag is set.

ADDITICN		SUBTRACTION	
9B H	1001 1011	89 H	1000 1001
+ 75 H	+ 0111 0101	− AB H	− 1010 1011
Carry [1] 10 H	[1] 0001 0000	Borrow [1] DE H	[1] 1101 1110

c) Instruction Register : In a typical processor operation, the processor first fetches the opcode of instruction from memory (i.e. it places an address on the address bus and memory responds by placing the data stored at the specified address on the data bus). The CPU stores this opcode in a register called the instruction register. This opcode is further sent to the instruction decoder to select one of the 256 alternatives.

4. Sixteen Bit Registers

a) Program Counter (PC) : Program is a sequence of instructions. As mentioned earlier, microprocessor fetches these instructions from the memory and executes them sequentially. The program counter is a special purpose register which, at a given time, stores the address of the next instruction to be fetched. Program Counter acts as a pointer to the next instruction. How processor increments program counter depends on the nature of the instruction; for one byte instruction it increments program counter by one, for two byte instruction it increments program counter by two and for three byte instruction it increments program counter by three such that program counter always points to the address of the next instruction.

In case of JUMP and CALL instructions, address followed by JUMP and CALL instructions is placed in the program counter. The processor then fetches the next instruction from the new address specified by JUMP or CALL instruction. In conditional JUMP and conditional CALL instructions, if the condition is not satisfied, the processor increments program counter by three so that it points the instruction followed by conditional JUMP or CALL instruction; otherwise processor fetches the next instruction from the new address specified by JUMP or CALL instruction.

b) Stack Pointer (SP) : The stack is a reserved area of the memory in the RAM where temporary information may be stored. A 16-bit stack pointer is used to hold the address of the most recent stack entry.

1.2.2 Arithmetic Logic Unit (ALU)

The 8085's ALU performs arithmetic and logical functions on eight bit variables. The arithmetic unit performs bitwise fundamental arithmetic operations such as addition and subtraction. The logic unit performs logical operations such as complement, AND, OR and EX-OR, as well as rotate and clear. The ALU also looks after the branching decisions.

1.2.3 Instruction Decoder

As mentioned earlier, the processor first fetches the opcode of instruction from memory and stores this opcode in the instruction register. It is then sent to the instruction decoder. The instruction decoder decodes it and accordingly gives the timing and control signals which control the register, the data buffers, ALU and external peripheral signals (explained in later sections) depending on the nature of the instruction.

The 8085 executes seven different types of machine cycles. It gives the information about which machine cycle is currently executing in the encoded form on the S_0, S_1 and $IO/\overline{M}$ lines. This task is done by machine cycle encoder.

1.2.4 Address Buffer

This is an 8-bit unidirectional buffer. It is used to drive external high order address bus (A_{15}-A_8). It is also used to tri-state the high order address bus under certain conditions such as reset, hold, halt, and when address lines are not in use.

1.2.5 Address/Data Buffer

This is an 8-bit bi-directional buffer. It is used to drive multiplexed address/data bus, i.e. low order address bus (A_7-A_0) and data bus (D_7-D_0). It is also used to tri-state the multiplexed address/data bus under certain conditions such as reset, hold, halt and when the bus is not in use.

The address and data buffers are used to drive external address and data buses respectively. Due to these buffers the address and data buses can be tri-stated when they are not in use.

1.2.6 Incrementer/Decrementer Address Latch

This 16-bit register is used to increment or decrement the contents of program counter or stack pointer as a part of execution of instructions related to them.

1.2.7 Interrupt Control

The processor fetches, decodes and executes instructions in a sequence. Sometimes it is necessary to have processor the automatically execute one of a collection of special routines whenever special condition exists within a program or the microcomputer system. The most important thing is that, after execution of the special routine, the program control must be transferred to the program which processor was executing before the occurrence of the special condition. The occurrence of this special condition is referred as interrupt. The interrupt control block has five interrupt inputs RST 5.5, RST 6.5, RST 7.5, TRAP and INTR and one acknowledge signal $\overline{\text{INTA}}$.

1.2.8 Serial I/O Control

In situations like, data transmission over long distance and communication with cassette tapes or a CRT terminal, it is necessary to transmit data bit by bit to reduce the cost of cabling. In serial communication one bit is transferred at a time over a single line. The 8085's serial I/O control provides two lines, SOD and SID for serial communication. The serial output data (SOD) line is used to send data serially and serial input data (SID) line is used to receive data serially.

1.2.9 Timing and Control Circuitry

The control circuitry in the processor 8085 is responsible for all the operations. The control circuitry and hence the operations in 8085 are synchronized with the help of clock signal. Along with the control of fetching and decoding operations and generating appropriate signals for instruction execution, control circuitry also generates signals required to interface external devices to the processor, 8085.

1.3 Pin Definitions of 8085

Fig. 1.4 (a) and (b) show 8085 pin configuration and functional pin diagram of 8085 respectively. The signals of 8085 can be classified into seven groups according to their functions.

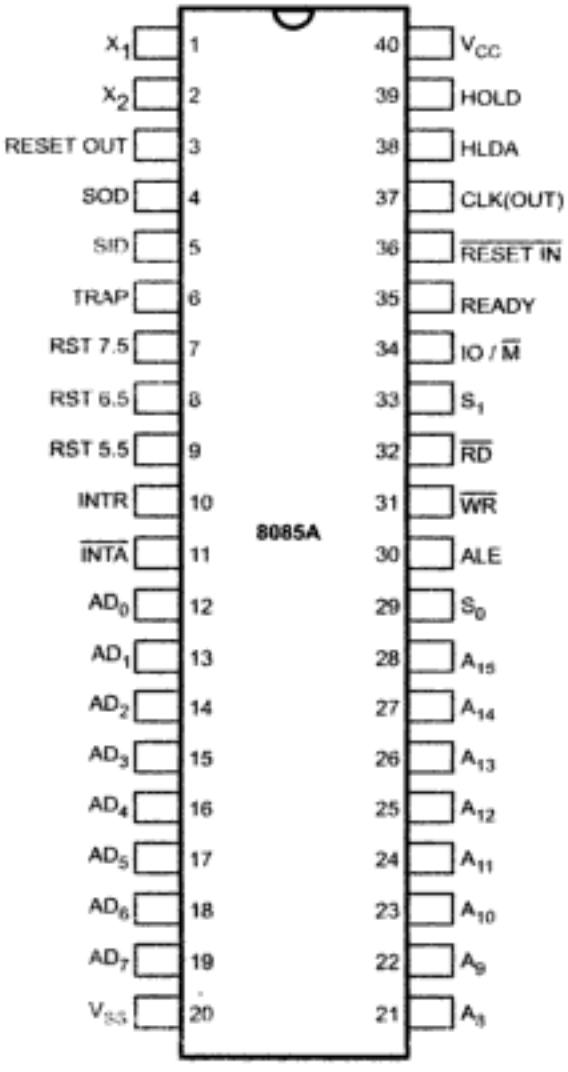


Fig. 1.4 (a) Pin configuration

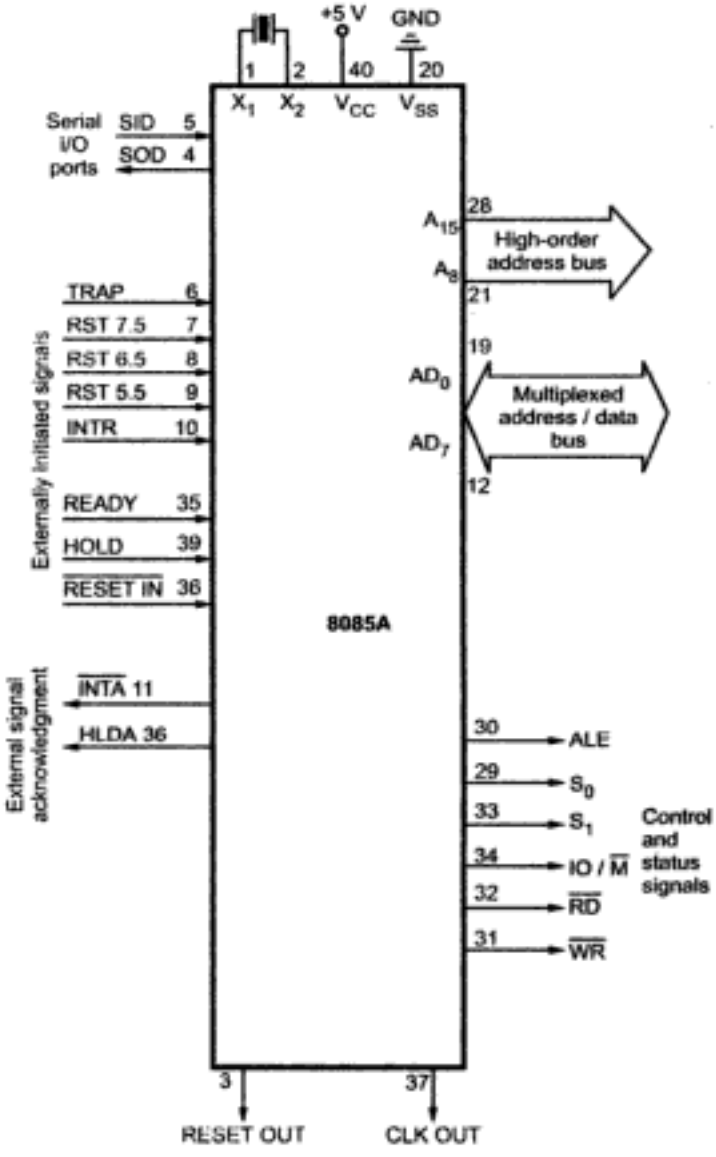


Fig. 1.4 (b) Functional pin diagram

a) Power supply and frequency signals.

b) Data bus and address bus

c) Control bus

d) Interrupt signals

e) Serial I/O signals

f) DMA signals

g) Reset signals

1.3.1 Power Supply and Frequency Signals

i) V_{CC} : It requires a single +5 V power supply.

ii) V_{SS} : Ground reference.

iii) X_1 and X_2 : A tuned circuit like LC, RC or crystal is connected at these two pins. The internal clock generator divides oscillator frequency by 2, therefore, to operate a system at 3 MHz, the crystal of tuned circuit must have a frequency of 6 MHz.

iv) **CLK OUT** : This signal is used as a system clock for other devices. Its frequency is half the oscillator frequency.

1.3.2 Data Bus and Address Bus

A) AD_0 to AD_7 : The 8 bit data bus (D_0 - D_7) is multiplexed with the lower half (A_0 - A_7) of the 16 bit address bus. During first part of the machine cycle (T_1), lower 8 bits of memory address or I/O address appear on the bus. During remaining part of the machine cycle (T_2 and T_3) these lines are used as a bi-directional data bus.

B) A_8 to A_{15} : The upper half of the 16 bit address appears on the address lines A_8 to A_{15}. These lines are exclusively used for the most significant 8 bits of the 16 bit address lines.

1.3.3 Control and Status Signals

A) ALE (Address Latch Enable) : We know that AD_0 to AD_7 lines are multiplexed and the lower half of address (A_0 - A_7) is available only during T_1 of the machine cycle. This lower half of address is also necessary during T_2 and T_3 of machine cycle to access specific location in memory or I/O port. This means that the lower half of an address must be latched in T_1 of the machine cycle, so that it is available throughout the machine cycle. The latching of lower half of an address bus is done by using external latch and ALE signal from 8085.

B) $\overline{RD}$ and $\overline{WR}$: These signals are basically used to control the direction of the data flow between processor and memory or I/O device/port. A low on $\overline{RD}$ indicates that the data must be read from the selected memory location or I/O port via data bus. A low on $\overline{WR}$ indicates that the data must be written into the selected memory location or I/O port via data bus.

C) IO/$\overline{M}$, S_0 and S_1 : IO/$\overline{M}$ indicates whether I/O operation or memory operation is being carried out. S_1 and S_0 indicate the type of machine cycle in progress.

D) READY : It is used by the microprocessor to sense whether a peripheral is ready or not for data transfer. If not, the processor waits. It is thus used to synchronize slower peripherals to the microprocessor.

1.3.4 Interrupt Signals

The 8085 has five hardware interrupt signals : RST 5.5, RST 6.5, RST 7.5, TRAP and INTR. The microprocessor recognizes interrupt requests on these lines at the end of the current instruction execution.

The $\overline{\text{INTA}}$ (Interrupt Acknowledge) signal is used to indicate that the processor has acknowledged an INTR interrupt.

1.3.5 Serial I/O Signals

A) SID (Serial I/P Data) : This input signal is used to accept serial data bit by bit from the external device.

B) SOD (Serial O/P Data) : This is an output signal which enables the transmission of serial data bit by bit to the external device.

1.3.6 DMA Signal

A) HOLD : This signal indicates that another master is requesting for the use of address bus, data bus and control bus.

B) HLDA : This active high signal is used to acknowledge HOLD request.

1.3.7 Reset Signals

A) $\overline{\text{RESET IN}}$: A low on this pin

1) Sets the program counter to zero (0000H).
2) Resets the interrupt enable and HLDA flip-flops.
3) Tri-states the data bus, address bus and control bus. (Note : Only during RESET is active).
4) Affects the contents of processor's internal registers randomly.

On reset, the PC sets to 0000H which causes the 8085 to execute the first instruction from address 0000H. For proper reset operation reset signal must be held low for at least 3 clock cycles. The power-on reset circuit can be used to ensure execution of first instruction from address 0000H.

B) RESET OUT : This active high signal indicates that processor is being reset. This signal is synchronized to the processor clock and it can be used to reset other devices connected in the system.

1.4 Bus Organisation

In this section we are going to see how we can use various buses of 8085, how to demultiplex address and data bus, how to generate control signals, how to provide clock and reset signals to 8085 and so on.

1.4.1 Clock Circuits

The 8085 has on chip clock generator. Fig. 1.5 shows the internal block diagram of the on chip clock generator. The internal clock generator requires tuned circuit like LC, RC or crystal, or external clock source as an input to generate the clock. The internal T-flip flop divides the frequency by 2. Hence the operating frequency of the 8085 is always half of the oscillator frequency.

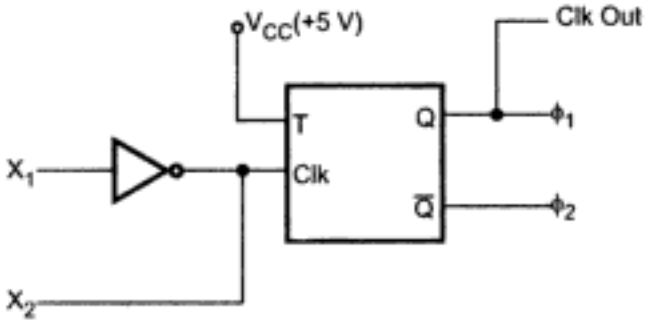


Fig. 1.5 Block diagram of built-in clock generator

LC Tuned Circuit :

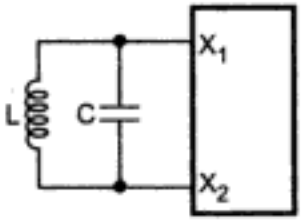


Fig. 1.6 LC circuit

It is a LC resonant tank circuit. The resonant frequency for this circuit is given by

$$f_r = \frac{1}{2\pi\sqrt{L(C_{ext} + C_{int})}}$$

Where C_{int} is the internal capacitance and it is normally 15 pF. The output frequency of this circuit has 10% variations. To minimize the variations in the output frequency, it is recommended to have C_{ext} at least twice that of C_{int} i.e. 30 pF.

RC Tuned Circuit : Fig. 1.7 shows the RC tuned circuit. The output frequency of this circuit is also not exactly stable. But this circuit has an advantage that its component cost is less.

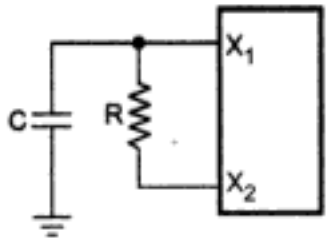


Fig. 1.7 RC Circuit

Crystal Oscillator Circuit : Fig. 1.8 shows the crystal oscillator circuit. It is the most stable circuit. The 20 pF capacitor in the circuit is connected to assure oscillator start-up at the correct frequency.

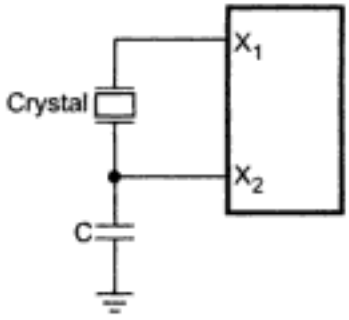


Fig. 1.8 Crystal clock circuit

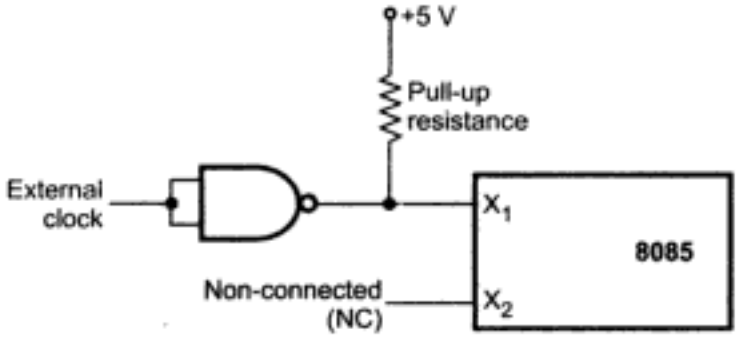


Fig. 1.9 External frequency source

External Clock :

Fig. 1.9 shows how to drive clock input of 8085 with external frequency source. Here external clock is applied at X_1 input and X_2 input is kept open.

1.4.2 Demultiplexing AD_7-AD_0

We know that AD_0 to AD_7 lines are multiplexed and the lower half of address (A_0 - A_7) is available only during T_1 of the machine cycle. This lower half of address is also necessary during T_2 and T_3 of machine cycle to access specific location in memory or I/O port. This means that the lower half of an address bus must be latched in T_1 of the machine cycle, so that it is available throughout the machine cycle. The latching of lower half of an address is done by using external latch and ALE signal from 8085. The Fig. 1.10 shows the hardware connection for latching the lower half of an address. The IC 74LS373 is an 8-bit latch, having 8 D flip-flops. The input is transferred to the output only when clock is high. This clock signal is driven by ALE signal from 8085. The ALE signal is activated only during T_1, so input is transferred to the output only during T_1 i.e. address (A_0 - A_7) on the AD_0 to AD_7 multiplexed bus. In the remaining part of the machine cycle, ALE signal is disabled so output of the latch (A_0 - A_7) remains unchanged. To latch lower half of an address, in each machine cycle, the 8085 gives ALE signal high during T_1 of every machine cycle.

1.4.3 Reset Circuit

On reset, the PC sets to 0000H which causes the 8085 to execute the first instruction from address 0000H. For proper reset operation reset signal must be held low for at least 3 clock cycles. The power-on reset circuit can be used to ensure execution of first instruction from address 0000H. Fig. 1.11 shows the power-on reset circuit with typical R, C values. (Note : R, C values may vary due to power supply ramp up time).

Upon power-up, $\overline{\text{RESET IN}}$ must remain low for at least 10 ms after minimum Vcc has been reached, in the circuit shown in Fig. 1.11. Upon power up or key press, the $\overline{\text{RESET IN}}$ goes low and slowly rises to +5V, providing sufficient time for the processor to reset the system. The diode is connected to discharge the capacitor immediately when power supply is switched OFF.

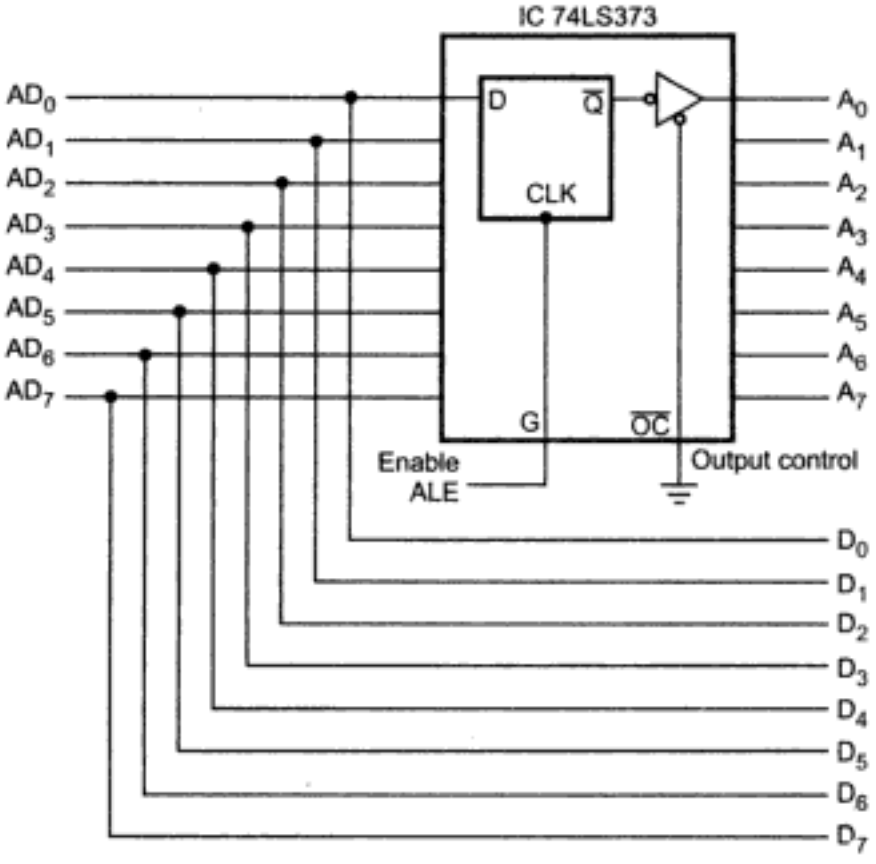


Fig. 1.10 Latching circuit

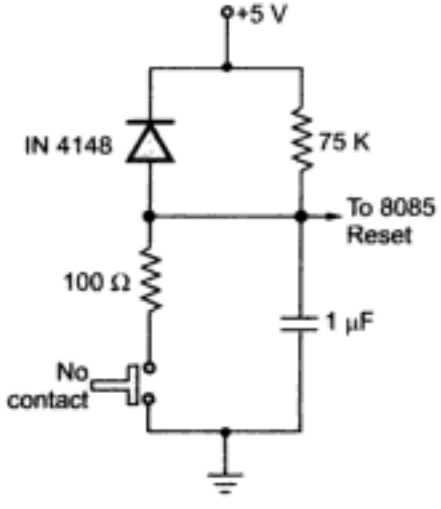


Fig. 1.11 Power on reset

After RESET, 8085 loads 0000H in PC register and clears the INTE flag. Before going to execute interrupt service routine, it is necessary to setup certain parameters, required to execute interrupt service routine. To avoid interrupt to occur before completion of these initial requirements, after power on or reset, INTE flip-flop is cleared to disable interrupts. It can be enabled by EI instruction after initial settings.

As we know that, after power up or reset 8085 fetches its first instruction from 0000H address, and it has to be the first instruction from monitor program. Therefore EPROM consisting of monitor program must be located from address 0000H in any 8085 microprocessor system.

1.4.4 Generation of Control Signals

The 8085 microprocessor provides $\overline{RD}$ and $\overline{WR}$ signals to initiate read or write cycle. Because these signals are used both for reading/writing memory and for reading/writing an input device, it is necessary to generate separate read and write signals for memory and I/O devices.

The 8085 provides IO/$\overline{M}$ signal to indicate whether the initiated cycle is for I/O device or for memory device. Using IO/$\overline{M}$ signal along with $\overline{RD}$ and $\overline{WR}$, it is possible to generate separate four control signals :

$\overline{MEMR}$	**(Memory Read)**	: To read data from memory.
$\overline{MEMW}$	**(Memory Write)**	: To write data in memory.
$\overline{IOR}$	**(I/O Read)**	: To read data from I/O device.
$\overline{IOW}$	**(I/O Write)**	: To write data in I/O device.

Fig. 1.12 shows the circuit which generates $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ and $\overline{IOW}$ signals.

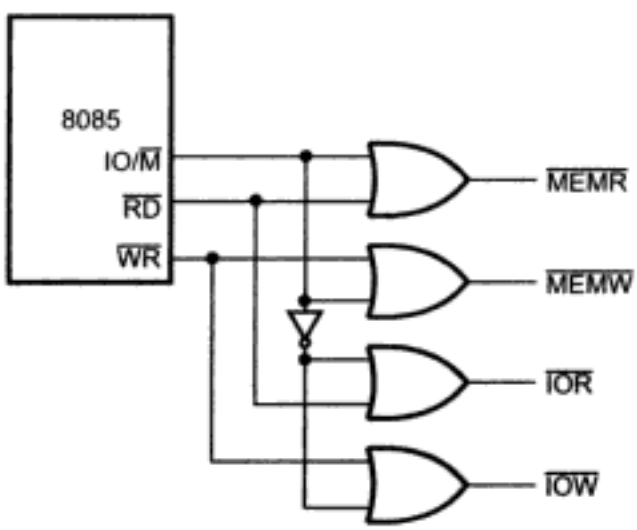


Fig. 1.12 Generation of $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ and $\overline{IOW}$ signals

We know that for OR gate, when both the inputs are low then only output is low. Table 1.1 shows the truth table used to generate $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ and $\overline{IOW}$ signals. The signal IO/$\overline{M}$ goes low for memory operation. This signal is logically ORed with $\overline{RD}$ and $\overline{WR}$ to get $\overline{MEMR}$ and $\overline{MEMW}$ signals. When both $\overline{RD}$ and IO/$\overline{M}$ signals go low, $\overline{MEMR}$ signal goes low. Similarly, when both $\overline{WR}$ and IO/$\overline{M}$ signals go low, $\overline{MEMW}$ signal goes low. To generate $\overline{IOR}$ and $\overline{IOW}$ signals for I/O operation, IO/$\overline{M}$ signal is first inverted and then logically ORed with $\overline{RD}$ and $\overline{WR}$ signals.

$IO/\overline{M}$	$\overline{RD}$	$\overline{WR}$	$\overline{MEMR}$ $\overline{RD} + IO/\overline{M}$	$\overline{MEMW}$ $\overline{WR} + IO/\overline{M}$	$\overline{IOR}$ $\overline{RD} + \overline{IO/M}$	$\overline{IOW}$ $\overline{WR} + \overline{IO/M}$
0	0	0	Condition never exists, because $\overline{RD}$ and $\overline{WR}$ signals does not go low simultaneously			
0	0	1	0	1	1	1
0	1	0	1	0	1	1
0	1	1	1	1	1	1
1	0	0	Condition never exists, because $\overline{RD}$ and $\overline{WR}$ signals does not go low simultaneously			
1	**0**	1	1	1	**0**	1
1	1	**0**	1	1	1	**0**
1	1	1	1	1	1	1

Table 1.1

Same truth table can be implemented using 3:8 decoder as shown in Fig. 1.13.

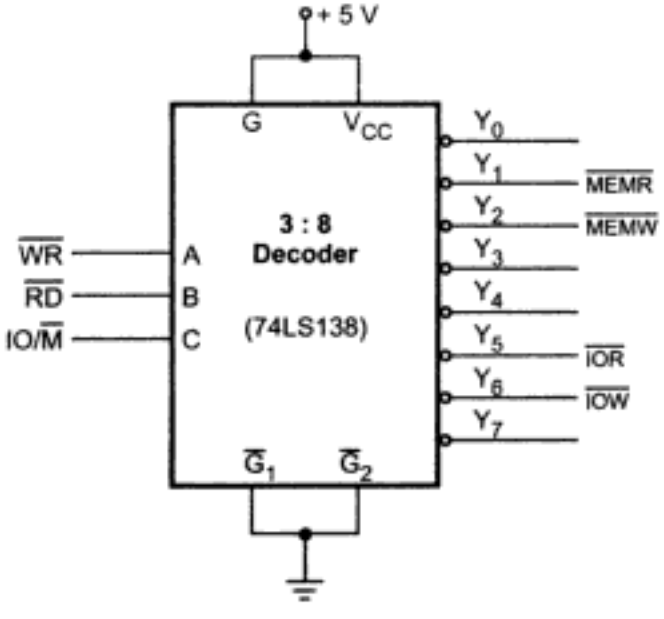


Fig. 1.13 Generation of control signals using 3:8 decoder

1.4.5 Bus Drivers

Typically, the 8085 buses can source 400 μA and sink 2 mA of current, i.e. it can drive only **one** TTL load. Therefore, it is necessary to increase driving capacity of the 8085 buses. Bus drivers, buffers are used to increase the driving capacity of the buses.

Unidirectional Buffers :

As we know, the address bus is unidirectional, 8-bit unidirectional buffer, 74LS244 is used to buffer higher address bus. The Fig. 1.14 shows the logic diagram of 74LS244. It consists of eight non-inverting buffers with tri-state outputs. Each one can sink 24 mA and source 15 mA of current. These buffers are divided into two groups. The enabling and disabling of these groups are controlled by $\overline{1G}$ and $\overline{2G}$ lines.

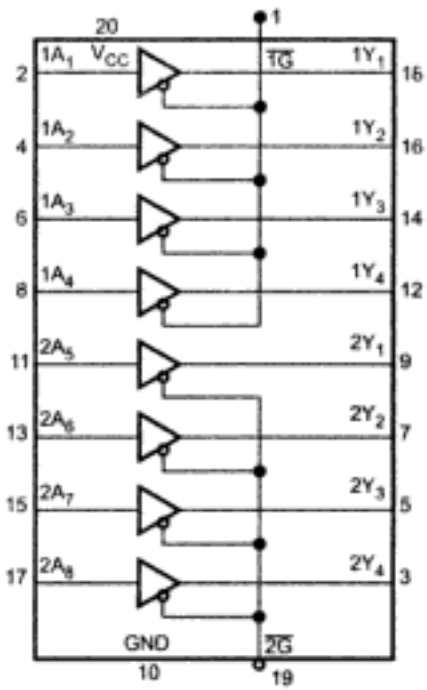
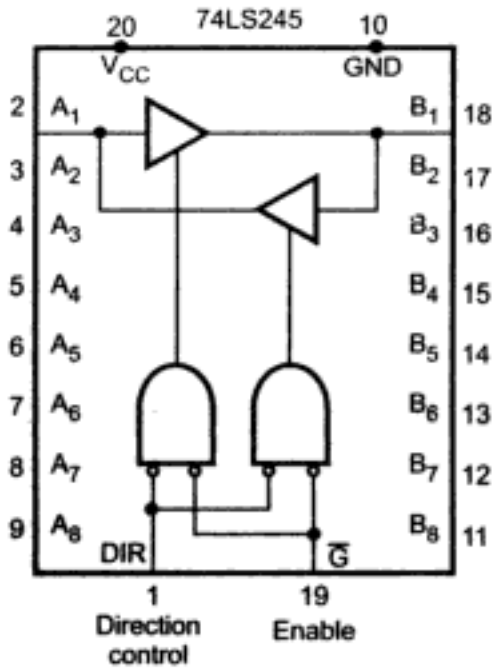


Fig. 1.14 Logic diagram of the 74LS244

Bi-directional Buffer :

To increase the driving capacity of data bus, bi-directional buffer is used. Fig. 1.15 shows the logic diagram of the bi-directional buffer 74LS245, also called an octal bus transceivers. It consists of sixteen non-inverting buffers, eight for each direction, with tri-state output. The direction of data flow is controlled by the pin DIR. When DIR is high, data flows from the A bus to the B bus; when it is low, data flows from B to A. The active low enable signal and the DIR signal are ANDed to activate the bus lines. Each buffer in this device can sink 24 mA and source 15 mA of current.



Function table

Enable $\overline{G}$	Direction control DIR	Operation
L	L	B Data to A Bus
L	H	A Data to B Bus
H	X	Isolation

H=High level,L=Low level,X=Irrelevant

Fig. 1.15 Logic diagram of the 74LS245

1.4.6 Typical Configuration

Fig. 1.16 shows schematic of the 8085 microprocessor demultiplexed address bus and control signals.

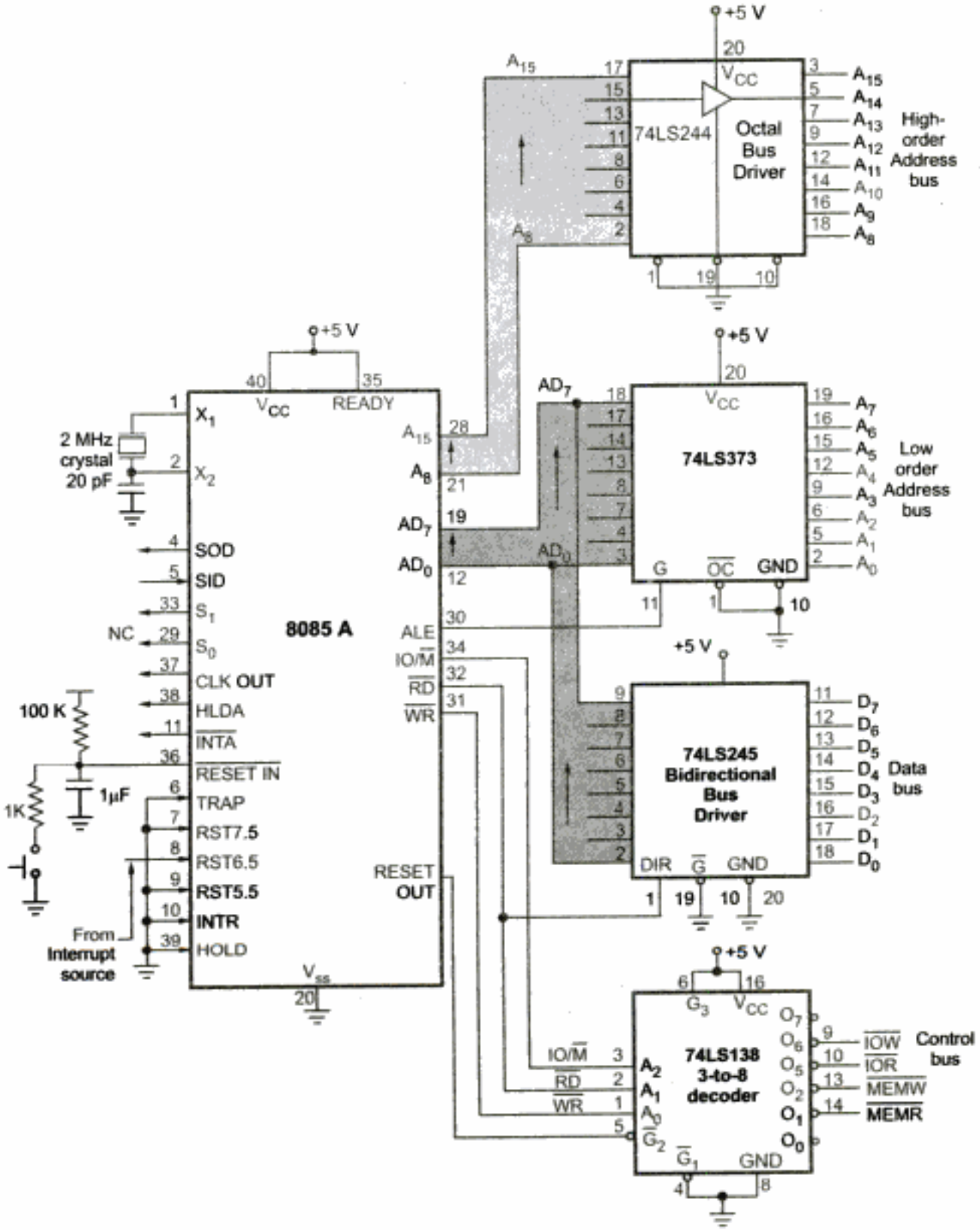


Fig. 1.16 Typical 8085 configuration

It also shows clock and reset circuits. Interrupt lines which are not in use are grounded. This is necessary because floating interrupt line may cause false triggering of interrupt. Similarly, since the DMA controller is not used, HOLD line is also grounded. As we know READY signal is used to synchronize slow peripherals with the microprocessor. When it is low, microprocessor enters in the wait state and when it is high, it indicates that the memory or peripheral is ready to send or receive data. Here, the READY signal is tied high to prevent the microprocessor from entering the wait state. ALE signal is connected to the clock input of the latch, to latch the low order address in T_1 of the machine cycle. To control the direction of the bi-directional buffer 74LS245, $\overline{RD}$ signal from 8085 is connected to DIR input of the bi-directional buffer. Thus, when $\overline{RD}$ signal is low, DIR is low and data flows from memory or I/O device to the microprocessor, performing read operation. When $\overline{RD}$ signal is high, DIR is high and data flows from microprocessor to memory or I/O device performing write operation

1.5 Timing and Control

During normal operation, the microprocessor sequentially fetches, decodes and executes one instruction after another until a halt instruction (HLT) is executed. The fetching, decoding and execution of a single instruction constitutes an **instruction cycle**, which consists of one to five read or write operations between processor and memory or input/output devices. Each memory or I/O operation requires a particular time period, called **machine cycle**. In other words, to move byte of data in or out of the microprocessor, a machine cycle is required. Each machine cycle consists of 3 to 6 clock periods/cycles, referred to as **T-states**. Therefore we can say that, one instruction cycle consists of one to five machine cycles and one machine cycle consists of three to six T-states i.e. three to six clock periods, as shown in the Fig. 1.17.

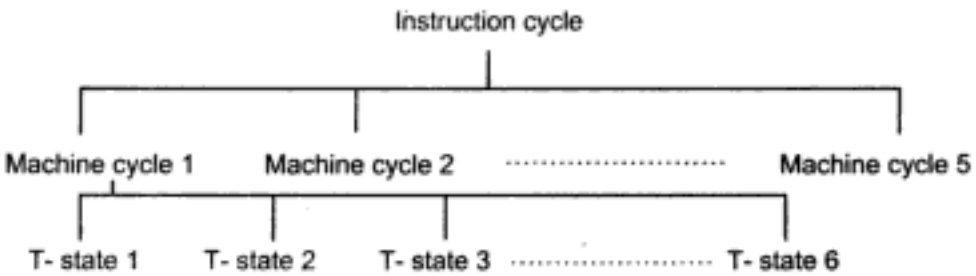


Fig. 1.17 Relation between instruction cycle, machine cycle and T-state

There are seven different types of machine cycles in the 8085A. Three status signals IO/$\overline{M}$, S_1 and S_0 identify each type as shown in Table 1.2. These signals are generated at the beginning of each machine cycle and remained valid for the duration of the cycle.

Machine Cycle	Status			Control		
	IO/$\overline{M}$	S_1	S_0	$\overline{RD}$	$\overline{WR}$	$\overline{INTA}$
Opcode Fetch	0	1	1	0	1	1
Memory Read	0	1	0	0	1	1
Memory Write	0	0	1	1	0	1
I/O Read	1	1	0	0	1	1
I/O Write	1	0	1	1	0	1
INTR Acknowledge	1	1	1	1	1	0
Bus Idle	0	0	0	1	1	1

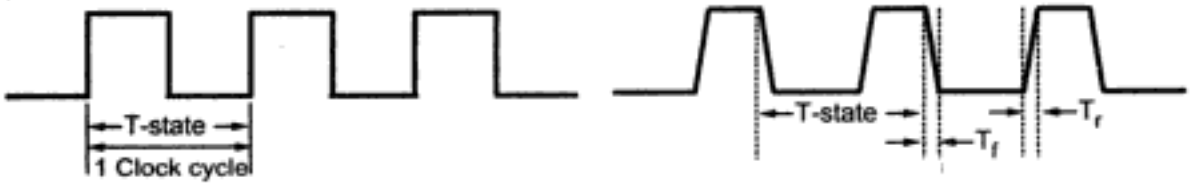
Table 1.2 8085 machine cycles

Representation of Signals

Before going to see the timing diagram, we will see the signals and their representation used in the timing diagrams.

1. Clock Signal :

The 8085 divides the clock frequency provided at X_1 and X_2 inputs by 2, which is called **operating frequency**. All the operations within the 8085 are synchronized with this operating frequency. Therefore in the timing diagram operating frequency clock is shown on the top and then the signals are shown with reference to operating frequency clock. Ideally, the clock signal should be square wave with zero rise time and fall time, as shown in the figure. But in practice, we don't get zero rise time and fall time. Therefore the clock and other signals are always shown with finite rise and fall times. Fig. 1.18 shows the practical way of representing clock signal.



(a) Ideal **(b) Practical**

Fig. 1.18 Clock signal representation

Single Signal :

Single signal is represented by a line. It may have status either logic 0 or logic 1 or tri-state. The change in the state of the signal takes finite time and hence the state change of signal is represented with finite rise time and fall time, as shown in the Fig. 1.19.

Fig. 1.19 Single signal representation

Group of Signals :

Group of signals is also called a bus e.g. address bus and data bus. To avoid complications in the timing diagram these signal are grouped and shown in the form of block as shown in Fig. 1.20.

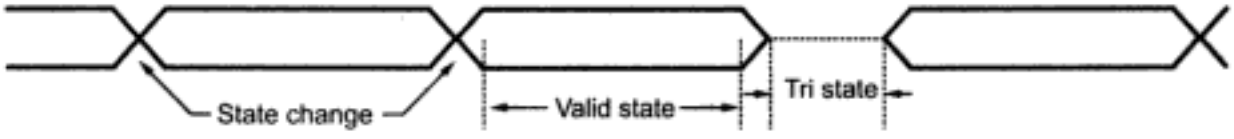


Fig. 1.20 Group of signals representation

In the group representation individual state is not considered, but the group state is considered. Change in state of single signal changes the state of group. It is represented by the cross as shown the Fig. 1.20. The tri-state condition of the group signals is shown by dotted lines. Two straight lines represent valid state/stable state.

In microprocessor systems, activation of signal/signals depends on the state of other signal/signals. Such situations are shown in the timing diagrams with the help of specific symbols. There are four possibilities :

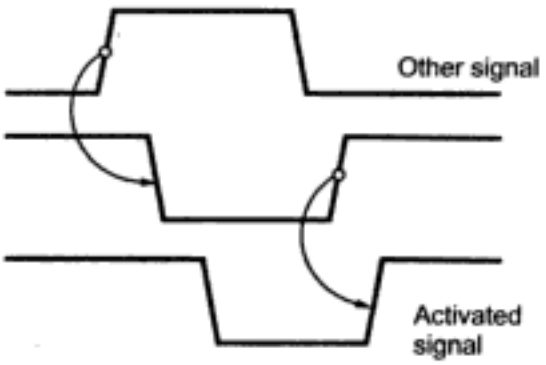
Activation of a signal with the change in state of other signal.

Activation of a signal with the change in state of other signals.

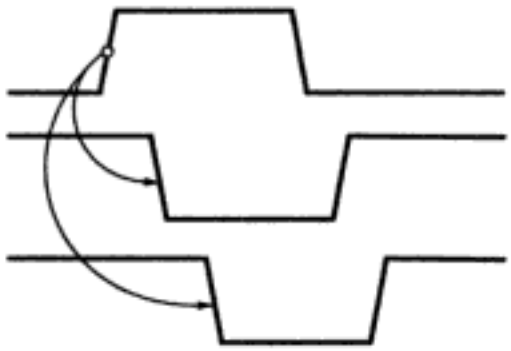
Activation of signals with the change in state of other signal.

Activation of signals with the change in state of other signals.

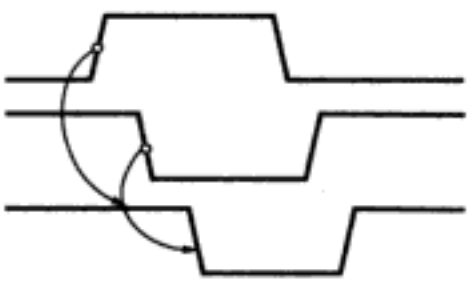
Fig. 1.21 shows the representation of dependence of the signal/signals, in the timing diagram.



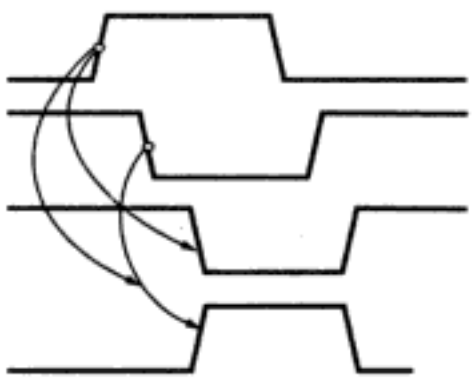
(a) Activation of signal with the change in state of other signal



(b) Activation of signal with the change in state of other signal



(c) Activation of a signal with the change in state of other signals



(d) Activation of signals with the change in state of other signals

Fig. 1.21

Signal Timings

In 8085 microprocessor, signals are activated at specific instant for specific time period. Once we understand this, it is very easy to draw timing diagrams. The following section explains when the signals are activated and for what period they remain in active state.

ALE (Address Latch Enable) :

This signal is active high signal. It is activated in the beginning of the T_1 state of each machine cycle, except bus idle machine cycle, and it remains active in the T_1 state as shown in the Fig. 1.22.

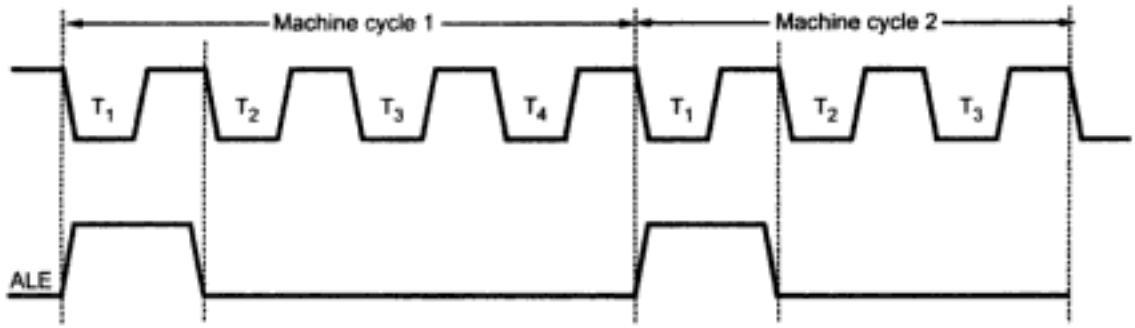


Fig. 1.22 ALE activation and its period

A_0–A_7 (Lower byte address) :

The lower byte of address is available on the multiplexed address/data bus (AD_0–AD_7) during T_1 state of each machine cycle, except bus idle machine cycle, as shown in Fig. 1.23.

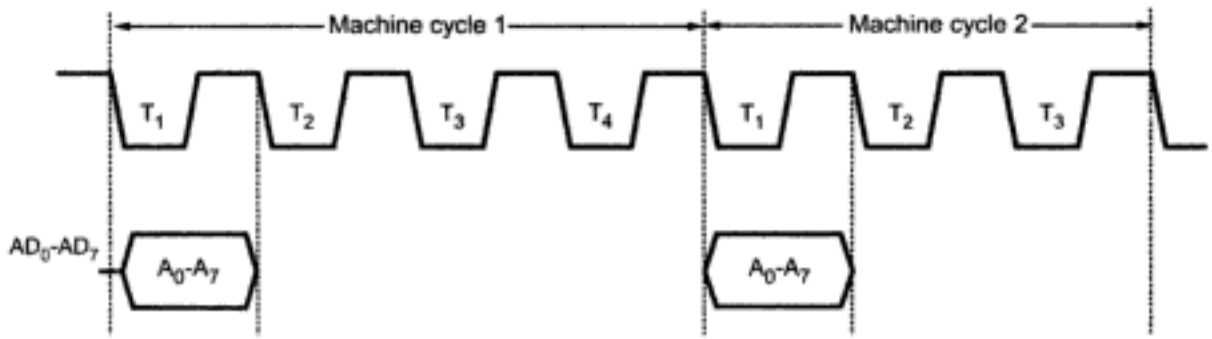
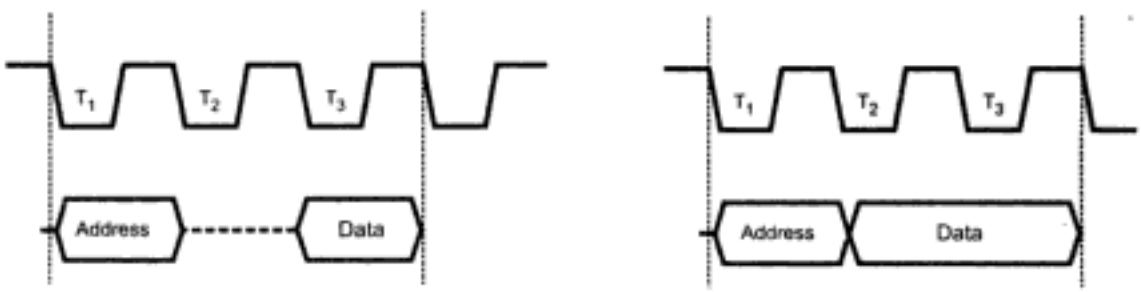


Fig. 1.23 Lower address on the multiplexed bus

D_0–D_7 (Data Bus) :

The data from memory or I/O device and from microprocessor to memory or I/O device is transferred during T_2 and T_3–states. It is important to note that in read machine cycle, data will appear on the data bus during the later part of the T_2–state, as shown in the Fig. 1.24, whereas in write cycle data will appear on the data bus at the beginning of the T_2–state, as shown in the Fig. 1.24.



(a) **Fig. 1.24 Data bus** (b)

To read data from memory or I/O device it is necessary to select memory or I/O device. After selection, device will put the data from selected location on the data bus. This action needs finite time. This time is referred to as **'access time'** . In case of write cycle, data is available in the registers of the microprocessor and it can put that data on the data bus with zero access time.

A_8–A_{15} (Higher byte address) :

The higher byte of address is available on the A_8–A_{15} bus during T_1, T_2 and T_3 – states of each machine cycle, except bus idle machine cycle, as shown in Fig. 1.25.

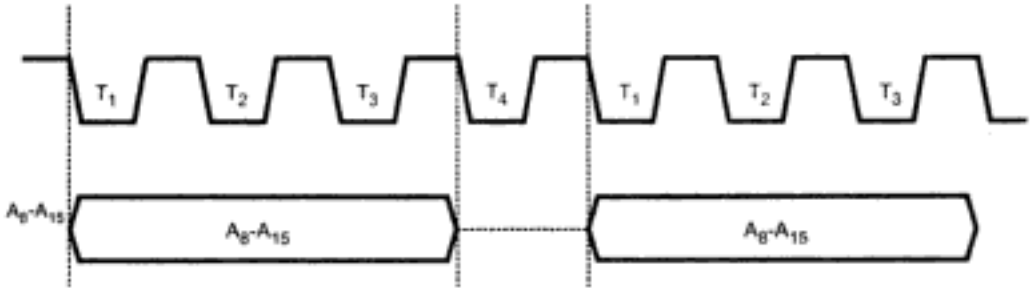


Fig. 1.25 Higher byte address on A8–A15

IO/$\overline{M}$, S_0, S_1 :

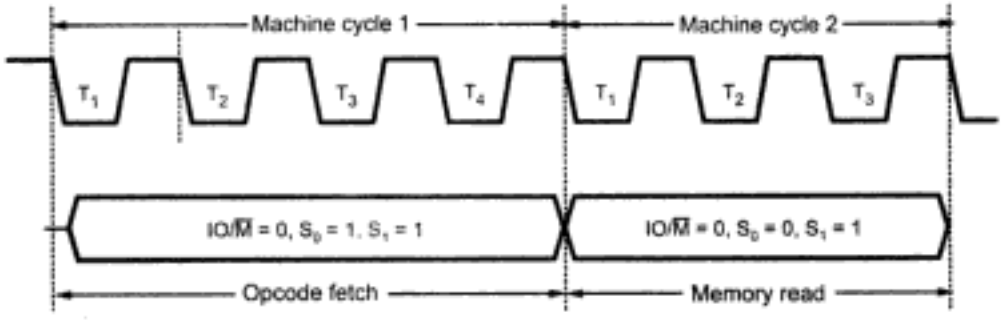


Fig. 1.26 Status signals

These signals are called **status signals**. They decide the type of machine cycle to be executed. They are activated at the beginning of T_1–state of each machine cycle and remain active till the end of the machine cycle.

$\overline{RD}$ and $\overline{WR}$:

These signals decide the direction of the data transfer. When $\overline{RD}$ signal is active, data is transmitted from memory or I/O device to the microprocessor, and when $\overline{WR}$ signal is active, data is transmitted from microprocessor to the memory or I/O device. Both signals are never active at a time.

As we know data transfer in 8085 takes place during T_2 and T_3, these signals are activated during T_2 and T_3, as shown in the Fig. 1.27.

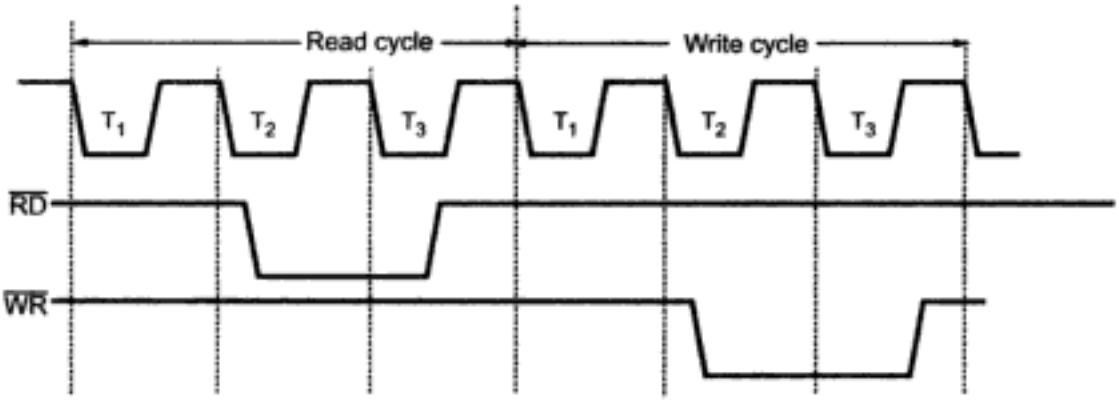


Fig. 1.27 $\overline{RD}$ and $\overline{WR}$ signals

1.5.1 8085 Machine Cycles and their Timings

The 8085 has seven machine cycles. These are :

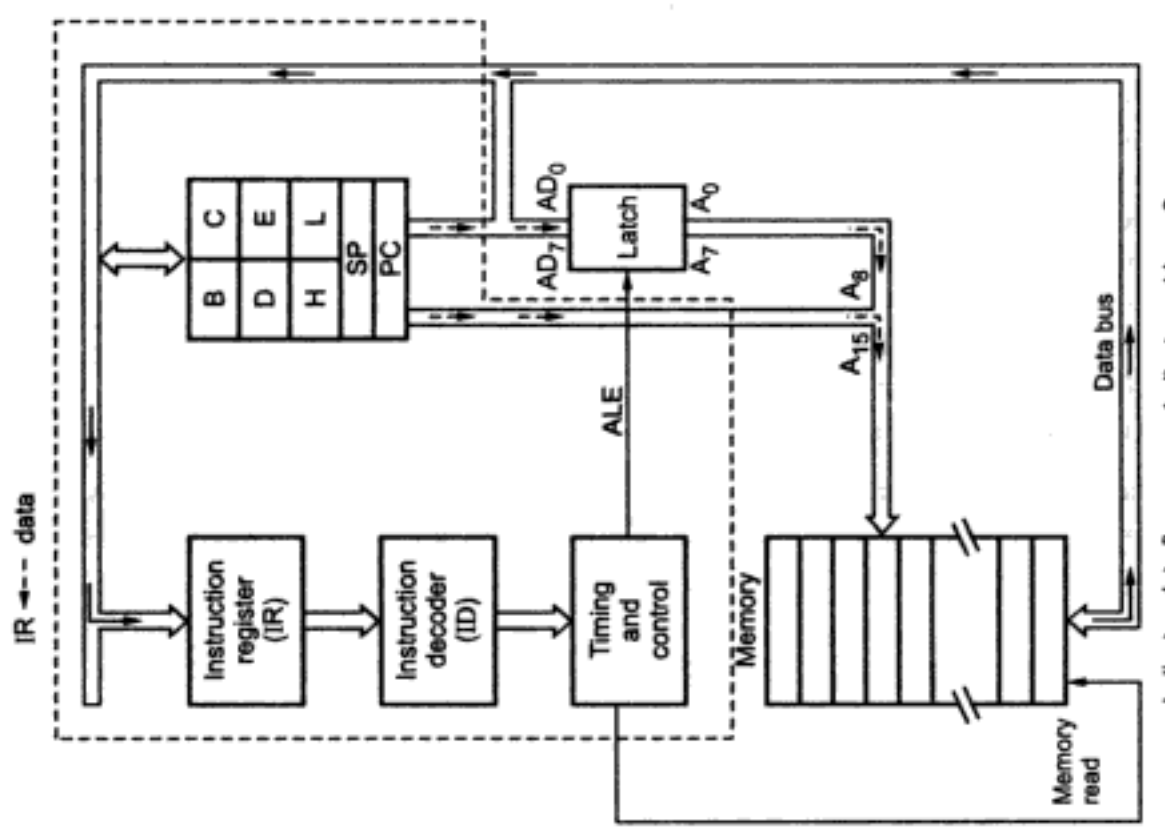
1. Opcode Fetch
2. Memory Read
3. Memory Write
4. I/O Read
5. I/O Write
6. Interrupt Acknowledge
7. Bus Idle

1. Opcode Fetch Cycle :

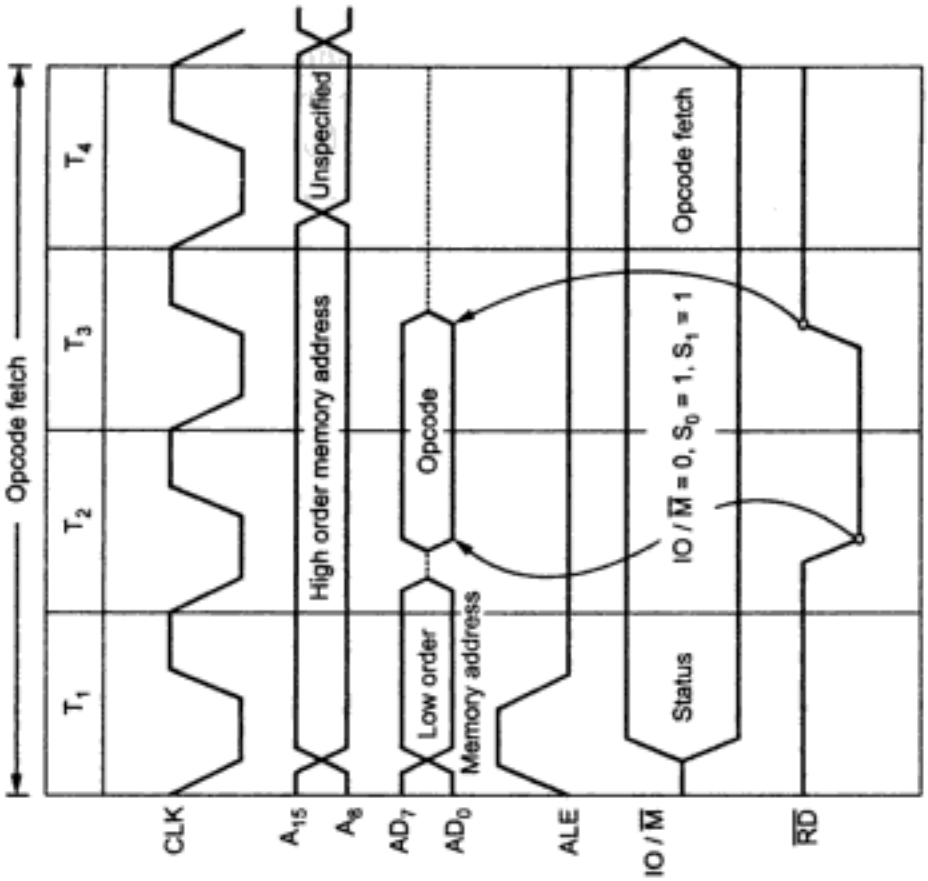
The first machine cycle of every instruction is opcode fetch cycle in which the 8085 finds the nature of the instruction to be executed. In this machine cycle, processor places the contents of the Program Counter on the address lines, and through the read process, reads the opcode of the instruction. Fig. 1.28 (a) (See Fig. on next page) shows flow of data (opcode) from memory to the microprocessor and Fig. 1.28 (b) shows the timing diagram for opcode fetch machine cycle. The length of this cycle is not fixed. It varies from 4T states to 6T states as per the instruction. The following section describes the opcode fetch cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, the 8085 places the contents of program counter on the address bus. The high-order byte of the PC is placed on the A_8-A_{15} lines. The low-order byte of the PC is placed on the AD_0 - AD_7 lines which stays on only during T_1. Thus microprocessor activates ALE (Address Latch Enable) which is used to latch the low-order byte of the address in external latch before it disappears.

In T_1, 8085 also sends status signals IO/$\overline{M}$, S_1, and S_0. IO/$\overline{M}$ specifies whether it is a memory or I/O operation, S_1 status specifies whether it is read/write operation; S_1 and S_0 together indicates read, write, opcode fetch, machine cycle operation, or whether it is in HALT state. In opcode fetch machine cycle status signals are : IO/$\overline{M}$ = 0, S_1 = 1, S_0 = 1.



(a) Data (opcode) flow from memory to micrprocessor



(b) Opcode fetch machine cycle

Fig. 1.28

Step 2 : (State T_2) In T_2, low-order address disappears from the AD_0 - AD_7 lines. (However A_0 - A_7 remain available as they were latched during T_1). In T_2, 8085 sends $\overline{RD}$ signal low to enable the addressed memory location. The memory device then places the contents of addressed memory location on the data bus (AD_0 - AD_7).

Step 3 : (State T_3) During T_3, 8085 loads the data from the data bus in its Instruction Register and raises $\overline{RD}$ to high which disables the memory device.

Step 4 : (State T_4) In T_4, microprocessor decodes the opcode, and on the basis of the instruction received, it decides whether to enter state T_5 or to enter state T_1 of the next machine cycle. One byte instructions those operate on eight bit data (8 bit operand) are executed in T_4.

For example : MOV A, B, ANA D, ADD B, INR L, DCR C, RAL and many more.

Note : For one byte instructions which operate on eight bit data, data is always available in the internal memory of 8085 i.e. registers.

Step 5 : (State T_5 and T_6)

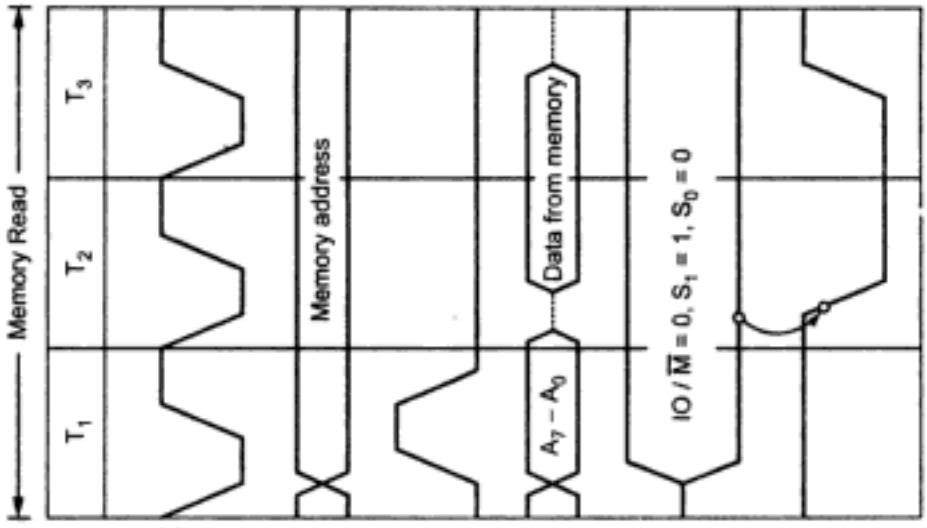
State T_5 and T_6, when entered, are used for internal microprocessor operations required by the instruction. During T_5 and T_6, 8085 performs stack write, internal 16 bit, and conditional return operations depending upon the type of instruction. One byte instructions those operate on sixteen bit data (16 bit operand) are executed in T_5 and T_6. For example DCX H, PCHL, SPHL, INX H, etc.

2. Memory Read Cycle :

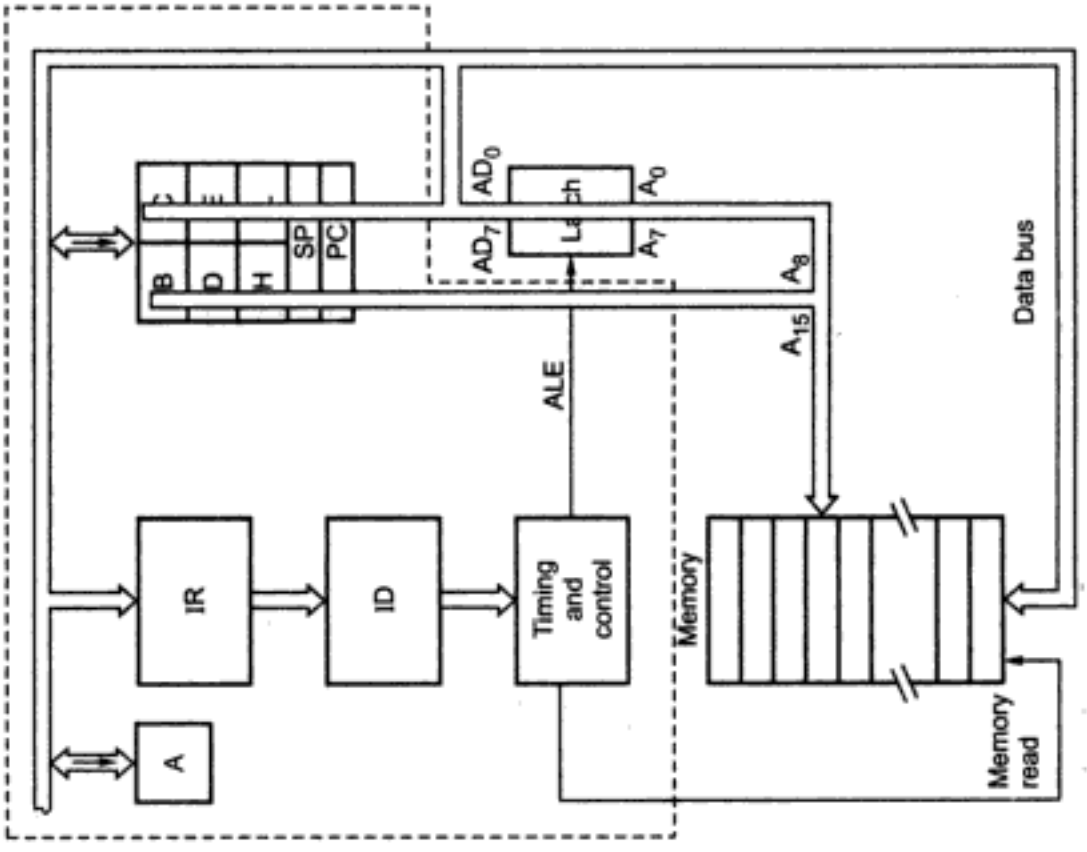
The 8085 executes the memory read cycle to read the contents of R/W memory or ROM. The length of this machine cycle is 3-T states (T_1 - T_3). In this machine cycle, processor places the address on the address lines from the stack pointer, general purpose register pair or program counter, and through the read process, reads the data from the addressed memory location. Fig. 1.29 (a) (See Fig. on next page) shows flow of data from memory to the microprocessor and Fig. 1.29 (b) shows the timing diagram for memory read machine cycle. Memory read machine cycle is similar to the opcode fetch machine cycle. However, they use only states T_1 to T_3, and the status signal values ($IO/\overline{M}$ = 0, S_1 = 1, S_0 = 0) appropriate for memory read machine cycle are issued in T_1. The following section describes the memory read machine cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, microprocessor places the address on the address lines from stack pointer, general purpose register pair or program counter and activates ALE signal in order to latch low-order byte of address.

During T_1, 8085 sends status signals : $IO/\overline{M}$ = 0, S_1 = 1, and S_0 = 0 for memory read machine cycle.

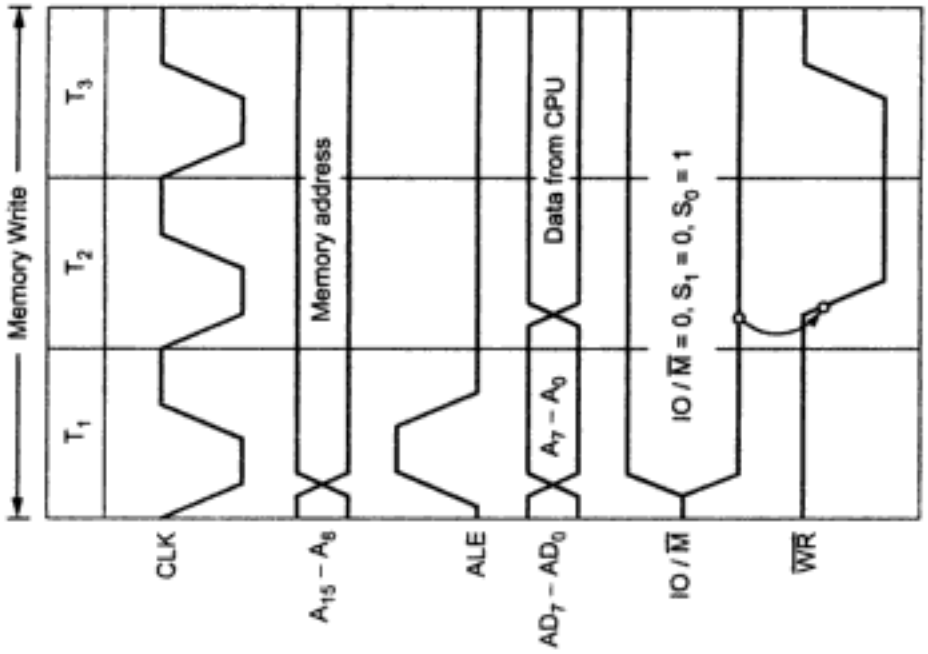


(b) Memory read machine cycle

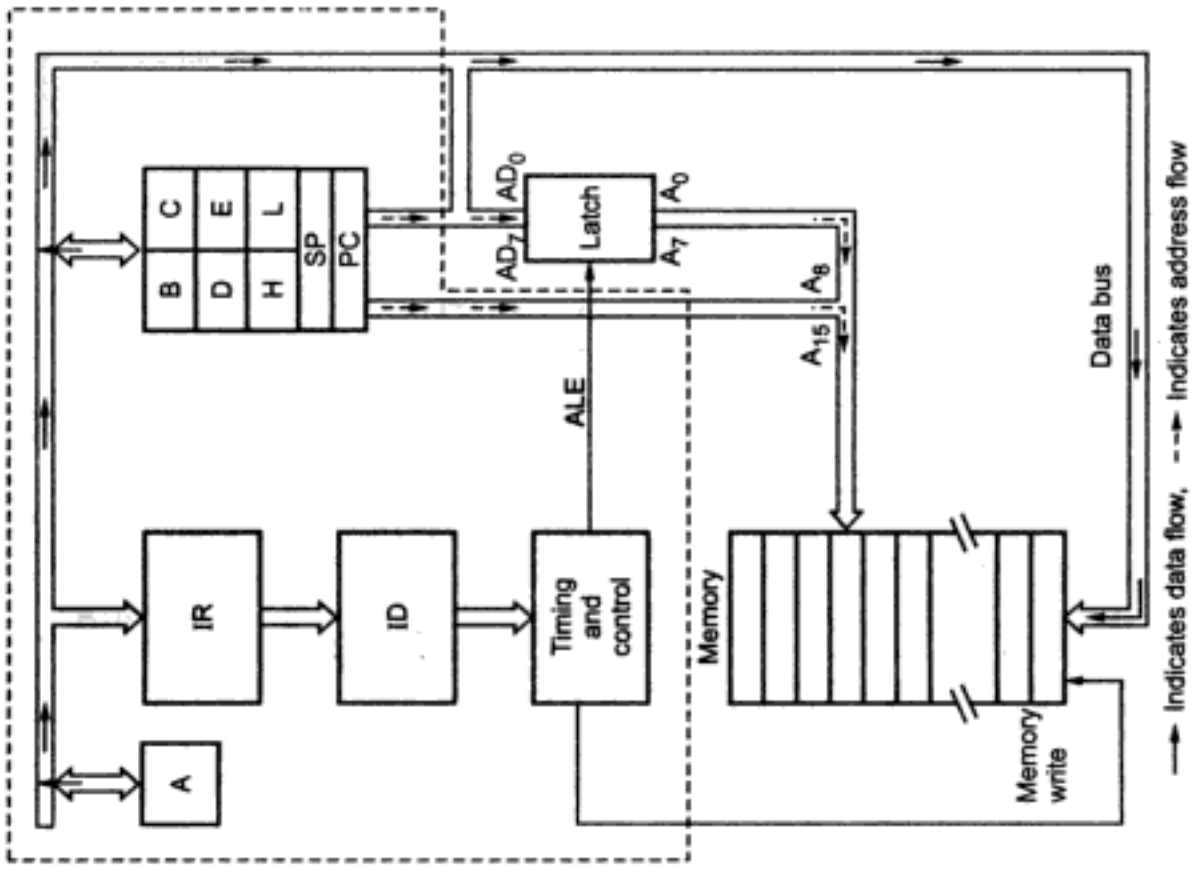


(a) Data flow from memory to microprocessor

Fig. 1.29



(b) Memory write machine cycle



(a) Data flow microprocessor to memory

Fig. 1.30

Step 2 : (State T_2) In T_2, 8085 sends $\overline{RD}$ signal low to enable the addressed memory location. The memory device then places the contents of addressed memory location on the data bus (AD_0 -AD_7).

Step 3 : (State T_3) During T_3, 8085 loads the data from the data bus into specified register (F, A, B, C, D, E, H, and L) and raises $\overline{RD}$ to high which disables the memory device.

3. Memory Write Cycle

The 8085 executes the memory write cycle to store the data into data memory or stack memory. The length of this machine cycle is 3T states (T_1 – T_3). In this machine cycle, processor places the address on the address lines from the stack pointer or general purpose register pair and through the write process, stores the data into the addressed memory location. Fig. 1.30 (See Fig. on previous page) shows the timing diagram for memory write machine cycle. The memory write timing diagram is similar to the memory read timing diagram, except that instead of $\overline{RD}$, $\overline{WR}$ signal goes low during T_2 and T_3. The status signals for memory write cycle are : $IO/\overline{M}$ = 0, S_1 = 0, S_0 = 1. The following section describes the memory write machine cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, the 8085 places the address on the address lines from stack pointer or general purpose register pair and activates ALE signal in order to latch low-order byte of address. During T_1, 8085 sends status signals :

$IO/\overline{M}$ = 0, S_1 = 0 and S_0 = 1 for memory write machir.e cycle.

Step 2 : (State T_2) In T_2, 8085 places data on the data bus and sends $\overline{WR}$ signal low for writing into the addressed memory location.

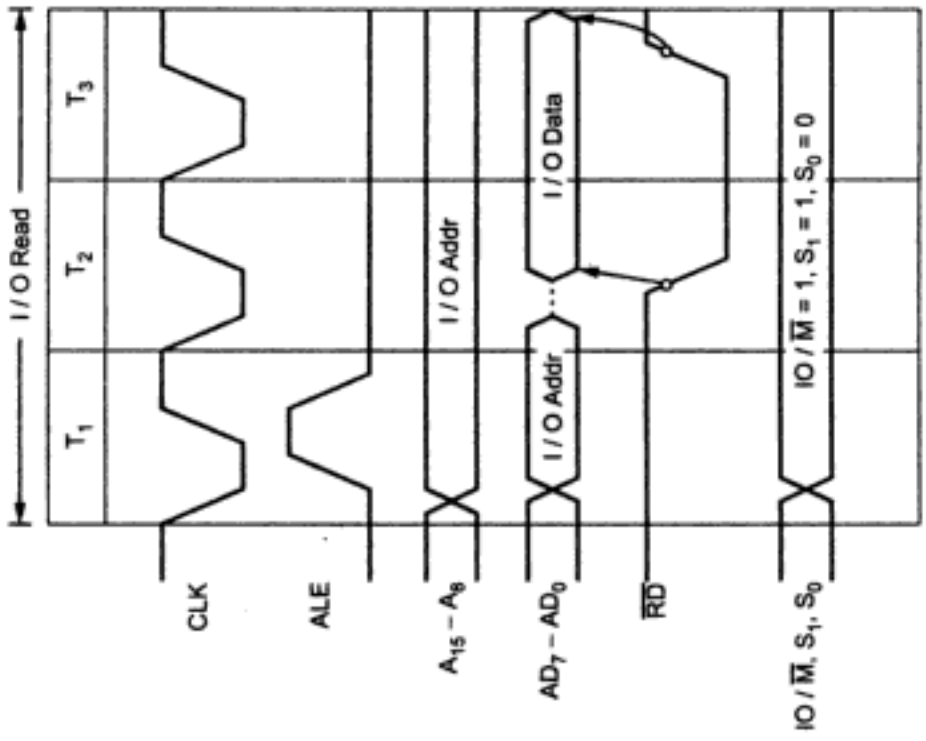
Step 3 : (State T_3) During T_3, $\overline{WR}$ signal goes high, which disables the memory device and terminates the write operation.

4, 5. I/O Read and I/O Write Cycles

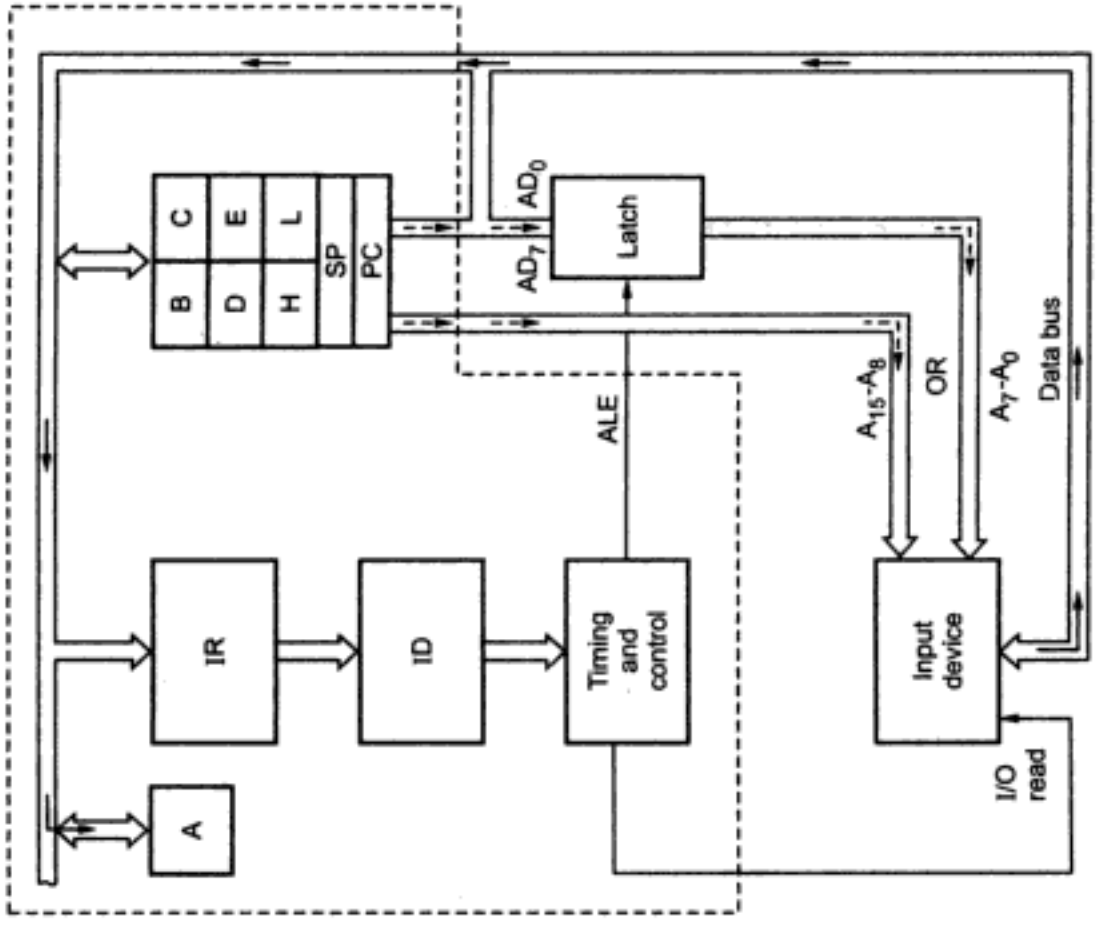
The I/O read and I/O write machine cycles are similar to the memory read and memory write machine cycles, respectively, except that the $IO/\overline{M}$ signal is high for I/O read and I/O write machine cycles. High $IO/\overline{M}$ signal indicates that it is an I/O operation. Fig. 1.31 (b) and Fig. 1.32 (b) show the timing diagrams for I/O read and I/O write cycles, respectively.

6. Interrupt Acknowledge Cycle

In response to INTR signal, 8085 executes interrupt acknowledge machine cycle to read an instruction from the external device. Theoretically, the external device can place any instruction on the data bus in response to $\overline{INTA}$. However, only RST and CALL, save the PC contents (return address) before transferring control to the interrupt service routine. The next sections explain interrupt acknowledge cycles for RST and CALL instructions.

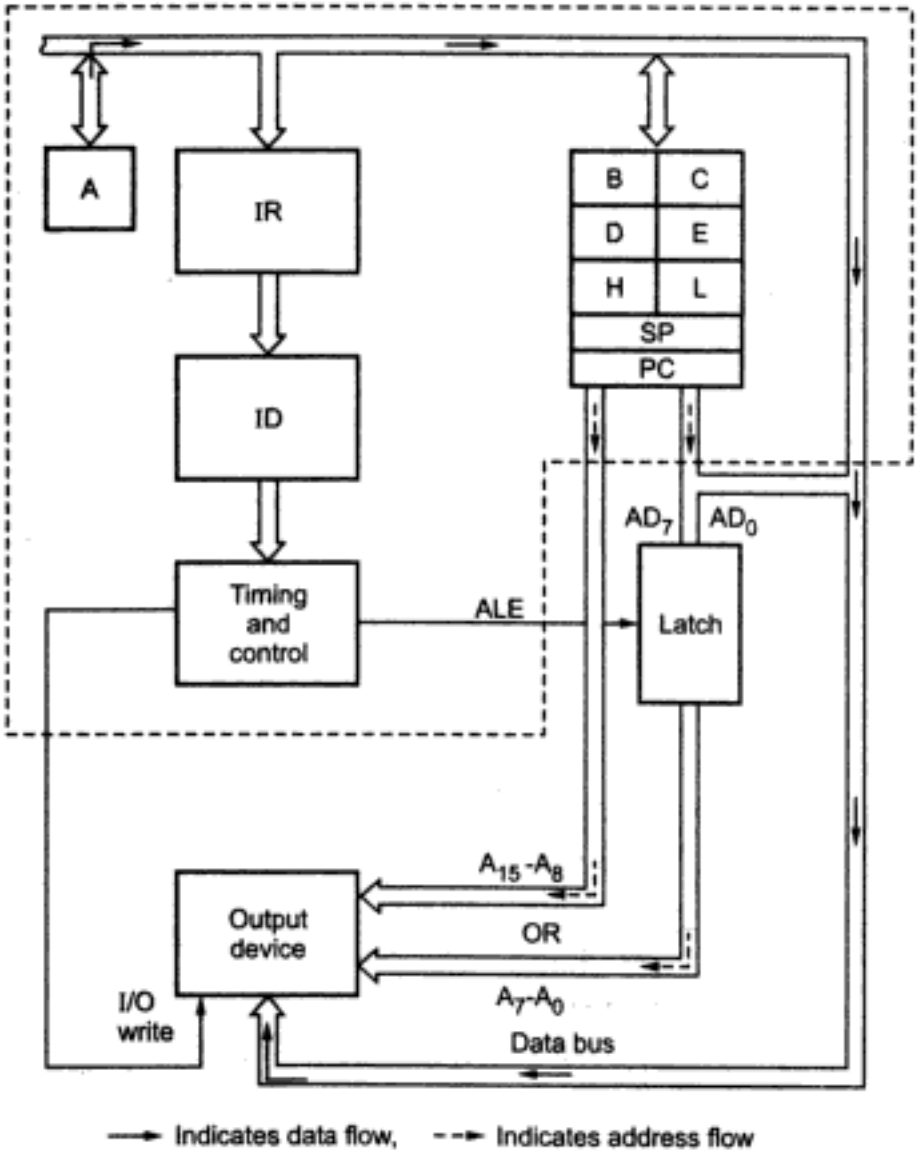


(b) I/O read memory cycle

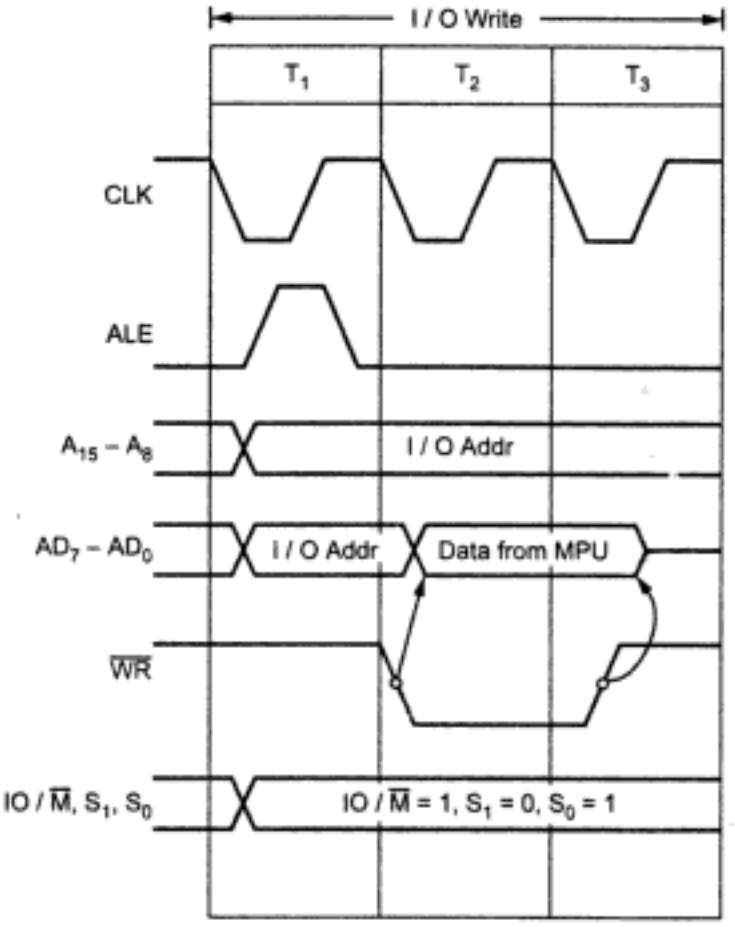


(a) Data flow from input device to microprocessor

Fig. 1.31



(a) Data flow from microprocessor to output device



(b) I/O write machine cycle

Fig. 1.32

Interrupt acknowledge cycle for RST instruction

Fig. 1.33 shows the timing diagram of the interrupt acknowledge machine cycle and execution of RST instruction. The interrupt acknowledge cycle is similar to the opcode fetch cycle, with two exceptions.

1. The $\overline{INTA}$ signal is activated instead of the $\overline{RD}$ signal.
2. The status lines (IO/$\overline{M}$, S_0 and S_1) are 111 instead of 011.

During interrupt acknowledge machine cycle (M_1), the RST is decoded, which initiates 1 byte CALL instruction to the specific vector location. The machine cycles M_2 and M_3 are memory write cycles that store the contents of the program counter on the stack, and then a new instruction cycle begins.

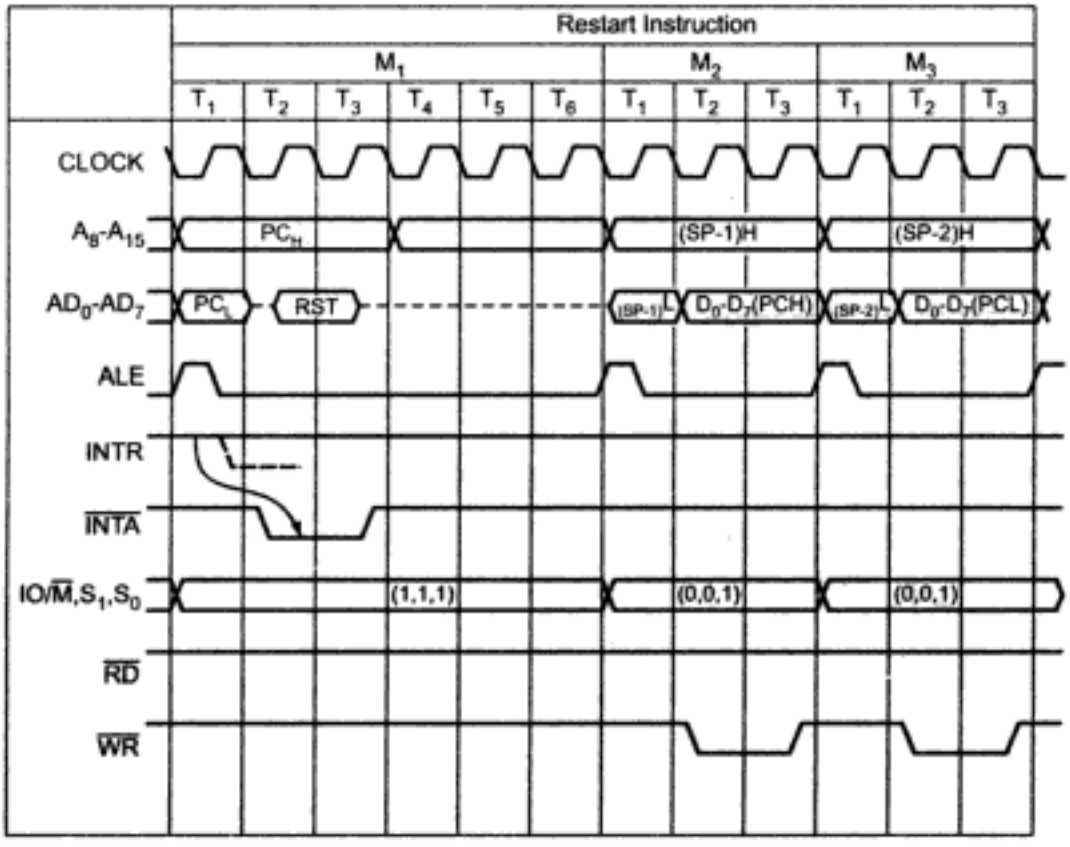


Fig. 1.33 Restart instruction

Interrupt acknowledge cycle for CALL instruction

Fig. 1.34 shows the timing diagram of the interrupt acknowledge machine cycle and execution of a CALL instruction. For CALL instruction, it is necessary to fetch the two bytes of the CALL address through two additional interrupt acknowledge machine cycles (M_2 and M_3 in the 3.21). The machine cycles M_4 and M_5 are memory write cycles that store the contents of the program counter on the stack, and then a new instruction cycle begins.

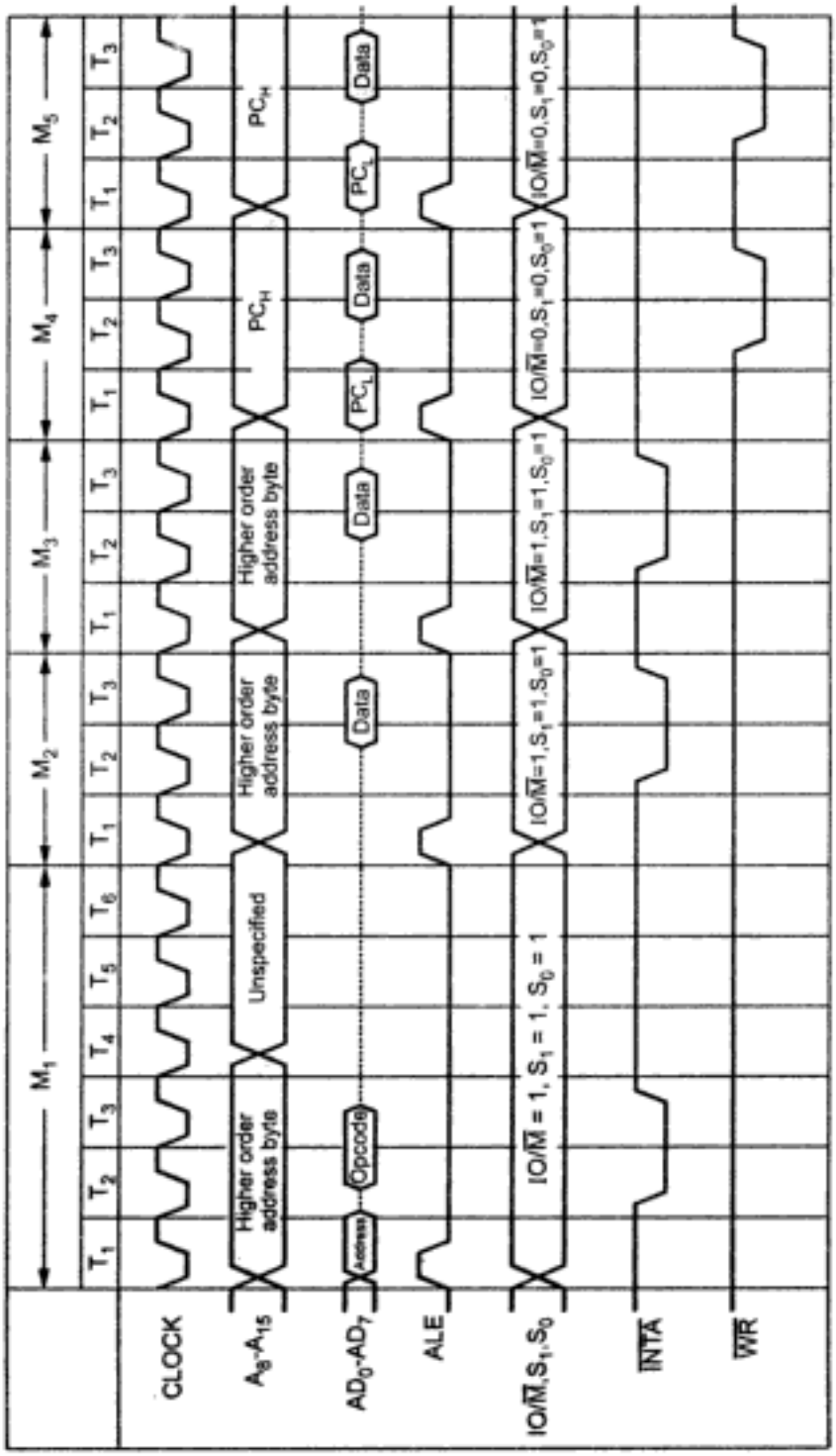


Fig. 1.34 Timing diagram of $\overline{INTA}$ machine cycle and execution of call instruction

7. Bus Idle Cycle

There are few situations where the machine cycles are neither Read nor Write. These situations are :

1. For execution of DAD instruction (this instruction adds the contents of a specified register pair to the contents of HL register pair) ten T states are required. This means that after execution of opcode fetch machine cycle, DAD instruction requires 6 extra T-states to add 16 bit contents of a specified register pair to the contents of HL register pair. These extra T-states which are divided into two machine cycles do not involve any memory or I/O operation. These machine cycles are called BUS IDLE machine cycles. Fig. 1.35 shows Bus Idle Machine Cycle for DAD instruction.

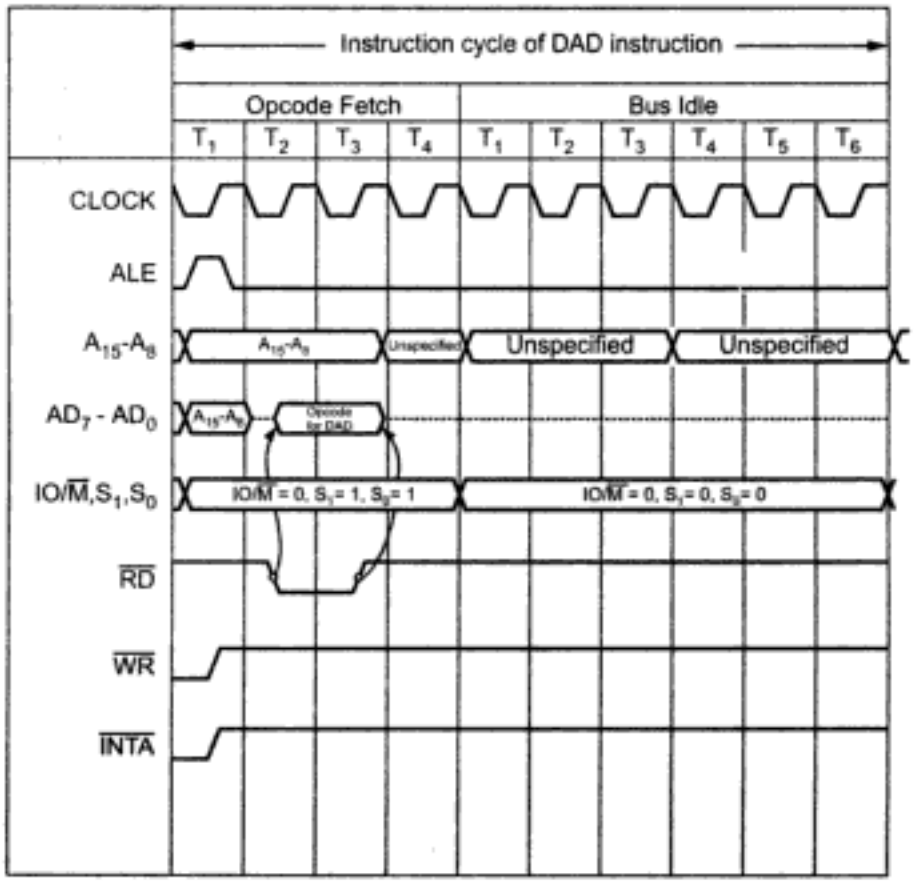


Fig. 1.35 Timing diagram for DAD instruction

In the case of DAD, these Bus Idle cycles are similar to memory read cycles, except $\overline{RD}$ and ALE signals are not activated.

2. During internal opcode generations, for TRAP and RST interrupts, 8085 executes Bus Idle Machine Cycles. Fig. 1.36 shows the Bus Idle Machine Cycle for TRAP. In response to TRAP interrupt, 8085 enters into a Bus Idle Machine Cycle during which it invokes restart instruction, stores the contents of PC onto the stack and places 0024H (Vector address of TRAP) onto the program counter.

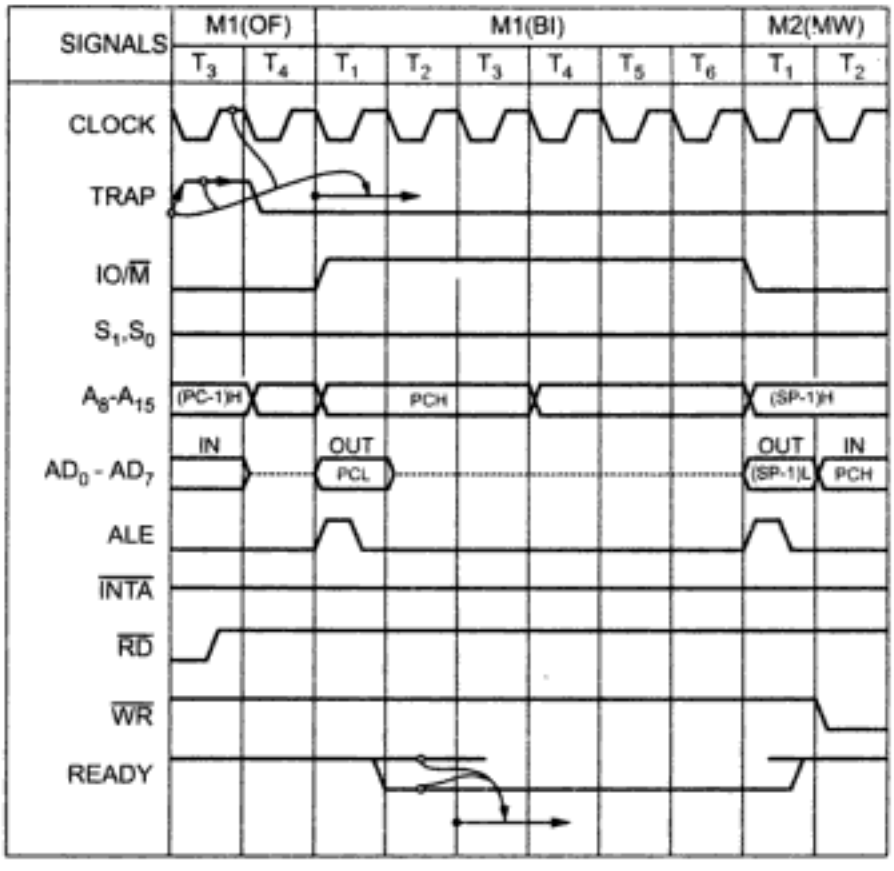


Fig. 1.36 Bus idle machine cycle for trap

The number of machine cycles required to fetch complete instruction depends on the instruction type :

1. One byte 2. Two byte or 3. Three byte

One byte instruction doesn't require any additional machine cycle. Two byte instruction requires one additional memory read machine cycle, whereas three byte instruction requires two additional memory read machine cycles.

The number of machine cycles required to execute the instruction depends on the particular instruction. The total number of machine cycles required varies from one to five. It is possible that memory read and memory write machine cycles occur more than once in a single instruction cycle. The following examples illustrate the timing diagrams and machine cycles used for few 8085 instructions.

1.5.2 Concept of Wait States

In some applications, speed of memory system and I/O system are not compatible with the microprocessor's timings. This means that they take longer time to read/write data. In such situations, the microprocessor has to confirm whether a peripheral is ready to

transfer data or not. If READY pin is high, the peripheral is ready otherwise 8085 enters wait state.

Fig. 1.37 shows the timing diagram for memory read machine cycle with and without wait state.

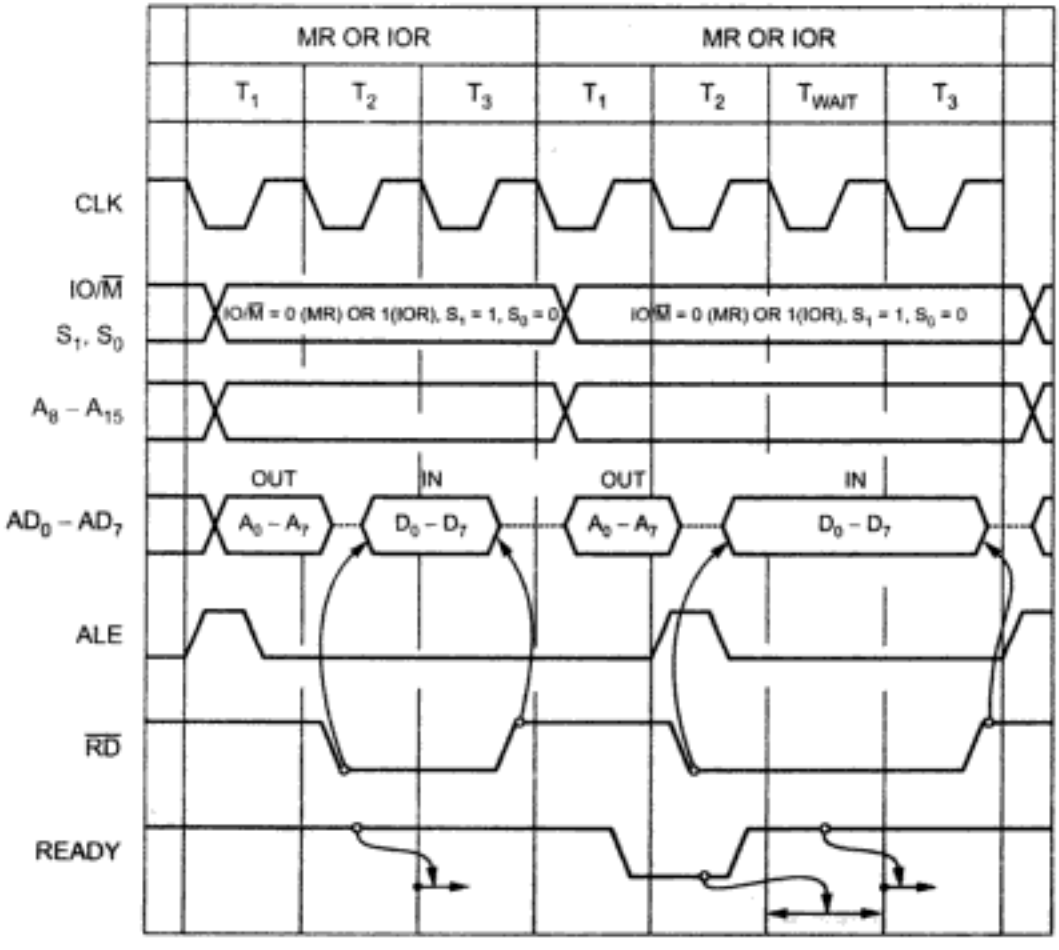


Fig. 1.37 Read machine cycle with and without wait state

Wait states continue to be inserted as long as READY is low. After the wait state, 8085 continues with T_3 of the machine cycle. During a wait state the contents of the address bus, the data bus, and the control bus are all held constant.

The wait state then gives an addressed memory or I/O port an extra clock cycle time to output valid data on the data bus. This feature allows to use cheaper memory or I/O devices that have longer access times.

1.6 Instruction Set of 8085

1.6.1 Data Transfer Group

1. MVI r, data (8) This instruction directly loads a specified register with an 8-bit data given within the instruction. The register r is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Example :

```
MVI  B, 60H     ; This instruction will load 60H directly into the B register.
```

2. MVI M, data (8) This instruction directly loads an 8-bit data given within the instruction into a memory location. The memory location is specified by the contents of HL register pair.

Example : H = 20H and L = 50H

```
MVI M, 40H      ; This instruction will load 40H into
                ; memory whose address is 2050H.
```

3. MOV rd, rs This instruction copies data from the source register into destination register. The rs and rd are general purpose registers such as A, B, C, D, E, H and L. The contents of the source register remain unchanged after execution of the instruction.

Example : A = 20H

```
MOV B, A        ; This instruction will copy the contents
                ; of register A (20H) into register B.
```

4. MOV M, rs This instruction copies data from the source register into memory location pointed by the HL register pair. The rs is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : If HL = 2050H, B = 30H.

```
MOV M, B        ; This instruction will copy the contents
                ; of B register (30H) into the memory location
                ; whose address is specified by HL (2050H).
```

5. MOV rd, M This instruction copies data from memory location whose address is specified by HL register pair into destination register. The contents of the memory location remain unchanged. The rd is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : HL = 2050H, contents at 2050H memory location = 40H

```
MOV C, M        ; This instruction will copy the contents
                ; of memory location pointed by HL
                ; register pair (40H) into the C register.
```

6. LXI rp, data (16) This instruction loads immediate 16 bit data specified within the instruction into register pair or stack pointer. The rp is 16-bit register pair such as BC, DE, HL or 16-bit stack pointer.

Example :

```
i. LXI B, 1020H    ; This instruction will load 10H into B
                   ; register and 20H into C register.
```

7. STA addr This instruction stores the contents of A register into the memory location whose address is directly specified within the instruction. The contents of A register remain unchanged.

Example : A = 50H

```
STA 2000H          ; This instruction will store the
                   ; contents of A register (50H) to
                   ; memory location 2000H.
```

8. LDA addr This instruction copies the contents of the memory location whose address is given within the instruction into the accumulator. The contents of the memory location remain unchanged.

Example : (2000H) = 30H

```
LDA 2000H          ; This instruction will copy the
                   ; contents of memory location
                   ; 2000H i.e. data 30H into the
                   ; A register
```

9. SHLD addr This instruction stores the contents of L register in the memory location given within the instruction and contents of H register at address next to it. This instruction is used to store the contents of H and L registers directly into the memory. The contents of the H and L registers remain unchanged.

Example : H = 30H, L = 60H

```
SHLD 2500H         ; This instruction will copy
                   ; the contents of L register at
                   ; address 2500H and the contents
                   ; of H register at address 2501H.
```

10. LHLD addr This instruction copies the contents of the memory location given within the instruction into the L register and the contents of the next memory location into the H register.

Example : (2500H) = 30H, (2501H) = 60H

```
LHLD 2500H    ; This instruction will copy the
              ; contents of memory location 2500H
              ; i.e. data 30H into the L register and
              ; the contents at memory location
              ; 2501H i.e. data 60H into the H register.
```

11. STAX rp This instruction copies the contents of accumulator into the memory location whose address is specified by the specified register pair. The rp is BC or DE register pair. This register pair is used as a memory pointer. The contents of the accumulator remain unchanged.

Example : BC = 1020H, A = 50H

```
STAX B    ; This instruction will copy the
          ; contents of A register (50H) to the
          ; memory location specified
          ; by BC register pair (1020H).
```

12. LDAX rp This instruction copies the contents of memory location whose address is specified by the register pair into the accumulator. The rp is BC or DE register pair. The register pair is used as a memory pointer.

Example : DE = 2030H, (2030H) = 80H

```
LDAX D    ; This instruction will copy the
          ; contents of memory location
          ; specified by DE register pair
          ; (2030H) into the accumulator.
```

13. XCHG This instruction exchanges the contents of the register H with that of D and of L with that of E.

Example : DE = 2040H, HL = 7080H

```
XCHG    ; This instruction will load the data into registers as follows
        ; H = 20H, L = 40H, D = 70H and E = 80
```

1.6.2 Arithmetic Group

1. ADD r This instruction adds the contents of the specified register to the contents of accumulator and stores result in the accumulator. The r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : A = 20H, C = 30H.

```
ADD C   ; This instruction will add the contents of C register, i.e. data
        ; 30H to the contents of accumulator, i.e. data 20H and it will
        ; store the result 50H in the accumulator.
```

2. ADD M This instruction adds the contents of the memory location pointed by HL register pair to the contents of accumulator and stores result in the accumulator. The HL register pair is used as a memory pointer. This instruction affects all flags.

Example : A = 20H, HL = 2050H,

; (2050H) = 10H

```
ADD M   ; This instruction will add the contents of memory location
        ; pointed by HL register pair, 2050H i.e. data 10H to the
        ; contents of accumulator i.e. data 20H and it will store the
        ; result, 30H in the accumulator.
```

3. ADI data (8) This instruction adds the 8 bit data given within the instruction to the contents of accumulator and stores the result in the accumulator.

Example : A = 50H

```
ADI 70H ; This instruction will add 70H to the contents of the
        ; accumulator (50H) and it will store the result in the
        ; accumulator (C0H).
```

4. ADC r This instruction adds the contents of specified register to the contents of accumulator with carry. This means, if the carry flag is set by some previous operation, it adds 1 and the contents of the specified register to the contents of accumulator, else it adds the contents of the specified register only. The r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : Carry flag = 1, A = 50H, C = 20H

```
ADC C   ; This instruction will add the contents of C (20H) register to
        ; the contents of accumulator (50H) with carry (1) and
        ; it will store result, 71H (50H + 20H + 1 = 71H) in the
        ; accumulator
```

5. ADC M This instruction adds the contents of memory location pointed by HL register pair to the contents of accumulator with carry and stores the result in the accumulator. HL register pair is used as a memory pointer.

Example : Carry flag = 1, HL = 2050H, A = 20H, (2050H) = 30H.

```
ADC M   ; This instruction will add the contents of memory location
        ; pointed by HL register pair, 2050H, i.e. data 30H to the
        ; contents of accumulator, i.e. data 20H with carry flag (1).
        ; It will store the result (30 + 20 + 1 = 51H) in the accumulator.
```

6. ACI data (8) This instruction adds 8 bit data given within the instruction to the contents of accumulator with carry and stores result in the accumulator.

Example : A = 30H, Carry flag = 1

```
ACI 20H ; This instruction will add 20H to the contents of accumulator,
        ; i.e. data 30H with carry (1) and stores the result,
        ; 51H (30 + 20 + 1 = 51H) in the accumulator.
```

7. DAD rp This instruction adds the contents of the specified register pair to the contents of the HL register pair and stores the result in the HL register pair. The rp is 16-bit register pair such as BC, DE, HL or stack pointer. Only higher order register is to be specified for register pair within the instruction.

Example : DE = 1020H, HL = 2050H

```
DAD D   ; This instruction will add the contents of DE register pair,
        ; 1020H to the contents of HL register pair, 2050H.
        ; It will store the result, 3070H in the HL register pair.
```

8. SUB r This instruction subtracts the contents of the specified register from the contents of the accumulator and stores the result in the accumulator. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : A = 50H, B = 30H.

```
SUB B   ; This instruction will subtract the contents of B register (30H)
        ; from the contents of accumulator (50H) and stores the result
        ; (20H) in the accumulator.
```

9. SUB M This instruction subtracts the contents of the memory location pointed by HL register pair from the contents of accumulator and stores the result in the accumulator. The HL register pair is used as a memory pointer.

Example : HL = 1020H, A = 50H, (1020H) = 10H

```
SUB M          ; This instruction will subtract the contents of memory location
               ; pointed by HL register pair, 1020H, i.e. data 10H from the
               ; contents accumulator, i.e. data 50H and stores the result
               ; (40H) in accumulator.
```

10. SUI data (8) This instruction subtracts an 8 bit data given within the instruction from the contents of the accumulator and stores the result in the accumulator.

Example : A = 40H,

```
SUI 20H        ; This instruction will subtract 20H from the contents of
               ; accumulator (40H). It will store the result (20H) in the
               ; accumulator.
```

11. SBB r This instruction subtracts the specified register contents and borrow flag from the accumulator contents. This means, if the carry flag (borrow for subtraction) is set by some previous operation, it subtracts 1 and the contents of the specified register from the contents of accumulator, else it subtracts the contents of the specified register only. The register r is 8-bit register such as A, B, C, D, E, H and L.

Example : Carry flag = 1, C = 20H, A = 40H

```
SBB C          ; This instruction will subtract the contents of C register (20H)
               ; and carry flag (1) from the contents of accumulator (40H).
               ; It will store the result (40H - 20H – 1 = 1FH) in the
               ; accumulator.
```

12. SBB M This instruction subtracts the contents of memory location pointed by HL register pair from the contents of accumulator and borrow flag and stores the result in the accumulator.

Example : Carry flag = 1, HL = 2050H, A = 50H, (2050H) = 10H.

```
SBB M          ; This instruction will subtract the contents of memory location
               ; pointed by HL register pair, 2050H, i.e. data 10H and borrow
               ; (Carry flag = 1) from the contents of accumulator (50H) and
               ; stores the result 3FH in the accumulator (50 – 10 – 1 = 3F).
```

13. SBI data (8) This instruction subtracts 8 bit data given within the instruction and borrow flag from the contents of accumulator and stores the result in the accumulator.

Example : Carry flag = 1, A = 50H

```
SBI 20H     ; This instruction will subtract 20H and the carry flag (1)
            ; from the contents of the accumulator (50H). It will store
            ; the result (50H – 20H – 1 = 2FH) in the accumulator.
```

14. DAA This instruction adjusts accumulator to packed BCD (Binary Coded Decimal) after adding two BCD numbers.

Example :

If, A = 0011 1001 = 39 BCD

and C = 0001 0010 = 12 BCD then

```
        ADD C       ; Gives A = 0100 1011 = 4BH
        DAA         ; adds 0110 because 1011 > 9, A = 0101 0001 = 51
                    ; BCD
```

If A = 1001 0110 = 96 BCD

and D = 0000 0111 = 07 BCD then

```
        ADD D       ; Gives A = 1001 1101 = 9DH
        DAA         ; adds 0110 because 1101 > 9,
                    ; A = 1010 0011 = A3H
                    ; 1010 > 9 so adds 0110 0000
                    ; A = 0000 0011 = 03 BCD, CF = 1.
```

15. INR r This instruction increments the contents of specified register by 1. The result is stored in the same register. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : B = 10H

```
INR B       ; This instruction will increment the contents of B register
            ; (10H) by one and stores the result (10 + 1 = 11H) in the
            ; same i.e. B register.
```

16. INR M This instruction increments the contents of memory location pointed by HL register pair by 1. The result is stored at the same memory location. The HL register pair is used as a memory pointer.

Example : HL = 2050H, (2050H) = 30H

```
INR M       ; This instruction will increment the contents of
            ; memory location pointed by HL register pair, 2050H, i.e.
```

; data 30H by one. It will store the result (30 + 1 = 31H) at the
; same place.

17. INX rp This instruction increments the contents of register pair by one. The result is stored in the same register pair. The rp is register pair such as BC, DE, HL or stack pointer (SP).

Example : HL = 10FFH

```
INX H   ; This instruction will increment the contents of HL register
        ; pair (10FFH) by one. It will store the result
        ; (10FF + 1 = 1100H) in the same i.e. HL register pair.
```

18. DCR r This instruction decrements the contents of the specified register by one. It stores the result in the same register. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example : E = 20H

```
DCR E   ; This instruction will decrement the contents of E register
        ; (20H) by one. It will store the result (20 – 1 = 1FH) in the
        ; same, i.e. E register.
```

19. DCR M This instruction decrements the contents of memory location pointed by HL register pair by 1. The HL register pair is used as a memory pointer. The result is stored in the same memory location.

Example : HL = 2050H, (2050H) = 21H

```
DCR M   ; This instruction will decrement the contents of memory
        ; location pointed by HL register pair, 2050H, i.e. data 21H by
        ; one. It will store the result (21 – 1 = 20H) in the same
        ; memory location.
```

20. DCX rp This instruction decrements the contents of register pair by one. The result is stored in the same register pair. The rp is register pair such as BC, DE, HL or stack pointer (SP). Only higher order register is to be specified within the instruction.

Example : DE = 1020H

```
DCX D   ; This instruction will decrement the contents of DE register
        ; pair (1020H) by one and store the result (1020 – 1 = 101FH)
        ; in the same, DE register pair.
```

1.6.3 Branch Group

1. JMP addr This instruction loads the PC with the address given within the instruction and resumes the program execution from this location.

Example :

```
JMP 2000H   ; This instruction will load PC with 2000H and processor will
            ; fetch next instruction from this address.
```

2. Jcond addr This instruction causes a jump to an address given in the instruction if the desired condition occurs in the program before the execution of the instruction. The table 1.3 shows the possible conditions for jumps.

Instruction code	Description	Condition for jump
JC	Jump on carry	CY = 1
JNC	Jump on not carry	CY = 0
JP	Jump on positive	S = 0
JM	Jump on minus	S = 1
JPE	Jump on parity even	P = 1
JPO	Jump on parity odd	P = 0
JZ	Jump on zero	Z = 1
JNZ	Jump on not zero	Z = 0

Table 1.3 Conditional jumps

Example : Carry flag = 1

```
JC 2000H    ; This instruction will cause a jump to an address 2000H
            ; i.e. program counter will load with 2000H since CF = 1.
```

3. CALL addr The CALL instruction is used to transfer program control to a subprogram or subroutine. This instruction pushes the current PC contents onto the stack and loads the given address into the PC. Thus the program control is transferred to the address given in the instruction. Stack pointer is decremented by two.

Example : Stack pointer = 3000H.

```
6000H CALL 2000H ; This instruction will store the address of instruction next to
6003H  —         ; CALL (i.e. 6003H) on the stack and load PC with 2000H.
```

4. C cond addr This instruction calls the subroutine at the given address if a specified condition is satisfied. Before call it stores the address of instruction next to the call on the stack and decrements stack pointer by two. The table 1.4 shows the possible conditions for calls.

Instruction code	Description	Condition for CALL
CC	Call on carry	CY = 1
CNC	Call on not carry	CY = 0
CP	Call on positive	S = 0
CM	Call on minus	S = 1
CPE	Call on parity even	P = 1
CPO	Call on parity odd	P = 0
CZ	Call on zero	Z = 1
CNZ	Call on not zero	Z = 0

Table 1.4 Conditional calls

Example : Carry flag = 1, stack pointer = 4000H.

```
2000H CC 3000H   ; This instruction will store the address of the next instruction
                 ; i.e. 2003H on the stack and load the program
                 ; counter with 3000H.
```

5. RET This instruction pops the return addr (address of the instruction next to CALL in the main program) from the stack and loads program counter with this return address. Thus transfers program control to the instruction next to CALL in the main program.

Example : If SP = 27FDH and contents on the stack are as shown then


```
RET              ; This instruction will load PC with 6200H and it will transfer
                 ; program control to the address 6200H. It will also increment
                 ; the stack pointer by two.
```

6. R condition This instruction returns the control to the main program if the specified condition is satisfied. Table 1.5 shows the possible conditions for return.

Instruction code	Description	Condition for RET
RC	Return on carry	CY = 1
RNC	Return on not carry	CY = 0
RP	Return on positive	S = 0
RM	Return on minus	S = 1
RPE	Return on parity even	P = 1
RPO	Return on parity odd	P = 0
RZ	Return on zero	Z = 1
RNZ	Return on not zero	Z = 0

Table 1.5 Conditions for return

7. PCHL This instruction loads the contents of HL register pair into the program counter. Thus the program control is transferred to the location whose address is in HL register pair.

Example : HL = 6000H

```
PCHL    ; This instruction will load 6000H into the program counter
```

8. RST n This instruction transfers the program control to the specific memory address as shown in Table 1.6. This instruction is like a fixed address CALL instruction. These fixed addresses are also referred to as vector addresses. The processor multiplies the RST number by 8 to calculate these vector addresses. Before transferring the program control to the instruction following the vector address RST instruction saves the current program counter contents on the stack like CALL instruction

Instruction code	Vector Address
RST 0	0 × 8 = 0000H
RST 1	1 × 8 = 0008H
RST 2	2 × 8 = 0010H
RST 3	3 × 8 = 0018H
RST 4	4 × 8 = 0020H
RST 5	5 × 8 = 0028H
RST 6	6 × 8 = 0030H
RST 7	7 × 8 = 0038H

Table 1.6 Vector addresses for return instructions

Example : SP = 3000H

```
2000H RST 6   ; This instruction will save the current contents of the program
              ; counter (i.e. address of next instruction 2001H) on the stack
              ; and it will load the program counter with vector address
              ; (6 × 8 = 48_10 = 30H) 0030H.
```

1.6.4 Logic Group

1. ANA r

This instruction logically ANDs the contents of the specified register with the contents of accumulator and stores the result in the accumulator. Each bit in the accumulator is logically ANDed with the corresponding bit in register r, i.e. D_0 bit in A with D_0 bit in register r, D_1 in A with D_1 in r and so on upto D_7 bit. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example :

```
                ; A = 10101010 (AAH), B = 00001111 (0FH)
ANA B           ; This instruction will logically AND the contents of B register
1010 1010       ; with the contents of accumulator. It will store the result
                ; (0AH)
0000 1111       ; in the accumulator.
---------
0000 1010 = 0AH
```

2. ANA M

This instruction logically ANDs the contents of memory location pointed by HL register pair with the contents of accumulator. The result is stored in the accumulator. The HL register pair is used as a memory pointer.

Example :

```
                ; A = 01010101 = (55H),  HL = 2050H
                ; (2050H) → 10110011 = (B3H)
ANA M           ; This instruction will logically AND the contents of memory
0101 0101       ; location pointed by HL register pair (B3H) with the contents
1011 0011       ; of accumulator (55H). It will store the result (11H) in
---------       ; the accumulator
0001 0001 = 11H
```

3. ANI data

This instruction logically ANDs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Example :

```
                A = 1011 0011 = (B3H)
ANI 3FH         ; This instruction will logically AND the contents
                ; of accumulator (B3H) with 3FH. It will store the result (33H)
                ; in the accumulator.
1011  0011
0011  1111
---------
0011 0011 = 33H
```

4. XRA r This instruction logically XORs the contents of the specified register with the contents of accumulator and stores the result in the accumulator. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example :

```
A = 1010 1010 (AAH)
; C = 0010 1101 (2DH)
XRA C      ; This instruction will logically XOR the contents of C register
1010 1010  ; with the contents of accumulator. It will store the result
0010 1101  ; (87H) in the accumulator.
-----------
1000 0111 = (87H)
```

5. XRA M This instruction logically XORs the contents of memory location pointed by HL register pair with the contents of accumulator. The HL register pair is used as a memory pointer.

Example :

```
; A = 0101 0101 = (55H),  HL = 2050H
; (2050H) → 1011 0011 = (B3H)
XRA M      ; This instruction will logically XOR the contents of memory
0101 0101  ; location pointed by HL register pair (2050H) i.e. data B3H
1011 0011  ; with the contents of accumulator (55H). It will store the
-----------; result (E6H) in the accumulator.
1110 0110 = E6H
```

6. XRI data This instruction logically XORs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Example :

```
; A = 10110011 = (B3H)
XRI  39H   ; This instruction will logically XOR the contents of
           ; accumulator (B3H) with 39H.
1011 0011  ; It will store the result (8AH) in the accumulator.
0011 1001
-----------
1000 1010 = 8AH
```

7. ORA r This instruction logically ORs the contents of specified register with the contents of accumulator and stores the result in the accumulator. Each bit in the accumulator is ORed with corresponding bit in register r. i.e. D_0 bit in accumulator is ORed with D_0 bit in register

r, D_1 in A with D_1 in r and so on upto D_7 bit. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example :

```
; A = 1010 1010 (AAH), B = 0001 0010 (12H)
ORA B          ; This instruction will logically OR the contents of B register
1010 1010      ; with the contents of accumulator. It will store the result
0001 0010      ; (BAH) in the accumulator.
---------
1011 1010 = BAH
```

8. ORA M

This instruction logically ORs the contents of memory location pointed by HL register pair with the contents of accumulator. The result is stored in the accumulator. The HL register pair is used as a memory pointer.

Example :

```
; A = 0101 0101 = (55H)   HL = 2050H
; (2050H) → 1011 0011 = (B3H)
ORA M          ; This instruction will logically OR the contents of memory
0101 0101      ; location pointed by HL register pair (B3H) with the contents
1011 0011      ; of accumulator (55H). It will store the result (F7H) in the
---------      ; accumulator.
1111 0111 = F7H
```

9. ORI data

This instruction logically ORs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Example :

```
A = 1011 0011 = (B3H)
ORI 08H        ; This instruction will logically OR the contents of accumulator
1011 0011      ; (B3H) with 08H. It will store the result (BBH) in the
0000 1000      ; accumulator.
---------
1011 1011  (BBH)
```

10. CMP r

This instruction subtracts the contents of the specified register from contents of the accumulator and sets the condition flags as a result of the subtraction. It sets zero flag if A = r and sets carry flag if A < r. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Example :

```
; A = 1011 1000 (B8H) and D = 1011 1001 (B9H)
CMP D          ; This instruction will compare the contents of D register with
               ; the contents of accumulator. Here A < D so carry flag will
               ; set after the execution of the instruction.
```

11. CMP M This instruction subtracts the contents of the memory location specified by HL register pair from the contents of the accumulator and sets the condition flags as a result of subtraction. It sets zero flag if A = M and sets carry flag if A < M. The HL register pair is used as a memory pointer.

Example :

```
; A = 1011 1000 (B8H), HL = 2050H
; and (2050H) = 1011 1000 (B8H)
CMP M   ; This instruction will compare the contents of memory
        ; location (B8H) and the contents of accumulator. Here A = M
        ; so zero flag will set after the execution of the instruction.
```

12. CPI data This instruction subtracts the 8 bit data given in the instruction from the contents of the accumulator and sets the condition flags as a result of subtraction. It sets zero flag if A = data and sets carry flag if A < data.

Example :

```
; A = 1011 1010 = (BAH)
CPI 30H ; This instruction will compare 30H with the contents of
        ; accumulator (BAH). Here A > data so zero and carry both
        ; flags will reset after the execution of the instruction.
```

13. STC This instruction sets carry flag = 1

Example :

```
Carry flag = 0
STC     ; This instruction will set the carry flag = 1
```

14. CMC This instruction complements the carry flag.

Example :

```
Carry flag = 1
CMC     ; This instruction will complement the carry flag
        i.e. carry flag = 0
```

15. CMA This instruction complements each bit of the accumulator.

Example :

```
A = 1000 1000 = 88H
CMA     ; This instruction will complement each bit of
        ; accumulator A = 0111 0111 = 77H
```

Rotate

1. RLC This instruction rotates the contents of the accumulator left by one position. Bit B_7 is placed in B_0 as well as in CY.

Example :

```
; A = 01010111  (57H) and CY = 1
RLC     ; After execution of the instruction the accumulator contents
        ; will be (1010 1110) AEH and carry flag will reset.
```

2. RRC This instruction rotates the contents of the accumulator right by one position. Bit B_0 is placed in B_7 as well as in CY.

Example :
; A = 1001 1010 (9AH) and CY = 1
; After execution of the instruction the accumulator contents
; will be (0100 1101) 4DH and carry flag will reset.

3. RAL This instruction rotates the contents of the accumulator left by one position. Bit B_7 is placed in CY and CY is placed in B_0.

Example :
; A = 10101101 (ADH) and CY = 0
RAL ; After execution of the instruction accumulator contents will
; be (0101 1010) 5AH and carry flag will set.

4. RAR This instruction rotates the contents of the accumulator right by one position. Bit B_0 is placed in CY and CY is placed in B_7.

Example :
RAR ; A = 1010 0011 (A3H) and CY = 0
; After execution of the instruction accumulator contents will
; be (0101 0001) 51H and carry flag will set.

1.6.5 Stack Operations

1. PUSH rp This instruction decrements stack pointer by one and copies the higher byte of the register pair into the memory location pointed by stack pointer. It then decrements the stack pointer again by one and copies the lower byte of the register pair into the memory location pointed by stack pointer. The rp is 16-bit register pair such as BC, DE, HL. Only higher order register is to be specified within the instruction.

Example : SP = 2000H, DE = 1050H.
PUSH D ; This instruction will decrement the stack pointer (2000H) by one (SP = 1FFFH) and copies the contents of D register (10H) into the memory location 1FFFH. It then decrements the stack pointer again by one (SP = 1FFEH) and copies the contents of E register (50H) into the memory location 1FFEH.

2. PUSH PSW This instruction decrements stack pointer by one and copies the accumulator contents into the memory location pointed by stack pointer. It then decrements the stack pointer again by one and copies the flag register into the memory location pointed by the stack pointer.

Example : SP = 2000H, A = 20H, Flag register = 80H

PUSH PSW This instruction decrements the stack pointer (SP = 2000H) by one (SP = 1FFFH) and copies the contents of the accumulator (20H)into the memory location 1FFFH. It then decrements the stack pointer

again by one (SP = 1FFEH) and copies the contents of the flag register (80H) into the memory location 1FFEH.

3. POP rp

This instruction copies the contents of memory location pointed by the stack pointer into the lower byte of the specified register pair and increments the stack pointer by one. It then copies the contents of memory location pointed by stack pointer into the higher byte of the specified register pair and increments the stack pointer again by one. The rp is 16-bit register pair such as BC, DE, HL. Only higher order register is to be specified within the instruction.

Example :

```
SP = 2000H, (2000H) = 30H, (2001H) = 50H
POP B   ; This instruction will copy the contents of memory location
        ; pointed by stack pointer, 2000H (i.e. data 30H) into the C
        ; register. It will then increment the stack pointer by one,
        ; 2001H and will copy the contents of memory location
        ; pointed by stack pointer, 2001H (i.e. data 50H) into B
        ; register, and increment the stack pointer again by one.
```

4. POP PSW

This instruction copies the contents of memory location pointed by the stack pointer into the flag register and increments the stack pointer by one. It then copies the contents of memory location pointed by stack pointer into the accumulator and increments the stack pointer again by one.

Example :

```
SP = 2000H, (2000H) = 30H, (2001H) = 50H
POP PSW ; This instruction will copy the contents of memory location
        ; pointed by the stack pointer, 2000H (i.e. data 30H) into the
        ; flag register. It will then increment the stack pointer by one,
        ; 2001H and will copy the contents of memory location
        ; pointed by stack pointer into the accumulator and increment
        ; the stack pointer again by one.
```

5. SPHL

This instruction copies the contents of HL register pair into the stack pointer. The contents of H register are copied to higher order byte of stack pointer and contents of L register are copied to the lower byte of stack pointer.

Example :

```
HL = 2500H
SPHL    ; This instruction will copy 2500H into stack pointer. So after
        ; execution of instruction stack pointer contents will be 2500H.
```

6. XTHL

This instruction exchanges the contents of memory location pointed by the stack pointer with the contents of L register and the contents

of the next memory location with the contents of H register. This instruction does not modify stack pointer contents.

Example :
; HL = 3040H and SP = 2700H, (2700H) = 50H, (2701H) = 60H
XTHL ; This instruction will exchange the contents of L register
; (40H) with the contents of memory location 2700H (i.e. 50H)
; and the contents of H register (30H) with the contents of
; memory location 2701H (i.e. 60H).

Input/Output

1. **IN addr(8-bit)** This instruction copies the data at the port whose address is specified in the instruction into the accumulator.

Example :
Port address = 80H, data stored at port address 80H, (80H) = 10H
IN 80H ; This instruction will copy the data stored at address 80H, i.e.
; data 10H in the accumulator.

2. **OUT addr(8-bit)** This instruction sends the contents of accumulator to the output port whose address is specified within the instruction.

Example :
A = 40H
OUT 50H ; This instruction will send the contents of accumulator
; (40H) to the output port whose address is 50H.

1.6.6 Machine Control Group

1. **EI** This instruction sets the interrupt enable flip flop to enable interrupts. When the microprocessor is reset or after interrupt acknowledge, the interrupt enable flip-flop is reset. This instruction is used to reenable the interrupts.

2. **DI** This instruction resets the interrupt enable flip-flop to disable interrupts. This instruction disables all interrupts except TRAP since TRAP is non-maskable interrupt (cannot be disabled. It is always enabled).

3. **NOP** No operation is performed.

4. **HLT** This instruction halts the processor. It can be restarted by a valid interrupt or by applying a RESET signal.

5. **SIM** This instruction masks the interrupts as desired. It also sends out serial data through the SOD pin. For this instruction command byte must be loaded in the accumulator.

Example : ; i) A = 0EH

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
SOD	SOE	X	RST7.5	MSE	M7.5	M6.5	M5.5	Register A
0	0	0	0	1	1	1	0	= 0EH

```
SIM          ; This instruction will mask RST 7.5 and RST 6.5 interrupts
             ; where as RST 5.5 interrupt will be unmasked. It will also
             ; disable serial output.
```

6. RIM This instruction copies the status of the interrupts into the accumulator. It also reads the serial data through the SID pin.

Example :

```
RIM          ; After execution of RIM instruction if the contents of
             ; accumulator are 4BH then we get following information.
```

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
SID	I 7.5	I 6.5	I 5.5	IE	M7.5	M6.5	M5.5	Register A
0	1	0	0	1	0	1	1	= 4BH

i.e. a) RST 7.5 is pending

b) RST 5.5 and RST 6.5 are masked

c) Interrupt Enable flip-flop is set

d) Serial i/p data is zero.

1.7 8085 Interrupt Structure and Operation

1.7.1 Types of Interrupts

The 8085 has multilevel interrupt system. It supports two types of interrupts:

a. Hardware b. Software

Hardware : Some pins on the 8085 allow peripheral device to interrupt the main program for I/O operations. When an interrupt occurs, the 8085 completes the instruction it is currently executing and transfers the program control to a subroutine that services the peripheral device. Upon completion of the service routine, the MPU returns to the main program. These types of interrupts, where MPU pins are used to receive interrupt requests, are called **hardware interrupts**.

Software : In software interrupts, the cause of the interrupt is an execution of the instruction. These are special instructions supported by the microprocessor. After execution of these instructions microprocessor completes the execution of the instruction it is currently executing and transfers the program control to the subroutine program. Upon completion of the execution of the subroutine program, program control returns to the main program.

1.7.2 Overall Interrupt Structure

1.7.2.1 Hardware Interrupts in 8085

The 8085 has five hardware interrupts :

1. TRAP 2. RST 7.5 3. RST 6.5 4. RST 5.5 5. INTR

When any of these pins, except INTR, is active, the internal control circuit of the 8085 produces a CALL to a predetermined memory location. This memory location, where the subroutine starts is referred to as **vector location** and such interrupts are called **vectored interrupts**. The INTR is not a vectored interrupt. It receives the address of the subroutine from the external device.

In 8085, all interrupts except TRAP are maskable. When logic signal is applied to a maskable interrupt input, the 8085 is interrupted only if that particular input is enabled. These interrupts can be enabled or disabled under program control. If disabled, 8085 disables an interrupt request. The interrupt TRAP is nonmaskable whish means that it is not maskable by program control. The Fig. 1.38 shows the interrupt structure of 8085. The

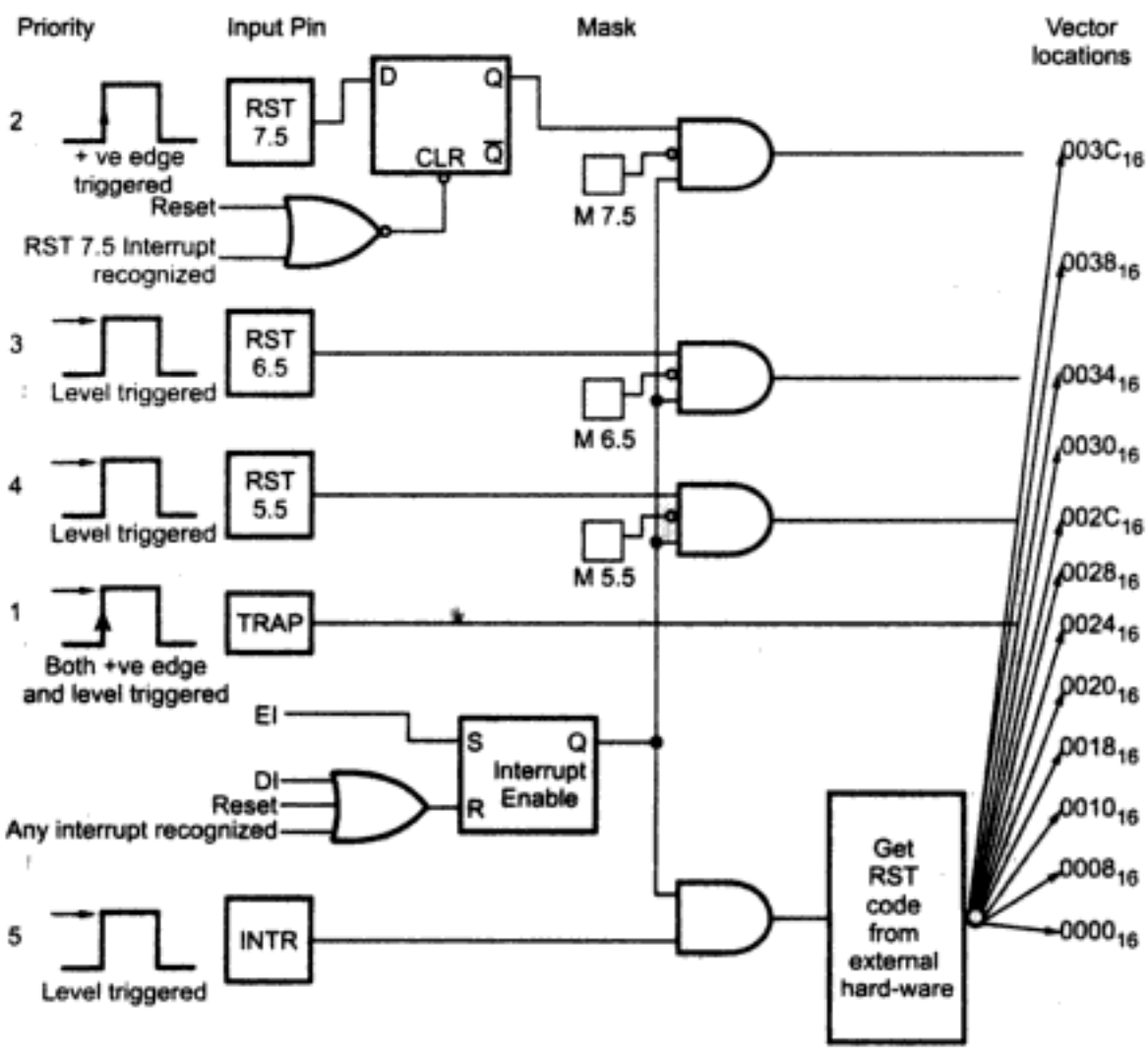


Fig. 1.38 Interrupt structure of 8085

figure indicates that, the 8085 is designed to respond to edge triggering, level triggering or both.

TRAP : This interrupt is a nonmaskable interrupt. It is unaffected by any mask or interrupt enable. TRAP has the highest priority. TRAP interrupt is edge and level triggered. This means that the TRAP must go high and remain high until it is acknowledged. This avoids false triggering caused by noise and transients.

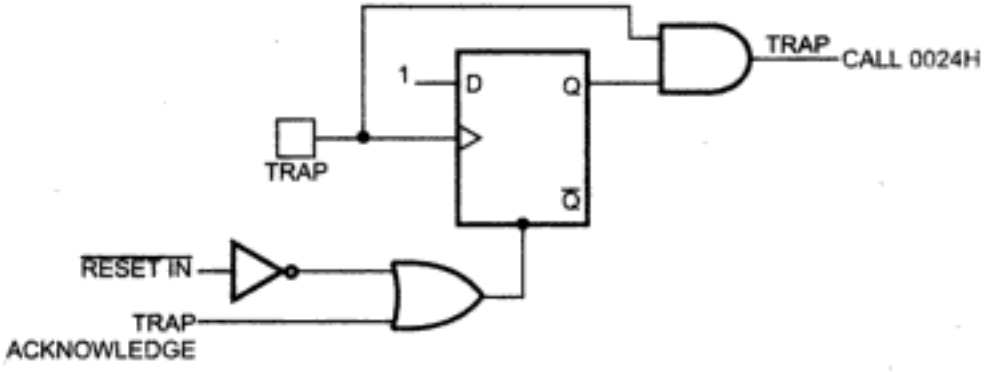


Fig. 1.39 Interrupt circuit for trap interrupt

As shown in the Fig. 1.39, the positive edge of TRAP signal sets the D flip-flop. However, due to the AND gate, it is necessary to sustain high level on the TRAP input. **There are two ways to clear TRAP interrupt :**

1. By resetting microprocessor i.e. giving a low signal on $\overline{\text{RESETIN}}$ pin (External signal).
2. By giving a high TRAP ACKNOWLEDGE (Internal signal).

After recognition of TRAP interrupt, 8085 internally generates a high TRAP ACKNOWLEDGE which clears the flip flop. Once the TRAP is acknowledged, the 8085 completes its current instruction. It then pushes the address of the next instruction i.e. return address onto the stack and loads PC with the fixed vector address 0024H. Due to this, 8085 starts execution of instructions from address 0024H which is the starting address of an interrupt service routine for TRAP.

RST 7.5 : The RST 7.5 interrupt is a maskable interrupt. It has the second highest priority. As shown in Fig. 1.38, it is **positive edge triggered** and the positive edge trigger is stored internally by the D-flip flop until it is cleared by software reset using SIM instruction or by internally generated ACKNOWLEDGE signal.

The positive edge signal on the RST 7.5 pin sets the D flip flop. If the mask bit M 7.5 is 0 i.e. RST 7.5 is unmasked then 8085 completes its current instruction. It then pushes the address of the next instruction onto the stack and loads PC with the fixed vector address 003CH. Due to this, 8085 starts execution of instructions from address 003CH which is the starting address of an interrupt service routine for RST 7.5.

RST 6.5 and RST 5.5 : The RST 6.5 and RST 5.5 both are **level triggered**. These interrupts can be masked using SIM instruction. The RST 6.5 has the third priority whereas RST 5.5 has the fourth priority. The vector addresses of RST 6.5 and RST 5.5 are 0034H and 002CH respectively. After recognition of RST 6.5 or RST 5.5 interrupt, 8085 completes its current instruction; pushes the address of next instruction onto the stack and loads PC with corresponding vector address.

INTR : INTR is a maskable interrupt, but not the vector interrupt. It has the lowest priority. The following sequence of events occur when INTR signal goes high.

1. The 8085 checks the status of INTR signal during execution of each instruction.
2. If INTR signal is high, then 8085 completes its current instruction and sends an active low interrupt acknowledge signal ($\overline{\text{INTA}}$) if the interrupt is enabled.
3. In response to the $\overline{\text{INTA}}$ signal, external logic places an instruction OPCODE on the data bus. In the case of multibyte instruction, additional interrupt acknowledge machine cycles are generated by the 8085 to transfer the additional bytes into the microprocessor.
4. On receiving the instruction, the 8085 saves the address of next instruction on stack and executes received instruction.

Note : Theoretically, the external logic can place any instruction code on the data bus in response to the $\overline{\text{INTA}}$. However, only CALL and RST codes save the contents of the PC on the stack and branch program control to the subroutine address.

Response for RST instruction : If the external device places an opcode for any one of the RST instruction (RST 0 - RST 7), then 8085 pushes the contents of PC onto the stack. It then branches the program control to the vector address of the corresponding RST instruction.

Response for CALL instruction : If the external device places an opcode for CALL instruction then 8085 generates two additional interrupt acknowledge cycles.

1. It sends an active low interrupt acknowledge signal second time.
2. In response to second $\overline{\text{INTA}}$ signal, external logic places the lower byte address for the CALL instruction.
3. After receiving lower byte address, 8085 sends the third interrupt acknowledge signal.
4. In response to third $\overline{\text{INTA}}$ signal, external logic places the higher byte address for the CALL instruction.
5. After receiving sixteen bit address for CALL, 8085 pushes the contents of the PC onto the stack and branches the program control to the subroutine whose address is received from the external logic.

Example : The Fig. 1.40 shows the diagram of external logic that gives the RST 7 instruction opcode on interrupt acknowledge.

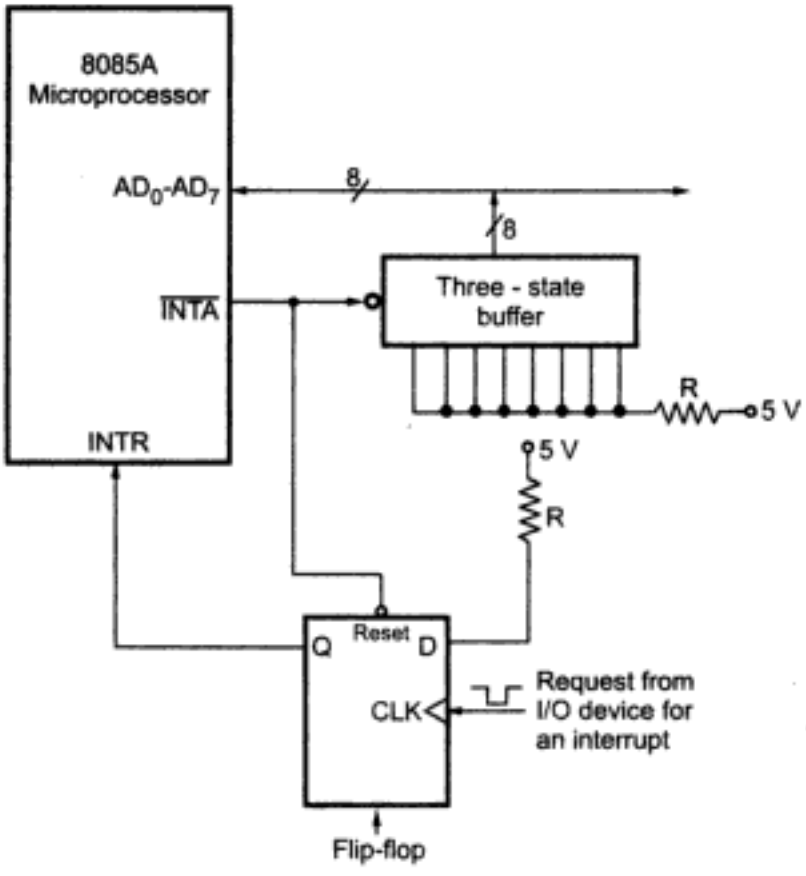


Fig. 1.40 External logic that gives the RST 7 instruction opcode

External logic controls a tri-state buffer with the $\overline{INTA}$ signal in order to place an opcode for RST 7 instruction. The $\overline{INTA}$ signal from the microprocessor is used as an Output Enable signal for the buffer as well as reset signal for D flip flop. The request from the I/O device is routed through the D flip-flop to the INTR. The D flip flop is used to hold the INTR signal high until 8085 gives interrupt acknowledge signal. The $\overline{INTA}$ signal that is generated enables the tri-state buffer whose data inputs are hardwired to the value equal to the opcode for RST 7 (FFH) instruction. The 8085 receives this opcode during interrupt acknowledge cycle. After receiving the opcode 8085 pushes the contents of program counter onto the stack, thus saving the return address. It then branches the program control to the address 0038H (Vector address of RST 7). Table 1.7 shows the summary of hardware interrupts in 8085.

Interrupt type	Trigger	Priority	Maskable	Vector address
TRAP	Edge and Level	1st (Highest)	No	0024 H
RST 7.5	Edge	2nd	Yes	003CH
RST 6.5	Level	3rd	Yes	0034H
RST 5.5	Level	4th	Yes	002CH
INTR	Level	5th (Lowest)	Yes	-

Table 1.7

1.7.2.2 Software Interrupts in 8085

The 8085 has eight software interrupts from RST 0 to RST 7. The vector address for these interrupts can be calculated as follows.

Interrupt number × 8 = vector address

For example : 5 × 8 = 40 = 28H

∴ Vector address for interrupt RST 5 is 0028H.

The Table 1.8 shows the vector addresses of all interrupts.

Instruction	HEX code	Vector Address
RST 0	C7	0000H
RST 1	CF	0008H
RST 2	D7	0010H
RST 3	DF	0018H
RST 4	E7	0020H
RST 5	EF	0028H
RST 6	F7	0030H
RST 7	FF	0038H

Table 1.8 Vector addresses for software interrupts

1.7.3 Masking / Unmasking of Interrupts

As mentioned earlier, maskable interrupts are enabled and disabled under program control. In this section we will see how interrupts can be masked or unmasked using program control. There are four instructions used for control of interrupts :

1. EI
2. DI
3. RIM
4. SIM

EI : Enable Interrupt

The EI instruction sets the interrupt enable flip-flop, as shown in Fig. 1.38. Thus RST 7.5, RST 6.5, RST 5.5 and INTR are enabled using EI instruction.

It is important to note that when any interrupt is acknowledged, interrupt enable flip flop resets and disables all interrupts. To enable interrupt in further process it is necessary to execute EI instruction within interrupt service routine.

DI : Disable Interrupt

The DI instruction resets the interrupt enable flip flop, as shown in Fig. 1.38. Thus it disables RST 7.5, RST 6.5, RST 5.5 and INTR interrupts.

SIM : Set Interrupt Mask

This instruction is used to set interrupt mask and to send serial output. It transfers the contents of accumulator to interrupt control logic and serial I/O port. Thus it is necessary to load appropriate contents in the accumulator before execution of SIM instruction.

SIM Instruction Format :

Bits 0 - 2 will set/reset the mask bits for RST 5.5, RST 6.5, and RST 7.5 of the interrupt mask register.

Bit 3 enables the functioning of bits 0 - 2. It enables or disables the masking control.

Bit 4 is used to reset RST 7.5 request; regardless of whether or not RST 7.5 is masked.

Bit 5 is don't care.

Bit 6 enables the serial output if it is set.

Bit 7 decides the data to be sent on the serial output pin of 8085.

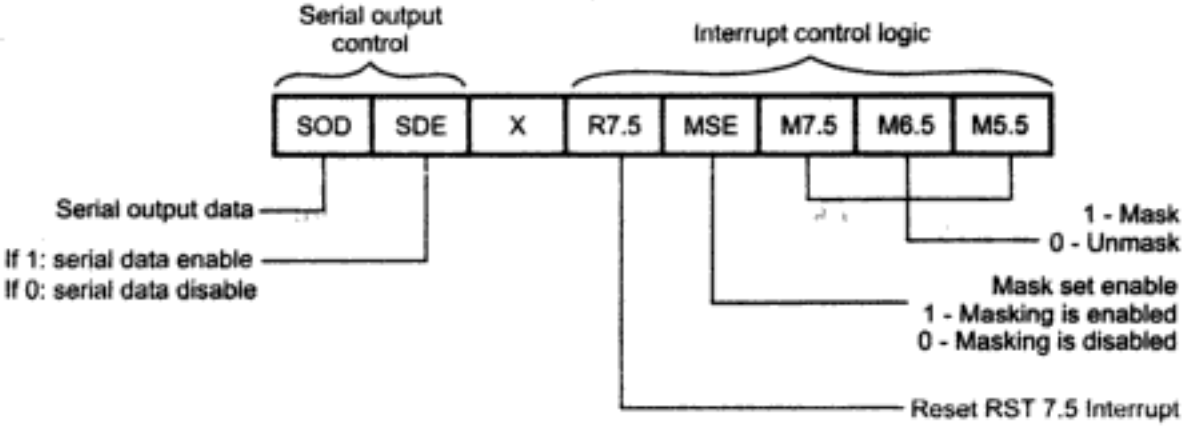


Fig. 1.41 SIM instruction format

Example 1 : To enable RST 6.5 and mask all other interrupts we must execute following instructions.

SOD	SDE	X	R 7.5	MSE	M 7.5	M 6.5	M 5.5		
0	0	0	0	1	1	0	1	=	0DH

```
MVI A, 0DH      ; Load control format in accumulator
SIM             ; Set interrupt mask.
```

Example 2 : The following instruction sequence enables RST 7.5 and RST 6.5 and disables RST 5.5

SOD	SDE	X	R 7.5	MSE	M 7.5	M 6.5	M 5.5		
0	0	0	0	1	0	0	1	=	09H

```
MVI A, 09H      ; load control format in accumulator
SIM             ; Set interrupt mask
```

1.7.4 Pending Interrupts

RIM : Read Interrupt Mask

The Read Interrupt Mask, RIM, instruction loads the status of the interrupt mask, the pending interrupts and the contents of the serial input data line, SID, into the accumulator. Thus, it is possible to monitor status of interrupt mask, pending interrupts and serial input. There are number of interrupts. When one interrupt is being serviced, other interrupt requests may occur. If the interrupt requests are of higher priority, 8085 branches program control to the requested interrupt service routines. But when the interrupt requests are of lower priority, 8085 stores the information about these interrupt requests. Such interrupts are called **pending interrupts**. The status of pending interrupts can be monitored using RIM instruction.

RIM Instruction Format :

Bits 0-2 give the status of interrupt mask. Logic 1 indicates the interrupt is masked.

Bit 3 gives the status of interrupt enable flag. If 1, interrupts are enabled.

Bits 4-6 give the status of pending interrupts.

Bit 7 gives the status of serial input data line.

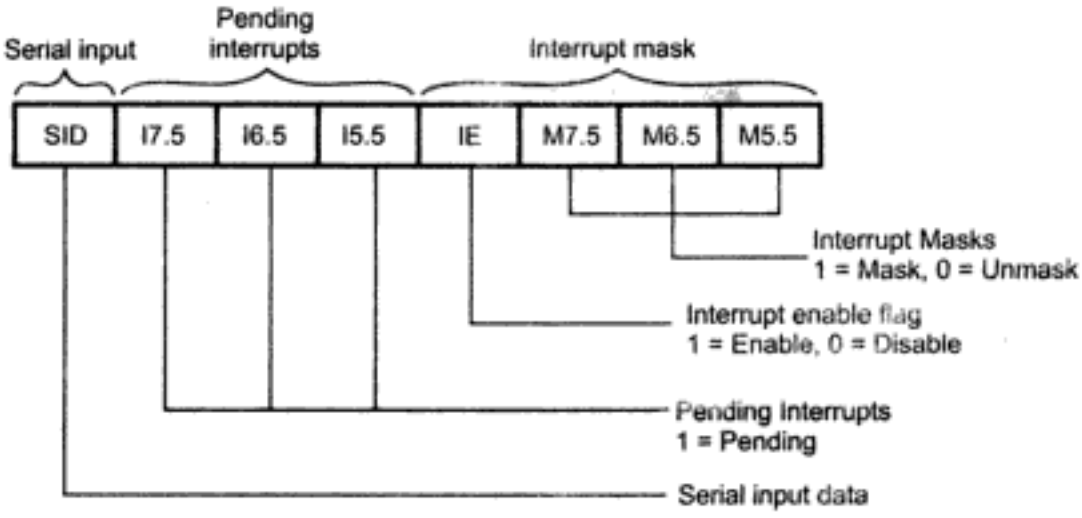


Fig. 1.42 RIM instruction format

Example 3 : To check if RST 5.5 is pending it is necessary to execute following instructions

SID	I 7.5	I 6.5	I 5.5	IE	M 7.5	M 6.5	M 5.5
0	0	0	1	0	0	0	0

= 10H

```
RIM          ; Read interrupt mask and pending interrupts
ANI 10 H     ; Mask all bits except pending RST 5.5 bit.
CNZ 002CH    ; Call interrupt service routine for RST 5.5 if it
             ; is pending.
```

Example 4 : The following instruction sequence checks whether RST 7.5 is individually masked or not.

SID	I 7.5	I 6.5	I 5.5	IE	M 7.5	M 6.5	M 5.5
0	0	0	0	1	1	0	1

= 0DH

```
RIM              ; Read interrupt mask
ANI 04H          ; Mask all bits except RST 7.5
JNZ unmask       ; Jump if not zero to unmask.
```

Example 5 : Write a program to diplay real time clock. Assume that a periodic signal is interrupting RST 7.5 signal after every 0.5 seconds.

Main program

```
MVI C, 00H       ; Initialize counter
LXI H, 0000H     ; Initialize seconds, and minutes
MVI D, 00H       ; Initialize hours
MVI A, 0BH       ;
```

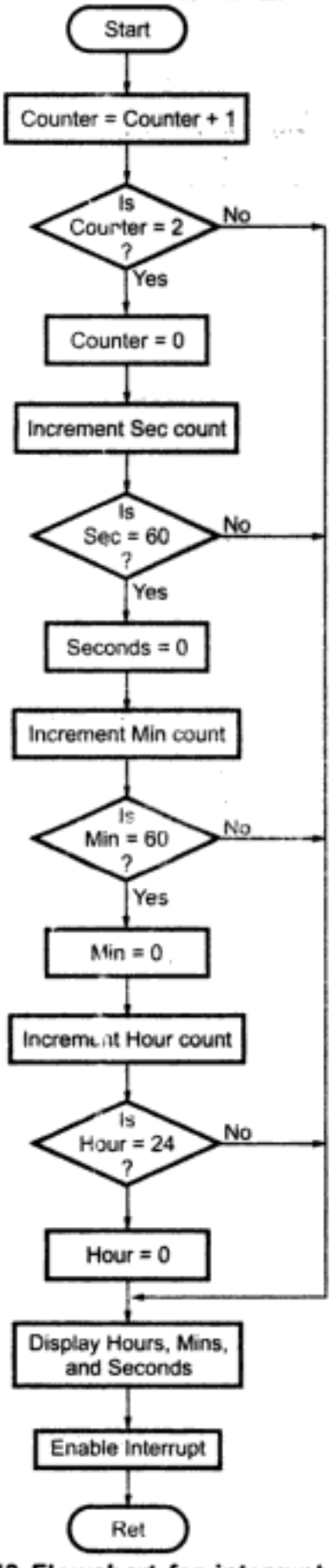


Fig. 1.43 Flowchart for interrupt subroutine

```
        SIM                 ;
        EI                  ; Enable RST 7.5 interrupt
HERE :  JMP HERE            ; Wait for interrupt.
```

ISR - Interrupt service Routine

```
        INR C               ; Increment counter
        MOV A, C
        CPI, 02H
        JNZ LAST            ; Check for 1 second
        MVI C, 00H          ; Reset counter
        MOV A, L            ; Get seconds counter
        ADI 01H             ; Increment seconds counter
        DAA                 ; Adjust for BCD
        MOV L, A            ; Save seconds counter
        CPI 60H             ; Check for 60 seconds
        JNZ LAST            ; If not 60, goto display
        MVI L, 00H          ; Reset seconds counter
        MOV A,H             ; Get minutes counter
        ADI 01H             ; Increment minutes counter
        DAA                 ; Adjust for BCD
        MOV H,A             ; Save minutes counter
        CPI 60H             ; Check for 60 minutes
        JNZ LAST            ; If not 60, goto display
        MVI H, 00H          ; Reset minutes counter
        MOV A,D             ; Get hours counter
        ADI 01H             ; Increment hours counter
        DAA                 ; Adjust for BCD
        MOV D,A             ; Save hours counter
        CPI 24 H            ; Check for 24 hours
        JNZ LAST            ; If not 24, goto display
        MVI D, 00H          ; Reset hours counter
LAST :  CALL DISPLAY        ; Call display subroutine
        EI                  ; Enable Interrupt
        RET                 ; Return to main program
```

Review Questions

1. *Explain the features of 8085.*
2. *Give the clock out frequency and state time, T, of an 8085A operating with each of the following frequency crystals : 6.25 MHz, 6.144 MHz, 5 MHz and 4 MHz.*
3. *List the internal registers in 8085A, their abbreviations and lengths. Describe the primary function of each register.*
4. *Give the format of flag register in 8085. Explain each flag.*
5. *Draw the functional block diagram of microprocessor 8085 and explain in brief.*
6. *Explain different control signals used by 8085.*
7. *Why AD_0-AD_7 lines are multiplexed ?*
8. *What is the use of ALE signal ?*
9. *What is the use of CLKOUT and RESET OUT signals of 8085 processor ?*

10. *Describe the function of following pins in 8085.*
    a. READY b. ALE c. IO/$\overline{M}$ d. HOLD e. RESET
11. *Explain the signals used in DMA operation in 8085.*
12. *Define*
    1. Instruction cycle 2. Machine cycle 3. T state
13. *What is the necessity to have two status lines S_1 and S_0 in 8085 ?*
14. *Explain various machine cycles supported by 8085.*
15. *Draw and explain the memory read cycle of 8085.*
16. *Draw and explain the memory write cycle of 8085.*
17. *Draw and explain the I/O read cycle of 8085.*
18. *Draw and explain the I/O write cycle of 8085.*
19. *Explain the classification of the instruction set of 8085 microprocessor with suitable examples.*
20. *With the help of one example in each case explain the effect of the following instructions in 8085.*
    a. LHLD addr b. ADD M
    c. RST 4 d. XTHL
    e. DAA f. CP 2000
    g. DAD B h. IN 20H
    i. RIM j. SIM
21. *Write the two ways to initialize stack pointer at FFFFH.*
22. *Compare the following pairs of instructions with their opcodes, operations, instruction bytes, addressing modes, affected flags and the resuts.*
    a. MVI A, 00H and XRA A
    b. SUB B and CMP B
    c. JMP 2700 and PCHL
    d. XTHL and SPHL
    e. LDA 2000H and LHLD 2000H
    f. RRC and RAR
23. *What do you mean by hardware interrupts?*
24. *What do you mean by software interrupts?*
25. *Explain the hardware interrupts supported by 8085.*
26. *What do you mean by vectored interrupts?*
27. *Explain how 8085 responds to INTR interrupt.*
28. *Explain the software interrupts supported by 8085.*
29. *What do you mean by masking the interrupt? How is it achieved in 8085?*
30. *What do you mean by pending interrupts?*

❑❑❑

2

Architecture of 8086 Microprocessor

In 1978, Intel came out with the 8086 processor. The Intel 8086 is a 16-bit microprocessor, implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged it in a 40 pin dual in line package. In this chapter, we study features, architecture, register organisation, bus operation and memory segmentation.

2.1 Features of 8086

1. The 8086 is a 16-bit microprocessor. The term "16-bit" means that its arithmetic logic unit, internal registers and most of its instructions are designed to work with 16-bit binary words.
2. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports 8 bits at a time.
3. The 8086 has a 20-bit address bus, so it can directly access 2^{20} or 10,48,576 (1Mb) memory locations. Each of the 10, 48, 576 memory locations is byte wide. Therefore, a sixteen-bit words are stored in two consecutive memory locations. The 8088 also has a 20-bit address bus, so it can also address 2^{20} or 10, 48, 576 memory locations.
4. The 8086 can generate 16-bit I/O address, hence it can access 2^{16} = 65536 I/O ports.
5. The 8086 provides fourteen 16-bit registers.
6. The 8086 has multiplexed address and data bus which reduces the number of pins needed, but does slow down the transfer of data (drawback).
7. The 8086 requires one phase clock with a 33% duty cycle to provide optimized internal timing.

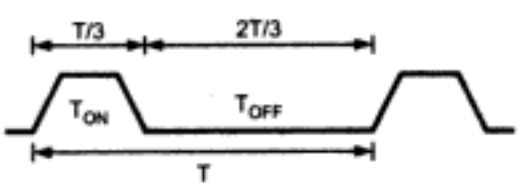


Fig. 2.1 Clock cycle

Range of clock rates (refer Fig. 2.1) are :-
- 5 MHz for 8086
- 8 MHz for 8086-2
- 10 MHz for 8086-1

8. The 8086 is possible to perform bit, byte, word and block operations in 8086. It performs the arithmetic and logical operations on bit, byte, word and decimal numbers including multiply and divide.
9. The Intel 8086 is designed to operate in two modes, namely the minimum mode and the maximum mode. When only one 8086 CPU is to be used in a microcomputer system, the 8086 is used in the minimum mode of operation. In this mode the CPU issues the control signals required by memory and I/O devices. In multiprocessor (more than one processor in the system) system 8086 operates in maximum mode. In maximum mode, control signals are generated with the help of external bus controller (8288).
10. The Intel 8086 supports multiprogramming. In multiprogramming, the code for two or more processes is in memory at the same time and is executed in a time-multiplexed fashion.
11. An interesting feature of the 8086 is that it fetches upto six instruction bytes (4 instruction bytes for 8088) from memory and queue stores them in order to speed up instruction execution. Later we will discuss this in detail.
12. The 8086 provides powerful instruction set with the following addressing modes : Register, immediate, direct, indirect through an index or base, indirect through the sum of a base and an index register, relative and implied.

2.2 Architecture of 8086

Fig. 2.2 shows a block diagram of the 8086 internal architecture. It is internally divided into two separate functional units. These are the Bus Interface Unit (BIU) and the Execution Unit (EU). These two functional units can work simultaneously to increase system speed and hence the throughput. Throughput is a measure of number of instructions executed per unit time.

2.2.1 Bus Interface Unit [BIU]

The bus interface unit is the 8086's interface to the outside world. It provides a full 16-bit bi-directional data bus and 20-bit address bus. The bus interface unit is responsible for performing all external bus operations, as listed below.

Functions of Bus Interface Unit

1. It sends address of the memory or I/O.
2. It fetches instruction from memory.
3. It reads data from port/memory.
4. It writes data into port/memory.
5. It supports instruction queuing.
6. It provides the address relocation facility.

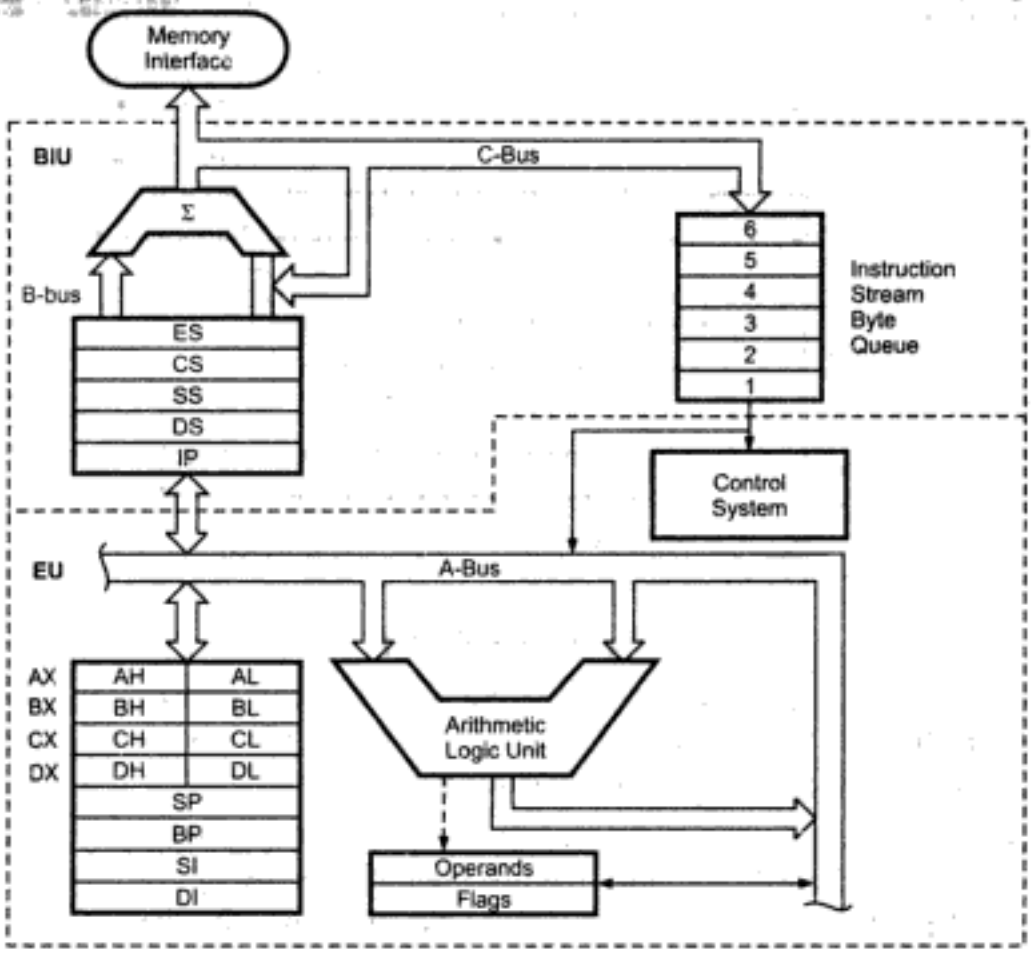


Fig. 2.2 8086 internal block diagram

To implement these functions the BIU contains the instruction queue, segment registers instruction pointer, address summer and bus control logic.

Instruction Queue

To speed up program execution, the BIU fetches **six instruction bytes** ahead of time from the memory. These prefetched instruction bytes are held for the execution unit in a group of registers called **Queue**. With the help of queue it is possible to fetch next instruction when current instruction is in execution. For example, current instruction in execution is a multiplication instruction. In 8086, operands for multiplication operations are within registers. Still it requires 100 clock cycles to execute multiply instruction. Like multiplication there are number of other instructions in 8086 which need quite a large number of clock cycles for execution. During this execution time the BIU fetches the next instruction or instructions from memory into the instruction queue instead of remaining idle. The BIU continues this process as long as the queue is not full. Due to this, execution unit gets the ready instruction in the queue and instruction fetch time is eliminated. This is illustrated in Fig. 2.3.

The queue operates on the principle first in first out (FIFO). So that the execution unit gets the instructions for execution in the order they are fetched. In case of JUMP and CALL instructions, instruction already fetched in queue are of no use. Hence, in these

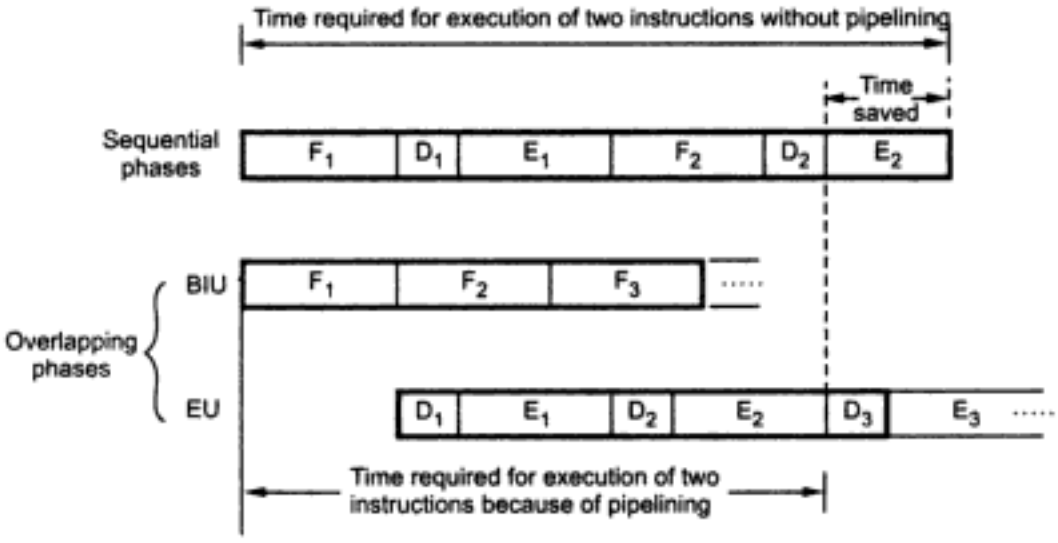


Fig. 2.3 Pipelining

cases queue is dumped and newly formed by loading instructions from new address specified by JUMP or CALL instruction. Feature of fetching the next instruction while the current instruction is executing is called **pipelining**.

The length of the queue should be such that EU should get the next instruction from the queue of the BIU immediately after the execution of the current instruction. To satisfy this, number of pre-fetched instruction in the queue and hence the queue length depends on the fetching speed and the execution speed. Sometime queue length may be restricted due to the space available on the CPU chip.

2.2.2 Execution Unit [EU]

The execution unit of 8086 tells the BIU from where to fetch instructions or data, decodes instructions and executes instructions. It contains

- Control Circuitry
- Instruction Decoder
- Arithmetic Logic Unit (ALU)
- Register Organisation
  - Flag Register
  - General Purpose Registers
  - Pointers and Index Registers

Control Circuitry, Instruction Decoder, ALU

The control circuitry in the EU directs the internal operations. A decoder in the EU translates the instructions fetched from memory into a series of actions which the EU performs. ALU is 16-bit. It can add, subtract, AND, OR, XOR, increment, decrements, complement and shift binary numbers.

2.3 Register Organisation

The 8086 has a powerful set of registers. It includes general purpose registers, segment registers, pointers and index registers, and flag register. The Fig. 2.4 shows the register organisation of 8086. It is also known as programmer's model of 8086. The registers shown in programmer's model are accessible to programmer. As shown in the Fig. 2.4, all the registers of 8086 are 16-bit registers.

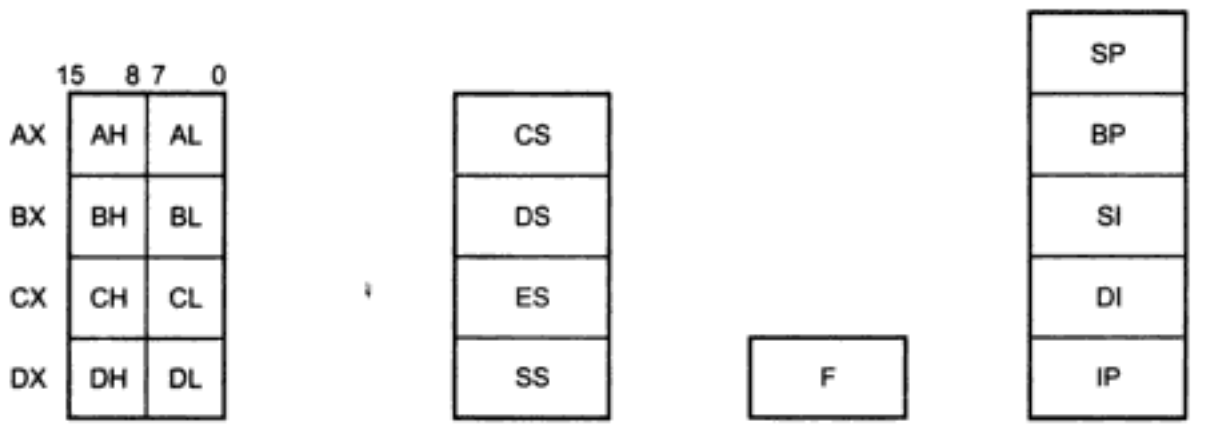


Fig. 2.4 Register organisation of 8086

2.3.1 General Purpose Registers

The 8086 has four 16-bit general purpose registers labeled AX, BX, CX and DX. Each 16-bit general purpose register can be split into two 8-bit registers. The letters L and H specify the lower and higher bytes of a particular register. For example, BH means the higher byte (8-bits) of the BX register and BL means the lower byte (8-bits) of the BX register. The letter X is used to specify the complete 16-bit register.

The general purpose registers are either used for holding data, variables and intermediate results temporarily. They can also be used as a counters or used for storing offset address for some particular addressing modes. The register AX is used as 16-bit accumulator whereas register AL (lowerbyte of AX) is used as 8-bit accumulator. The register BX is also used as offset storage for generating physical addresses in case of certain addressing modes. On the other hand, the register CX is also used as a default counter in case of string and loop instructions.

2.3.2 Segment Registers

The physical address of the 8086 is 20-bits wide to access 1 Mbyte memory locations. However, its registers and memory locations which contain logical addresses are just 16-bits wide. Hence 8086 uses memory segmentation. It treats the 1 Mbyte of memory as divided into segments, with a maximum size of a segment as 64 Kbytes. Thus any location within the segment can be accessed using 16 bits. The 8086 allows only four active

segments at a time, as shown in the Fig. 2.5. For the selection of the four active segments the 16-bit segment registers are provided by the bus interface unit (BIU) of the 8086. These four registers are :

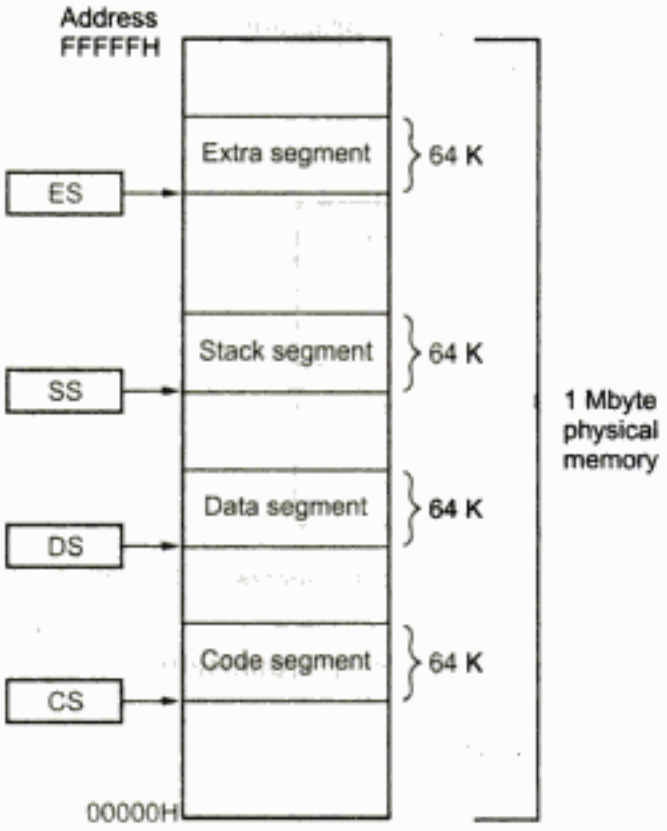


Fig. 2.5 Memory segmentation and segment registers

Code segment (CS) register, the data segment (DS) register, the stack segment (SS) register, and the extra segment (ES) register. These are used to hold the upper 16-bits of the starting addresses of the four memory segments, on which 8086 works at a particular time. For example, the value in CS identifies the starting address of 64 K-byte segment known as code segment. By "starting address", we mean the lowest addressed byte in the active code segment. The starting address is also known as **base address** or **segment base**.

The BIU always inserts zeros for the lower 4 bits (nibble) in the contents of segment register to generate 20-bit base address. For example, if the code segment register contains 348AH, then code segment will start at address 348A0H.

Functions of Segment Registers

1. The CS register holds the upper 16-bits of the starting address of the segment from which the BIU is currently fetching the instruction code byte.
2. The SS register is used for the upper 16-bits of the starting address for the program stack (all stack related instructions will operate on stack).

3. ES register and DS register are used to hold the upper 16-bits of the starting address of the two memory segments which are used for data.

2.3.3 Pointers and Index Registers

All segment registers are 16-bit wide. But it is necessary to generate 20-bit address (physical address) on the address bus. To get 20-bit physical address one or more pointer or index registers are associated with each segment register. The pointer registers IP, BP and SP are associated with code, data and stack segments, respectively. They hold the offset within the code, data and stack segments, respectively. The index registers DI and SI are used as a general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. The detail description of pointers and index register is given in section 2.5.

2.3.4 Flag Register

A flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU. The flag register contains nine active flags as shown in the Fig. 2.6.

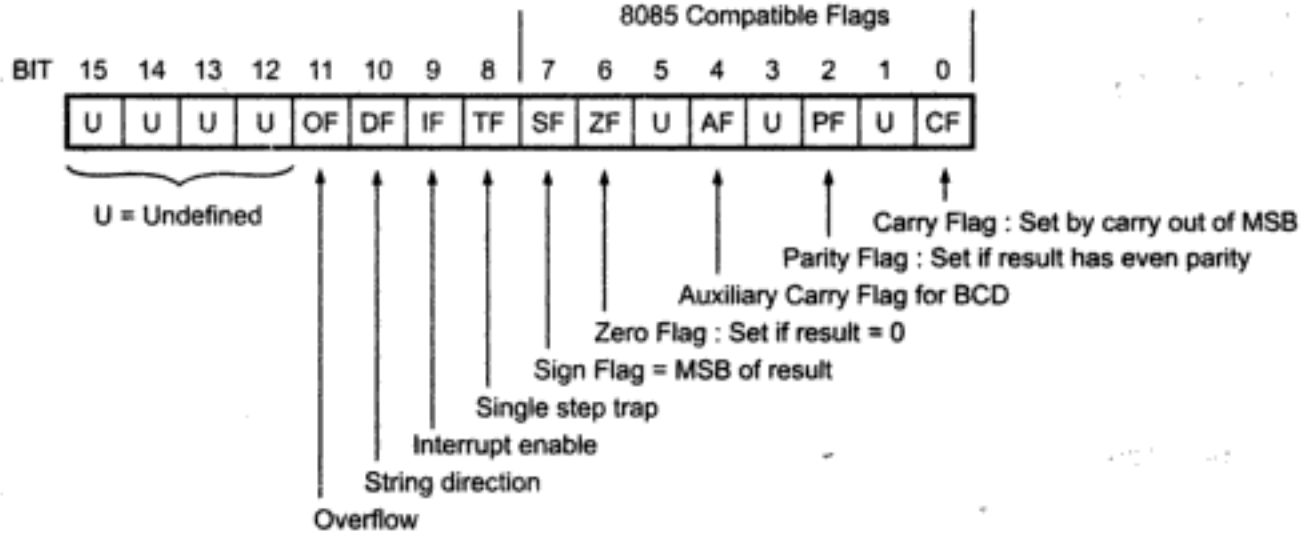


Fig. 2.6 8086 flag register bit pattern

Six of them are used to indicate some condition produced by instruction.

1. Carry Flag (CF) : In case of addition this flag is set if there is a carry out of the MSB. The carry flag also serves as a borrow flag for subtraction. In case of subtraction it is set when borrow is needed.

2. Parity Flag (PF) : It is set to 1 if result of byte operation or lower byte of the word operation contain an even number of ones; otherwise it is zero.

3. Auxiliary Flag (AF): This flag is set if there is an overflow out of bit 3 i.e., carry from lower nibble to higher nibble (D_3 bit to D_4 bit). This flag is used for BCD operations and it is not available for the programmer.

4. Zero Flag (ZF) : The zero flag sets if the result of operation in ALU is zero and flag resets if the result is nonzero. The zero flag is also set if a certain register content becomes zero following an increment or decrement operation of that register.

5. Sign Flag (SF): After the execution of arithmetic or logical operations, if the MSB of the result is 1, the sign bit is set. Sign bit 1 indicates the result is negative; otherwise it is positive.

6. Overflow Flag (OF): This flag is set if result is out of range. For addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice-versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice-versa.

➠ **Example 1 :** *Give the contents of the flag register after execution of following addition.*

$$\begin{array}{r} 0110\ 0101\ 1101\ 0001 \\ +\ 0010\ 0011\ 0101\ 1001 \\ \hline 1000\ 1001\ 0010\ 1010 \end{array}$$

Solution : SF = 1, ZF = 0, PF = 1, CF = 0, AF = 0 , OF = 1

➠ **Example 2 :** *Give the contents of the flag register after execution of following subtraction*

$$\begin{array}{r} 0110\ 0111\ 0010\ 1001 \\ -\ 0011\ 0101\ 0100\ 1010 \\ \hline 0011\ 0001\ 1101\ 1111 \end{array}$$

Solution : SF = 0, ZF = 0, PF = 1, CF = 0, AF = 1, OF = 0

The three remaining flags are used to control certain operations of the processor.

1. Trap Flag (TF): One way to debug a program is to run the program one instruction at a time and see the contents of used registers and memory variables after execution of every instruction. This process is called '**single stepping**' through a program. Trap flag is used for single stepping through a program. If set, a trap is executed after execution of each instruction, i.e. interrupt service routine is executed which displays various registers and memory variable contents on the display after execution of each instruction. Thus programmer can easily trace and correct errors in the program.

2. Interrupt Flag (IF) : It is used to allow/prohibit the interruption of a program. If set, a certain type of interrupt (a maskable interrupt) can be recognized by the 8086; otherwise, these interrupts are ignored.

3. Direction Flag (DF) : It is used with string instructions. If DF = 0, the string is processed from its beginning with the first element having the lowest address. Otherwise, the string is processed from the high address towards the low address.

2.4 Bus Operation

The 8086 has a common address and data bus. The address and data are time multiplexed, i.e. address and data appear on this bus at different time intervals. Thus bus is commonly known as multiplexed address and data bus. The multiplexed address and data bus provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This multiplexed address and data bus has to be demultiplexed externally with the use of latches and the ALE signal provided by 8086. This bus can be buffered directly and used throughout the system with address latching provided on memory and I/O modules or it can be demultiplexed at the processor with a single set of address latches if a standard non-multiplexed bus is desired for the system.

The control operation of 8086 is different in two different modes : minimum mode and maximum mode. The 8086 provides some signals which have different meanings in minium mode and maximum mode. The minimum mode is used for a small systems with a single processor and maximum mode is for medium size to large systems, which often include two or more processors.

2.5 Memory Segmentation

Two types of memory organisations are commonly used. These are **linear addressing** and **segmented addressing**. In linear addressing the entire memory space is available to the processor in one linear array. In the segmented addressing, on the other hand, the available memory space is divided into "chunks" called segments. Such a memory is known as **segmented memory**. In 8086 system the available memory space is 1Mbytes. This memory is divided into number of logical segments. Each segment is 64 K bytes in size and addressed by one of the segment registers. The 16-bit contents of the segment register gives the starting/base address of a particular segment, as shown in Fig. 2.7. To address a specific memory location within a segment we need an offset address. The offset address is also 16-bit wide and it is provided by one of the associated pointer or index register.

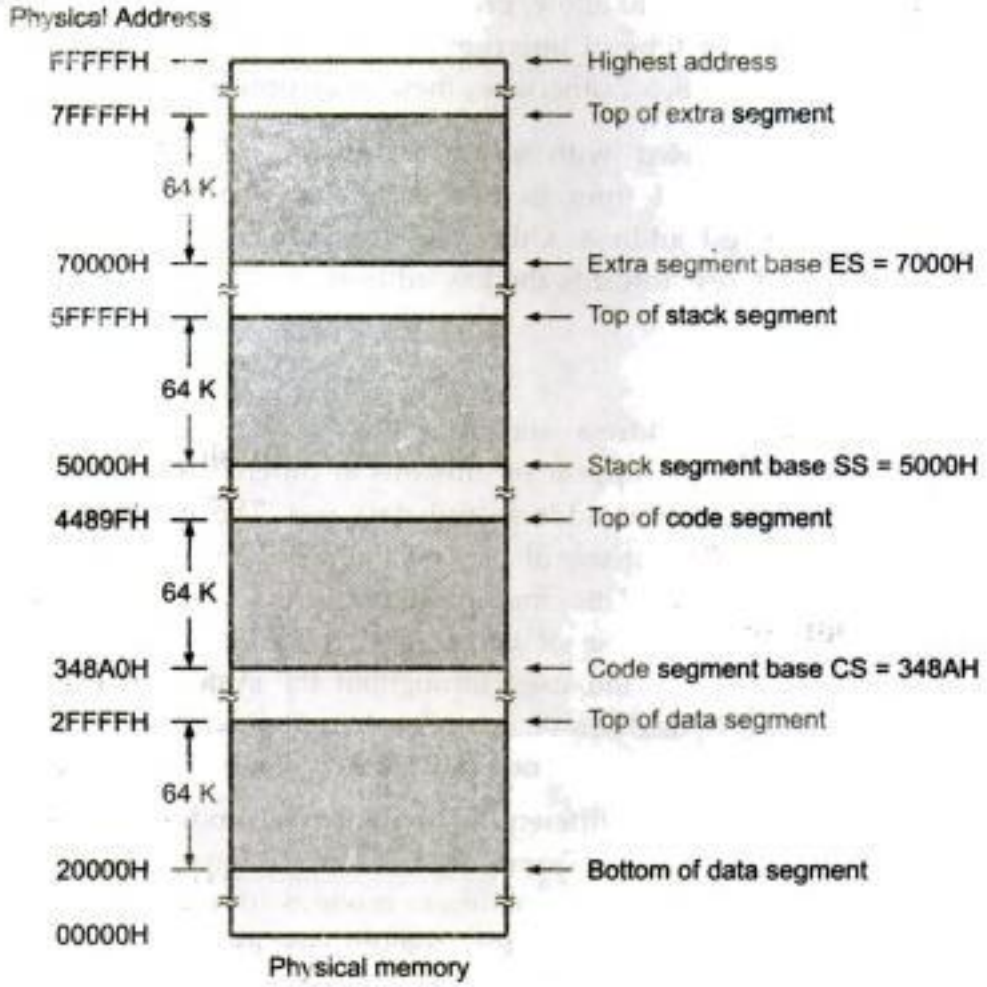


Fig. 2.7 Memory segmentation

Rules for Memory Segmentation

1. The four segments can overlap for small programs. In a minimum system all four segments can start at the address 00000H.
2. The segment can begin/start at any memory address which is divisible by 16.

Advantages of Memory Segmentation

1. It allows the memory addressing capacity to be 1 Mbyte even though the address associated with individual instruction is only 16-bit.
2. It allows instruction code, data, stack, and portion of program to be more than 64 KB long by using more than one code, data, stack segment, and extra segment.
3. It facilitates use of separate memory areas for program, data and stack.
4. It permits a program or its data to be put in different areas of memory, each time the program is executed i.e. program can be relocated which is very useful in multiprogramming.

Generation of 20-bit Address

To access a specific memory location from any segment we need 20-bit physical address. The 8086 generates this address using the contents of segment register and the offset register associated with it. Let us see how 8086 access code byte within the code segment.

We know that the CS register holds the base address of the code segment. The 8086 provides an instruction pointer (IP) which holds the 16-bit address of the next code byte within the code segment. The value contained in the IP is referred to as an **offset**. This value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address.

The contents of the CS register are multiplied by 16. i.e. shifted by 4 position to the left by inserting 4 zero bits and then the offset i.e. the contents of IP register are added to the shifted contents of CS to generate physical address. As shown in the Fig. 2.8, the contents of CS register are 348AH, therefore the shifted contents of CS register are 348A0H. When the BIU adds the offset of 4214H in the IP to this starting address, we get 38AB4H as a 20-bit physical address of memory. This is illustrated in Fig. 2.8 (b).

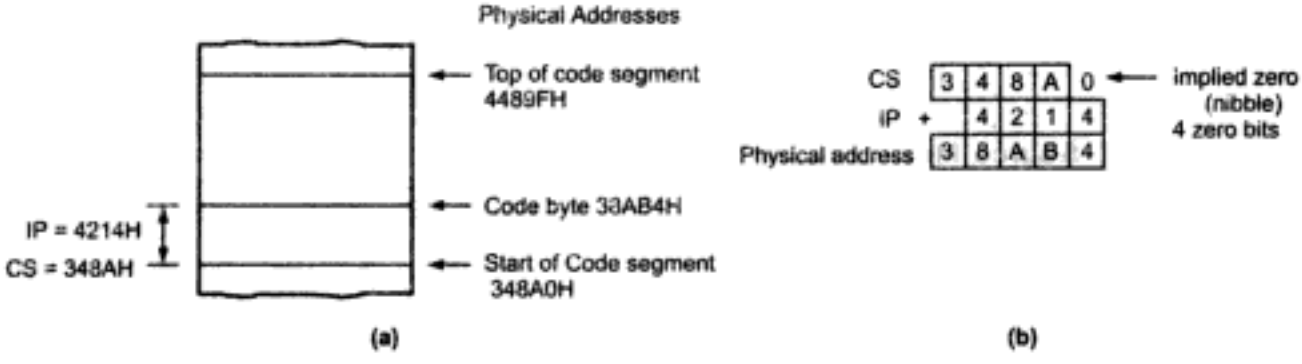


Fig. 2.8

We have seen that how 20-bit physical address is generated within the code segment. In the similar way the 20-bit physical address is generated in the other segments. However, it is important to note that each segment requires particular segment register and offset register to generate 20-bit physical address.

Pointers and Index Registers

All segment registers are 16-bit. But it is necessary to put 20-bit address (physical address) on the address bus. To get 20-bit physical address one more register is associated with each segment register the way IP is associated with CS.

These additional registers belong to the pointer and index group. The pointer and index group consists of instruction pointer (IP), stack pointer (SP), BP (base pointer), source index (SI) and destination index (DI) registers.

Stack Pointer (SP) : The stack pointer (SP) register contains the 16-bit offset from the start of the segment to the top of stack. For stack operation, physical address is produced by adding the contents of stack pointer register to the segment base address in SS. To do this the contents of the stack segment register are shifted four bits left and the contents of SP are added to the shifted result. If the contents of SP are 9F20H and SS are 4000H then the physical address is calculated as follows. (Refer Fig. 2.9)

SS = 4000H after shifting four bits left SS = 40000H

Now

SS	40000H
+ SP	9F20H
Physical address	49F20H

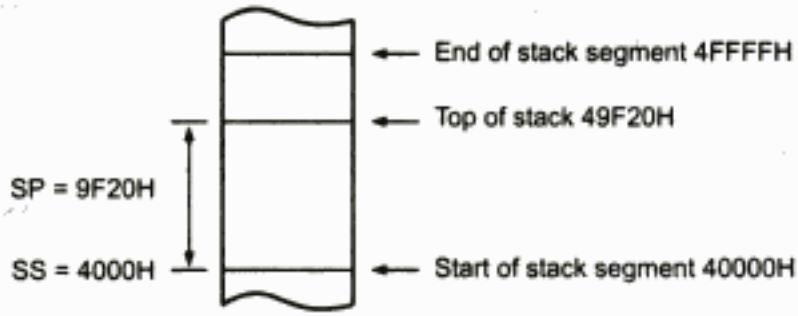


Fig. 2.9 Stack and stack pointer

Base Pointer, Source Index and Destination Index (BP, SI and DI)

These three 16-bit registers can be used as general purpose registers. However, their main use is to hold the 16-bit offset of the data word in one of the segments.

Base pointer : We can use the BP register instead of SP for accessing the stack using the based addressing mode. In this case, the 20-bit physical stack address is calculated from BP and SS. Addressing modes are discussed in later section.

Source Index : Source index (SI) can be used to hold the offset of a data word in the data segment. In this case, the 20-bit physical data address is calculated from SI and DS.

Destination Index : The ES register points to the extra segment in which data is stored. String instructions always use ES and DI to determined the 20-bit physical address for the destination.

Default and Alternate Register Assignments

Table 2.1 shows that some memory references and their default and alternate segment definitions. For example, instruction codes can only be stored in the code segment with IP used as an offset. Similarly, for stack operations only SS and SP or BP registers can be used to give segment and offset addresses respectively. On the other hand, for accessing general data, string source, data pointed by BX and BP registers; it is possible to use alternate segments by using segment override prefix. See examples given after Table 2.1.

Type of Memory Reference	Default Segment	Alternate Segment	Offset (Logical Address)
Instruction fetch	CS	None	IP
Stack operation	SS	None	SP, BP
General data	DS	CS, ES, SS	Effective address
String source	DS	CS, ES, SS	SI
String destination	ES	None	DI
BX used as pointer	DS	CS, ES, SS	Effective address
BP used as pointer	SS	CS, ES, DS	Effective address

Table 2.1 Default and alternate register assignments

For the following examples we have assumed

CS = 1000H, DS = 2000H, SS = 3000H, ES = 4000H, BP = 0010 H,

BX = 0020H, SP = 0030H, SI = 0040H, DI = 0050H

Example 3 :

1) MOV AL, [BP]

```
        3000[0] H   SS
      +   001 0 H   BP
      ------------
Physical Address  3001 0 H
```

This instruction copies a byte from memory location to the AL register. The effective address for the memory location is contained in the BP register. By default, an effective address is added to the stack segment (SS) to produce the physical memory address (30010 H).

2) MOV CX, [BX]

```
        2000[0] H   DS
      +   002 0 H   BX
      ------------
Physical Address  2002 0 H
```

This instruction copies a word from memory location to the CX register. The effective address is contained in the BX register. By default an effective address is added to the data segment (DS) to produce the physical memory address (20020 H).

3) MOV AL, [BP+SI]

```
          001 0 H   BP
        +   004 0 H   SI
        ------------
Effective Address   005 0 H

        3000[0] H   SS
      +   005 0 H   EA
      ------------
Physical Address  3005 0 H
```

This instruction copies a byte from memory location to the AL register. The effective address is the summation of the contents of the BP and SI register.

The effective address is added to the stack segment (SS) to get the physical address.

4) MOV CS : [BX], AL

	1000 [0] H	CS
+	0020 H	BX
Physical Address	10020 H	

This instruction copies a byte from the AL register to a memory location. The effective address for the memory location is contained in the BX register. By default an effective address in BX will be added to the data segment (DS) to produce the physical memory address. In this instruction, the CS: in front of [BX] indicates that we want BIU to add the effective address to the code segment (CS) to produce the physical address. The CS: is called **segment override prefix.**

Segment Override Prefix

The segment override prefix allows the programmer to deviate from the default segment. The segment override prefix is an additional 8-bit code which is put in memory before the code for the rest of the instruction. This additional code selects the alternate segment register. The code byte for the segment override prefix as the format 001XX110. The XX represents a 2 bits which are as follows : ES = 00, CS = 01, SS = 10 and DS = 11. It is important to note that the segment override prefix may be added to almost any instruction in any memory addressing mode.

Review Questions

1. *List the features of 8086 microprocessor.*
2. *Explain the architecture of 8086 processor with the help of neat block diagram.*
3. *What is the function of bus interfacing unit ?*
4. *What is the instruction queue ? Explain its advantage.*
5. *What is pipelining ?*
6. *Explain the register organisation of 8086.*
7. *What are segment registers ? Explain the purpose of them.*
8. *Explain the purpose of pointers and index registers.*
9. *What is the function of flag register ?*
10. *How physical address is generated in 8086 ?*
11. *Draw the bit pattern for flag register of 8086 and explain the significance of each bit.*
12. *List the rules for memory segmentation.*
13. *What are the advantages of using memory segmentation ?*
14. *What do you mean by index registers ?*
15. *What are the functions of SI and DI registers ?*

❑❑❑

3 8086 Instruction Set and Assembly Language Programming

3.1 Introduction

The 8086 instruction set includes equivalents of the 8085 instructions plus many new ones. The new instructions contain operations such as signed/unsigned multiplication and division, bit manipulation instructions, string instructions, and interrupt instructions.

The 8086 has approximately 117 different instructions with about 300 opcodes. The 8086 instruction set contains no operand, single operand, and two operand instructions. Except for string instructions which involve array operations, the 8086 instructions do not permit memory to memory operations.

In this chapter we study the addressing modes, instruction set of 8086 and assembler directives.

3.2 Addressing Modes

We have seen how the 8086 fetches code bytes from memory by generating 20-bit physical address with the help of IP and CS. We have also seen how the 8086 accesses the stack using SS and SP. In this section we will see the different ways that an 8086 can access the data. The different ways that a processor can access data are referred to as **addressing modes**.

The addressing modes of any processor can be broadly classified as :

- Data addressing modes.
- Program memory addressing modes.
- Stack memory addressing modes.

3.2.1 Data Addressing Modes

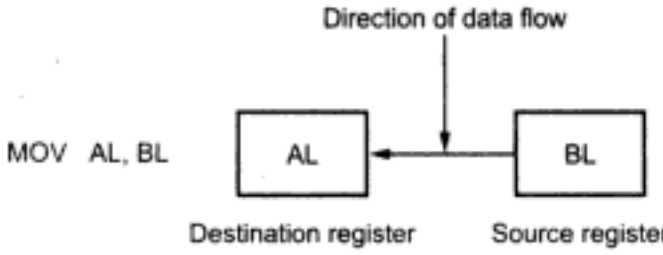
The data addressing modes can be further classified as

1. Addressing modes for accessing immediate and register data (register and immediate modes).
2. Addressing modes for accessing data in memory (memory modes).
3. Addressing modes for accessing I/O ports (I/O modes).

Addressing Modes for Accessing Immediate and Register Data

1. Register Addressing Mode

This mode specifies the source operand, destination operand, or both to be contained in an 8086 register.



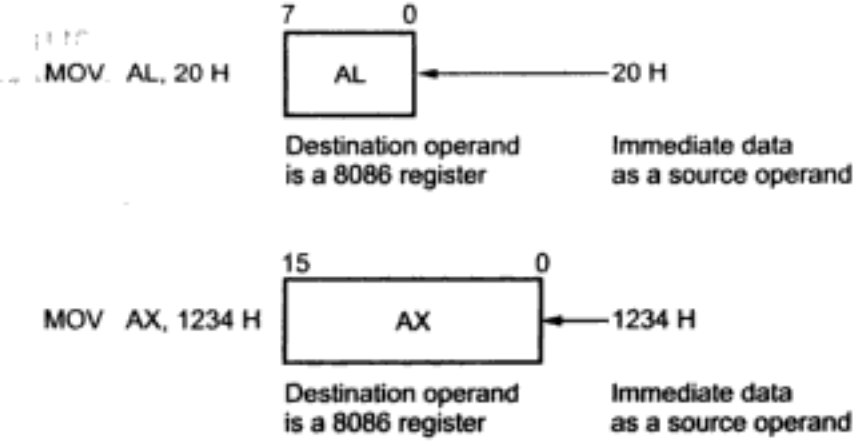
Note : Both source and destination operands are in 8086 register

Examples :

```
MOV BX, CX      ; Copies the 16-bit contents of CX into BX
MOV CL, BL      ; Copies 8-bit contents of BL into CL.
```

2. Immediate Addressing Mode

In an immediate mode, 8 or 16-bit data can be specified as a part of instruction.



Note : Arrow indicates direction of data flow

Examples :

```
MOV BL, 26H     ; Copies the 8-bit data 26H into BL
MOV CX, 4567H   ; Copies the 16-bit data 4567H into CX.
```

Addressing Modes for Accessing Data in Memory

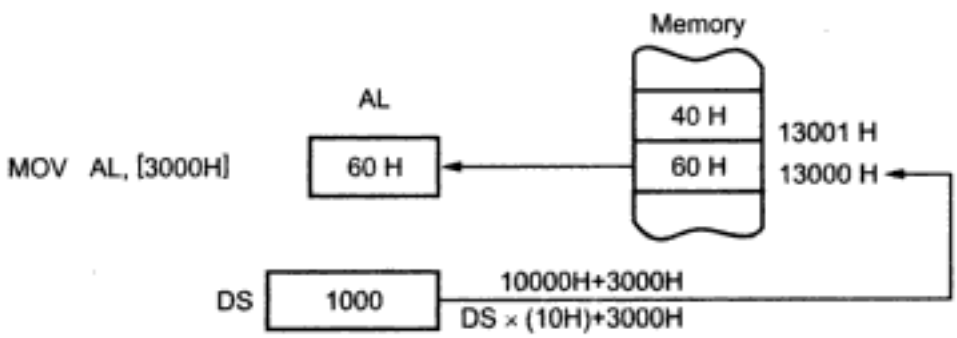
As mentioned before, the Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However, the EU cannot directly access the memory operands. It must use the BIU segment registers to access memory operands. For example, when the EU needs to access a memory location, it sends an offset value to the BIU. This offset is also called the **Effective Address (EA).** Note that EA is displacement of the desired location from the segment base. As mentioned before, the BIU generates a 20-bit physical address after shifting the contents of the desired segment register four bits to the left and then adding the 16-bit EA to it.

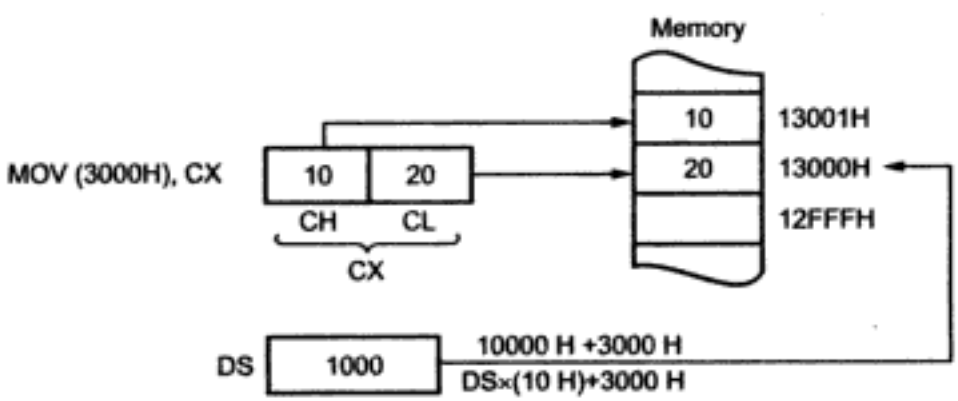
There are six ways to specify effective address (EA) in the instruction.

a. Direct addressing mode
b. Register indirect addressing mode
c. Based addressing mode
d. Indexed addressing mode
e. Based indexed addressing mode
f. String addressing mode.

1. Direct Addressing Mode :

In this mode, the 16-bit effective address (EA) is taken directly from the displacement field of the instruction. The displacement (unsigned 16-bit or sign-extended 8-bit number) is stored in the location following the instruction opcode.





Note : 1. Assume DS = 1000

$\therefore$ Physical address = DS ×(10H)+ 3000H
= 1000[0]+ 3000H = 13000H

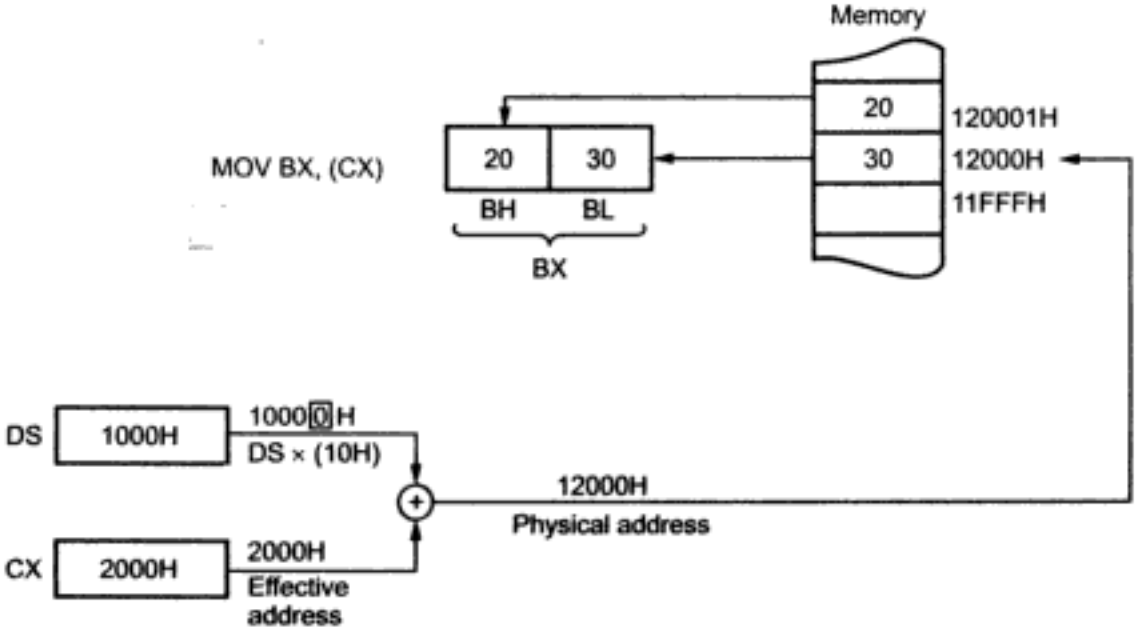
2. Arrow indicates direction of data flow.

Example :

```
MOV CL, [9823H]    ; This instruction will copy the contents of the
                   ; memory location, at a displacement of 9823H from the
                   ; data segment base, into the CL register. Here, 9823H is
                   ; the effective address (EA) which is written
                   ; directly in the instruction.
```

2. Register Indirect Addressing Mode

In this mode, the EA is specified in either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either Source Index (SI) register or Destination Index (DI) register. The 20-bit physical address is computed using DS and EA.



Example :

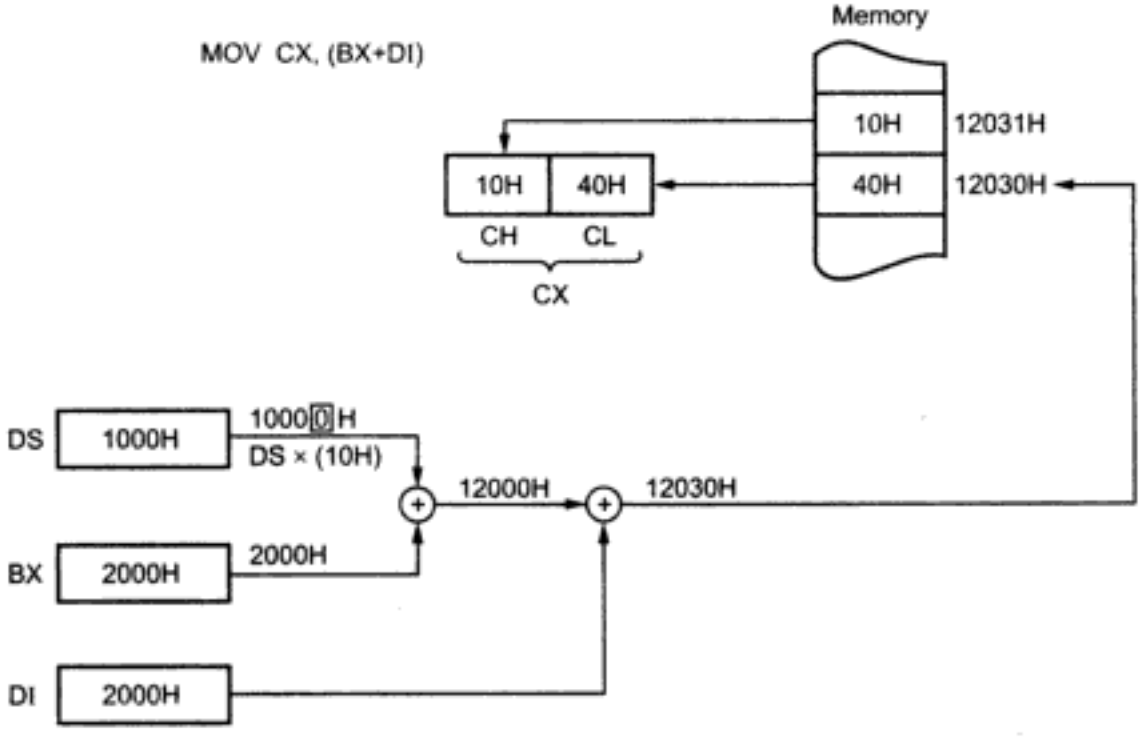
```
1. MOV [DI], BX    ; The instruction copies the 16-bit contents of BX into a
                   ; memory location offset by the value of EA specified in DI
                   ; from the current contents in DS. Now, if [DS] = 7205H,
                   ; [DI] = 0030H, and [BX] = 8765H, then after MOV [DI], BX,
                   ; content of BX (8765H) is copied to memory locations
                   ; 72080H and 72081H.
```

```
2. MOV DL, [BP]     ; This instruction copies the 8-bit
                    ; contents in DL from the memory location offset by the
                    ; value of EA specified in BP from the contents of SS.
                    ; Because data addressed by BP are by default located in
                    ; stack segment (SS).
```

3. Base-Plus-Index-Addressing :

Base-plus-index addressing is similar to indirect addressing because it indirectly addresses memory data. This addressing uses one base register (BP or BX), and one index register (DI or SI) to indirectly address memory. The base register often holds the beginning location of a memory array, while the index register holds the relative position of an element in the array. Remember that whenever BP addresses the memory data, the contents of stack segment, BP and index register are used to generate physical address.

Locating Data with Base-Pulse-Index Addressing :



Locating Array Data Using Base-Plus-Index Addressing :

A main use of the base-plus-index addressing mode is to address elements in a memory array. Suppose that the array is located in the data segment beginning from memory location ARRAY. To access a particular element within the array we have to load the BX register (base) with the beginning address of the array, and the DI register (index) with the element number to be accessed. This is illustrated in Fig. 3.1.

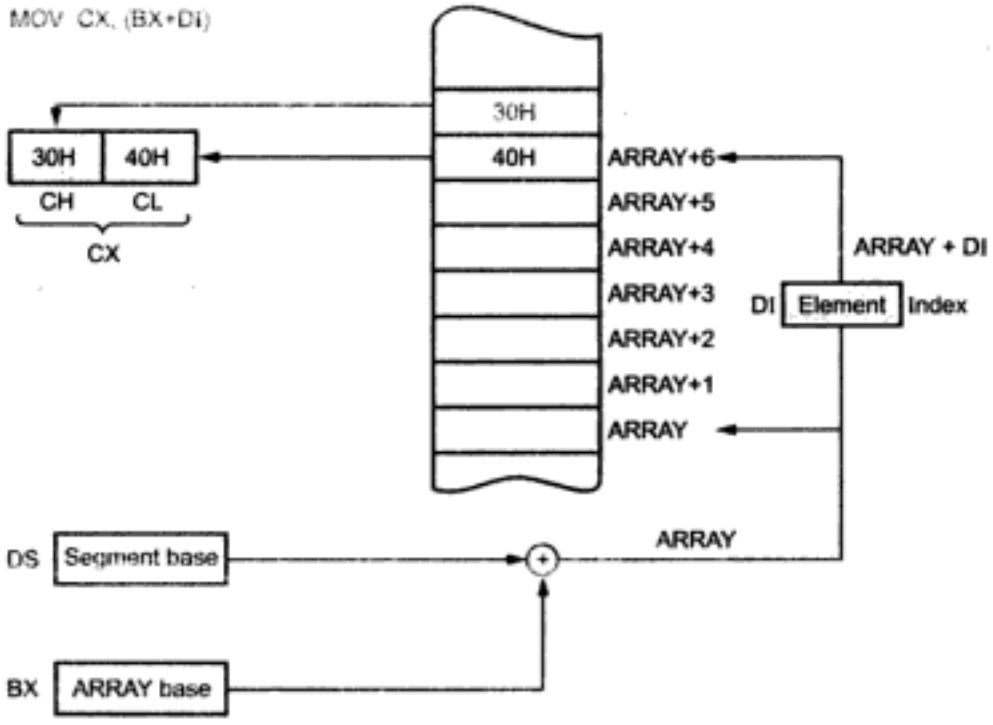


Fig. 3.1

4. Register Relative Addressing :

Register relative addressing is similar to base-plus-index addressing. Here, the data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register (BP, BX, DI or SI). Remember that displacement should be added to the register within the [ ]. This is illustrated in the Fig. 3.2. Displacement can be any 8-bit or 16-bit number.

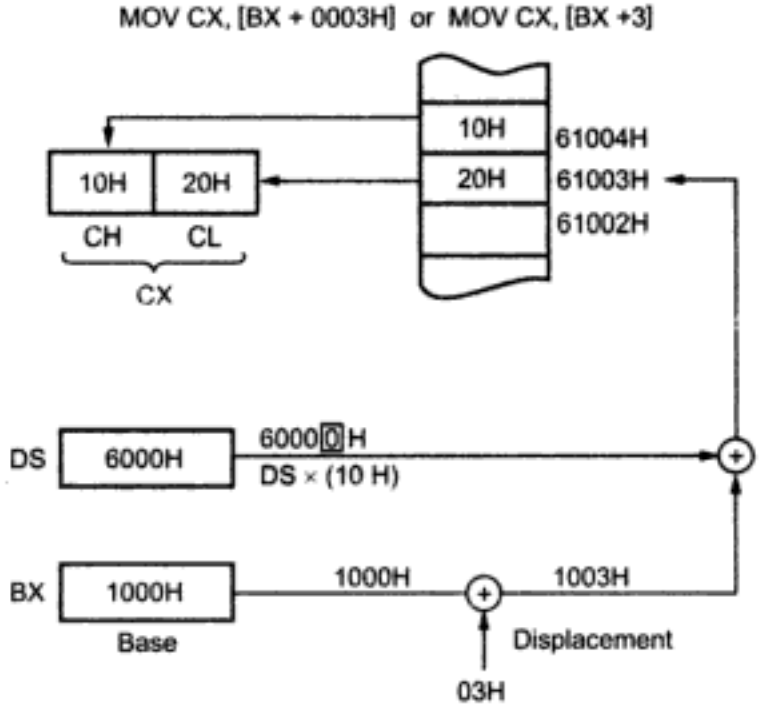


Fig. 3.2

Note :

- Displacement can be subtracted from the register : MOV AL, [DI–2].
- Displacement can be an offset address appended to the front of the [ ] : MOV AL, OFF_ADD [DI + 4].

Example : MOV AL, LAST [SI + 2] ; This instruction copies the contents of the 20-bit address computed from the displacement LAST, SI + 2 and DS into AL.

Addressing Array Data with Register Relative :

The Fig. 3.3 shows how to address data element within the array with register relative addressing.

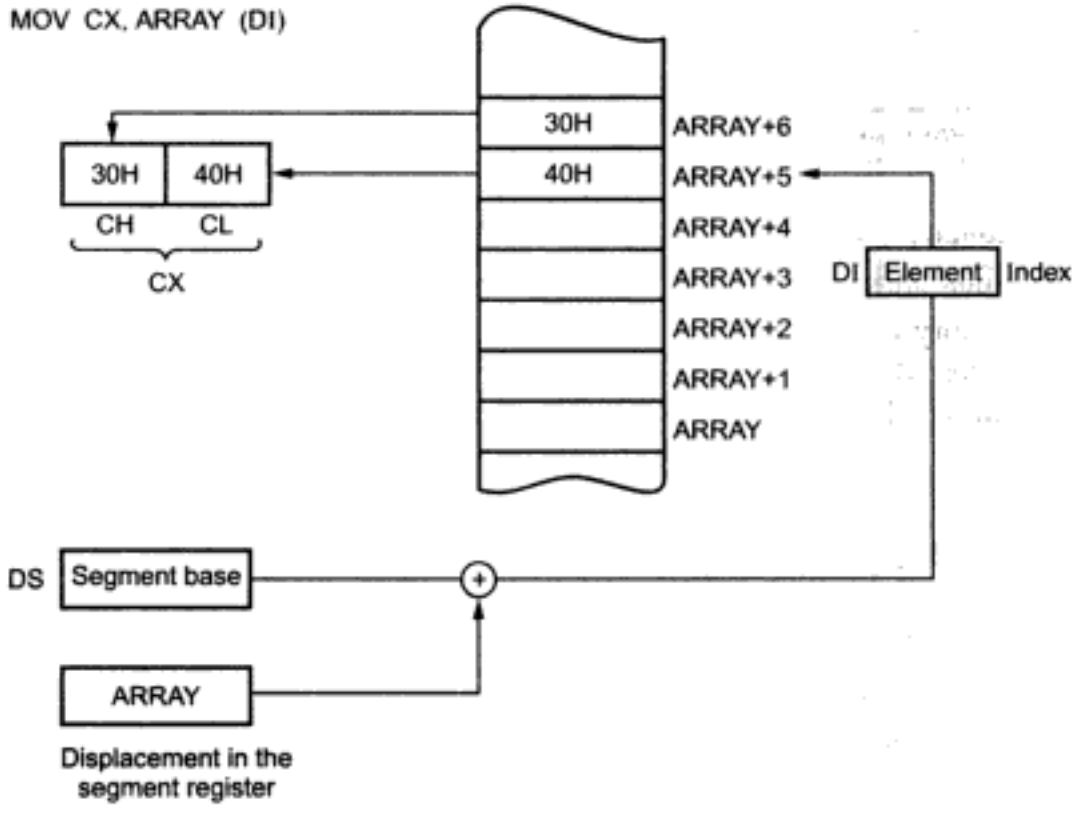


Fig. 3.3

5. Base Relative Plus Index Addressing :

The base relative plus index addressing mode is similar to the base plus index addressing mode, but it adds a displacement, besides using a base register and an index register to generate a physical address of the memory. This addressing mode is suitable to address data within the two dimensional array.

Addressing Data with Base Relative Plus Index :

The Fig. 3.4 shows how data can be accessed with base relative plus index addressing mode.

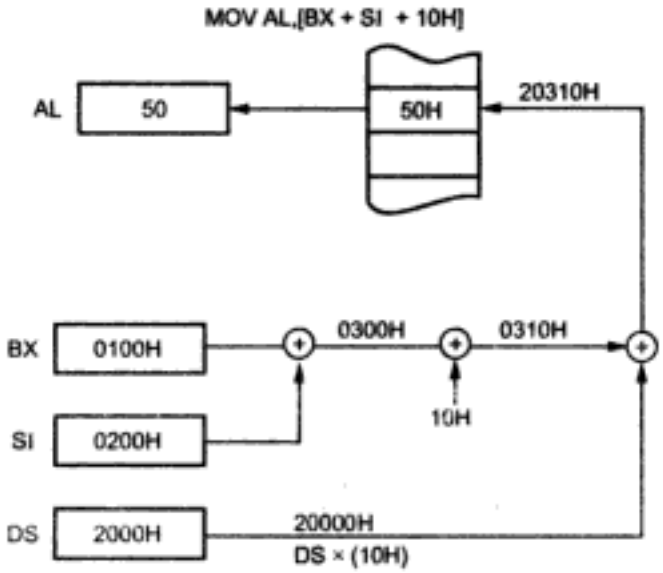


Fig. 3.4

Addressing Arrays with Base Relative-Plus-Index :

As mentioned earlier this addressing mode is useful in addressing two dimensional array. Two dimensional array usually stores records. For example, student record such as its name, roll no etc. Therefore, each record contains number of data elements. To access data element from a particular record we use base register to hold the beginning address of the array of records, index register to point a particular record in the array of records and displacement to point a particular element in the record. This is illustrated in Fig. 3.5.

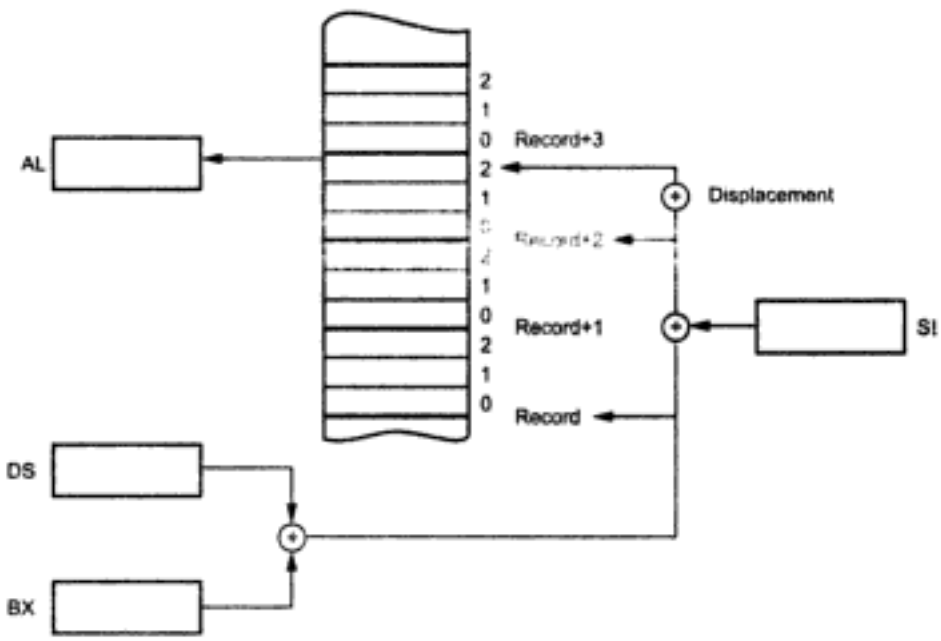


Fig. 3.5

6. String Addressing Mode :

This mode uses index registers. The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) or decremented (by setting DF to 1 by STD instruction) to point to the next byte or word. The segment register for the source is DS. The segment register for the destination must be ES.

Example :

```
MOVS BYTE       ; If [DF] = 0, [DS] = 3000H, [SI] = 0600H, [ES] =5000H,
                ; [DI] = 0400H, [30600H] = 38H, and [50400H] = 45H, then
                ; after execution of the MOVS BYTE, [50400H] = 38H,
                ; [SI] = 0601H, and [DI] = 0401H.
```

Addressing Modes for Accessing I/O Ports (I/O Modes)

Standard I/O devices uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes : direct and indirect.

In direct port mode, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255.

Example :

```
OUT 05H, AL     ; Sends the contents of AL to 8-bit port 05H.
IN AX, 80H      ; Copies 16-bit contents of port 80H
```

In indirect port mode, the port number is taken from DX allowing 64K 8-bit ports or 32K 16-bit ports.

Example :

```
IN AL, DX       ; If [DX] = 7890H, then it copies 8-bit content of port 7890H
                ; into AL.
IN AX, DX       ; Copies the 8-bit contents of ports 7890H and 7891H into AL
                ; and AH, respectively.
```

Note : The 8-bit and 16-bit I/O transfers must take place via AL and AX, respectively.

3.2.2 Program Memory Addressing Modes

JMP (Jump) and CALL instructions use program memory addressing modes. These instruction have three distinct forms : direct, relative and indirect. Let us see these forms and corresponding addressing modes.

Direct program memory addressing :

In this addressing mode address where to transfer program control is specified within the instruction alongwith the opcode. The Fig. 3.6 shows the direct intersegment JMP instruction and the four bytes required to store the address 20000H. This JMP instruction loads CS with 2000H and IP with 0000H to jump to memory location 20000H for the next instruction. An **intersegment jump** is a jump where destination location is from a different segment; it can be any memory location within the entire memory locations. Therefore, intersection jump is also known as **far jump.**

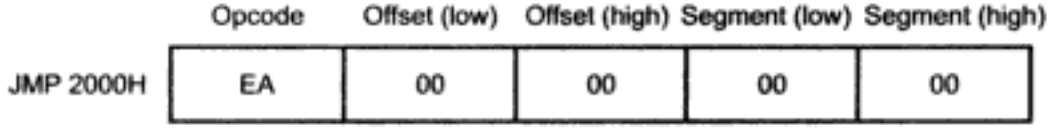


Fig. 3.6

Like JMP instruction, CALL instruction also uses direct program addressing with intersegment or far CALL instruction. Usually, in both instructions (JMP or CALL) the name of a memory address, called a **label** is specified in the instruction instead of address.

Relative program memory addressing :

In this addressing mode, the term relative is restricted to instruction pointer (IP). For example, if a JMP instruction skips the next 5 bytes of memory, the address in relation to the instruction pointer is a 5 that adds to the instruction pointer. This generates the address of the next program instruction. This is illustrated in Fig. 3.7.

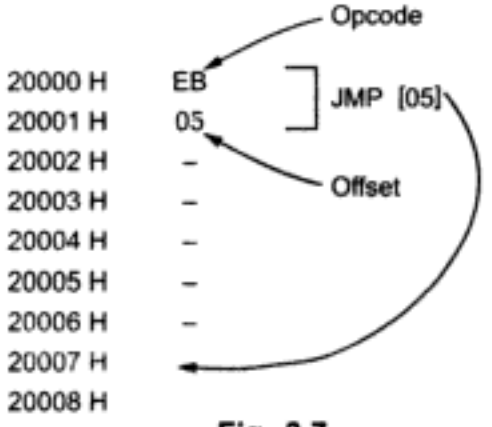


Fig. 3.7

It is important to note that in JMP instruction, opcode takes one byte and displacement may take one or two byte. When displacement is one byte (8-bit), it is called **short jump.** When displacement is two byte (16-bit), it is called **near jump.** In both (short and near) cases only contents of IP register are modified; contents of CS register are not modified. Such jumps are called **intrasegment jumps** because jumps are within the current code segment.

The relative JMP and CALL instructions can have either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference.

Indirect program memory addressing :

The 8086 allows several forms of program indirect memory addressing for the JMP and CALL instructions. In this addressing mode, it is possible to use any 16-bit register (AX, BX, CX, DX, SP, BP, DI or SI); any relative register ([BP], [BX], [DI], or [SI]); and any relative register with displacement to specify the jump address. This is illustrated in Table 3.1.

Instruction	Operation
JMP BX	Jumps to memory location addressed by BX within current code segment. IP ← BX
JMP NEAR PTR [BX]	Jumps to memory location addressed by the contents of the data segment memory location addressed by BX within the current code segment. IP ← ([BX + 1], [BX]) High byte Low byte
JMP NEAR PTR [DI + 2]	Jumps to memory location addressed by the contents of the data segment memory location addressed by DI plus 2 within the current code segment. IP ← ([DI + 3], [DI + 2]) High byte Low byte
JMP ARRAY [BX]	Jumps to memory location addressed by the contents of the data segment memory location addressed by ARRAY plus BX with the current code segment. IP ← ([ARRAY + BX + 1], [ARRAY + BX]) High byte Low byte

Table 3.1

3.2.3 Stack Memory Addressing Modes

The stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. When the information is written on the stack, the operation is called PUSH. When the information is read from stack, the operation is called a POP.

The microprocessor stores the information, much like stacking plates. Using this analogy of stacking plates it is easy to illustrate the stack operation.

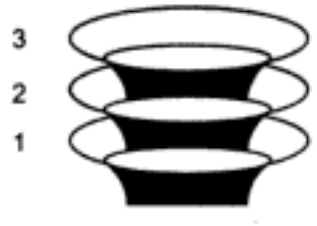


Fig. 3.8 Stacked plates

Fig. 3.8 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on to the stack is the last information popped off from the stack. This type of operation is known as a first in, last out (FILO). This stack is implemented with the help of special memory pointer register.

The special pointer register is called the **stack pointer.** During PUSH and POP operation, stack pointer register gives the address of memory where the information is to be stored or to be read. The stack pointer's contents are automatically manipulated to point to stack top. The memory location currently pointed by stack pointer is called **top of stack.**

Stack Structure of 8086/88

The 8086/88 has a special 16-bit register, SP to work as a stack pointer. The stack pointer (SP) register contains the 16-bit offset from the start of the segment to the top of stack. For stack operation, physical address is produced by adding the contents of stack pointer register to the segment base address in SS. To do this the contents of the stack segment register are shifted four bits left and the contents of SP are added to the shifted result. If the contents of SP are 9F20H and SS are 4000H then the physical address is calculated as follows. (Refer Fig. 3.9)

SS = 4000H after shifting four bits left SS = 40000H

Now

SS	40000H
+ SP	9F20H
Physical address	49F20H

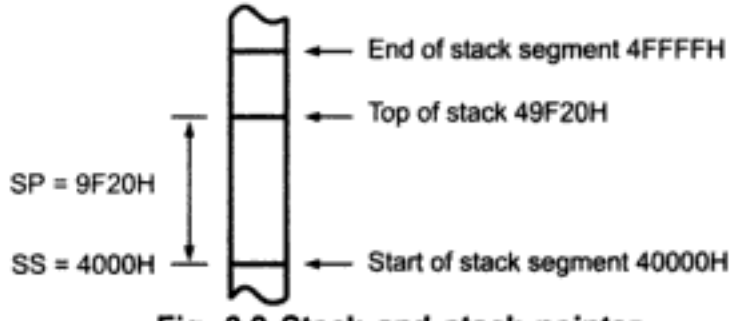


Fig. 3.9 Stack and stack pointer

PUSH and POP Operations

Temporarily stores the contents of 16-bit register or memory location or program status word, and retrieves when required. When programmer realizes the shortage of the registers, he stores the present contents of the registers in the stack with the help of PUSH instruction and then uses the registers for other function. After completion of other function programmer loads the previous contents of the register from the stack with the help of POP instruction.

PUSH Operation :

The PUSH instruction decrements stack pointer by two and copies a word from some source to the location in the stack where the stack pointer points. Here the source must be a **word** (16 bit). The source of the word can be a general purpose register, a segment register or memory. The Fig. 3.10 shows the map of the stack before and after execution of PUSH AX and PUSH CX instructions.

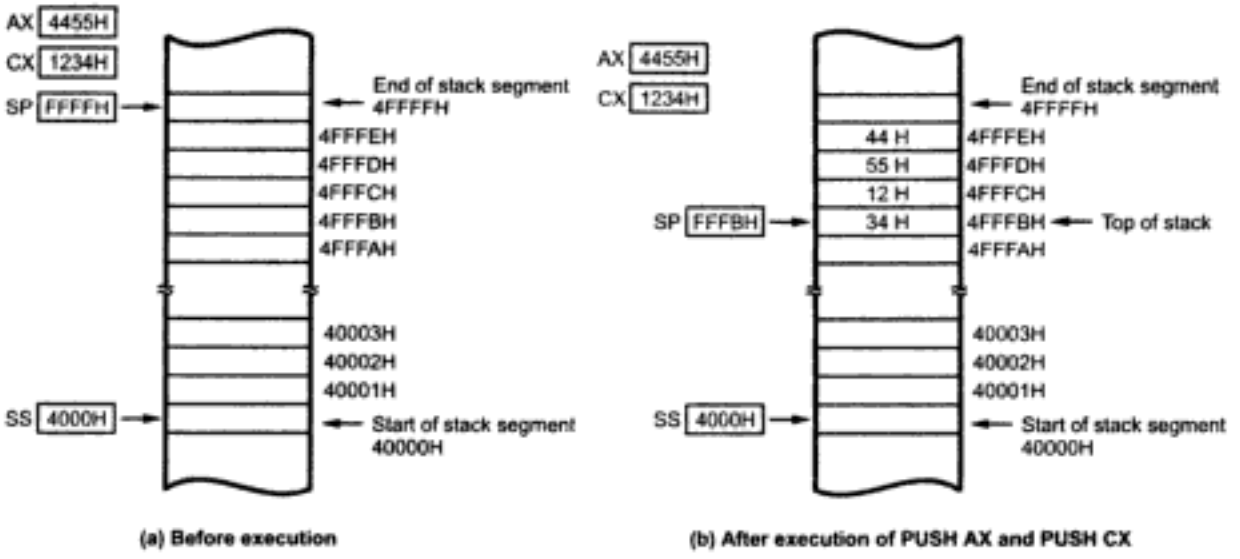


Fig. 3.10

POP Operation :

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a general purpose register, a segment register, or a memory location. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2. The Fig. 3.11 shows the map of the stack before and after execution of POP DX and POP BX instructions.

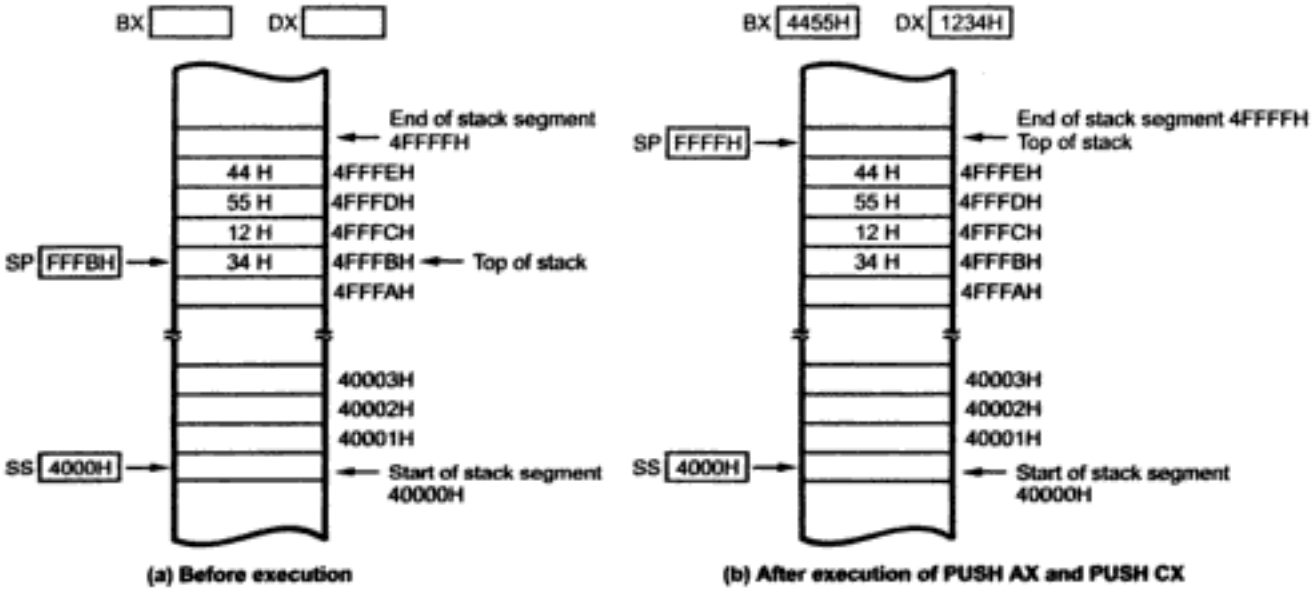


Fig. 3.11

CALL Operation

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A near CALL is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a

near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A far CALL is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

RET Operation

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

Overflow and Underflow of Stack

We have seen the PUSH operation. During this operation stack pointer is decremented by two. We know that maximum length of stack segment is 64K. If we go on performing PUSH operations successively, at one time the contents of SP will be 0000H. Any further attempt to PUSH data on the stack will result in **stack overflow**.

On the other hand, if we go on performing POP operations successively, at one time the contents of SP will be FFFFH. Any further attempt to POP data from the stack will result in **stack underflow**.

3.3 Instruction Set of 8086/8088

The instruction set of the 8086 is divided into Eight major groups as follows :

- Data Movement Instructions
- Arithmetic and Logic Instructions
- String Instructions and
- Program Control Transfer Instructions
- Iteration Control Instructions
- Processor Control Instructions

- External Hardware Synchronization Instructions
- Interrupt Instructions

3.4 Data Movement Instructions

The data movement instructions can be classified as

- MOV instructions to transfer byte or word.
- PUSH/POP instructions.
- Load effective address instructions.
- String data transfer instructions.
- Miscellaneous data transfer instructions.

3.4.1 MOV Instruction

It is a general purpose instruction to transfer byte or word from register to register, register to memory or from memory to register.

MOV destination, source

The MOV instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number. The source and destination in an instruction can't both be memory locations. The source and destination in a MOV instruction must be of same type i.e. either both must be byte or word.

MOV instruction does not affect any flags.

Examples :

```
MOV BX, 592FH          ; Load the immediate number 592FH in BX
MOV CL, [357AH]        ; Copy the contents of memory location, at a
                       ; displacement of 357AH from data segment base,
                       ; into the CL register.
MOV [734AH], BX        ; Copy the contents of BX register to two memory
                       ; locations in the data segment. Copy the contents
                       ; of BL register to memory location at a
                       ; displacement of 734AH and BH register
                       ; to memory location at a displacement of
                       ; 734BH.
MOV DS, CX             ; Copy word from CX register to data
                       ; segment register.
MOV TOTAL [BP], AX     ; Copy AX to two memory locations. AL to
                       ; first location, AH to second. Effective
                       ; address, EA, is the sum of displacement
```

```
                        ; represented by TOTAL and contents of BP.
                        ; Physical address = EA + SS.
MOV CS : TOTAL [BP], AX ; Same as above instruction, but physical
                        ; address = EA+CS. Because the segment
                        ; overide prefix is CS.
```

3.4.2 PUSH/POP Instructions

These instructions are used to load or receive data from the stack memory.

PUSH source

The PUSH instruction decrements stack pointer by two and copies a word from some source to the location in the stack where the stack pointer points. Here the source must be a **word** (16 bit). The source of the word can be a general purpose register, a segment register or memory.

It is important to note that whenever data is pushed onto the stack, the first (most significant) data byte moves into the stack segment memory location addressed by SP–1. The second (least significant) data byte moves into the stack segment memory location addressed by SP–2.

Examples :

1. PUSH CX ; Decrements SP by 2, copy CX to stack

The Fig. 3.12 shows the execution of PUSH CX instruction.

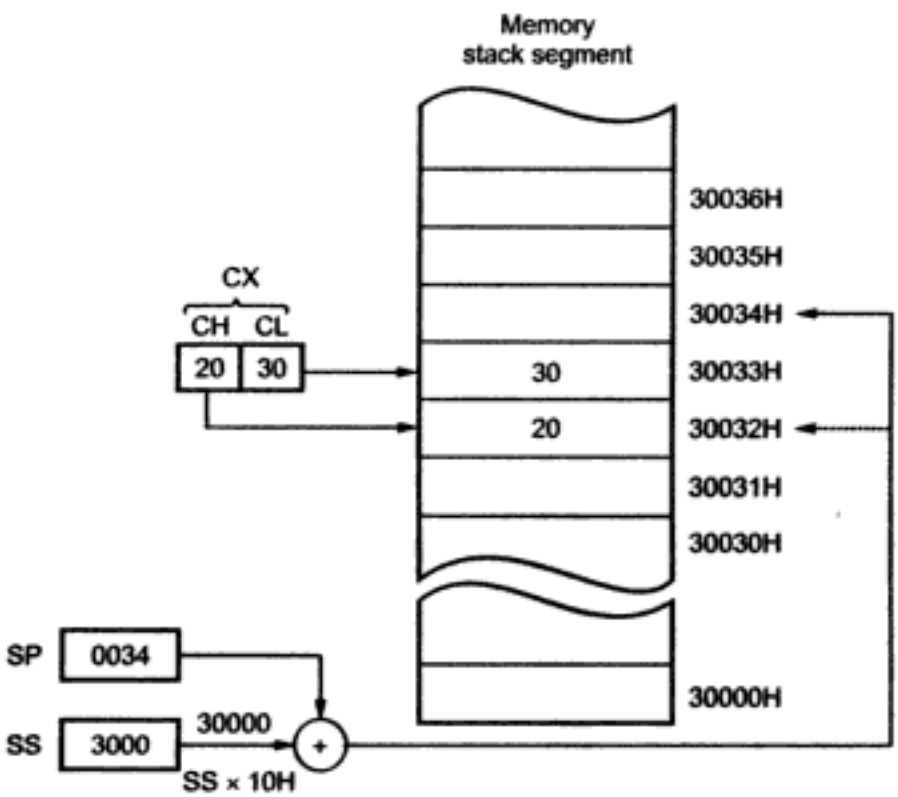


Fig. 3.12

Note : After execution of installation SP = 0032 H and it is indicated by dotted arrow.

```
2.  PUSH DS              ; Decrement SP by 2, copy DS to stack
3.  PUSH NEXT [BX]       ; Decrement SP by 2, copy a word from memory in
                         ; DS (i.e. PA = EA + DS) to stack with
                         ; EA = NEXT + [BX]
```

PUSHF

Puts the flag register contents on the stack. Whenever this instruction is executed, the most significant byte of flag register moves into the stack segment memory location addressed by SP–1. The least significant byte of flag register moves into the stack segment memory location addressed by SP–2.

POP destination

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a general purpose register, a segment register, or a memory location. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2. Whenever data is removed from the stack, the byte from the stack segment memory location addressed by SP moves into the most significant byte of the destination register and the byte from the stack segment memory location addressed by SP + 1 moves into the least significant byte of the destination register.

Examples :

```
1.  POP CX               ; Copy a word from top of stack
                         ; to CX and increment SP by 2.
```

The Fig. 3.13 shows the execution of POP CX instruction.

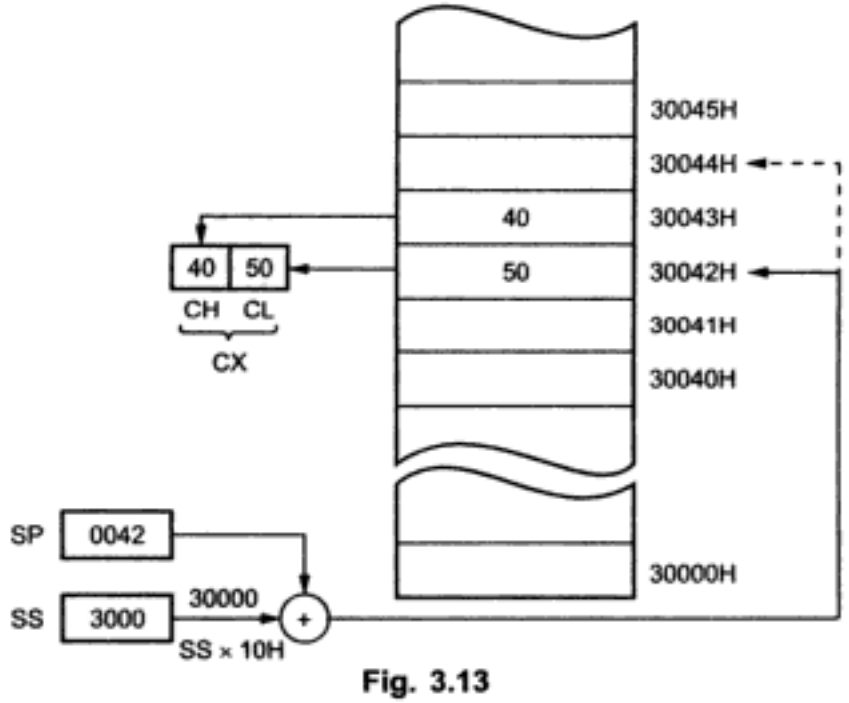


Fig. 3.13

Note : After execution of instruction SP = 0044H and it is indicated by dotted arrow.

```
2.  POP DS              ; Copy a word from top of stack
                        ; to DS and increment SP by 2.
3.  POP NEXT [BX]       ; Copy a word from top of stack to memory in DS
                        ; (i.e. PA = EA + DS) with EA = NEXT + [BX], and
                        ; increment SP by 2.
```

Note : POP CS is illegal.

POPF

Removes the word from top of stack to the flag register. Whenever this instruction is executed, the byte from the stack segment memory location addressed by SP moves into the most significant byte of the flag register and the byte from the stack segment memory location addressed by SP+1 moves into the least significant byte of the flag register.

Initializing the stack

Before going to use any instruction which uses stack for its operation we have initialize stack segment, and we have reverse the memory area required for the stack. The stack can be initialized by including following sequence of instructions in the program.

METHOD 1 :

```
ASSUME CS : CODE, DS : DATA, SS : STACK
:

STACK SEGMENT

S_DATA DB 100 DUP (?)

STACK ENDS
```

Note : Matter typed in Bold letters is included to initialize stack. This program sequence reserves 100 bytes for the stack operation.

METHOD 2 :

Syntax : · Stack [size]

Example : · Stack 100

The · stack is a directive, which provides shortcut in definition of the stack segment. The default size is 1024 bytes. The instruction · stack 100 reserves 100 bytes for the stack operation.

3.4.3 Load Effective Address

The load effective address group includes following instructions.

- LEA

- LDS
- LES

LEA Instruction : Load Effective Address : LEA register, source

This instruction determines the offset of the variable or memory location named as the source and loads this address in the specified 16-bit register. Flags are not affected by LEA instruction.

Examples :

```
LEA CX, TOTAL          ; Load CX with offset of TOTAL in DS.
LEA BP, SS : STACK_TOP ; Load BP with offset of STACK_TOP in SS.
LEA AX, [BX] [DI]      ; Load AX with EA = [BX] + [DI]
```

LDS Instruction : Load register and DS with words from memory. LDS register, memory address of first word.

This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into the DS register.

Examples :

```
LDS CX, [391AH]        ; Copy contents of memory at displacement of
                       ; 391AH and 391BH to CX. Then copy contents at
                       ; displacement of 391CH and 391DH in DS.
```

LES Instruction : Load register and ES with words from memory. LES register, memory address of first word.

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory location is copied into the specified register and the word from the next two memory locations is copied into the ES register.

Example :

```
LES CX, [3483H]        ; Copy contents of memory at displacement of 3483H0
                       ; in DS to CL, contents of 3484H in DS to CH and
                       ; copy the contents of memory at displacement of
                       ; 3485H and 3486H in DS to ES register.
```

3.4.4 String Data Transfer Instructions

MOVS/MOVSB/MOVSW

These instructions copy a byte or word from a location in the data segment to a location in the extra segment. The offset of the source byte or word in the data segment must be in the SI register. The offset of the destination in the extra segment must be contained in the DI register. For multiple byte or multiple word moves the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or word is moved SI and DI are automatically adjusted to point to the next source

and the next destination. If the direction flag is 0, then SI and DI will be incremented by 1 after a byte move and they will incremented by 2 after a word move. If the DF is a 1, then SI and DI will be decremented by 1 after a byte move and they will be decremented by 2 after a word move. MOVS affects no flags.

The way to tell the assembler whether to code the instruction for a byte or word move is to add a "B" or a "W" to the MOVS mnemonic. MOVSB, for example, says move a string as bytes. MOVSW says move a string as words.

Examples :

```
CLD                 ; Clear Direction Flag to autoincrement SI and DI
MOV AX, 0000H
MOV DS, AX          ; Initialize data segment register to 0
MOV ES, AX          ; Initialize extra segment register to 0
MOV SI, 2000H       ; Load offset of start of source string into SI
MOV DI, 2400H       ; Load offset of start of destination into DI
MOV CX, 04H         ; Load length of string in CX as counter
REP MOVSB           ; Decrement CX and MOVSB until CX will be 0.
```

After move SI will be one greater than offset of last byte in source string. DI will be one greater than offset of last byte of destination string. CX will be 0.

REP is a prefix which is written before MOVSB to repeat execution of it until CX = 0.

REP/REPE/REP2/REPNE/REPNZ Prefix

REP is a prefix which is written before one of the string instructions. These instructions repeat until specified condition exists.

Instruction Code	Condition for Exit
REP	CX = 0
REPE/REPZ	CX = 0 or ZF = 0
REPNE/REPNZ	CX = 0 or ZF = 1

Examples :

```
REPZ CMP SB         ; Compare string bytes until CX = 0
                    ; or until string bytes not equal.
```

LODS/LODSB/LODSW

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. LODS does not affect any flags. LODSB copies byte and LODSW copies a word.

Examples :

```
CLD                          ; Clear direction flag so SI is autoincremented
MOV SI, OFFSET S_STRING      ; Point SI at string
LODS S_STRING.
```

STOS/STOSB/STOSW

The STOS instruction copies a byte from AL or a word from AX to a memory location in the extra segment. DI is used to hold the offset of the memory location in the extra segment. After the copy, DI is automatically incremented or decremented to point to the next string element in memory. If the direction flag, DF, is cleared, then DI will automatically be incremented by one for a byte string or incremented by two for a word string. If the direction flag is set, DI will be automatically decremented by one for a byte string or decremented by two for a word string. STOS does not affect any flags. STOSB copies byte and STOSW copies a word.

Examples :

```
MOV DI, OFFSET D_STRING      ; Point DI at destination string
STOS D_STRING                ; Assembler uses string name to determine
                             ; whether string is of type byte or type word.
                             ; If byte string, then string byte replaced
                             ; with contents of AL. If word string, then
                             ; string word replaced with contents of AX.
MOV DI, OFFSET D_STRING      ; Point DI at destination string
STOSB                        ; "B" added to STOS mnemonic directly
                             ; tells assembler to replace byte in string with byte from
                             ; AL. STOSW would tell assembler directly to replace a
                             ; word in the string with a word from AX.
```

3.4.5 Miscellaneous Data Transfer Instructions

This group consists of following instructions.

- XCHG
- LAHF
- SAHF
- XLAT
- IN and OUT

XCHG Instruction : XCHG destination, source.

The XCHG instruction exchanges the contents of a register with the contents of another register or the contents of a register with the contents of a memory location(s). The instruction cannot exchange the contents of two memory locations. The source and destination both must be words or bytes. The segment registers can't be used in these instructions.

Examples :

```
XCHG BX, CX          ; Exchange word in BX with word in CX.
XCHG AL, CL          ; Exchange byte in AL with byte in CL.
XCHG AL, SUM [BX]    ; Exchange byte in AL with byte in memory at
                     ; EA = SUM + [BX]. PA = EA + DS.
```

LAHF Instruction : Load lower byte of flag register in AH.

This instruction copies the contents of lower byte of 8086 flag register to AH register.

SAHF Instruction : Copy AH register to low byte of flag register.

The contents of the AH register are copied into the lower byte of the 8086 flag register.

XLAT Instruction : Translate byte in AL.

The XLAT instruction replaces a byte in the AL register with a byte from a lookup table in memory. BX register stores the offset of the starting address of the lookup table and AL register stores the byte number from the lookup table. This instruction copies byte from address pointed by [BX + AL] back into AL.

IN and OUT Instructions

IN Instruction : Input a byte or word from port.

The IN instruction will copy data from a port to the accumulator. If an 8-bit port is read, the data will go to AL and if an 16-bit port is read the data will goto AX.

The IN instruction can be executed in two different addressing modes,

1. Direct : In direct addressing mode 8-bit address of the port is a part of the instruction.

Examples :

```
IN AL, 0F8H          ; Copy a byte from port 0F8H to AL.
IN AX, 95H           ; Copy a word from port 95H to AX.
```

2. Indirect : In indirect addressing, the address of the port is referred from DX register. Since DX is a 16-bit register, the port address can be any number between 0000H to FFFFH. Therefore it is possible address to upto 65,536 ports in this mode.

Examples :

```
MOV DX, 30F8H        ; Load 16-bit address of the port in DX.
IN AL, DX            ; Copy a byte from 8-bit port 30F8H to AL.
IN AX, DX            ; Copy a word from 16-bit port 30F8H to AX.
```

OUT Instruction : Send a byte or word to a port.

The OUT instruction copies a byte from AL or a word from AX to the specified port.

The OUT instruction can be executed in two different addressing modes.

1. Direct : In direct addressing mode 8-bit address of the port is a part of the instruction.

Examples :

```
OUT 0F8H, AL         ; Copy contents of AL to 8 bit port 0F8H.
OUT 0FBH, AX         ; Copy contents of AX to 16-bit port 0FBH.
```

2. Indirect : In indirect addressing, the address of the port is referred from DX register. It has advantage of accessing 2^{16} i.e. 65536 ports as mentioned earlier.

Examples :

```
MOV DX, 30F8H        ; Load 16-bit address of the port in DX.
OUT DX, AL           ; Copy the contents of AL to port 30F8H.
OUT DX, AX           ; Copy the contents of AX to port 30F8H.
```

3.5 Arithmetic and Logic Instructions

The arithmetic and logic group of instructions include

- Addition instructions
- Subtraction instructions
- Multiplication instructions
- Division
- BCD and ASCII arithmetic instructions
- Comparison
- Basic logic instructions - AND, OR NOT, XOR
- Shift and rotate instructions

3.5.1 Addition

This group of instructions consist of following instructions

- ADD : Addition
- ADC : Addition with carry
- INC : Increment (Add 1)

ADD/ADC Instruction : ADD destination, source / ADC destination, source.

These instructions add a number from source to a number from destination and put the result in the destination. The ADC, instruction also adds the status of carry flag into the result. The source may be an immediate number, a register, or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and destination both must be a word or byte. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

Flags affected : AF, CF, OF, PF, SF, ZF.

Examples :

```
ADD AL, 0F0H          ; Add immediate number 0F0H to contents of AL.
ADC DL, CL            ; Add contents of CL to contents of DL with carry
                      ; and store result in DL i.e. DL ← DL + CL + CY
ADC DX, BX            ; Add contents of BX to contents of DX with carry
                      ; and store result in DX i.e. DX ← DX + BX + CY
ADD CL, TOTAL [BX]    ; Add byte from effective address
                      ; TOTAL [BX] to contents of CL
ADD CX, TOTAL [BX]    ; Add word from effective address
                      ; TOTAL [BX] to contents of CX.
```

INC Instruction : Increment destination.

The INC instruction adds 1 to the specified destination. The destination may be a register or memory location. The AF, OF, PF, SF and ZF flags are affected.

Examples :

```
INC AL                ; Add 1 to contents of AL.
INC BX                ; Add 1 to contents of BX.
```

NOTE : The carry flag CF is not affected.

If contents of 8-bit register are FFH and 16-bit register are FFFFH, after INC instruction contents of registers will be zero without affecting carry flag.

```
INC BYTE PTR [BX]     ; Increment byte at offset of BX in DS.
                      ; BYTE PTR directive indicates to the assembler
                      ; that the byte from memory is to be incremented.
INC WORD PTR [BX]     ; Increment word at offset of BX in DS.
                      ; WORD PTR directive indicates to the assembler
                      ; that the word from memory is to be incremented.
```

3.5.2 Subtraction

This group of instructions consist of following group of instructions.

- SUB : Subtraction
- SBB : Subtraction with borrow
- DEC : Decrement (subtract 1)
- NEG : 2's complement of a number

SUB/SBB Instruction : SUB destination, Source.

SBB destination, Source.

These instructions subtract the number in the source from the number in the destination and put result in the destination. The SBB, instruction also subtracts the status of carry flag from the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination both cannot be memory locations. The source and destination both must be word or byte. If you want to subtract a byte from a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before subtracting.

Flags affected : AF, CF, OF, PF, SF, and ZF.

Examples :

```
SUB AL, 0F0H          ; Subtract immediate number 0F0H
                      ; from contents of AL store result in AL.
SBB DL, CL            ; Subtract contents of CL and status of carry flag
                      ; from the contents of DL and store result in DL.
                      ; i.e. DL ← DL – CL – CY
SBB DX, BX            ; Subtract contents of BX and status of carry
                      ; flag from the DX and store result in DX.
                      ; i.e. DX ← DX – BX – CY
SUB CL, TOTAL [BX]    ; Subtract byte from effective address TOTAL [BX]
                      ; from the contents of CL and store result in CL
SUB CX, TOTAL [BX]    ; Subtract word from effective address TOTAL [BX]
                      ; from the contents of CX and store result in CX.
```

DEC Instruction : Decrement destination.

The DEC instruction subtract 1 from the specified destination. The destination may be a register or a memory location. The AF, OF, PF, SF and ZF flags are affected.

Examples :

```
DEC AL          ; Subtracts 1 from the contents of AL.
DEC BX          ; Subtracts 1 from the contents of BX.
```

Note : The carry flag CF is not affected.

If the contents of 8-bit register are 00H and 16-bit register are 0000H, after DEC instruction contents of registers will be FFH and FFFFH respectively without affecting carry flag.

```
DEC BYTE PTR [BX]   ; Decrement byte at offset of BX in DS.
                    ; BYTE PTR directive indicates to the assembler
                    ; that the byte from memory is to be decremented.
DEC WORD PTR [BX]   ; Decrement word at offset of BX in DS.
                    ; WORD PTR directive indicates to the assembler
                    ; that the word from memory is to be decremented.
```

NEG Instruction : Form 2's complement.

This instruction replaces the number in a destination with the 2's complement of that number. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

The negate instruction updates the AF, CF, SF, PF, ZF and OF flags.

Examples :

```
                    ; AL = 0011 0101  35H
NEG AL              ; Replace number in AL with its 2's complement
                    ; AL = 1100 1011 = CBH
```

3.5.3 Comparison

The comparison instruction (CMP) compares a byte/word from the specified source with a byte/word from the specified destination. The source and destination both must be byte or word. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. However the source and destination both can't be memory locations. The comparison is done by subtracting the source byte or word from the destination byte or word. But the result is not stored in the destination. Source and destination remain unchanged, only flags are updated.

Flags : The AF, OF, SF, ZF, PF and CF are updated by the CMP instruction.

Examples :

```
CMP BL, 01H         ; Compare immediate number 01H with byte in BL.
CMP CX, BX          ; Compare word in BX with word in CX.
CMP CX, TOTAL       ; Compare word at displacement
                    ; TOTAL in DS with word in CX.
```

Note : It is not possible to compare segment registers.

The result of comparison is checked by conditional jump, conditional call and conditional return instructions. We discuss these instructions later in this chapter.

3.5.4 Multiplication

This group of instructions consist of following group of instructions.

- MUL : Unsigned multiplication
- IMUL : Signed multiplication

MUL Instruction : MUL source.

This instruction multiplies an unsigned byte from source and unsigned byte in AL register or unsigned word from source and unsigned word in AX register. The source can be a register or a memory location. When the byte is multiplied by the contents of AL, the result is stored in AX. The most significant byte is stored in AH and least significant byte is stored in AL. When a word is multiplied by the contents of AX, the most significant word of result is stored in DX and least significant word of result is stored in AX.

Flags : MUL instruction affect AF, PF, SF, and ZF flags.

Examples :

```
MUL BL                  ; AL × BL, result in AX.
MUL BX                  ; AX × BX, result high word in DX low word in AX.
MUL WORD PTR [BX]       ; AX times word in DS pointed by [BX]
                        ; result high word in DX low word in AX.
```

IMUL Instruction :

This instruction multiplies a signed byte from some source and a signed byte in AL, or a signed word from some source and a signed word in AX. The source can be register or memory location. When a signed byte is multiplied by AL a signed result will be put in AX. When a signed word is multiplied by AX, the high-order word of the signed result is put in DX and the low-order word of the signed result is put in AX.

If the upper byte of a 16-bit result or the upper word of 32-bit result contains only copies of the sign bit (all 0's or all 1's), then the CF and the OF flags will both be 0's. The AF, PF, SF, and ZF flags are undefined after IMUL.

To multiply a signed **byte** by a signed **word** it is necessary to move the byte into a word location and fill the upper byte of the word with copies of the sign bit. This can be done using CBW instruction.

Examples :

```
IMUL BL                 ; AL × BL, result in AX
IMUL CX                 ; AX × CX, high-order word of result in DX and
                        ; low-order word of result in AX.
```

3.5.5 Division

This group of instructions consists of following group of instructions

- DIV
- IDIV

DIV Instruction : DIV source

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

When dividing a word by a byte, the word must be in AX register. After the division AL will contain an 8-bit quotient and AH will contain an 8-bit remainder. If an attempt is made to divide by 0 or the quotient is too large to fit in AL (greater than FFH), the 8086 will automatically execute a type 0 interrupt.

When a double word is divided by a word, the most significant word of the double word must be in DX and the least-significant word must be in AX. After the division AX will contain a 16-bit quotient and DX will contain a 16-bit remainder. Again, if an attempt is made to divide by 0 or the quotient is too large to fit in AX register (greater than FFFFH), the 8086 will do a type 0 interrupt. For DIV instruction source may be a register or memory location.

To divide a byte by a byte, it is necessary to put the dividend byte in AL and fill AH with all 0's. Similarly, to divide a word by a word, it is necessary to put the dividend word in AX and fill DX with all 0's.

Flags : All flags are undefined after a DIV instruction.

Examples :

```
DIV CL          ; Word in AX/byte in CL,
                ; Quotient in AL, remainder in AH.
DIV CX          ; Double word in DX and AX/word in CX,
                ; Quotient in AX, remainder in DX.
```

IDIV Instruction : IDIV source

This instruction is used to divide a signed word by a signed byte, or to divide a signed double word (32-bits) by a signed word. Rest all is similar to DIV instruction.

3.5.6 BCD and ASCII Arithmetic

The 8086 allows arithmetic manipulation of both BCD (Binary coded decimal) and ASCII (American Standard Code for Information Interchange) data. This is accomplished by instructions that adjust the numbers for BCD and ASCII arithmetic. Let us see instructions used for BCD and ASCII arithmetic.

3.5.6.1 BCD Arithmetic

The 8086 provides two instructions to support BCD arithmetic. They correct result of a BCD addition and a BCD subtraction. The DAA (decimal adjust after addition) instruction follows BCD addition, and the DAS (decimal adjust after subtraction) follows BCD subtraction. Both instructions correct the result of the addition or subtraction so that it is a BCD number.

DAA Instruction : Decimal Adjust Accumulator.

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number.

Instruction works as follows :

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction adds 6 (60) to the high-order four bits.

Examples :

```
1.                  ; AL = 0011 1001 = 39 BCD
                    ; CL = 0001 0010 = 12 BCD
Add AL, CL          ; AL = 0100 1011 = 4BH
DAA                 ; Add 0110 Because 1011 > 9
                    ; AL = 0101 0001 = 51 BCD
2.                  ; AL = 1001 0110 = 96 BCD
                    ; BL = 0000 0111 = 07 BCD
ADD AL, BL          ; AL = 1001 1101 = 9DH
DAA                 ; Add 0110 Because 1101 > 9
                    ; AL = 1010 0011 = A3H
                    ; 1010 > 9 so add 0110 0000
                    ; AL = 0000 0011 = 03 BCD, CF = 1. The result is 103.
```

The instruction updates the AF, CF, PF, and ZF. The OF is undefined after DAA instruction.

Note : only works for AL.

DAS Instruction : Decimal Adjust After Subtraction.

This instruction is used after subtracting two packed BCD numbers to make sure the result is correct packed BCD. Instruction works as follows :

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set; the instruction subtracts 6 (06) from the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction subtracts 6 (60) from the high-order four bits.

Examples :

```
1.                  ; AL = 0011 0010 = 32 BCD
                    ; CL = 0001 0111 = 17 BCD
SUB AL, CL          ; AL = 0001 1011 = 1BH
                    ; Subtract 0110 Because 1011 > 9
                    ; AL = 0001 0101 = 15 BCD
2.                  ; AL = 0010 0011 = 23 BCD
                    ; CL = 0101 1000 = 58 BCD
SUB AL, CL          ; AL = 1100 1011 = CBH CF = 1
                    ; Subtract 0110 (6) Because 1011 > 9
                    ; AL = 1100 0101 = C5H
                    ; Subtract 0110 0000 Because 1100 > 9
                    ; AL = 0110 0101 = 65 BCD CF = 1,
                    ; CF = 1 means borrow
                    ; is needed means number is negative (– 65).
```

The DAS instruction updates the AF, CF, PF, and ZF. The OF flag is undefined after DAS instruction.

Note : DAS only works for AL

3.5.6.2 ASCII Arithmetic

ASCII numbers range in value from 30H to 39H for the numbers 0-9. The 8086 provides four instructions for ASCII arithmetic.

- AAA:ASCII adjust after addition
- AAS :ASCII adjust after subtraction
- AAM:ASCII adjust after multiplication
- AAD :ASCII adjust before division

AAA Instruction : ASCII Adjust for Addition.

The numbers from 0-9 are represented as 30H-39H in ASCII code. When you want to add two decimal digits which are represented in ASCII code, it is necessary to mask upper nibble (3) from the code before addition. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each digit. The AAA instruction can be used after addition to get the current result in unpacked BCD form.

Examples :

```
                    ; AL    = 0011 0100 ASCII 4
                    ; CL    = 0011 1000 ASCII 8
ADD AL, CL          ; AL    = 0110 1100
                    ; 6CH   = Incorrect temporary result
AAA                 ; AL    = 0000 0010 Unpacked BCD for 2.
```

```
                    ; Carry    = 1 to indicate correct answer is 12
                                 decimal.
```

The AAA instruction updates the AF and the CF, but the OF, PF, SF, and ZF are left undefined.

Note : The AAA instruction only works on the AL register.

AAS Instruction : ASCII Adjust After Subtraction.

The numbers from 0-9 are represented as 30-39 in ASCII code. When you want to subtract two decimal digits which are represented in ASCII code, it is necessary to mask upper nibble (3) from the code before subtraction. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each digit. The AAS instruction can be used after subtraction to get the current result in unpacked BCD form.

Examples :

```
1.                  ; AL = 0011 1000 ASCII 8
                    ; CL = 0011 0010 ASCII 2
SUB AL, CL          ; AL = 0000 0110 BCD 06
                    ; CF = 0
AAS                 ; AL = 0000 0010 = BCD 06
                    ; CF = 0 no borrow required
2.                  ; AL = 0011 0010 ASCII 2
                    ; CL = 0011 1000 ASCII 8
SUB AL, CL          ; AL = 1111 1010 = FAH
                    ; CF = 1
AAS                 ; AL = 0000 0110 = BCD 6
                    ; CF = 1 borrow needed means (– 6)
```

AAM Instruction : ASCII Adjust After Multiplication.

After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.

Examples :

```
                    ; AL = 0000 0100 = Unpacked BCD 4
                    ; CL = 0000 0110 = Unpacked BCD 6
MUL CL              ; AL × CL Result in AX.
                    ; AX = 0000 0000 0001 1000 = 0018H
AAM                 ; AX = 0000 0010 0000 0100 = 0204H
                    ; Which is unpacked BCD for 24.
```

Now by adding 3030H in AX register we get the result in ASCII form.

AAD Instruction : ASCII Adjust Before Division.

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the division AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. The PF, SF and ZF are updated. The AF, CF and OF are undefined after AAD.

Examples :

```
                    ; AX = 0403 unpacked BCD for 43 decimal, CL = 07H
AAD                 ; Adjust to binary before division,
                    ; AX = 002BH = 2BH = 43 decimal.
DIV CL              ; Divide AX by unpacked BCD in CL.
                    ; AL = quotient = 06 unpacked BCD
                    ; AH = remainder = 01 unpacked BCD
```

Now by adding 3030H in AX register we get the quotient and remainder in ASCII form.

3.5.7 Basic Logic Instructions

The basic logic instructions include AND, OR, Exclusive-OR, and NOT. This group also includes TEST instruction which is a special form of the AND instruction.

AND Instruction : AND destination, source.

We know that, AND operation with two inputs produces result logic 1 only when both the inputs are logic 1. i.e. $Y = A \cdot B$.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.2 : Truth table for AND gate

This instruction logically ANDs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be an immediate number, a register or a memory location. The destination may be a register

or a memory location. The source and destination both cannot be memory locations in the same instruction. The CF and OF are both 0 after AND. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.                  ; AL = 1001 0011 = 93H
                    ; BL = 0111 0101 = 75H
AND BL, AL          ; AND byte in AL with byte in BL
                    ; BL = 0001 0001 = 11H
2.                  ; CX = 0110 1011 1001 1110
AND CX, 00F0H       ; CX = 0000 0000 1001 0000
```

The AND operation clears bits of a binary number. The task of clearing a bit in a binary number is called **masking.** The Fig. 3.14 shows the process of masking.

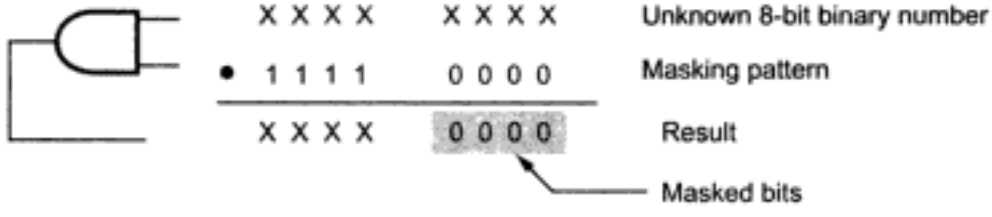


Fig. 3.14 Masking using AND operation

OR Instruction : OR destination, source.

We know that, OR operation with two inputs produces result logic 1 when any one or both inputs are logic 1 i.e., Y = A + B.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.3 Truth table for OR gate

This instruction logically ORs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be an immediate number, a register or a memory location. The destination may be a register

or a memory location. The source and destination both can not be memory locations in the same instruction. The CF and OF are both 0 after OR. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.                  ; AL = 1001 0011 = 93H
                    ; BL = 0111 0101 = 75H
OR BL, AL           ; OR byte in AL with byte in BL.
                    ; BL = 1111 0111 = F7H
2.                  ; CX = 0110 1011 1001 1110
OR CX, 00F0H        ; CX = 0110 1011 1111 1110
```

The OR instruction is used to set (make one) any bit in the binary number. This is illustrated in Fig. 3.15.

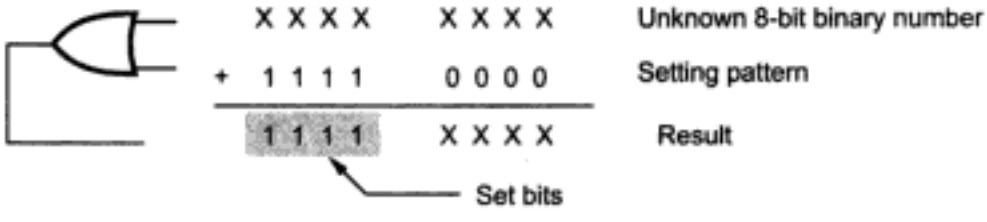


Fig. 3.15 Setting bit/s using OR operation

XOR Instruction : XOR destination, source.

We know that, XOR operation produces result logic 1 when odd number of inputs are logic 1 i.e. $Y = A\overline{B} + \overline{A}B$.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.4 : Truth table for XOR gate

This instruction logically XORs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be

an immediate number, a register or a memory location. The destination may be a register or a memory location. The source and destination both cannot be memory locations in the same instruction. The CF and OF are both 0 after XOR. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.              ; AL =  1010      1111 = AFH
                ; BL =  1111      0000 = F0H
XOR BL, AL      ; XOR byte in AL with byte in BL
                ; BL =  0101      1111 = 5FH
```

The XOR instruction is used if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. This is illustrated in Fig. 3.16.

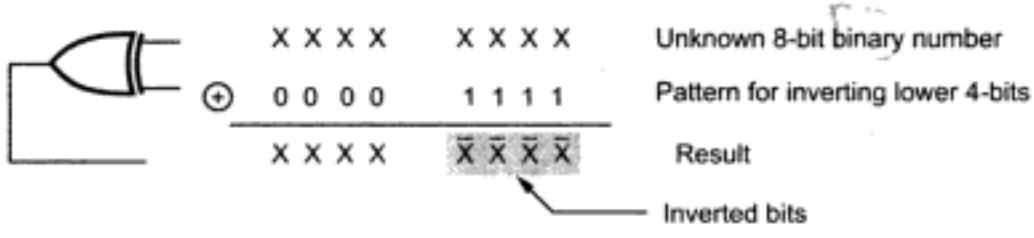


Fig. 3.16 Inversion of part of a number using XOR operation

NOT Instruction : NOT destination.

The NOT instruction inverts each bit of a byte or a word. The destination can be register or a memory location.

Flags : NOT instruction affects no flag.

Examples :

```
                ; AL = 0110 1100
NOT AL          ; AL = 1001 0011
                ; CX = 1010 1111 0010 0110
NOT CX          ; CX = 0101 0000 1101 1001
```

Test and bit test instructions :

The TEST instruction performs the AND operation. The difference is that the AND instruction changes the destination operand, while the TEST instruction does not. A TEST only affects the condition of the flag register, which indicates the result of the test.

PF, SF and ZF will be updated to show the results of the ANDing. PF has meaning only for the lower 8 bits of the destination. AF will be undefined.

Examples :

```
TEST AL, CL        ; AND CL with AL.
                   ; Update flags, result is not stored.
TEST BX, CX        ; AND CX with BX.
                   ; Update flags, result is not stored.
```

The TEST instruction functions in the similar manner as a CMP instruction. The difference is that the TEST instruction normally tests a single bit (or occasionally multiple bits), while the CMP instruction tests the entire byte or word. The Fig. 3.17 shows the bit pattern and test operation for testing of bit 0. If zero flag is set (Z = 1) after this operation, the bit under test bit-0 is zero ; otherwise bit-0 is 1.

The zero flag is usually tested by JZ or JNZ instructions. Therefore, the TEST instruction is usually followed by either the JZ or JNZ instruction.

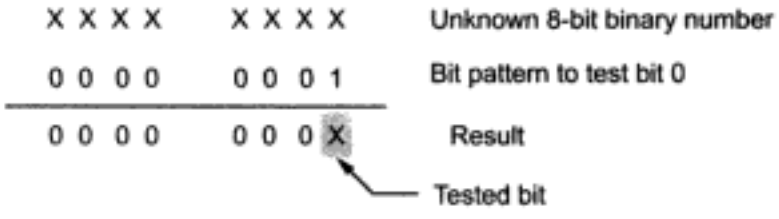


Fig. 3.17 TEST operation

3.5.8 Shift and Rotate

3.5.8.1 Shift

Shift instructions position or move binary data to the left or right by shifting them within the register or memory location. They also perform multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift). The shift operations can be classified as logical shifts and arithmetic shifts. The logical shifts move a 0 into the rightmost bit position for a logical left shift (SHL) and a 0 into the leftmost bit position for a logical right shift (SHR). The arithmetic left shift (SAL) and logical left shift operations are identical. However, arithmetic and logical right shifts are different because the arithmetic right shift (SAR) copies the sign bit through the number, while the logical right shift copies a 0 through the number. This is illustrated in Fig. 3.18. Logical shift operations are used with unsigned numbers; they perform multiplication or division of unsigned numbers. On the otherhand, arithmetic shift operations are used with signed numbers; they perform multiplications or division of signed numbers.

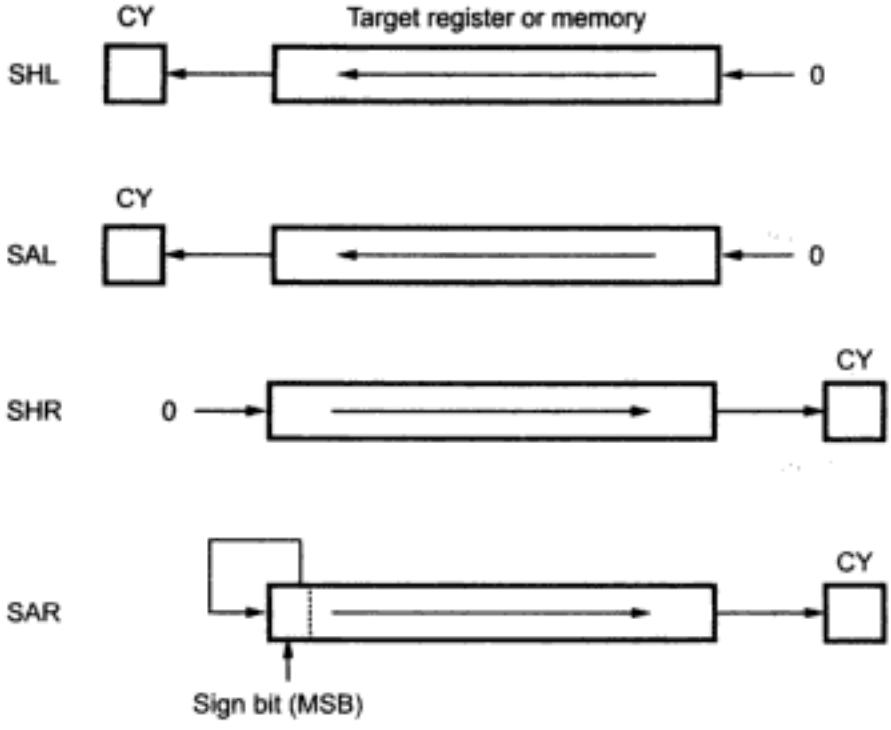
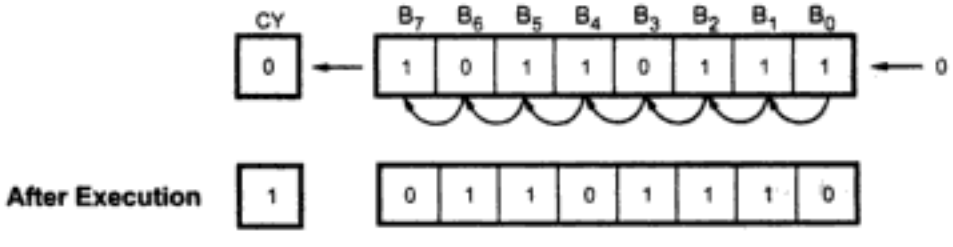


Fig. 3.18 Shift operations

SAL/SHL Instruction : SAL/SHL destination, count.

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. But if the number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SAL instruction for byte operation.



Flags : All flags are affected.

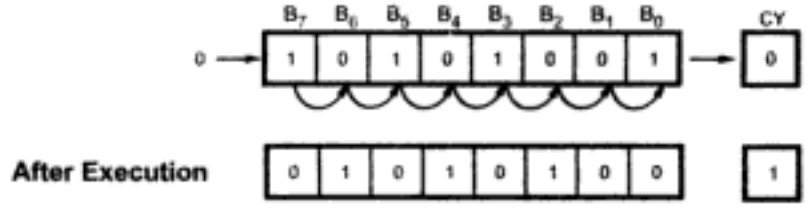
Examples :

```
SAL CX, 1          ; Shift word in CX 1 bit position left, 0 in LSB
MOV CL, 05H        ; Load desired number of shifts in CL
SAL AX, CL         ; Shift word in AX left 5 times
                   ; 0s in 5 least-significant bits.
```

SHR Instruction : SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. But if the number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SHR instruction for byte operation.



Flags : All flags are affected.

Examples :

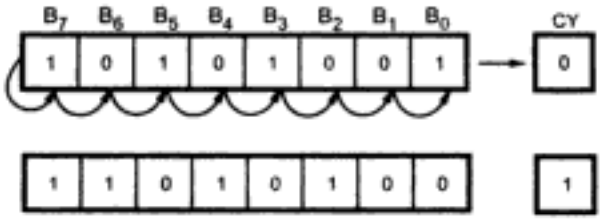
```
SHR CX, 1          ; Shift word in CX 1 bit position right, 0 in MSB.
MOV CL, 05H        ; Load desired number of shifts in CL.
SHR AX, CL         ; Shift word in AX right 5 times
                   ; 0's in 5 most significant bits.
```

SAR Instruction : SAR destination, count.

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF. In the case of multiple shifts, CF will contain the bit most recently shifted in from the LSB. Bits shifted into CF previously will be lost.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SAR instruction for byte operation.



Flags : All flags are affected.

Examples :

```
SAR BL, 1          ; Shift byte in BL one bit position right.
MOV CL, 04H        ; Load desired number of shifts in CL.
SAR DX, CL         ; Shift word stored in DX 4 bit positions right.
```

3.5.8.2 Rotate

Rotate instructions position or move binary data by rotating the information in a register or memory location, either from one end to another or through the carry flag. This is illustrated in Fig. 3.19.

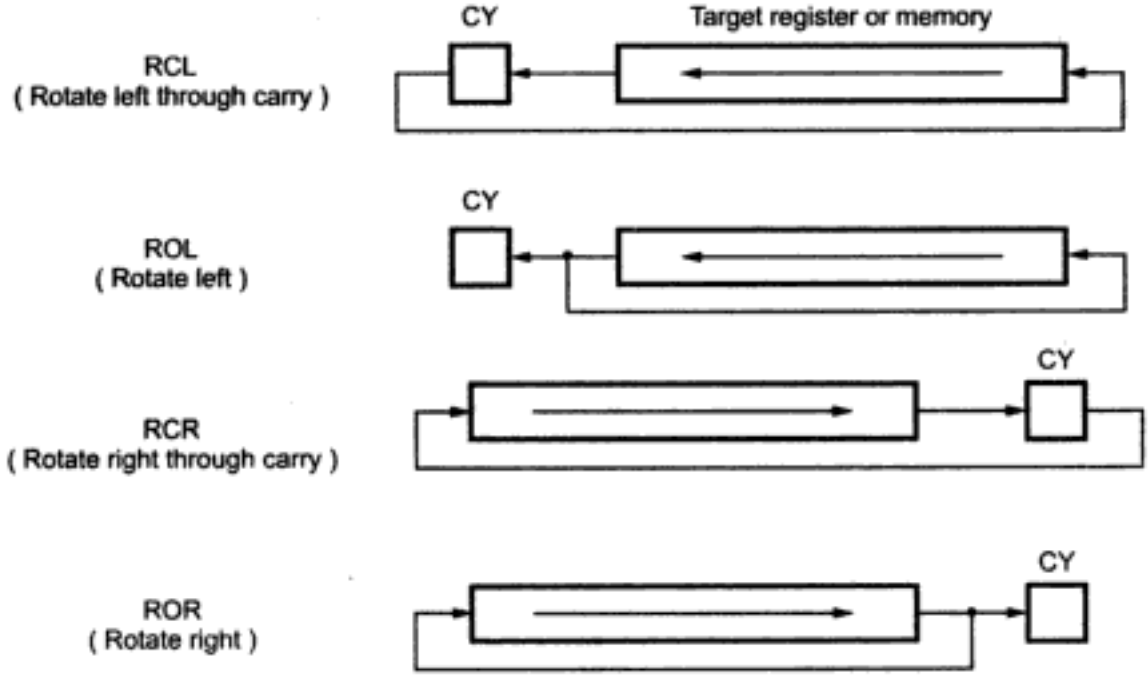
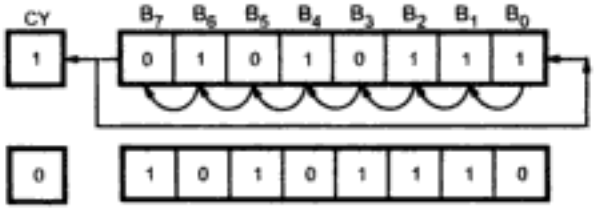


Fig. 3.19 Rotate operations

ROL Instruction : ROL destination, count.

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Diagram shows ROL instruction for byte rotation.



The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Examples :

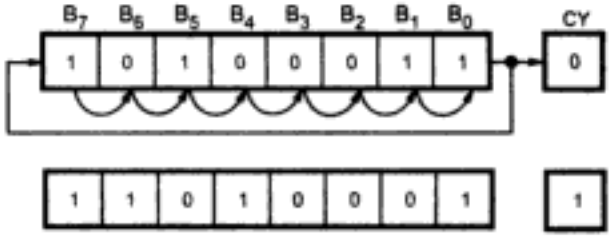
```
ROL CX, 1       ; Word in CX one bit position left, MSB to LSB and CF
MOV CL, 03H     ; Load desired number of bits to rotate in CL.
ROL BL, CL      ; Rotate BL three positions.
```

ROR Instruction : ROR destination, count.

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows ROR instruction for byte rotation.



Examples :

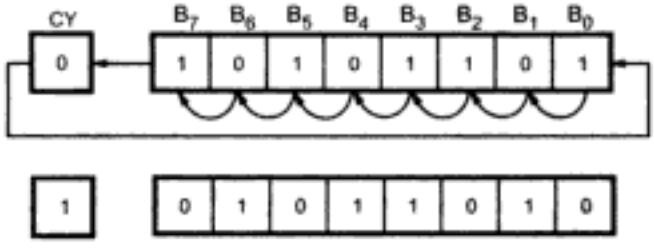
```
ROR CX, 1          ; Rotated word in CX one bit position right,
                   ; LSB to MSB and CF.
MOV CL, 03H        ; Load number of bits to rotate in CL.
ROR BL, CL         ; Rotate BL three positions.
```

RCL Instruction : RCL destination, count.

This instruction rotates all of the bits in a specified word or byte some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is placed as a new LSB.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows RCL instruction for byte rotation.



Examples :

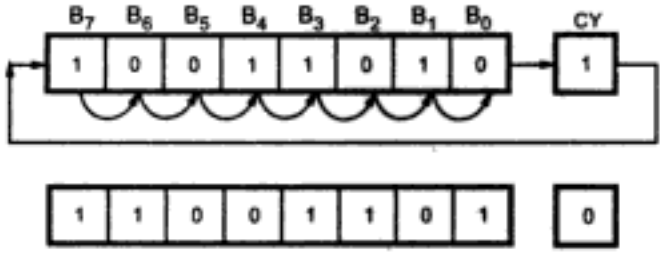
```
RCL CX, 1          ; Rotated word in CX 1 bit left, MSB to CF, CF to LSB.
MOV CL, 04H        ; Load number of bit positions to rotate in CL.
RCL AL, CL         ; Rotate AL 4 bits left.
```

RCR Instruction : RCR destination, count.

This instruction rotates all of the bits in a specified word or byte some number of bit positions to the right along with the carry flag. LSB is placed as a new carry and previous carry is placed as a new MSB.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position in the instruction.

Diagram shows RCR instruction for byte rotation.



Examples :

```
RCR CX, 1          ; Word in CX 1 bit right, LSB to CF, CF to MSB.
MOV CL, 04H        ; Load number of bit positions to rotate in CL.
RCR AL, CL         ; Rotate AL 4 bits right.
```

3.6 String Instructions

The 8086 instruction set provides following string instructions.

- REP/REPE/REPZ/REPNE/REPNZ
- MOVS/MOVSB/MOVSW
- LODS/LODSB/LODSW
- STOS/STOSB/STOSW
- CMPS/CMPSB/CMPSW
- SCAS/SCASB/SCASW

From the above six instructions we have already studied first four instructions in section 3.4. the remaining two instructions are string compare instructions. The string comparison instructions allow the programmer to test a section of memory against a constant or against another section of memory.

CMPS/CMPSB/CMPSW Instruction :

A string is a series of the same type of data items in sequential memory locations. The CMPS instruction can be used to compare a byte in one string with a byte in another string or to compare a word in one string with a word in another string. SI is used to hold the offset of a byte or word in the source string and DI is used to hold the offset of a byte or a word in the other string. The comparison is done by subtracting the byte or word pointed to by DI from the byte or word pointed to by SI. The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but neither operand is affected.

Examples :

```
                          ; Point SI at source string, Point DI at
                          ; destination string
MOV SI, OFFSET F_STRING
MOV DI, OFFSET S_STRING
CLD                       ; DF cleared so SI and DI will
                          ; autoincrement after compare
CMPS F_STRING, S_STRING   ; The assembler uses names to determine whether
                          ; strings were declared as type byte or as type
                          ; word.
MOV CX, 100               ; Put number of string elements in CX, Point SI at
                          ; source of string and DI at destination of string
MOV SI, OFFSET F_STRING
MOV DI, OFFSET S_STRING
STD                       ; DF set so SI and DI will autodecrement after
                          ; compare
REPE CMPSB                ; Repeat the comparison of string bytes until end
                          ; of string or until compared bytes are not equal.
```

After the comparison SI and DI will be automatically incremented or decremented according to direction flag to point to the next element in the two strings (if DF = 0, SI and DI ↑ otherwise ↓) CX functions as a counter which is decremented after each comparison. This will go on until CX = 0.

SCAS/SCASB/SCASW Instruction :

SCAS compares a string byte with a byte in AL or a string word with word in AX. The instruction affects the flags, but it does not change either the operand in AL (AX) or the operand in the string. The string to be scanned must be in the extra segment and DI must contain the offset of the byte or the word to be compared.

After the comparison DI will be automatically incremented or decremented according to direction flag to point to the next element in the two strings (if DF = 0, SI and DI ↑ otherwise ↓) CX functions as a counter which is decremented after each comparison. This will go on until CX = 0. SCAS affects the AF, CF, OF, PF, SF and ZF flags.

Examples :

```
                          ; Scan a text string of 80 characters
                          ; for a carriage return
MOV AL, 0DH               ; Byte to be scanned for into AL
MOV DI, OFFSET TEXT_STRING ; Offset of string to DI
MOV CX, 80                ; CX used as element counter
CLD                       ; Clear DF, so DI autoincrements
```

```
REPNE SCAS TEXT_STRING          ; Compare byte in string with byte in
                                ; AL.
```

SCASB says compare strings as bytes and SCASW says compare strings as words.

3.7 Program Control Transfer Instructions

These instructions are classified as

- Unconditional transfer instructions - CALL, RET, JMP
- Conditional transfer instructions - J cond

3.7.1 CALL and RET Instructions

Whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them. One way is to write the group of instructions as a separate procedure. We can then just CALL the procedure whenever we need to execute that group of instructions. For calling the procedure we have to store the return address onto the stack. This process takes some time. If the group of instructions is big enough then this overhead time is negligible with respect to execution time. But if the group of instructions is too short, the overhead time and execution time are comparable. In such cases, it is not desirable to write procedures. For these cases, we can use macros. Macro is also a group of instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a procedure. The disadvantage of macro is that it generates inline code each time when the macro is called which takes more memory. In this section we discuss the procedures.

From the above discussions, we know that the procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called **near procedure** otherwise it is referred to as **far procedure**. For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program and procedure. But for far procedures CALL instruction pushes both IP and CS on the stack. Let us see the detail description and examples of CALL instruction to enter the procedure and RET instruction to return from the procedure.

CALL Instruction :

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A **far CALL** is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

Examples :

Direct within segment (near)

```
CALL PRO            ; PRO is the name of the procedure.
                    ; The assembler determines displacement of pro
                    ; from the instruction after the CALL and codes
                    ; this displacement in as part of the instruction.
```

Indirect within-segment (near)

```
CALL CX             ; CX contains, the offset of the first instruction
                    ; of the procedure. Replaces contents of IP with
                    ; contents of register CX.
```

Indirect to another segment (far)

```
CALL DWORD PTR [BX] ; New values for CS and IP are fetched from four
                    ; memory locations in DS. The new value for CS
                    ; is fetched from [BX] and [BX + 1], the new IP
                    ; is fetched from [BX + 2] and [BX + 3].
```

RET Instruction :

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

A RET instruction can be followed by a number, for example, RET 4. In this case the stack pointer will be incremented by an additional four addresses after the IP or the IP and CS are popped off the stack. This form is used to increment the stack pointer up over parameters passed to the procedure on the stack.

Flags : The RET instruction affects no flags.

3.7.2 JMP Instruction

This group of instructions will always cause the 8086 to fetch its next instruction from the location specified or indicated by instruction rather than from the next location after the JMP instruction. The JMP instructions are basically classified as unconditional jump (JMP) and conditional jump instructions. A conditional jump instruction allows the programmer to make decisions based upon numerical tests. The results of numerical tests are held in the flag bits, which are then tested by conditional jump instructions.

The jump instructions are further classified as short, near and far jump instructions. A **short jump** is a two-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes from the address following the jump. A three byte **near jump** allows a branch or jump within ± 32 Kbytes (or anywhere in the current code segment) from the instruction in the current code segment. The segments are cyclic in nature. This means that, one location above offset address FFFFH is offset address 0000H, two locations above offset address FFFFH is offset address 0001H and so on. Thus, a displacement of ± 32 kbytes allow a jump to any location within the current code segment. In near jump only IP is changed, the contents of CS remains same. A five byte **far jump** allows a jump to any memory location within the real memory system. A far jump is a jump where destination location is from a different segment. In this case both IP and CS are changed as specified in the destination. The short and near jump are often called **intrasegment** jumps, and the far jumps are often called **intersegment** jumps. The short jumps are also called **relative jumps** because in such instructions the destination location is specified relative to the current location. The Fig. 3.20 shows instruction formats for short, near and far jump instructions.

Near and far jumps are further described as either direct or indirect. If the destination address for the jump is specified directly within the instruction, then the jump is described as direct. If the destination address for the jump is contained in a register or memory location, the jump is referred as indirect, because the 8086 has to access the specified register or memory location to get the required destination address.

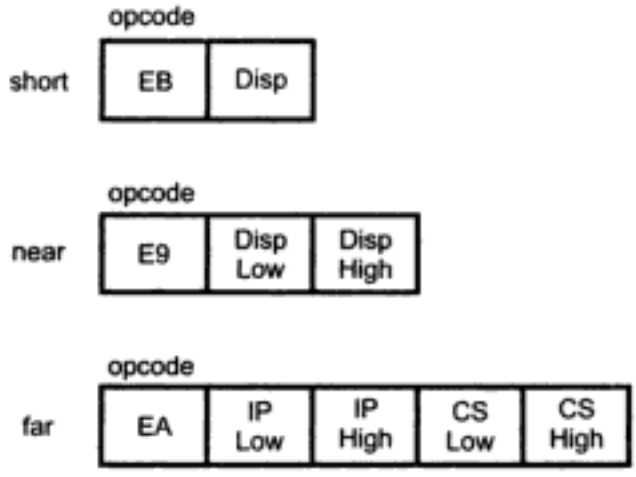


Fig. 3.20 Instruction formats for short, near and far jumps

Examples : (Unconditional jump)

```
JMP NEXT            ; Fetch next instruction from address at label NEXT.
                    ; If label is in same segment, an offset coded as part of
                    ; the instruction will be added to the instruction pointer
                    ; to produce the new fetch address. If the label is in
                    ; another segment then IP and CS will be replaced with
                    ; values coded in as part of the instruction.
                    ; This type of jump is referred to as direct
                    ; because the displacement of the destination or the
                    ; destination itself is specified directly in the instruction.
JMP BX              ; Replace the contents of IP with the contents of BX.
                    ; BX must first be loaded with the offset of the
                    ; destination instruction in CS. This is a near jump. It is
                    ; referred to as an indirect jump because the new value
                    ; for IP comes from a register rather than from the
                    ; instruction itself as in a direct-type jump.
JMP WORD PTR [BX]   ; Replace IP with a word from a memory location
                    ; pointed to by BX in DS. This is an indirect near jump.
JMP DWORD PTR [SI]  ; Replace IP with word pointed to by SI in DS.
                    ; Replace CS with word pointed to by SI + 2 in DS.
                    ; This is an indirect far jump.
```

As explained earlier, a near type jump instruction can cause the next instruction to be fetched from anywhere in the current code segment. To produce the new instruction fetch address, this instruction adds a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer register. A 16 bit signed displacement means that the jump can be to a location anywhere from +32,767 to –32,768 bytes from the current instruction pointer location. A positive displacement usually means jump is "ahead" in the program, and a negative displacement usually means jump is "backward" in the program.

A special case of the direct near jump instruction is direct short jump. If the destination for the jump is within a displacement range of +127 to –128 bytes from the current instruction pointer location, the destination can be reached with just an 8 bit displacement.

3.7.3 Cond - Conditional Jump

Conditional jumps are always short jumps in the 8086. These instructions will cause a jump to a label given in the instruction if the desired condition(s) occurs in the program before the execution of the instruction. The destination must be in the range of –128 bytes to +127 bytes from the address of the instruction after the conditional transfer instruction. If the jump is not taken, execution simply goes on to the next instruction.

Instruction Code	Description	Condition for jump
JA/JNBE	Jump if above/Jump if not below or equal.	CF = 0 and ZF = 0
JAE/JNB	Jump if above or equal/Jump if not below.	CF = 0 and ZF = 1
JB/JNAE/JC	Jump if below/Jump if not above or equal.	CF = 1 and ZF = 0
JBE/JNA	Jump if below or equal/Jump if not above.	CF = 1 and ZF = 1
JE/JZ	Jump if equal/Jump if zero flag.	ZF = 1
JG/JNLE	Jump if greater/Jump if not less than nor equal.	ZF = 0 and CF = 0
JGE/JNL	Jump if greater than or equal/Jump if not less than.	SF = 0
JL/JNGE	Jump if less than/Jump if not greater than or equal.	SF ≠ 0
JLE/JNG	Jump if less than or equal/Jump if not greater	ZF = 1 or SF ≠ 0
JNC	Jump if no carry	CF = 0
JNE/JNZ	Jump if not equal/Jump if not zero	ZF = 0
JNO	Jump if no overflow	OF = 0
JNP/JPO	Jump if not parity/Jump if parity odd	PF = 0
JNS	Jump if not sign or jump if positive	SF = 0
JO	Jump if overflow flag = 1.	OF = 1
JP/JPE	Jump if parity/Jump if parity even	PF = 1
JS	Jump if sign flag = 1 or jump if negative	SF = 1
JCXZ	Jump if CX is zero	CX = 0

Note : The terms greater and less are used to refer to the relationship of two signed numbers.

3.8 Iteration Control Instructions

These instructions are used to execute a series of instructions some number of times. The number is specified in the CX register. The CX register is automatically decremented by one, each time after execution of LOOP instruction. Until CX = 0, execution will jump to a destination specified by a label in the instruction.

The destination address for the jump must be in the range of – 128 bytes to + 127 bytes from the address of the instruction after the iteration control instruction. For LOOPE/LOOPZ and LOOPNE/LOOPNZ instructions there is one more condition for exit from loop, which is given below. If the loop is not taken, execution simply goes on to the next instruction after the iteration control instruction.

Instruction Code	Description	Condition for Exit
1. LOOP	Loop through a sequence of instructions	CX = 0
2. LOOPE/LOOPZ	Loop through a sequence of instructions	CX = 0 or ZF = 0
3. LOOPNE/LOOPNZ	Loop through a sequence of instructions	CX = 0 or ZF = 1

3.9 Processor Control Instructions

- STC
- CLC
- CMC
- STD
- CLD
- STI
- CLI

STC Instruction :

This instruction sets the carry flag, STC does not affect any other flag.

CLC Instruction :

This instruction resets the carry flag to zero. CLC does not affect any other flag.

CMC Instruction :

This instruction complements the carry flag. CMC does not affect any other flag.

STD Instruction :

This instruction is used to set the direction flag to one so that SI and/or DI can be decremented automatically after execution of string instructions. STD does not affect any other flag.

CLD Instruction :

This instruction is used to reset the direction flag to zero, so that SI and/or DI can be incremented automatically after execution of string instructions. CLD does not affect any other flag.

STI Instruction :

This instruction sets the interrupt flag to one. This enables INTR interrupt of the 8086. STI does not affect any other flag.

CLI Instruction :

This instruction resets the interrupt flag to zero. Due to this 8086 will not respond to an interrupt signal on its INTR input. CLI does not affect any other flag.

3.10 External Hardware Synchronization Instructions

- HLT
- WAIT
- ESC
- LOCK
- NOP

HLT Instruction :

The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input.

WAIT Instruction :

When this instruction executes, the 8086 enters an idle condition where it is doing no processing. The 8086 will stay in this idle state until a signal is asserted on the 8086 TEST input pin, or until a valid interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the execution of interrupt service procedure. WAIT affects no flags. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 math coprocessor.

ESC Instruction :

This instruction is used to pass instructions to a coprocessor such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the escape instruction. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the

action specified by the 6-bit code specified in the instruction. In most cases the 8086 treats the ESC instruction as a NOP. In some cases the 8086 will access a data item in memory for the coprocessor.

LOCK Instruction :

In a multiprocessor system each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drives or memory. Each microprocessor only takes control of the system bus when it needs to access some system resources. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller device which then prevents any other processor from taking over the system bus. LOCK affects no flags.

Examples :

```
LOCK  XCHG  SEMAPHORE, AL   ; The XCHG instruction requires two bus accesses.
                            ; The LOCK prefix prevents another processor
                            ; from taking control of the system bus between
                            ; the two accesses.
```

NOP Instruction :

At the time of execution of NOP instruction, no operation is performed except fetch and decode. It takes three clock cycles to execute the instruction. NOP instruction does not affect any flag. This instruction is used to fill in time delays or to delete and insert instructions in the program while trouble shooting.

3.11 Interrupt Instructions

INT Instruction : INT Type

This instruction causes the 8086 to call a far procedure. The term type in the instruction refers to a number between 0-255 which identifies the interrupt. The address of the procedure is taken from the memory whose address is four times the type number.

INTO Instruction :

If the overflow flag is set, this instruction will cause the 8086 to do an indirect far call to a procedure you write to handle overflow condition. To do call the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

IRET Instruction :

The IRET instruction is used at the end of the interrupt service routine to return execution to the interrupted program. The 8086 copies return address from stack into IP and CS registers and the stored value of flags back to the flag register.

Note : The RET instruction does not copy the flags from the stack back to the flag register.

3.12 Assembler Directives

There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker, and loader. These are referred to as **pseudo-operations** or as **assembler directives.** The assembler directives enable us to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.

There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

ALIGN : The align directive forces the assembler to align the next segment at an address divisible by specified divisor. The general format for this directive is as shown below.

ALIGN number

where number can be 2, 4, 8 or 16.

Example :

```
ALIGN 8   ; This forces the assembler to align the next segment
          ; at an address that is divisible by 8. The assembler fills
          ; the unused bytes with 0 for data and NOP instructions
          ; for code.
```

Usually ALIGN 2 directive is used to start the data segment on a word boundary and ALIGN 4 directive is used to start the data segment on a double word boundary.

ASSUME : The 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment and an extra segment. The 8086 may contain a number of logical segments. The ASSUME directive assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the assembler what addresses will be in the segment registers at execution time.

Example : ASSUME CS : code, DS : Data, SS : stack.

.CODE : This directive provides shortcut in definition of the code segment. General format for this directive is as shown below.

.code [name]

The name is optional. It is basically specified to distinguish different code segments when there are multiple code segments in the program.

.DATA : This directive provides shortcut in definition of the data segment.

DB, DW, DD, DQ, and DT : These directives are used to define different types of variables, or to set aside one or more storage locations of corresponding data type in memory. Their definitions are as follows :

DB - Define Byte

DW - Define Word

DD - Define Doubleword

DQ - Define Quadword

DT - Define Ten Bytes

Example :

```
AMOUNT DB 10H, 20H, 30H, 40H      ; Declare array of 4 bytes named
                                  ; AMOUNT
MES DB 'WELCOME'                  ; Declare array of 7 bytes and
                                  ; initialize with ASCII codes for letters in
                                  ; WELCOME.
```

DUP : The DUP directive can be used to initialize several locations and to assign values to these locations.

Format : Name Data_Type Num DUP (value)

Example :

```
TABLE DW 10 DUP (0)               ; Reserve an array of 10
                                  ; words of memory and initialize all 10
                                  ; words with 0. Array is named TABLE.
```

END : The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler ignores any statement after an END directive.

EQU : The EQU directive is used to redefine a data name or variable with another data name, variable, or immediate value. The directive should be defined in a program before it is referenced.

Formats :

Numeric Equate : name EQU expression

String Equate : name EQU <string>

Example :

```
PORT EQU 80                       ; Numeric value
NUM EQU <'Enter the first number  :'>
MES DB NUM                        ; Replace with string
```

EVEN : EVEN tells the assembler to advance its location counter if necessary so that the next defined data item or label is aligned on an even storage boundary. This feature makes processing more efficient on processors that access 16 or 32 bits at a time.

Example :

```
EVEN LOOKUP DW 10 DUP (0)      ; Declares the array of ten words
                               ; starting from even address.
```

EXTRN : The EXTRN directive is used to inform assembler that the names or labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in a program module assembled at a different time, you must tell the assembler that the procedure is EXTRN. The assembler will then put information in the object code file so that the linker can connect the two modules together. For a reference it is necessary to specify whether the label is near or far.

NOTE : Names and labels referred to as external in one module must be declared public.

Example :

```
CALLING PROGRAM          CALLED PROGRAM
DATA SEGMENT             EXTRN VAR : FAR
PUBLIC VAR               DATA SEGMENT
VAR DW                      . .
 . .                     MOV AX, VAL
DATA ENDS                   . .
 . .                     DATA ENDS
```

GROUP : A program may contain several segments of the same type (code, data, or stack). The purpose of the GROUP is to collect them all under one name, so that they reside within one segment, usually a data segment.

Format : Name GROUP Seg-name, . . . , Seg-name.

Example :

```
SEG GROUP SEG1, SEG2
SEG1 SEGMENT PARA 'DATA'
ASSUME DS : SEG
 . . .
SEG1 ENDS
SEG2 SEGMENT PARA 'DATA'
ASSUME DS : SEG
 . . .
SEG2 ENDS
```

LABEL : Assembler uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The label directive can be used to specify destination for jump or call instruction or to specify reference to a data item. When label is used as destination for a jump or a call, then the label must be specified as type near or

as type far. When label is used to refer a data item it must be specified as type byte, type word, or type double word.

Example :

```
NEXT    LABEL FAR       ; Can jump to NEXT from
                        ; another segment
NEXT :  MOV AX, BX      ; Cannot do far jump directly to a label
                        ; with a colon.
                        ; Initialization of stack pointer using
                        ; label directive
```

LENGTH : It is an operator which tells the assembler to determine the number of elements in some named data item such as a string or array.

Example :

```
MOV BX, LENGTH STRING1      ; Loads the Length of string in BX
```

MACRO and ENDM : The macros in the programs can be defined by MACRO directive. ENDM directive is used along with the MACRO directive. ENDM defines the end of the macro.

.MODEL : It is available in MASM version 5.0 and above. This directive provides short-cuts in defining segments. It is initializes memory model before defining any segment. The memory model can be SMALL, MEDIUM, COMPACT or LARGE. We can choose the memory model based on our requirement by referring following table.

Model	Code segments	Data segments
Small	One	One
Medium	Multiple	One
Compact	One	Multiple
Large	Multiple	Multiple

Table 3.5

NAME : The name directive is used at the start of a source program to give specific names to each assembly module.

OFFSET : It is an operator which tells the assembler to determine the offset or displacement of a named data item (variable) from the start of the segment which contains it.

Example :

```
MOV AX, OFFSET MES1     ; Loads the offset of variable, MES1 in
                        ; AX register
```

ORG : The assembler uses a location counter to account for its relative position in a data or code segment.

Format : ORG expression

Example : ORG 1000H, Set the location counter to 1000H

PTR : PTR is used to assign a specific type to a variable or to a label. It is also used to override the declared type of a variable.

Example :
```
WORD_LEN DW
        . .
        . .
        . .
    MOV BL, BYTE PTR WORD_LEN   ; Byte accesses byte from word
```

PAGE : The PAGE directive helps to control the format of a listing of an assembled program. At the start of a program the PAGE directive specifies the maximum number of lines to list on a page and the maximum number of characters on a line.

Format : PAGE [length], [width]

Example : PAGE 52, 132 ; 52 lines per page and 132 characters per line

PROC and ENDP :

PROC : The procedures in the programs can be defined by PROC directive. The procedure name must be present, must be unique, and must follow naming conventions for the language. After the PROC directive the term NEAR or FAR are issued to specify the type of the procedure.

Example : FACT PROC FAR ; Identifies the start of a procedure named FACT and tells the assembler that the procedure is far (in a segment with a different name from that which contains the instruction which calls the procedure)

ENDP : ENDP directive is used along with the PROC directive. ENDP defines the end of the procedure.

PUBLIC : Large programs are usually written as several separate modules. Each module is individually assembled, tested and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared public in the module where it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules.

Format : PUBLIC Symbol [. . . .]

Example : PUBLIC SETPT ; Makes SETPT available for other modules.

SEGMENT and ENDS : An assembly program in .EXE format consists of one or more segments. The start of these segments are defined by SEGMENT directive and the ENDS statement indicates the end of the segment.

Format : name SEGMENT [options] ; Begin segment

.

.

name ENDS ; End segment

Example :
```
CODE  SEGMENT
        .
        .
        .
CODE  ENDS
```

SHORT : A short is a operator. It is used to tell the assembler that only 1-byte displacement is needed to code a jump instruction. If the jump destination is after the jump instruction in the program, the assembler will automatically reserve 2-bytes for the displacement. Using the short operator saves 1-byte of memory by telling the assembler that it only needs to reserve 1-byte for this particular jump. The short operator should be used only when the destination is in the range of –128 bytes to +127 bytes from the address of the instructions after the jump.

Example : JMP SHORT NEAR_LABEL

.STACK : This directive provides shortcut in definition of the stack segment. General format for this directive is as shown below.

.stack [size]

The default size is 1024 bytes.

Example : .STACK 100 ;This reserves 100 bytes for the stack operation.

When stack is not used in the program .stack command can be omitted. This will reserve in the warning message "no stack segment" after linking the program. This warning may be ignored.

TITLE : The TITLE directive help to control the format of a listing of an assembled program. TITLE directive causes a title for a program to print on line 2 of each page of the program listing. Maximum 60 characters are allowed as title.

Format : TITLE text

Example : TITLE Program to find maximum number

TYPE : It is an operator which tells assembler to determine the type of specified variable. Assembler determines the type of specified variable in number of bytes. For byte type variable the assembler gives a value of 1. For word type variable the assembler gives a value of 2 and for double word type variable the assembler gives a value of 4.

3.12.1 Summary of Assembler Directives

Directive	Action
ALIGN	aligns next variable or instruction to byte which is multiple of operand
ASSUME	selects segment register(s) to be the default for all symbol in segment(s)
COMMENT	indicates a comment
DB	allocates and optionally initializes bytes of storage
DW	allocates and optionally initializes words of storage
DD	allocates and optionally initializes doublewords of storage
DQ	allocates and optionally initializes quadwords of storage
DT	allocates and optionally initializes 10-byte-long storage units
END	terminates assembly; optionally indicates program entry point
ENDM	terminates a macro definition
ENDP	marks end of procedure definition
ENDS	marks end of segment or structure
EQU	assigns expression to name
EVEN	aligns next variable or instruction to even byte
EXITM	terminates macro expansion
EXTRN	indicates externally defined symbols
LABEL	creates a new label with specified type and current location counter
LOCAL	declares local variables in macro definition
MACRO	starts macro definition
.MODEL	specifies mode for assembling the program.
ORG	sets location counter to argument
PAGE	sets length and width of program listing; generates page break
PROC	starts procedure definition
PTR	assigns a specific type to a variable or to a label
PUBLIC	identifies symbols to be visible outside module
TITLE	defines the program listing title

Table 3.6

3.12.2 Variables, Suffix and Operators

Variable : A variable is an identifier that is associated with the first byte of data item. In assembly language statement : COUNT DB 20H, COUNT is a variable.

Example : Array DB 10, 20, 30, 40, 50

Here, array is the variable which is associated with the first byte of the data item, i.e. 10.

Suffix : In assembly language programming base of the number of indicated by a suffix as follows :

- B - Binary
- D - Decimal
- O - Octal
- H - Hexadecimal

The default is decimal. The first digit in a hexadecimal number must be 0 through 9; therefore, if the most significant digit is a letter (A–F), then it must be prefixed with a 0.

Examples : 1010 B = 1010_2

2967 D = 2967 = 2967_{10}

3F2AH = $2F2A_{16}$

0B129H = $B129_{16}$

Operators : Arithmetic operators : "+", "–", "*", and "/".

Logical Operators : "AND", "OR", "NOT, and "XOR".

Logical operators are specially used for binary operands.

3.12.3 Accessing a Procedure and Data from another Assembly Module

As mentioned earlier, usually a large program is divided into a series of modules. Each module is individually written, assembled, and tested. The object code files for the modules are then linked together to generate a linked file or executable file.

In order for a linker to be able to access data or a procedure in another assembly module correctly we have to use two assembly language directives : PUBLIC and EXTRN.

In the module where a variable or procedure is declared we must use the PUBLIC directive to let the linker know that the variable or procedure can be accessed from other modules.

In a module which calls a procedure or accesses a variable in another module, we must use the EXTRN directive to let the assembler know that the procedure or variable is not in this module but it has to access from another module. The EXTRN statement also gives the linker some needed information about the procedure. For example : EXTRN ROUTINE : FAR, TOKEN : BYTE tells the linker that ROUTINE is a FAR procedure and TOKEN is a variable of type byte.

➠ **Example 1 :** *"File1.asm" contains a program segment which calls a subroutine (procedure) in "File2.asm". Give the necessary declarations in File1.asm and "File2 asm" (to make the subroutine of file2.asm available to file1.asm which is not locally available) and the assembling and linking to obtain the executable file.*

Solution :

File1.asm

```
EXTRN   ROUTINE : FAR
```

File2.asm

```
PUBLIC ROUTINE PROC FAR
        :
        :
ROUTINE ENDP
```

3.13 Assembly Language Programming

A program is a set of instructions arranged in the specific sequence to do the specific task. It tells the microprocessor what it has to do. The process of writing the set of instructions which tells the microprocessor what to do is called "**Programming**". In other words, we can say that programming is the process of telling the processor exactly how to solve a problem. To do this, the programmer must "**speak**" to the processor in a language which processor can understand.

Steps Involved in Programming

- **Specifying the problem :** The first step in the programming is to find out which task is to be performed. This is called specifying the problem. If the programmer does not understand what is to be done, the programming process cannot begin.
- **Designing the problem-solution :** During this process, the exact step by step process that is to be followed (program logic) is developed and written down.
- **Coding :** Once the program is specified and designed, it can be implemented. Implementation begins with the process of coding the program. Coding the program means to tell the processor the exact step by step process in its language. Each processor has a set of instructions. Programmer has to choose appropriate instructions from the instruction set to build the program.
- **Debugging :** Once the program or a part of program is coded, the next step is debugging the code. Debugging is the process of testing the code to see if it does the given task. If program is not working properly, debugging process helps in finding and correcting errors.

To write a program, programmer should know :

- How to develop program logic?
- How to tell the program to the processor?
- How to code the program?
- How to test the program?

Flow Chart

To develop the programming logic programmer has to write down various actions which are to be performed in proper sequence. The flow chart is a graphical tool that allows programmer to represent various actions which are to be performed. The graphical representation is very useful for clear understanding of the programming logic.

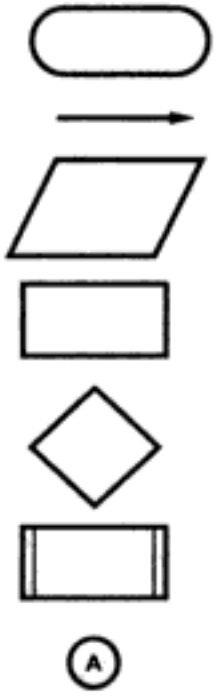


Fig. 3.21 Graphic symbols used in flow chart

The Fig. 3.21 shows the graphic symbols used in the flow chart.

Oval : It indicates start or stop operation.

Arrow : It indicates flow with direction.

Parallelogram : It indicates input/output operation.

Rectangle : It indicates process operation.

Diamond : It indicates decision making operation.

Double sided rectangle : It indicates execution of pre-defined process (subroutine).

Circle with alphabet : It indicates continuation.

A: Any alphabet

The Fig. 3.22 shows sample flow chart.

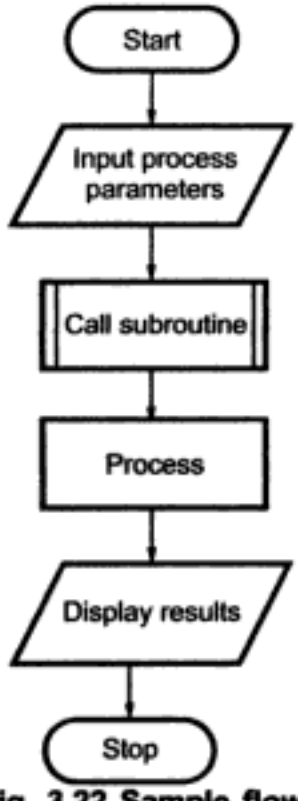


Fig. 3.22 Sample flowchart

3.13.1 Assembly Language Programs

A program which has simply a sequence of the binary codes for the instructions is called machine level language program. This binary form of the program is referred to as machine language because it is the form required by the machine. However, to write a program in machine language, programmer has to memorize the thousands of binary instruction codes for a processor. This task is difficult and error prone.

To make programming easier, usually programmers write programs in assembly language. They then translate the assembly language program to machine language so that it can be loaded into memory and executed. Assembly language uses two, three or four letter words to represent each instruction types. These words are referred to as **mnemonics**. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for addition is ADD, the mnemonic for logic AND operation is AND, and the mnemonic for the instruction for copy data from one location to another is MOV. Therefore, the meaning expressed by mnemonics help us to remember the operation performed by the instruction.

Assembly language statements are usually written in a standard form and assembly language has its own unique syntactical structure, such as requiring upper case or lower case, or requiring colons after label definitions. Here we discuss the common features that assembler shares.

The assembly text is usually divided into fields, separated by spaces and tabs. A format for a typical line from assembly language program can be given as

Label : Mnemonic Operand1, Operand2 ; Comment

The first field, which is optional, is the label field, used to specify symbolic labels. A label is an identifier that is assigned to the address of the first byte of the instruction in which it appears. As mentioned earlier, the presence of a label is optional, but if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction.

The second field is mnemonic, which is compulsory. All instructions must contain a mnemonic. The third and following fields are operands. The presence of the operands depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma.

The last field is a comment field. It begins with a delimiter such as the semicolon and continues to the end of the line. The comments are for our benefits, they tell us what the program is trying to accomplish. Fig. 3.23 shows a typical 8086 assembly language instruction.

Fig. 3.23 Typical assembly language instruction

The Table 3.7 shows the comparison between machine level and assembly languages.

Sr. No.	Machine Language	Assembly Language
1.	Language consists of binary codes which specify the operation.	Language consists of mnemonics which specify the operation.
2.	Processor dependent and hence requires knowledge of internal details of processor to write a program.	Processor dependent hence requires knowledge of internal details of processor to write a program.
3.	Programs require less memory.	Programs require less memory.
4.	Programs have less execution time.	Programs have less execution time.
5.	Program development is difficult.	Program development is simpler than machine language.
6.	It is not user friendly.	It is less user friendly.

Table 3.7 Comparison between various microcomputer languages

3.13.2 Assembly Language Programming Tips

We know that a program is a set of instructions arranged in the specific sequence to perform the specific task. For writing a program for specific task, programmer may find a number of solutions (instruction sequences). A skilled programmer has to choose an optimum solution out of them for that specific task. The technique of choosing an optimum solution is an art and we can name this as an art of assembly language programming. In this section we will see some tips regarding this with the help of examples.

- **What is an optimum solution ? :** The optimum solution is the solution which takes minimum memory space for the program and minimum time for the execution of a task. When we say memory space for the program we consider space for program storage (program length), space for data storage and space used by the stack.
- **Use of proper instructions :** Many times we come across the situation where more than one set of instructions are available to perform particular function. For example, if the function is add 01 in the BX register of 8086 we have two options : ADD BX, 0001H or INC BX. In such situations we must check the space and time for both the options and then select the option which requires less

space and time. Let us see the space and time required for these two instructions. The instruction ADD BX, 0001H is 4 byte instruction and requires 4 clock cycles to execute. On the other hand, INC BX is a single byte instruction and requires 2 cycles for the execution. That is instruction INC BX requires less memory space and execution time than instruction ADD BX, 0001H. Therefore, programmer must use INC BX instruction in such situation.

- **Use of advanced instructions :** We must optimally utilize the processor capabilities. For example, when it is necessary to write a program to move a block of data from the source to destination location, a programmer may initialize a pointer to indicate source location, a pointer to indicate destination location, a counter to count the number of data elements to be transferred. After transfer of one data element from source to destination location programmer may use INC, DEC and JNZ instructions to increment source and destination pointers, decrement counter and to check whether all data elements are transferred or not, respectively.

The same task can be implemented by MOVS instruction supported by 8086. Let us see the part listing of the program with both the approaches and then we compare them.

1. Part listing of program with general approach

```
        MOV SI, 1000H       ; Initialise source pointer
        MOV DI, 2000H       ; Initialise destination pointer
        MOV CX, 0020H       ; Initialise counter
BACK :  MOV AX, [SI]        ; Get data element from source
        MOV [DI], AX        ; Store it at destination
        INC SI              ; Increment source pointer
        INC DI              ; Increment destination pointer
        DEC CX              ; Decrement counter
        JNZ BACK            ; If count is not zero, repeat
```

2. Part listing of program with MOVS instruction

```
        MOV SI, 1000H       ; Initialise source pointer
        MOV DI, 2000H       ; Initialise destination pointer
        MOV CX, 0020H       ; Initialise counter
        CLD                 ; Clear direction flag
   REP  MOVSB               ; Move the entire block
```

Looking at the two programs we can easily notice that the MOVSB instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. Because MOVSB instruction copies multiple bytes from source to destination. After each byte transfer it automatically increments SI and DI pointers by 1 (since DF is 0) and decrements count in CX register and it repeats this process until CX = 0.

In the second approach, we require less number of instructions and memory space. As number of instructions are less, fetching time required for the instructions is also saved and hence we can say that the second approach requires less memory space and less time to execute the same task. So skill programmer uses second approach.

- **Use of proper addressing modes :** We know that the different ways that a processor can access data are referred to as addressing modes. If we compare the various addressing modes reagrding access time required for accessing operands, we can easily make out that the register addressing takes less time to access operand than the index and indirect addressing modes. It is obvious that when operands are available in CPU registers they are immediately available for operation; however when they are in memory we have to fetch them from memory. Fetching operands takes more time. So it is advisible to store most of the operands in the CPU registers. We know that CPU registers are limited in numbers. Therefore, when they are not enough then only we should use memory space for storing the operands.
- **Prepare documentation :** Program must provide enough information so that other users can utilize the program module without having to examine its internal structure. So along with program it is advised to give the following information.

1. Description of the purpose of the program module.
2. In case of subroutine program list of passing parameters and return value.
3. Register and memory locations used.
4. Proper comments for each instruction used.

3.13.3 Programming with an Assembler

Let us see what are the steps involved in developing and executing assembly language programs. Fig. 3.24 shows these steps. The left side of the figure shows the time period, at which each step in the overall process takes place.

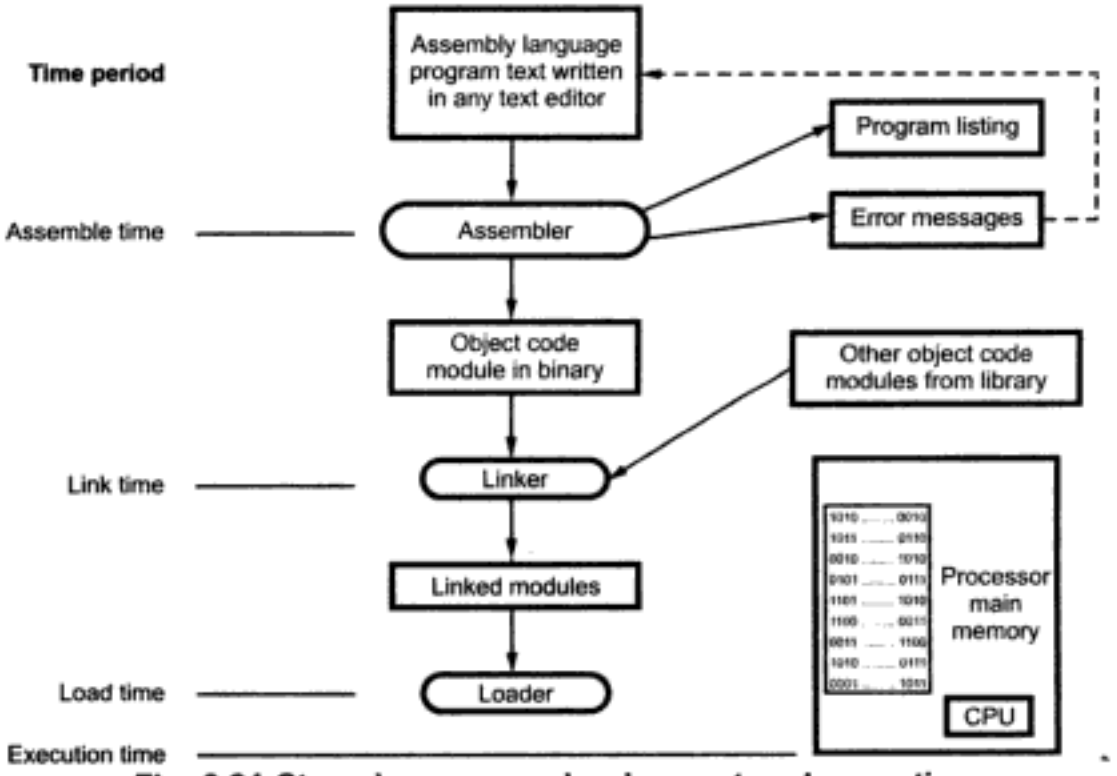


Fig. 3.24 Steps in program development and execution

The first step in the development process is to write an assembly language program. The assembly language program can be written with an ordinary text editor such as word star, edit and so on. The assembly language program text is an input to the assembler. The assembler translates assembly language statements to their binary equivalents, usually known as object code. Time required to translate assembly code to object code is called **assemble time**. During assembling process assembler checks for syntax errors and displays them before giving object code module.

The object code module contains the information about where the program or module to be loaded in memory. If the object code module is to be linked with other separately assembled modules then it contains additional linkage information. At link time, separately assembled modules are combined into one single load module, by the linker. The linker also adds any required initialization or finalization code to allow the operating system to start the program running and to return control to the operating system after the program has completed. Most linkers allow assembly language modules to be linked with object code modules compiled from high-level languages as well. This allows the programmer to insert a time-critical assembly language routines, library modules into a program.

At load time, the program loader copies the program into the computer's main memory, and at execution time, program execution begins.

3.13.3.1 Assembling Process

As mentioned earlier, assembler translates a source file that was created using the editor into machine language such as binary or object code. The assembler reads the source file of our program from the disk where we saved it after editing. An assembler usually reads our source file more than once.

The assembler generates two files on the floppy or hard during these two passes. The first file is called the **object file**. The object file contains the binary codes for the instructions and information about the addresses of the instructions. The second file generated by the assembler is called **assembler list file**. This file contains the assembly language statements, the binary code for each instruction, and the offset for each instruction.

In the first pass, the assembler performs the following operations :

1. Reading the source program instructions.
2. Creating a symbol table in which all symbols used in the program, together with their attributes, are stored.
3. Replacing all mnemonic codes by their binary codes.
4. Detecting any syntax errors in the source program.
5. Assigning relative addresses to instructions and data.

On a second pass through the source program, the assembler extracts the symbol from the operand field and searches for it in the symbol table. If the symbol does not appear in

the table, the corresponding statement is obviously erroneous. If the symbol does appear in the table, the symbol is replaced by its address or value.

We can use a suitable Editor to type .asm file. We can convert object file from .asm file using popular assemblers MASM (Microsoft macro assembler) or TASM (Turbo assembler). The command on command prompt performing this operation is as given below

C:\ MASM\ BIN\> MASM myprog.asm;

where myprog.asm is name of the .asm file which is to be converted to .obj file.

3.13.3.2 Linking Process

A linker is a program used to join together several object files into one large object file. When writing large programs, it is usually much more efficient to divide the large program into smaller modules. Each module can be individually written, tested and debugged. When all the modules work, they can be linked together to form a large functioning program.

The linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map which contains the address information about the link files. The linker, however, does not assign absolute addresses to the program, it only assigns relative addresses starting from zero. This form of the program is said to be relocatable, because it can be put anywhere in memory to be run.

The command on command prompt for converting .obj file to .EXE file is as given below :

C : \ MASM \ BIN \ > LINK myprog.obj;

3.13.3.3 Debugging Process

A debugger is a program which allows us to load our object code program into system memory, execute the program, and debug it.

How does a debugger help in debugging a program ?

1. The debugger allows us to look at the contents of registers and memory locations after our program runs.
2. It allows us to change the contents of register and memory locations and rerun the program.
3. Some debugger allows us to stop execution after each instruction so we can check or alter memory and register contents.
4. A debugger also allows us to set a breakout at any point in our program. When we run a program, the system will execute instructions up to this breakpoint and stop. We can then examine register and memory contents to see if the results are correct at that point. If the results are correct, we can move the break point to a later point in our program. If results are not correct, we can check the program up to that point to find out why they are not correct.

In short, debugger tools can help us to isolate problems in our program.

Debug Commands

Command	Command Syntax and Description
Assembler	– A [address] A command allows you to enter the mnemonic, or human-readable, instructions directly.
Compare	– C range address C command compares two memory blocks.
Dump	– D [range] D command displays a portion of memory in hex and ASCII.
Enter	– E address [list] E command places individual bytes in memory.
Fill	– F range list F command fills a range of memory with a single value or a list of values.
GO	– G [= address] [addresses] G command execute the program in memory.
Hex	– H value 1 value 2 H command performs addition and subtraction on two hexadecimal numbers.
Load	– L [address] [drive] [first sector] [number] L command loads a file (or disk sectors) into memory.
Move	– M range address M command copies a block of data from one memory location to another.
Name	N [pathname] [arglist] N command initializes a filename (and file control block) in memory before using load or write commands.
Proceed	– P [= address] [number] P command traces the program without entering the subroutine or interrupt. If such instruction appears in the program it executes entire subroutine or interrupt routine and immediately proceeds to next instruciton in the sequence.
Quit	– Q Q command quits from debug.
Register	– R [register] R command displays the register contents on the screen.
Search	– S range list S command search a range of addresses for a list of bytes or a string.
Trace	– T [= address] [value] T command execute one or more instructions from the current CS : IP location or optional address, if specified.
Unassemble	– U [range] U command translates memory into assembly language mnemonics.
Write	– W [address] [drive] [first sector] [number] W command write a block of memory to a file or to individual disk sectors.

See detail description of debug command in Appendix C.

3.14 Assembly Language Example Programs

Program 1 : (Softcopy of this program, P1.asm is available at www.vtubooks.com)

```
        NAME      Addition
        PAGE      52,80
        TITLE     8086 assembly language program to add two numbers.
    .model small
    .stack 100
    .data
        No1       DB 63H          ; First number storage
        No2       DB 2EH          ; Second number storage
        Result    DW ?            ; Double byte reserved for result
    .code
    START:    MOV AX,@data        ; [ Initialises
              MOV DS,AX           ;  data segment ]
              MOV AL,NO1          ; Get first number in AL
              ADD AL,NO2          ; Add second to it
              ADC AH,00H          ; Put carry in AH
              MOV Result,AX       ; Copy result to memory
              END START
```

Program 2 : (Softcopy of this program, P2.asm is available at www.vtubooks.com)

```
        NAME      Average
        PAGE      52,80
        TITLE     8086 ALP to find average of two numbers.
    .model small
    .stack 100
    .data
        No1       DB 63H          ; First number storage
        No2       DB 2EH          ; Second number storage
        Avg           DB ?        ; Average of two numbers
    .code
    START:    MOV AX,@data        ; [ Initialises
              MOV DS,AX           ;  data segment ]
              MOV AL,NO1          ; Get first number in AL
              ADD AL,NO2          ; Add second to it
              ADC AH,00H          ; Put carry in AH
              SAR AX,1            ; Divide sum by 2
              MOV Avg,AL          ; Copy result to memory
              END START
```

Program 3 : (Softcopy of this program, P3.asm is available at www.vtubooks.com)

```
        NAME      Maximum number
        PAGE      52,80
        TITLE     8086 ALP to find maximum in the array.
    .model small
    .stack 100
    .data
```

```
   ARRAY  DB 63H,32H,45H,75H,12H,42H,09H,14H,56H,38H
                              ; Array of ten numbers
     MAX   DB 0               ; Maximum number
.code
START:   MOV AX,@data         ; [ Initialises
         MOV DS,AX            ;  data segment ]
         XOR DI,DI            ; Initialise pointer
         MOV CL,10            ; Initialise counter
         LEA BX,ARRAY         ; Initialise base pointer for array
         MOV AL,MAX           ; Get maximum number
BACK:    CMP AL,[BX+DI]       ; Compare number with maximum
         JNC SKIP
         MOV DL,[BX+DI]       ; [ If number > MAX
         MOV AL,DL            ; MAX = number ]
SKIP:    INC DI               ; Increment pointer
         DEC CL               ; Decrement counter
         JNZ BACK             ; IF count = 0 stop
                              ; otherwise go BACK
         MOV MAX,AL           ; Store maximum number
         END START
```

Program 4 : (Softcopy of this program, P4.asm is available at www.vtubooks.com)

```
     NAME Find number
     PAGE     52,80
     TITLE    8086 ALP to search a number in the array.
.model small
.stack 100
.data
  ARRAY  DB 63H,32H,45H,75H,12H,42H,09H,14H,56H,38H
                              ; Array of ten numbers
  SER_NO DB 09H               ; Number to be searched
  SER_POS    DB ?             ; Position of the searched number
.code
START:   MOV AX,@data         ; [ Initialises
         MOV DS,AX            ;  data segment ]
         MOV ES,AX
         MOV CX,000AH         ; Initialise counter
         LEA DI,ARRAY         ; Initialise base pointer for array
         MOV AL,SER_NO        ; Get the number to be searched in AL
         CLD                  ; Clear direction flag
         REPNE SCAS ARRAY     ; Repeat until match occurs or CX = 0
         MOV AL,10            ; [ Find the searched number position
         SUB AL,CL            ;  in the array if SER_POS is 0
         MOV SER_POS,AL       ;  number is not in array; otherwise
                              ;  SER_POS gives the position of
                              ;  number in the array ]
         END START
```

Program 5 : (Softcopy of this program, P5.asm is available at www.vtubooks.com)

```
     NAME     Array sum
     PAGE     52,80
```

```
     TITLE    8086 ALP to find sum of numbers in the array.
.model small
.data
     ARRAY    DB 12H,24H,26H,63H,25H,86H,2FH,33H,10H,35H
     SUM      DW 0
.code
START:   MOV AX,@data      ; [ Initialise
         MOV DS,AX         ;   data segment ]
         MOV CL,10         ; Initialise counter
         XOR DI,DI         ; Initialise pointer
         LEA BX,ARRAY      ; Initialise array base pointer
BACK:    MOV AL,[BX+DI]    ; Get the number
         MOV AH,00H        ; Make higher byte 00h
         ADD SUM,AX        ; SUM = SUM + number
         INC DI            ; Increment pointer
         DEC CL            ; Decrement counter
         JNZ BACK          ; If not 0 go to back
         END START
```

Program 6 : (Softcopy of this program, P6.asm is available at www.vtubooks.com)

```
     NAME     Separate even-odd
     PAGE     52,80
     TITLE    Separate even and odd numbers in the array.
.model small
.STACK 100
.data
     ARRAY        DB 12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H
     ARR_ODD      DB 10 DUP (?)
     ARR_EVEN     DB 10 DUP (?)
.code
START:   MOV AX,@data      ; [ Initialise
         MOV DS,AX         ; data segment ]
         MOV CL,10         ; Initialise counter
         XOR DI,DI         ; Initialise odd_pointer
         XOR SI,SI         ; Initialise even_pointer
         LEA BP,ARRAY      ; Initialise array base_pointer
BACK:    MOV AL,DS:[BP]    ; Get the number
         AND AL,01H        ; Mask all bits except LSB
         JZ  NEXT          ; If LSB = 0 go to next
         LEA BX,ARR_ODD    ; [ Otherwise
         MOV AH,[BX+DI]    ; Initialise pointer to odd array
         MOV ARR_ODD,AH    ; and save number in odd array ]
         INC DI            ; Increment odd_pointer
         JMP SKIP
NEXT:    LEA BX,ARR_EVEN   ; [ Initialise pointer
         MOV AH,[BX+SI]    ;  to even array and save number
         MOV AH,ARR_EVEN   ;  in even array ]
         INC SI            ; Increment even_pointer
SKIP:    INC BP            ; Increment array base_pointer
         DEC CL            ; Decrement counter
```

```
        JNZ BACK            ; If not 0 go to back
        END START
```

It is important to note that programs discussed so far do not accept any input from keyboard and do not display any result on the video screen. This is done purposely to maintain simplicity. To accept input in various formats from keyboard and to display data on the video screen we have to use routines provided by Disk Operating System (DOS). These routines are discussed in Chapter 9. The programs given in the subsequent sections use routines provided by DOS. Therefore, students are suggested to refer these routines before further reading the remaining part of this text.

3.15 Timings and Delays

In the real time applications, such as traffic light control, digital clock, process control, serial communication, it is important to keep a track with time. For example in traffic light control application, it is necessary to give time delays between two transitions. These time delays are in few seconds and can be generated with the help of executing group of instructions number of times. This software timers are also called time delays or software delays. Let us see how to implement these time delays or software delays.

As you know microprocessor system consists of two basic components, hardware and software. The software component controls and operates the hardware to get the desired output with the help of instructions. To execute these instructions, microprocessor takes fix time as per the instruction, since it is driven by constant frequency clock. This makes it possible to introduce delay for specific time between two events. In the following section we will see different delay implementation techniques.

3.15.1 Timer Delay using NOP Instruction

NOP instruction does nothing but takes 3 clock cycles of processor time to execute. So by executing NOP instruction in between two instructions we can get delay of 3 clock cycles.

3.15.2 Timer Delay using Counters

Counting can create time delays. Since the execution times of the instructions used in a counting routine are known, the initial value of the counter, required to get specific time delay can be determined.

```
                                          Clock cycles required
        MOV CX, COUNT     ; Load count              4
BACK :  DEC CX            ; Decrement count         2
        JNZ BACK          ; If count ≠ 0, repeat    16/4
```

In this program, the instructions DEC CX and JNZ BACK execute number of times equal to count stored in the CX register. The time taken by this program for execution can be calculated with the help of clock cycles. The column to the right of the comments indicates the number of clock cycles required for the execution of each instruction. Two values are specified for the number of clock cycles for the JNZ instruction. The smaller

value is applied when the condition is not met, and the larger value is applied when it is met. The first instruction MOV CX, count is executed only once and it requires 4 clock cycles. There are count–1 passes through the loop where the condition is met and control is transferred back to the first instruction in the loop (DEC CX). The number of clock cycles that elapse while CX register is not zero are (count–1) × (2 + 16). On the last pass through the loop the condition is not met and the loop is terminated. The number of clock cycles that elapse in this pass are 2 + 4.

∴ Total clock cycles required to execute the given program

$$= \underbrace{4}_{\text{MOV CX, Count}} + \underbrace{(\text{Count}-1)\times(2+16)}_{\text{Loop}} + \underbrace{(2+4)}_{\text{Last loop}}$$

For count = 100, the number of clock cycles required are

$$= 4 + (100 - 1) \times (2 \times 16) + (2 + 4)$$

$$= 1792$$

Assuming operating frequency of 8086 system 10 MHz,

$$\text{Time required for 1 clock-cycle} = \frac{1}{10\,\text{MHz}} = 0.1\ \mu\text{sec}$$

∴ Total time required for execution of a given program with count equal to 100 is 179.2 μsec (1792 × 0.1).

In the above example, we have calculated the time required for the execution of program or delay introduced by the program when count value is given. However, in most of the situations we know the waiting time or delay time and it is necessary to determine what count should be loaded in the CX register to get the specified delay. Let us consider that we have to generate a delay of 50 ms using an 8086 system that runs at 10 MHz frequency. Then using same program we can calculate the count value as follows :

Step 1 : Calculate the number of required clock cycles

$$\text{Number of required clock cycles} = \frac{\text{Required delay time}}{\text{Time for 1 - clock cycle}}$$

$$= \frac{50\ \text{ms}}{0.1\ \mu\text{s}} = 500\ 000$$

Step 2 : Find the required count

$$\text{Count} = \frac{\text{Number of required clock cycles} - 4 - (2+4)}{\text{Execution Time for one loop}} + 1$$

$$= \frac{500\,000 - 4 - 6}{(16+2)} + 1$$

$$\approx 27778 = 6\text{C82H}$$

3.15.3 Timer Delay using Nested Loops

In this program one more external loop is added to execute the internal loop multiple times. So that we can get larger delays. The inner loop is nothing but the program we have seen in the last section.

```
          MOV BX, Multiplier count   ; Load multiplier count
REPE :    MOV CX, COUNT              ; Load count
BACK :    DEC CX                     ; Decrement count
          JNZ BACK                   ; If count ≠ 0, repeat
          DEC BX                     ; Decrement multiplier count
          JNZ REPE                   ; If not zero repeat
```

In the delay calculations of nested loops, the delay introduced by inner loop is very large in comparison with the delay provided by MOV BX, COUNT, DEC BX and JNZ instructions. Therefore, it is not necessary to consider the last loop for the external loop delay calculations separately. The inner loop delay calculations will remain as it is.

∴Total clock cycles required to execute the given program

$$= [\underbrace{4}_{\text{MOV CX, Count}} + \underbrace{(\text{count}-1)\times(2+16)}_{\text{Loop}} + \underbrace{(2+4)}_{\text{Last loop}}] \times \text{multiplier count}$$

For count = 100 and multiplier count 50, the number of clock cycles required are

$$= [4 + (100 - 1) \times (2 + 16) + (2 + 4)] \times 50$$

$$= 89600$$

Assuming operating frequency of 8086 system 10 MHz,

Total time required for execution of a given program

$$= 89600 \times 0.1\ \mu\text{sec} = 8.96\ \text{ms}$$

Example 2 : *Write an 8086 ALP to generate a delay of 100 ms, if 8086 system frequency is 10 MHz.*

Solution :

Program :

```
          MOV CX, COUNT      ; 4
BACK :    DEC CX             ; 2
          JNZ BACK           ; 16/4
```

Step 1 : Calculate the number of required clock cycles

$$\text{Number of required clock cycles} = \frac{\text{Required delay time}}{\text{Time for 1 clock cycle}}$$

$$= \frac{100\ \text{ms}}{0.1\ \mu\text{s}} = 1000\ 000$$

Step 2 : Find the required count

$$\text{Count} = \frac{\text{Number of required clock cycles} - 4 - (2+4)}{\text{Execution time for one loop}} + 1$$

$$= \frac{1000\,000 - 4 - 6}{(16+2)} + 1$$

$$= 55556 = \text{D904H}$$

➠ **Example 3 :** *Write an 8086 ALP to generate a delay of 1 minute if 8086 system frequency is 10 MHz.*

Solution :

```
Program :
MOV BX, multiplier count
REPE :   MOV CX, Count          ; 4
BACK :   DEC CX                 ; 2
         JNZ BACK               ; 16/4
         DEC BX
         JNZ REPE
```

Step 1 : Calculate the delay generated by inner loop with maximum count (FFFFH)

Delay generated by inner loop for count (FFFFH = 65535)

$$= [4 + (65535 - 1) \times (2 + 16) + (2 + 4)] \times 0.1\ \mu s$$

$$= 118.1422 \text{ msec}$$

Step 2 : Calculate the multiplier count to get delay of 1 minute

$$\text{multiplier count} = \frac{\text{Required delay}}{\text{Delay provided by inner loop}}$$

$$= \frac{1 \times 60\,\text{sec}}{118.1422\,\text{m sec}}$$

$$\approx 509 = \text{1FDH}$$

3.16 Data Conversions

Before going to write and execute any assembly language program on a computer we must understand which type of data processor understands and which type of data user understands, and how they communicate with each other. User communicates with computer using input devices and computer gives outcome of process or result on the display devices or hardcopy devices such as printer or plotter. Most commonly used input device is keyboard and most commonly used output device is a display device, video monitor. These devices understand the information in ASCII format. Keyboard gives the pressed key number or character in its ASCII equivalent and for display certain number or character we have to send the ASCII equivalent of the number or character to the display

device. On the other hand, processor does not understand the ASCII format. It uses binary numbers. Therefore, it is necessary to convert input from keyboard to its binary equivalent (ASCII to binary conversion) and convert processed data by processor into ASCII format for the display (binary to ASCII conversion). Let us see how we can perform these conversions. In this section we study the routines for these conversions. Once we understand these routines we can use these routines to accept input using keyboard and to display data on video monitor.

3.16.1 Routines to Convert Binary to ASCII

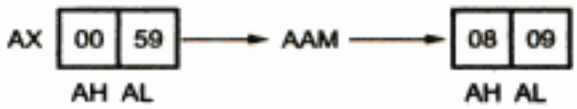
There are two ways to convert binary number into its ASCII equivalent :

- By the AAM instruction if the number is less than 100.
- By a series of decimal divisions (divide by 10).

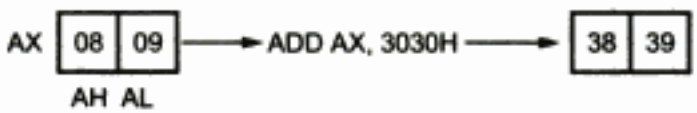
3.16.1.1 By AAM Instruction (For number less than 100)

The AAM instruction converts the value in AX into a two-digit unpacked BCD number in AX. For example, if number in AX is 0059H (89 decimal) before execution of AAM instruction, AX contains 0809H after execution of AAM instruction. Now we can get ASCII equivalent by adding 3030H to AX.

Algorithm :



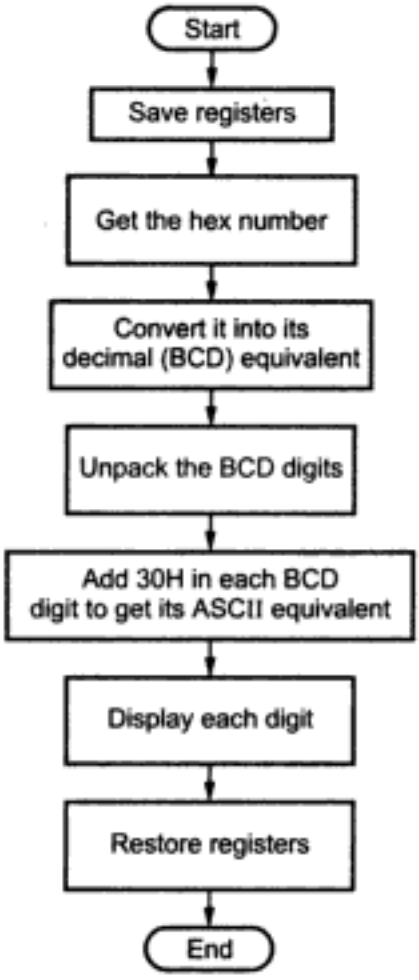
Note : 59H ⟶ 89 Decimal



Note : 38H and 39H are the ASCII equivalents of 8 and 9 respectively

1. Save contents of all registers which are used in the routine.
2. Get the data in AL register and make AH equal to 00.
3. Use AAM instruction to convert number in its decimal equivalent in the unpacked format.
4. Add 30H in each digit to get its ASCII equivalent.
5. Display digit one by one using function 2 of INT 21H.
6. Restore contents of registers.

Flow Chart



Routine : Convert Binary to ASCII for number less than 100

Passing Parameter : Hex number in AL register.

```
; Routine to convert binary number into its
; decimal and then ASCII equivalent, and then display the number

BTA PROC NEAR

     PUSH DX            ; Save registers
     PUSH BX
     PUSH AX

     MOV AH, 00H        ; Clear AH
     AAM                ; Convert to BCD
     ADD AX, 3030H      ; Convert to ASCII
     MOV BX,AX          ; Save result
     MOV DL, BH         ; Load first digit (MSD)
     MOV AH, 02         ; Load function number
     INT 21H            ; Display first digit (MSD)
```

```
    MOV DL, BL              ; Load second digit (LSD)
    INT 21H                 ; Display second digit (LSD)

    POP AX                  ; Restore registers
    POP BX
    POP DX
    RET
    ENDP
```

Sample Program

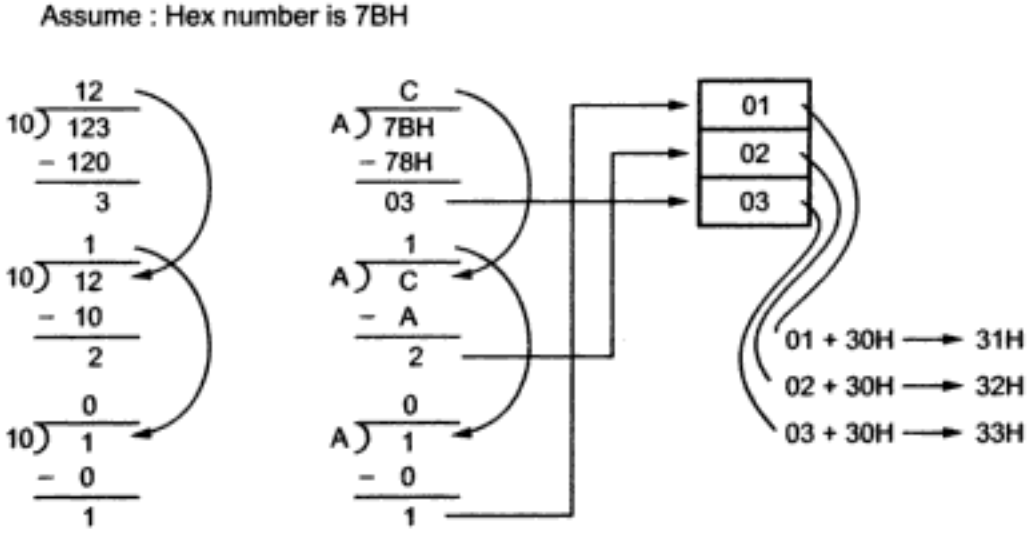
```
; Sample program to convert binary number into its
; decimal and then ASCII equivalent, and then display the number

.MODEL SMALL            ; Select SMALL mode
.STACK 100              ; Initialization of stack
.CODE

    MOV AL,59H          ; Load number in AL
    CALL BTA            ; Call routine
    MOV AH, 4CH         ; [Exit
    INT 21H             ;  to DOS]

BTA PROC NEAR

    PUSH DX             ; Save registers
    PUSH BX
    PUSH AX

    MOV AH, 00H         ; Clear AH
    AAM                 ; Convert to BCD
    ADD AX, 3030H       ; Convert to ASCII
    MOV BX,AX           ; Save result
    MOV DL, BH          ; Load first digit (MSD)
    MOV AH, 02          ; Load function number
    INT 21H             ; Display first digit (MSD)
    MOV DL, BL          ; Load second digit (LSD)
    INT 21H             ; Display second digit (LSD)

    POP AX              ; Restore registers
    POP BX
    POP DX
    RET
    ENDP
    END
```

```
C:\tasm\tasm s_bta.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_bta.asm
Error messages:       None
```

```
Warning messages:       None
Passes:                    1
Remaining memory:       410k

C:\tasm\tlink s_bta.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_bta
89
```

3.16.1.2 By Series of Decimal Division

If number is greater than 99 we can not use AAM instruction to convert given number in the BCD format. In such case we use scheme of dividing by 10 to convert any whole number from binary to an ASCII coded character string that can be displayed on the video monitor.

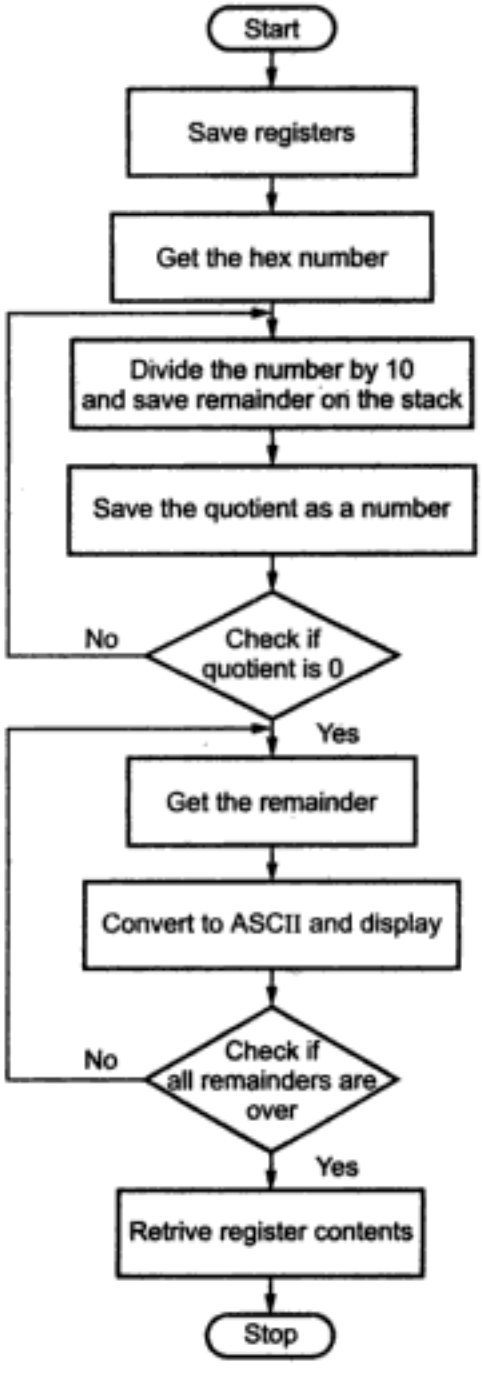


Let us see the algorithm for converting number from binary to ASCII code.

Algorithm

1. Save contents of all registers which are used in the routine.
2. Divide the number by 10 and save the remainder on the stack as a significant BCD digit.
3. Save the quotient as a number.
4. Repeat step 1 and 2 until quotient is 0.
5. Retrieve each remainder from stack and add 30 H to convert to ASCII before displaying or printing.
6. Restore contents of registers.

Flowchart



Routine : Convert Binary to ASCII

Passing parameter : 4-digit hex number in AX register.

```
; Routine to convert 4-digit hex into its decimal
; and then to ASCII equivalent, and display it

BTA4D PROC NEAR

          PUSH DX                    ; Save registers
          PUSH CX
```

```
        PUSH BX
        PUSH AX
        MOV CX, 0              ; Clear digit counter
        MOV BX, 10             ; Load 10 decimal in BX
BACK:   MOV DX, 0              ; Clear DX
        DIV BX                 ; Divide DX : AX by 10
        PUSH DX                ; Save remainder
        INC CX                 ; Counter remainder
        OR AX, AX              ; Test if quotient equal to zero
        JNZ BACK               ; If not zero divide again
        MOV AH, 02H            ; Load function number
DISP:   POP DX                 ; Get remainder
        ADD DL, 30H            ; Convert to ASCII
        INT 21H                ; Display digit
        LOOP DISP
        POP AX                 ; Restore registers
        POP BX
        POP CX
        POP DX
        RET
        ENDP
        END
```

Sample Program

```
; Sample program to convert 4-digit hex into its decimal
; and then to ASCII equivalent, and display it

.MODEL SMALL            ; Select SMALL model
.STACK 100              ; Initialise stack segment
.CODE
        MOV AX,2ABCH
        CALL BTA4D          ; Call routine
        MOV AH, 4CH         ; [Exit
        INT 21H             ;  to DOS]
BTA4D   PROC NEAR
        PUSH DX             ; Save registers
        PUSH CX
        PUSH BX
        PUSH AX
        MOV CX, 0           ; Clear digit counter
        MOV BX, 10          ; Load 10 decimal in BX
BACK:   MOV DX, 0           ; Clear DX
        DIV BX              ; divide DX : AX by 10
        PUSH DX             ; Save remainder
        INC CX              ; Counter remainder
        OR AX, AX           ; Test if quotient equal to zero
        JNZ BACK            ; If not zero divide again
        MOV AH, 02H         ; Load function number
DISP:   POP DX              ; Get remainder
        ADD DL, 30H         ; Convert to ASCII
```

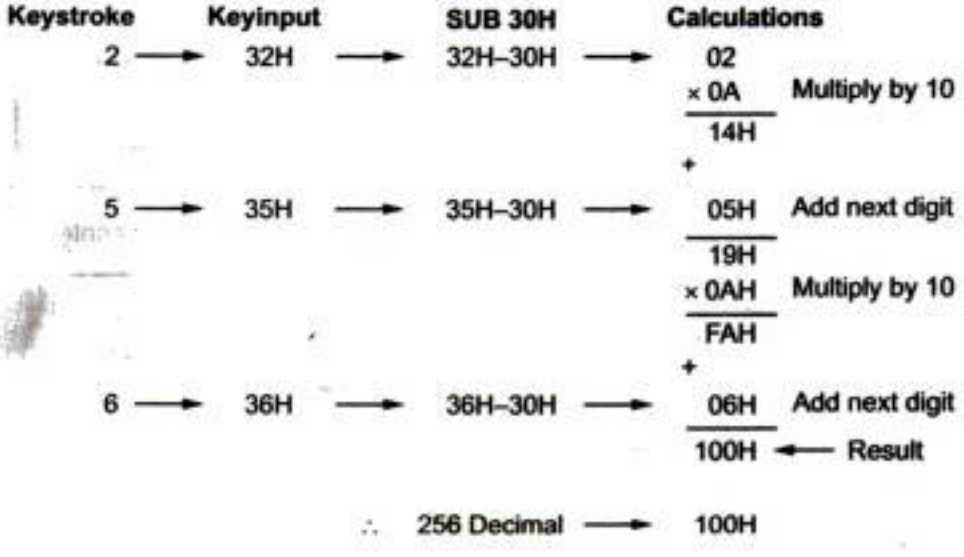
```
        INT 21H             ; Display digit
        LOOP DISP
        POP AX              ; Restore registers
        POP BX
        POP CX
        POP DX
        RET
        ENDP
        END
```

```
C:\tasm\tasm s_bta4d.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_bta4d.asm
Error messages:       None
Warning messages:     None
Passes:                  1
Remaining memory:     410k

C:\tasm\tlink s_bta4d.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_bta4d
10940
```

3.16.2 Routine to Convert ASCII to Binary

When we accept decimal number from keyboard we get ASCII code of each decimal digit. This information from the keyboard must be converted from ASCII to binary. When a single key is pressed conversion can be achieved by subtracting 30H. However, when more than one key is typed conversion from ASCII to binary requires 30H to be subtracted, but there is additional step. After subtracting 30H, the number is added to the result after the prior result is first multiplied by 10.

∴ 256 Decimal → 100 H

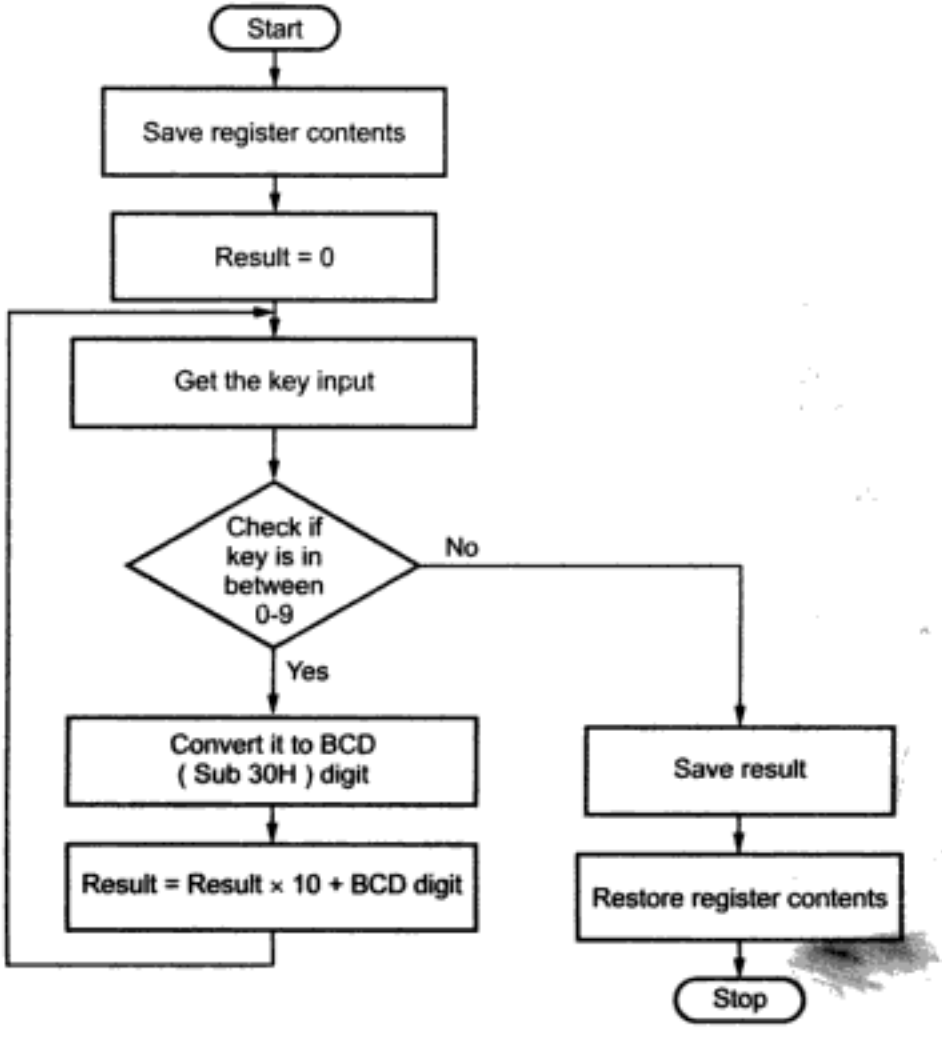


Let us see the algorithm for converting number from ASCII to binary code

Algorithm

1. Save contents of all registers which are used in the routine.
2. Make binary result = 0.
3. Subtract 30H from the character typed on the keyboard to convert it to BCD.
4. Multiply the result by 10, and then add the new BCD digit.
5. Repeat steps 2 and 3 until the character typed is not an ASCII coded number.
6. Restore register contents.

Flowchart



Routine : Convert BCD number from keyboard to its Hex equivalent.

```
; Routine to convert ASCII coded decimal from keyboard into its HEX
equivalent

ATB PROC NEAR
          PUSH CX              ; Save registers
          PUSH BX
          PUSH AX
          MOV CX, 10           ; Load 10 decimal in CX
          MOV BX, 0            ; Clear result
BACK:     MOV AH,01H           ; [Read key
          INT 21H              ; with echo]
          CMP AL,'0'
          JB SKIP              ; Jump if below '0'
          CMP AL,'9'
          JA SKIP              ; Jump if above '9'
          SUB AL, 30H          ; Convert to BCD
          PUSH AX              ; Save digit
          MOV AX, BX
          MUL CX               ; Multiply previous result by 10
          MOV BX, AX           ; Get the result in BX
          POP AX               ; Retrieve digit
          MOV AH, 00H
          ADD BX, AX           ; Add digit value to result
          JMP BACK             ; Repeat
SKIP:     MOV NUMBER,BX        ; Save the result in NUMBER
          POP AX               ; Restore registers
          POP BX
          POP CX
          RET
          ENDP
```

Sample Program

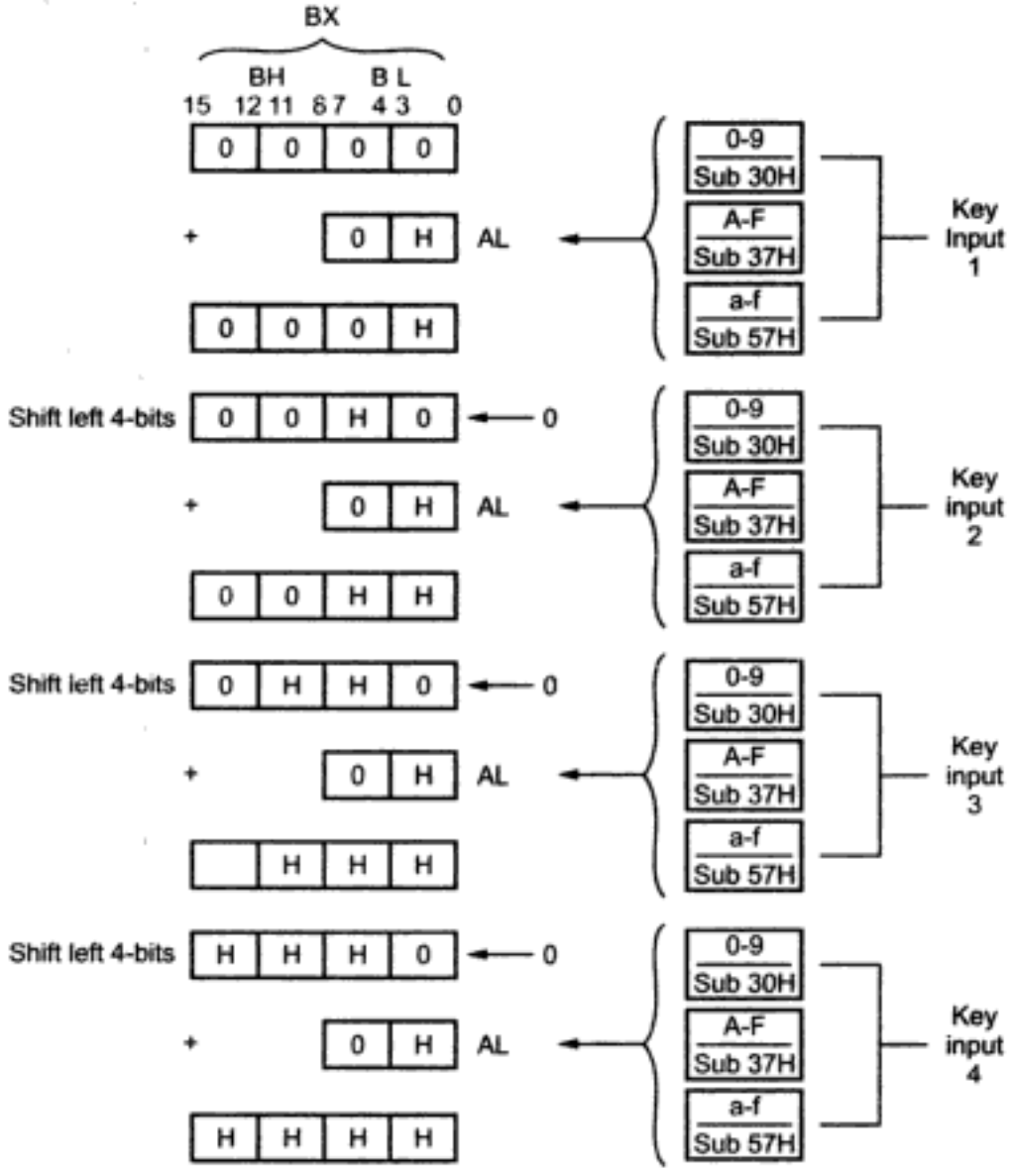
```
; Sample program to convert ASCII coded decimal from keyboard into
its HEX equivalent

.MODEL SMALL
.DATA
    NUMBER DW ?                ; Define number
.CODE
START:    MOV AX, @DATA        ; [Initialize
          MOV DS, AX           ; data segement]
          CALL ATB             ; convert ASCII coded decimal from
                               ; keyboard into its HEX equivalent

          MOV AH, 4CH          ; [Exit to
          INT 21H              ;   DOS]
ATB PROC NEAR
          PUSH CX              ; Save registers
          PUSH BX
          PUSH AX
          MOV CX, 10           ; Load 10 decimal in CX
          MOV BX, 0            ; Clear result
BACK:     MOV AH,01            ; [Read key
          INT 21H              ;  with echo]
          CMP AL,'0'
          JB SKIP              ; Jump if below '0'
          CMP AL,'9'
```

```
        JA SKIP             ; Jump if above '9'
        SUB AL, 30H         ; Convert to BCD
        PUSH AX             ; Save digit
        MOV AX, BX          ; Multiply previous result by 10
        MUL CX
        MOV BX, AX          ; Get the result in BX
        POP AX              ; Retrieve digit
        MOV AH, 00H
        ADD BX, AX          ; Add digit value to result
        JMP BACK            ; Repeat
SKIP:   MOV NUMBER,BX       ; Save the result in NUMBER

        POP AX              ; Restore registers
        POP BX
        POP CX
        RET
        ENDP
        END
```

```
C:\tasm\tasm s_atb.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_atb.asm
Error messages:       None
Warning messages:     None
Passes:                  1
Remaining memory:     410k

C:\tasm\tlink s_atb.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_atb
1234
```

3.16.3 Routine to Read Hexadecimal Data

We know that hexadecimal numbers range from 0 to 9 and from A to F. The keyboard gives ASCII codes for these hexadecimal numbers. It gives 30H to 39H for numbers 0 to 9 and gives 41H to 46H for A to F letters or gives 61H to 66H for a to f letters. Hence, to convert ASCII input from keyboard to corresponding hexadecimal number we have to first check whether it is a number or letter and then if letter whether it is a small letter or capital letter and accordingly convert it into hexadecimal number.

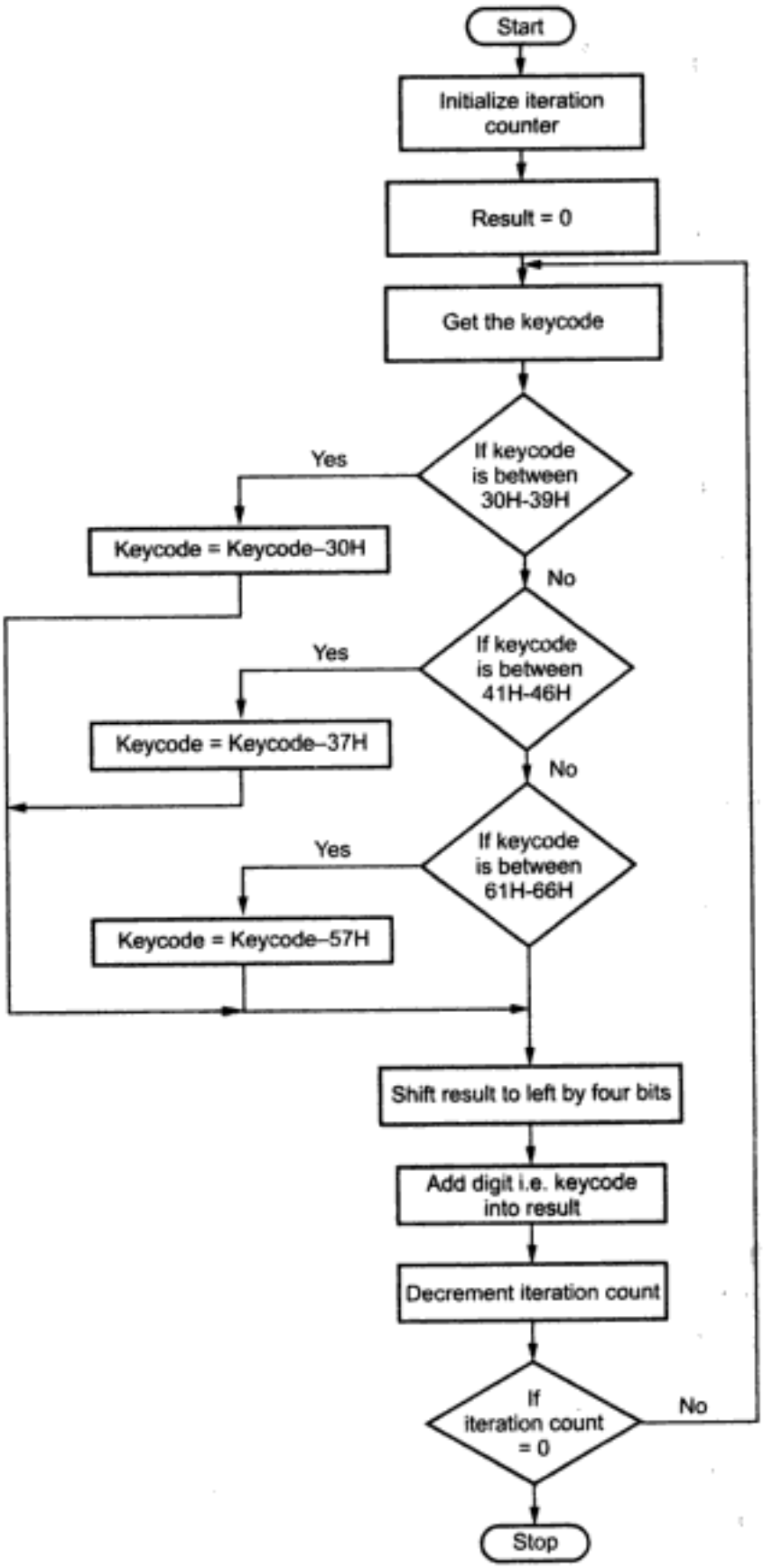


Note : H represents any hexadecimal digit (0-F).

Algorithm

1. Save registers
2. Make result = 0
3. Get the ASCII code of character from keyboard and
   - Subtract 30H from it if character is 0 – 9
   - Subtract 37H from it if character is A – F
   - Subtract 57H from it if character a – f
4. Shift the result by 4-bits and add digit to pack binary digits.
5. Repeat steps 2 and 3 four times to get 4-digit hex number.
6. Restore registers.

Flowchart



Routine : Reading hexadecimal data

Returns : Hex number in variable number

```
; Routine to read 4-digit Hex number from the keyboard

R_HEX PROC NEAR

          PUSH CX                ; Save registers
          PUSH BX
          PUSH AX
          PUSH SI
          MOV CL, 04             ;  Load shift count
          MOV SI, 04             ;  Load iteration count
          MOV BX, 0              ;  Clear result
BACK:     MOV AH, 01             ; [Read a key
          INT 21H                ;  with echo]
          CALL CONV              ;  convert to binary
          SHL BX, CL             ; [pack four
          ADD BL, AL             ;  binary digits
          DEC SI                 ;  as 16-bit
          JNZ BAC                ;  number]
          MOV NUMBER, BX         ; Save result at NUMBER
          POP SI                 ; Restore registers
          POP AX
          POP BX
          POP CX
          RET
          ENDP

; The procedure to convert contents of AL into hexadecimal
equivalent

CONV PROC NEAR
               CMP AL,'9'
               JBE SUBTRA30      ; If number is between 0 through 9
               CMP AL,'a'
               JB SUBTRA37       ; If letter is uppercase
               SUB AL, 57H       ; Subtract 57H if letter is lowercase
               JMP LAST1
SUBTRA30:      SUB AL, 30H       ; Convert number
               JMP LAST1
SUBTRA37:      SUB AL, 37H       ; Convert uppercase letter
LAST1:         RET
CONV           ENDP
```

Sample Program

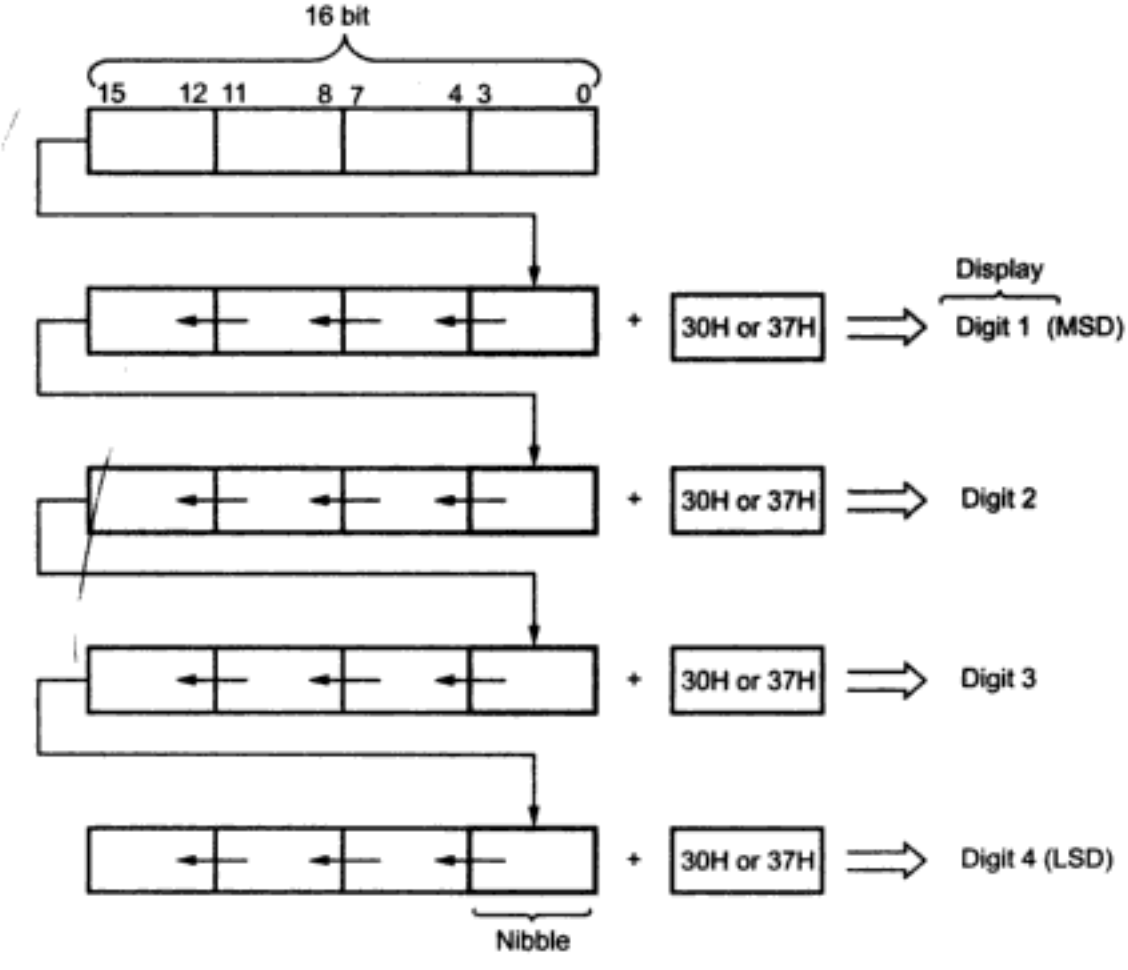
```
; Sample example to read 4-digit Hex number from the keyboard

.MODEL SMALL                     ; Select small model
.STACK 100                       ; Initialise stack
```

```
.DATA                          ; Start data segment
        NUMBER DW?             ; Define NUMBER
.CODE                          ; Start code segment
START:MOV AX, @DATA            ; [Initialize
        MOV DS, AX             ;  data segment]
        CALL R_HEX             ; Read 4-digit hex number
        MOV AH, 4CH            ; [Exit to
        INT 21H                ;  DOS]
R_HEX PROC NEAR
        PUSH CX                ; Save registers
        PUSH BX
        PUSH AX
        PUSH SI
        MOV CL, 04             ;  Load shift count
        MOV SI, 04             ;  Load iteration count
        MOV BX, 0              ;  Clear result
BAC:    MOV AH, 01             ; [Read a key
        INT 21H                ;  with echo]
        CALL CONV              ;  Convert to binary
        SHL BX, CL             ; [Pack four
        ADD BL, AL             ; binary digits
        DEC SI                 ;  as 16-bit
        JNZ BAC                ;  number]
        MOV NUMBER, BX         ; Save result at NUMBER
        POP SI                 ; Restore registers
        POP AX
        POP BX
        POP CX
        RET
        ENDP

; The procedure to convert contents of AL into hexadecimal
equivalent

CONV PROC NEAR
            CMP AL,'9'
            JBE SUBTRA30       ; If number is between 0 through 9
            CMP AL,'a'
            JB SUBTRA37        ; If letter is uppercase
            SUB AL, 57H        ; Subtract 57H if letter is
                               ; Lowercase
            JMP LAST1
SUBTRA30:   SUB AL, 30H        ; Convert number
            JMP LAST1
SUBTRA37:   SUB AL, 37H        ; Convert uppercase letter
LAST1:      RET
CONV        ENDP
            END
```

```
C:\tasm\tasm s_rdhex.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_rdhex.asm
Error messages:       None
Warning messages:     None
Passes:                   1
Remaining memory:     410k

C:\tasm\tlink s_rdhex.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_rdhex
   12AB
```

3.16.4 Routine to Display Hexadecimal Data

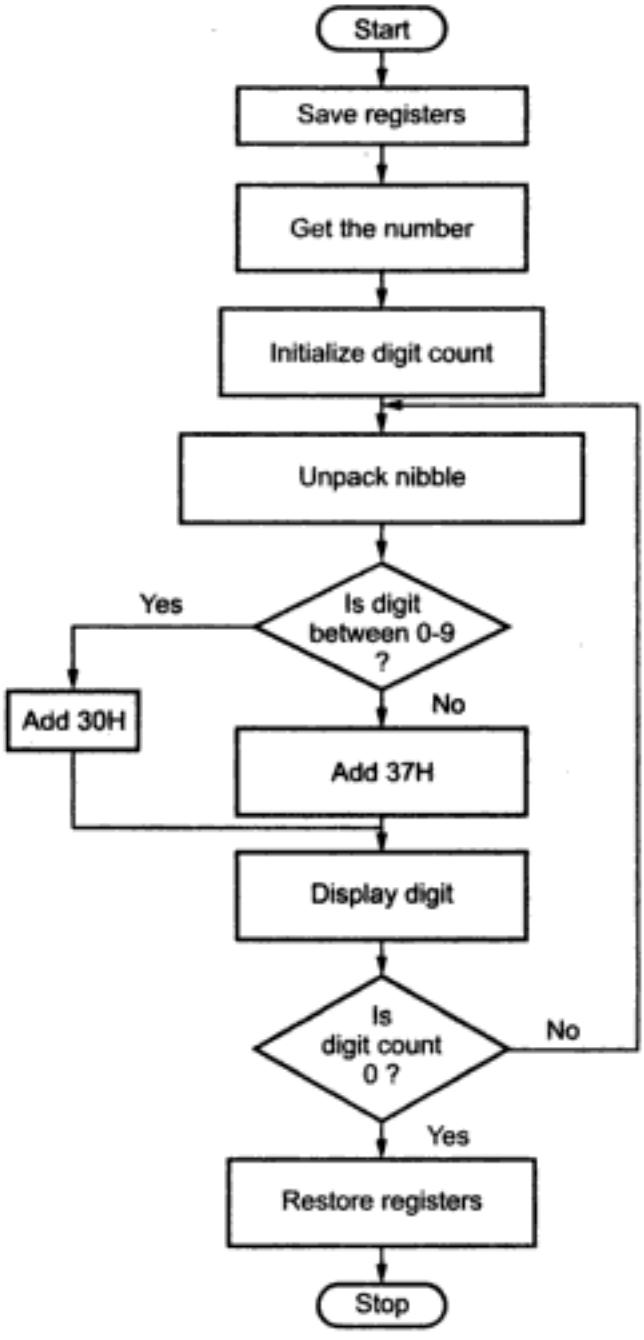
To display hexadecimal data we have to first unpack each digit (nibble) in the given number. Then by adding 30H to digit having number between 0 to 9 and by adding 37H to digit having letter between A to F we can get the ASCII equivalent of given hexadecimal number. This can be achieved by rotating number left (nibble by nibble) and adding 30H or 37H into it. By rotating left we can display left most digit (MSD) first.



Algorithm

1. Save registers.
2. Get the number and unpack digit from it.
3. Add 30H if digit is 0 – 9 or add 37H if digit is A – F to get the ASCII code of digit.
4. Display digit.
5. Repeat steps 2, 3 and 4.
6. Restore registers.

Flowchart



Routine

```
; Routine to display 4-digit hex number in AX

D_HEX PROC NEAR

     PUSH DX            ; Save registers
     PUSH CX
     PUSH AX
     MOV CL, 04H        ; Load rotate count
     MOV CH, 04H        ; Load digit count
BACK:    ROL AX, CL     ; Rotate digits
     PUSH AX            ; Save contents of AX
     AND AL, 0FH        ; [Convert
     CMP AL,9           ;  number
     JBE ADD30          ; to
     ADD AL, 37H        ; its
     JMP DISP           ; ASCII
ADD30:
     ADD AL,30H         ; equivalent]
DISP:    MOV AH,02H
     MOV DL,AL          ; [Display the
     INT 21H            ;   number]
     POP AX             ; Restore contents of AX
          DEC CH        ; Decrement digit count
          JNZ BACK      ; If not zero repeat

          POP AX        ; Restore registers
          POP CX
          POP DX

          RET
          ENDP
```

Sample Program

```
; Sample program displays 4-digit hex number in AX

.MODEL SMALL
.STACK 100
.CODE

     MOV AX, 12ABH      ; Load AX with test data

     CALL D_HEX         ; Call procedure

     MOV AH, 4CH        ; [Exit
     INT 21H            ;  to DOS]
```

```
D_HEX PROC NEAR

    PUSH DX          ; Save registers
    PUSH CX
    PUSH AX

    MOV CL, 04H      ; Load rotate count
    MOV CH, 04H      ; Load digit count
BACK:  ROL AX, CL    ; Rotate digits
    PUSH AX          ; Save contents of AX
    AND AL, 0FH      ; [Convert
    CMP AL,9         ;  number
    JBE ADD30        ; to
    ADD AL, 37H      ; its
    JMP DISP         ; ASCII
ADD30:  ADD AL,30H   ; equivalent]
DISP:   MOV AH,02H
    MOV DL,AL        ; [Display the
    INT 21H          ;  number]
    POP AX           ; Restore contents of AX
    DEC CH           ; Decrement digit count
    JNZ BACK         ; If not zero repeat

    POP AX           ; Restore registers
    POP CX
    POP DX

    RET
    ENDP
    END
```

```
C:\tasm\tasm s_d_hex.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_d_hex.asm
Error messages:       None
Warning messages:     None
Passes:                  1
Remaining memory:     410k

C:\tasm\tlink s_d_hex.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_d_hex
12AB
```

3.16.5 Lookup Tables for Data Conversions

For certain data conversion, when number of possible data conversions are small in numbers then lookup tables are often used to convert data from one form to another. For example, for conversion of BCD to 7-segment code there are only 10 possible conversions. A lookup table is nothing but a array form in the memory as a list of data that is referenced by a procedure to perform conversions.

Converting from BCD to 7-segment code

Let us see how to perform BCD to 7-segment code conversion. For BCD to 7-segment code conversion a lookup table contains the 7-segment codes for the numbers 0 to 9. These codes are determined from Fig. 3.25. The 7-segment display shown in Fig. 3.25 uses active high (logic 1) input to light a segment. The code is formed by placing the a segment in the bit position 0 and the g segment in the bit position 6. It position 7 is kept 0.

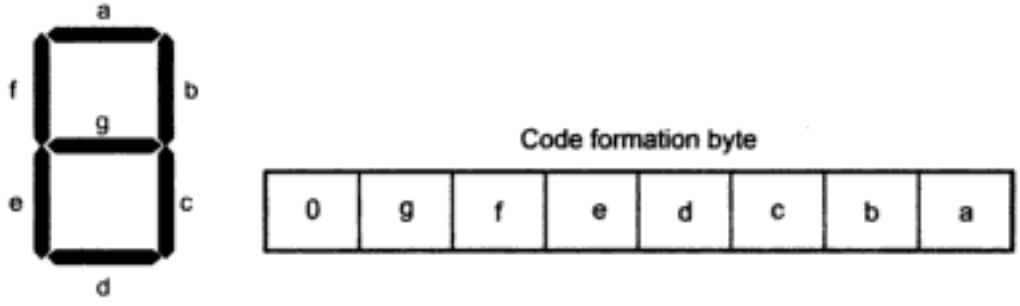


Fig. 3.25 7-segment code formation

A look-up table can be stored in the program memory (code segment) or in the data memory (data segment). Let us see the program which uses lookup table stored in the data memory to convert BCD code into its 7-segment equivalent code.

Program statement : Write an assembly language program to convert BCD to 7-segment code.

Program

```
        .MODEL SMALL
        .DATA
        TABLE DB 3FH        ; 0
              DB 06H        ; 1
              DB 5BH        ; 2
              DB 4FH        ; 3
              DB 66H        ; 4
              DB 6DH        ; 5
              DB 7DH        ; 6
              DB 07H        ; 7
              DB 7FH        ; 8
              DB 6FH        ; 9
        .CODE
START:    MOV AX, @DATA     ; [Initialize
          MOV DS, AX        ;  Data segment]
          MOV AL, 08H       ; Loads AL with any BCD digit,
                            ; for example 8, to be converted to
                            ; 7-segment code
        MOV BX,OFFSET TABLE ; Load BX with the offset of
                            ; starting address of lookup table
```

```
        XLAT TABLE              ; Copy byte from address pointed by
                                ; [BX + AL] back into AL
        MOV AH, 4CH             ; [Exit
        INT 21H                 ;  to DOS]
        END START
        END
```

Note : When look-up table is stored in the code segment we have to include a segment override prefix in the XLAT instruction because XLAT instruction by default access, byte from data segment. To access byte from code segment we have modify XLAT instruction as XLAT CS : TABLE.

Look-up table to access ASCII data

Many program require that numeric codes to be converted to ASCII character strings. For example, if we need to display month in the text format we should use lookup table to reference the ASCII coded months of the year. Let us see program to access ASCII string corresponding to given month of the year using look-up table stored in the data segment.

Program statement : Write an assembly language program to access ASCII string corresponding to given month of the year.

Program:

```
        .MODEL SMALL
        .DATA
        DPOINTER DW JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
                 DW OCT, NOV, DIC
             JAN DB        'JANUARY $'
             FEB DB        'FEBRUARY $'
             MAR DB        'MARCH $'
             APR DB        'APRIL $'
             MAY DB        'MAY $'
             JUN DB        'JUNE $'
             JUL DB        'JULY $'
             AUG DB        'AUGUST $'
             SEP DB        'SEPTEMBER $'
             OCT DB        'OCTOBER $'
             NOV DB        'NOVEMBER $'
             DIC DB        'DECEMBER $'
        .CODE
    START:   MOV AX,@ DATA      ; [Initialize
             MOV DS, AX         ;  Data segment]
             MOV AL, 07H        ; Loads AL with any month in its
                                  numerical value
        MOV SI, OFFSET DPOINTER ; Address table find month of year
        MOV AH,00H              ; [Multiply the AL by 2
        ADD AX, AX              ;  to point to correct
        ADD SI, AX              ;  month of the year]
        MOV DX, [SI]            ;  Get month of year
        MOV AH, 09H             ; [Display month
```

```
INT 21H                    ;  of year string]
MOV AH,4CH                 ; [Exit
INT 21H                    ;  to DOS]
END START
END
```

3.17 Procedures

Whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them. One way is to write the group of instructions as a separate procedure. We can then just CALL the procedure whenever we need to execute that group of instructions. For calling the procedure we have to store the return address onto the stack. This process takes some time. If the group of instructions is big enough then this overhead time is negligible with respect to execution time. But if the group of instructions is too short, the overhead time and execution time are comparable. In such cases, it is not desirable to write procedures. For these cases, we can use macros. Macro is also a group of instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a procedure. The disadvantage of macro is that it generates inline code each time when the macro is called which takes more memory. In this section we discuss the procedures.

From the above discussions, we know that the procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called **near procedure** otherwise it is referred to as **far procedure**. For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program and procedure. But for far procedures CALL instruction pushes both IP and CS on the stack. Let us see the detail description and examples of CALL instruction to enter the procedure and RET instruction to return from the procedure.

CALL Instruction :

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A **far CALL** is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack

pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

Examples :

Direct within segment (near)

```
CALL PRO          ; PRO is the name of the procedure.
                  ; The assembler determines displacement of pro
                  ; from the instruction after the CALL and codes
                  ; this displacement in as part of the instruction.
```

Indirect within-segment (near)

```
CALL CX           ; CX contains, the offset of the first instruction
                  ; of the procedure. Replaces contents of IP with
                  ; contents of register CX.
```

Indirect to another segment (far)

```
CALL DWORD PTR [BX]  ; New values for CS and IP are fetched from four
                     ; memory locations in DS. The new value for CS
                     ; is fetched from [BX] and [BX + 1], the new IP
                     ; is fetched from [BX + 2] and [BX + 3].
```

RET Instruction :

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

A RET instruction can be followed by a number, for example, RET 4. In this case the stack pointer will be incremented by an additional four addresses after the IP or the IP and CS are popped off the stack. This form is used to increment the stack pointer up over parameters passed to the procedure on the stack.

Flags : The RET instruction affects no flags.

3.17.1 Reentrant Procedure

In some situations it may happen that procedure1 is called from main program, procedure2 is called from procedure1 and procedure1 is again called from procedure2. In this situation program execution flow reenters in the procedure1. This type of procedures are called **reentrant procedures**. The flow of program execution for reentrant procedure is shown in Fig. 3.26.

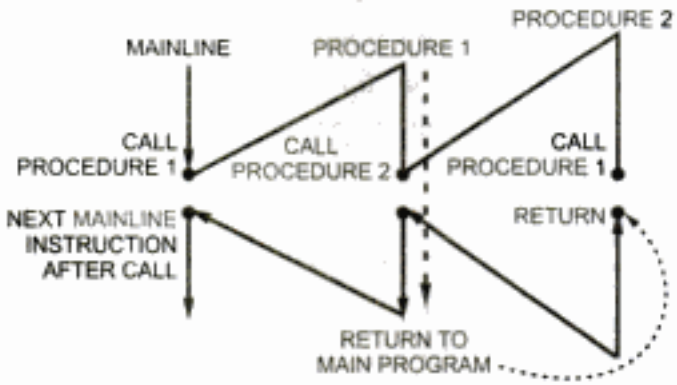


Fig. 3.26 Flow of program execution for reentrant procedure

3.17.2 Recursive Procedure

A recursive procedure is a procedure which calls itself. Recursive procedures are used to work with complex data structures called trees. If the procedures is called with N (recursion depth) = 3. Then the n is decremented by one after each procedure CALL and the procedure is called until n = 0. Fig. 3.27 shows the flow diagram and pseudo-code for recursive procedure.

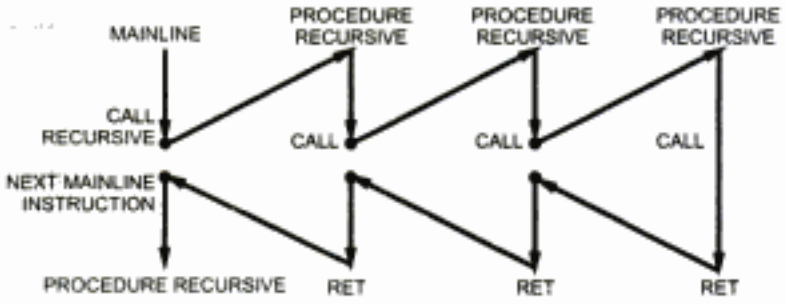


Fig. 3.27 Flow diagram and pseudo-code for recursive procedure

3.18 Macro

Macro is a group of instructions. The macro assembler generates the code in the program each time where the macro is 'called'. Macros can be defined by MACRO and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can be used in the program, as shown below.

Example : Macro definition for initialization of segment registers.

```
INIT MACRO          ; Define macro
MOV AX, @data       ;
MOV DS              ;   Body of macro definition
MOV ES, AX          ;
ENDM                ;  End macro
```

It is important to note that macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as **Macro expansion.**

Comparison of Procedure and Macro

Sr. No.	Procedure	Macro
1.	Accessed by CALL and RET instruction during program execution.	Accessed during assembly with name given to macro when defined.
2.	Machine code for instructions is put only once in the memory.	Machine code is generated for instructions each time when macro is called.
3.	With procedures less memory is required.	With macros more memory is required.
4.	Parameters can be passed in registers, memory locations, or stack.	Parameters passed as part of statement which calls macro.

Table 3.8

Passing Parameters in Macro

In Macro, parameters are passed as a part of statement which calls Macro.

Example :

```
PROMPT MACRO MESSAGE ;Define macro with MESSAGE as a parameter
        MOV AH, 09H
        LEA MESSAGE
        INT 21H
        ENDM                    ;End macro
        DATA
        MES1     DB 10, 13, 'Student Name : $'
        MES2     DB 10, 13, 'Student Address : $'
    .CODE
START:  MOV AX, @data           ; [ Initialize
        MOV DS, AX              ;   data segment ]
```

```
        PROMPT MES1                ; Display MES1
        PROMPT MES2                ; Display MES2
        MOV AH,4CH                 ; Return to DOS
        INT 21H
        END START
```

The above example shows that parameters can be passed in macro with the help of dummy argument. Argument tells the assembler to match its name with any occurrence of the same name in the macro body. For example the dummy argument MESSAGE also occurs in the LEA instruction. The macro instruction "PROMPT MES1" passes the MES1 as a parameter and macro accepts that as an argument.

Local Variables in a Macro

Body of the Macro can use local variables. A local variable defined in the Macro is available in the Macro, however it is not available outside the Macro. To define a local variable, LOCAL directive is used. Example shows how local variable is used as a jump address. If this jump address is not defined as a local, the assembler give an error message on the second and subsequent attempts to use the Macro.

Example

```
DISPLAY  MACRO A           ; Displays ASCII character in uppercase
             LOCAL J_LABEL; Defines J_LABEL as local
             PUSH DX
             CMP AL,'Z'
             JBE J_LABEL   ; Check if uppercase
             SUB AL,20H    ; Convert to uppercase
   J_LABEL:  MOV DL,AL
             MOV AH,02H
             INT 21H
             POP DX
             ENDM
```

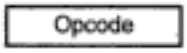
The above Macro accepts ASCII code for character. (A-Z or a-z). If it is for lowercase character, Macro converts it to uppercase character and displays the uppercase character on video screen.

It is important to note that local variable or variables must be defined using LOCAL directive immediately after MACRO directive.

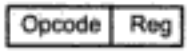
3.19 Instruction Formats

The instructions of 8086 vary from 1 to 6 bytes in length. Fig. 3.28 shows the instruction formats for 1 to 6 bytes instruction for each instruction format first field is the operation code field, commonly known as opcode field. Opcode field indicates the type of operation to be performed by the processor. The other field in the instruction format is operand field. The operand field may consists of source/destination operand, source operand address, destination operand address or next instruction address. The operand and the relative address of the operand (displacement) may be either 8-bit or 16-bit long depend on the instruction and its addressing mode.

One byte instruction - implied operands



One byte instruction register mode



Register to register

Register to/from memory with no displacement

Register to/from memory with displacement (8-bit)

Register to/from memory with displacement (16-bit)

Immediate operand to register (8-bit)

Immediate operand to register (16-bit)

Immedia.e operand to memory with 16-bit displacement

Fig. 3.28 Sample 8086 instruction formats

The opcode and the addressing mode is specified using first two bytes of an instruction. The opcode/addressing mode byte(s).

The opcode/addressing mode byte(s) may be followed by :

- No additional byte
- Two byte EA (For direct addressing only).
- One or two byte displacement
- One or two byte immediate operand
- One or two byte displacement followed by a one or two byte immediate operand

Two byte displacement and a two byte segment address (for direct intersegment addressing only).

Most of the opcodes in 8086 has a special 1-bit indicators. They are :

W-bit : Some instructions of 8086 can operate on byte or a word. The W-bit in the opcode of such instruction specify whether instruction is a byte instruction (W = 0) or a word instruction (W = 1).

D-bit : The D-bit in the opcode of the instruction indicates that the register specified within the instruction is a source register (D = 0) or destination register (D = 1).

S-bit : An 8-bit 2's complement number can be extended to a 16-bit 2's complement number by making all of the bits in the higher-order byte equal the most significant bit in the low order byte. This is known as sign extension. The S-bit along with the W-bit indicate :

S	W	Operation
0	0	8-bit operation
0	1	16-bit operation with 16-bit immediate operand
1	0	–
1	1	16-bit operation with a sign extended 8-bit immediate operand

Table 3.9

V-bit : V-bit decides the number of shifts for rotate and shift instructions. If V = 0, then count = 1; if V = 1, the count is in CL register. For example, if V = 1 and CL = 2 then shift or rotate instruction shifts or rotates 2-bits.

Z-bit : It is used for string primitives such as REP for comparison with ZF Flag. If it is 1, the instruction with REP prefix is executed until the zero flag matches the Z-bit.

(Refer Appendix A for instruction formats)

As seen from the Fig. 3.28 if an instruction has two opcode/addressing mode bytes, then the second byte is of one of the following two forms :

MOD	Opcode	R/M

or

MOD	Reg	R/M

where Mod, Reg and R/M fields specify operand as described in the following tables.

Mode		Displacement
0	0	Disp = 0 Low order and High order displacement are absent
0	1	Only Low order displacement is present with sign extended to 16-bits.
1	0	Both Low-order and High-order displacements are present.
1	1	r/m field is treated as a 'Reg' field.

Table 3.10 'Mod' field assignments

Word Operand (W = 1)		Byte Operand (W = 0)		Segment	
0 0 0	AX	0 0 0	AL	0 0	ES
0 0 1	CX	0 0 1	CL	0 1	CS
0 1 0	DX	0 1 0	DL	1 0	SS
0 1 1	BX	0 1 1	BL	1 1	DS
1 0 0	SP	1 0 0	AH		
1 0 1	BP	1 0 1	CH		
1 1 0	SI	1 1 0	DH		
1 1 1	DI	1 1 1	BH		

Table 3.11 'Reg' field assignment

R/M	Operand Address
0 0 0	EA = [BX] + [SI] + Displacement (optional)
0 0 1	EA = [BX] + [DI] + Displacement (optional)
0 1 0	EA = [BP] + (SI) + Displacement (optional)
0 1 1	EA = [BP] + [DI] + Displacement (optional)
1 0 0	EA = [SI] + Displacement (optional)
1 0 1	EA = [DI] + Displacement (optional)
1 1 0	EA = [BP] + Displacement (optional)
1 1 1	EA = [BX] + Displacement (optional)

Table 3.12 'R/M' field assignment

➠ **Example 4 :** *Write the instruction format for PUSH BX instruction.*

Solution : This instruction will put BX register contents on stack. Referring the table in Appendix A we find that the 5-bit opcode for this instruction is 01010. We put 011 in the REG field to represent the BX register. The codes for each registers are shown in table 3.11. The resultant code for PUSH BX will be 01010011.

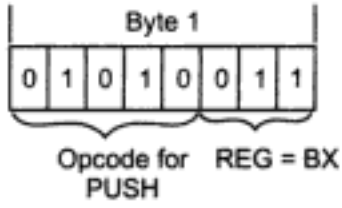


Fig 3.29 Instruction format for PUSH BX

➠ **Example 5 :** *Write the instruction format for MOV AX, CX instruction.*

Solution : This instruction will copy a word from the CX register to the AX register. Referring the table in Appendix A we find the 6-bit opcode for this instruction is 100010. Because we are moving a word, W=1. The D bit for this instruction may be somewhat confusing . Since two registers are involved, we can think of the move as either to AX or from CX. It actually does not matter which we assume as long as we are consistent in coding the rest of the instruction. If we think of the instruction as moving a word to AX, then make D=1 and put 000 in the REG field to represent the AX register. The MOD field will be 11 to represent register addressing mode. We make the R/M field 001 to represent the other register CX. The resultant code for the instruction MOV AX, CX will be 10001011 11000001. The Fig 3.30 shows the meaning of all these bits.

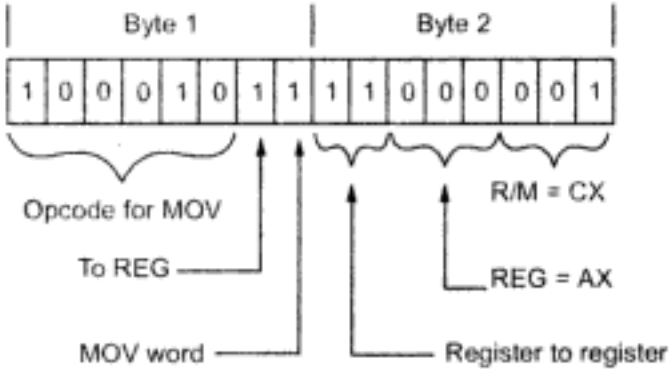


Fig. 3.30 Instruction format for MOV AX, CX

If we change D field to a 0 and swap the codes in the REG and R/M field, we will get 10001001 11001000, which is another equally valid code for the instruction.

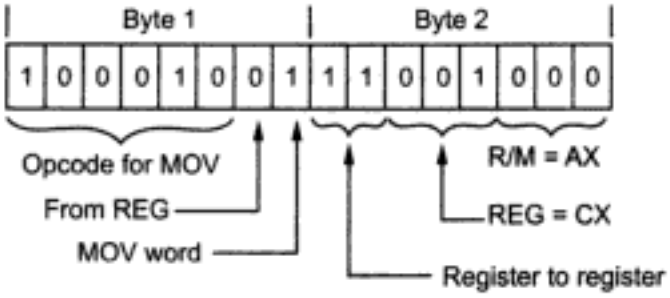


Fig. 3.31 Alternative instruction format for MOV AX, CS

➠ **Example 6 :** *Write the instruction format for MOV 56H[SI], BH*

Solution : This instruction will copy a byte from the BH register to a memory location. The BIU will compute the effective address of the memory location by adding the indicated displacement of 56H to the contents of SI register. The BIU then produce the physical address by adding the effective address with the base represented by 16-bit contents of DS register. The 6-bit opcode for this instruction is again 100010. We put 111 in the REG field to represent the BH register. D = 0 because we are moving data from BH register. W = 0 because we are moving a byte. The R/M field will be 100 because SI contains part of the effective address. The MOD field will be 01 because the displacement contained in the instruction, 56H, will fit in 1 byte. The 8-bit displacement forms the third byte of the instruction. The resultant sequence of code bytes will be 10001000 01111100 01010110.

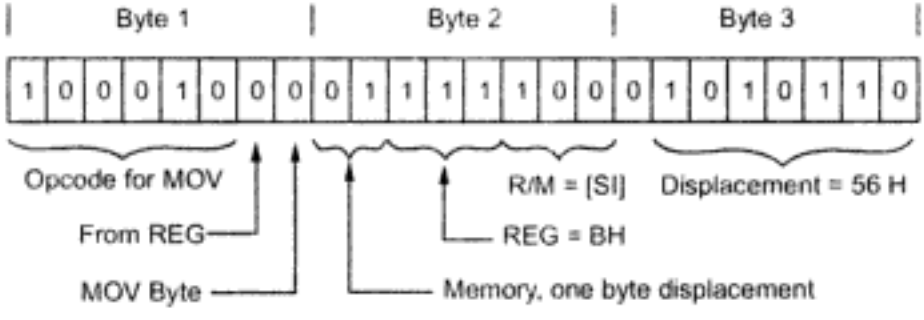


Fig. 3.32 Instruction format for MOV 56H [SI], BH

➠ **Example 7 :** *Write the instruction format for MOV DL, [BX].*

Solution : This instruction will copy a byte to DL from the memory location whose effective address is contained in BX. The effective address will be added to the data segment base in DS to produce the physical address. Referring the table in Appendix A,

we find opcode for this instruction is 100010. We make D = 1 because data is being moved to register DL. We make W = 0 because the instruction is moving a byte into DL. We put 010 in REG field to represent DL register. We make MOD field 00 to represent memory with no displacement. For this instruction R/M field will be 111. The resultant sequence of code bytes will be 1000101000010111.

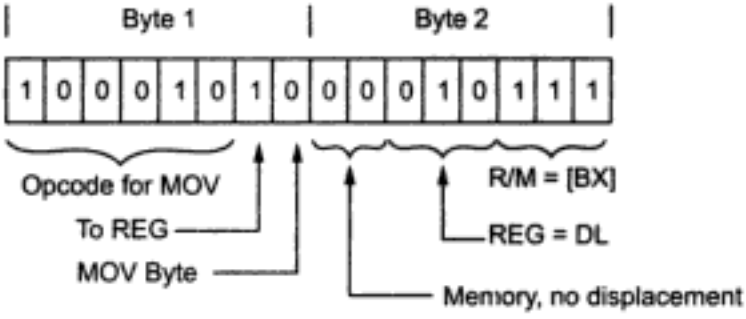


Fig. 3.33 Instruction format for MOV DL, [BX]

Example 8 : *Write the instruction format for MOV BX, [1234 H]*

Solution : This instruction copies the contents of two memory locations into the BX register. The direct address or displacement of the first memory location from the start of the data segment is 1234H. The BIU will produce the physical memory address by adding this displacement to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. We make D = 1 because we are moving data to the BX register, and we make W = 1 because the data being moved is a word. We put 011 in the REG field to represent the BX register. Referring tables 3.11 and 3.12 we get MOD = 00 and R/M field = 110. Then the first two bytes of instruction code will be 10001011 00011110. These two bytes will be followed by the low byte of the direct address, 34H (0011 0100 binary), and the high byte of the direct address, 12H (0001 0010 binary). The instruction will be coded into four successive memory addresses as 8BH, 1EH, 34H and 12H.

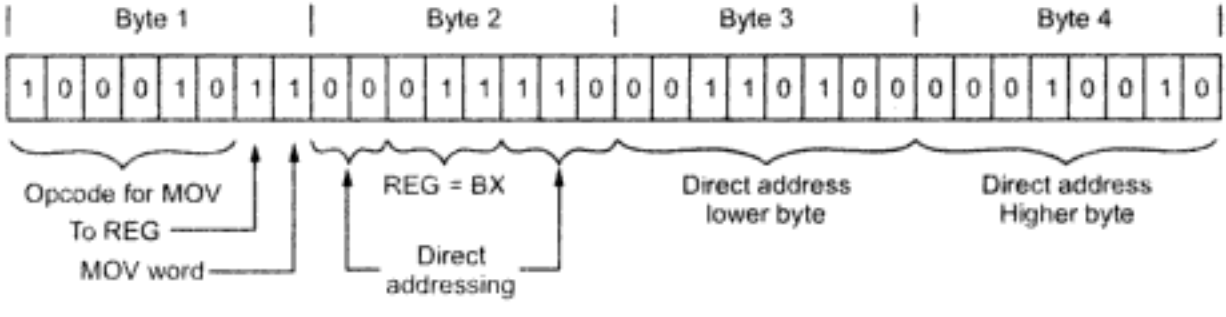


Fig. 3.34 Instruction format for MOV BX, [1234H]

Example 9 : *Write the instruction format for MOV CS : [BX], CL.*

Solution : This instruction copies a byte from the CL register to a memory location. The effective address for the memory location is contained in the BX register. Usually an effective address in BX will be added to the data segment base in DS to produce the physical memory address. In this instruction, the CS in front of [BX] indicates that we want the BIU to add the effective address to the code segment base in CS to produce the physical address. The CS : is called segment override prefix.

When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put in memory before the code for the instruction. The code byte for the segment override prefix has the format 001 XX 110. We can be replace XX with : the segment code. The segment codes are : ES = 00, CS = 01, SS = 10 and DS = 11. The segment override prefix byte for CS, then, is 00101110.

The opcode for this instruction is 100010. D = 0 because we are moving data from the CL register. W = 0 because we are moving a byte. We put 001 in REG field to represent CL register. We make MOD field 00 to represent memory with no displacement. For this instruction R/M field will be 111. The resultant sequence of code bytes will be 00101110 10001000 00001111.

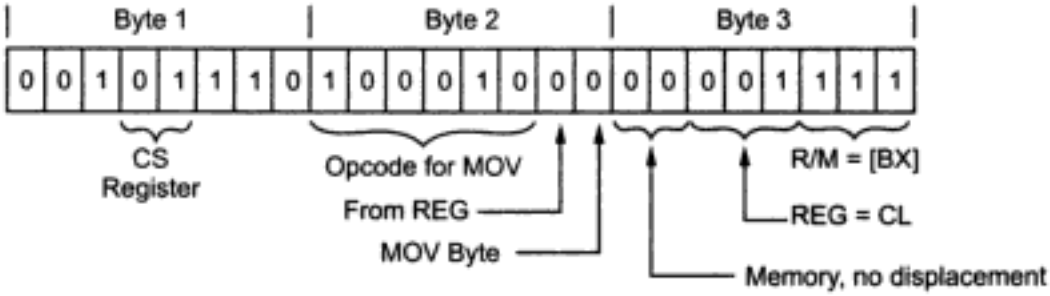


Fig. 3.35 Instruction format for MOV CS : [BX], CL

Review Questions

1. *Explain various data addressing modes of 8086 with the help of examples.*
2. *Explain the difference between direct and indirect addressing mode.*
3. *Explain base-plus-index addressing mode.*
4. *Explain how base-plus-index addressing mode can be used to locate array data.*
5. *Explain register relating addressing.*
6. *Explain base relative-plus-index addressing.*
7. *Explain how base relative-plus-index addressing can be used to locate data from two dimensional array.*
8. *Explain the string addressing mode.*
9. *Explain various I/O addressing modes supported by 8086.*

10. Explain direct program memory addressing with the help of example.
11. What is short, near and far jumps ?
12. Explain the difference between intersegment and intrasegment jump instructions.
13. Explain relative program memory addressing.
14. Explain indirect program memory addressing.
15. What is stack ?
16. What is the function of stack pointer ?
17. What do you mean by top of stack ?
18. Explain the usefulness of the following instructions in 8086
    a. LOCK b. TEST c. XLAT d. LES
19. Write the difference between the following instructions
    a. MOV CX, 437AH and MOV CX, [437AH]
    b. MOV BL, 437AH and MOV BL, DS:BYTE PTR [437AH]
20. Code the given following instructions for microprocessor 8086 ?
    a. MOV CX, AL b. MOV DS, 437AH
    c. MOV CL, [BX] d. MOV 43H[SI], DH
    e. MOV CS:[BX], DL
21. With the help of an example describe the action performed by microprocessor 8086 for each of the following instructions :
    a. AAM b. CMPSB c. IMUL d. ROL
22. Explain the use of the following prefixes
    a. REP b. REPE
23. Describe the response of 8086 to the following five primitive string operations.
    MOVS, CMPS, SCAS, LODS and STOS
24. Discuss all types of jump instructions used in 8086 microprocessor.
25. Write the operations performed by the 8086 microprocessor CALL instruction.
26. Explain in detail the difference between near CALL and far CALL.
27. For the following instruction compute the address of memory operand for 8086 :
    a. MOV AX, [BX] b. MOV AL, [BP + SI]
    Assume :
    CS = 0100H DS = 0200H SS = 0400H ES = 0030H
    BP = 0010H DX = 0020H SI = 0030H SP = 0030H
    Clearly show computations.
28. Describe the difference between a jump and a call instruction ? What does the processor do in executing it ? You may use 8085, 8086 instructions to explain.
29. Explain what operation is performed by the following instructions :
    a. SHL BYTE PTR [0400 H], CL
    b. MOV [BX] [DI] + 4, AX
    c. XLAT d. XTHL e. PCHL

30. *Explain the use of PUSH and POP instructions in 8086.*
31. *Explain the function of the following instructions of 8086 :*
    XLAT, CWD and CMPSB.
32. *What is the function of assembler directives ?*
33. *Explain the following assembler directives*
    a. DB b. EXTRN c. .MODEL SMALL d. PROC e. PUBLIC
34. *Explain variables, suffix and operators used in assembly language programming.*
35. *What do you mean by machine language program ?*
36. *What do you mean by assembly language program ?*
37. *Give the difference between machine language and assembly language.*
38. *Explain the assembly language programming tips.*
39. *What do you mean by optimum solution ?*
40. *Explain the steps that assembler follows to convert .ASM file to .OBJ file.*
41. *Explain the function of linker.*
42. *What is debugger ? Explain its advantages.*
43. *Explain various debugger commands.*
44. *What is time delay ? Write an assembly language program to generate a delay of 500 ms.*
45. *Explain the two techniques to convert binary to ASCII.*
46. *Explain the process of converting ASCII to binary.*
47. *Explain the process of displaying hexadecimal data.*
48. *Explain how look up tables can be used to convert BCD to 7-segment code.*
49. *What is macro ? When it should be used ? What are its advantages ?*
50. *Explain the structure of macro with the help of example.*
51. *Give the comparison between procedure and macro.*
52. *How are parameters passed to a macro ?*

❑❑❑

4

Assembly Language Programs

In this chapter, we see the programs involving logical, branch and call instructions, sorting, evaluation of arithmetic expressions and string manipulation. Most of the programs use DOS function calls. The details of DOS function calls are given in chapter 9.

Program 1 : Read keyboard input and display it on monitor

```
TITLE Read Keyboard Input and Display it on Monitor
.model small
.code
start:   mov ax,@data                ; [loads the address of data
         mov ds,ax                   ; segment in DS]
back:    mov ah,01
         int 21h
         cmp al,'0'
         jz Last
         jmp back
Last:    mov ah,4ch                  ; [ Exit
         int 21h                     ; to DOS ]
end start
end
```

Program 2 : Addition of two 32-bit numbers

```
; This program adds two numbers
TITLE Addition of two 32-bit numbers
.model small
.data
no1      dd   8111FFFFh
no2      dd    92224444h
result   dd    ?
carry    db    0
.code
start:   mov ax, @data          ; [loads the address of data
         mov ds, ax             ;  segment in DS]
         mov ax,word ptr no1    ; Get the LS word of first
                                ; number in AX add ax,word
                                ; ptr no2 Add the LS word of
                                ; second number to it
```

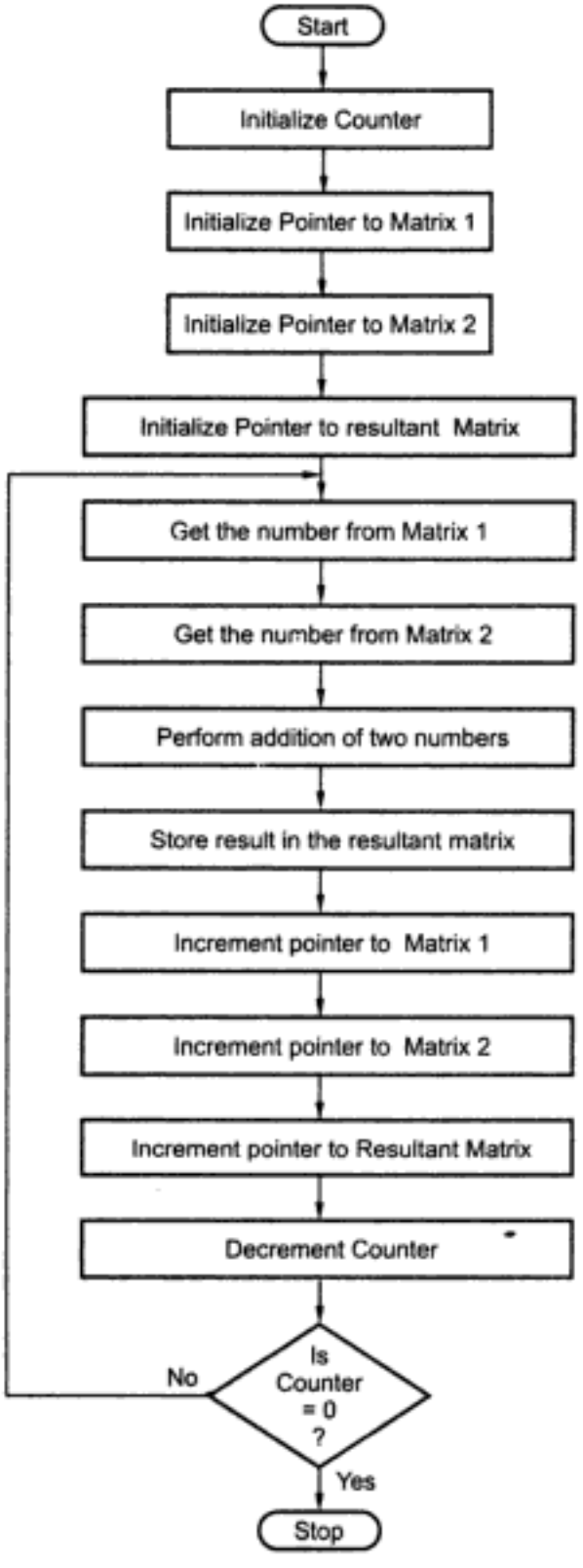
```
        mov word ptr result,ax; Save LS word of result
        mov bx, offset[no1]
        mov ax,word ptr [bx+2]; Get the MS word of first
                              ; number in AX
            mov bx, offset[no2]
            adc ax,word ptr [bx+2]     ; Add the MS word of second
                                       ; number to it with carry
            mov bx, offset result
            mov [bx+2],ax              ; Save MS word of result
            adc carry,0                ; save any carry after
                                       ; MS word addition
            mov ah,4ch                 ; [ Exit
            int 21h                    ; to DOS ]
            end start
            end
```

Program 3 : Addition of 3 × 3 matrix

```
; This program adds 3 × 3 matrix. The matrices are stored in
; form of lists (row wise).
TITLE Addition of 3 × 3 Matrix
.model small
.data
m1      db 10h,20h,30h,40h,50h,60h,70h,80h,90h
m2      db 10h,20h,30h,40h,50h,60h,70h,80h,90h
result dw 9 dup(0)
.code
start:      mov ax,@data           ; [loads the address of data
            mov ds,ax              ; segment in DS]
            mov cx,9               ; Initialise the counter
            mov di, offset m1      ; Initialise the pointer to
                                   ; matrix1
            mov bx, offset m2      ; Initialise the pointer to
                                   ; matrix2
            mov si, offset result ; Initialise the pointer to
                                   ; resultant matrix
back:       mov ah,00              ; Make MSB of result zero
            mov al,[di]            ; Get the number from matrix1
            add al,[bx]            ; Get the number from matrix2
                                   ; and add it in corresponding
                                   ; number of matrix1
            adc ah,00              ; Save the carry of addition
                                   ; in MSB
            mov [si],ax            ; Store the result in
                                   ; corresponding position of
                                   ; resultant matrix
            inc di                 ; increment pointer to matrix1
            inc bx                 ; increment pointer to matrix2
            inc si                 ; [ increment pointer
            inc si                 ; to resultant matrix ]
            loop back              ; Repeat the process for all
                                   ; matrix elements
```

```
                mov ah,4ch              ; [ Exit
                int 21h                 ;   to DOS ]
end start
end
```

Flowchart



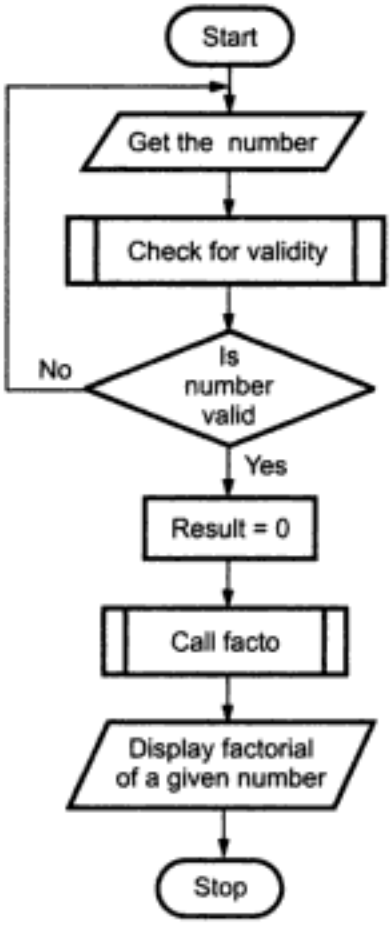
Program 4 : Program to read a password and validate user

```
.MODEL SMALL
.DATA
.STACK 100
PASS DB 'MBS1234'
MES1 DB 10,13,'ENTER 7 CHARACTER PASSWORD $'
MES2 DB 10,13,'PASSWORD IS CORRECT $'
MES3 DB 10,13,'INVALID PASSWORD$'
.CODE
START:   MOV AX,@DATA          ;[ Initialise
         MOV DS,AX             ; data segment ]
         MOV AH,09H
         LEA DX,MES1
         INT 21H               ; Display message
         MOV CL,00             ; Clear count
         MOV DH,00H            ; Clear number of match
         XOR DI,DI             ; Intialise pointer
         .WHILE CL != 7        ; Check if count = 7 if not
                               ; Continue
         MOV AH,07H
         INT 21H               ; Read character
         PUSH AX               ; Save character
         MOV AH,02H            ; [ Display
         MOV DL, '*'           ; '*' instead of
         INT 21H               ; character ]
         POP AX                ; Restore character
         LEA BX,PASS           ; [ Set pointer
         MOV AH,[BX+DI]        ; to password ]
         .IF AL==AH            ; Compare read character with
                               ; password
         ADD DH,01             ; Increament match count if match
                               ; occurs
         .ENDIF
         INC DI                ; Increment pointer
         INC CL                ; Increment counter
         .ENDW
         .IF DH == 7           ; [ if match count = 7
         MOV AH,09H            ; display message
         LEA DX,MES2           ; password is correct ]
         INT 21H
         .ELSE                 ; [ if match count <> 7
         MOV AH,09H            ; display message
         LEA DX,MES3           ; password is wrong ]
         INT 21H
         .ENDIF
         MOV AH,4CH            ; [ Exit to
         INT 21H               ; DOS ]
END START
END
```

Program 5 : Program to calculate factorial of a number

(Softcopy of this program, P18.asm is available at www.vtubooks.com)

Flowchart :




```
.MODEL SMALL
.STACK 100
.DATA
        MS1  DB 10,13,'ENTER THE NO.:$'
        MS2  DB 10,13,'THE FACTORIAL IS : $'
        NUM      DW 0
        ANS  DW 0
.CODE
START:      MOV     DX,@data      ; [ Initialise
            MOV     DS,DX         ; data segment ]
ERROR:      LEA     DX,MS1
            MOV     AH,09H        ; Display message MS1
            INT     21H
            MOV     AH,01H        ; Input number with echo
            INT     21H
            CMP     AL,30H        ; If zero display 1
            JE      DISPLY2
            CMP     AL,30H        ; If < 30 then input
```

```
            JB      ERROR           ; Next no
            CMP     AL,39H          ; If >39 then input
            JA      ERROR           ; Next no
            SUB     AL,30H          ; Convert to HEX
            MOV     AH,00H
            SUB     SP,0004H        ; Space in stack for
            PUSH    AX              ; Factorial
            CALL    FACTO
            ADD     SP,0002         ; After execution
            POP     AX              ; Of facto space for
            POP     DX              ; Result
            MOV     BX,0010         ; Convert HEX to BCD
            MOV     CX,0006         ; Max input no is 9
BACK:       DIV     BX              ; To get remainder
            OR      DX,0030H        ; Convert to ASCII
            PUSH    DX
            XOR     DX,DX           ; Clear DX
            LOOP    BACK
            LEA     DX,MS2          ; Output MS2
            MOV     AH,09
            INT     21H
            MOV     CX,0006
DISPLY1:    POP     DX
            MOV     AH,02H          ; Output factorial
            INT     21H
            LOOP    DISPLY1
            JMP     LAST
DISPLY2:    MOV     AH,09
            LEA     DX,MS2          ; Display factorial of
            INT     21H             ; Zero = 1
            MOV     AH,02H
            MOV     DL,31H
            INT     21H
LAST:       MOV     AH,4CH          ; [ Terminate and
            INT     21H             ;   Exit to DOS ]
            FACTO   PROC
            PUSHF
            PUSH    AX
            PUSH    DX
            PUSH    BP
            MOV     BP,SP           ; Point BP at TOS
            MOV     AX,[BP + 10]    ; Copy no from stack to
            CMP     AX,0001H        ; AX & if no not = 1 then
                                    ; GO_ON
            JNE     GO_ON           ; To compute factorial
            MOV     WORD PTR[BP+12],0001H
                                    ; Else load FFACT
            MOV     WORD PTR [BP+14],0000H
                                    ; 0 and 1 in stack
            JMP     EXIT
```

```
GO_ON:      SUB     SP,0004H          ; Space for preliminary
            DEC     AX                ; Factorial
            PUSH    AX
            CALL    FACTO
            MOV     BP,SP
            MOV     AX,[BP+2]         ; Last (N - 1)! from
                                      ; stack to AX
            MUL     WORD PTR [BP+16]  ; Multiply by previous N
            MOV     [BP+18],AX        ; Copy new facto to stack
            MOV     [BP+20],DX
            ADD     SP,0006H          ; Point SP at pushed REGR
EXIT:       POP     BP
            POP     DX
            POP     AX
            POPF
            RET
FACTO       ENDP
            END     START
```

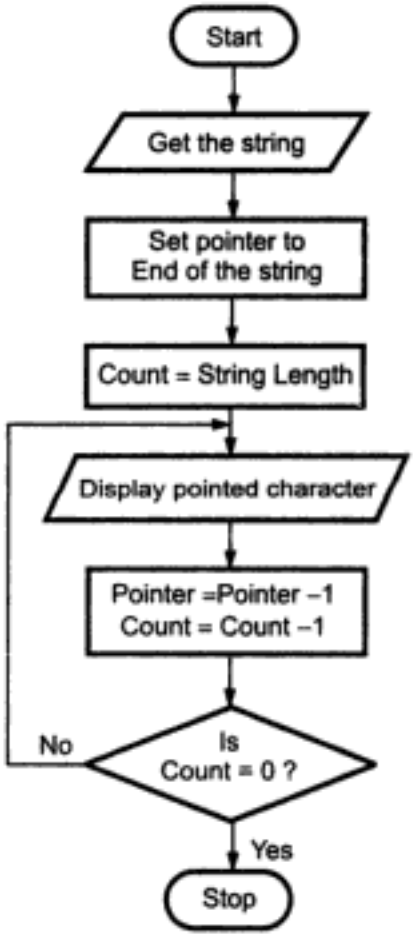
Program 6 : Reverse the words in string

(Softcopy of this program, P19.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.STACK 100
.DATA
TITLE    REVERSE THE WORDS IN STRING
         M1       DB 10,13, 'ENTER THE STRING:$'
         M2       DB 10,13, 'THE REVERSE STRING :$'
         BUFF     DB 80
                  DB 0
                  DB 80 DUP(0)
    COUNTER1      DW 0
    COUNTER2      DW 0
.CODE
START:            MOV      AX,@data      ; [ Initialise
                  MOV      DS,AX         ;   data segment ]
                  MOV      AH,09H        ; Display message M1.
                  MOV      DX,OFFSET M1
                  INT      21H
                  MOV      AH,0AH
                  LEA      DX,BUFF       ; I/P the string.
                  INT      21H
                  MOV      AH,09H
                  MOV      DX,OFFSET M2  ; Display message M2
                  INT      21H
                  LEA      BX,BUFF
                  INC      BX
                  MOV      CH,00H        ; [ Take character
                  MOV      CL,BUFF + 1   ;   count in
                  MOV      DI,CX         ;   DI ]
```

```
BACK:   MOV   DL,[BX+DI]      ; Point to the end
                              ; character and read it
        MOV   AH,02H
        INT   21H             ; Display the character
        DEC   DI              ; Decrement count
        JNZ   BACK            ; Repeat until count is0
EXIT:   MOV   AH,4CH          ; [ Terminate
        INT   21H             ;   Exit to DOS ]
        END   START
```

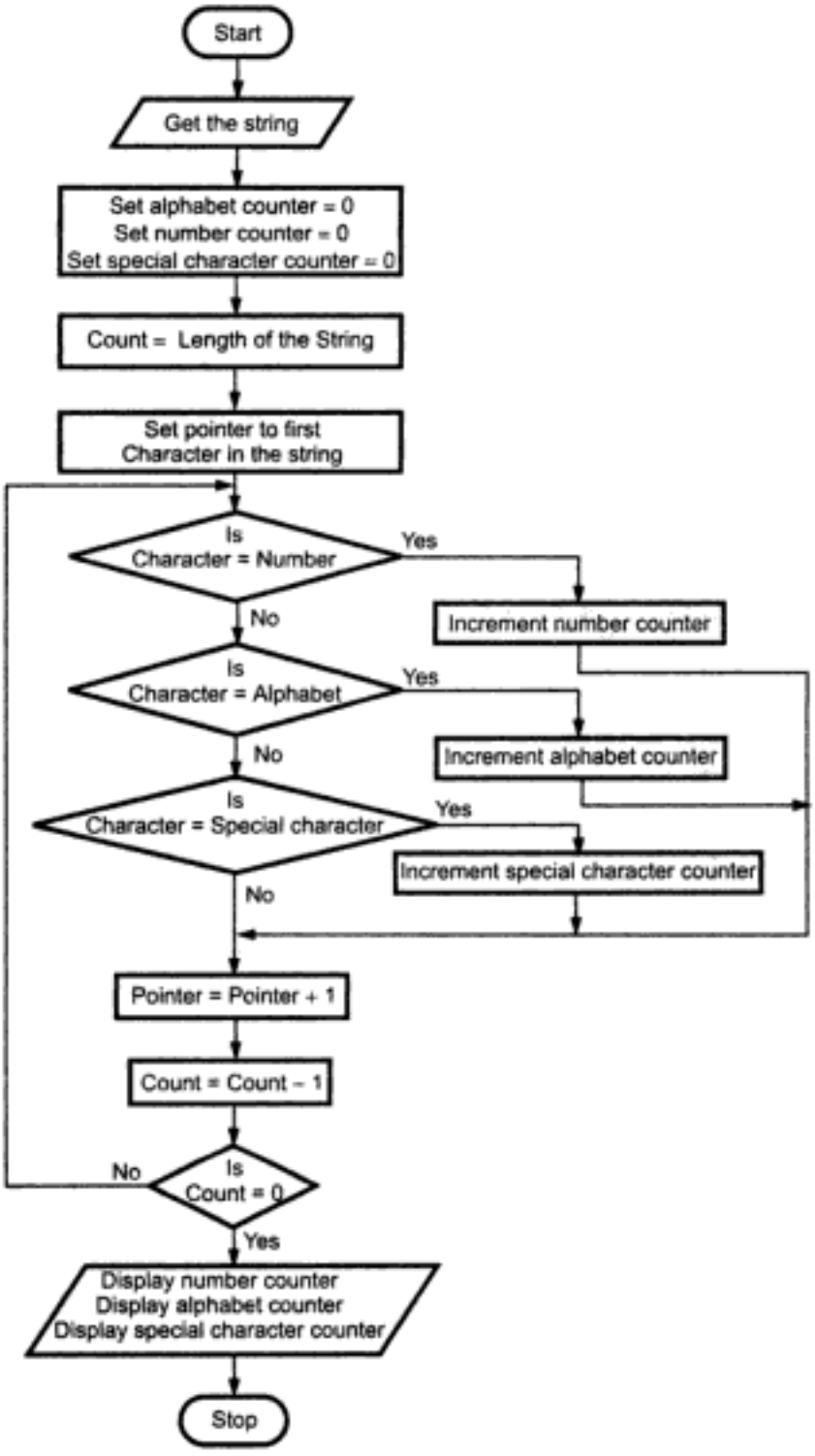
Flowchart :



Program 7 : Search numbers, alphabets, special characters

(Softcopy of this program, P20.asm is available at www.vtubooks.com)

Flowchart :



```
.MODEL SMALL
.STACK 100
TITLE    TOTAL
; (THIS PROGRAM GIVES THE TOTAL NUMBERS, ALPHABETS, SPECIAL
; CHARACTERS IN THE GIVEN STRING)
.DATA
         BUF      DB 80                    ;(MAX LENGTH OF ARRAY)
                  DB 00                    ;(ACTUAL LENGTH OF ARRAY)
                  DB 80 DUP (0)            ;(STARTING OF ARRAY)
     STR1         DB 10,13,'ENTER THE STRING:$'
     STR2         DB 10,13,'TOTAL NO:$'
     STR3         DB 10,13,'TOTAL ALPHABETS:$'
     STR4         DB 10,13,'TOTAL SPECIAL CHAR:$'
          NUM     DB 0
          SPC     DB 0
          ALPHA   DB 0

.CODE
START:   MOV      AX,@data          ; [ Initialise
         MOV      DS,AX             ;   data segment ]
         MOV      AH,09H
         MOV      DX,OFFSET STR1    ; Address of STR1
         INT      21H               ; Display message STR1
         MOV      AH,0AH
         MOV      DX,OFFSET BUF     ; Get address of the buffer
         INT      21H               ; Input the string
         MOV      BX,OFFSET BUF     ; Get address of the buffer
         INC      BX                ; Increment address of buffer
         MOV      DL,[BX]           ; Get the length of string
         INC      BX                ; Get the starting of array
NEXT:    MOV      AL,[BX]           ; Read the character
         CMP      AL,30H            ; Check for special character
         JB       INCSPC            ; If yes goto INCSPC
         CMP      AL,3AH            ; Check for number
         JB       INCNUM            ; If number goto INCNUM
         CMP      AL,41H            ; Check for special character
         JB       INCSPC            ; If yes goto INCSPC
         CMP      AL,5BH            ; Check for alphabet
         JB       INALP             ; If yes goto INALP
         CMP      AL,61H            ; Check for special character
         JB       INCSPC            ; If yes goto INCSPC
         CMP      AL,7BH            ; Check for alphabet
         JB       INALP             ; If yes goto INALP
INCSPC:  MOV      AL,SPC
         ADD      AL,01H            ; [ INCR special character
                                    ;   counter and
         DAA                        ;    adjust it to decimal ]
         MOV      SPC,AL
         INC      BX                ; Increment pointer to point
                                    ; the next character
```

```
        DEC     DL              ; Decrement counter
        JNZ     NEXT
        JMP     DISPLY          ; Otherwise goto DISPLY
INCNUM: MOV     AL,NUM
        ADD     AL,01H          ; [ Increment number counter
        DAA                     ;  and adjust it to decimal ]
        MOV     NUM,AL
        INC     BX              ; Increment pointer to point
                                ; the next character
        DEC     DL              ; Decrement counter
        JNZ     NEXT            ; If count not = 0, repeat
        JMP     DISPLY          ; Otherwise goto DISPLY
INALP:  MOV     AL,ALPHA
        ADD     AL,01H          ; [ Increment alphabet counter
        DAA                     ;  and adjust it to decimal ]
        MOV     ALPHA,AL
        INC     BX              ; Increment pointer to point
                                ; the next character
        DEC     DL              ; Decrement counter
        JNZ     NEXT            ; If count not = 0, repeat
        JMP     DISPLY          ; Otherwise goto DISPLY
DISPLY: MOV     DX,OFFSET STR2  ; Get the address of STR2
        MOV     AH,09H
        INT     21H             ; Display message STR2
        MOV     AL,NUM          ; Read the number count
        AND     AL,0F0H         ; Get MS digit in AL rotate AL
        MOV     CL,04H          ; Four times
        ROR     AL,CL
        ADD     Al,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H          ; Display the MS digit
        INT     21H
        MOV     AL,NUM          ; Read the number count
        AND     AL,0FH          ; Get LS digit in AL
        ADD     AL,30H          ; Convert to ASCIII
        MOV     DL,AL
        INT     21H             ; Display the LS digit
        MOV     DX,OFFSET STR3  ; Get address of STR3
        MOV     AH,09H
        INT     21H             ; Display message STR3
        MOV     AL,ALPHA        ; Read the alphabet count
        AND     AL,0F0H         ; Get MS digit in AL rotate AL
        MOV     CL,04H          ; Four times
        ROR     AL,CL
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the MS digit
        MOV     AL,ALPHA        ; Read the alphabet count
        AND     AL,0FH          ; Get LS digit in AL
```

```
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the LS digit
        MOV     DX,OFFSET STR4  ; Get the address of STR4
        MOV     AH,09H
        INT     21H             ; Display message STR4
        MOV     AL,SPC          ; Read the special character
                                ; count
        AND     AL,0F0H         ; Get MS digit in AL rotate AL
        MOV     CL,04           ; Four times
        ROR     AL,CL
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the MS digit
        MOV     AL,SPC          ; Read the special character count
        AND     AL,0FH          ; Get LS digit in AL
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the LS digit
        MOV     AH,4CH          ; [ Terminate and
        INT     21H             ;     Exit to DOS ]
        END     START
```

Program 8 : Program to find whether string is palindrome or not

(Softcopy of this program, P21.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.DATA
        M1      DB 10, 13, 'Enter the string : $'
        M2      DB 10, 13, 'String is palindrome $'
        M3      DB 10, 13, 'String is not palindrome $'
        BUFF    DB 80
                DB 0
                DB 80 DUP (0)
.CODE
START:  MOV AX,@data        ; [ Initialise
        MOV DS,AX           ;   data segment ]
        MOV AH,09H
        MOV DX,OFFSET M1
        INT 21H             ; Display message M1
        MOV AH,0AH          ; Input the string
        LEA DX,BUFF
        INT 21H
        LEA BX,BUFF+2       ; Get starting address of string
        MOV CH,00H
        MOV CL,BUFF+1
        MOV DI,CX
```

```
        DEC DI
        SAR CL,1
        MOV SI,00H
BACK:   MOV AL,[BX + DI]        ; Get the right most character
        MOV AH,[BX + SI]        ; Get the left most character
        CMP AL,AH               ; Check for palindrome
        JNZ LAST                ; If not exit
        DEC DI                  ; Decrement end pointer
        INC SI                  ; Increment starting pointer
        DEC CL                  ; Decrement counter
        JNZ BACK                ; If count not = 0, repeat
        MOV AH,09H              ; Display message 2
        MOV DX,OFFSET M2
        INT 21H
        JMP TER
LAST:   MOV AH,09H
        MOV DX,OFFSET M3        ; Display message 3
        INT 21H
TER:    MOV AH,4CH              ; [ Terminate and
        INT 21H                 ;   Exit to DOS ]
        END START
```

Program 9 : Program to display string in lowercase

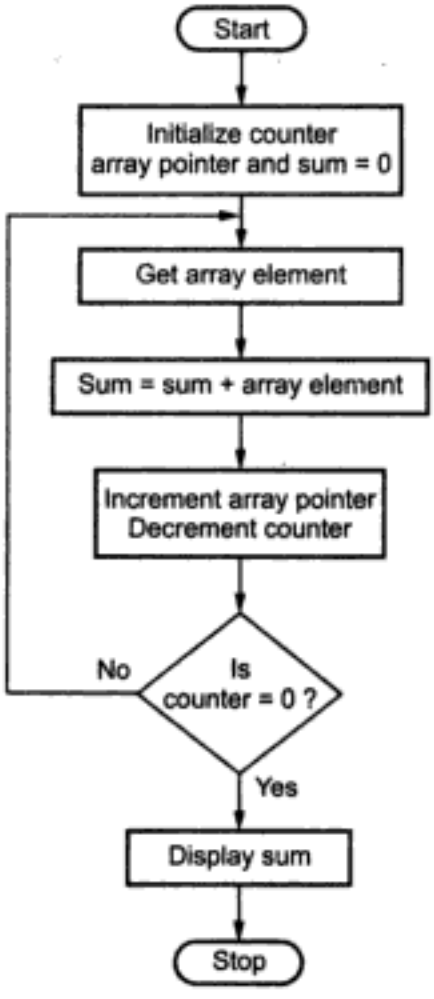
(Softcopy of this program, P22.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.DATA
        M1      DB  10, 13, 'ENTER THE STRING : $'
        M2      DB  10, 13, 'THE LOWERCASE STRING : $'
        BUFF    DB  80
                DB  0
                DB  80 DUP (0)
.CODE
START : MOV AX,@data            ; [ Initialise
        MOV DS,AX               ;   data segment ]
        MOV AH,09H              ; Display message1
        MOV DX,OFFSET M1
        INT 21H
        MOV AH,09H
        MOV DX,OFFSET M2        ; Display message M2
        INT 21H
        MOV AH,0AH              ; Input the string
        LEA DX,BUFF
        INT 21H
        MOV CH,00H
```

```
        MOV CL,BUFF+1          ; Take character count in CX
        LEA BX,BUFF+2
        MOV DI,00H
BACK :  MOV DL,[BX+DI]         ; point to the first character
        ADD DL,20H             ; convert to lowercase
        MOV AH,02H
        INT 21H                ; Display the character
        INC DI
        DEC CX                 ; Decrement character counter
        JNZ BACK               ; If not = 0, repeat
        MOV AH,4CH             ; [ Terminate and
        INT 21H                ;   Exit to DOS ]
        END START
```

Program 10: Write an 8086 assembly language program (ALP) to add array of N number stored in the memory.

Flowchart :



Algorithm :

1. Initialize counter = N
2. Initialize array pointer.
3. Sum = 0
4. Get the array element pointed by array pointer.
5. Add array element in the sum.
6. Increment array pointer decrement counter.
7. Repeat steps 4, 5 and 6 until counter equal to zero.
8. Display sum.
9. Stop.

Sum of array having HEX numbers

```
          PAGE     52,80
          TITLE    8086 ALP to find sum of numbers in the array.
.MODEL  SMALL
.DATA
     ARRAY     DB  10H,20H,30H,40H,50H,60H,70H,80H,90H,00H
     SUM       DW  0
     MES       DB  10,13,  'Sum of array elements is : $'
.CODE
START:    MOV  AX,@data      ; [ Initialise
          MOV  DS,AX         ;    data segment ]
          MOV  CL,10         ; Initialise counter
          XOR  DI,DI         ; Initialise pointer
          LEA  BX,ARRAY      ; Initialise array base pointer
BAC:      MOV  AL,[BX+DI]    ; Get the number
          MOV  AH,00H        ; Make higher byte 00h
          ADD  SUM,AX        ; SUM = SUM + number
          INC  DI            ; Increment pointer
          DEC  CL            ; Decrement counter
          JNZ  BAC           ; if not 0 go to back
          MOV  AX,SUM        ; Get sum in AX
          CALL D_HEX         ; Display sum of array
          MOV  AH,  4CH
          INT  21H
```

```
D_HEX  PROC  NEAR

          PUSH  DX                ; Save registers
          PUSH  CX
          PUSH  AX

          MOV  CL,  04H           ; Load rotate count
          MOV  CH,  04H           ; Load digit count
BACK:     ROL  AX,  CL            ; rotate digits
          PUSH  AX                ; save contents of AX
          AND  AL,  0FH           ; [Convert
          CMP  AL,9               ;  number
          JBE  ADD30              ; to
          ADD  AL,  37H           ; its
          JMP  DISP               ; ASCII
ADD30:
          ADD  AL,30H             ; equivalent]
DISP:     MOV  AH,02H
          MOV  DL,AL              ; [Display the
          INT  21H                ;  number]
          POP  AX                 ; restore contents of AX
          DEC  CH                 ; decrement digit count
          JNZ  BACK               ; if not zero repeat

          POP  AX                 ; Restore registers
          POP  CX
          POP  DX

          RET
          ENDP
          END
```

Sum of array having decimal numbers

```
     PAGE     52,80
     TITLE    8086 ALP to find sum of numbers in the array.
.MODEL  SMALL
.DATA
```

```
     ARRAY    DB  12,24,26,63,25,86,20,33,10,35
     SUM      DW  0
     MES      DB  10,13,  'Sum of array elements is : $'
.CODE
START:   MOV  AX,@data      ; [ Initialise
         MOV  DS,AX         ;    data segment ]
         MOV  CL,10         ; Initialise counter
         XOR  DI,DI         ; Initialise pointer
         LEA  BX,ARRAY      ; Initialise array base pointer
BAC:     MOV  AL,[BX+DI]    ; Get the number
         MOV  AH,00H        ; Make higher byte 00h
         ADD  SUM,AX        ; SUM = SUM + number
         INC ·DI            ; Increment pointer
         DEC  CL            ; Decrement counter
         JNZ  BAC           ; if not 0 go to back
         MOV  AX,  SUM      ; Get the result
         CALL  ATB4D        ; Display sum of array
         MOV  AH,  4CH
         INT  21H

ATB4D  PROC  NEAR

         PUSH     DX            ; Save registers
         PUSH     CX
         PUSH     BX
         PUSH     AX

         MOV      CX,  0        ; Clear digit counter
         MOV      BX,  10       ; Load 10 decimal in BX
BACK :   MOV      DX,  0        ; Clear DX
         DIV      BX            ; divide DX : AX by 10
         PUSH     DX            ; Save remainder
         INC      CX            ; Counter remainder
         OR       AX,  AX       ; test if quotient equal to zero
         JNZ      BACK          ; if not zero divide again
         MOV      AH,  02H      ; load function number
```

```
DISP:   POP  DX          ; get remainder
        ADD  DL, 30H     ; Convert to ASCII
        INT  21H         ; display digit
        LOOP DISP

        POP  AX          ; Restore registers
        POP  BX
        POP  CX
        POP  DX

        RET
        ENDP
        END
```

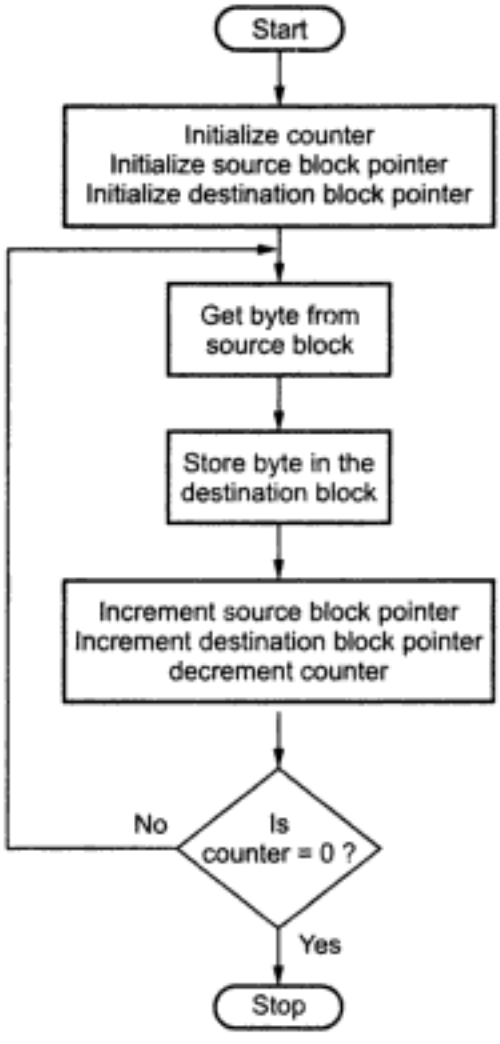
Program 11 : Write 8086 ALP to perform non-overlapped block transfer.

In non overlapped block transfer, source block and destination blocks are different. Here, we can transfer byte-by-byte or word-by-word data from one block to another block.

Algorithm :

1. Initialize counter.
2. Initialize source block pointer.
3. Initialize destination block pointer.
4. Get the byte from source block.
5. Store the byte in the destination block.
6. Increment source, destination pointers and decrement counter.
7. Repeat steps 4, 5 and 6 unit counter equal to zero.
8. Stop.

Flowchart :



Non overlapped block transfer

```
        PAGE      52,80
        TITLE     Non overlapped block transfer.
.MODEL  SMALL
.STACK  100
.DATA
        ARRAY           DB  12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H
        NEW_ARR         DB  10 DUP (?)
  .CODE
START:   MOV  AX,@data      ; [ Initialise
         MOV  DS,AX         ; data segment and
         MOV  ES,AX         ; extra segment ]
         MOV  CX,10         ; Initialise counter
         LEA  SI,ARRAY      ; Initialise source_pointer
```

```
        LEA     DI,NEW_ARR    ; Initialise destination_pointer
        CLD                   ; Clear direction flag to
                              ; autoincrement SI and DI
        MOV     AL,[SI]       ; [Get the number
        MOV     [DI],AL       ; and save number in new array ]
        REP     MOVSB         ; Decrement CX and MOVSB until Cx
                                 will be 0

        LEA     DI,NEW_ARR    ; Initialise destination_pointer
        MOV     CX,10         ; Initialize counter
BACK1:  MOV     AH,[DI]       ; Get number
        CALL    D_HEX2        ; Display number
        CALL    SPACE         ; Display space
        INC     DI            ; Increment destination_pointer
        LOOP    BACK1         ; if counter not zero, repeat
        MOV     AH,4CH        ; Return to DOS
        INT     21H


D_HEX2  PROC  NEAR
        PUSH    CX
        MOV     CL,  04H      ; Load rotate count
        MOV     CH,  02H      ; Load digit count
BAC:    ROL     AX,  CL       ; rotate digits
        PUSH    AX            ; save contents of AX
        AND     AL,  0FH      ; [Convert
        CMP     AL,9          ;   number
        JBE     Add30         ; to
        ADD     AL,  37H      ; its
        JMP     DISP          ; ASCII
Add30:  ADD     AL,30H        ; equivalent]
DISP:   MOV     AH,02H
        MOV     DL,AL         ; [Display the
        INT     21H           ;  number]
        POP     AX            ; restore contents of AX
        DEC     CH            ; decrement digit count
        JNZ     BAC           ; if not zero repeat
        POP     CX
        RET
        ENDP
```

```
SPACE  PROC  NEAR
          PUSH   AX              ;  Save  registers
          PUSH   DX
          MOV    AH,02           ;  Display  space
          MOV    DL,'  '
          INT    21H
          POP    DX              ;  restore  registers
          POP    AX
          RET                    ;  return  to  main  program
          ENDP
          END
```

Overlapped block transfer

We call two blocks are overlapped when some portion of source and destination blocks are common. As shown in the Fig. 4.1, source and destination blocks can be overlapped in two ways. In first case Fig. 4.1 (a) we can begin transfer from starting location of source block to the starting location of destination block, i.e. [20000H] ← [20005H]

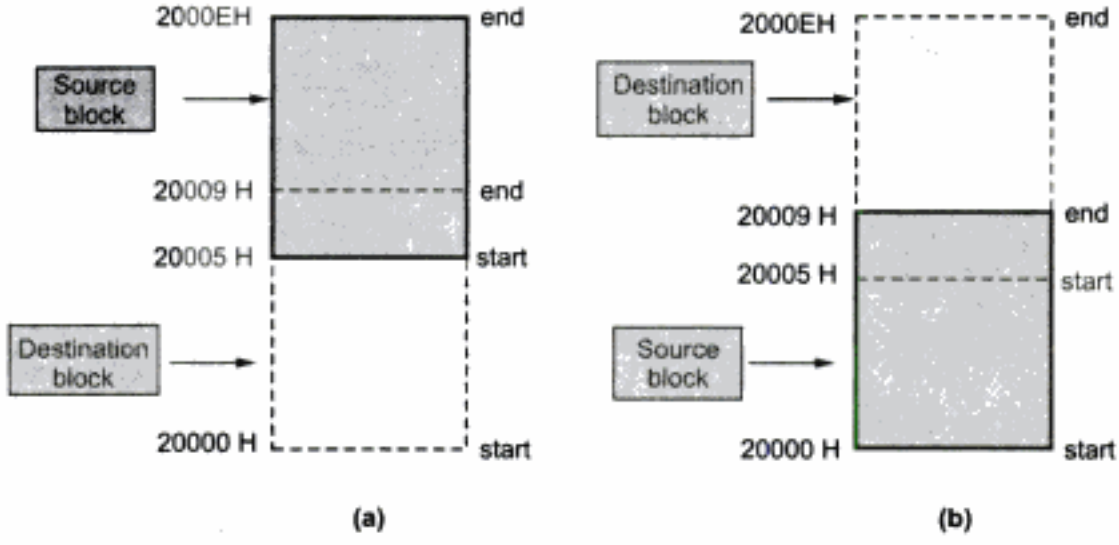


Fig. 4.1

We can then increment source and destination block pointers and carry on byte transfer until the pointers reach the end of two blocks, i.e. upto [20009H] ← [2000EH].

In second case Fig. 4.1 (b) we cannot use the same block transfer procedure, because there will be over writing of data within the source block, i.e. at first byte transfer contents of 20000H will be over written in the location 20005H and data at 20005H in the source block get lost. To avoid over writing in such cases we have to transfer data from source block to destination block from the end of the block, i.e. we have to begin with the

transfer [2000EH] ← [20009H], decrement the source and destination pointers and carry on the byte transfer until the pointer reach the start of the blocks, i.e. upto [20005H] ← [20000H]

```
          PAGE      52,80
          TITLE     Overlapped block transfer.
.MODEL  SMALL
.STACK  100
.DATA
    ARRAY   DB,12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H,?,?,?,?,?
.CODE
START:    MOV  AX,@data      ; [ Initialise
          MOV  DS,AX         ; data segment and
          MOV  ES,AX         ; extra segment ]
          MOV  CX,10         ; Initialise counter
          LEA  SI,ARRAY+9    ; Initialise source_pointer
          LEA  DI,ARRAY+14   ; Initialise destination_pointer
          STD                ; SET direction flag to
                               autodecrement SI and DI
          MOV  AL,[SI]       ; Get the number
          MOV  [DI],AL       ; and save number in new array ]
          REP  MOVSB         ; Decrement CX and MOVSB until
                             ; Cx will be 0

          LEA  DI,ARRAY+5    ; Initialise destination_pointer
          MOV  CX,10         ; Initialize counter
 BACK1:   MOV  AH,[DI]       ; Get number
          CALL D_HEX2        ; Display number
          CALL SPACE         ; Display space
          INC  DI            ; Increment distination_pointer
          LOOP BACK1         ; If counter not zero repeat
          MOV  AH,4CH        ; Return to DOS
          INT  21H
```

```
D_HEX2  PROC  NEAR
        PUSH    CX
        MOV     CL,  04H        ; Load rotate count
        MOV     CH,  02H        ; Load digit count
BAC:    ROL     AX,  CL         ; rotate digits
        PUSH    AX              ; save contents of AX
        AND     AL,  0FH        ; [Convert
        CMP     AL,9            ;  number
        JBE     Add30           ; to
        ADD     AL,  37H        ; its
        JMP     DISP            ; ASCII
Add30:  ADD     AL,30H          ; equivalent]
DISP:    MOV    AH,02H
        MOV     DL,AL           ; [Display the
        INT     21H             ;  number]
        POP     AX              ; restore contents of AX
        DEC     CH              ; decrement digit count
        JNZ     BAC             ; if not zero repeat
        POP     CX
        RET
        ENDP
SPACE  PROC  NEAR
        PUSH    AX              ; save registers
        PUSH    DX
        MOV     AH,02           ; display space
        MOV     DL,' '
        INT     21H
        POP     DX              ; restore registers
        POP     AX
        RET                     ; return to main program
        ENDP
        END
```

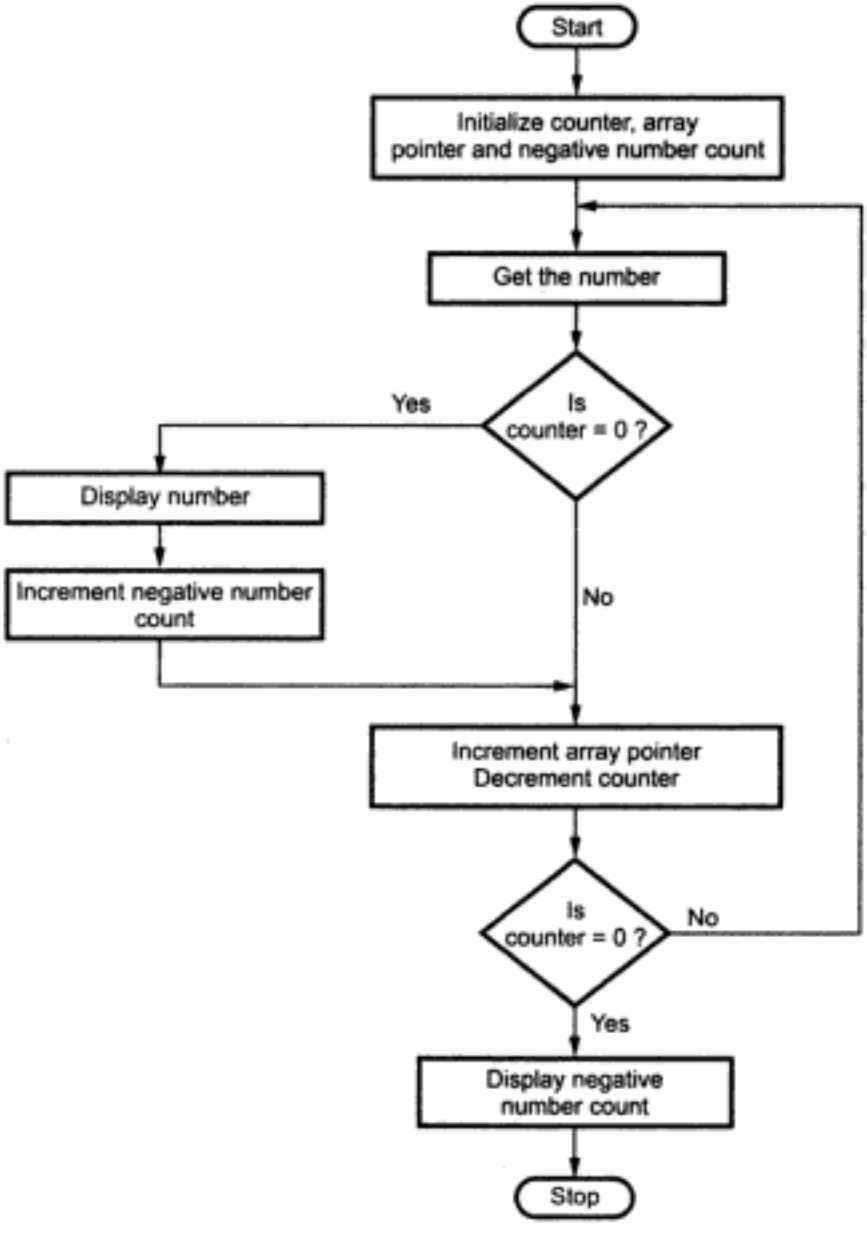
Program 12: Write 8086 ALP to find and count negative numbers from the array of signed numbers stored in memory.

In sign number representation, number is called negative when its most significant bit (MSB) is 1. This bit can be checked by masking all other bits with the help of logical AND instruction.

Algorithm :

1. Initialize counter.
2. Initialize array pointer.
3. Initialize negative number count.
4. Get the number.
5. Check sign of number by checking its MSB. If negative increment negative number count and display the number.
6. Decrement counter and increment array pointer.
7. Repeat steps 4, 5 and 6 until counter equal to zero.
8. Display negative number count.
9. Stop.

Flowchart



```
     PAGE    52,80
     TITLE   Find and count the negative numbers in the array.
     .MODEL  SMALL
     .STACK  100
     .DATA
         ARRAY    DB  92H,23H,96H,0A3H,25H,86H,2FH,33H,10H,35H
         MES      DB  10,13,  'Negative numbers are : $'
         MES1     DB  10,13,  'Total Negative number count is : $'
     .CODE
START:   MOV      AX,@data      ; [ Initialise
         MOV      DS,AX         ; data segment ]
         MOV      CX,10         ; Initialise counter
         MOV      BH,0          ; Initialise negative number count
                                   equal to 0
         LEA      BP,ARRAY      ; Initialise array base_pointer
         LEA      DX,  MES
         MOV      AH,  09H
         INT      21H
BACK:    MOV      AL,DS:[BP]    ; Get the number
         MOV      AH,AL         ; Save number in AH
         AND      AL,80H        ; Mask all bits except MSB
         JZ       NEXT          ; If MSB = 0 go to next
         CALL     D_HEX2        ; Otherwise display number
         CALL     SPACE
         INC      BH             ; Increment negative number count
NEXT :   INC      BP            ; Increment array base_pointer
         LOOP     BACK          ; Decrement counter
                                ; if not 0 go to back
         LEA      DX,  MES1
         MOV      AH,  09H
         INT      21H
         MOV      AH,02H
         ADD      BH,30H
         MOV      DL,BH
         INT      21H
         MOV      AH,4CH        ; [ Exit
         INT      21H           ;    to DOS ]
```

```
D_HEX2  PROC  NEAR
        MOV     CL,  04H        ; Load rotate count
        MOV     CH,  02H        ; Load digit count
BAC:    ROL     AX,  CL         ; rotate digits
        PUSH    AX              ; save contents of AX
        AND     AL,  0FH        ; [Convert
        CMP     AL,9            ;  number
        JBE     Add30           ; to
        ADD     AL,  37H        ; its
        JMP     DISP            ; ASCII
Add30:
        ADD     AL,  30H        ; equivalent]
DISP:   MOV     AH,  02H
        MOV     DL,  AL         ; [Display the
        INT     21H             ;  number]
        POP     AX              ; restore contents of AX
        DEC     CH              ; decrement digit count
        JNZ     BAC             ; if not zero repeat
        ENDP
SPACE  PROC  NEAR
    PUSH    AX                  ; save AX
    MOV     AH,02H              ; [ Call DOS routine
    MOV     DL,  ' '            ;    to leave space ]
    INT     21H                 ; restore AX
    POP     AX                  ; return to main program
    RET
    ENDP

    END
```

Program 13 : Convert BCD to HEX and HEX to BCD

Write 8086 ALP to convert 4-digit HEX number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choices from user for :

a. HEX to BCD

b. BCD to HEX

c. EXIT

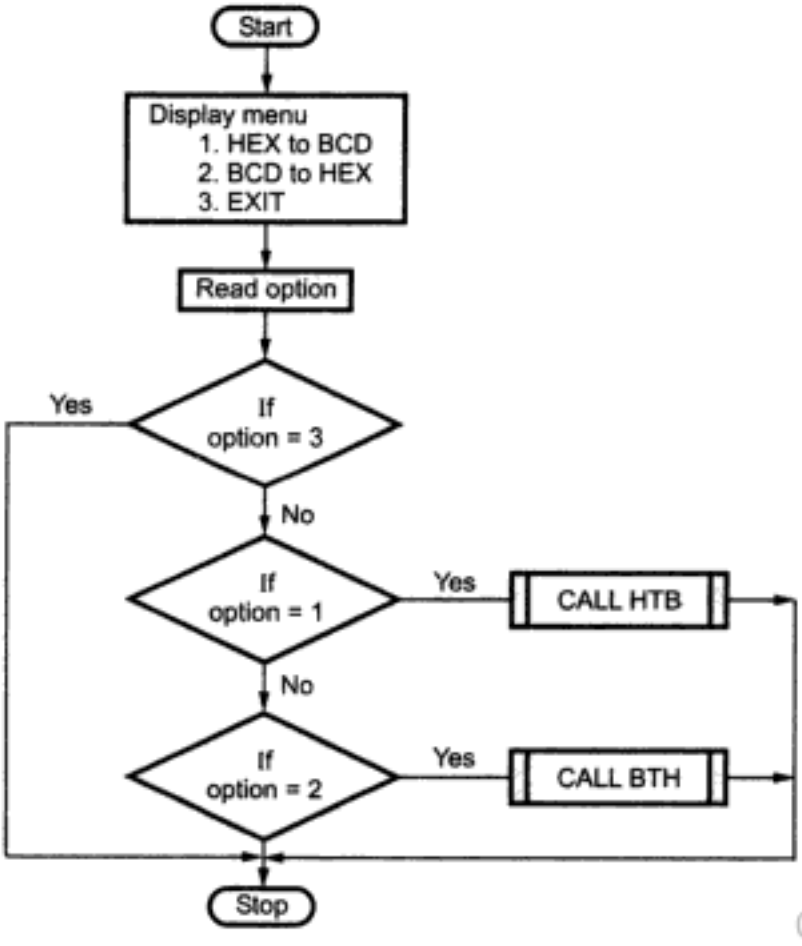
Display proper strings to prompt the user while accepting the input and displaying the result.

In this program we use the standard routines explained in the chapter 3 to convert data from one form to other. However, to select the conversion we display menu on the screen and display proper messages on the screen to guide user. Therefore, in this program separate macro named PROMPT is written for display the message. After accepting the option from the user, the option is checked and proper routine is called to perform desired operation.

Algorithm :

1. Display menu
   a. HEX To BCD
   b. BCD To HEX
   c. EXIT
   ENTER THE CHOICE :
2. Read the option
   It option is 3-exit
   1 - Do HEX to BCD conversion
   2 - Do BCD to HEX conversion
3. Stop

Flowchart :



```
PROMPT  MACRO  MESSAGE              ; Define macro with MESSAGE as a
                                    ; parameter
          PUSH    AX                ; Save AX register
          MOV     AH,  09H          ; display message
          LEA     DX,  MESSAGE
          INT     21H
          POP     AX                ; restore register
          ENDM


.MODEL  SMALL       ; select small model
.STACK  100


.DATA               ; start data segment
    MES1      DB  10,  13,  '1. HEX TO BCD $'
    MES2      DB  10,  13,  '2. BCD TO HEX $'
    MES3      DB  10,  13,  '3. EXIT $'
    MES4      DB  10,  13,  'ENTER THE CHOICE : $'
    MES5      DB  10,  13,  'ENTER CORRECT CHOICE : $'
    MES6      DB  10,  13,'$'
    MES7      DB  10,  13,  'ENTER THE FOUR DIGIT HEX NUMBER : $'
    MES8      DB  10,  13,  'EQUIVALENT BCD NUMBER IS : $'
    MES9      DB  10,  13,  'ENTER THE BCD NUMBER : $'
    MES10     DB  10,  13,  'EQUIVALENT HEX NUMBER IS : $'


          NUMBER  DW  ?       ; define NUMBER
.CODE                         ; start code segment
START:    MOV  AX,  @DATA     ; [Initialize
          MOV  DS,  AX        ;  data segment]

          PROMPT  MES1        ; Display MES1
          PROMPT  MES2        ; Display MES2
          PROMPT  MES3        ; Display MES3
          PROMPT  MES4        ; Display MES4
```

```
AGAIN:  MOV     AH,01    ; [ READ
        INT     21H      ;    OPTION ]


        PROMPT  MES6     ; Display MES6

        CMP     AL,'3'   ; [ If choice is 3
        JZ      LAST     ;    exit ]

        CMP     AL,'1'   ; [ If choice is 1
        JNZ     NEXT1
        CALL    HTB      ;    Do HEX to BCD conversion
        JMP     LAST     ;    exit ]

NEXT1:  CMP     AL,'2'   ; [ If choice is 2
        JNZ     NEXT2
        CALL    BTH      ;    Do BCD to HEX conversion
        JMP     LAST     ;    exit ]

NEXT2:  PROMPT  MES5     ; Display MES5
        JMP     AGAIN

LAST:   MOV     AH,4CH   ; Return to DOS
        INT     21H

HTB  PROC  NEAR

        PROMPT  MES7
        CALL  R_HEX
        PROMPT  MES8
        CALL  D_BCD
        RET
        ENDP
```

```
BTH  PROC  NEAR
          PROMPT   MES9
          MOV      CX, 10          ; load 10 decimal in CX
          MOV      BX, 0           ; clear result
BACK2:    MOV      AH,01H          ;[Read key
          INT      21H             ; with echo]
          CMP      AL,'0'
          JB       SKIP            ; jump if below '0'
          CMP      AL,'9'
          JA       SKIP            ; jump if above '9'
          SUB      AL, 30H         ; convert to BCD
          PUSH     AX              ; save digit
          MOV      AX, BX          ; multiply previous result by 10
          MUL      CX
          MOV      BX, AX          ; get the result in BX
          POP      AX              ; retrieve digit
          MOV      AH, 00H
          ADD      BX, AX          ; Add digit value to result
          JMP      BACK2           ; Repeat
SKIP :    MOV      AX,BX           ; save the result in AX
          PROMPT   MES10
          CALL     D_HEX
          RET
          ENDP

R_HEX  PROC  NEAR
     MOV  CL,  04        ;   load shift count
     MOV  SI,  04        ;   load iteration count
     MOV  BX,  0         ;   clear result
BAC: MOV  AH,  01        ;  [Read a key
     INT  21H            ;   with echo]

     CALL  CONV          ;   convert to binary

     SHL  BX,  CL        ;  [pack four
     ADD  BL,  AL        ;   binary digits
     DEC  SI             ;   as 16-bit
     JNZ  BAC            ;   number]
     MOV  NUMBER,  BX    ;  save result at NUMBER
     ENDP
```

```
; The procedure to convert contents of AL into
; hexadecimal equivalent
    CONV  PROC  NEAR

          CMP  AL,'9'
          JBE  SUBTRA30      ; if number is between 0 through 9
          CMP  AL,'a'
          JB  SUBTRA37       ; if letter is uppercase
          SUB  AL,  57H      ; subtract 57H if letter is lowercase
                             JMP  LAST1
SUBTRA30:SUB  AL,  30H       ; convert number
                             JMP  LAST1
SUBTRA37:SUB  AL,  37H       ; convert uppercase letter
LAST1:    RET
CONV      ENDP

D_BCD  PROC  NEAR

     MOV  AX,NUMBER
     MOV  CX,  0           ; Clear digit counter
     MOV  BX,  10          ; Load 10 decimal in BX
BACK:MOV  DX,  0           ; Clear DX
     DIV  BX               ; divide DX : AX by 10
     PUSH DX               ; Save remainder
     INC  CX               ; Counter remainder
     OR   AX,  AX          ; test if quotient equal to zero
     JNZ  BACK             ; if not zero divide again
     MOV  AH,  02H         ; load function number
DISP:POP  DX               ; get remainder
     ADD  DL,  30H         ; Convert to ASCII
     INT  21H              ; display digit
     LOOP DISP
     RET
     ENDP

D_HEX  PROC  NEAR
          MOV  CL,  04H      ; Load rotate count
          MOV  CH,  04H      ; Load digit count
BAC1:     ROL  AX,  CL       ; rotate digits
          PUSH AX            ; save contents of AX
```

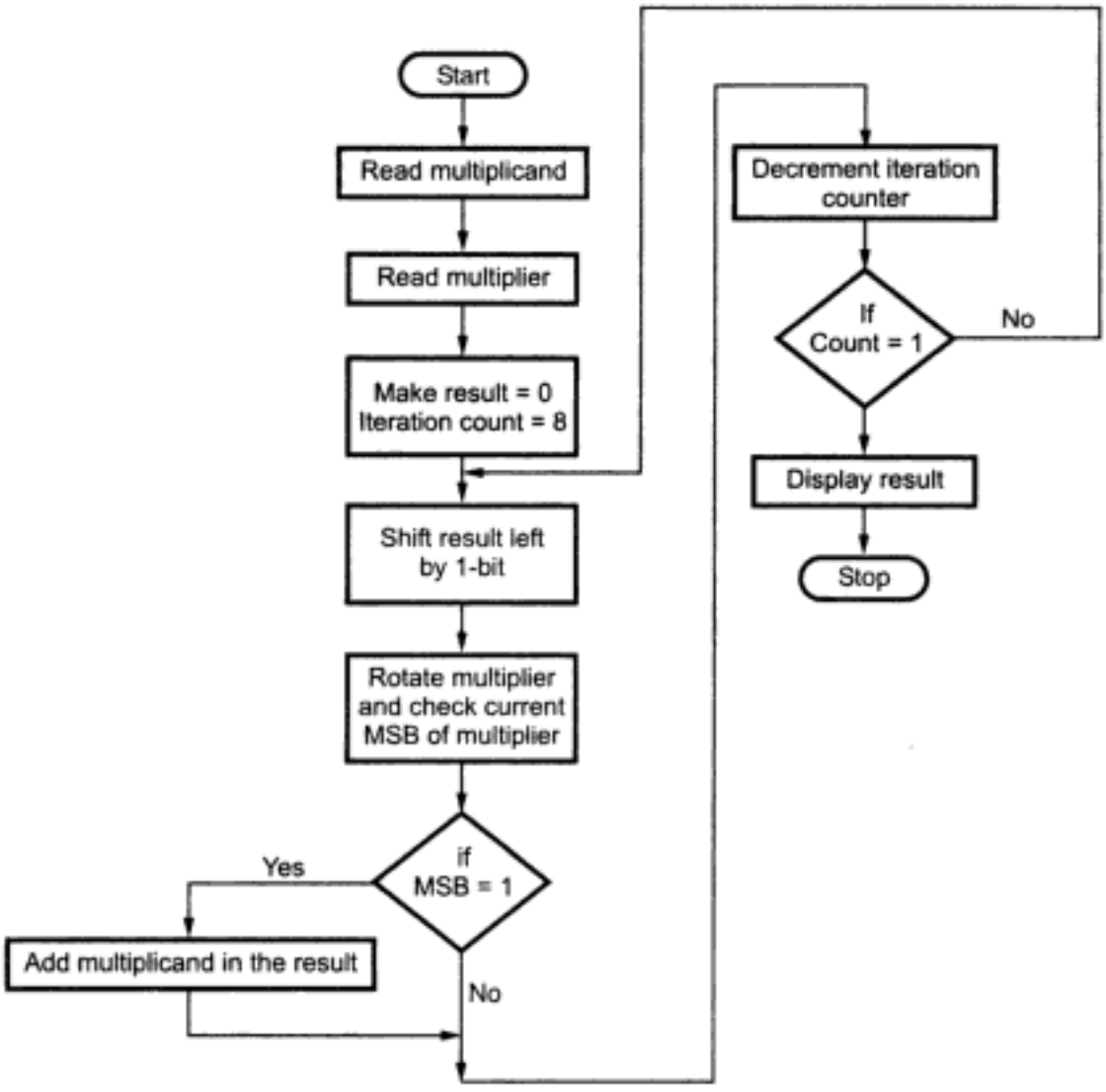
```
        AND  AL,  0FH       ; [Convert
        CMP  AL,9           ;  number
        JBE  Add30          ; to
        ADD  AL,  37H       ; its
        JMP  DISP1          ; ASCII
Add30:
        ADD  AL,30H         ; equivalent]
DISP1:  MOV  AH,02H
        MOV  DL,AL          ; [Display the
        INT  21H            ;  number]
        POP  AX             ; restore contents of AX
        DEC  CH             ; decrement digit count
        JNZ  BAC1           ; if not zero repeat
        RET
        ENDP

        END
```

Program 14 : Multiplication of two 8-bit numbers

Algorithm :

1. Read 2-digit hex number as a multiplicand.
2. Read 2-digit hex number as a multiplier.
3. Initialize iteration count = 8 since multiplier is 8-bit.
4. Make result = 0.
5. Shift result left by 1-bit.
6. Rotate multiplier 1-bit to check current MSB if bit is 1, Add multiplicand in the result.
7. Decrement iteration count and repeat steps 5 and 6 fill iteration count is zero.
8. Display result.
9. Stop.

Flowchart :



Multiplication of HEX numbers

```
PROMPT  MACRO  MESSAGE          ;  Define  macro  with  MESSAGE  as  a
                                ;  parameter
          PUSH    AX            ;  save  register
          MOV     AH,  09H      ;  display  message
          LEA     DX,  MESSAGE
          INT     21H
          POP     AX            ;  restore  register
          ENDM


.MODEL  SMALL      ;  select  small  model
.STACK  100
```

```
.DATA                          ; start data segment
     MUL_ER  DB  ?             ; define NUMBER
     MUL_AND DB  ?             ; define NUMBER
     MES1  DB  10,13,  'Enter 2-digit hex number as a multiplicand:$'
     MES2  DB  10,13,  'Enter 2-digit hex number as a multiplier :$'
     MES3  DB  10,13,  'The result of multiplication is :$'
.CODE                          ; start code segment
START:   MOV  AX,  @DATA       ; [Initialize
         MOV  DS,  AX          ;   data segment]

         PROMPT   MES1
         CALL     READ_HEX2
         MOV      MUL_AND,BL

         PROMPT   MES2
         CALL     READ_HEX2
         MOV      MUL_ER,BL

         MOV      DH,00
         MOV      DL,MUL_AND
         MOV      CX,0008
         MOV      AX,0000
REP1:    SHL      AX,1
         ROL      BL,1
         JNC      SKIP
         ADD      AX,DX
SKIP:    LOOP     REP1
         PROMPT   MES3
         CALL     D_HEX
         MOV      AH,02H
         MOV      DL,'H'
         INT      21H
         MOV      AH,  4CH      ; [Exit to
         INT      21H           ;   DOS]

READ_HEX2  PROC  NEAR

         MOV      CL,  04       ;   load shift count
         MOV      SI,  02       ;   load iteration count
         MOV      BL,  0        ;   clear result
BACK  :  MOV      AH,  01       ; [Read a key
```

```
        INT     21H             ;   with echo]
        CALL    CONV            ;   convert to binary
        SHL     BL, CL          ; [pack two
        ADD     BL, AL          ;   binary digits
        DEC     SI              ;   as 8-bit
        JNZ     BACK            ;   number]
        RET
        ENDP

    ; The procedure to convert contents of AL into
    ; hexadecimal equivalent
CONV PROC NEAR
        CMP AL,'9'
        JBE SUBTRA30        ; if number is between 0 through 9
                              CMP AL,'a'
        JB  SUBTRA37        ; if letter is uppercase
        SUB AL, 57H         ; subtract 57H if letter is lowercase
        JMP LAST
SUBTRA30:SUB AL, 30H        ; convert number
        JMP LAST
SUBTRA37:SUB AL, 37H        ; convert uppercase letter
LAST:   RET
CONV    ENDP

D_HEX PROC NEAR
        MOV CL, 04H         ; Load rotate count
        MOV CH, 04H         ; Load digit count
BAC1:   ROL AX, CL          ; rotate digits
        PUSH AX             ; save contents of AX
        AND AL, 0FH         ; [Convert
        CMP AL,9            ;  number
        JBE Add30           ; to
        ADD AL, 37H         ; its
        JMP DISP1           ; ASCII
Add30:
        ADD AL,30H          ; equivalent]
DISP1:  MOV AH,02H
        MOV DL,AL           ; [Display the
        INT 21H             ;  number]
        POP AX              ; restore contents of AX
```

```
        DEC  CH             ; decrement digit count
        JNZ  BAC1           ; if not zero repeat
        RET
        ENDP
        END
```

Multiplication of BCD numbers

```
PROMPT  MACRO  MESSAGE  ;Define macro with MESSAGE as a parameter
          PUSH    AX
          MOV     AH,  09H
          LEA     DX,  MESSAGE
          INT     21H
          POP     AX
          ENDM


.MODEL  SMALL                ; select small model
.STACK  100



.DATA                        ; start data segment
        MUL_ER  DB  ?        ; define NUMBER
        MUL_AND DB  ?        ; define NUMBER
        MES1  DB  10,13,  'Enter 2-digit BCD number (<256) as a
                           multiplicand : $'
        MES2  DB  10,13,  'Enter 2-digit BCD number (<256) as a
                           multiplier : $'
        MES3  DB  10,13,  'The result of multiplication is : $'
.CODE                        ; start code segment
START:  MOV  AX,  @DATA      ; [Initialize
        MOV  DS,  AX         ;  data segment]

        PROMPT   MES1
        CALL     BTH
        MOV      MUL_AND,AL

        PROMPT   MES2
        CALL     BTH
        MOV      MUL_ER,AL
        MOV      BL,AL
```

```
        MOV     DH,00
        MOV     DL,MUL_AND
        MOV     CX,0008
        MOV     AX,0000
REP1:   SHL     AX,1
        ROL     BL,1
        JNC     SKIP1
        ADD     AX,DX
SKIP1:  LOOP    REP1
        PROMPT  MES3
        CALL    D_BCD

        MOV     AH, 4CH         ; [Exit to
        INT     21H             ;  DOS]

BTH  PROC  NEAR
        MOV     CX, 10          ; load 10 decimal in CX
        MOV     BX, 0           ; clear result
BACK2:  MOV     AH,01H          ;[Read key
        INT     21H             ; with echo]
        CMP     AL,'0'
        JB      SKIP            ; jump if below '0'
        CMP     AL,'9'
        JA      SKIP            ; jump if above '9'
        SUB     AL, 30H         ; convert to BCD
        PUSH    AX              ; save digit
        MOV     AX, BX          ; multiply previous result by 10
        MUL     CX
        MOV     BX, AX          ; get the result in BX
        POP     AX              ; retrieve digit
        MOV     AH, 00H
        ADD     BX, AX          ; Add digit value to result
        JMP     BACK2           ; Repeat
SKIP :  MOV     AX,BX           ; save the result in AX
        RET
        ENDP
```

```
D_BCD  PROC  NEAR

          MOV  CX,  0          ;  Clear  digit  counter
          MOV  BX,  10         ;  Load  10  decimal  in  BX
BACK1:    MOV  DX,  0          ;  Clear  DX
          DIV  BX              ;  divide  DX  :  AX  by  10
          PUSH DX              ;  Save  remainder
          INC  CX              ;  Counter  remainder
          OR   AX,  AX         ;  test  if  quotient  equal  to  zero
          JNZ  BACK1           ;  if  not  zero  divide  again
          MOV  AH,  02H        ;  load  function  number
DISP:     POP  DX              ;  get  remainder
          ADD  DL,  30H        ;  Convert  to  ASCII
          INT  21H             ;  display  digit
          LOOP DISP
          RET
          ENDP

          END
```

Program 15 : Divide 4 digit BCD number by 2 digit BCD number.

```
PROMPT  MACRO  MESSAGE  ;Define  macro  with  MESSAGE  as  a  parameter
          PUSH      AX
          MOV       AH,  09H
          LEA       DX,  MESSAGE
          INT       21H
          POP       AX
          ENDM


.MODEL  SMALL                        ;  select  small  model
.STACK  100

.DATA                                ;  start  data  segment
     DIVISOR   DB   ?                ;  define  NUMBER
     DIVIDEND  DW   ?                ;  define  NUMBER
     MES1      DB   10,13,'Enter  4-digit  BCD  number  as  dividend:$'
     MES2      DB   0,13,  'Enter  2-digit  BCD  number  as  a  divisor:$'
     MES3      DB  10,13,  'The  Quotient  of  Division  is    :  $'
     MES4      DB  10,13,  'The  Remainder  of  Division  is  :  $'
```

```
.CODE                              ; start code segment
START:   MOV      AX, @DATA        ; [Initialize
         MOV      DS, AX           ;  data segment]

         PROMPT   MES1
         CALL     ATB

         PROMPT   MES2
         CALL     BTH
         MOV      DIVISOR,AL

         MOV      AX, DIVIDEND
         DIV      DIVISOR

         MOV      BX,AX
         PROMPT   MES3
         MOV      AH,00
CALL  D_BCD

         PROMPT   MES3

         MOV      AH,00
         MOV      AL,BH
CALL  D_BCD
         MOV      AH, 4CH          ; [Exit to
         INT      21H              ;  DOS]

BTH  PROC  NEAR
         MOV      CX, 10           ; load 10 decimal in CX
         MOV      BX, 0            ; clear result
BACK2:   MOV      AH,01H           ;[Read key
         INT      21H              ; with echo]
         CMP      AL,'0'
         JB       SKIP1            ; jump if below '0'
         CMP      AL,'9'
         JA       SKIP1            ; jump if above '9'
         SUB      AL, 30H          ; convert to BCD
         PUSH     AX               ; save digit
         MOV      AX, BX           ; multiply previous result by 10
```

```
        MUL  CX
        MOV  BX,  AX        ; get the result in BX
        POP  AX             ; retrieve digit
        MOV  AH,  00H
        ADD  BX,  AX        ; Add digit value to result
        JMP  BACK2          ; Repeat
SKIP1:  MOV  AX,BX          ; save the result in AX
        RET
        ENDP

D_BCD  PROC  NEAR
        PUSH BX
        MOV  CX,  0         ; Clear digit counter
        MOV  BX,  10        ; Load 10 decimal in BX
BACK1:  MOV  DX,  0         ; Clear DX
        DIV  BX             ; divide DX : AX by 10
        PUSH DX             ; Save remainder
        INC  CX             ; Counter remainder
        OR   AX,  AX        ; test if quotient equal to zero
        JNZ  BACK1          ; if not zero divide again
        MOV  AH,  02H       ; load function number
DISP:   POP  DX             ; get remainder
        ADD  DL,  30H       ; Convert to ASCII
        INT  21H            ; display digit
        LOOP DISP
        POP  BX
        RET
        ENDP

ATB  PROC  NEAR

        PUSH  CX            ; Save registers
        PUSH  BX
        PUSH  AX

        MOV  CX,  10        ; load 10 decimal in CX
        MOV  BX,  0         ; clear result
BACK:   MOV  AH,01H         ;[Read key
        INT  21H            ; with echo]
        CMP  AL,'0'
```

```
        JB   SKIP           ; jump if below '0'
        CMP  AL,'9'
        JA   SKIP           ; jump if above '9'
        SUB  AL, 30H        ; convert to BCD
        PUSH AX             ; save digit
        MOV  AX, BX         ; multiply previous result by 10
        MUL  CX
        MOV  BX, AX         ; get the result in BX
        POP  AX             ; retrieve digit
        MOV  AH, 00H
        ADD  BX, AX         ; Add digit value to result
        JMP  BACK           ; Repeat
SKIP:   MOV  DIVIDEND,BX    ; save the result in NUMBER

        POP  AX             ; Restore registers
        POP  BX
        POP  CX
        RET
        ENDP

        END
```

Program 16 : To perform conversion of temperature from °F to °C.

```
PROMPT MACRO MESSAGE ;Define macro with MESSAGE as a parameter
        PUSH  AX
        MOV  AH,  09H
        LEA  DX,  MESSAGE
        INT  21H
        POP  AX
        ENDM

.MODEL  SMALL           ; select small model
.STACK  100

.DATA                   ; start data segment
    NUMBER  DW  ?       ; define NUMBER
    MES1  DB  10,13, 'Enter Temperature in Degree FAHRENEIT : $'
    MES2  DB  10,13,  'The Temperature in Degree Celsius is  : $'
```

```
.CODE                                ; start code segment
START:  MOV     AX, @DATA            ; [Initialize
        MOV     DS, AX               ;  data segment]

        PROMPT  MES1

        CALL    ATB                  ; Get the Temperature in F

        MOV     AX,NUMBER
        SUB     AX,20H               ; Subtract 32
        MOV     NUMBER,AX


        MOV     BX,05
        MOV     CX,09
        MUL     BX                   ; Multiply by 5
        DIV     CX                   ; Divide by 9

        MOV     NUMBER, DX           ; Save remainder
        PROMPT  MES2
        CALL    D_BCD                ; Display result in decimal
        CMP     NUMBER,0             ; If remainder is zero exit
        JZ      LAST
        MOV     DL, '.'              ; Display decimal point
        MOV     AH,02H
        INT     21H

        MOV     AX,0064H             ; Multiply remainder by 100
        MUL     NUMBER               ; Divide result by 9
        DIV     CX
        CALL    D_BCD                ; display fractions

LAST:   MOV     AH, 4CH              ; [Exit to
        INT     21H                  ;  DOS]
```

```
D_BCD  PROC  NEAR
       PUSH CX
       MOV  CX,  0          ; Clear digit counter
       MOV  BX,  10         ; Load 10 decimal in BX
BACK1: MOV  DX,  0          ; Clear DX
       DIV  BX              ; divide DX : AX by 10
       PUSH DX              ; Save remainder
       INC  CX              ; Counter remainder
       OR   AX,  AX         ; test if quotient equal to zero
       JNZ  BACK1           ; if not zero divide again
       MOV  AH,  02H        ; load function number
DISP:  POP  DX              ; get remainder
       ADD  DL,  30H        ; Convert to ASCII
       INT  21H             ; display digit
       LOOP DISP
       POP  CX
       RET
       ENDP
```

```
ATB  PROC  NEAR

       PUSH CX              ; Save registers
       PUSH BX
       PUSH AX

       MOV  CX,  10         ; load 10 decimal in CX
       MOV  BX,  0          ; clear result
BACK:  MOV  AH,01H          ;[Read key
       INT  21H             ; with echo]
       CMP  AL,'0'
       JB   SKIP            ; jump if below '0'
       CMP  AL,'9'
       JA   SKIP            ; jump if above '9'
       SUB  AL,  30H        ; convert to BCD
       PUSH AX              ; save digit
       MOV  AX,  BX         ; multiply previous result by 10
       MUL  CX
       MOV  BX,  AX         ; get the result in BX
       POP  AX              ; retrieve digit
       MOV  AH,  00H
```

```
        ADD  BX,  AX        ; Add digit value to result
        JMP  BACK           ; Repeat
SKIP:   MOV  NUMBER,BX      ; save the result in NUMBER

        POP  AX             ; Restore registers
        POP  BX
        POP  CX
        RET
        ENDP
    END
```

Program 17 : String operations

Program Statement : Write 8086 ALP for the following operations on the string entered by the user.

a. Calculate length of the string.

b. Reverse the string.

c. Check whether the string is palindrome or not.

Make your program user friendly by providing MENU like :

a. Enter the string.

b. Calculate length of string.

c. Reverse string.

d. Check palindrome.

e. Exit.

Here we use PROMPT macro to display the message on the screen, accept choice from the user and call proper procedure to perform desired task. To enter a string we use function 0AH of INT21. This function accepts a string and stores it in the buffer along with its length. Let us see the algorithm and flow chart.

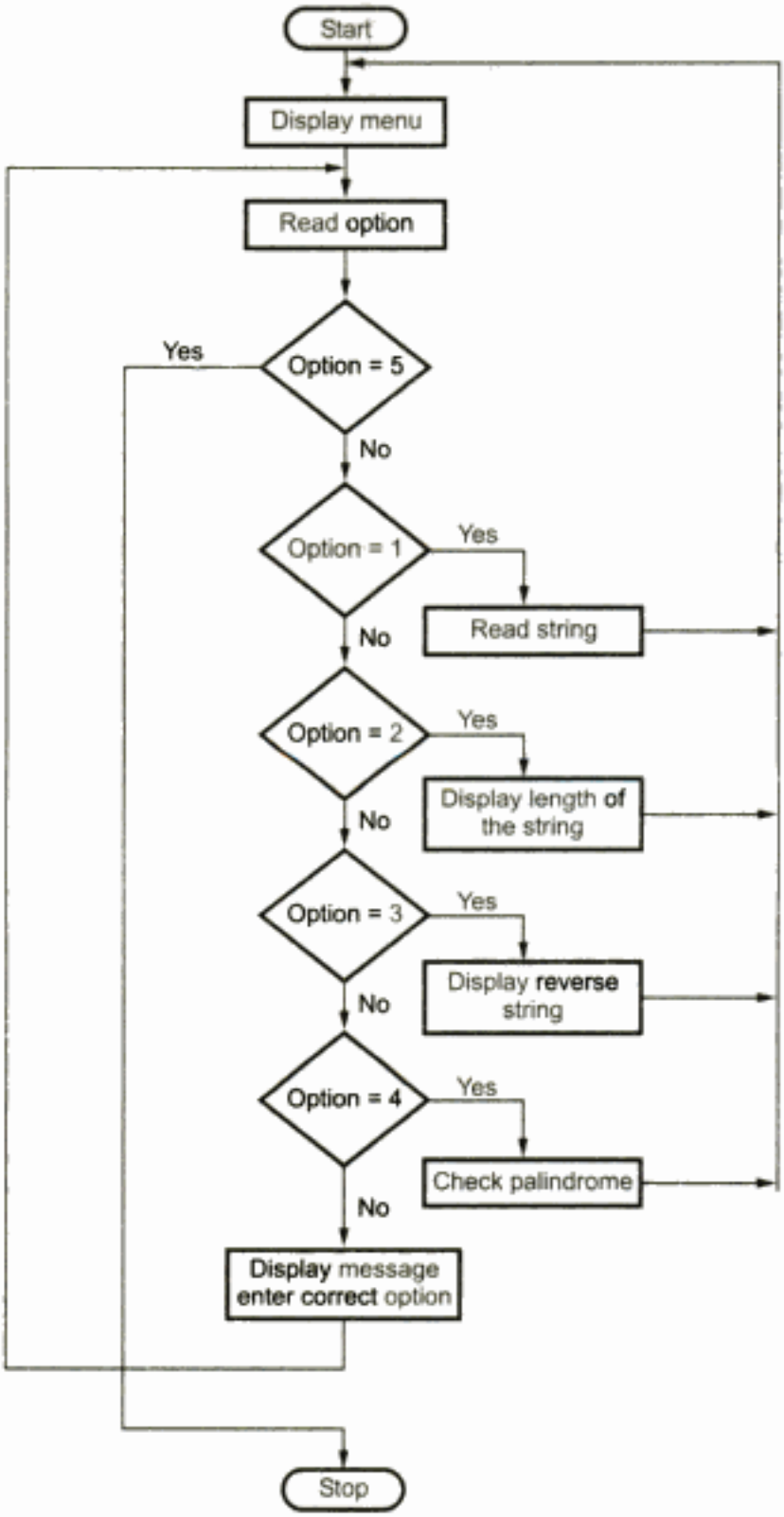
Algorithm :

1. Display Menu
   a. Enter the string.
   b. Calculate length of the string.
   c. Reverse the string.
   d. Check whether the string is palindrome or not.
   e. Exit.

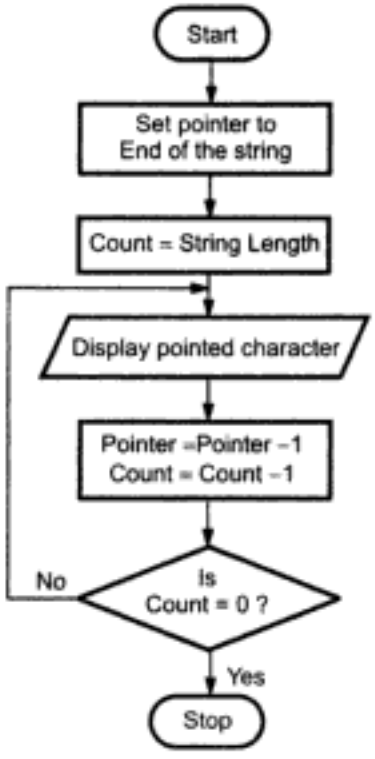
   Enter the option : -
2. Read the option

   If option is
   a. Read the string.
   b. Read the string length and display it.
   c. Initialize pointer at the end of the string and display the string from end to start.
   d. i) Initialize two pointers one at start and other at the end.
      ii) Compare two bytes; if not equal stop and display string is not palindrome.
      iii) Increment start pointer and decrement end pointer.
      iv) Repeat step ii) and iii) until two pointers overlap i.e. until start pointer reach the half the string.
   e. Exit to DOS.
3. Stop.

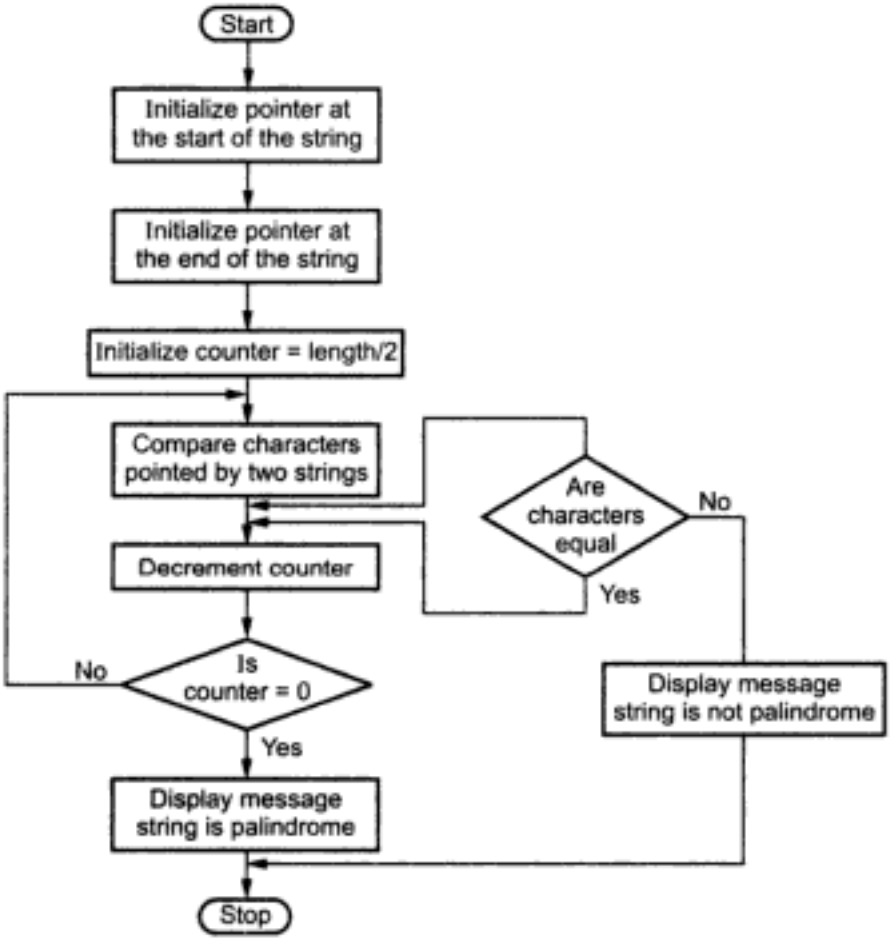
Flowchart :



Flowchart : String Reverse



Flowchart : String Palindrome



```
PROMPT MACRO MESSAGE ;Define macro with MESSAGE as a parameter
          PUSH    AX                ; save AX register
          MOV     AH, 09H           ; display message
          LEA     DX, MESSAGE
          INT     21H
          POP     AX                ; restore AX register
          ENDM

.MODEL SMALL                        ; select small model
.STACK 100


.DATA                               ; start data segment
    MES1      DB 10, 13, '1. ENTER THE STRING $'
    MES2      DB 10, 13, '2. CALCULATE THE LENGTH OF STRING $'
    MES3      DB 10, 13, '3. REVERSE THE STRING $'
    MES4      DB 10, 13, '4. PALINDROME $'
    MES5      DB 10, 13, '5. EXIT $'
    MES6      DB 10, 13, 'ENTER THE CHOICE : $'
    MES7      DB 10, 13, 'ENTER CORRECT CHOICE : $'
    MES8      DB 10, 13, '$'
    MES9      DB 10, 13, 'FAILED : STRING IS MISSING - PLEASE
                          ENTER THE STRING$'
    MES10     DB 10, 13, 'STRING LENGTH IN DECIMAL IS : $'
    MES11     DB 10, 13, 'STRING IS NOT PALINDROME $'
    MES12     DB 10, 13, 'STRING IS PALINDROME $'

    FLAG      DB 0
    MES13     DB 10, 13, 'ENTER THE STRING: $'
    MES14     DB 10, 13, 'THE STRING IS : $'

    BUFF      DB 80
              DB 0
              DB 80 DUP(0)
    COUNTER1  DW 0
    COUNTER2  DW 0
```

```
        NUMBER   DW ?      ; define NUMBER
.CODE                      ; start code segment
START:  MOV AX, @DATA      ; [Initialize
        MOV DS, AX         ;  data segment]

BEGIN:  PROMPT MES8        ; Display MES8
        PROMPT MES8        ; Display MES8
        PROMPT MES1        ; Display MES1
        PROMPT MES2        ; Display MES2
        PROMPT MES3        ; Display MES3
        PROMPT MES4        ; Display MES4
        PROMPT MES5        ; Display MES5
        PROMPT MES6        ; Display MES6

AGAIN:  MOV    AH,01       ; [ READ
        INT    21H         ;   OPTION ]


        PROMPT MES8        ; Display MES8

        CMP    AL,'5'      ; [ If choice is 5
        JZ     LAST        ;   exit ]

        CMP    AL,'1'      ; [ If choice is 1
        JNZ    NEXT1
        CALL   E_STR       ;  Enter the string
        PROMPT MES8
        PROMPT MES14
        CALL   D_STR       ;  Display the string
        JMP    BEGIN       ;   exit ]


NEXT1:  CMP    AL,'2'      ; [ If choice is 2
        JNZ    NEXT2
        CALL   L_STR       ;   Calculate the length of the string
        JMP    BEGIN       ;   exit ]

NEXT2:  CMP    AL,'3'      ; [ If choice is 3
        JNZ    NEXT3
        CALL   R_STR       ;   Reverse the string
```

```
        JMP     BEGIN   ;     exit ]

NEXT3:  CMP     AL,'4'  ; [ If choice is 4
        JNZ     NEXT4
        CALL    P_STR   ; Palindrome of the string
        JMP     BEGIN   ;     exit ]

NEXT4:  PROMPT  MES7    ; Display MES7
        JMP     AGAIN

LAST:   MOV     AH,4CH  ; Return to DOS
        INT     21H

E_STR PROC NEAR
        PROMPT  MES13   ; Display message MES13
        MOV     AH,0AH
        LEA     DX,BUFF ; I/P the string.
        INT     21H
        MOV     FLAG,1
        RET
        ENDP

L_STR PROC NEAR
        CMP     FLAG,0
        JNZ     SKIP
        PROMPT  MES9
        RET
SKIP:   MOV     AL,BUFF+1
        PROMPT  MES10
        CALL    D_HEX
        RET
        ENDP

R_STR PROC NEAR
        CMP     FLAG,0
        JNZ     SKIP1
        PROMPT  MES9
        RET
```

```
SKIP1:  CALL    DR_STR
        RET
        ENDP

P_STR  PROC  NEAR
        LEA     BX,BUFF+2       ; Get starting address of string
        MOV     CH,00H
        MOV     CL,BUFF+1
        MOV     DI,CX
        DEC     DI
        SAR     CL,1
        MOV     SI,00H
BACK4:  MOV     AL,[BX + DI]    ; Get the right most character
        MOV     AH,[BX + SI]    ; Get the left most character
        CMP     AL,AH           ; Check for palindrome
        JNZ     LAST2           ; If not exit
        DEC     DI              ; Decrement end pointer
        INC     SI              ; Increment starting pointer
        DEC     CL              ; Decrement counter
        JNZ     BACK4           ; If count not = 0, repeat
        PROMPT  MES12           ; Display message 12
        RET
LAST2:  PROMPT  MES11           ; Display message 11
        RET
        ENDP


D_STR  PROC  NEAR
        LEA     BX,BUFF
        MOV     CH,00H          ; [ Take character
        MOV     CL,BUFF +1      ;    count in
        MOV     DI,00           ;    DI ]
BACK:   MOV     DL,[BX+DI+2]    ; Point to the start
                                ; character and read it
        MOV     AH,02H
        INT     21H             ; Display the character
        INC     DI              ; Decrement count
        LOOP    BACK            ; Repeat until count is 0
        RET
        ENDP
```

```
DR_STR  PROC  NEAR
          LEA  BX,BUFF
          MOV  CH,00H          ;  [ Take character
          MOV  CL,BUFF+1       ;     count in
          MOV  DI,CX           ;     DI ]
BACK3:    MOV  DL,[BX+DI+1]    ;  Point to the start
                               ;  character and read it
          MOV  AH,02H
          INT  21H             ;  Display the character
          DEC  DI              ;  Decrement count
          JNZ  BACK3           ;  Repeat until count is 0
          RET
          ENDP

D_HEX   PROC  NEAR
          MOV  AH,  00H        ;  Clear AH
          AAM                  ;  Convert to BCD
          ADD  AX,  3030H        ;  Convert to ASCII
          MOV  BX,AX           ;  Save result
          MOV  DL,  BH         ;  Load first digit (MSD)
          MOV  AH,  02         ;  Load function number
          INT  21H             ;  Display first digit (MSD)
          MOV  DL,  BL         ;  Load second digit (LSD)
          INT  21H             ;  Display second digit (LSD)
          RET
          ENDP
END
```

Program 18 : String Manipulations

Program Statement :

Write 8086 ALP to perform string manipulation. The strings to be accepted from the user is to be stored in code segment Module_1 and write FAR PROCEDURES in code segment Module_2 for following operations on the string.

a. Concatenation of two strings.

b. Compare two strings.

c. Number of occurrences of a sub-string in the given string.

d. Find number of words, characters and capital letters from the given text in the data segment.

Note : Use PUBLIC and EXTERN directive. Create •OBJ files of both the modules and link them to create an EXE file. Command : Tlink M1.0BJ M2.0BJ

In this experiment we have to write two •asm programs one for accepting strings and one for procedures.

Algorithm : Module_1

1. Display Menu
   a. Enter the strings.
   b. Concatenation of two strings.
   c. Compare two strings.
   d. Find number of occurrences of a substring.
   e. Find words, characters and capital letters.
   f. Exit.
2. Read option
   It's option is
   1. Read two strings.
   2. Concatenate two strings.
   3. Compare two strings.
   4. Find number of occurrences of a substring.
   5. Find words, characters and capital letters.
   6. Exit.
3. Stop

M1 : String operations

```
PROMPT  MACRO  MESSAGE  ;Define macro with MESSAGE as a parameter
              PUSH     AX                ; save registers
              PUSH     DX
              MOV      AH,  09H          ; display message
              LEA      DX,  MESSAGE
              INT      21H
              POP      DX                ; restore registers
              POP      AX
              ENDM


.MODEL  SMALL                           ; select small model
.STACK  100
```

```
.DATA               ; start data segment
                PUBLIC BUFF1
                PUBLIC BUFF2
                PUBLIC BUFF3
          MES1      DB 10, 13, '1. ENTER THE STRING $'
          MES2      DB 10, 13, '2. CONCATENATION OF TWO STRINGS $'
          MES3      DB 10, 13, '3. COMPARE TWO STRINGS $'
          MES4      DB 10, 13, '4. NUMBER OF OCCURENCES OF A
                               SUBSTRING $'
          MES5      DB 10, 13, '5. FIND WORDS,CHARACTERS AND CAPITAL
                               LETTERS $'
          MES6      DB 10, 13, '6. EXIT $'
          MES7      DB 10, 13, 'ENTER THE CHOICE : $'
          MES8      DB 10, 13, 'ENTER CORRECT CHOICE : $'
          MES9      DB 10, 13,  '$'
          MES10     DB 10, 13, 'STRING IS MISSING - PLEASE ENTER
                               THE STRING$'
          MES11     DB 10, 13, 'CONCATENATED STRING IS : $'
          MES12     DB 10, 13, 'TWO STRINGS ARE SAME $'
          MES13     DB 10, 13, 'TWO STRINGS ARE NOT SAME $'

          FLAG      DB 0
          MES14     DB 10,13, 'ENTER THE STRING: $'
          MES15     DB 10,13, 'THE STRING IS : $'

          BUFF1     DB 80
                    DB 0
                    DB 80 DUP(0)
          BUFF2     DB 80
                    DB 0
                    DB 80 DUP(0)
          BUFF3     DB 80
                    DB 0
                    DB 80 DUP(0)
.CODE                   ; start code segment

             EXTRN CON_STR:FAR
             EXTRN COM_STR:FAR
             EXTRN SUB_STR:FAR
             EXTRN FWCC_STR:FAR
```

```
START:  MOV     AX, @DATA     ; [Initialize
        MOV     DS, AX        ;  data segment]
        MOV     ES, AX
BEGIN:  PROMPT  MES9          ; Display MES8
        PROMPT  MES9          ; Display MES8
        PROMPT  MES1          ; Display MES1
        PROMPT  MES2          ; Display MES2
        PROMPT  MES3          ; Display MES3
        PROMPT  MES4          ; Display MES4
        PROMPT  MES5          ; Display MES5
        PROMPT  MES6          ; Display MES6
        PROMPT  MES7          ; Display MES7

AGAIN:  MOV     AH,01         ; [ READ
        INT     21H           ;   OPTION ]

        CMP     AL,'6'        ; [ If choice is 6
        JZ      LAST          ;   exit ]

        MOV     BL,FLAG       ; Check for first occurrence
        CMP     BL,0
        JNZ     SKIP          ; if not skip
        CMP     AL,'1'        ; check if choice is 1
        JE      SKIP          ; if yes skip
        PROMPT  MES10         ; otherwise give error message
        JMP     BEGIN         ; and enter the strings

  SKIP: PROMPT  MES9          ; Display MES9

        CMP     AL,'1'        ; [ If choice is 1
        JNZ     NEXT1

        LEA     DX, BUFF1
        CALL    E_STR         ;  Enter the string1

        LEA     DX, BUFF2
        CALL    E_STR         ;  Enter the string2
        MOV     FLAG,1
        JMP     BEGIN         ;   exit ]
```

```
NEXT1:  CMP     AL,'2'    ; [ If choice is 2
        JNZ     NEXT2
        CALL    CON_STR   ;    Concatenate two strings
        JMP     LAST      ;    exit ]


NEXT2:  CMP     AL,'3'    ; [ If choice is 3
        JNZ     NEXT3
        CALL    COM_STR   ;    Compare two string
        JMP     BEGIN     ;    exit ]

NEXT3:  CMP     AL,'4'    ; [ If choice is 4
        JNZ     NEXT4
        CALL    SUB_STR   ; Find number of occurences of a
                          ; sub-string in the given string
        JMP     BEGIN     ;    exit ]
NEXT4:  CMP     AL,'5'    ; [ If choice is 4
        JNZ     NEXT5
        CALL    FWCC_STR  ; Find word, character and capital
                          ; letters in the string
        JMP     BEGIN     ;    exit ]

NEXT5:  PROMPT  MES8      ; Display MES8
        JMP     AGAIN

LAST:   MOV     AH,4CH    ; Return to DOS
        INT     21H

E_STR  PROC  NEAR
        PROMPT  MES1      ; Display message MES1
        MOV     AH,0AH    ; I/P the string.
        INT     21H
        RET
        ENDP

END
```

M2 : For string operations

```
.MODEL  SMALL
.STACK  100
.DATA
        EXTRN  BUFF1:BYTE
        EXTRN  BUFF2:BYTE
        EXTRN  BUFF3:BYTE
        MESS1  DB  10,13,'STRINGS  ARE  SAME  $'
        MESS2  DB  10,13,'STRINGS  ARE  NOT  SAME  $'
        MESS3  DB  10,13,'NUMBER  OF  ALPHABETS  IN  THE  STRING
               ARE:$'
        MESS4  DB  10,13,'NUMBER  OF  CAPITAL  LETTERS  IN  THE  STRING
               ARE:$'
        MESS5  DB  10,13,'NUMBER  OF  WORDS  IN  THE  STRING  ARE  :  $'
        MESS6  DB  10,13,'NUMBER  OF  OCCURRENCES  OF  SUBSTRING  IN
               THE  STRING  ARE  :  $'
        WFLAG  DB   0
        ACOUNT DB   0
        CCOUNT DB   0
        WCOUNT DB   1
        C_ADDR DW   ? ;  current  address  of  pointer
        E_ADDR DW   ? ;  End  address  of  string

.CODE

        PUBLIC  CON_STR
        PUBLIC  COM_STR
        PUBLIC  SUB_STR
        PUBLIC  FWCC_STR

CON_STR  PROC  FAR

        CLD
        MOV  CH,00      ;  copy  string  1
        MOV  CL,  BUFF1+1
        LEA  SI,  BUFF1+2
        LEA  DI,  BUFF3+2
        REPZ  MOVSB

        MOV  CH,00      ;  copy  string  2
        MOV  CL,  BUFF2+1
```

```
        LEA  SI,  BUFF2+2
        REPZ MOVSB

        MOV  CL,  BUFF1+1  ; calculate and store length of
                             concatenated string
        ADD  CL,  BUFF2+1  ;
        MOV  BUFF3+1,CL

        MOV  CH,00         ; Display concatenated string
        LEA  SI,BUFF3+2
DISNEXT: MOV  AH,02H
        MOV  DL,[SI]
        INT  21H
        INC  SI
        LOOP  DISNEXT
        RET
        CON_STR   ENDP

COM_STR  PROC  FAR

        MOV  CH,BUFF1+1    ; check two string character by character
        MOV  CL,BUFF2+1
        CMP  CH,CL
        JNZ  NOTEQ
        CLD
        MOV  CH,00
        LEA  SI,BUFF1+2
        LEA  DI,BUFF2+2
        REPE  CMPSB
        JNZ  NOTEQ

        MOV  AH,09H        ; if equal display message accordingly
        LEA  DX,MESS1
        INT  21H
        JMP  RE
NOTEQ:  MOV  AH,09H        ; if not equal display message
                             accordingly
        LEA  DX,MESS2
        INT  21H
```

```
RE:       RET
          COM_STR   ENDP

  SUB_STR  PROC  FAR

          MOV  BL,00
          LEA  SI,  BUFF1+2
          MOV  C_ADDR,SI     ; Load current address
          MOV  DL,  BUFF1+1
          MOV  DH,00
          MOV  AX,  SI
          ADD  AX,DX
          MOV  E_ADDR,AX     ; load end address
ST1:      MOV  CH,0
          MOV  CL,BUFF2+1    ; load length of substring
          LEA  DI,BUFF2+2    ; initialize pointer to substring
BBB:      MOV  BH,[SI]
          CMP  BH,BYTE PTR [DI] ; compare substring characters
          JNZ  NNNEXT        ; if not equal go to NNNEXT
          INC  SI            ; otherwise increment character pointers
          INC  DI            ; and confine
          LOOP  BBB
          INC  BL            ; if substring occurs increment count
          CMP  SI,  E_ADDR   ; check for end of string
          JNZ  ST1           ; if not zero go to check more
                               occurrences
          JMP  LLLAST        ; if end of string go to display number
                               of occurrences
NNNEXT:  MOV  SI,C_ADDR
          INC  SI
          MOV  C_ADDR,SI     ; modify current address
          CMP  SI,  E_ADDR
          JNZ  ST1
LLLAST:
          MOV  AH,09H        ; display number of occurrences of string
          LEA  DX,  MESS6
          INT  21H
          MOV  AL,BL
          CALL  DIS_HEX
          RET
```

```
          SUB_STR   ENDP

FWCC_STR  PROC  FAR

          MOV  CH,  00
          MOV  CL,  BUFF1+1
          LEA  SI,  BUFF1+2
BB:       MOV  AL,[SI]          ; check of space
          CMP  AL,' '
          JNZ  NNEXT
          MOV  AL,WFLAG         ; if space occurs increment word count
          CMP  AL,0
          JZ   LLAST
          MOV  WFLAG,0
          INC  WCOUNT
          JMP  LLAST
NNEXT:    MOV  WFLAG,1
                                ; .IF AL >= 'A' && AL <= 'Z'
          CMP  AL,'A'
          JB   LLAST              ; check if alphabet
          CMP  AL,'Z'           ; if yes increment alphabet count
          JA   NNEXT1
          INC  ACOUNT
          INC  CCOUNT
                                ;.ENDIF

 NNEXT1:                        ;.IF AL >= 'a' && AL <= 'z'
          CMP  AL,'a'
          JB   LLAST            ; check if alphabet
          CMP  AL,'z'           ; if yes increment alphabet count
          JA   LLAST
          INC  ACOUNT
                                ;.ENDIF
LLAST:    INC  SI
          LOOP BB

          MOV  AH,09H           ; display alphabet count
          LEA  DX,MESS3
          INT  21H
          MOV  AL,ACOUNT
```

```
        CALL DIS_HEX

        MOV  AH,09H         ; display character count
        LEA  DX,MESS4
        INT  21H
        MOV  AL,CCOUNT
        CALL DIS_HEX

        MOV  AH,09H         ; display word count
        LEA  DX,MESS5
        INT  21H
        MOV  AL,WCOUNT
        CALL DIS_HEX

        MOV  ACOUNT,0
        MOV  CCOUNT,0
        MOV  WCOUNT,1

        RET
        FWCC_STR   ENDP
```

```
DIS_HEX  PROC  NEAR
    MOV  AH,  00H                ; Clear AH
    AAM
                                 ; Convert to BCD
    ADD  AX,  3030H              ; Convert to ASCII
    MOV  BX,AX                   ; Save result
    MOV  DL,  BH                 ; Load first digit (MSD)
    MOV  AH,  02                 ; Load function number
    INT  21H                     ; Display first digit (MSD)
    MOV  DL,  BL                 ; Load second digit (LSD)
    INT  21H                     ; Display second digit (LSD)
    RET
    ENDP
END
```

Program 19 : Sorting of Array

Program Statement : Write 8086 ALP to arrange the numbers stored in the array in ascending as well as descending order. Assume that the first location in the array holds the number of elements in the array and successive memory locations will be actual array elements. Write separate subroutine to arrange the numbers in ascending and descending order. Accept key from the user.

If user enters 1 : Arrange in ascending order

If user enters 2 : Arrange in descending order

Sorting of Array

```
PROMPT  MACRO  MESSAGE          ; Define macro with MESSAGE as a
          PUSH    AX           ; parameter save register
          MOV     AH,  09H     ; display message
          LEA     DX,  MESSAGE
          INT     21H
          POP     AX           ; restore register
          ENDM

.MODEL  SMALL
.STACK  100
.DATA
    ARRAY    DB  10,  53H,20H,30H,25H,50H,09H,70H,13H,90H,00H
    MES1     DB   10,13,  '1.  SORT  ARRAY  IN  THE  ASCENDING  ORDER  $'

    MES2     DB  10,13,  '2.  SORT  ARRAY  IN  THE  DESCENDING ORDER  $'
    MES3     DB  10,13,  '3.  EXIT  $'
    MES4     DB  10,13,  'ENTER  THE  CHOICE  :    $'
    MES5     DB  10,13,  'SORTED  ARRAY  IS  :  $'
    MES6     DB  10,13,  'ENTER  CORRECT  CHOICE  :  $'
    MES7     DB  10,13,  '$'
.CODE
START:   MOV      AX,@data      ; [  Initialise
         MOV      DS,AX         ;     data  segment  ]

         PROMPT   MES1
         PROMPT   MES2
         PROMPT   MES3
         PROMPT   MES4
```

```
    ST1: MOV     AH,01H
         INT     21H

         CMP     AL,'3'
         JZ      LAST

         CMP     AL,'1'
         JNZ     NEXT
         PROMPT  MES7
         CALL    ASC
         JMP     LAST
NEXT:    CMP     AL,'2'
         JNZ     NEXT1
         PROMPT  MES7
         CALL    DSC
         JMP     LAST
NEXT1:   PROMPT  MES6
         JMP     ST1
LAST:    MOV     AH,  4CH
         INT     21H
ASC  PROC  NEAR
         MOV     CL,ARRAY      ; Initialise counter1
BBB1:    MOV     CH,ARRAY      ; Initialise counter2
         DEC     CH
         XOR     DI,DI         ; Initialise pointer
         LEA     BX,ARRAY      ; Initialise array base pointer
BACK1:   MOV     DL,[BX+DI+1]  ; Get the number
         CMP     DL,[BX+DI+2]  ; Compare it with next number
         JBE     SKIP1

         MOV     AH,[BX+DI+2]  ; Otherwise
         MOV     [BX+DI+2],DL  ; exchange
         MOV     [BX+DI+1],AH  ; two numbers

SKIP1:   INC     DI
         DEC     CH
         JNZ     BACK1
         DEC     CL
         JNZ     BBB1
```

```
        PROMPT  MES5
        MOV     CH,00
        MOV     CL,ARRAY
        LEA     DI,ARRAY
        INC     DI
AG1:    INC     DI
        MOV     AL,[DI]
        CALL    D_HEX2          ; Display sorted array
        PUSH    AX
        PUSH    LX
        MOV     AH,02H
        MOV     DL,' '
        INT     21H
        POP     DX
        POP     AX
        LOOP    AG1
        RET
        ENDP

DSC  PROC  NEAR
        MOV  CL,ARRAY           ; Initialise counter1
BBB:    MOV  CH,ARRAY           ; Initialise counter2
        DEC  CH
        XOR  DI,DI              ; Initialise pointer
        LEA  BX,ARRAY           ; Initialise array base pointer
BACK:   MOV  DL,[BX+DI+1]       ; Get the number
        CMP  DL,[BX+DI+2]       ; Compare it with next number
        JAE  SKIP

        MOV  AH,[BX+DI+2]       ; Otherwise
        MOV  [BX+DI+2],DL       ; exchange
        MOV  [BX+DI+1],AH       ; two numbers

SKIP:   INC  DI
        DEC  CH
        JNZ  BACK
        DEC  CL
        JNZ  BBB
```

```
        PROMPT  MES5
        MOV     CH,00
        MOV     CL,ARRAY
        LEA     DI,ARRAY
        INC     DI
AG:     INC     DI
        MOV     AL,[DI]
        CALL    D_HEX2          ; Display sorted array
        PUSH    AX
        PUSH    DX
        MOV     AH,02H
        MOV     DL,' '
        INT     21H
        POP     DX
        POP     AX
        LOOP    AG
        RET
        ENDP

D_HEX2  PROC NEAR
        PUSH    CX
        MOV     CL, 04H         ; Load rotate count
        MOV     CH, 02H         ; Load digit count
BAC:    ROL     AX, CL          ; rotate digits
        PUSH    AX              ; save contents of AX
        AND     AL, 0FH         ; [Convert
        CMP     AL,9            ;  number
        JBE     Add30           ; to
        ADD     AL, 37H         ; its
        JMP     DISP            ; ASCII
Add30:
        ADD     AL,30H          ; equivalent]
```

```
DISP:   MOV  AH,02H
        MOV  DL,AL          ; [Display the
        INT  21H            ;  number]
        POP  AX             ; restore contents of AX
        DEC  CH             ; decrement digit count
        JNZ  BAC            ; if not zero repeat
        POP  CX
        RET
        ENDP
        END
```

Program 20 : Program to search a given byte in the string

```
.MODEL SMALL
.DATA
        M1      DB   10, 13, 'ENTER THE STRING : $'
        M2      DB   10, 13, 'GIVEN BYTE IS NOT IN THE STRING $'
        CHAR    DB   0
        ADDR    DB   0
        BUFF    DB   80
                DB   0
                DB   80 DUP (0)
.CODE
START :         MOV  AX,@data       ; [ Initialise
                MOV  DS,AX          ;   data segment ]
                MOV  AH,09H         ; Display message1
                MOV  DX,OFFSET M1
                INT  21H
                MOV  AH,0AH         ; Input the string
                LEA  DX,BUFF
                INT  21H
                MOV  AH,01          ; [Read character
                INT  21H            ;  from keyboard]
                MOV  CHAR,AL        ; save character
                MOV  CH,00H
                MOV  CL,BUFF+1      ; Take character count in CX
                LEA  BX,BUFF+2
                MOV  DI,00H
BACK :          MOV  DL,[BX+DI]     ; point to the first character
                CMP  DL, CHAR       ; compare string character with
                                    ; given character
                JZ   NEXT           ; if match occurs go to next
                INC  DI
                DEC  CX             ; Decrement character counter
                JNZ  BACK           ; If not = 0, repeat
                MOV  AH,09H         ; [Display message M2
                LEA  DX,M2          ;  on the
                INT  21H            ;  monitor]
```

```
                JMP   LAST
NEXT:           MOV ADDR,DI        ; save relative address of the
                                   ; byte from the starting
                                   ; location of the string
LAST:           MOV AH,4CH         ; [ Terminate and
                INT 21H            ;   Exit to DOS ]
                END START
```

Program 21 : Program to find LCM of two 16-bit unsigned numbers

(Softcopy of this program, P24.asm is available at www.vtubooks.com)

If we divide the first number by the second number and there is no remainder, then the first number is the LCM. In case of remainder, it is necessary to add first number to it to get the new first number. After addition we have to divide the new first number by the second number to check if the remainder is zero. If remainder is not zero again add the original first number to new one and repeat the process.

For example, if two numbers are 20 and 15 then we get LCM as follows :

$20 \div 15 = 1$ Remainder 5 i.e. $\neq 0$

$\therefore \quad 20 + 20 = 40 \div 15 = 2$ Remainder 10 i.e. $\neq 0$

$\therefore \quad 40 + 20 = 60 \div 15 = 4$ Remainder 0

$\therefore \quad$ LCM = 60

```
NAME LCM
PAGE 60,80
TITLE program to find LCM of two 16-bit unsigned numbers
.MODEL SMALL
.STACK 64
.DATA
    NUMBERS  DW 0020, 0015
    LCM      DW 2 DUP (?)
.CODE
START:          MOV AX,@DATA       ; [Initialize
                MOV DS,AX          ;  data segment]
                MOV DX,0
                MOV AX,NUMBERS     ; Get the first number
                MOV BX,NUMBERS+2   ; Get the second number
BACK:           PUSH AX            ; [ Save the
                PUSH DX            ;   first number]
                DIV BX             ; Divide if by second number
                CMP DX,0           ; Check if remainder = 0
                JE EXIT            ; if remainder = 0 then exit
                POP DX
                POP AX
                ADD AX,NUMBERS     ; First number + first number
                JNC SKIP
                INC DX
```

```
SKIP:       JMP  BACK          ; Goto BACK
EXIT:       POP  LCM+2         ; [ Get
            POP  LCM           ;   the LCM ]
            MOV  AH,4CH        ; [ Terminate and
            INT  21H           ;   Exit to DOS ]
            END  START
```

Program 22 : Program to find HCF of two numbers.

(Softcopy of this program, P25.asm is available at www.vtubooks.com)

To find the HCF of two numbers we have to divide greater number by smaller number, if remainder is zero, divisor is a HCF. If remainder is not zero, remainder becomes new divisor and previous divisor becomes dividend and this process is repeated until we get remainder 0.

For example, if numbers are 20 and 15 we can find HCF as follows :

$$20 \div 15 = 1 \text{ Remainder } 5 \text{ i.e. } \neq 0$$

$$\therefore \quad 15 \div 5 = 3 \text{ Remainder } 0$$

$$\therefore \quad \text{HCF} = 5$$

```
.model small
.stack 100
.data
     CR        EQU 0AH
     LF        EQU 0DH
     MES_1     DB  CR,LF,'ENTER 4-DIGIT FIRST HEX NO',CR,LF,'$'
     MES_2     DB  CR,LF,'ENTER 4-DIGIT SECOND HEX NO',CR,LF,'$'
     MES_3     DB  CR,LF,'INPUT IS INVALID BCD $'
     MES_4     DB  CR,LF,'THE HCF IS : $'
     MULTI         DW 1,10,100,1000
     RESULT        DW (00)
     DIVISOR       DW (00)
     DIVIDEND      DW (00)

     INP1          DB 05
                   DB 00
                   DB 05 DUP(0)
     INP2          DB 05
                   DB 00
                   DB 05 DUP(0)
.code
MAIN:       MOV  AX,@data          ; [ Initialise
            MOV  DS,AX             ;   data segment ]
            MOV  AH,09H            ; [  Display
            MOV  DX,OFFSET MES_1   ;    MES_1
            INT  21H               ;    on video screen
            LEA  DX,INP1           ; [  Get the
            MOV  AH,0AH            ;    First
```

```
        INT 21H                 ;     HEX number ]
        MOV AH,09H              ; [   Display
        MOV DX,OFFSET MES_2     ;     MES_2
        INT 21H                 ;     on video screen ]
        LEA DX,INP2             ; [Get the
        MOV AH,0AH              ;   Second
        INT 21H                 ;   HEX number ]
        MOV CH,02H              ; Initialize buffer counter
        LEA BX,INP1             ; Get the address of buffer
AGAIN:  INC BX                  ; [   Adjust buffer
        INC BX                  ;     pointer ]
        XOR DI,DI               ; Clear pointer
        MOV CL,04               ; Initialize counter for digits
BACK:   MOV AL,[BX+DI]          ; Get the digit from buffer
        CMP AL,39H              ; [ Convert
        JG   NEXT               ;   the ASCII
        SUB AL,30H              ;   code of
        JMP SKIP                ;   the actual number
NEXT:   SUB AL,37H              ;   and store it in the same
SKIP:   MOV [BX+DI],AL          ;  position ]
        INC DI                  ; Increment pointer
        DEC CL                  ; Decrement digit counter
        JNZ BACK                ; If not zero goto BACK
        LEA BX,INP2             ; Point to second buffer
        DEC CH                  ; Decrement buffer counter
        JNZ AGAIN               ; If not zero goto AGAIN
        MOV CL,4                ; Initialize rotation counter
        LEA BX,INP1             ; Point to first buffer
        INC BX                  ; [Adjust buffer
        INC BX                  ;  pointer ]
        MOV AH,[BX+0]           ; [Forms the
        SAL AH,CL               ;  packed BCD
        AND AH,0F0H             ;  Higher
        MOV AL,[BX+1]           ;  Byte ]
        OR  AH,AL
        MOV AL,[BX+2]           ; [Forms the
        SAL AL,CL               ;  packed BCD
        AND AL,0F0H             ;  Lower
        MOV DH,[BX+3]           ;  byte ]
        OR  AL,DH
        MOV RESULT,AX           ; Save packed word as a RESULT
        MOV CL,4                ; Initialize rotation counter
        LEA BX,INP2             ; Point to second buffer
        INC BX                  ; [   Adjust buffer
        INC BX                  ;     pointer ]
        MOV AH,[BX+0]           ; [Forms the
        SAL AH,CL               ;  packed BCD
        AND AH,0F0H             ;  Higher
        MOV AL,[BX+1]           ;  byte]
        OR  AH,AL
        MOV AL,[BX+2]           ; [Forms the
```

```
            SAL  AL,CL              ;  packed BCD
            AND  AL,0F0H            ;  lower
            MOV  DH,[BX+3]          ;  byte ]
            OR   AL,DH
            CMP  AX,RESULT          ; Compare two packed words
            JNC  NEXT1
            MOV  DIVISOR,AX         ; Assign smaller word as a
            MOV  CX,RESULT          ; DIVISOR and
            MOV  DIVIDEND,CX        ; greater word as a DIVIDEND
            JMP  SKIP1
NEXT1:      MOV  DIVIDEND,AX        ; Assign greater word as a
            MOV  CX,RESULT          ; DIVIDEND and
            MOV  DIVISOR,CX         ; smaller word as a DIVISOR
SKIP1:      MOV  DX,0
            MOV  AX,DIVIDEND
            DIV  DIVISOR            ; Perform division
            CMP  DX,0               ; Check remainder for zero
            MOV  CX,DIVISOR
            MOV  DIVISOR,DX         ; Load remainder as a new
                                    ; DIVISOR
            MOV  DIVIDEND,CX        ; Load previous DIVISOR as a
                                    ; new DIVIDEND
            JNZ  SKIP1              ; If remainder is not zero
                                    ; goto SKIP1
            MOV  AH,09H             ; [Display
            LEA  DX,MES_4           ;  MES_4
            INT  21H                ;  on video screen ]
            ADD  CL,30H             ; [Display the DIVISOR
            MOV  DL,CL              ;  when remainder
            MOV  AH,02H             ;  is zero
            INT  21H                ;  i.e. HCF ]
            MOV  AH,4CH             ; [Terminate and
            INT  21H                ;   Exit to DOS ]
            END  MAIN
            END
```

Program 23 : Program to find LCM of two given numbers.

(Softcopy of this program, P26.asm is available at www.vtubooks.com)

There is a one more method to find LCM of two number if HCF is known. We can find LCM as follows :

$$LCM = [number1 \times number\ 2] \div HCF$$

This program accepts two four digit numbers from keyboard, finds HCF first and using above equation it then finds LCM of the two numbers.

```
.model  small
.stack  100
.data
        CR      EQU  0AH
        LF      EQU  0DH
        MES_1   DB  CR,LF,'ENTER  4-DIGIT  FIRST  HEX  NO',CR,LF,'$'
        MES_2   DB  CR,LF,'ENTER  4-DIGIT  SECOND  HEX  NO',CR,LF,'$'
        MES_3   DB  CR,LF,'INPUT  IS  INVALID  BCD  $'
        MES_4   DB  CR,LF,'THE  HCF  IS  :  $'
        MULTI       DW  1,10,100,1000
        RESULT      DW  (00)
        DIVISOR     DW  (00)
        DIVIDEND    DW  (00)

        INP1        DB  05
                    DB  00
                    DB  05  DUP(0)
        INP2        DB  05
                    DB  00
                    DB  05  DUP(0)
.code
MAIN:       MOV AX,@data            ;  [  Initialise
            MOV DS,AX               ;     data  segment]
            MOV AH,09H              ;  [  Display
            MOV DX,OFFSET  MES_1    ;     MES_1
            INT 21H                 ;     on  video  screen  ]
            LEA  DX,INP1            ;  [  Get  the
            MOV  AH,0AH             ;     First
            INT  21H                ;     HEX  number  ]
            MOV AH,09H              ;  [  Display
            MOV  DX,OFFSET  MES_2   ;     MES_2
            INT 21H                 ;     on  video  screen  ]
            LEA  DX,INP2            ;  [  Get  the
            MOV  AH,0AH             ;     Second
            INT  21H                ;     HEX  number  ]
            MOV  CH,02H             ;  Initialize  buffer  counter
            LEA  BX,INP1            ;  Get  the  buffer  pointer
AGAIN:      INC  BX                 ;  [  Adjust  buffer
            INC  BX                 ;     pointer  ]
            XOR  DI,DI              ;  Clear  pointer
            MOV  CL,04              ;  Initialize  counter  for  digits
BACK:       MOV  AL,[BX+DI]         ;  Get  the  digit  from  buffer
            CMP  AL,39H             ;  [  Convert
            JG    NEXT              ;     the  ASCII
```

```
        SUB  AL,30H          ;    code
        JMP  SKIP            ;    the actual number
NEXT:   SUB  AL,37H          ;    and store it in the same
SKIP:   MOV  [BX+DI],AL      ;    position ]
        INC  DI              ; Increment pointer
        DEC  CL              ; Decrement digit counter
        JNZ  BACK            ; If not zero goto BACK
        LEA  BX,INP2         ; Point to second buffer
        DEC  CH              ; Decrement buffer counter
        JNZ  AGAIN           ; If not zero goto AGAIN
        MOV  CL,4            ; Initialize rotation counter
        LEA  BX,INP1         ; Point to first buffer
        INC  BX              ; [ Adjust buffer
        INC  BX              ;   pointer ]
        MOV  AH,[BX+0]       ; [ Forms the
        SAL  AH,CL           ;   packed BCD
        AND  AH,0F0H         ;   Higher
        MOV  AL,[BX+1]       ;   Byte ]
        OR   AH,AL           ;
        MOV  AL,[BX+2]       ; [ Forms the
        SAL  AL,CL           ;   packed BCD
        AND  AL,0F0H         ;   Lower
        MOV  DH,[BX+3]       ;   byte ]
        OR   AL,DH
        MOV  RESULT,AX       ; Save the packed word as a
                             ; RESULT
        MOV  CL,4            ; Initialize rotation counter
        LEA  BX,INP2         ; Point to second buffer
        INC  BX              ; [ Adjust buffer
        INC  BX              ;   pointer ]
        MOV  AH,[BX+0]       ; [ Forms the
        SAL  AH,CL           ;   packed BCD
        AND  AH,0F0H         ;   Higher
        MOV  AL,[BX+1]       ;   byte ]
        OR   AH,AL
        MOV  AL,[BX+2]       ; [ Forms the
        SAL  AL,CL           ;   packed BCD
        AND  AL,0F0H         ;   lower
        MOV  DH,[BX+3]       ;   byte ]
        OR   AL,DH
        MOV  RESULT1,AX      ; Save second pack word as
                             ; a RESULT2
        CMP  AX,RESULT       ; Compare two packed words
        JNC  NEXT1
```

```
            MOV  DIVISOR,AX        ; Assign smaller word as a
            MOV  CX,RESULT         ; DIVISOR and
            MOV  DIVIDEND,CX       ; greater word as a DIVIDEND
            JMP  SKIP1
NEXT1:
            MOV  DIVIDEND,AX       ; Assign greater word as a
            MOV  CX,RESULT         ; DIVIDEND and
            MOV  DIVISOR,CX        ; smaller word as a DIVISOR
SKIP1:      MOV  DX,0
            MOV  AX,DIVIDEND
            DIV  DIVISOR           ; Perform division
            CMP  DX,0              ; Check remainder for zero
            MOV  CX,DIVISOR
            MOV  DIVISOR,DX        ; Load remainder as a new
                                   ; DIVISOR
            MOV  DIVIDEND,CX       ; Load previous DIVISOR as a
                                   ; new DIVIDEND
            JNZ  SKIP1             ; If remainder is not zero
                                   ; goto SKIP1
            MOV  AH,09H            ; [ Display
            LEA  DX,OFFSET MES_3   ;   MES_3
            INT  21H               ;   on video screen ]
; Number1 × Number2 = HCF × LCM ∴ LCM =(Number1 × Number2)/HCF
            MOV  HCF,CX
            MOV  DX,0
            MOV  AX,RESULT         ; Get the first number
            MUL  RESULT1           ; Multiply number1 and number2
            DIV  HCF               ; Divide multiplication by HCF
            MOV  CL,4              ; Initialize rotation counter
            MOV  BX,AX             ; Save the quotient (LCM)
            AND  AH,0F0H           ; [ Display the LCM
            SAR  AH,CL             ;   on the video screen ]
            CMP  AH,09H
            JNC  SKIP2
            ADD  AH,30H
            JMP  NEXT2
SKIP2:      ADD  AH,37H
NEXT2:      MOV  DL,AH
            MOV  AH,02H
            INT  21H
            MOV  AX,BX
            AND  AH,0FH
            CMP  AH,09H
            JNC  SKIP3
```

```
            ADD  AH,30H
            JMP  NEXT3
SKIP3:      ADD  AH,37H
NEXT3:      MOV  DL,AH
            MOV  AH,02H
            INT  21H
            MOV  AX,BX
            AND  AL,0F0H
            SAR  AL,CL
            CMP  AL,09H
            JNC  SKIP4
            ADD  AL,30H
            JMP  NEXT4
SKIP4:      ADD  AL,37H
NEXT4:      MOV  DL,AL
            MOV  AH,02H
            INT  21H
            MOV  AX,BX
            AND  AL,0FH
            CMP  AL,09H
            JNC  SKIP5
            ADD  AL,30H
            JMP  NEXT5
SKIP5:      ADD  AL,37H
NEXT5:      MOV  DL,AL
            MOV  AH,02H
            INT  21H
            MOV  AH,4CH          ; [ Terminate and
            INT  21H             ;   Exit to DOS ]
            END  MAIN
            END
```

❑❑❑

5

8086 System Configuration

5.1 Introduction

Unlike 8085, 8086 and 8088 can be operated in two modes : Minimum mode and Maximum mode. In this chapter we study the topics related to Minimum mode and Maximum mode operation of 8086. Topics include clock generation, bus buffering, bus latching, timings, minimum mode operation and maximum mode operation. Let us begin with signal description of 8086.

5.2 Signal Description of 8086

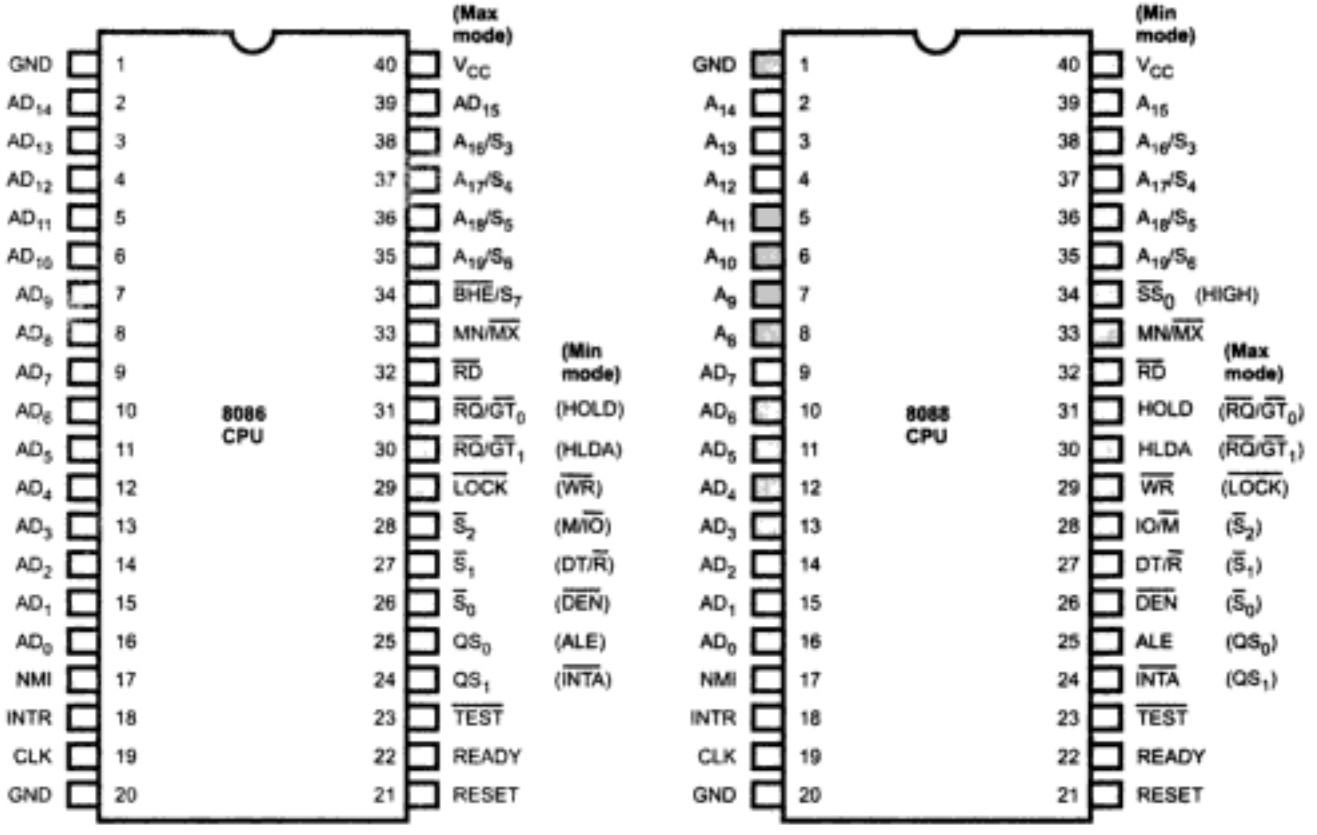
In order to implement many situations in the microcomputer system the 8086 and 8088 has been designed to work in two operating modes :

1. Minimum Mode
2. Maximum Mode

The minimum mode is used for a small systems with a single processor and maximum mode is for medium size to large systems, which often include two or more processors. Fig. 5.1 shows the pin diagram of 8086 and 8088 in minimum as well as maximum mode. As a close comparison reveals, there is no much difference between two microprocessors - both are packaged in 40-pin dual-in-line package (DIPs). As mentioned in section 2.1, the 8086 is a 16-bit microprocessor with a 16-bit data bus, and the 8088 is a 16-bit microprocessor with an 8-bit data bus. The pin-out shows, the 8086 has pin connections AD_0-AD_{15}, and the 8088 has pin connections AD_0-AD_7. There is one more minor difference in one of the control signals. The 8086 has an M/$\overline{IO}$ pin, and the 8088 has an IO/$\overline{M}$ pin. The only hardware difference appears on pin 34 of both chips : on the 8086 it is a $\overline{BHE}/S_7$ pin, while on the 8088 it is a $\overline{SS}_0$ pin.

The 8086 signals can be categorised in three groups.

- Signals having common functions in both minimum and maximum modes.
- Signals having special functions for minimum mode.
- Signals having special functions for maximum mode.



(a) Pin diagram of 8086 (b) Pin diagram of 8088

Fig. 5.1

5.2.1 Signals with Common Functions in Both Modes

1. **AD_{15}-AD_0** : Acts as address bus during the first part of machine cycle and data bus for the remaining part of the machine cycle.
2. **A_{19}/S_6-A_{16}/S_3** : During the first part of machine cycle these are used to output upper 4-bits of address. During remaining part of the machine cycle these are used to output status, which indicates the type of operation to be performed in that cycle. S_3 and S_4 indicate the segment register being used as follows :

S_4	S_3	Register
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

   S_5 gives the current setting of the interrupt flag (IF) and S_6 is always zero.
3. **$\overline{BHE}/S_7$** : BHE (Bus High Enable) : Low on this pin during first part of the machine cycle, indicates that at least one byte of the current transfer is to be made

on higher order byte AD_{15}-AD_8; otherwise the transfer is made on lower order byte AD_7-AD_0.

$\overline{BHE}$	A_0	Data accesses
0	0	Word
0	1	Upper byte from odd address
1	0	Lower byte from even address
1	1	None

Status S_7 is output during the later part of the machine cycle, but, presently, S_7 has not been assigned a meaning.

4. **NMI :** It is a positive edge triggered nonmaskable interrupt request.
5. **INTR :** It is a level triggered maskable interrupt request. It is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt service routine.
6. **CLK :** 8086 requires clock signal (with 33 % duty cycle) from some external, crystal controlled generator to synchronize internal operations. Clock frequency depends on the version of 8086.

Processor	Required clock signal
8086	5 MHZ
8086-2	8 MHZ
8086-1	10 MHZ

7. **RESET :** It clears PSW, IP, DS, SS, ES, and the instruction queue. It then sets CS to FFFFH. This signal must be high for at least 4 clock cycles. When RESET is removed, 8086 will fetch its next instruction from physical address FFFF0H.
8. **READY :** If this signal is low the 8086 enters into wait state. This signal is used primarily to synchronize slower peripherals with the microprocessor.
9. **$\overline{TEST}$ (Input) :** This signal is only used by the WAIT instruction. The 8086 enters into a wait state after execution of the WAIT instruction until a LOW signal on the $\overline{TEST}$ pin. $\overline{TEST}$ signal is synchronized internally during each clock cycle on the leading edge of the clock cycle.
10. **$\overline{RD}$ (Output) :** $\overline{RD}$ is low whenever the 8086 is reading data from memory or an I/O device.
11. **MN/$\overline{MX}$ (Input) :** The 8086 can be configured in either minimum mode or maximum mode using this pin. This pin is tied high for minimum mode.

5.2.2 Signal Definitions (24 to 31) for Minimum Mode

INTA (Interrupt Acknowledge) Output : This indicates recognition of an interrupt request. It consists of two negative going pulses in two consecutive bus cycles. The first pulse informs the interface that its request has been recognized and upon receipt of the second pulse, the interface is to send the interrupt type to the processor over the data bus.

ALE (Address Latch Enable) Output : This signal is provided by 8086 to demultiplex the AD_0-AD_{15} into A_0-A_{15} and D_0-D_{15} using external latches.

$\overline{DEN}$ (Data Enable) Output : This signal informs the transceivers that the CPU is ready to send or receive data.

DT/$\overline{R}$ (Data transmit/Receive) Output : This signal is used to control data flow direction. High on this pin indicates that the 8086 is transmitting the data and low indicates that the 8086 is receiving the data.

M/$\overline{IO}$ Output : It is used to distinguish memory data transfer, (M/$\overline{IO}$ = HIGH) and I/O data transfer (M/$\overline{IO}$ = LOW).

$\overline{WR}$: Write Output : $\overline{WR}$ is low whenever the 8086 is writing data into memory or an I/O device.

HOLD input, HLDA Output : A HIGH on HOLD pin indicates that another master (DMA) is requesting to take over the system bus. On receiving HOLD signal processor outputs HLDA signal HIGH as an acknowledgment. At the same time, processor tristates the system bus. A low on HOLD gives the system bus control back to the processor. Processor then outputs low signal on HLDA.

5.2.3 Signal Definitions (24 to 31) for Maximum Mode

1. **QS_1, QS_0 (output) :** These two output signals reflect the status of the instruction queue. This status indicates the activity in the queue during the previous clock cycle.

QS_1	QS_0	Status
0	0	No operation (queue is idle)
0	1	First byte of an opcode
1	0	Queue is empty
1	1	Subsequent byte of an opcode

2. **$\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$ (output) :** These three status signals indicate the type of transfer to be take place during the current bus cycle.

$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	Machine cycle
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt

$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	Machine cycle
1	0	0	Instruction fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Inactive-Passive

3. **$\overline{\text{LOCK}}$:** This signal indicates that an instruction with a LOCK prefix is being executed and the bus is not to be used by another processor.
4. **$\overline{RQ}/\overline{GT}_1$ and $\overline{RQ}/\overline{GT}_0$:** In the maximum mode, HOLD and HLDA pins are replaced by $\overline{RQ}$ (Bus request)/$\overline{GT}_0$ (Bus Grant), and $\overline{RQ}/\overline{GT}_1$ signals. By using bus request signal another master can request for the system bus and processor communicate that the request is granted to the requesting master by using bus grant signal. Both signals are similar except the $\overline{RQ}/\overline{GT}_0$ has higher priority than $\overline{RQ}/\overline{GT}_1$.

5.3 Physical Memory Organisation

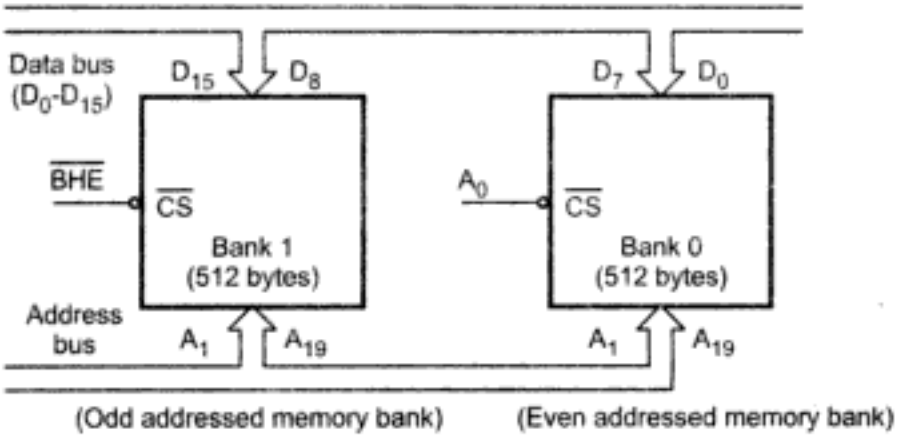


Fig. 5.2 Memory interfacing

Most of the memory ICs are byte oriented i.e. each memory location can store only one byte of data. The 8086 is a 16-bit microprocessor, it can transfer 16-bit data. So in addition to byte, word (16-bit) has to be stored in the memory. This is stored by using two consecutive memory locations, one for least significant byte and other for most significant byte. The address of word is the address of least significant byte. To implement this, the entire memory is divided into two memory banks : $bank_0$ and $bank_1$. Fig. 5.2 shows the interfacing diagram to these memory banks. $Bank_0$ is selected only when A_0 is zero and $Bank_1$ is selected only when $\overline{BHE}$ is zero. A_0 is zero for all even addresses. So $Bank_0$ is usually referred as **even addressed memory** bank. $\overline{BHE}$ is used to access higher order memory bank, referred to as **odd addressed memory** bank.

Together $\overline{BHE}$ and A_0 tell the interface how the data appears on bus. Four possible combinations are shown in the table.

No.	Operation	$\overline{BHE}$	A_0	Data Lines Used
1.	Read/Write a byte at an even address	1	0	D_7 - D_0
2.	Read/Write a byte at an odd address	0	1	D_{15} - D_8
3.	Read/Write a word at an even address	0	0	D_{15} - D_0
4.	Read/Write a word at an odd address	0	1	D_{15}-D_0 in first operation byte from odd bank is transferred.
		1	0	D_7-D_0 in second operation byte from even bank is transferred.

Note : To access odd addressed word two bus cycles are required.

Every microprocessor based system has a memory system. Almost all systems contain two basic types of memory, read-only memory (ROM) and random access memory (RAM) or read/write memory. Read only memory contains system software and permanent system data such as lookup tables, while Random Access Memory contains temporary data and application software. ROMs/PROMs/EPROMs are mapped to cover the CPU's reset address, since these are non-volatile. When the 8086 is reset, the next instruction is fetched from memory location FFFF0H. So in the 8086 systems, the location FFFF0H must be ROM location.

The Fig. 5.3 shows memory map for 8086. Certain locations in 1 Mbyte memory are reserved and some are dedicated for specific CPU operations. Locations from FFFF0H to

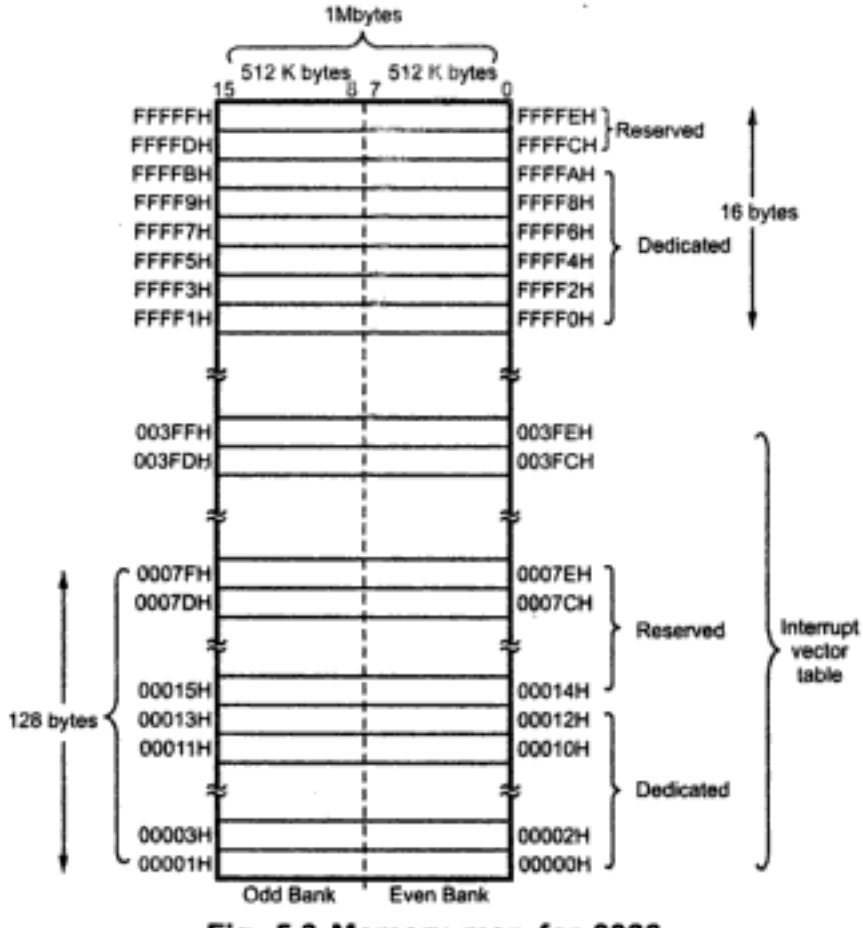


Fig. 5.3 Memory map for 8086

FFFF5H are dedicated to the initialisation procedure of the 8086, while locations FFFF6H to FFFFBH are dedicated to the initialisation procedure of the 8089 input/output processor. Locations 00000H to 00013H are dedicated to store the vector addresses of the dedicated interrupts. The dedicated locations are used for processing of specific system initialisation, interrupt and reset function.

Intel has also reserved several locations for future hardware and software products. Locations from 00014H to 0007FH and locations from FFFFCH to FFFFFH are reserved locations. The locations from 00000H to 003FFH are used for interrupt vector table (IVT). The interrupt vector table provides the starting location/address of the interrupt service routine for the interrupt supported by 8086. The detail description of interrupt vector table is given in sections 8.2 and 8.3.

5.4 I/O Addressing Capability

The 8086 can generate 16-bit of I/O address. Thus it can address upto 64 kbyte I/O locations or 32 K word I/O locations. The 16-bit I/O address appears on A_0 to A_{15} address lines; A_{16} to A_{19} lines are at logic 0 during the I/O operations. The 16-bit DX register is used as 16-bit I/O address pointer to address upto 64 K devices in in-direct addressing mode. The I/O instructions with direct addressing mode can directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space. See Fig. 5.4.

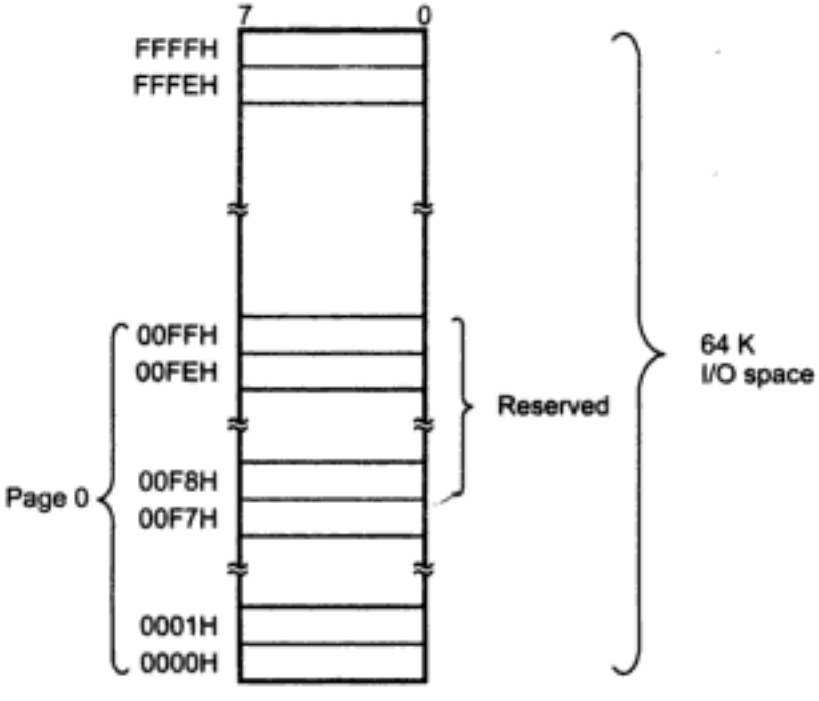


Fig. 5.4 I/O map for 8086

I/O ports are addressed in the same manner as memory locations. Even addressed bytes are transferred on the D_7-D_0 bus lines and odd addressed bytes on D_{15}-D_8. Care must be taken to assure that each register within an 8-bit peripheral located on the lower portion of the bus be addressed as even. In the I/O space, Intel has reserved 00F8H to 00FF locations.

5.5 General 8086 System Bus Structure and Operation

The 8086 has a common address and data bus. The address and data are time multiplexed, i.e. address and data appear on common bus at different time intervals. Thus bus is commonly known as multiplexed address and data bus. The multiplexed address and data bus provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This multiplexed address and data bus has to be demultiplexed externally with the use of latches and the ALE signal provided by 8086, as shown in Fig. 5.5.

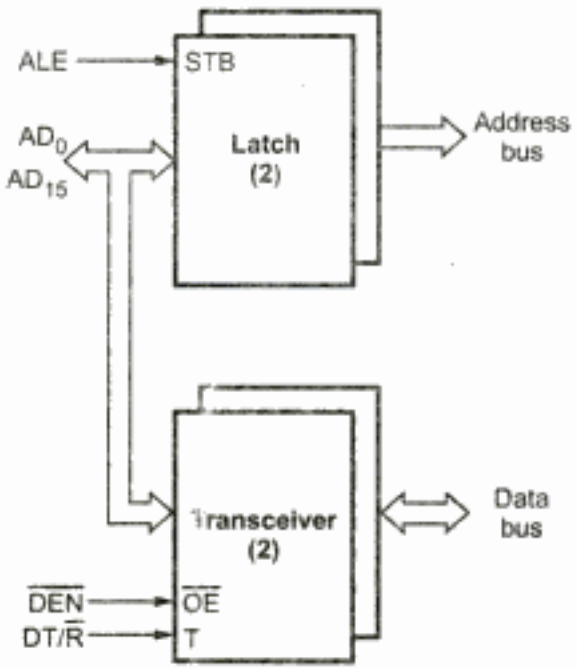


Fig. 5.5 Demultiplexing of address and data bus

Each processor bus cycle consists of at least four clock cycles. These are referred to as T_1, T_2, T_3 and T_4. Refer Fig. 5.5. During T_1, processor sends address on the address bus and activates ALE signal. The ALE signal is used activate latches and thus to latch the address. The data transfer occurs on the bus during T_3 and T_4. The time interval T_2 is used primarily for changing the direction of the bus during read operations. Ready signal is sampled during T_3. The slower peripheral devices use this signal to indicate that the device is not ready to send the desired data within specified time. In the event that a "NOT Ready" indication is given by the slower peripheral device, 'WAIT' states (T_w) are inserted between T_2 and T_3 to give enough access time for the slower peripheral device.

Each WAIT state is of the same duration as a clock cycle. During a WAIT state, the signals on the buses remain the same as they were at the start of the WAIT state. If the Ready input is made high during wait state, then after WAIT state the 8086 will go on with the regular T_4 of the machine cycle. However, if the 8086 Ready input is still low at the end of a WAIT state, then the 8086 will insert another WAIT state. The 8086 will continue inserting WAIT states until the Ready input is made high again.

The status bits $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$ are used, in maximum mode, by the bus controller to identify the type of bus transaction according to the table given below.

$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	Machine cycle
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt

$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	Machine cycle
1	0	0	Instruction fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Inactive-Passive

Status bits S_3 through S_7 are multiplexed with high-order address bits and the $\overline{BHE}$ signal. These bits are also demultiplexed using latch and the ALE signal during T_1. Therefore, the status bits S_3 through S_7 are valid during T_2 through T_4. Status bits S_3 and S_4 indicate which segment register was used for this bus cycle in forming the address, according to the following Table.

S_4	S_3	Characteristics
0	0	Alternate Data (extra segment)
0	1	Stack
1	0	Code or None
1	1	Data

Out of remaining status bits, S_5 is a reflection of the interrupt enable bit of the flag register, S_6 is always 0 and S_7 is a spare status bit.

If a system is large enough to need data bus buffers, then the 8086 DT/$\overline{R}$ signal connected to these buffers will set them for input during a read operation or set them for output during a write operation. The 8086 $\overline{DEN}$ signal will enable the buffers at the appropriate time in the machine cycle, as shown in the Fig. 5.6.

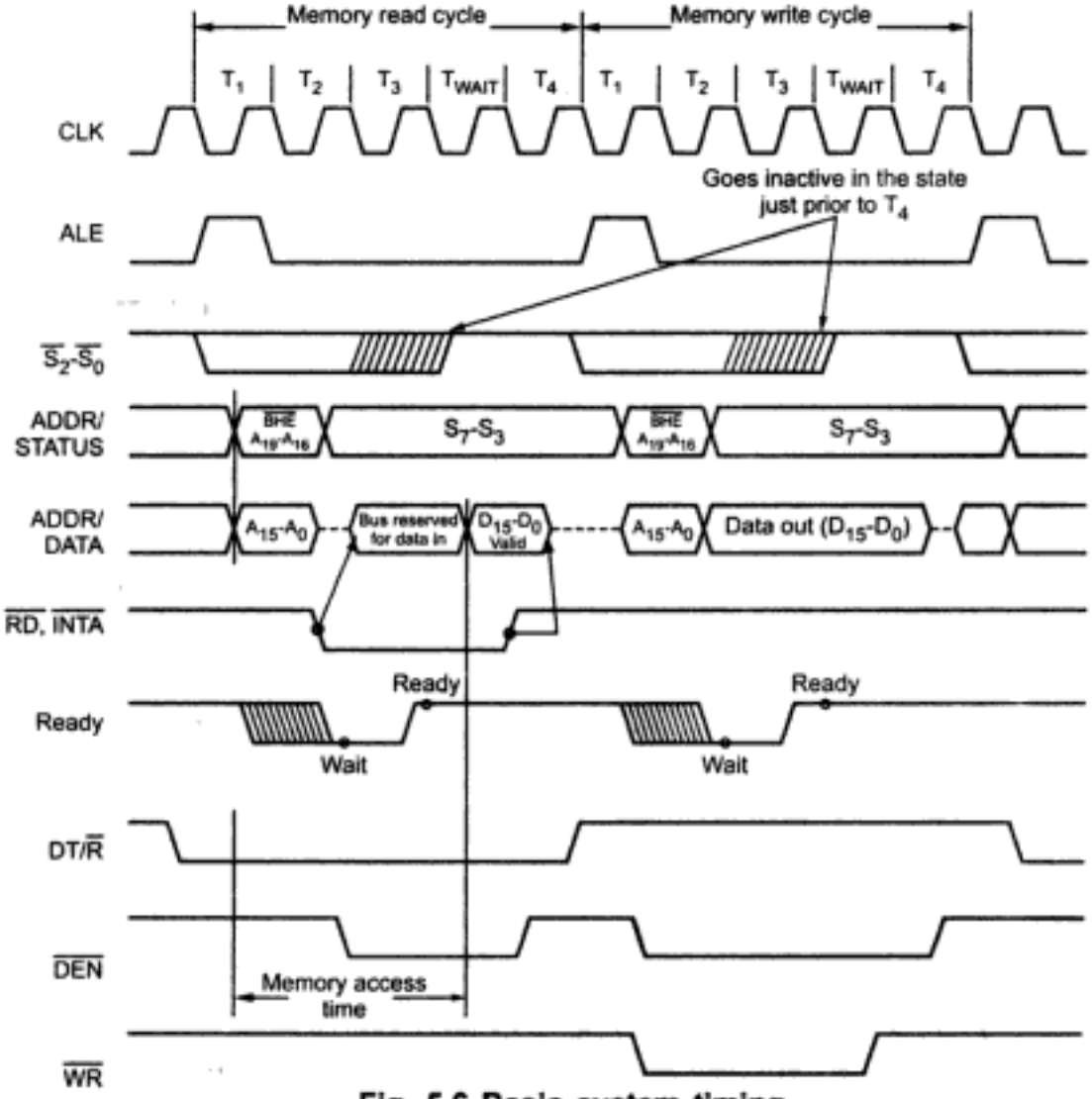


Fig. 5.6 Basic system timing

5.6 Minimum Mode 8086 System and Timings

5.6.1 Minimum Mode Configuration

Latching

Fig. 5.7 shows the typical minimum mode configuration. As shown in the figure, AD_0-AD_{15}, A_{16}/S_3-A_{19}/S_6, and $\overline{BHE}/S_7$ signals are multiplexed. These signals are demultiplexed by external latches and ALE signal generated by the processor. This is accomplished by using three latch ICs (Intel 8282/8283), two of them are required for a 16-bit address and three are needed if a full 20-bit address is used. In case of 8088, only two external latches are required. One for demultiplexing AD_0-AD_7 and other for demultiplexing A_{16}/S_3 and AD_{19}/S_6. Fig. 5.8 shows the internal block diagram of 8282/8283 latches. The 8282 provides noninverting outputs while the 8283 version inverts the input data. In addition to their demultiplexing function, these chips also buffer the address lines, providing increased output driving capability. The output low level is specified as 0.45 V maximum with a sink current of 32 mA maximum. The high level is specified as 2.4 V minimum while supplying a 5 mA maximum high level load current.

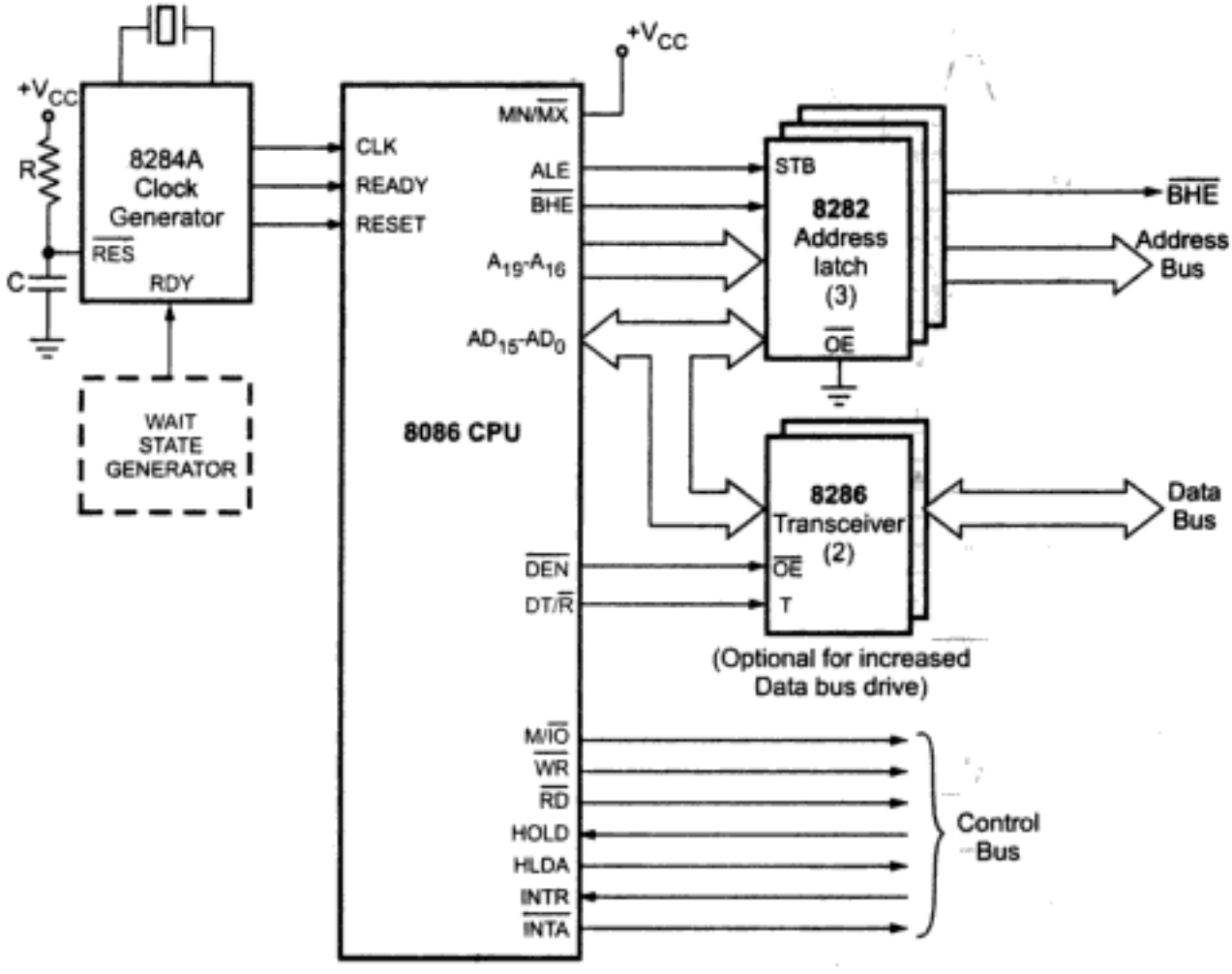


Fig. 5.7 Typical minimum mode configuration

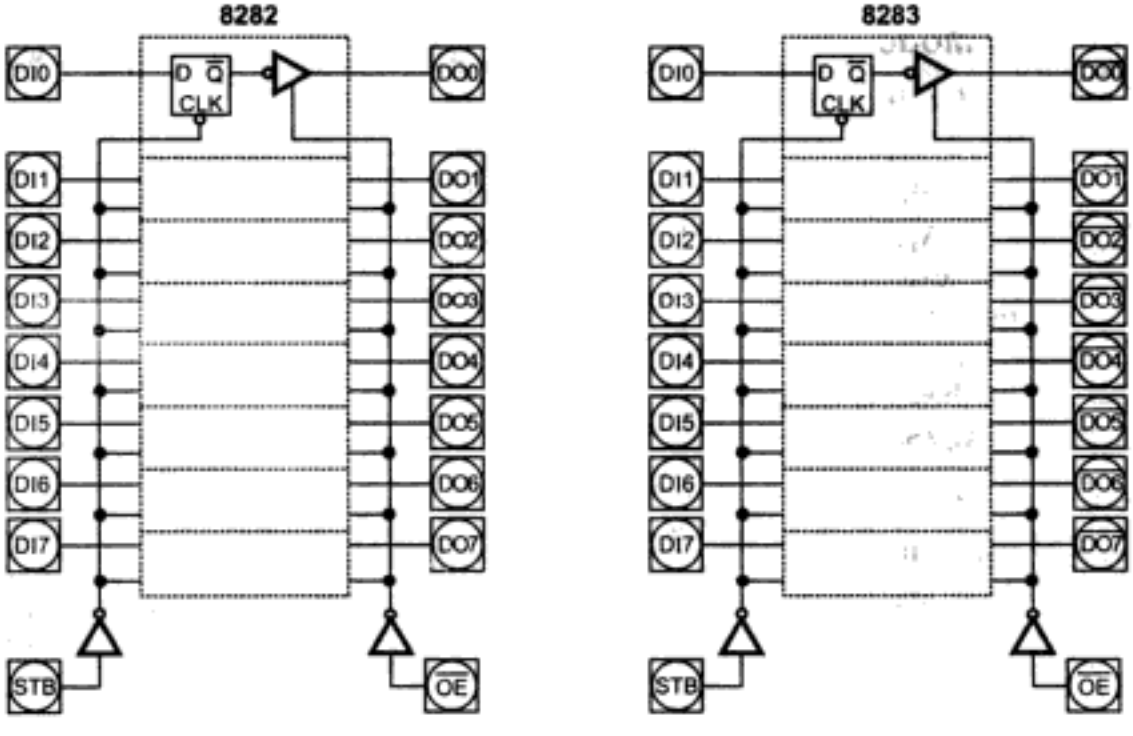


Fig. 5.8 Internal diagram of 8282 and 8283

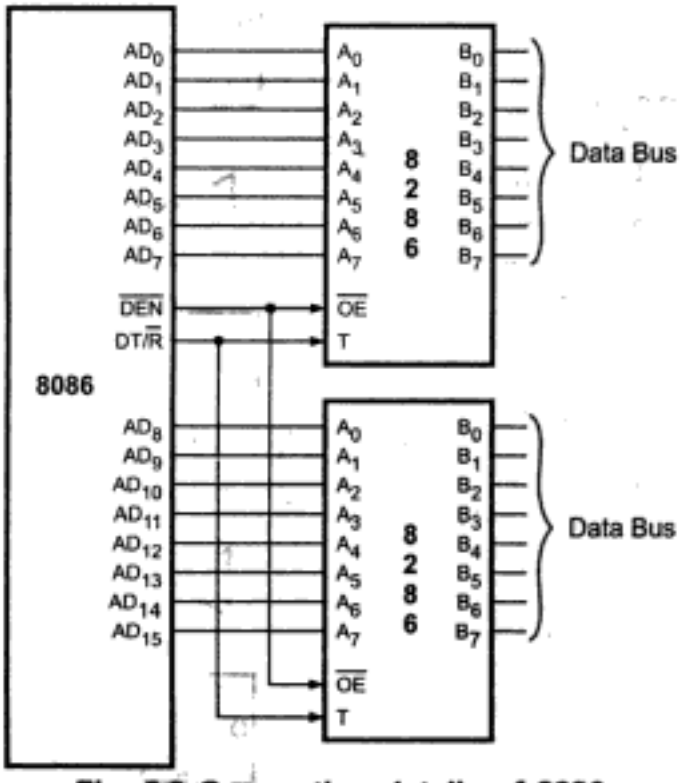


Fig. 5.9 Connection details of 8286

Buffering

If a system includes several interfaces then to increase current sourcing/sinking capacities it is necessary to use drivers and receivers (transceiver) for data bus also. The Intel 8286 device is used to implement the transceiver block shown in Fig. 5.7 The 8286 contains 16 tristate elements, eight receivers, and eight drivers. Therefore two 8286s are required to service 16 data lines of 8086. Fig. 5.9 shows the detailed connections of 8286.

$DT/\overline{R}$ signal is connected to the T input, which controls the direction of the data flow. When this signal is low, receivers are enabled, so that 8086 can read data from memory or input device. To write data into memory or output device, the 8086's $DT/\overline{R}$ signal goes high. Due to this drivers are enabled to transfer data from 8086 to the memory or the output device. At the time of data transfer, to enable output of transceiver its $\overline{OE}$ should be low. This is accomplished by connecting $\overline{DEN}$ signal of 8086 to the $\overline{OE}$ pin of 8286, since $\overline{DEN}$ signal goes low when CPU is ready to send or receive data.

Clock generator

The third component, other than the processor that appears in Fig. 5.7 is an 8284 clock generator. The 8284 clock generator does the following functions :

- Clock generation
- RESET synchronization
- READY synchronization
- Peripheral clock generation.

The Fig. 5.10 shows the internal logic diagram of 8284.

The top half of the logic diagram represents the clock and reset synchronization section of the 8284 clock generator. As shown in the logic diagram, the crystal oscillator has two inputs : X_1 and X_2. If a crystal is attached to X_1 and X_2, the oscillator generates a square-wave signal at the same frequency as the crystal. The output of oscillator is fed to an AND gate and also to an inverter buffer that provides the oscillator output signal. The $F/\overline{C}$ signal selects one of the oscillator inputs. When $F/\overline{C}$ input is 1, the EFI input

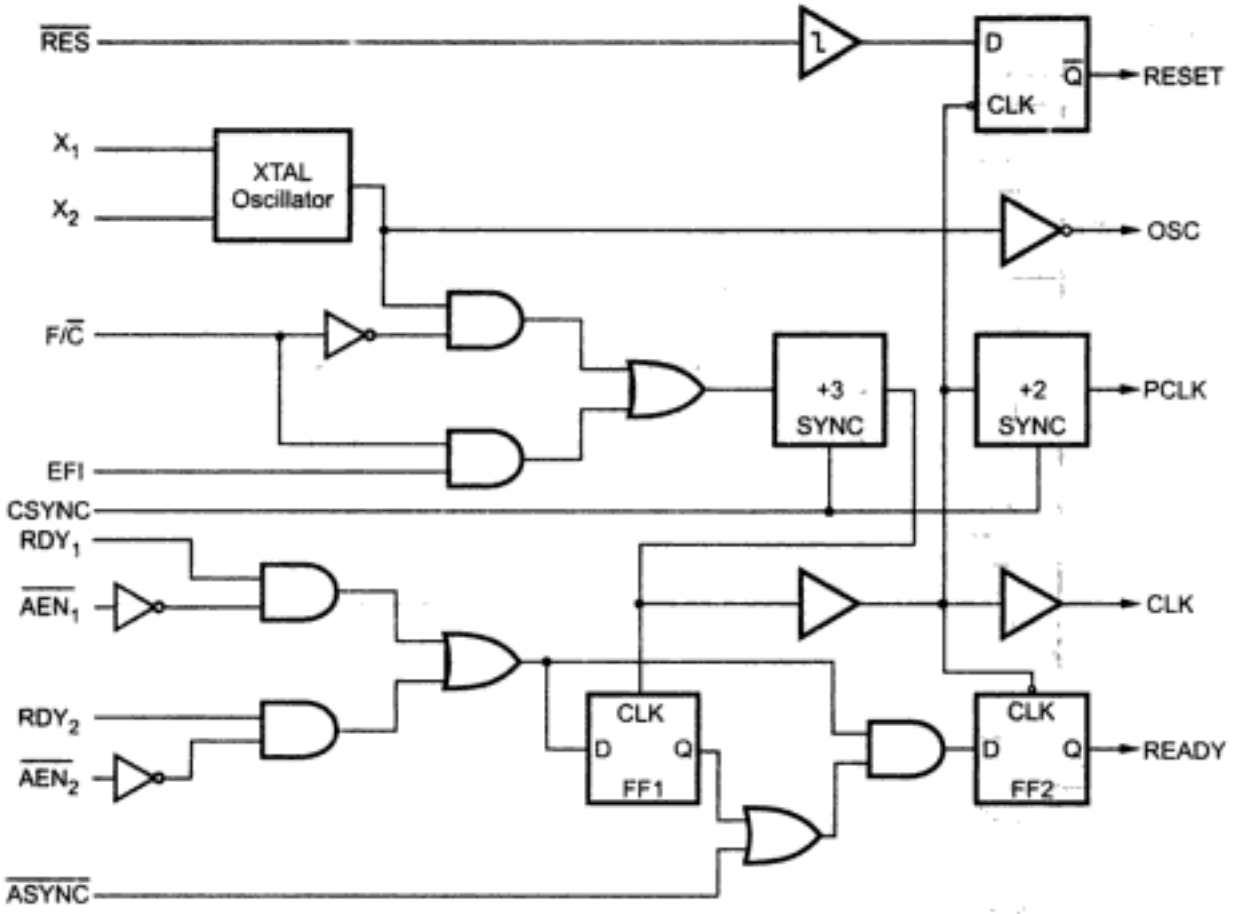


Fig. 5.10 The internal logic diagram of 8284

determines the frequency; otherwise oscillator determines the frequency. When EFI input is used, CSYNC signal is used for multiple processor system synchronization. If the internal crystal oscillator is used, CSYNC signal is grounded. In both the cases the output clock frequency is one third of the input frequency. The CLK signal is also buffered before it leaves the clock generator. As shown in the Fig. 5.10, the output of the divide-by-3 counter generates the timing for ready synchronization, a signal for another counter (divide-by-2), and the CLK signal to the 8086/8088 microprocessors. The two cascaded counters (divide-by-3 and divide-by-2) provide the divide-by-6 output at PCLK, which can be used to provide clock input for peripherals. The address enable pins, $\overline{AEN_1}$ and $\overline{AEN_2}$ are provided to qualify the bus ready signals, RDY_1 and RDY_2, respectively.

The reset circuit of 8284 consists of a schmitt trigger buffer and a single D flip-flop circuit. The D flip-flop ensures that the timing requirements of the 8086/8088 RESET input are met. This circuit applies the RESET signal to the microprocessor on the negative edge (1 to 0 transition) of each clock. The 8086/8088 microprocessors sample RESET at the positive edge (0 to 1 transition) of the clocks; therefore, this circuit meets the timing requirements of the 8086/8088.

The Fig. 5.11 shows the circuit connection for 8284 clock generator. The RC circuit provides a logic 0 to the $\overline{RES}$ input pin when power is first applied to the system. After a short time, the $\overline{RES}$ input becomes a logic 1 because the capacitor charges toward + 5.0 V through the register. A push button switch allows the microprocessor to be reset by the operator.

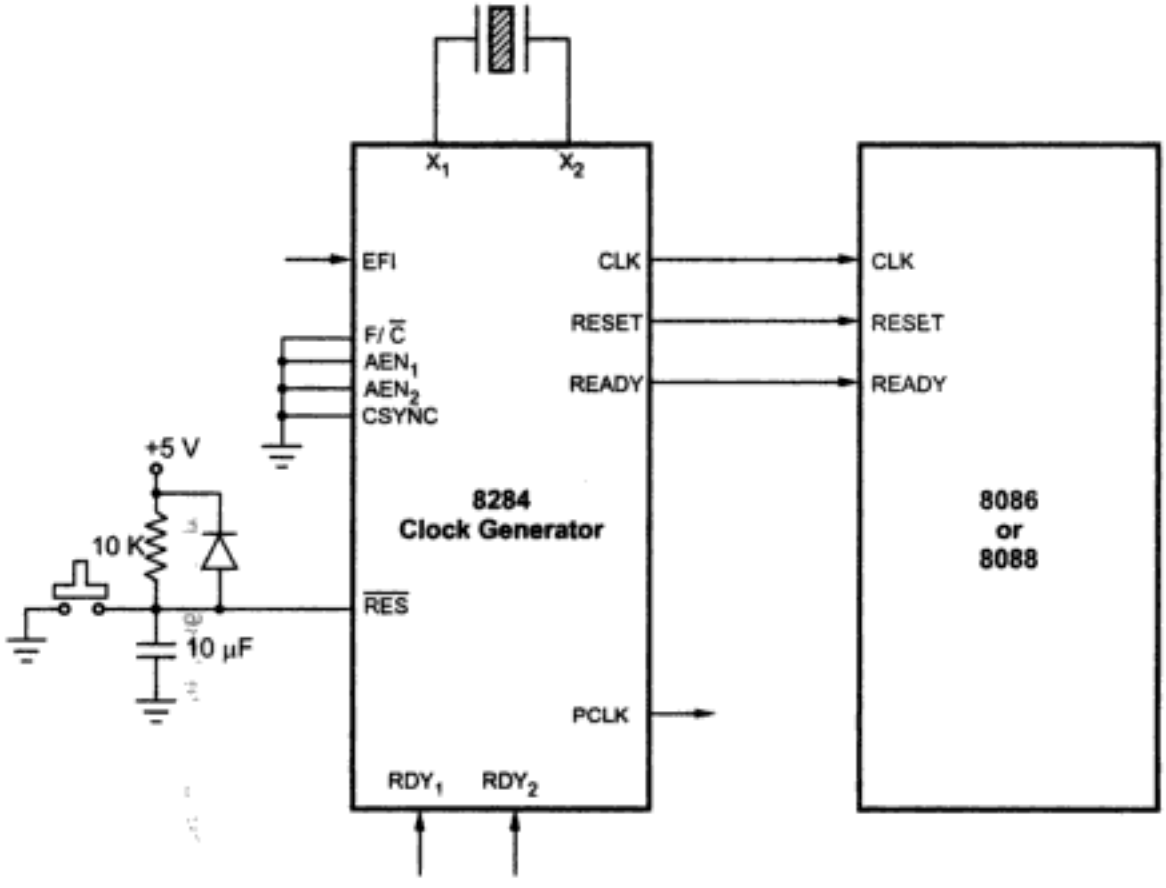


Fig. 5.11 Interfacing of 8284 clock generator with 8086 or 8088

Other signals

The status on the M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ lines decides the type of data transfer, as listed in the Table 5.1.

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

Table 5.1

HOLD and HLDA signals are used to interface other bus masters like DMA controller. Interrupt request (INTR) and interrupt acknowledge ($\overline{INTA}$) are used to extend the interrupt handling capacity of the 8086 with the help of interrupt controller.

5.6.2 Minimum Mode 8086 System

The Fig. 5.12 shows the typical minimum mode 8086 system. Here, interfacing of memory and I/O devices are shown with the basic minimum mode 8086 configuration.

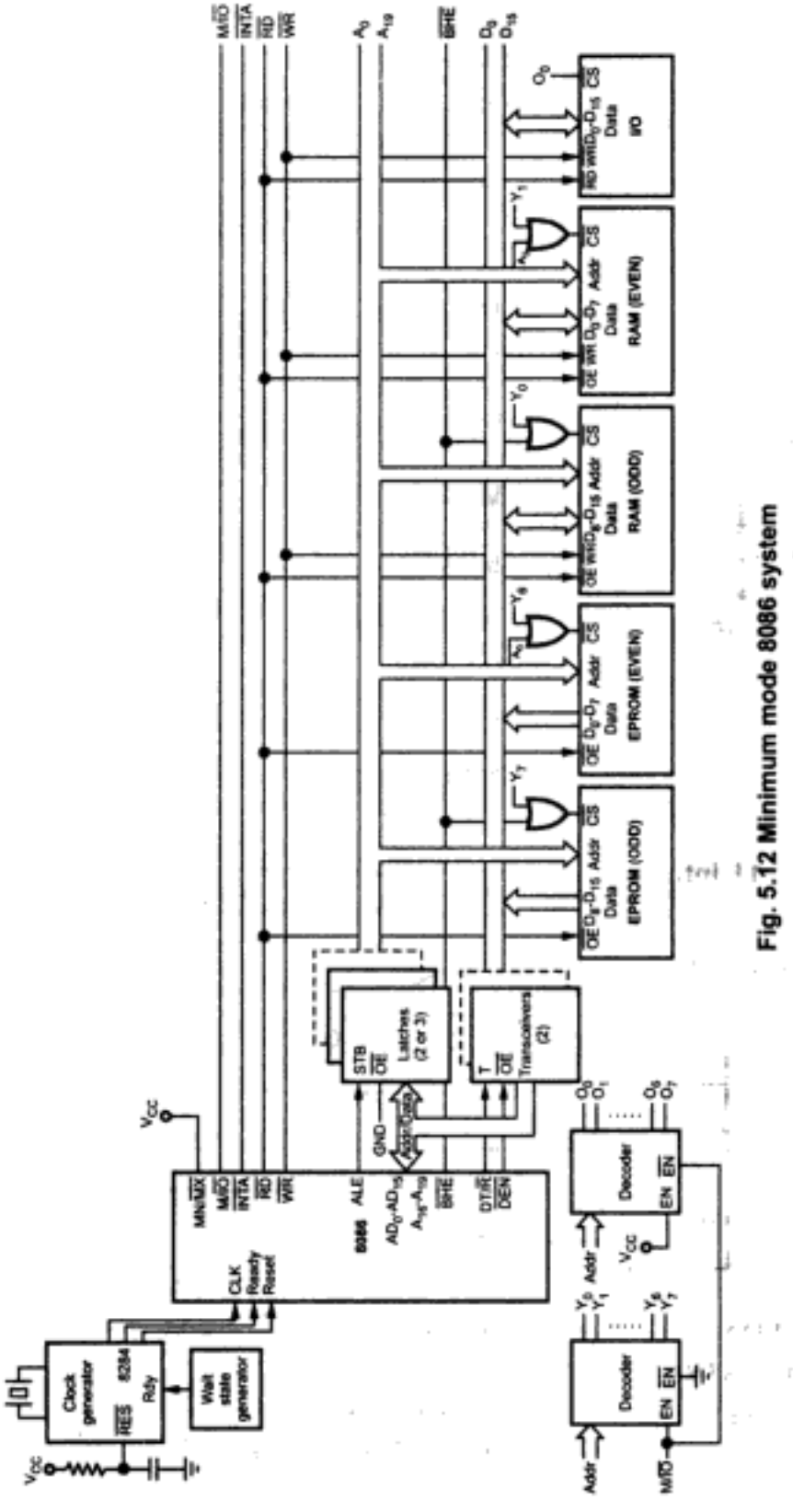


Fig. 5.12 Minimum mode 8086 system

For interfacing memory module to 8086, it is necessary to have odd and even memory banks. This is implemented by using two EPROMs and two RAMs. Data lines D_{15}-D_8 are connected to odd bank of EPROM and RAM, and data lines D_7-D_0 are connected to even bank of EPROM and RAM. Address lines are connected to EPROM and RAM as per their capacities. $\overline{RD}$ signal is connected to the output enable ($\overline{OE}$) signals of EPROMs and RAMs. $\overline{WR}$ signal is connected to $\overline{WR}$ signal of RAMs. Two separate decoders are used to generate chip select signals for memory and I/O devices. These chip select signals are logically ORed with either $\overline{BHE}$ or A_0 to generate final chip select signals. For generating final chip select signals for odd bank decoder outputs are logically ORed with $\overline{BHE}$ signal. On the other hand to generate final chip select signals for even bank decoder outputs are logically ORed with A_0 signal.

The 16-bit I/O interface is shown in the Fig. 5.12. $\overline{RD}$ and $\overline{WR}$ signals are connected to the $\overline{RD}$ and $\overline{WR}$ signals of I/O device. Data lines D_{15}-D_0 are connected to the data lines of I/O device. The chip select signal for I/O device is generated using separate decoder whose output is enabled only when M/$\overline{IO}$ signal is low.

5.6.3 Bus Timings for Minimum Mode

5.6.3.1 Timings for Read and Write Operations

The timing diagrams of input and output transfers for 8086 minimum mode are shown in the Fig. 5.13 (a) and (b) respectively.

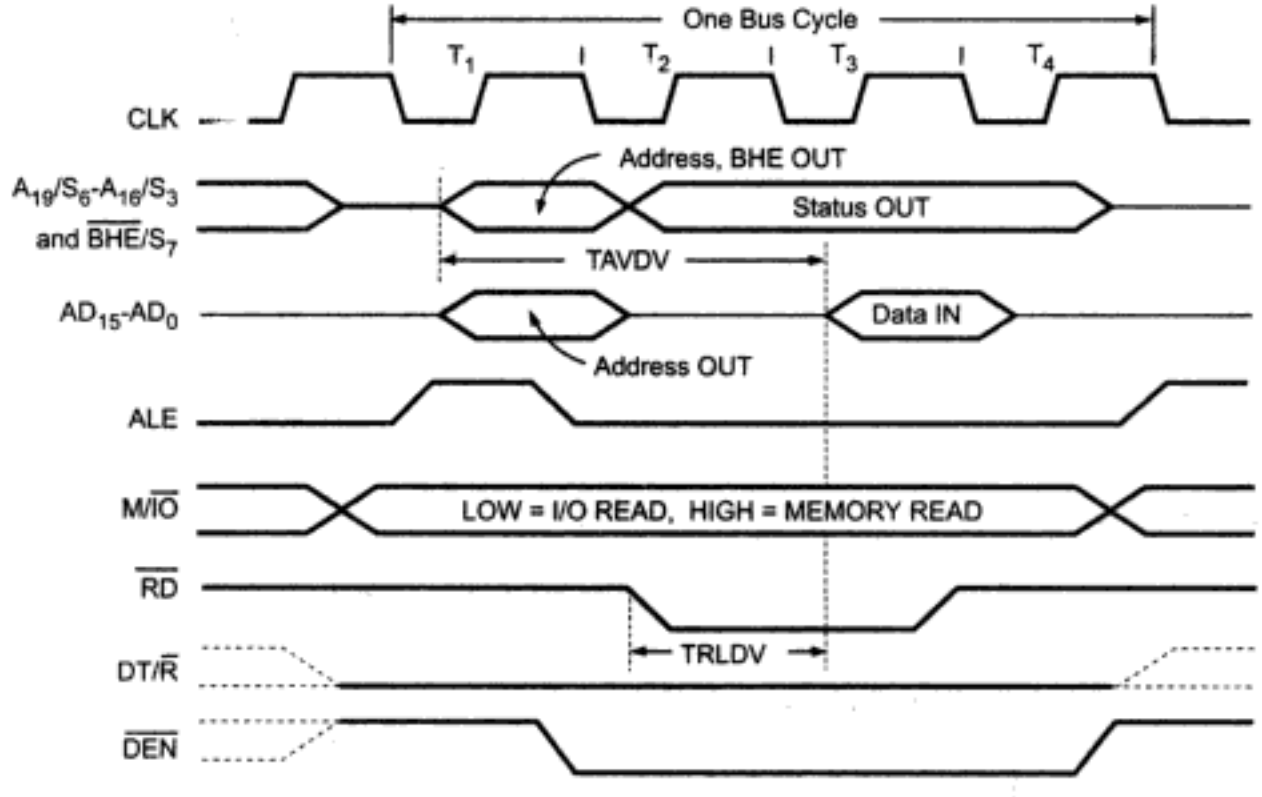


Fig. 5.13 (a) Input (read operation)

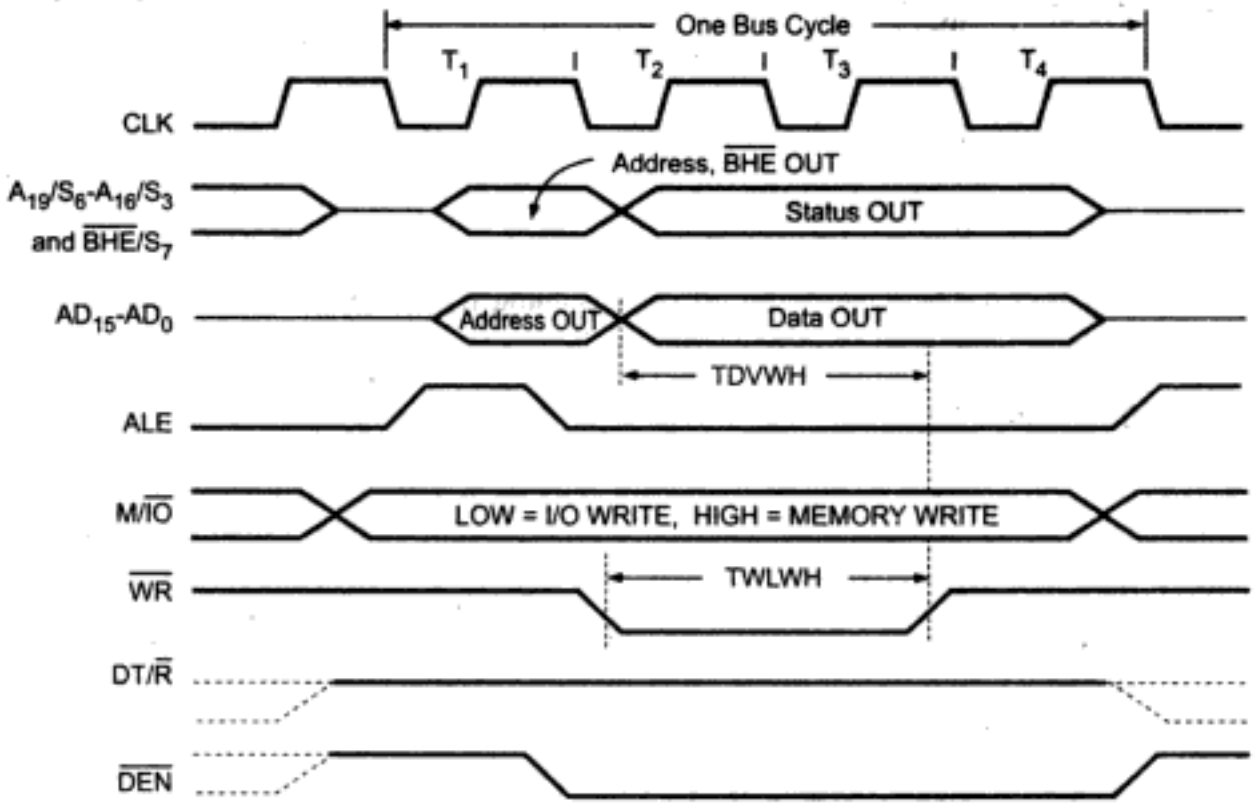


Fig. 5.13 (b) Output (write operation)

These are explained in steps.

1. When processor is ready to initiate the bus cycle, it applies a pulse to ALE during T_1. Before the falling edge of ALE, the address, $\overline{BHE}$, M/$\overline{IO}$, $\overline{DEN}$ and DT/$\overline{R}$ must be stable i.e. DEN = high and DT/$\overline{R}$ = 0 for input or DT/$\overline{R}$ = 1 for output.
2. At the trailing edge of ALE, ICs 74LS373 or 8282 latches the address.
3. During T_2 the address signals are disabled and S_3-S_7 are available on AD_{16}/S_3-AD_{19}/S_6 and $\overline{BHE}/S_7$. Also $\overline{DEN}$ is lowered to enable transceiver.
4. In case of input operation, $\overline{RD}$ is activated during T_2 and AD_0 to AD_{15} go in high impedance preparing for input.
5. If memory or I/O interface can perform the transfer immediately; there are no wait states and data is output on the bus during T_3.
6. After the data is accepted by the processor, $\overline{RD}$ is raised high at the beginning of T_4.
7. Upon detecting this transition during T_4, the memory or I/O device will disable its data signals.
8. For an output operation, processor applies $\overline{WR}$ = 0 and then the data on the data bus during T_2.
9. In T_4, $\overline{WR}$ is raised high and data signals are disabled.

Hidden page

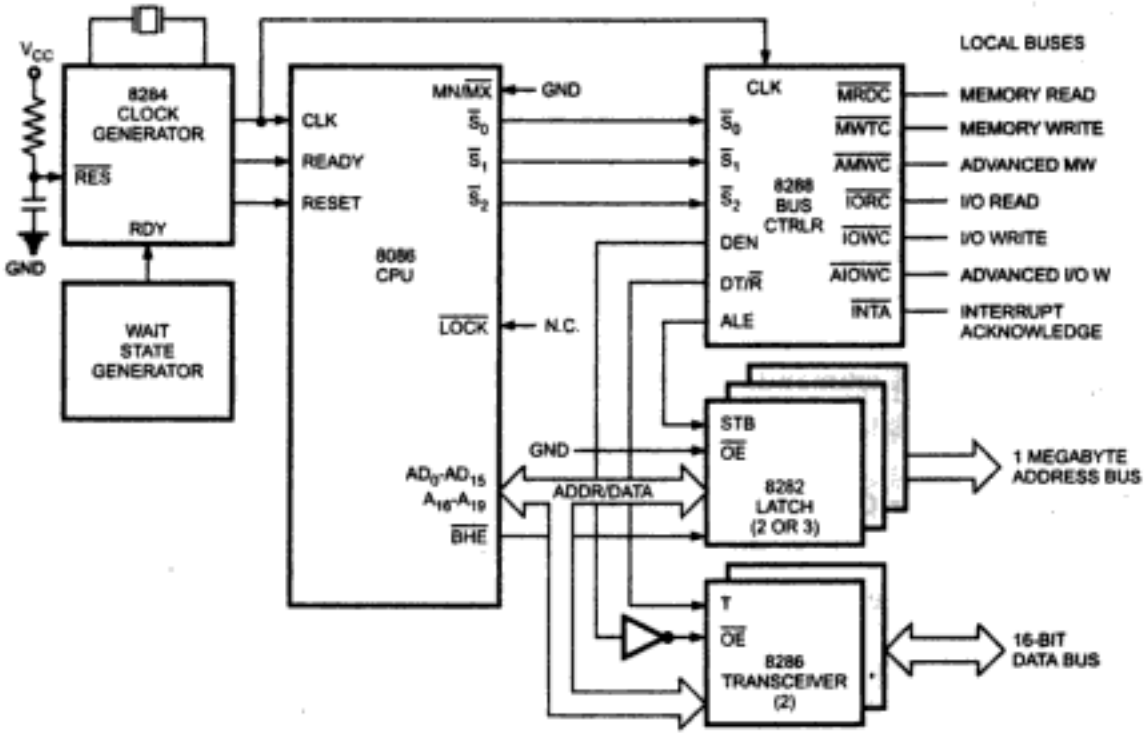


Fig. 5.15 Typical maximum mode configuration

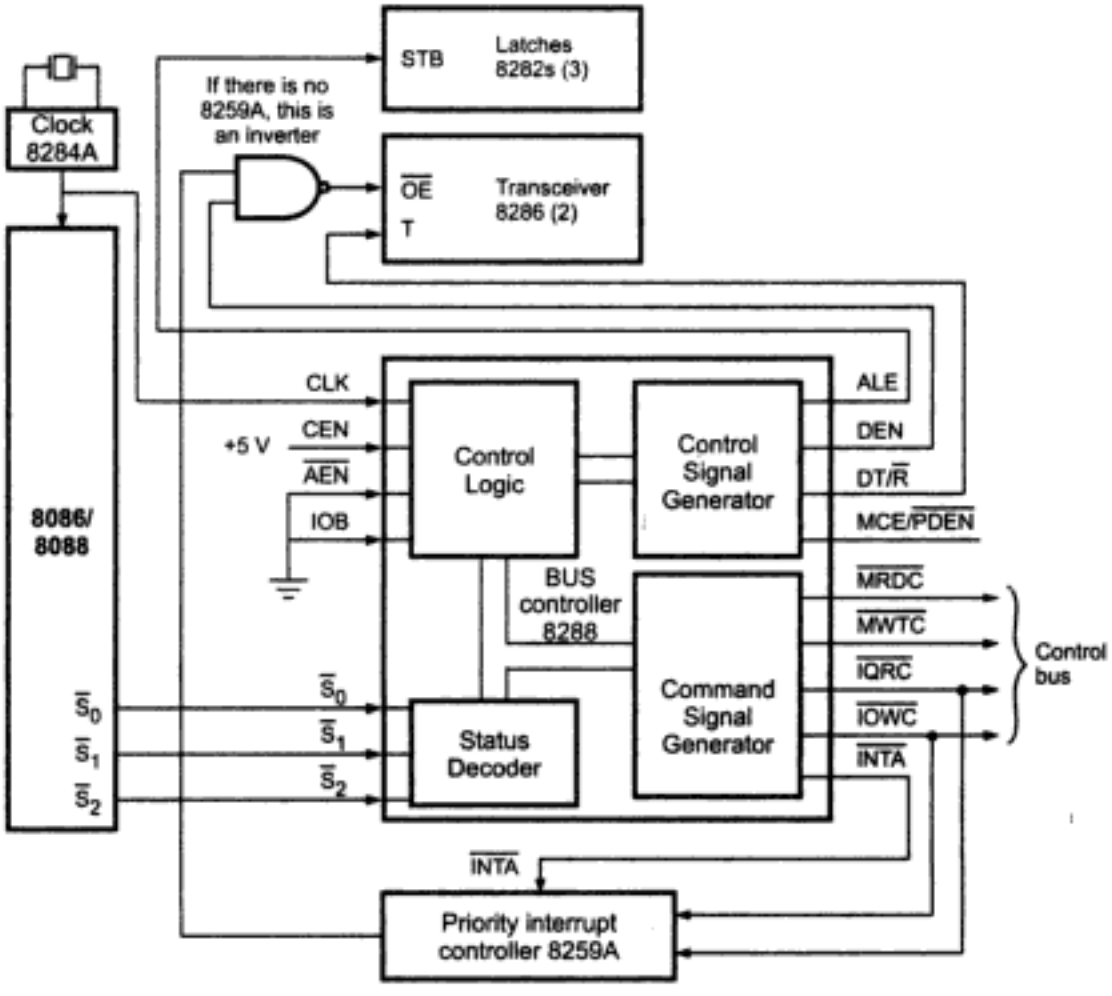


Fig. 5.16 8288 bus controller

Hidden page

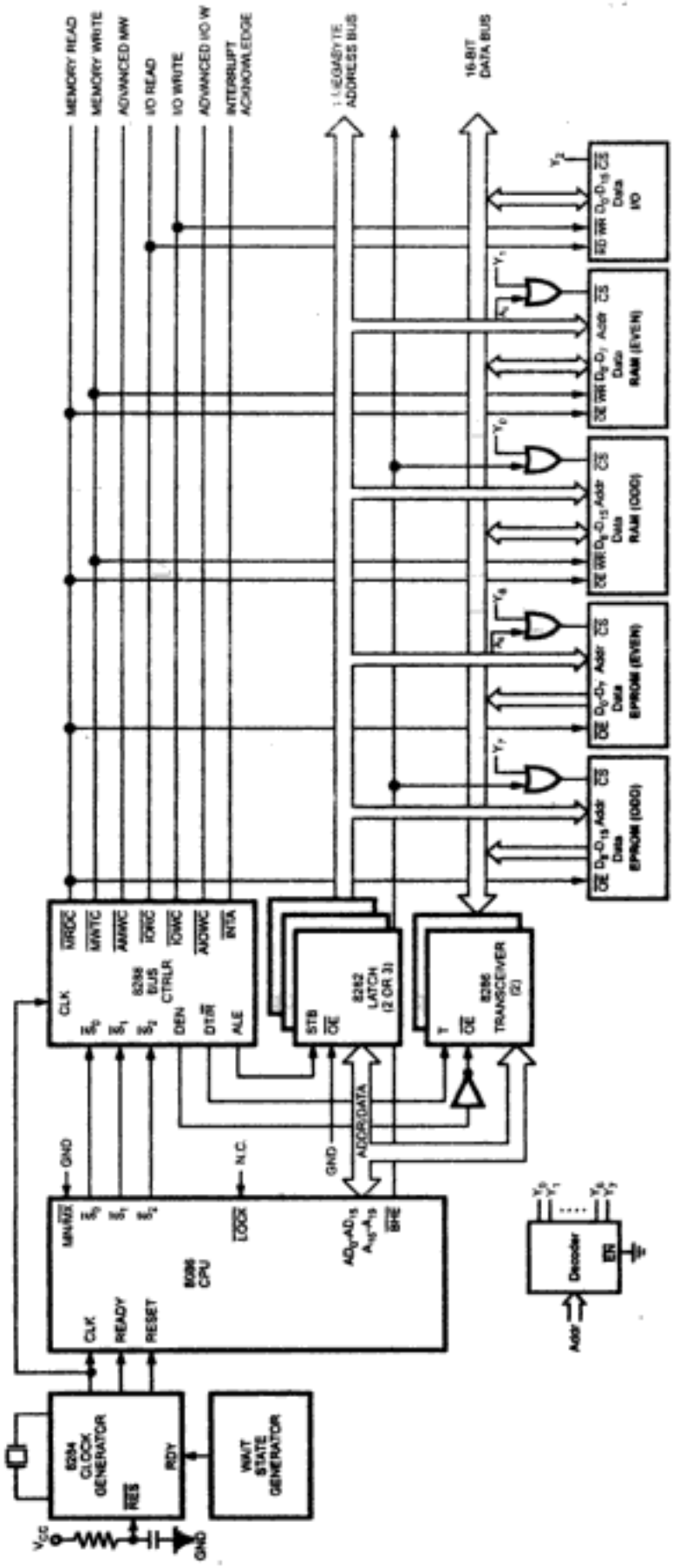


Fig. 5.17 Typical maximum mode 8086 system

5.7.3 Bus Timings for Maximum Mode

5.7.3.1 Timings for Read and Write Operations

The timing diagrams of input and output transfers are shown in the Fig. 5.18 (a) and (b) respectively.

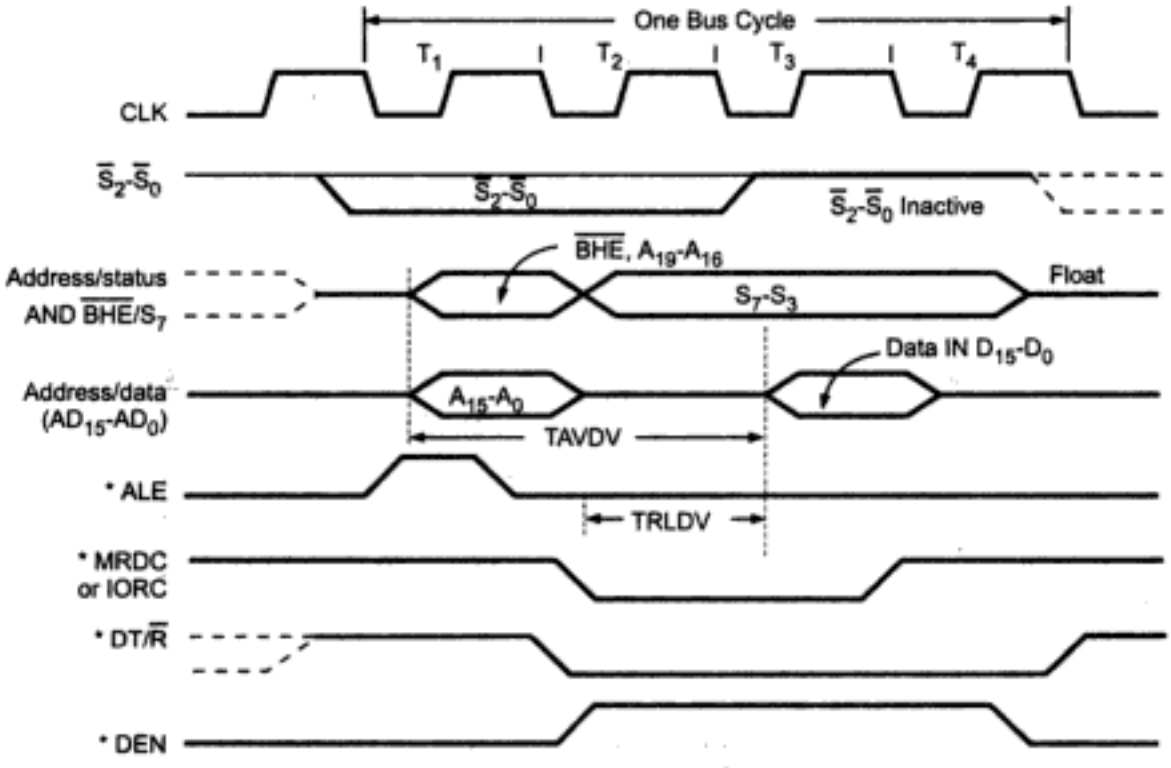


Fig. 5.18 (a) Input (read operation)

These are explained in steps.

1. S_0, $\overline{S}_1$, $\overline{S}_2$ are set at the beginning of bus cycle. On detecting the change on passive state $\overline{S}_0 = \overline{S}_1 = \overline{S}_2 = 1$, the 8288 bus controller will output a pulse on its ALE and apply a required signal to its DT/$\overline{R}$ pin during T_1.
2. In T_2, 8288 will set DEN = 1 thus enabling transceiver. For an input, 8288 it will activates $\overline{MRDC}$ or $\overline{IORC}$. These signals are activated until T_4. For an output, the $\overline{AMWC}$ or $\overline{AIOWC}$ is activated from T_2 to T_4 and $\overline{MWTC}$ or $\overline{IOWC}$ is activated from T_3 to T_4.
3. The status bits $\overline{S}_0$ to $\overline{S}_2$ remain active until T_3, and become passive during T_3 and T_4.
4. If ready input is not activated before T_3, wait state will be inserted between T_3 and T_4.

Hidden page

Hidden page

Word length = M bit

= 8 bit

➠ **Example 2 :** If memory has 8192 memory locations, then it has 13 address lines.

The Table 5.2 summarizes the memory capacity and address lines required for memory interfacing.

Memory Capacity	Address Lines Required
1 K = 1024 memory locations	10
2 K = 2048 memory locations	11
4 K = 4096 memory locations	12
8 K = 8192 memory locations	13
16 K = 16384 memory locations	14
32 K = 32768 memory locations	15
64 K = 65536 memory locations	16

Table 5.2

As shown in the Fig. 5.20 (a) memory chip has 11 address lines A_0-A_{10}, one chip select ($\overline{CS}$), and two control lines, read ($\overline{RD}$) to enable output buffer and write ($\overline{WR}$) to enable the input buffer. The internal decoder is used to decode the address lines. Fig. 5.20 (b) shows the logic diagram of a typical EPROM (Erasable Programmable Read-Only Memory) with 4096 (4 K) registers. It has 12 address lines A_0-A_{11}, one chip select ($\overline{CS}$), one Read control signal. Since EPROM is a read only memory, it does not require the ($\overline{WR}$) signal.

5.9 Basic Concepts in Memory Interfacing

For interfacing memory devices to microprocessor 8086 following important points are to be kept in mind.

1. Microprocessor 8086 can access 1 Mbytes memory since address bus is 20-bit. But it is not always necessary to use full 1 Mbytes address space. The total memory size depends upon the application.
2. Generally EPROM (or EPROMs) is used as a program memory and RAM (or RAMs) as a data memory. When both, EPROM and RAM are used, the total address space 1Mbytes is shared by them.
3. The individual capacities of program memory and data memory depend on the application.
4. It is not always necessary to select 1 EPROM and 1 RAM. We can have multiple EPROMs and multiple RAMs as per the requirement of application.

Hidden page

memory interface with absolute decoding. Two 8 K EPROMs (2764) are used to provide even and odd memory banks. Control signals $\overline{BHE}$ and A_0 are used to enable outputs of odd and even memory banks respectively. As each memory chip has 8 K memory locations, thirteen address lines are required to address each locations, independently. All remaining address lines are used to generate an unique chip select signal. This addressing technique is normally used in large memory systems.

2) Linear Decoding :

In small systems, hardware for the decoding logic can be eliminated by using only required number of addressing lines (not all). Other lines are simply ignored. This technique is referred as **linear decoding** or **partial decoding**. Fig 5.22 shows the addressing of 16 K RAM (6264) with linear decoding. Control signals $\overline{BHE}$ and A_0 are used to enable odd and even memory banks, respectively. The address line A_{19} is used to select the RAM chips. When A_{19} is low, chip is selected, otherwise it is disabled. The status of A_{14} to A_{18} does not affect the chip selection logic. This gives you multiple addresses (shadow addresses). This technique reduces the cost of decoding circuit, but it has drawback of multiple addresses.

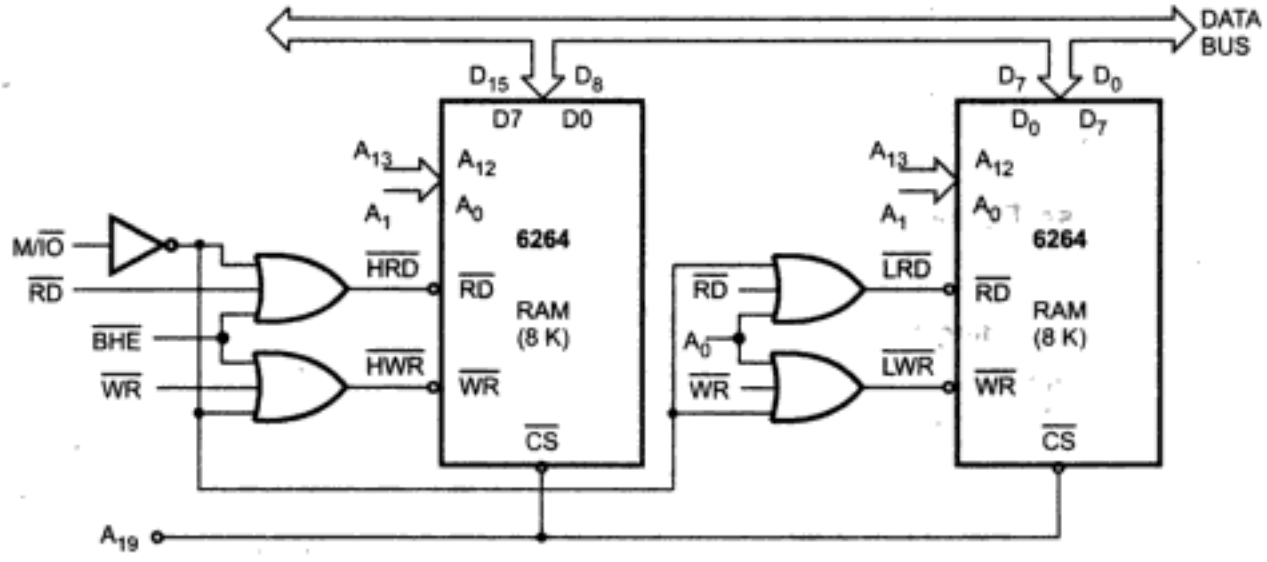


Fig. 5.22 Linear decoding

3) Block Decoding :

In a microcomputer system the memory array is often consists of several blocks of memory chips. Each block of memory requires decoding circuit. To avoid separate decoding for each memory block special decoder IC is used to generate chip select signal for each block. Fig. 5.23 shows the block decoding technique using 74138, 3:8 decoder.

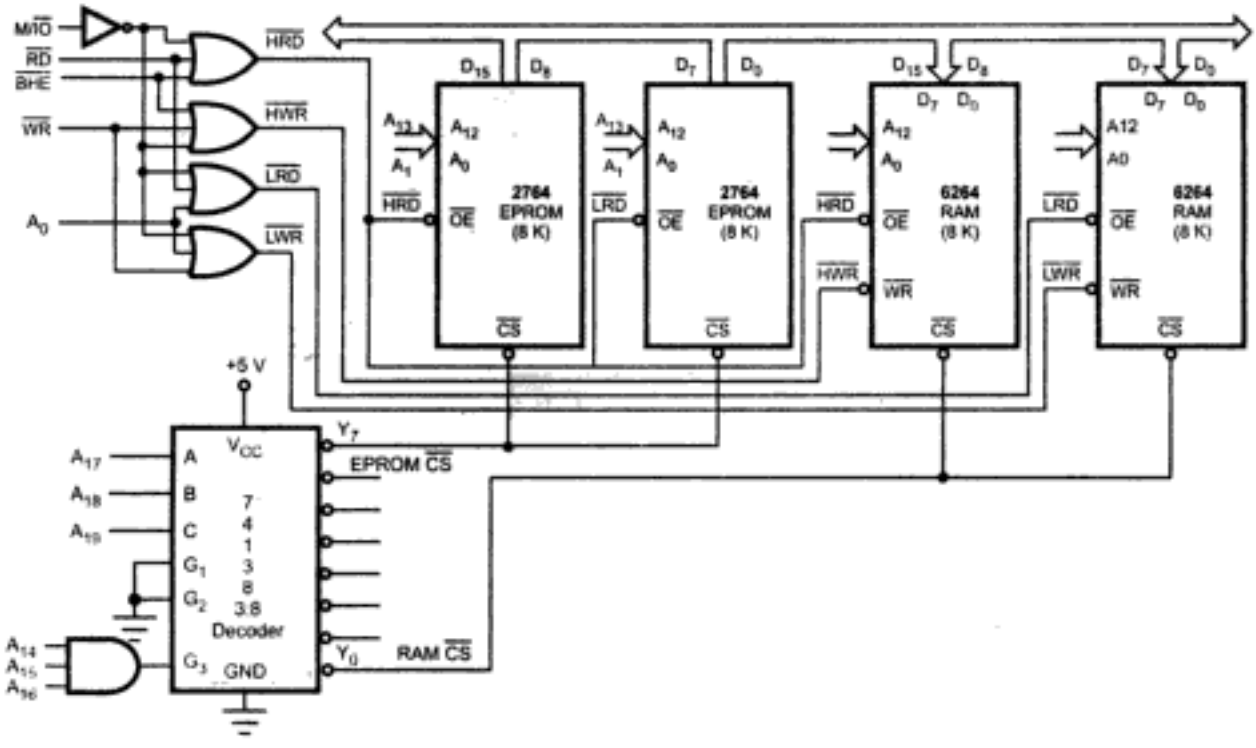


Fig. 5.23 Block decoding

5.10 Interfacing Examples

Example 1 : *Design an 8086 based system with the following specifications.*
i) 8086 in minimum mode.
ii) 64 KByte EPROM
iii) 64 KByte RAM
Draw the complete schematic of the design indicating address map.

Solution : The 8086 is a 16 bit microprocessor. It can access 16 bit data simultaneously. For interfacing memory module to 8086 CPU, it is necessary to have odd and even memory banks. This can be achieved by using two 32 Kbyte EPROMs and two 32 byte RAMs, one for odd bank and another for even bank.

As 32 Kbyte RAM and EPROM need 15 address lines, A_1 to A_{15} lines are used. A_0 and $\overline{BHE}$ are used to select even and odd memory banks respectively. Fig. 5.24 shows the interface between 8086 and two memory chips.

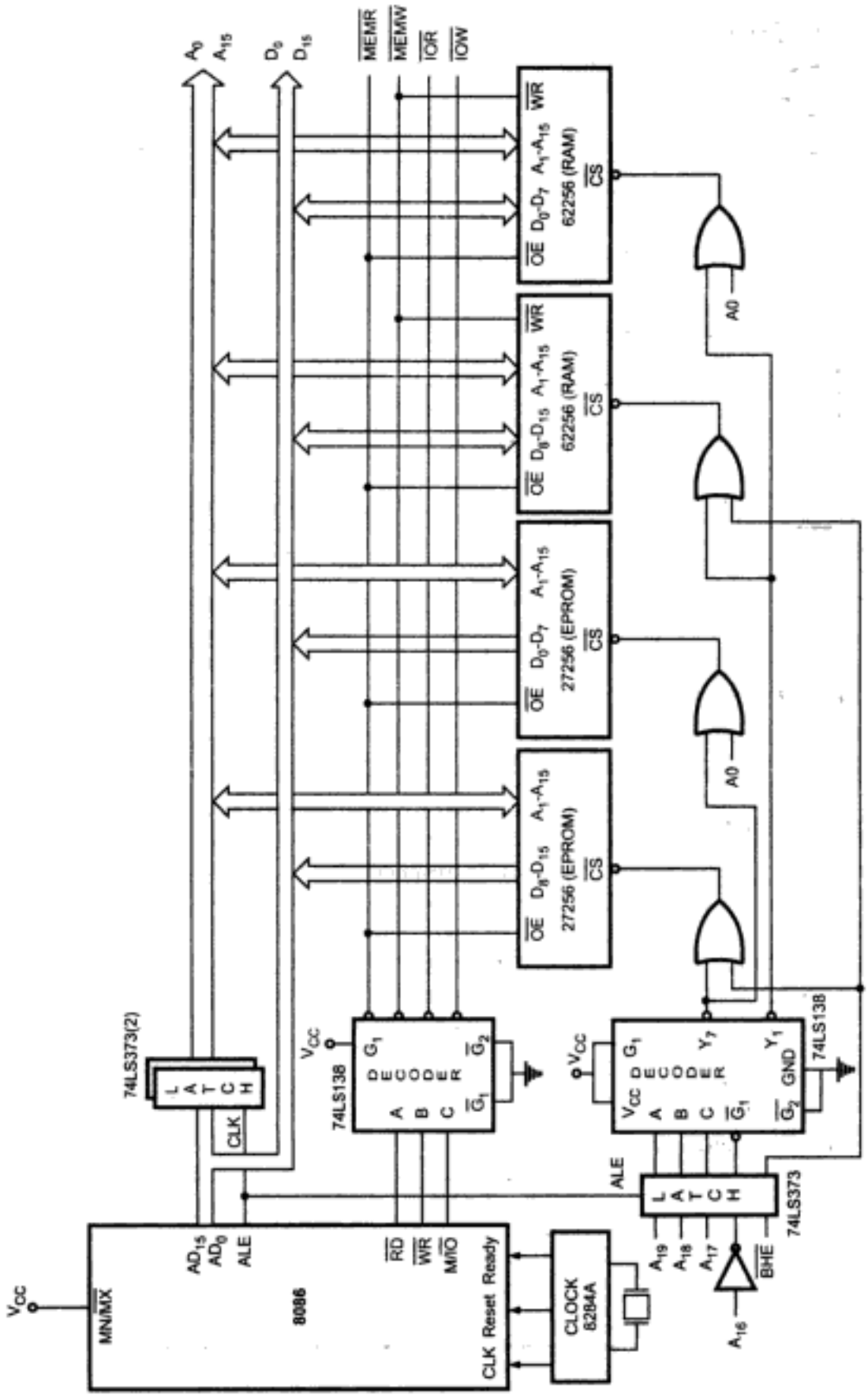


Fig. 5.24 Interfacing 64 K RAM and 64 K EPROM with 8086 in minimum mode

Memory Map :

$\overline{BHE}$	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address	Memory
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F0000H	Even
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFFH	EPROM1
0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F0001H	Odd
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFHH	EPROM2
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30000H	Even
1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	3FFFEH	RAM1
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	30001H	Odd
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFFH	RAM2

Example 2 : *Design an 8086 based system with the following specifications.*

i) 8086 in maximum mode ii) 64 KByte EPROM iii) 64 KByte RAM

Draw the complete schematic of the design indicating address map.

Solution : The 8086 is a 16 bit microprocessor. It can access 16 bit data simultaneously. For interfacing memory module to 8086 CPU, it is necessary to have odd and even memory banks. This can be achieved by using two 32 Kbyte EPROMs and two 32 Kbyte RAMs, one for odd bank and another for even bank.

In the maximum mode, memory and I/O read/write, address latch enable (ALE), Data Enable (DEN), Data transmit/receive (DT/$\overline{R}$) signals must be decoded externally using bus controller 8288. Fig 5.25 shows the memory interface with 8086 in the maximum mode. As 32 Kbyte RAM and EPROM need 15 address lines, A_1 to A_{15} lines are used. A_0 and $\overline{BHE}$ are used to select even and odd memory banks respectively.

Memory Map :

$\overline{BHE}$	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address	Address
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F0000H	Even
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFEH	EPROM1
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	F0001H	Odd
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH	EPROM2
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30000H	Even
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	3FFFEH	RAM1
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	30001H	Odd
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFFH	RAM2

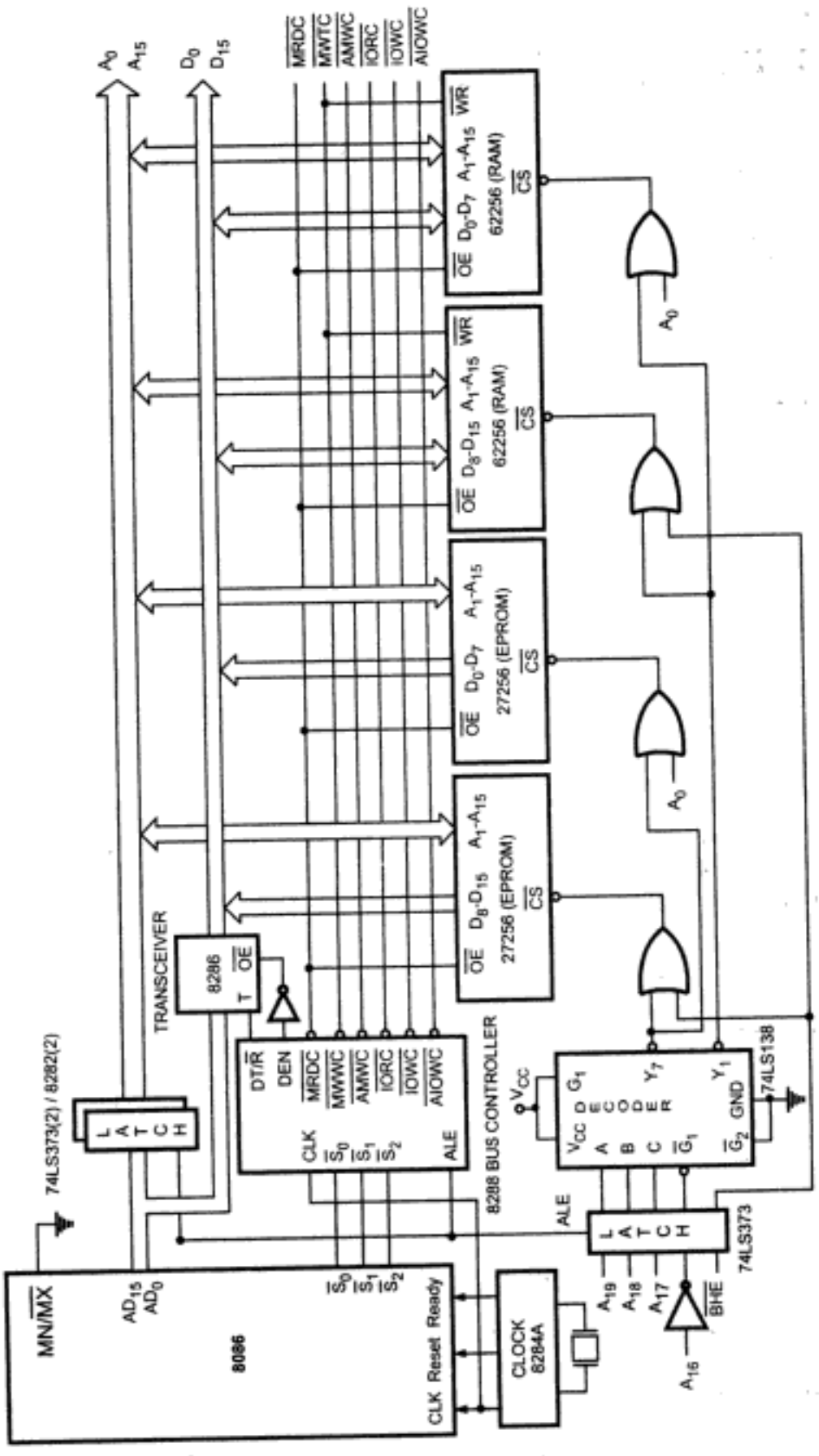


Fig. 5.25 Interfacing 64 K RAM and 64 K EPROM with 8086 in maximum mode

Hidden page

clear input, CLR, of the shift register. The outputs of the shift register will then all be low. One of these lows will be coupled through a jumper (jumper 4 in the Fig. 5.26), will cause the RDY1 input of the 8284 to be pulled low. However, WAIT states will not be inserted unless RDY1 remains low long enough. Now, when $\overline{RD}$, $\overline{WR}$, or $\overline{INTA}$ goes low in the machine cycle, the $\overline{CLR}$ input of the 74LS164 shift register will go high, and the shift register will function normally. The highs on the INA and INB inputs will be loaded on to the Q_A output on the next positive edge of the clock. If the WAIT state jumper is in the J_0 position, then this high on the Q_A output will cause the RDY1 input of the 8284 to go high again. For this case, the RDY1 input goes high soon enough that no WAIT states are inserted.

The high loaded into the 74LS164 shift register is shifted one stage to the right by each successive clock pulse. When the high reaches the jumper connected to the RDY1 input, it will cause the RDY1 input of the 8284 to go high, as shown in the Fig. 5.27. The 8086 will then exit from a WAIT state on the next clock pulse. The number of WAIT states inserted in a machine cycle is determined by how many states the high has to be shifted before it reaches the installed jumper.

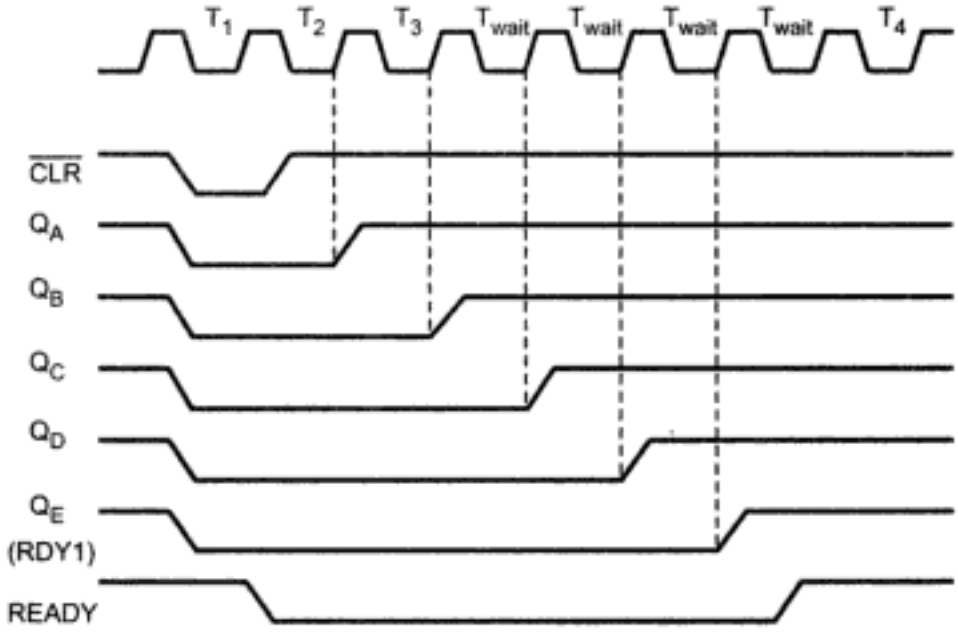


Fig. 5.27 Timing diagram for wait state generator

Review Questions

1. *Explain the function of following pins in 8086.*
   i) NMI ii) MN/$\overline{MX}$ iii) $\overline{TEST}$ iv) $\overline{BHE}$ v) DT/$\overline{R}$ vi) DEN vii) QS_0-QS_1.
2. *Explain the maximum mode signals of 8086.*
3. *Explain the minimum mode signals 8086.*
4. *With the help of block diagram explain memory interfacing with 8086 and explain why two bus cycles are required to access odd address word ?*
5. *Draw and explain the memory map for 8086.*

6. *Explain the I/O addressing capabilities of 8086.*
7. *Draw and explain the I/O map of 8086.*
8. *Explain the general bus operation of 8086 with the help of timing diagram.*
9. *Explain the purpose of Ready, $\overline{DEN}$ and $DT/\overline{R}$ signals.*
10. *With the help of block schematic diagrams explain the operation of 8284 clock generator and 8286 transceiver.*
11. *Sketch block diagram showing basic 8086 minimum mode system. Explain functions of 8282 latches and 8286 transceiver.*
12. *Define bus cycle, and explain the minimum mode read and write bus cycle with proper timing diagram.*
13. *Explain the HOLD response sequence in the minimum mode of 8086 with the help of timing diagram.*
14. *Draw and explain a block diagram showing 8086 in maximum mode configuration.*
15. *Draw and explain the timing diagrams of input and output transfers of 8086 in maximum mode.*
16. *Indicate the signals which are different when 8086 in minimum mode and in maximum mode.*
17. *Explain the operation of bus request and bus grant signal with the help of timing diagram.*
18. *Explain the function of wait state generator.*
19. *Design the wait state generator to insert wait states from zero to seven.*

❑❑❑

6

Direct Memory Access (DMA) - 8237/8257

In microprocessor based systems data transfer can be controlled by either software or hardware. Upto this point we have used program instructions to transfer data from I/O device to memory or from memory to I/O device. To transfer data by this method microprocessor has to do following tasks :

1. To fetch the instruction
2. To decode the instruction and
3. To execute the instruction.

To carryout these tasks microprocessor requires considerable time, so this method of data transfer is not suitable for large data transfers such as data transfer from magnetic disk or optical disk to memory. In such situations hardware controlled data transfer technique is used.

Software Controlled Data Transfer

In this method programmer executes a series of instructions to carry out data transfer. The sample flow chart and program required to transfer data from memory to I/O device is shown in Fig. 6.1. (Refer Fig. 6.1 on next page.)

Program :

Transfer Subroutine

```
        MOV  CX, COUNT          ; Initialize counter
        MOV  DX, PORT_addr      ; Load port address in DX
BACK :  MOV  AL, [SI]           ; Get byte from memory
        OUT  DX, AL             ; Send byte to output port
        INC  DX                 ; Increment port address
        INC  SI                 ; Increment memory pointer
        LOOP BACK               ; Repeat until CX = 0
        RET
```

Hardware Controlled Data Transfer

In this technique external device is used to control data transfer. External device generates address and control signals required to control data transfer and allows

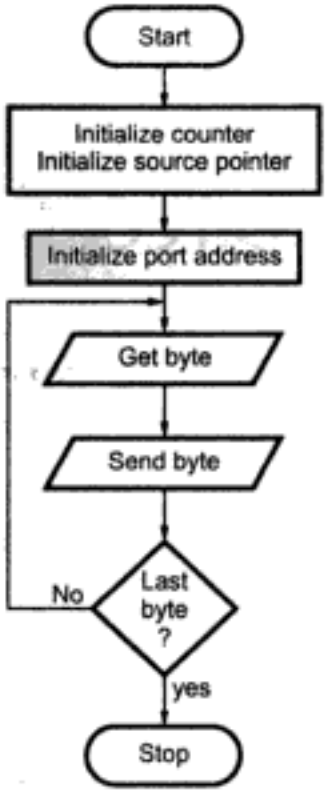


Fig. 6.1 Flowchart

peripheral device to directly access the memory. Hence this technique is referred to as **Direct Memory Access** (DMA) and external device which controls the data transfer is referred to as **DMA controller**. Fig. 6.2 shows that how DMA controller operates in a microprocessor system.

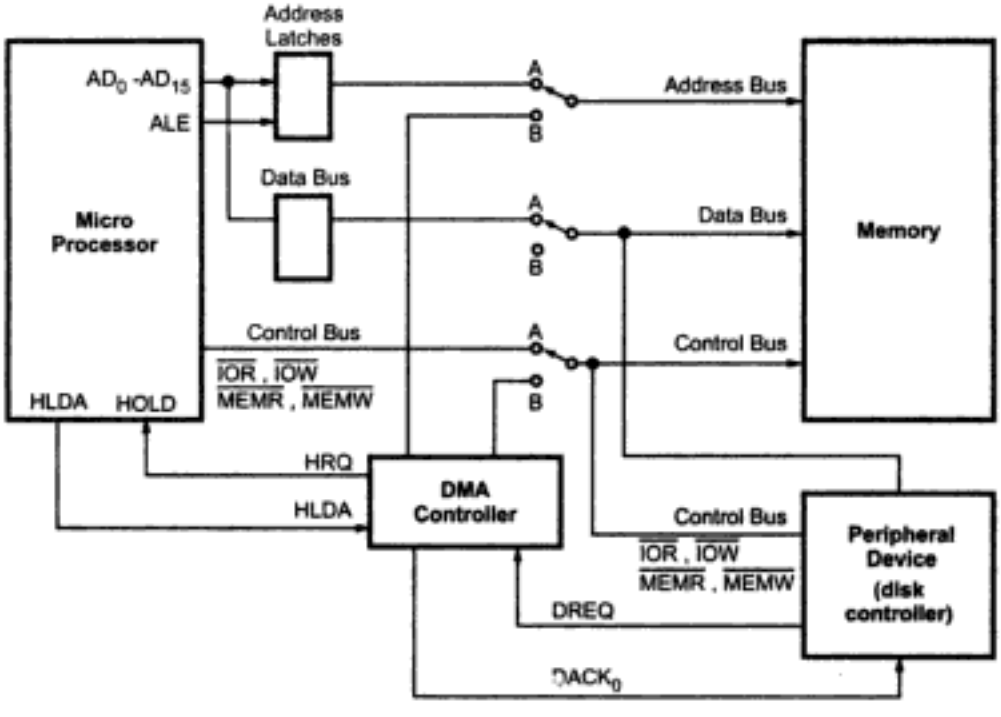


Fig. 6.2 DMA controller operating in a microprocessor system

DMA Idle Cycle

When the system is turned on, the switches are in the A position, so the buses are connected from the microprocessor to the system memory and peripherals. Microprocessor then executes the program until it needs to read a block of data from the disk. To read a block of data from the disk microprocessor sends a series of commands to the disk controller device telling it to search and read the desired block of data from the disk. When disk controller is ready to transfer first byte of data from disk, it sends DMA request DRQ signal to the DMA controller. Then DMA controller sends a hold request HRQ signal to the microprocessor HOLD input. The microprocessor responds this HOLD signal by floating its buses and sending out a hold acknowledge signal HLDA, to the DMA controller. When the DMA controller receives the HLDA signal, it sends a control signal to change switch position from A to B. This disconnects the microprocessor from the buses and connects DMA controller to the buses.

DMA Active Cycle

When DMA controller gets control of the buses, it sends the memory address where the first byte of data from the disk is to be written. It also sends a DMA acknowledge, DACK signal to the disk controller device telling it to get ready for data transfer. Finally (in case of DMA write operation), it asserts both the $\overline{IOR}$ and $\overline{MEMW}$ signals on the control bus. Asserting the $\overline{IOR}$ signal enables the disk controller to output the byte of data from the disk on the data bus and asserting the $\overline{MEMW}$ signal enables the addressed memory to accept data from the data bus. In this technique data is transferred directly from the disk controller to the memory location without passing through the CPU or the DMA controller.

When the data transfer is complete, the DMA controller unasserts the HOLD request signal to the microprocessor and releases the bus by changing switch position from B to A. After getting the control of all buses the microprocessor executes the remaining program.

6.1 Features of 8257

1. It is a programmable, 4-channel, direct memory access controller. Each channel can be programmed individually. Therefore, we can interface 4 input/output devices with 8257.
2. Each channel includes a 16-bit DMA address register and a 14-bit counter. DMA address register gives the address of the memory location and counter specifies the number of DMA cycles to be performed. As counter is 14-bit, each channel can transfer 2^{14} (16 kbytes) without intervention of microprocessor.
3. It maintains the DMA cycle count for each channel and activates a control signal TC (Terminal count) to indicate the peripheral that the programmed number of DMA cycles are complete.
4. It provides another control signal MARK to indicate peripheral that the current DMA cycle is the 128th cycle since the previous MARK output.

5. It has priority logic that resolves the peripherals requests. The priority logic can be programmed to work in two modes, either in fixed mode or rotating priority mode.
6. It provides inhibit logic which can be used to inhibit individual channels.
7. It allows data transfer in two modes : burst mode and cycle steal (single byte transfer) mode.
8. It can execute three DMA cycles : DMA read, DMA write and DMA verify.
9. Auto load feature of 8257 permits repeat block or block chaining operations.
10. It operates in two modes : slave and master.
11. When DMA is in master mode, AEN signal provided by 8257 allows to isolate CPU buffers, latches and other devices from the system bus.
12. Extended write mode of 8257 prevents the unnecessary occurrence of wait states in the 8257, increasing the system throughput.
13. It operates on single TTL clock and it is completely TTL compatible.
14. It can be interfaced with all Intel microprocessor.
15. It transfers one byte of data in four clock cycles. Thus giving high transfer rate such as 500 Kbytes/second at 2 MHz clock input.
16. Like 8085, 8257 also has READY input which allows 8257 to interface slower memory or I/O devices that can not meet bus setup times required by the 8257.

6.2 Pin Diagram of 8257

Fig. 6.3 shows pin diagram of 8257.

Data Bus (D_0-D_7) : These are bi-directional tri-state signals connected to the system data bus. When CPU is having control of system bus it can access contents of address register, status register, mode set register, and a terminal count register and it can also program, control registers of DMA controller, through the data bus.

During DMA cycles these lines are used to send the most significant bytes of the memory address from one of the DMA address registers.

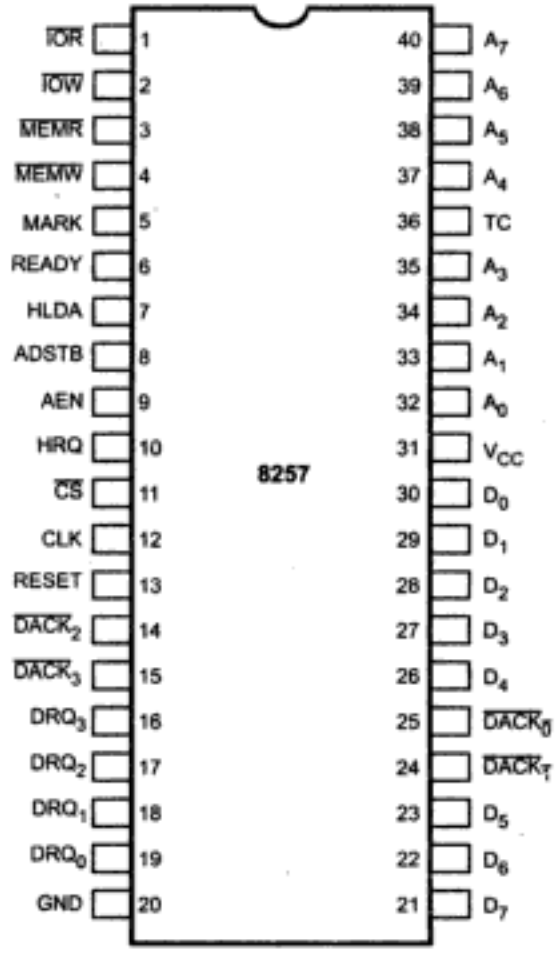


Fig. 6.3 Pin diagram of 8257

Address Bus (A_0-A_3 and A_4-A_7) : The four least significant lines A_0-A_3 are bi - directional tri - state signals. In the idle cycle they are inputs and used by the CPU to address the register to be loaded or read. In the active cycle they output the lower 4 bits of the address for DMA operation. A_4-A_7 are unidirectional lines, provide 4-bits of address during DMA service.

Address Strobe (ADSTB) : This signal is used to demultiplex higher byte address and data using external latch.

Address Enable (AEN) : This active high signal enables the 8-bit latch containing the upper 8-address bits onto the system address bus. AEN can also be used to disable other system bus drivers during DMA transfers.

Memory Read and Memory Write ($\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$) :

These are active low tri-state signals. The $\overline{\text{MEMR}}$ signal used to access data from the addressed memory location during a DMA read or memory-to-memory transfer and $\overline{\text{MEMW}}$ signal is used to write data to the addressed memory location during DMA write or memory to memory transfer.

I/O Read and I/O Write ($\overline{\text{IOR}}$ AND $\overline{\text{IOW}}$) : These are active low bi-directional signals. In idle cycle, these are an input control signals used by CPU to read/write the control registers. In the active cycle $\overline{\text{IOR}}$ signal is used to access data from a peripheral and $\overline{\text{IOW}}$ signal is used to send data to the peripheral.

Chip Select ($\overline{\text{CS}}$) : This is an active low input, used to select the 8257 as an I/O device during the idle cycle. This allows CPU to communicate with 8257.

Reset : This active high signal clears the command, status, request and temporary registers. It also clears the first/last flip-flop and sets the Master Register. After reset the device is in the idle cycle.

Ready : This input is used to extend the memory read and write signals from the 8257 to interface slow memories or I/O devices.

Hold Request (HRQ) : Any valid DREQ causes 8257 to issue the HRQ. It is used for requesting CPU to get the control of system bus.

Hold Acknowledge (HLDA) : The active high Hold Acknowledge from the CPU indicates that it has relinguished control of the system bus.

$DREQ_0$-$DREQ_3$: These are DMA request lines, which are activated to obtain DMA service, until the corresponding DACK signal goes active.

$DACK_0$-$DACK_3$: These are used to indicate peripheral devices that the DMA request is granted.

Terminal Count (TC) : This is active high signal concern with the completion of DMA service. The TC output signal is activated at the end of DMA service, i.e. when present cycle is a last cycle for the current data block.

MARK : This output notifies the selected peripheral that the current DMA cycle is the 128 th cycle since the previous MARK output. MARK always occurs at 128 (all multiplies of 128) cycles from the end of the data block.

6.3 Block Diagram of 8257

Fig. 6.4 shows the functional block diagram of IC 8257.

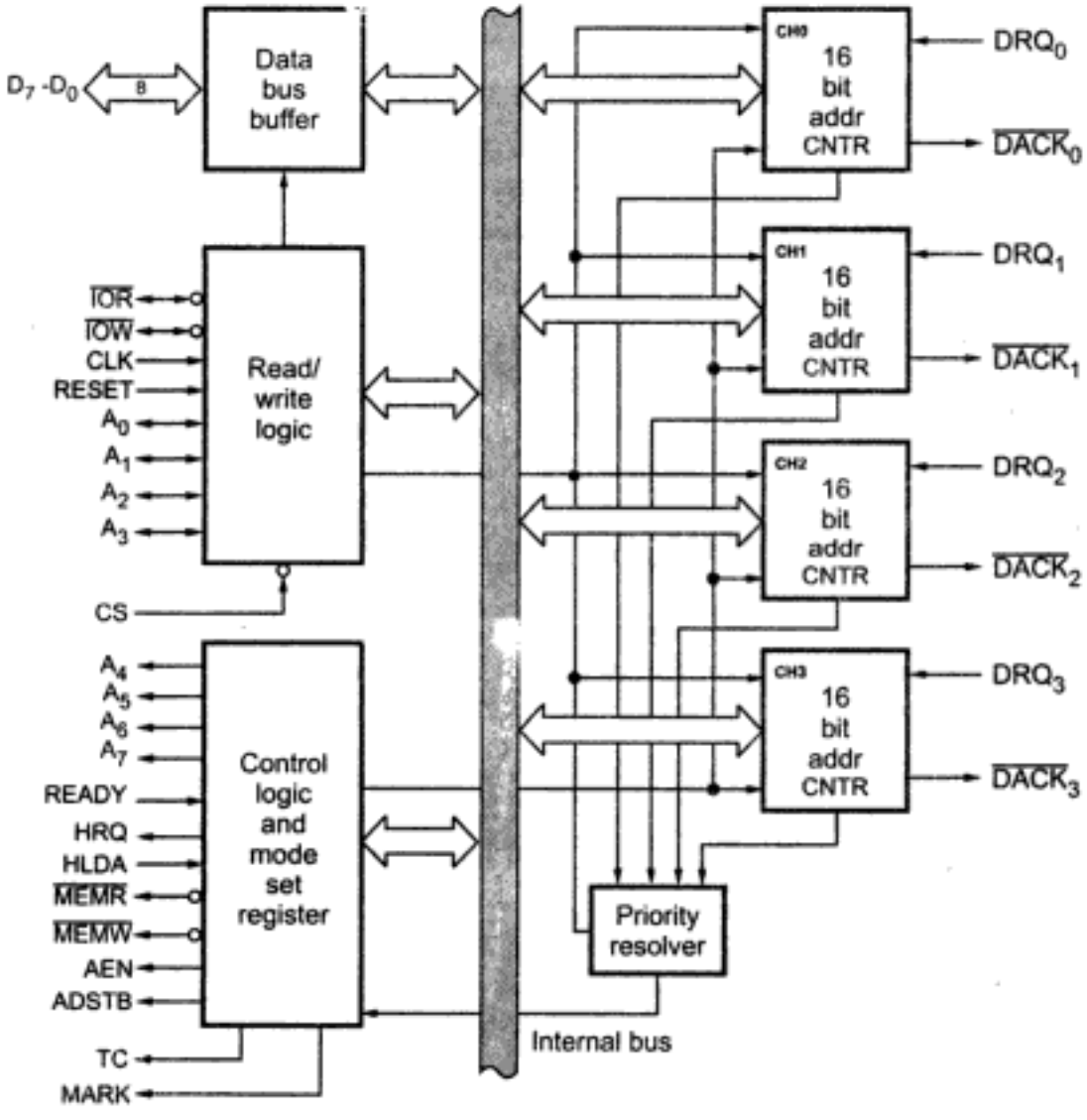


Fig. 6.4 Functional block diagram of 8257

Data Bus Buffer

It is a tri-state, bi-directional, eight bit buffer which interfaces the 8257 to the system data bus. In the slave mode, it is used to transfer data between microprocessor and

internal registers of 8257. In master mode, it is used to send higher byte address (A_8-A_{15}) on the data bus.

Read/Write logic

When the CPU is programming or reading one of the internal registers of 8257 (i.e. when the 8257 is in the slave mode), the Read/Write logic accepts the I/O Read ($\overline{IOR}$) or I/O Write ($\overline{IOW}$) signal, decodes the the least significant four address bits (A_0 - A_3) and either writes the contents of the data bus into the addressed register (if $\overline{IOW}$ is low) or places the contents of the addressed register onto the data bus (if $\overline{IOR}$ is low).

During DMA cycles (i.e. when the 8257 is in the master mode) the Read/Write logic generates the I/O read and memory write (DMA write cycle) or I/O write and memory read (DMA read cycle) signals which control the data transfer between peripheral and memory device.

DMA Channels

The 8257 provides four identical channels, labeled CH_0 to CH_3. Each channel has two sixteen bit registers : i) A DMA address register, and ii) A terminal count register.

DMA Address Register :

Fig. 6.5 shows the format of DMA address register. It specifies the address of the first memory location to be accessed. It is necessary to load valid memory address in the DMA address register before channel is enabled.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0

Fig. 6.5 Format of DMA address register

Terminal Count Register : Fig. 6.6 shows the format of terminal count register.

T_1	T_0	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

← 14 bit binary count (N–1) →

Fig. 6.6

T_1	T_0	Type of operation
0	0	DMA Verify cycle
0	1	DMA Write cycle
1	0	DMA READ cycle
1	1	Illegal

Note : N is number of bytes to be transferred.

The value loaded into the low order 14 bits ($C_{13} - C_0$) of the terminal count register specifies the number of DMA cycles minus one before the terminal count (TC) output is activated. Therefore, for N number of desired DMA cycles it is necessary to load the value N–1 into the low order 14-bits of the terminal count register. The most significant 2 bits of the terminal count register specifies the type of DMA operation to be performed. It is necessary to load count for DMA cycles and operational code for valid DMA cycle in the terminal count register before channel is enabled.

Control logic

It controls the sequence of operations during all DMA cycles (DMA read, DMA write, DMA verify) by generating the appropriate control signals and the 16-bit address that specifies the memory location to be accessed. It consists of mode set register and status register. Mode set register is programmed by the CPU to configure 8257 whereas the status register is read by CPU to check which channels have reached a terminal count condition and status of update flag.

Mode Set Register

Fig. 6.7 gives the format of mode set register. Least significant four bits of mode set register, when set, enable each of the four DMA channels. Most significant four bits allow four different options for the 8257.

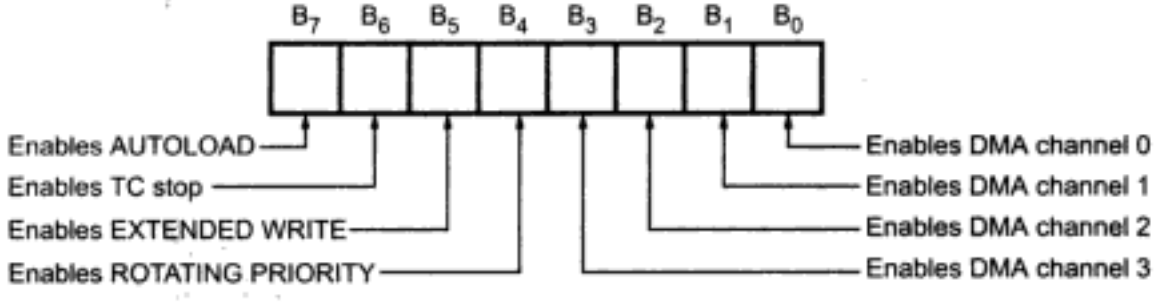


Fig. 6.7 Mode set register

It is normally programmed by the CPU after initializing the DMA address registers and terminal count registers. It is cleared by the RESET input, thus disabling all options, inhibiting all channels, and preventing bus conflicts on power-up.

Status Register

Fig. 6.8 shows the status register format. As said earlier, it indicates which channels have reached a terminal count condition and includes the update flag described previously.

The TC status bit, if one, indicates terminal count has been reached for that channel. TC bit remains set until the status register is read or the 8257 is reset. The update flag, however, is not affected by a status read operation.

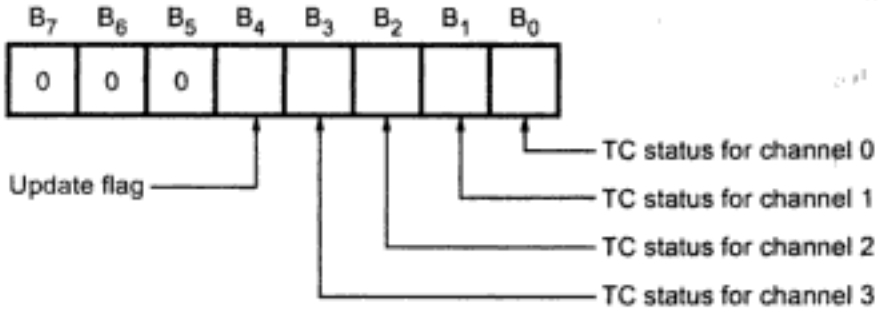


Fig. 6.8 Status register

The update flag bit, if one, indicates CPU that 8257 is executing update cycle. In update cycle 8257 loads parameters in channel 3 to channel 2.

Priority Resolver

It resolves the peripherals requests. It can be programmed to work in two modes, either in fixed mode or rotating priority mode.

6.4 Operating Modes of 8257

The 8257 can be programmed to operate in following modes :

1. Rotating Priority Mode

In rotating priority mode, the priority of the channels has a circular sequence. In this, channel being serviced gets the lowest priority and the channel next to it gets the highest priority as shown in Fig. 6.9.

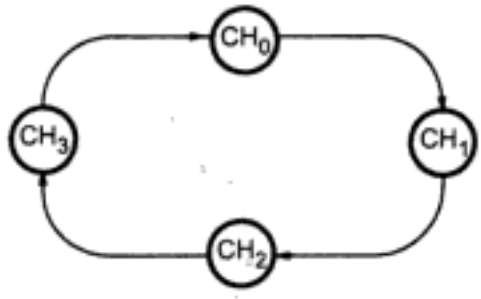


Fig. 6.9 Rotating priority

Thus, with rotating priority in a single chip DMA system, any device requesting service is guaranteed to be recognized after no more than three higher priority services have occurred. This prevents any one channel from monopolizing the system. The rotating priority mode can be set by writing logic '1' in the bit 4 of the mode set register.

Fixed Priority Mode

In the fixed priority, channel 0 has the highest priority and channel 3 has the lowest priority. Table 6.1 shows the priority ratings.

	Priority	Channel
Highest	1	0
	2	1
	3	2
Lowest	4	3

Table 6.1 Priority ratings

Hidden page

Hidden page

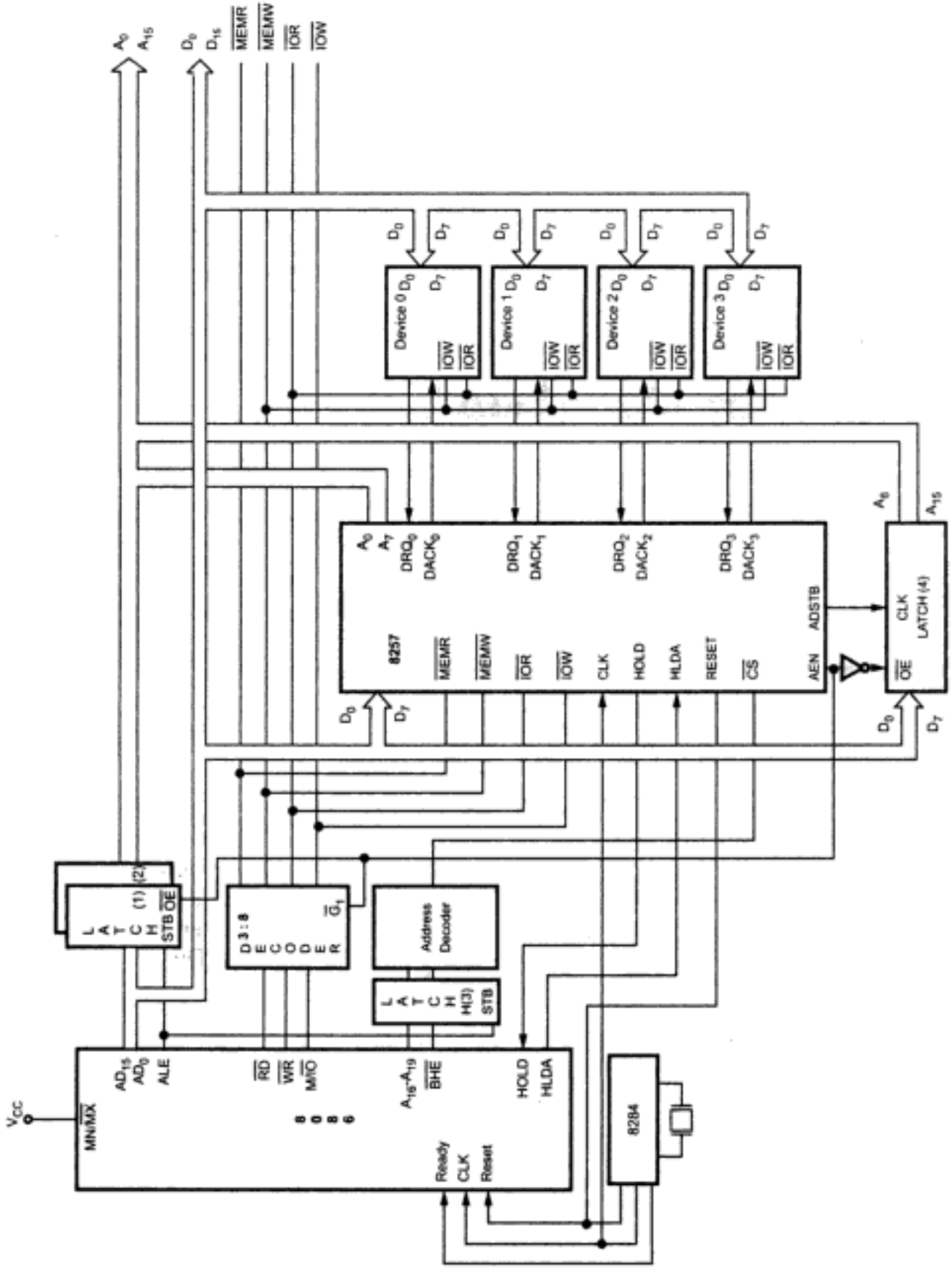


Fig. 6.10

Hidden page

Hidden page

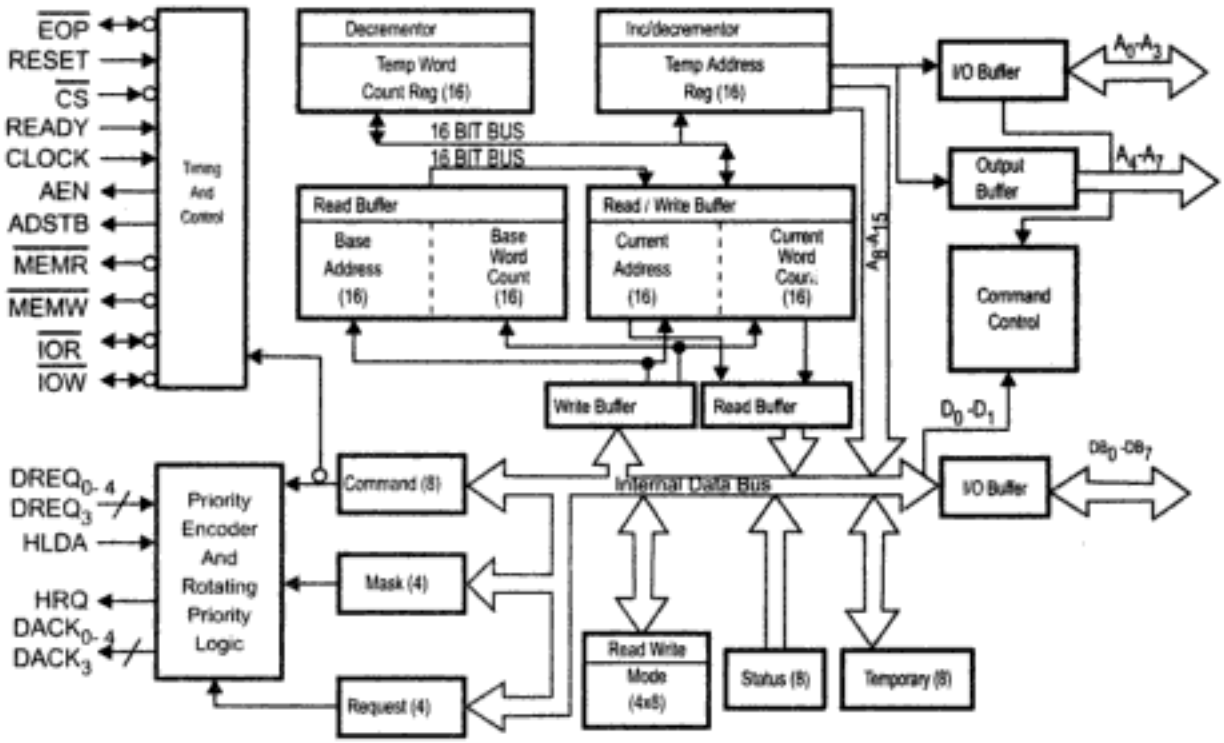


Fig. 6.12 Internal block diagram of 8237A

2. Program Command Control Block : It decodes various commands given to the 8237A by the microprocessor before servicing a DMA request. It also decodes the Mode Control Word, which is used to select the type of DMA during the servicing.

3. Priority Encoder Block : It resolves the priority between DMA channels requesting service simultaneously.

Internal Registers : The 8237A contains 344 bits internal memory in the form of registers. Table 6.2 gives the name, size and number of each register.

Name	Size	Number
Base Address Registers	16 bits	4
Base Word Count Registers	16 bits	4
Current Address Registers	16 bits	4
Current Word Count Registers	16 bits	4
Temporary Address Registers	16 bits	1
Temporary Word Count Registers	16 bits	1
Status Registers	8 bits	1
Command Registers	8 bits	1
Temporary Registers	8 bits	1
Mode Registers	6 bits	4
Mask Registers	4 bits	1
Request Registers	4 bits	1

Table 6.2

Hidden page

Hidden page

7. The word/byte transfer count is decremented and the memory address is incremented.
8. The DMAC continues to execute transfer cycles until the I/O device deasserts DRQ indicating its inability to continue delivering data. The DMAC deasserts HOLD signal, giving the buses back to microprocessor. It also deasserts $\overline{DACK}$.
9. I/O device can re-initiate demand transfer by reasserting DRQ signal.
10. Transfer continues in this way until the transfer count has been exhausted.

The flowcharts in the Fig. 6.13 summarized the three data transfer modes of DMA

(See Fig. 6.13 on next page).

Cascade Mode

DMA channels can be expanded using this mode. Fig. 6.14 shows that two additional devices are cascaded to the master device using two channels of the master device. This is two level DMA system. In this the HRQ and HLDA signals from the additional 8237A are connected to the DREQ and DACK signals of a channel of the master 8237A. This allows the DMA requests of the additional devices to communicate through the priority network circuitry of the preceding device.

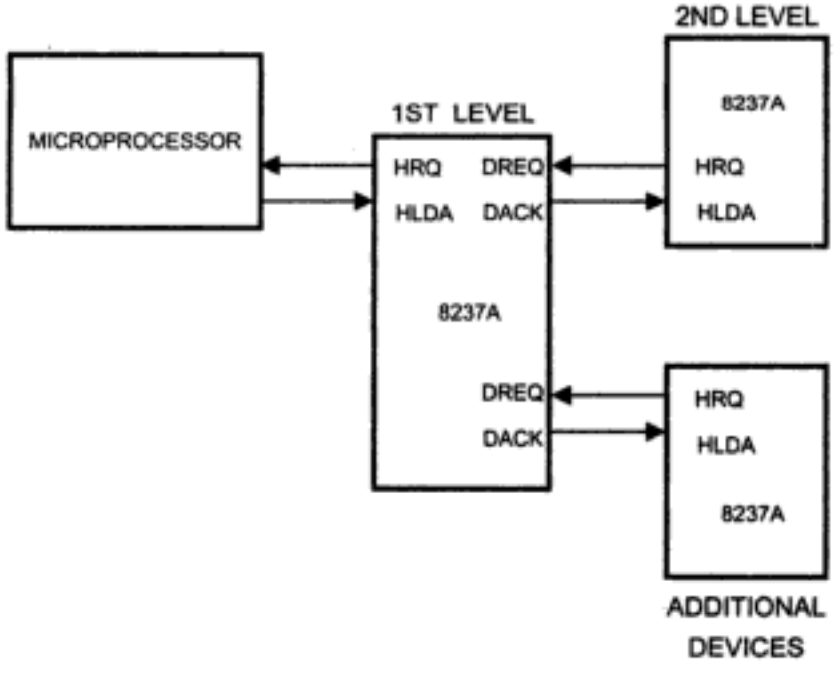


Fig. 6.14 Cascade 8237s

Note : More 8237As can be added by adding more levels in the DMA system.

Fig. 6.15 (See Fig. on page 6-20) shows the detail connections for master and slave DMAC's.

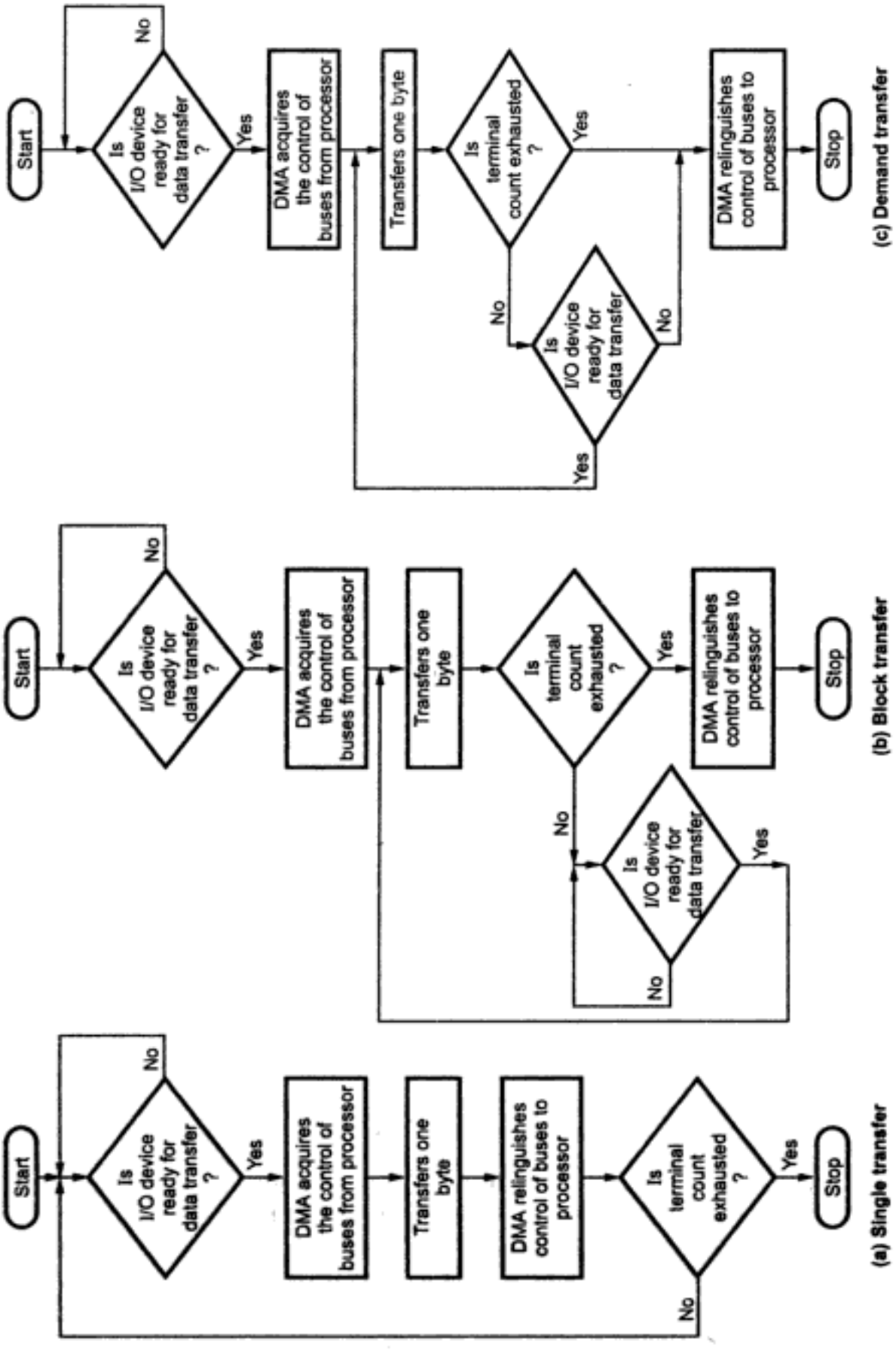


Fig. 6.13 Three data transfer modes of DMA

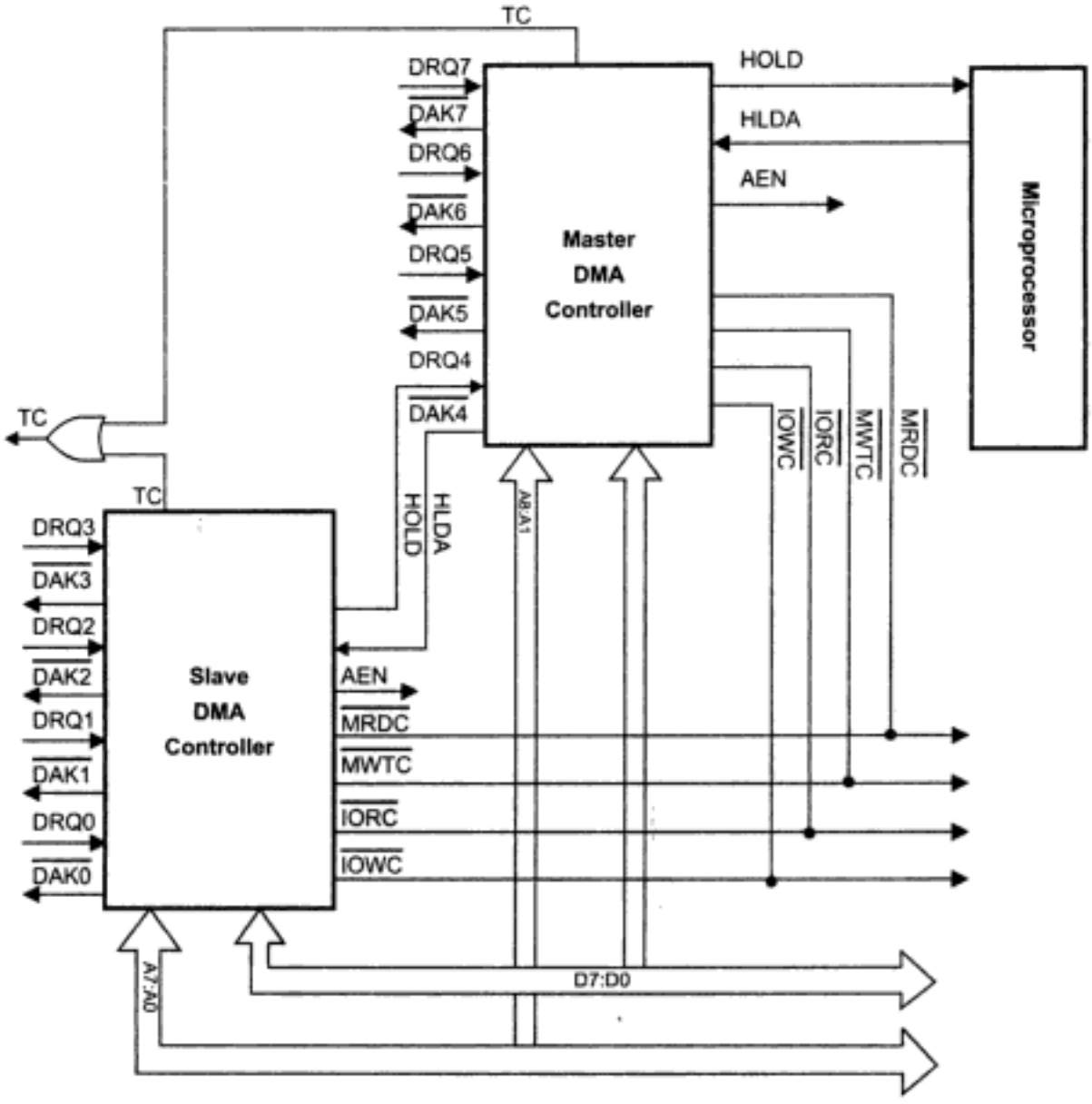


Fig. 6.15 Cascaded DMA controllers

6.10 Transfer Types

6.10.1 Memory-to-Memory Transfer

In this mode block of data from one memory address is moved to another memory address. In this mode current address register of channel 0 is used to point the source address and the current address register of channel 1 is used to point the destination address in the first transfer cycle, data byte from the source address is loaded in the temporary register of the 8237A and in the next transfer cycle the data from the temporary register is stored in the memory pointed by destination address. After each data transfer current address registers are decremented or incremented according to current settings. The channel 1 current word count register is also decremented by 1 after each data transfer. When the word count of channel 1 goes to FFFFH, a TC is generated which activates $\overline{EOP}$ output terminating the DMA service.

6.10.2 Autoinitialize

In this mode, during the initialization the base address and word count registers are loaded simultaneously with the current address and word count registers by the microprocessor. The address and the count in the base registers remain unchanged throughout the DMA service.

After the first block transfer i.e. after the activation of the $\overline{EOP}$ signal, the original values of the current address and current word count registers are automatically restored from the base address and base word count register of that channel. After autoinitialization the channel is ready to perform another DMA service, without CPU intervention.

6.11 Priority

In the 8237A there are two priority selection options.

1. Fixed Priority
2. Rotating Priority.

6.11.1 Fixed Priority

In the fixed priority channel 0 has the highest priority and the channel 3 has the lowest priority. Table 6.3 shows the priority ratings.

	Priority	Channel
Highest	1	0
	2	1
	3	2
Lowest	4	3

Table 6.3

In the fixed priority, after recognition of any one channel for service, the other channels are prevented from interfering with that service until it is completed.

6.11.2 Rotating Priority

In this, channel being serviced gets the lowest priority and the channel next to it gets the highest priority as shown in Fig. 6.16.

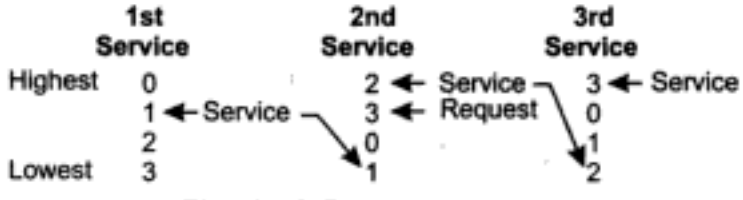


Fig. 6.16 Rotating priority

With rotating priority in a single chip DMA system, any device requesting service is guaranteed to be recognized after no more than three higher priority services have occurred. This prevents any one channel from monopolizing the system.

6.12 Register Description

1. Current Address Register : Each channel has 16-bit current address register. This register stores the value of the address used during DMA transfers. The address in the current address register is automatically incremented or decremented after each transfer. This register is loaded or read by the microprocessor and it also be re-initialized back to its original value after $\overline{EOP}$ in the autoinitialization mode.

2. Current Word Register : Each channel has a 16-bit current word count register. This register determines the number of transfers to be performed. The actual number of transfers will be one more than the number stored in the current word count register. After each transfer the contents of word count register is decremented by 1. When the value in the register goes from zero to FFFFH, a TC will be generated. This register is loaded or read by the microprocessor and it also be reinitialized back to its original value after $\overline{EOP}$ in the autoinitialize mode.

3. Base Address and Base Word Count Registers : Each channel has base address and base word count registers. These 16-bit registers store the original value of their associated current registers. During autoinitialization these values are used to restore the current registers to their original values. The base registers are stored simultaneously with their corresponding current registers.

4. Request Register : The 8237A can respond to requests for DMA service which are initiated by software as well as by a DREQ. Each channel has a request bit associated with it in the 4-bit request register. Each bit in the request register is set or reset separately under software control and is automatically cleared upon generation of a TC or external $\overline{EOP}$.

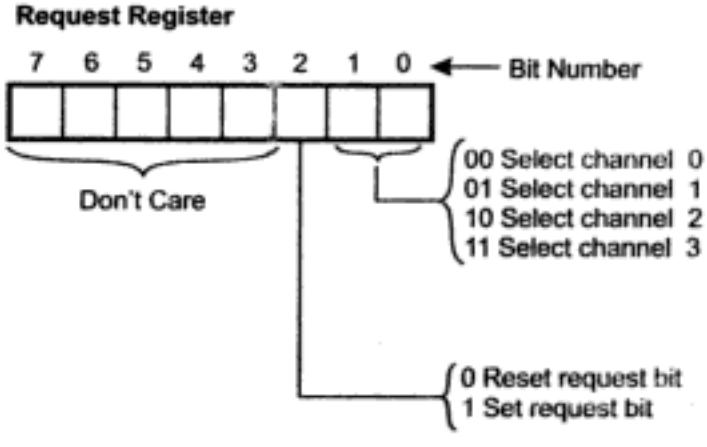


Fig. 6.17 Request register

5. Command Register : Fig. 6.18 shows the bit pattern of the command register. It is 8-bit register which controls the operation of 8237A.

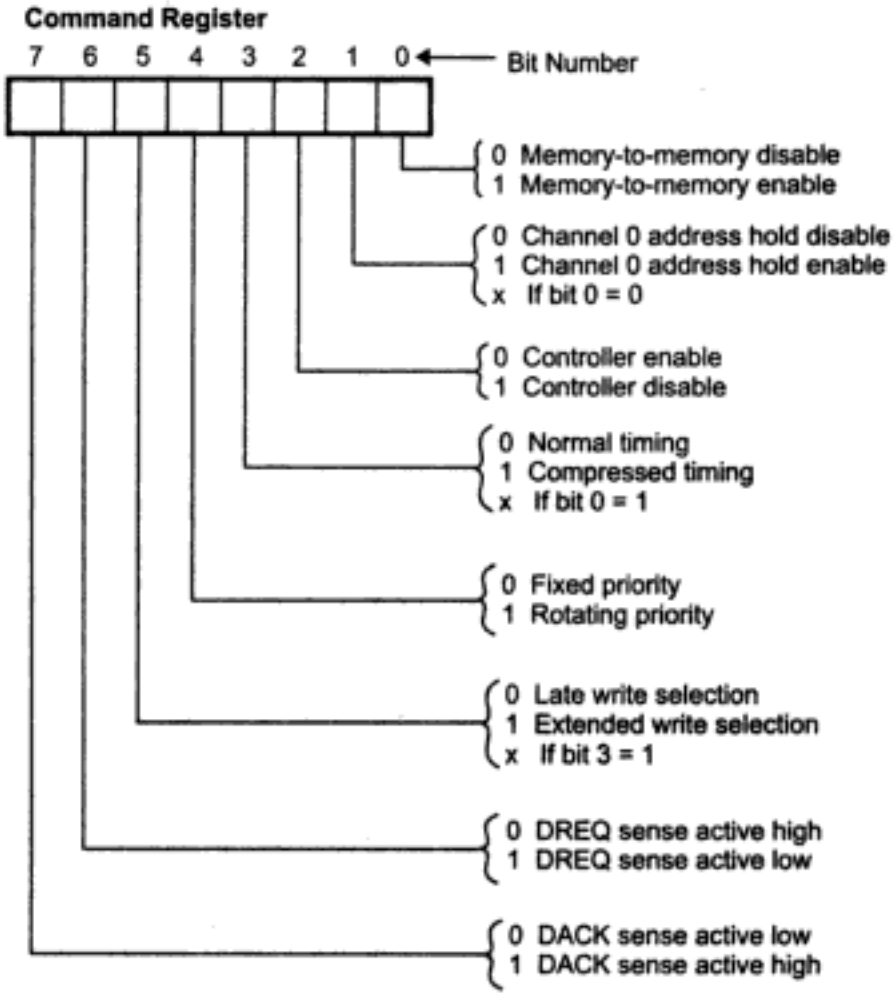


Fig. 6.18 Command register

6. Mask Register : Each channel request can be individually masked by setting the proper bit pattern in the mask register. Fig. 6.19 shows the bit patterns of the mask register.

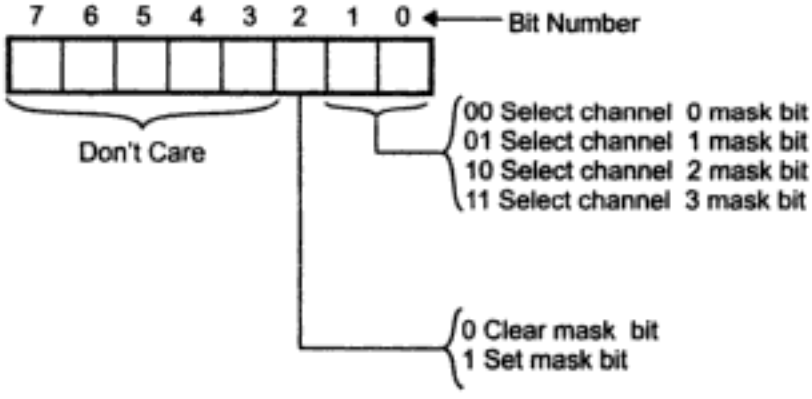


Fig. 6.19 Mask register

Note : All four bits of the mask register can be written with a single command. Fig. 6.20.

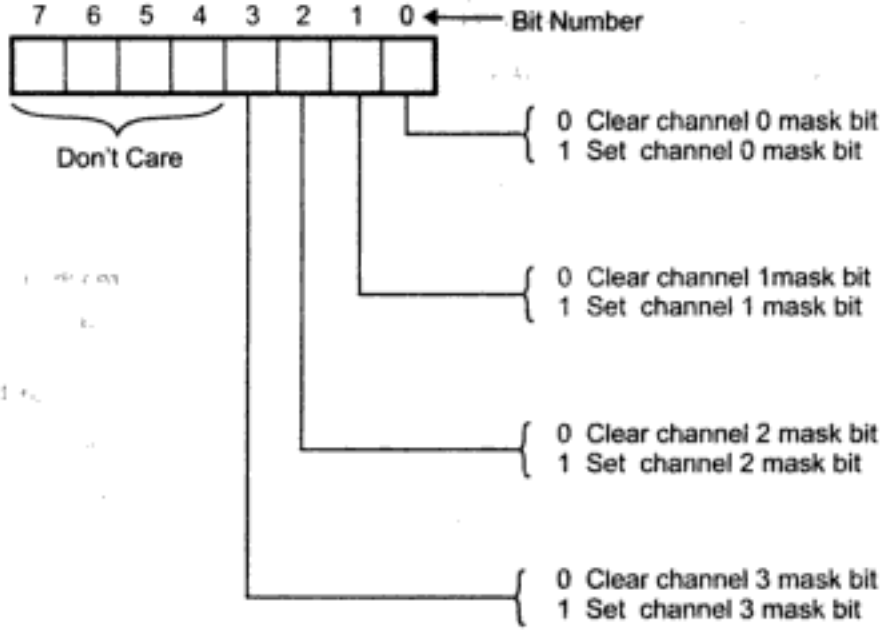


Fig. 6.20 Mask register using single command

7. Mode Register : Each channel has a 6-bit mode register associated with it. The bit pattern of the mode register is as shown in the Fig. 6.21.

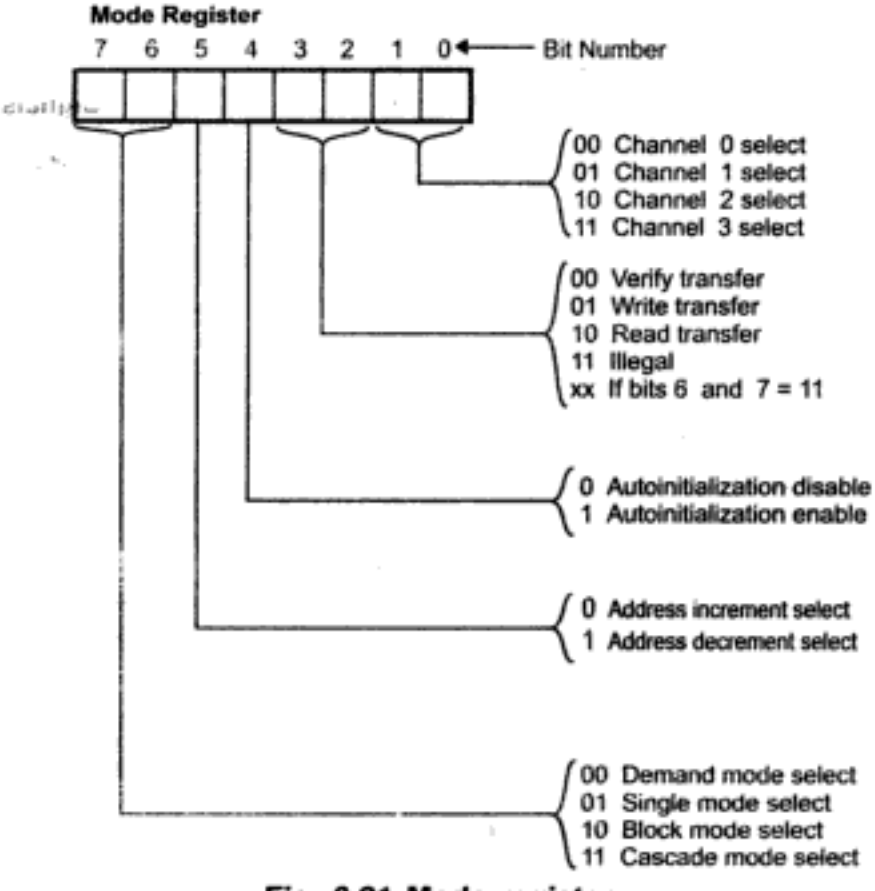


Fig. 6.21 Mode register

8. Status Register : The status register contains the information about the status of the DMA channels. It includes which channels have reached a terminal count and which channels have pending DMA requests. The bit pattern for status register is shown in Fig. 6.22.

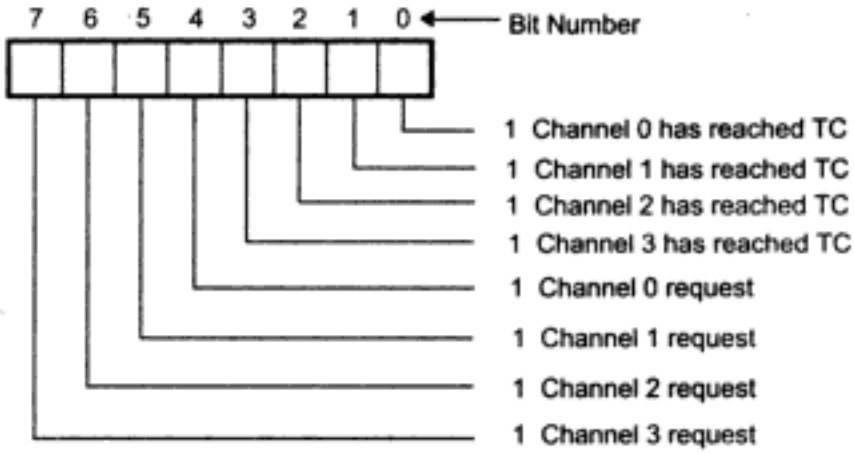


Fig. 6.22 Status register

Temporary Register : It is used to hold data during memory to memory transfers.

9. Register Addresses : Table 6.4 gives the addresses for different registers of the 8237A.

Channel	Register	Operation	Signals						
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A_3	A_2	A_1	A_0
0	Base and Current Address	Write	0	1	0	0	0	0	0
			0	1	0	0	0	0	0
	Current Address	Read	0	0	1	0	0	0	0
			0	0	1	0	0	0	0
	Base and Current Word Count	Write	0	1	0	0	0	0	1
			0	1	0	0	0	0	1
	Current Word Count	Read	0	0	1	0	0	0	1
			0	0	1	0	0	0	1

Hidden page

Software Commands : The 8237A responds to the special software commands in the program mode. Each software command has the specific code. The Table 6.5 lists the code for special software commands provided by 8237A.

Signals						Operation
A_3	A_2	A_1	A_0	$\overline{IOR}$	$\overline{IOW}$	
1	0	0	0	0	1	Read Status Register
1	0	0	0	1	0	Write Command Register
1	0	0	1	0	1	Illegal
1	0	0	1	1	0	Write Request Register
1	0	1	0	0	1	Illegal
1	0	1	0	1	0	Write Single Mask Register Bit
1	0	1	1	0	1	Illegal
1	0	1	1	1	0	Write Mode Register
1	1	0	0	0	1	Illegal
1	1	0	0	1	0	Clear Byte Pointer Flip/Flop
1	1	0	1	0	1	Read Temporary Register
1	1	0	1	1	0	Master Clear
1	1	1	0	0	1	Illegal
1	1	1	0	1	0	Clear Mask Register
1	1	1	1	0	1	Illegal
1	1	1	1	1	0	Write All Mask Register Bits

Table 6.5

6.13 Interfacing

Fig. 6.23 shows that a typical method for configuring a DMA system with the 8237A controller and an 8088 microprocessor system. The multimode DMA controller issues a HRQ signal to the microprocessor whenever there is at least one valid DMA request from a peripheral device. When processor responds with a HLDA signal, the 8237A takes control of the address bus, data bus and control bus. The 8237A sends lower byte of the address on the A_0-A_7 bus and higher byte on the data bus. The contents of the data bus are then latched into the external latch to complete the full 16-bits of the address bus.

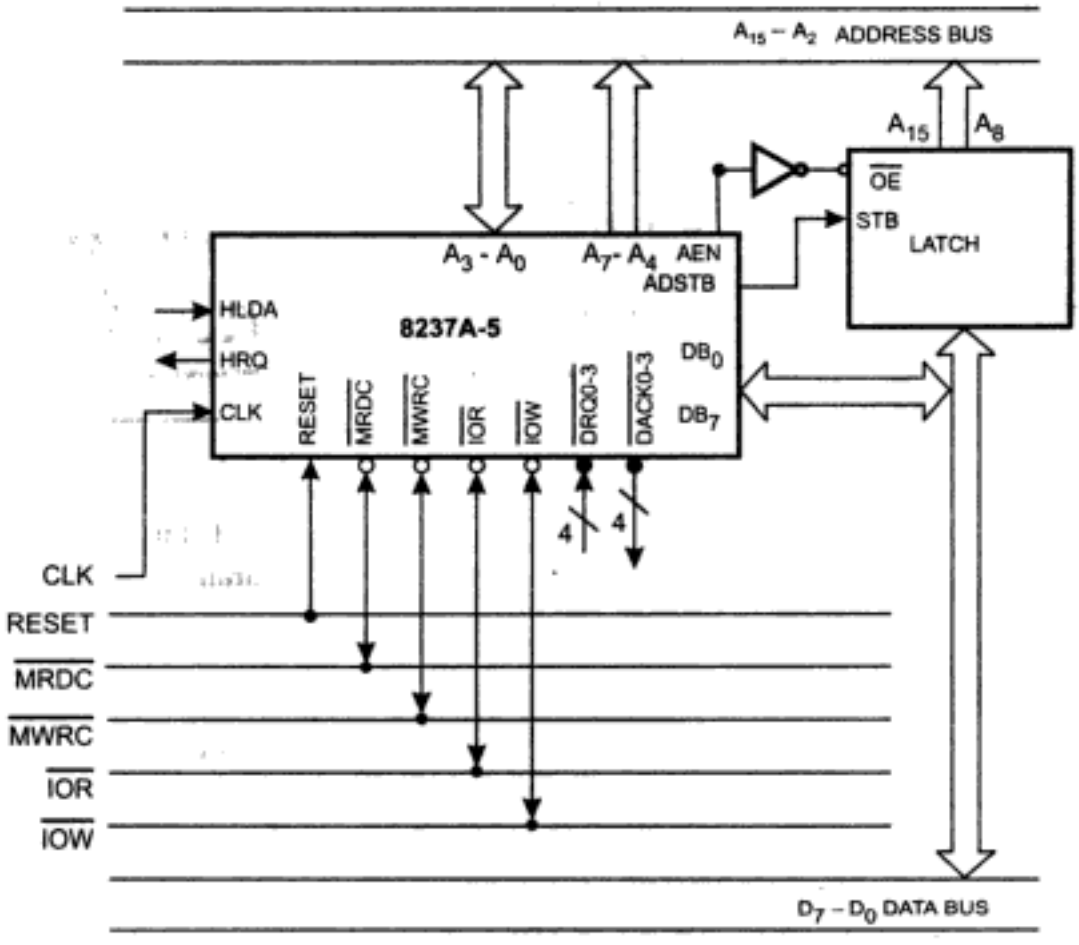


Fig. 6.23 Interfacing of 8237 and 8088

Review Questions

1. *What is the need of DMA in microprocessor applications?*
2. *Explain the architecture, organisation and various modes of operation of a programmable DMA controller 8257.*
3. *Explain in brief the different types of DMA data transfer.*
4. *What do you understand by the following terms ?*
   - *Rotating priority mode.*
   - *TC STOP mode.*
5. *Give the interfacing scheme of 8257/8237 and 8086.*
6. *List the features of 8237 A DMA controller.*
7. *Draw and explain the architecture of 8237 A.*
8. *Explain the operating modes of 8237 A.*
9. *Explain the data transfer types supported by 8237 A.*
10. *Explain the priority options available in 8237 A.*
11. *Draw and explain the interfacing of 8237 A and 8088.*

❑❑❑

7

8255 PPI (Programmable Peripheral Interface)

The 8255 is a general purpose programmable I/O device used for parallel data transfer. It has 24 I/O pins which can be grouped in three 8-bit parallel ports : Port A, Port B and Port C. The eight bits of port C can be used as individual bits or be grouped in two 4-bit ports : C_{upper} (C_U) and C_{lower} (C_L).

The 8255, primarily, can be programmed in two basic modes : Bit Set/Reset (BSR) mode and I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into three modes :

Mode 0 : Simple Input/Output
Mode 1 : Input/Output with handshake
Mode 2 : Bi-directional I/O data transfer

The function of I/O pins (input or output) and modes of operation of I/O ports can be programmed by writing proper control word in the control word register. Each bit in the control word has a specific meaning and the status of these bits decides the function and operating mode of the I/O ports.

7.1 Features of 8255A

1. The 8255A is a widely used, programmable, parallel I/O device.
2. It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O.
3. It is compatible with all Intel and most other microprocessors.
4. It is completely TTL compatible.
5. It has three 8-bit ports : Port A, Port B, and Port C, which are arranged in two groups of 12 pins. Each port has an unique address, and data can be read from or written to a port. In addition to the address assigned to the three ports, another address is assigned to the control register into which control words are written for programming the 8255 to operate in various modes.
6. Its bit set/reset mode allows setting and resetting of individual bits of Port C.

7. The 8255 can operate in 3 I/O modes : (i) Mode 0, (ii) Mode 1, and (iii) Mode 2.

a) In Mode 0, Port A and Port B can be configured as simple 8-bit input or output ports without handshaking. The two halves of Port C can be programmed separately as 4-bit input or output ports.

b) In Mode 1, two groups each of 12 pins are formed. Group A consists of Port A and the upper half of Port C while Group B consists of Port B and the lower half of Port C. Ports A and B can be programmed as 8-bit Input or Output ports with three lines of Port C in each group used for handshaking.

c) In Mode 2, only Port A can be used as a bidirectional port. The handshaking signals are provided on five lines of Port C (PC_3 - PC_7). Port B can be used in Mode 0 or in Mode 1.

8. All I/O pins of 8255 has 2.5 mA DC driving capacity (i.e. sourcing current of 2.5 mA).

7.2 Pin Diagram

Fig. 7.1 shows the pin diagram of 8255.

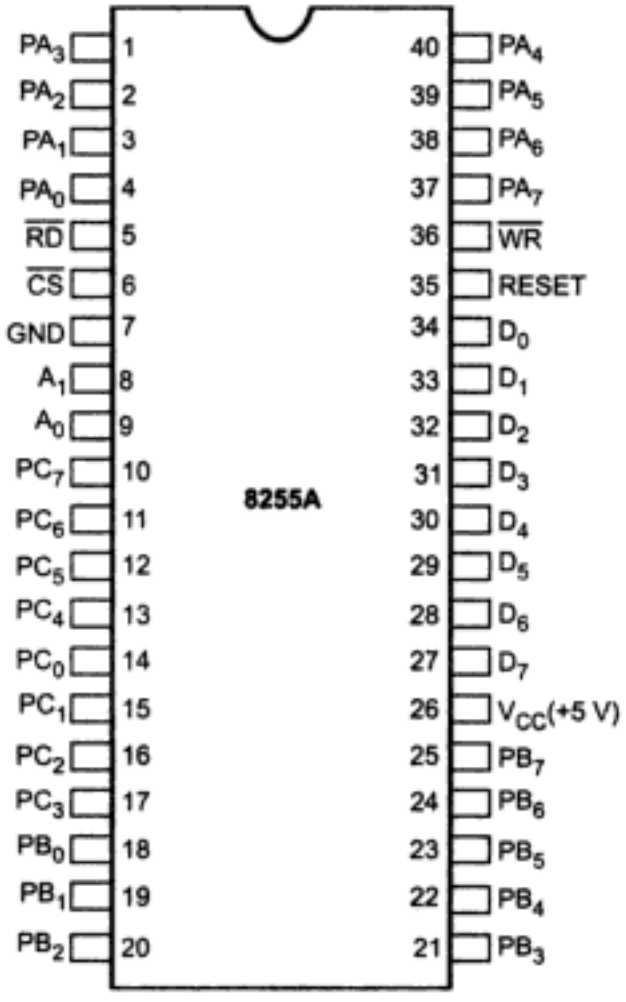


Fig. 7.1 Pin diagram of 8255A

Pin Symbols	Function
D_0-D_7 (Data Bus)	These bi-directional, tri-state data bus lines are connected to the system data bus. They are used to transfer data and control word from microprocessor (8085) to 8255 or to receive data or status word from 8255 to the 8085.
PA_0-PA_7 (Port A)	These 8-bit bi-directional I/O pins are used to send data to output device and to receive data from input device. It functions as an 8-bit data output latch/buffer, when used in output mode and an 8-bit data input buffer, when used in input mode.
PB_0-PB_7 (Port B)	These 8-bit bi-directional I/O pins are used to send data to output device and to receive data from input device. It functions as an 8-bit data, output latch/buffer when used in output mode and an 8-bit data input buffer, when used in input mode.
PC_0-PC_7	These 8-bit bi-directional I/O pins are divided into two groups PC_L (PC_3-PC_0) and PC_U (PC_7-PC_4). These groups individually can transfer data in or out when programmed for simple I/O, and used as handshake signals when programmed for handshake or bi-directional modes.
$\overline{RD}$ (Read)	When this pin is low, the CPU can read the data in the ports or the status word, through the data buffer.
$\overline{WR}$ (Write)	When this input pin is low, the CPU can write data on the ports or in the control register through the data bus buffer.
$\overline{CS}$ (Chip Select)	This is an active low input which can be enabled for data transfer operation between the CPU and the 8255.
RESET	This is an active high input used to reset 8255. When RESET input is high, the control register is cleared and all the ports are set to the input mode. Usually RESET OUT signal from 8085 is used to reset 8255.
A_0 and A_1	These input signals along with $\overline{RD}$ and $\overline{WR}$ inputs control the selection of the control/status word registers or one of the three ports. Table. 7.1. summarizes the status of A_0 , A_1, $\overline{CS}$, $\overline{RD}$ and $\overline{WR}$ to access the control word/ports. A_0 and A_1 are generally connected to the A_0, A_1 pins of the address bus; the 8255 therefore occupies four consecutive locations in the I/O space.

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Operations
					Input (Read) Operation
0	0	0	1	0	Port A to Data Bus
0	1	0	1	0	Port B to Data Bus
1	0	0	1	0	Port C to Data Bus
					Output (Write) Operation
0	0	1	0	0	Data Bus to Port A
0	1	1	0	0	Data Bus to Port B
1	0	1	0	0	Data Bus to Port C
1	1	1	0	0	Data Bus to Control Register

					Disable Function
X	X	X	X	1	Data Bus Tri-stated
1	1	0	1	0	Illegal Condition
X	X	1	1	0	Data Bus Tri-stated

Table 7.1 Port and register select signals summary

7.3 Block Diagram

Fig. 7.2 shows the internal block diagram of 8255A. It consists of data bus buffer. control logic and Group A and Group B controls.

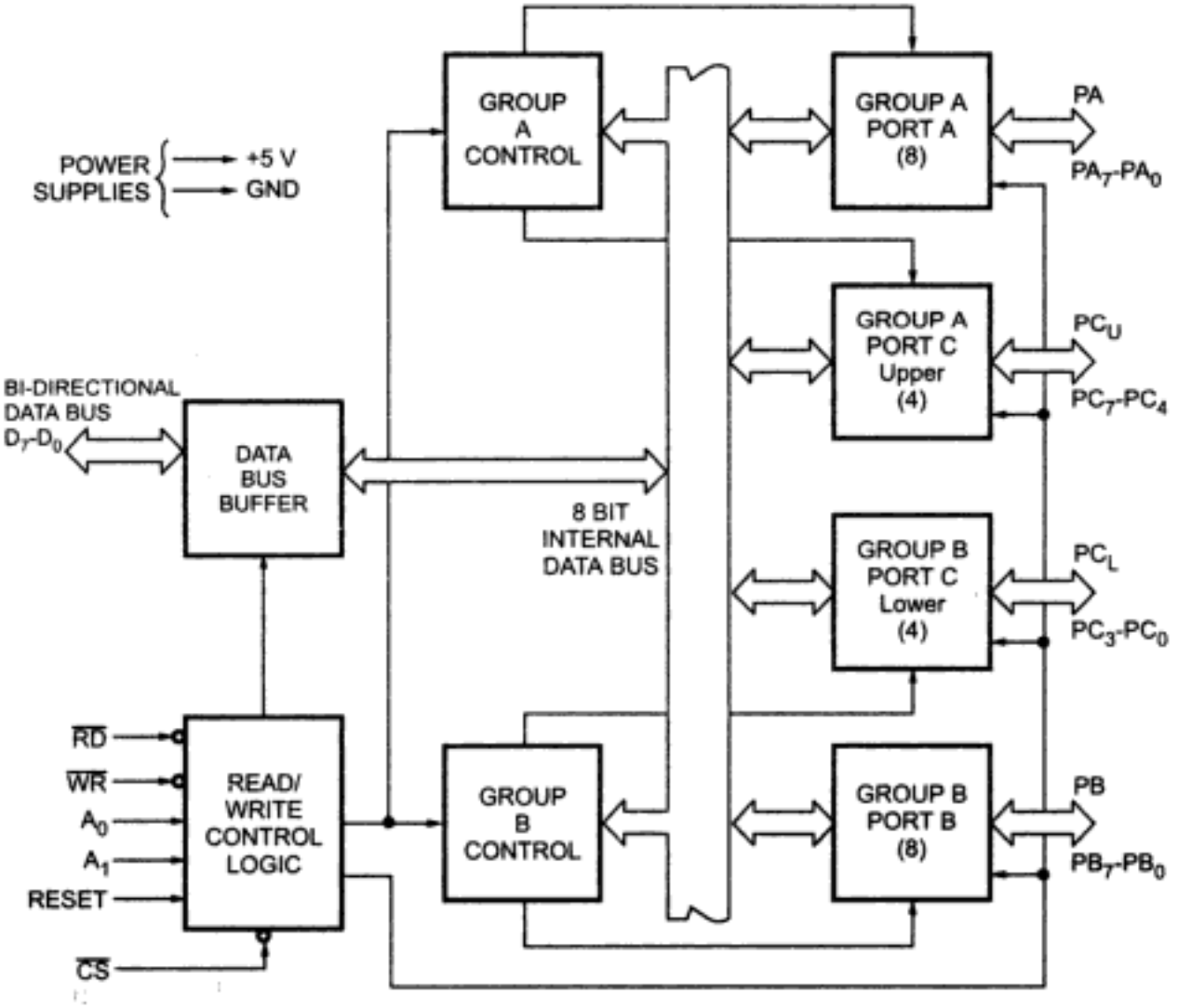


Fig. 7.2 Block diagram of 8255A

Hidden page

1. Outputs are latched. 2. Inputs are buffered, not latched.
3. Ports do not have handshake or interrupt capability.

Mode 1 : Input/Output with handshake

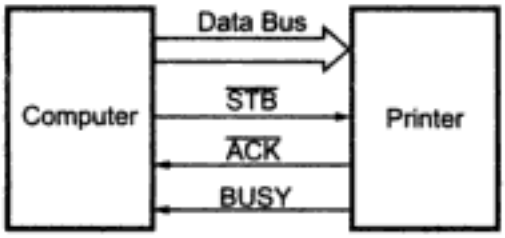


Fig. 7.3 Data transfer between computer and printer using handshaking signals

In this mode, input or output data transfer is controlled by handshaking signals. Handshaking signals are used to transfer data between devices whose data transfer speeds are not same. For example, computer can send data to the printer with large speed but printer can't accept data and print data with this rate. So computer has to send data with the speed with which printer can accept. This type of data transfer is achieved by using handshaking signals alongwith data signals. Fig. 7.3 shows data transfer between computer and printer using handshaking signals.

These handshaking signals are used to tell computer whether printer is ready to accept the data or not. If printer is ready to accept the data then after sending data on data bus, computer uses another handshaking signal ($\overline{STB}$) to tell printer that valid data is available on the data bus.

The 8255 mode 1 which supports handshaking has following features.

1. Two ports (A and B) function as 8-bit I/O ports. They can be configured either as input or output ports.
2. Each port uses three lines from Port C as handshake signals. The remaining two lines of Port C can be used for simple I/O functions.
3. Input and output data are latched.
4. Interrupt logic is supported.

Mode 2 : Bi-directional I/O data transfer

This mode allows bi-directional data transfer (transmission and reception) over a single 8-bit data bus using handshaking signals. This feature is available only in Group A with Port A as the 8-bit bi-directional data bus; and $PC_3 - PC_7$ are used for handshaking purpose. In this mode, both inputs and outputs are latched. Due to use of a single 8-bit data bus for bi-directional data transfer, the data sent out by the CPU through Port A appears on the bus connecting it to the peripheral, only when the peripheral requests it. The remaining lines of Port C i.e. PC_0-PC_2 can be used for simple I/O functions. The Port B can be programmed in mode 0 or in mode 1. When Port B is programmed in mode 1, PC_0-PC_2 lines of Port C are used as handshaking signals.

7.5 Control Word Formats

A high on the RESET pin causes all 24 lines of the three 8-bit ports to be in the input mode. All flip-flops are cleared and the interrupts are reset. This condition is maintained even after the RESET goes low. The ports of the 8255 can then be programmed for any other mode by writing a single control word into the control register, when required.

For Bit Set/Reset Mode

Fig. 7.4 shows bit set/reset control word format.

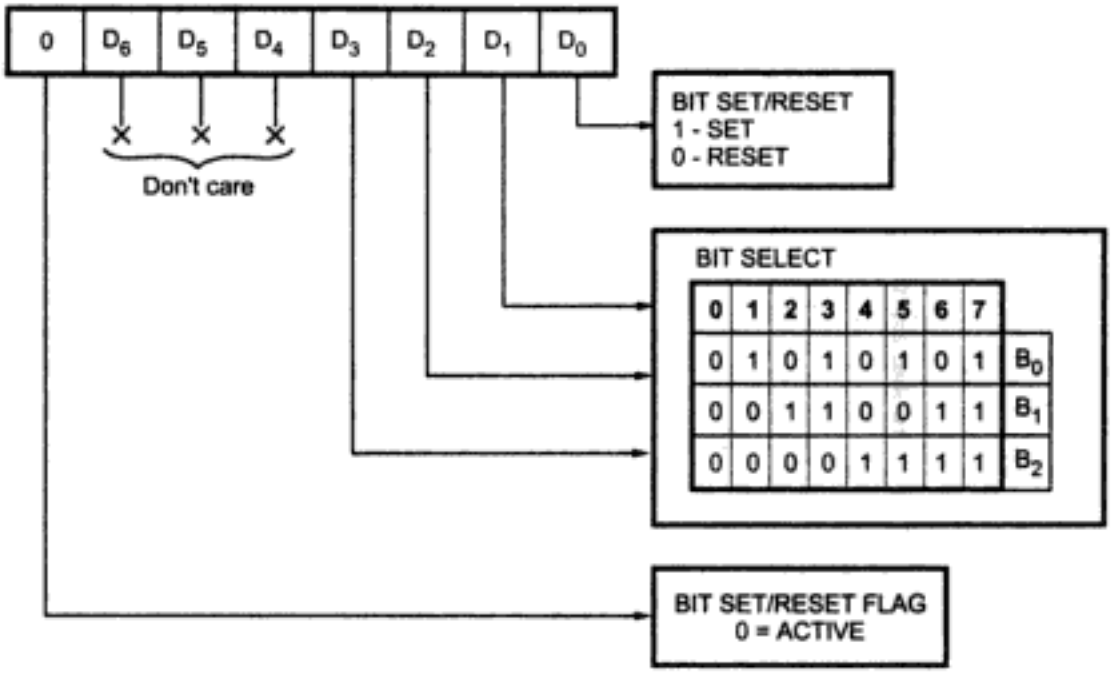


Fig. 7.4 Bit set/reset control word format

The eight possible combinations of the states of bits D_3 - D_1 (B_2 B_1 B_0) in the Bit Set-Reset format (BSR) determine particular bit in PC_0 - PC_7 being set or reset as per the status of bit D_0. A BSR word is to be written for each bit that is to be set or reset. For example, if bit PC_3 is to be set and bit PC_4 is to be reset, the appropriate BSR words that will have to be loaded into the control register will be, 0XXX0111 and 0XXX1000, respectively, where X is don't care.

The BSR word can also be used for enabling or disabling interrupt signals generated by Port C when the 8255 is programmed for Mode 1 or 2 operation. This is done by setting or resetting the associated bits of the interrupts. This is described in detail in next section.

For I/O Mode

The mode definition format for I/O mode is shown in Fig. 7.5. The control words for both, mode definition and Bit Set-Reset are loaded into the same control register, with bit D_7 used for specifying whether the word loaded into the control register is a mode

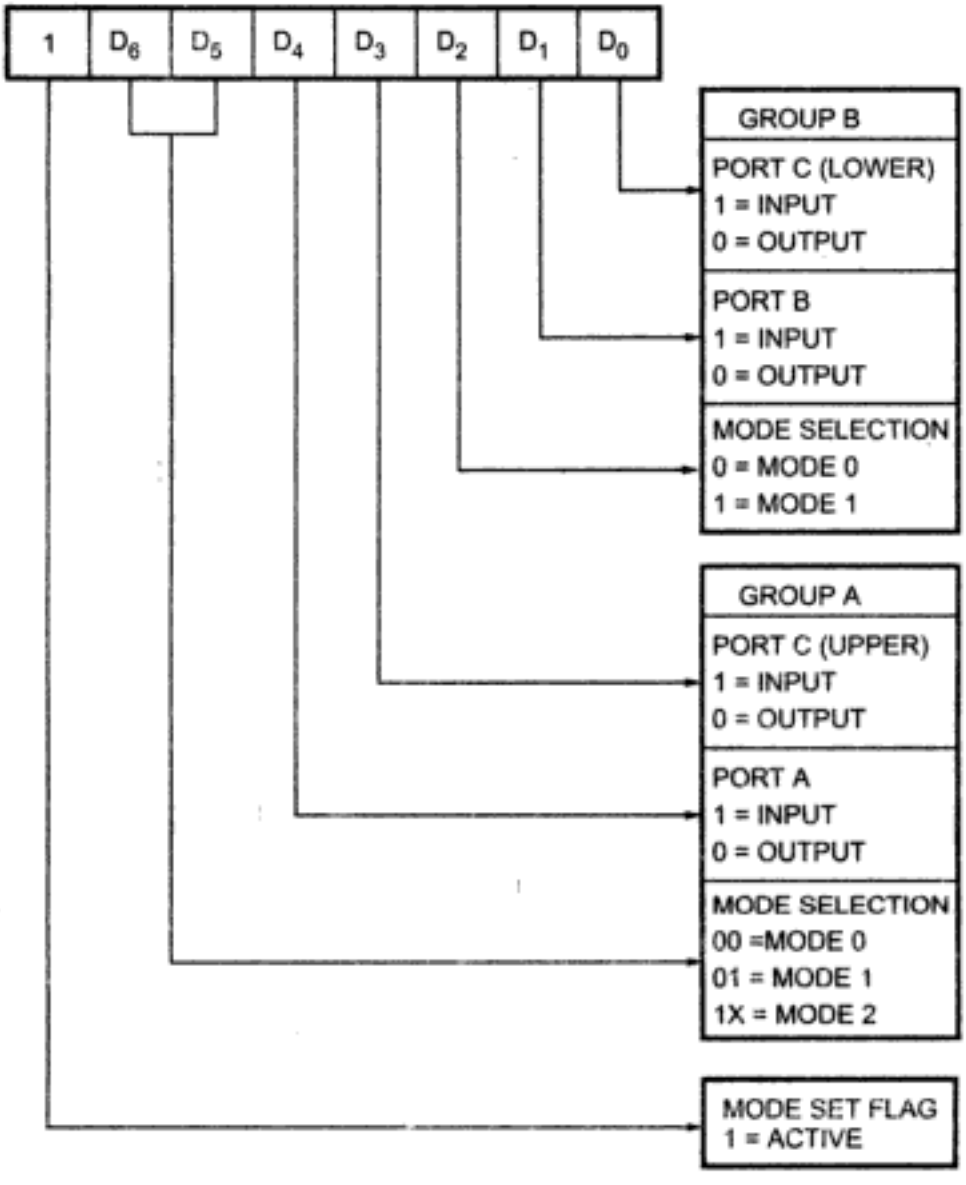


Fig. 7.5 8255 Mode definition format

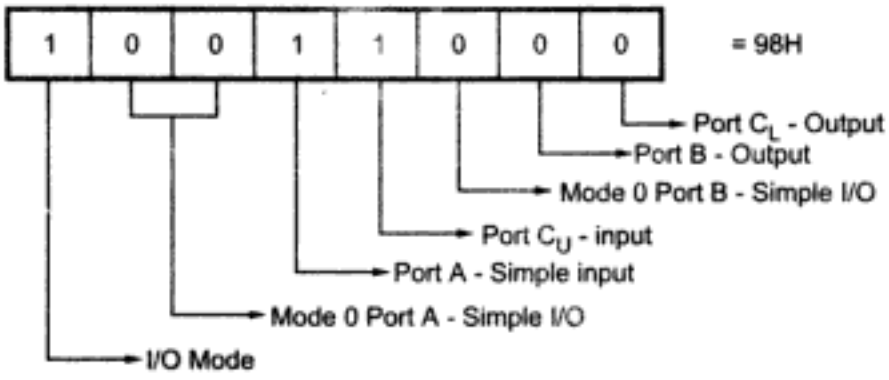
definition word or Bit Set-Reset word. If D_7 is high, the word is taken as a mode definition word, and if it is low, it is taken as a Bit Set-Reset word. The appropriate bits are set or reset depending on the type of operation desired, and loaded into the control register.

➠ **Example 1 :** *Write a program to initialize 8255 in the configuration given below :*

1. *Port A : Simple input*
2. *Port B : Simple output*
3. *Port* C_L *: Output*
4. *Port* C_U *: Input*

Assume address of the control word register of 8255 is 83H.

Solution :



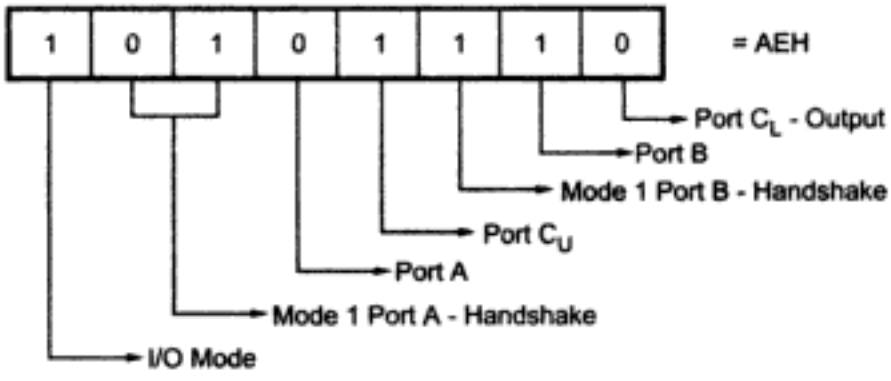

```
Source program:  MOV AL,98H   ; Load control word
                 OUT 83H,AL   ; Send control word
```

➠ **Example 2 :** *Write a program to initialize 8255 in the configuration given below :*

1. *Port A : Output with handshake*
2. *Port B : Input with handshake*
3. *Port C_L : Output*
4. *Port C_U : Input*

Assume address of the control word register of 8255 is 23H.

Solution :




```
Source program:  MOV AL,0AEH  ; Load control word
                 OUT 23H,AL   ; Send control word
```

Program : Blink port C bit 0 of 8255.

Program Statement :

Write a program to blink Port C bit 0 of the 8255. Assume address of control word register of 8255 is 83H. Use Bit Set/Reset mode.

Hidden page

7.6 8255 Programming and Operation

7.6.1 Programming in Mode 0

The Ports A, B and C can be configured as simple input or output ports by writing the appropriate control word in the control word register. In the control word, D_7 is set to '1' (to define a mode set operation) and D_6, D_5 and D_2 are all set to '0' to configure all the ports in Mode 0 operation. The status of bits D_4, D_3, D_1 and D_0 then determine (refer to Fig. 7.5) whether the corresponding ports are to be configured as Input or Output.

For example in mode 0, if Port A and Port B are to operate as output ports with Port C lower as input, and Port C upper as output, the control word that will have to be loaded into the control register will be as follows.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
1	0	0	0	0	0	0	1	=	81H

As mentioned earlier, this mode provides simple input and output operations for each of the three ports. No handshaking is required, data is simply written to or read from a specified port.

Input Mode : Fig. 7.6 shows the timing diagram for mode 0 input mode.

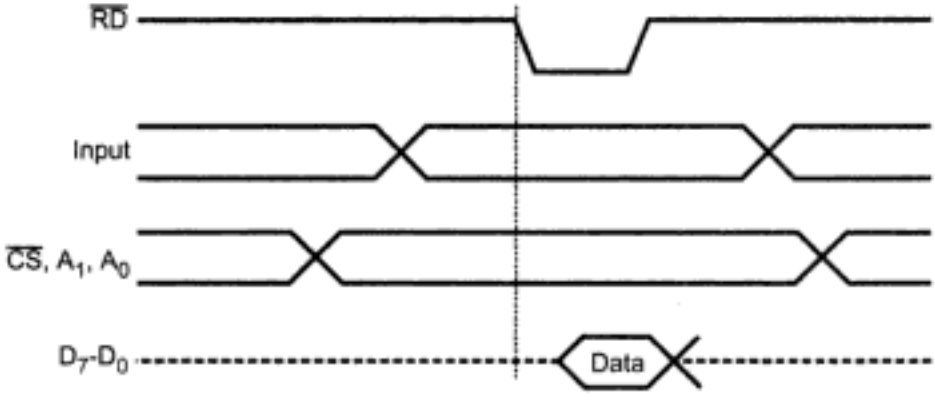


Fig. 7.6 Timing diagram for mode 0 input mode

After initialization of 8255 in the input mode 0, CPU can read data through the input port by initiating read command with proper port address. Read command activates $\overline{RD}$ signal. Upon activation of $\overline{RD}$ signal CPU reads the data from the selected input port into the CPU register.

Output Mode : Fig. 7.7 shows the timing diagram for mode 0 output mode.

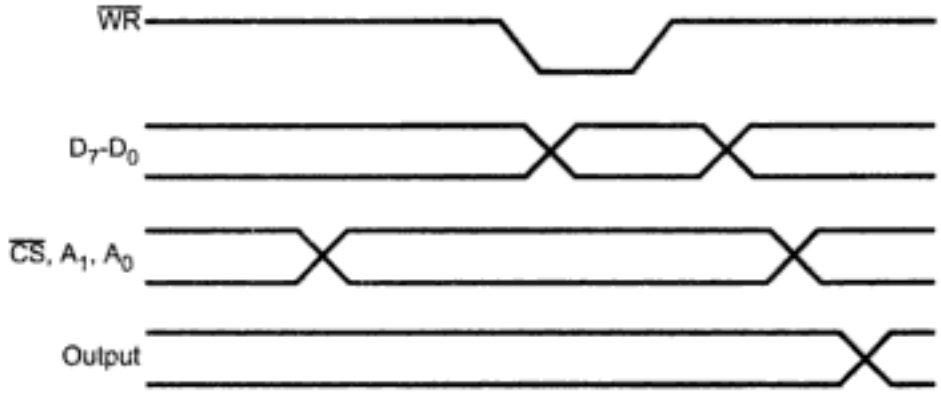


Fig. 7.7 Timing diagram for mode 0 output mode

After initialization of 8255 in the output mode 0, CPU can write data into the output port by initiating write command with proper port address. CPU sends data on the data bus and upon activation of $\overline{WR}$ signal, data on the data bus gets latched on the selected output port.

Mode 0 Configurations :

A		B		GROUP A			GROUP B	
D_4	D_3	D_1	D_0	PORT A	PORT C (Upper)	#	PORT B	PORT C (Lower)
0	0	0	0	OUTPUT	OUTPUT	0	OUTPUT	OUTPUT
0	0	0	1	OUTPUT	OUTPUT	1	OUTPUT	INPUT
0	0	1	0	OUTPUT	OUTPUT	2	INPUT	OUTPUT
0	0	1	1	OUTPUT	OUTPUT	3	INPUT	INPUT
0	1	0	0	OUTPUT	INPUT	4	OUTPUT	OUTPUT
0	1	0	1	OUTPUT	INPUT	5	OUTPUT	INPUT
0	1	1	0	OUTPUT	INPUT	6	INPUT	OUTPUT
0	1	1	1	OUTPUT	INPUT	7	INPUT	INPUT
1	0	0	0	INPUT	OUTPUT	8	OUTPUT	OUTPUT
1	0	0	1	INPUT	OUTPUT	9	OUTPUT	INPUT
1	0	1	0	INPUT	OUTPUT	10	INPUT	OUTPUT
1	0	1	1	INPUT	OUTPUT	11	INPUT	INPUT
1	1	0	0	INPUT	INPUT	12	OUTPUT	OUTPUT
1	1	0	1	INPUT	INPUT	13	OUTPUT	INPUT
1	1	1	0	INPUT	INPUT	14	INPUT	OUTPUT
1	1	1	1	INPUT	INPUT	15	INPUT	INPUT

7.6.2 Programming in Mode 1 (Input / Output with Handshake)

Both Group A and Group B can operate in Mode 1, either together, or individually, with each port containing an 8-bit latched Input or Output data port, and a 4-bit port which is used for control and status of the 8-bit port.

When Port A is to be programmed as an input port, PC_3 , PC_4 and PC_5 are used for control. PC_6 and PC_7 are not used and can be Input or Output, as programmed by bit D_3 of the control word. When Port A is programmed as an output port, PC_3 , PC_6, and PC_7 are used for control and PC_4 and PC_5 can be Input or Output, as programmed by bit D_3, of the control word.

When port B is to be programmed as an input or output port, PC_0, PC_1 and PC_2 are used for control.

Mode 1 Input Control Signals :

1. $\overline{STB}$ (Strobe Input) :

This is an active low input signal for 8255 and output signal for the input device. The input device activates this signal to indicate CPU that the data to be read is already sent on the port lines of 8255 port. Upon activation of this signal 8255 loads the data from the input port lines into the input buffer of that port.

2. IBF (Input Buffer Full) :

This is an active high output signal for 8255 and an input signal for input device. This signal is generated by 8255 in response to $\overline{STB}$ signal as an acknowledgment to input device. It also indicates to the input device that the input buffer is full and it is not ready to accept next byte from the input device. Therefore input device sends data on the port lines only when IBF signal is not active. The IBF signal is deactivated when CPU reads the data from input buffer of the respective port by activation of $\overline{RD}$ signal.

3. INTR (Interrupt Request) :

This is an active high output signal generated by 8255. A 'high' on this output can be used to interrupt the CPU when an input device is requesting service. The 8255 sets the INTR when $\overline{STB}$ signal is 'one', IBF signal is 'one' and INTE is 'one', indicating CPU that the data from the input device is available in the input buffer. This signal is reset by the falling edge of the $\overline{RD}$ signal i.e. immediately after reading the data from the input buffer.

INTE (Interrupt Enable) flip-flop is used to enable or disable INTR (Interrupt request) signal. If INTE flip-flop is set, the interrupt request is generated depending on the status of $\overline{STB}$ and IBF signals. If INTE flip-flop is reset, the interrupt request is not generated, allowing masking facility for the interrupt.

Mode 1 : Port A Input Operation

Fig. 7.8 (a) shows Port A as an input port along with the control word and control signals (for handshaking with a peripheral). When the control word (as in Fig. 7.8 (a) is loaded into the control register, Group A is configured in Mode 1 with Port A as an input

port. Port A can accept parallel data from a peripheral (like a keyboard) and this data can be read by the CPU. The peripheral first loads data into Port by making the $\overline{STB}_A$ input low. This latches the data placed by the peripheral on the common data bus into Port A. Port A acknowledges reception of data by making IBF_A (Input Buffer Full) high. IBFA is set when the $\overline{STB}_A$ input is made low, as shown in Fig. 7.8 (b).

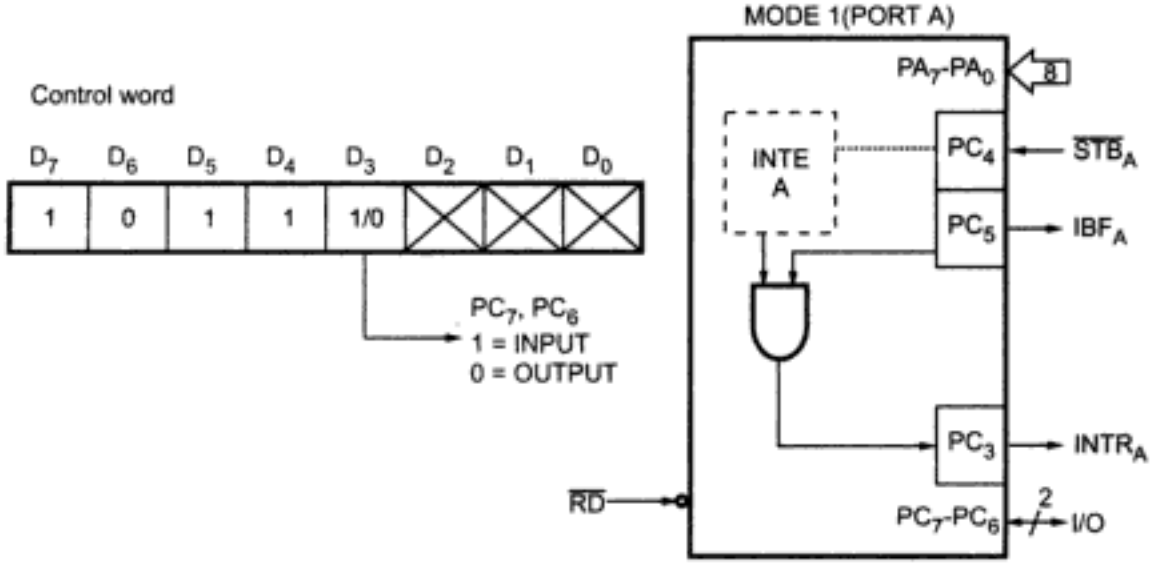


Fig. 7.8 (a) Port A in mode 1

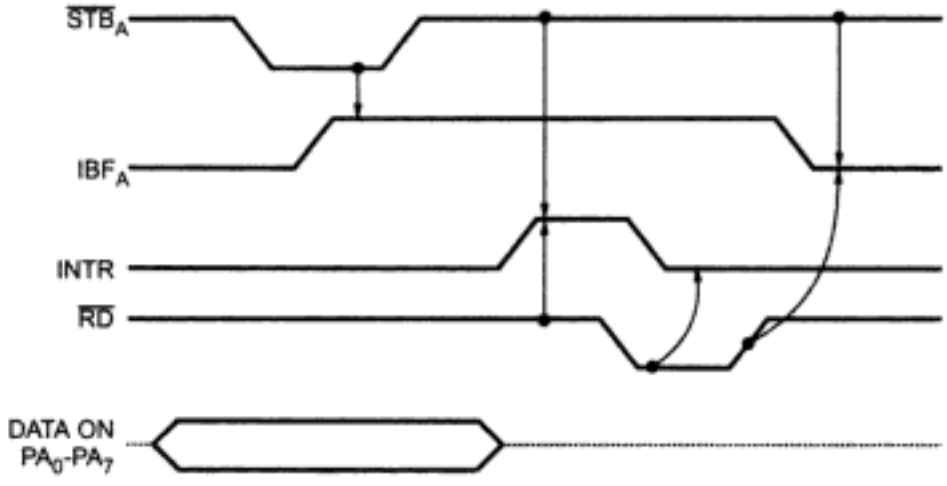


Fig. 7.8 (b) Timing diagram for port A in mode 1

$INTR_A$ is an active high output signal which can be used to interrupt the CPU so that the CPU can suspend its current operation and read the data written into Port A by the peripheral. $INTR_A$ can be enabled or disabled by the $INTE_A$ flip-flop which is controlled by Bit Set-Reset operation of PC_4. $INTR_A$ is set (if enabled by setting the $INTE_A$ flip-flop) after the $\overline{STB}_A$ has gone high again, and if IBF_A is high.

On receipt of the interrupt, the CPU can be forced to read Port A. The falling edge of the $\overline{RD}$ input resets IBFA and it goes low. This can be used to indicate to the peripheral that the input buffer is empty and that data can again be loaded into it.

Mode 1 : Port B Input operation

Fig. 7.9 shows Port B as an input port (when in Mode 1). The timing diagram and operation of Port B is similar to that of Port A except that it uses different bits of Port C for control. $INTE_B$ is controlled by Bit Set/Reset of PC_2.

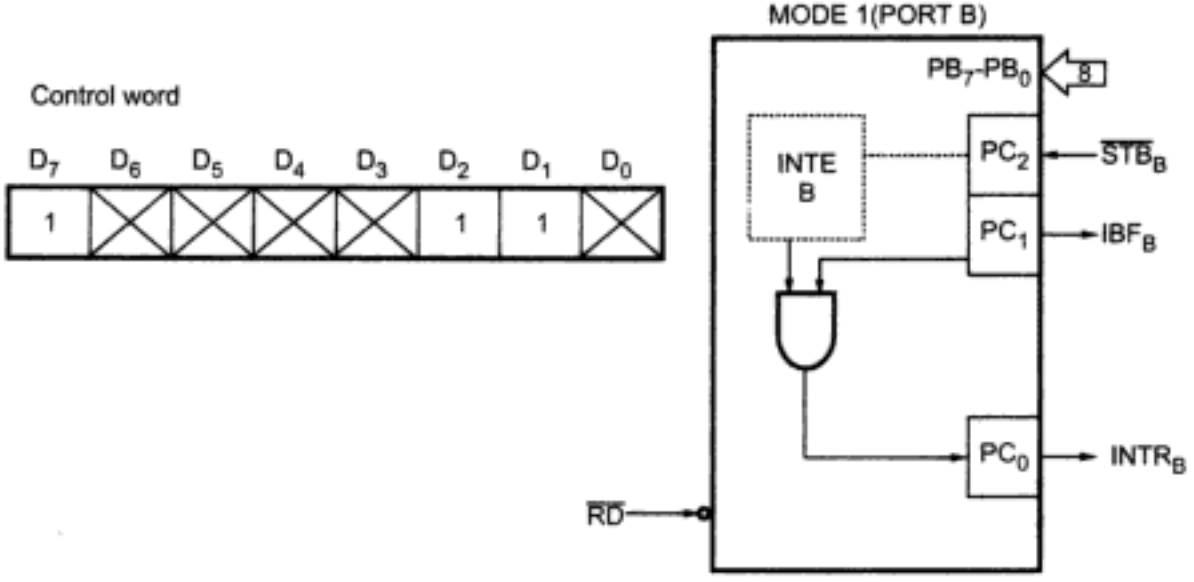


Fig. 7.9 Port B in mode 1

If the CPU is busy with other system operations, it can read data from the input port when it is interrupted. This is often called Interrupt driven I/O. However, if the CPU is otherwise not busy with other jobs, it can continuously poll (read) the status word to check for an IBF_A. This is often called Program Controlled I/O. The status word is accessed by reading Port C (A_1 A_0 must be 10, $\overline{RD}$ and $\overline{CS}$ must be low). The status word format when Ports A and B are input ports in Mode 1, is shown in Fig. 7.10.

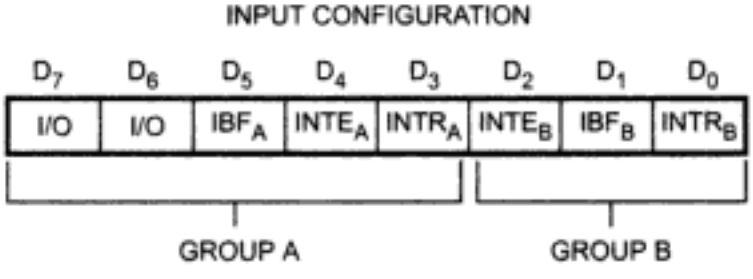


Fig. 7.10 Mode 1 status word (Input)

Mode 1 : Output control signals

1. $\overline{OBF}$ (Output Buffer Full) :

This is an active low output signal for 8255 and input signal for the output device. The 8255 activates this signal to indicate output device that data is available on the output port. Upon activation of $\overline{OBF}$ signal, output device reads data from the output port and acknowledges it by $\overline{ACK}$ signal. The $\overline{OBF}$ signal is activated at the rising edge of the $\overline{WR}$ signal and de-activated at the falling edge of the $\overline{ACK}$ signal.

2. $\overline{ACK}$ (Acknowledge Input) :

This is an active low input signal for 8255 and output signal for the output device. The output device generates this signal to indicate 8255 that the data from port A or Port B has been accepted.

3. INTR (Interrupt Request) :

This is an active high output signal generated by 8255. A 'high' on this output can be used to interrupt the CPU when an output device has accepted data transmitted by the CPU. The 8255 sets the INTR when $\overline{ACK}$ signal is 'one', $\overline{OBF}$ is 'one' and INTE is 'one', indicating that the output device is ready to accept next data byte. This signal is reset by the falling edge of the $\overline{WR}$ signal i.e. immediately after sending the data to the output port.

INTE (Interrupt Enable) flip-flop is used to enable or disable INTR (Interrupt Request) signal. If INTE flip-flop is set, the interrupt request is generated depending on the status of $\overline{ACK}$ and $\overline{OBF}$ signals. If INTE flip-flop is reset, the interrupt request is not generated, allowing masking facility for the interrupt.

Mode 1 : Port A output operation

Fig. 7.11 (a) shows Port A configured as an output port (When in Mode 1) along with the control word and control signals (for handshaking with a peripheral). When the control word (as in Fig. 7.11 (a)) is loaded into the control register, Group A is configured in Mode 1 with Port A as an output port. The CPU can send data to a peripheral (like a display device) through Port A of the 8255.

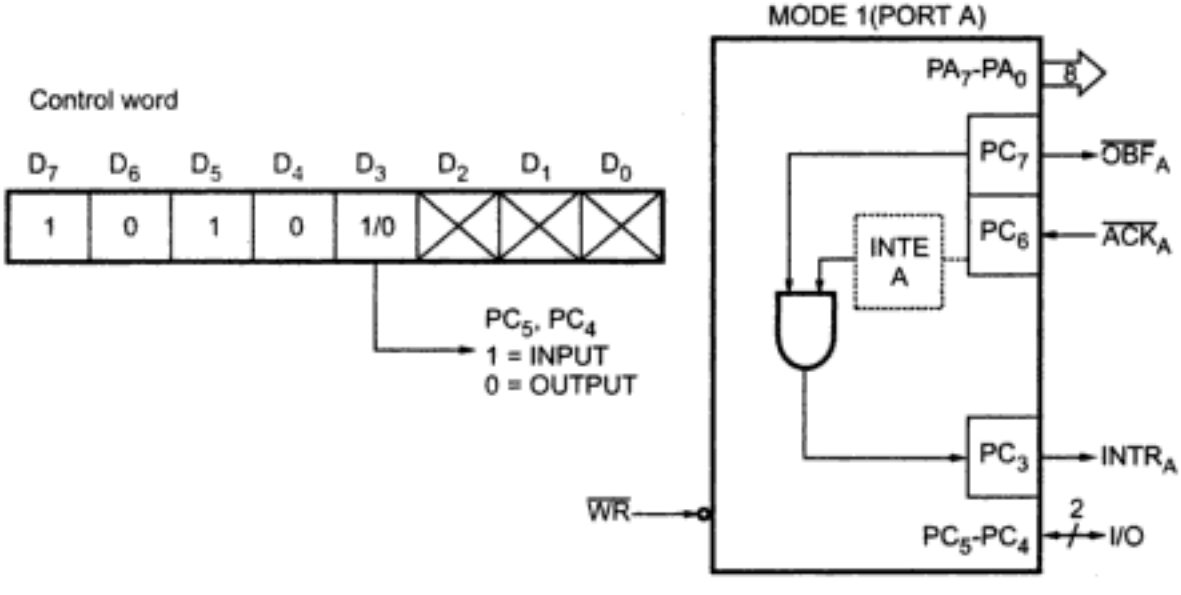


Fig. 7.11 (a) Port A in mode 1

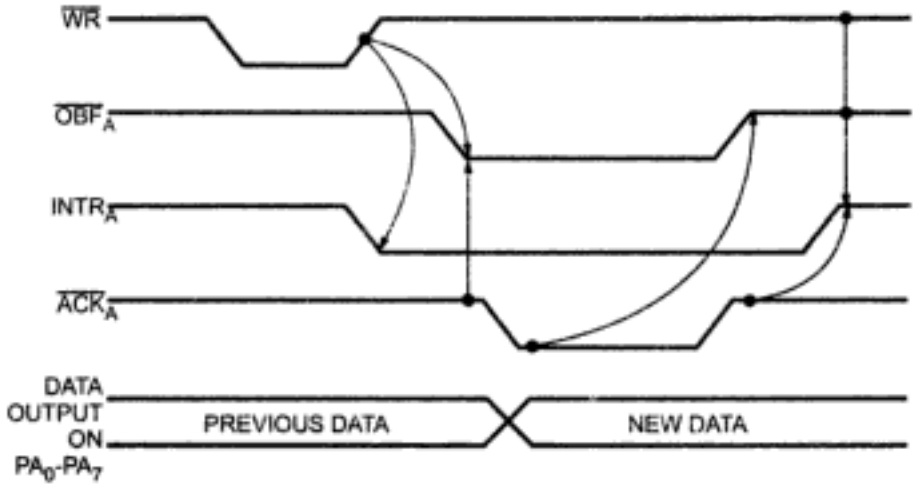


Fig. 7.11 (b) Port A in mode 1 (Output)

The $\overline{OBF}_A$ output (Output Buffer Full) goes low on the rising edge of the $\overline{WR}$ signal (when the CPU writes data into the 8255). The $\overline{OBF}_A$ cutput from 8255 can be used as a strobe input to the peripheral to latch the contents of Port A. The peripheral responds to the receipt of data by making the $\overline{ACK}_A$ input of the 8255 low, thus acknowledging that it has received the data sent by the CPU through Port A. The $\overline{ACK}_A$ low sets the OBF_A signal, which can be polled by the CPU through OBF_A of the status word to load the next data when it is high again.

$INTR_A$ is an active high output of the 8255 which is made high (if the associated $INTE_A$ flip-flop is set) when $\overline{ACK}_A$ is made high again by the peripheral, and when $\overline{OBF}_A$ goes high again (see timing diagram in Fig. 7.11 (b)). It can be used to interrupt the CPU whenever the output buffer is empty. It is reset by the falling edge of $\overline{WR}$ when the CPU writes data onto Port A. It can be enabled or disabled by writing a '1' or a '0' respectively to PC_6 in the BSR mode.

Mode 1 : Port B output operation

Fig. 7.12 shows Port B as an output port when in Mode 1. The operation of Port B is similar to that of Port A. $INTR_A$ is controlled by writing a '1' or a '0' to PC_2 in the BSR mode.

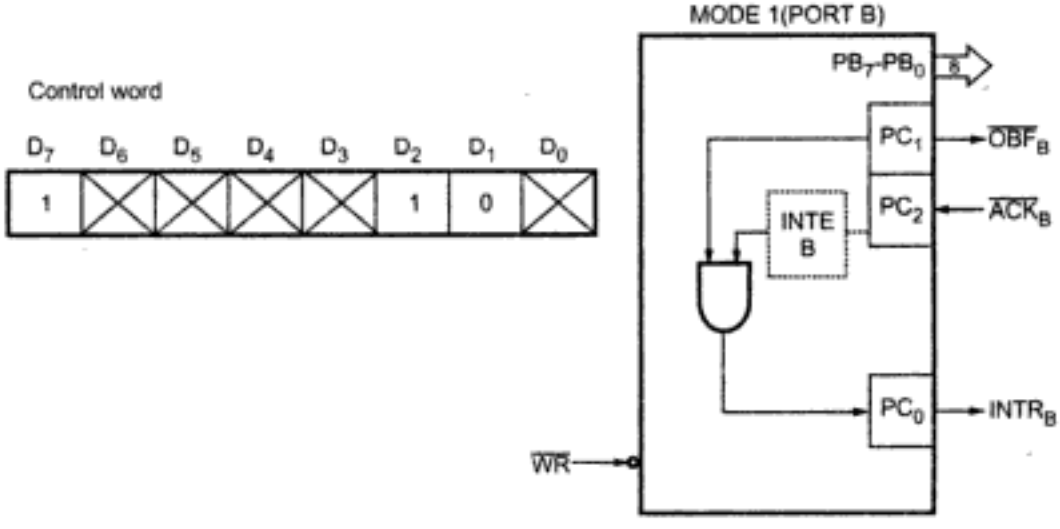


Fig. 7.12 Port B in mode 1 (Output)

The status word is accessed by issuing a Read to Port C. The format of the status word when Ports A and B are Output ports in Mode 1 is shown in Fig. 7.13.

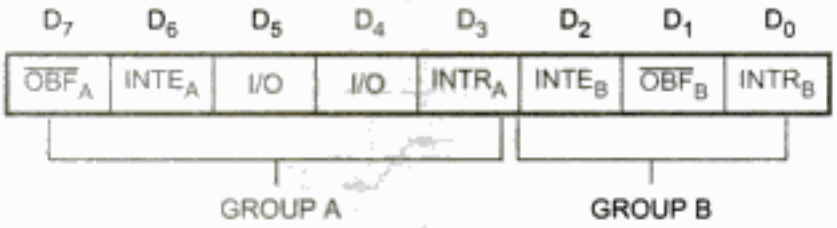


Fig. 7.13 Mode 1 status word (Output)

7.6.3 Programming in Mode 2 (Strobes Bi-directional Bus I/O)

When the 8255 is operated in Mode 2 (by loading the appropriate control word), Port A can be used as a bi-directional 8-bit I/O bus using for handshaking. Port B can be programmed in Mode 0 or in Mode 1. When Port B is programmed in mode 1, PC_0 - PC_2 lines of Port C are used as handshaking signals.

Fig. 7.14 shows the control word that should be loaded into the control port to configure 8255 in Mode 2.

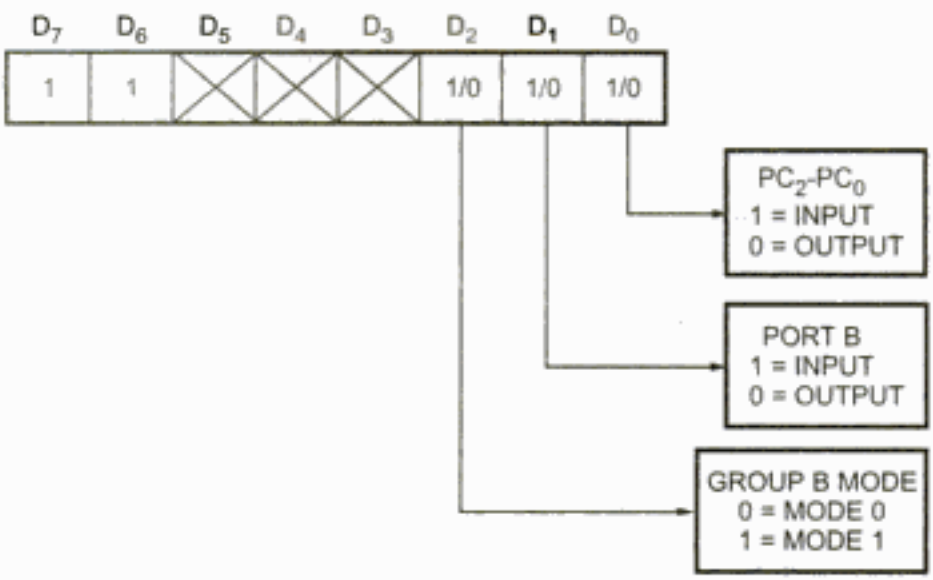


Fig. 7.14 Mode 2 control word

Mode 2 : Control signals

INTR (Interrupt Request) : A 'high' on this output can be used to interrupt the CPU for input or output operations.

Output Control Signals :

$\overline{OBF}_A$ (Output Buffer Full)

This is an active low output which indicates that the CPU has written data into Port A.

$\overline{ACK}_A$ (Acknowledge)

This is an active low input signal (generated by the peripheral) which enables the tri-state output buffer of Port A and makes Port A data available to the peripheral. In Mode 2, Port A outputs are in tri-state until enabled.

INTE 1

This is the flip-flop associated with Output Buffer Full. INTE 1 can be used to enable or disable the interrupt by setting or resetting PC_6 in the BSR Mode.

Input Control Signals :

$\overline{STB}$ (Strobe Input)

This is an active low input signal which enables Port A to latch the data available at its input.

IBF (Input Buffer Full Flip-Flop)

This is an active high output which indicates that data has been loaded into the input latch of Port A.

INTE 2

This is an Interrupt enable flip-flop associated with Input Buffer Full. It can be controlled by setting or resetting PC_4 in the BSR Mode.

Mode 2 : Port A operation.

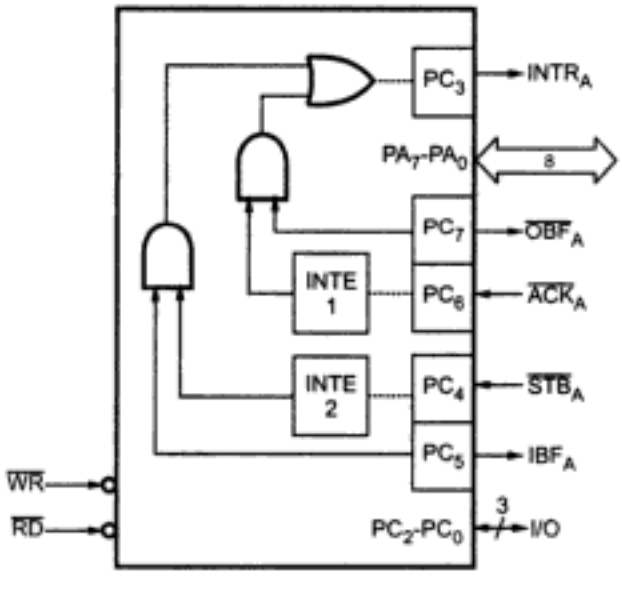


Fig. 7.15 Mode 2 operation

Fig. 7.15 shows Port A and associated control signals when 8255 is in Mode 2. Interrupts are generated for both output and input operations on the same $INTR_A$ (PC_3) line.

Status Word In Mode 2

The status word for Mode 2 (accessed by reading Port C) is shown in Fig. 7.16. D_7 - D_3 of the status word carry information about $\overline{OBF}_A$, $INTE_1$, IBF_A, $INTE_2$, $INTR_A$. The status of the bits D_2 - D_0 depend on the mode setting of Group B. If B is programmed in Mode 0, D_2 - D_0 are the same as PC_2 - PC_0 (simple I/O); however if

B is in Mode 1, D_2 - D_0 carry information about the control signals for Port B (as in Fig. 7.10, or Fig. 7.13), depending upon whether Port B is an Input port or Output port respectively.

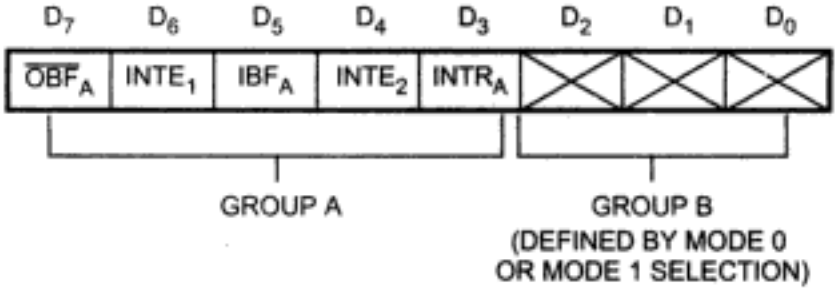


Fig. 7.16 Status word for mode 2

Mode Definition Summary

	MODE 0	
	IN	OUT
PA_0	IN	OUT
PA_1	IN	OUT
PA_2	IN	OUT
PA_3	IN	OUT
PA_4	IN	OUT
PA_5	IN	OUT
PA_6	IN	OUT
PA_7	IN	OUT

	MODE 0	
	IN	OUT
PB_0	IN	OUT
PB_1	IN	OUT
PB_2	IN	OUT
PB_3	IN	OUT
PB_4	IN	OUT
PB_5	IN	OUT
PB_6	IN	OUT
PB_7	IN	OUT
PC_0	IN	OUT
PC_1	IN	OUT
PC_2	IN	OUT
PC_3	IN	OUT
PC_4	IN	OUT
PC_5	IN	OUT
PC_6	IN	OUT
PC_7	IN	OUT

MODE 1	
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT

MODE 1	
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
IN	OUT
$INTR_B$	$INTR_B$
IBF_B	OBF_B
STB_B	ACK_B
$INTR_A$	$INTR_A$
STB_A	I/O
IBF_A	I/O
I/O	ACK_A
I/O	OBF_A

MODE 2
GROUP A ONLY
↔
↔
↔
↔
↔
↔
↔
↔

MODE 2
GROUP A ONLY
–
–
–
–
–
–
–
–
I/O
I/O
I/O
$INTR_A$
STB_A
IBF_A
ACK_A
OBF_A

Mode0 or Mode1 Only

7.7 Interfacing 8255 to 8086 in I/O Mapped I/O Mode

The 8086 has four special instructions IN, INS, OUT, and OUTS to transfer data through the input/output ports in I/O mapped I/O system. M/$\overline{IO}$ signal is always low when 8086 is executing these instructions. So M/$\overline{IO}$ signal is used to generate separate addresses for, memory and input/output. Only 256 (2^8) I/O addresses can be generated when direct addressing method is used. By using indirect address method this range can be extended upto 65536 (2^{16}) addresses.

Fig. 7.17 shows the interfacing of 8255 with 8086 in I/O mapped I/O technique. Here, $\overline{RD}$ and $\overline{WR}$ signals are activated when M/$\overline{IO}$ signal is low, indicating I/O bus cycle. Only lower data bus (D_0 – D_7) is used as 8255 is 8-bit device. Reset out signal from clock generator is connected to the Reset signal of the 8255. In case of interrupt driven I/O INTR signal (PC_3 or PC_0) from 8255 is connected to INTR input of 8088.

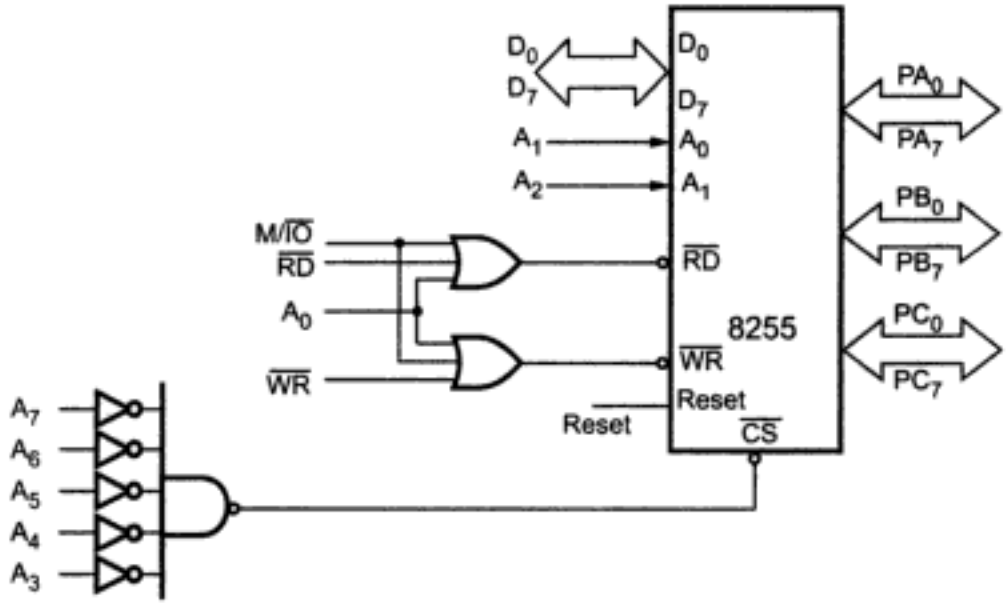


Fig. 7.17 I/O mapped I/O

I/O Map :

Port / control Register	Address lines								Address
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Port A	0	0	0	0	0	0	0	0	00H
Port B	0	0	0	0	0	0	1	0	02H
Port C	0	0	0	0	0	1	0	0	04H
Control register	0	0	0	0	0	1	1	0	06H

Note : It is assumed that the direct addressing is used.

7.8 Interfacing 8255 to 8086 in Memory Mapped I/O

In this type of I/O interfacing, the 8086 uses 20 address lines to identify an I/O device; an I/O device is connected as if it is a memory register. The 8086 uses same control signals and instructions to access I/O as those of memory. Fig. 7.18 shows the interfacing of 8255 with 8086 in memory mapped I/O technique. Here $\overline{RD}$ and $\overline{WR}$ signals are activated when $M/\overline{IO}$ signal is high, indicating memory bus cycle. Address lines A_0 - A_1 are used by 8255 for internal decoding. To get absolute address, all remaining address lines (A_3 - A_{19}) are used to decode the address for 8255. Other signal connections are same as in I/O mapped I/O.

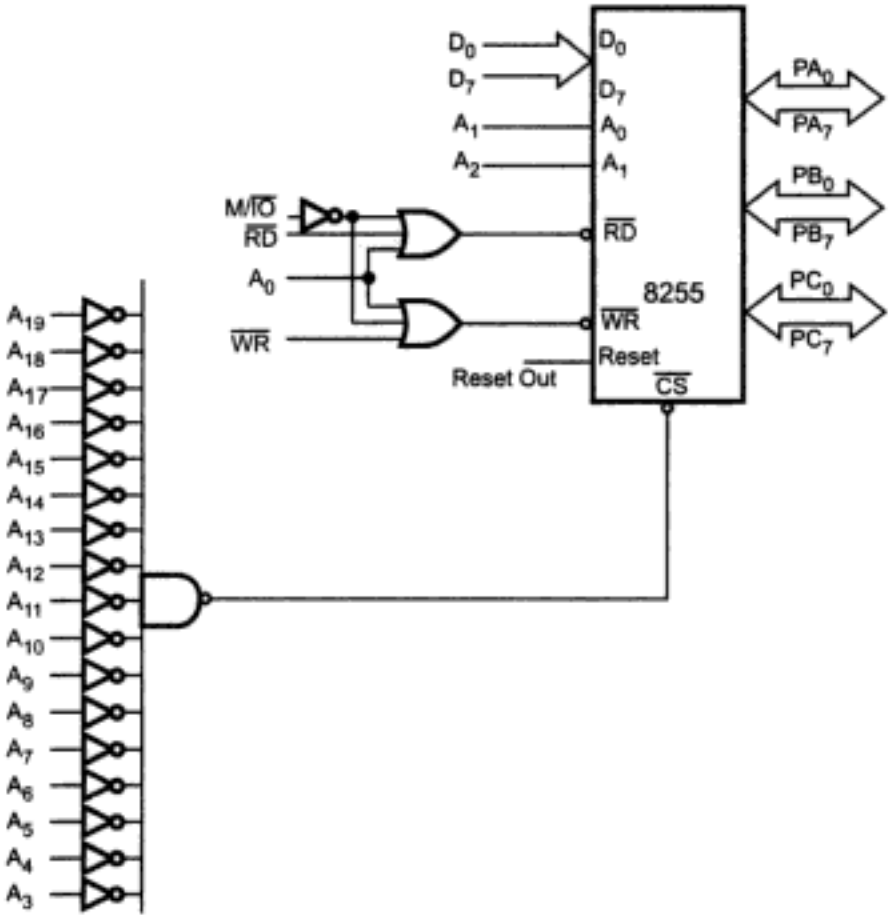


Fig. 7.18 Memory mapped I/O

I/O Map :

Register	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
Port A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
Port B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	00002H
Port C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	00004H
Control register	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	00006H

Hidden page

The IC 1408 consists of a reference current amplifier, an R/2R ladder and eight high speed current switches. It has eight input data lines A_1 (MSB) through A_8 (LSB) which control the positions of current switches.

It requires 2 mA reference current for full scale input and two power supplies V_{CC} = + 5 V and V_{EE} = – 15 V (V_{EE} can range from – 5 V to – 15 V).

The voltage V_{ref} and resistor R_{14} determines the total reference current source and R_{15} is generally equal to R_{14} to match the input impedance of the reference current amplifier.

Fig. 7.20 shows a typical circuit for IC 1408.

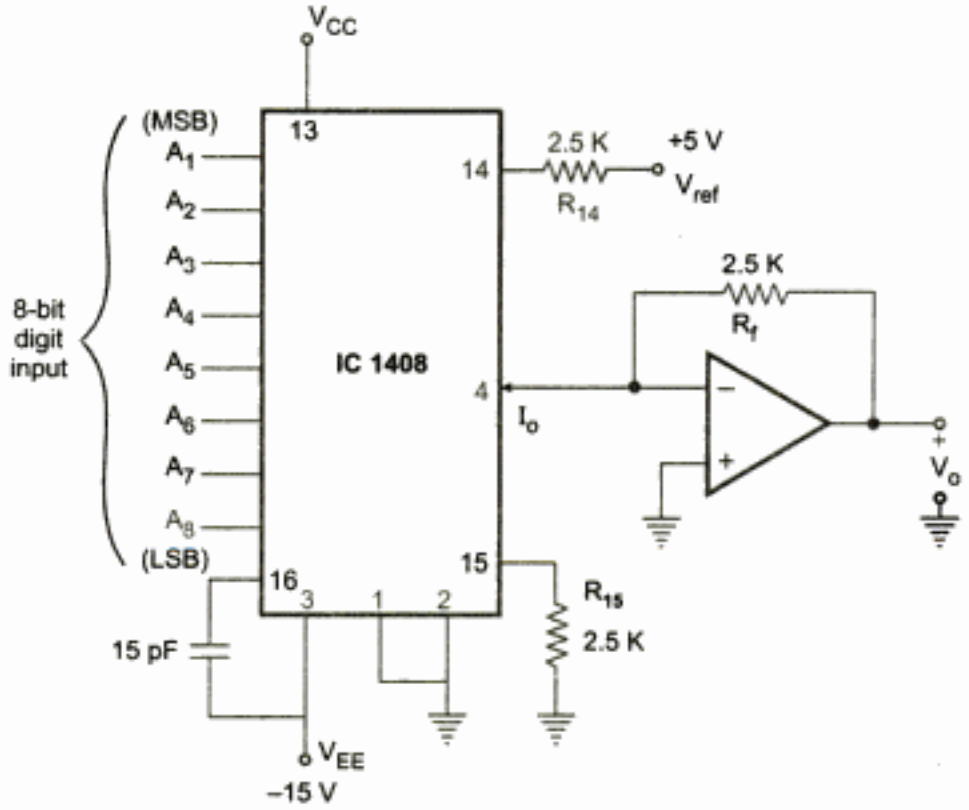


Fig. 7.20 Typical circuit for IC 1408

The output current I_o can be given as

$$I_o = \frac{V_{ref}}{R_{14}}\left(\frac{A_1}{2} + \frac{A_2}{4} + \frac{A_3}{8} + \frac{A_4}{16} + \frac{A_5}{32} + \frac{A_6}{64} + \frac{A_7}{128} + \frac{A_8}{256}\right) \quad ...(1)$$

Note : Input A_1 through A_8 can be either 0 or 1. Therefore, for typical circuit full scale current can be given as,

$$I_o = \frac{5}{2.5\,K}\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}\right)$$

$$= \frac{2\,mA \times 255}{256} = 1.992\ mA$$

It shows that the full scale output current is always 1 LSB less than the reference current source of 2 mA. This output current is converted into voltage by I to V converter. The output voltage for full scale input can be given as

$$V_o = 1.992 \times 2.5 \text{ K}$$

$$= 4.98 \text{ V}$$

Note : The arrow on the pin 4 shows the output current direction. It is inward. This means that IC 1408 sinks current. At $(0000\,0000)_2$ binary input it sinks zero current and at $(1111\,1111)_2$ binary input it sinks 1.992 mA.

The circuit shown in the Fig. 7.20 gives output in the unipolar range. When digital input is 00H, the output voltage is 0 V and when digital input is FFH $(1111\,1111)_2$, the output voltage is + 5 V. This circuit can be modified to give bipolar output.

Fig. 7.21 shows the circuit for giving output in the bipolar range. Here, resistor R_B (5 K) is connected between V_{ref} and the output terminal of IC 1408. This gives a constant current source of 1 mA.

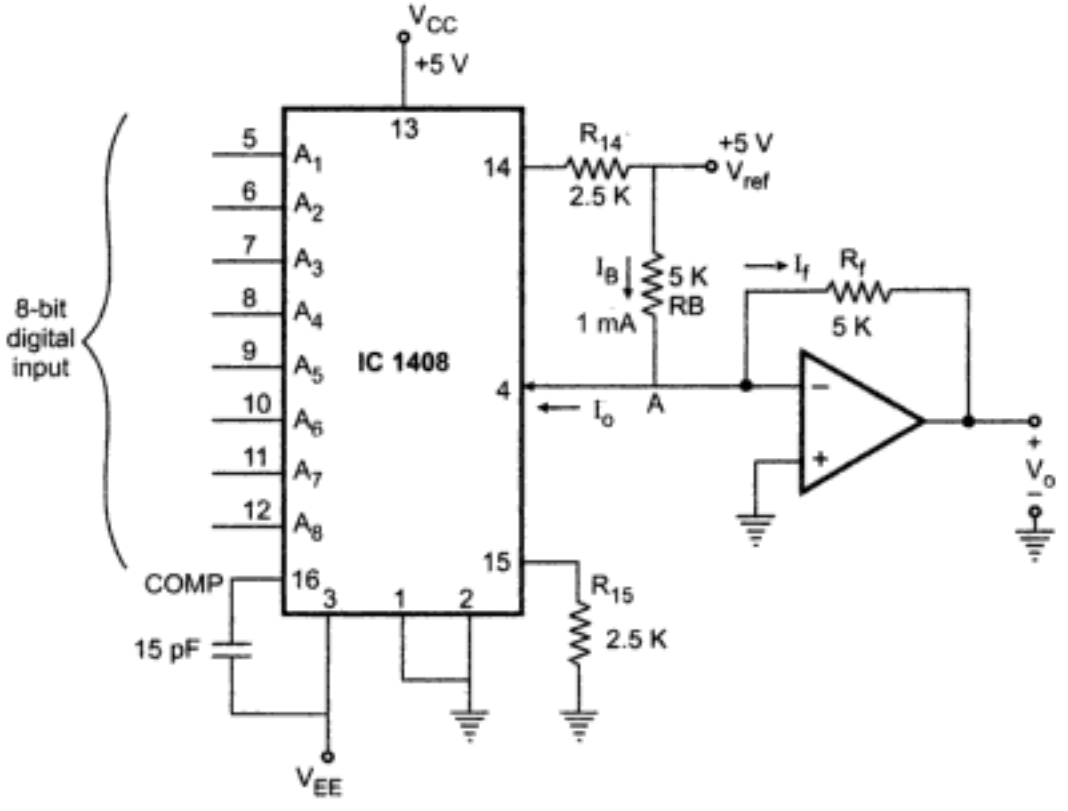


Fig. 7.21 Interfacing DAC in the bipolar range

The circuit operation can be observed for three conditions :

Condition 1 : For binary input (00H)

When binary input is 00H, the output current I_o at pin 4 is zero. Due to this current flowing through R_B (1 mA) flows through R_f giving $V_o = -5$ V.

Condition 2 : For binary input 80H

When binary input is 80H, the output current I_o at pin 4 is 1 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

Substituting values of I_B and I_o we get,

$$-(1\text{ mA}) + (1\text{ mA}) + I_f = 0$$

$$\therefore \quad I_f = 0$$

and therefore the output voltage is zero.

Condition 3 : For binary input FFH

When binary input is FFH, the output current I_o at pin 4 is 2 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

Substituting values of I_B and I_o we get,

$$-(1\text{ mA}) + (2\text{ mA}) + I_f = 0$$

$$\therefore \quad I_f = -1\text{ mA}$$

Therefore, the output voltage is + 5 V. In this way, circuit shown in the Fig. 7.21 gives output in the bipolar range.

Important Electrical Characteristics for IC 1408

- Reference current : 2 mA
- Supply voltage : + 5 V V_{CC} and – 15 V V_{EE}
- Setting time : 300 ns
- Full scale output current : 1.992 mA
- Accuracy : 0.19 %

7.9.2 DAC0830

The DAC0830 is an advanced CMOS 8-bit DAC designed to interface directly with the 8080, 8048, 8085, Z80, and other popular microprocessors. A deposited silicon-chromium R-2R resistor ladder network divides the reference current and provides the circuit with excellent temperature tracking characteristics (0.05% of Full Scale Range maximum linearity error over temperature). The circuit uses CMOS current switches and control logic to achieve low power consumption and low output leakage current errors. Special circuitry provides TTL logic input voltage level compatibility.

Double buffering feature allows this DAC to output a voltage corresponding to one digital word while holding the next digital word. This permits the simultaneous updating of any number of DACs.

The DAC0830 series (DAC0830/DAC0831/DAC0832) are the 8-bit members of a family of microprocessor-compatible DACs. For applications demanding higher resolution, the DAC1000 series (10-bits) and the DAC1208 and DAC1230 (12-bits) are available alternatives.

Features

- Double-buffered, single-buffered or flow-through digital data inputs.
- Easy interchange and pin-compatible with 12-bit DAC1230 series.
- Direct interface to all popular microprocessors.
- Built-in facility for zero adjustment.
- Works with ± 10 V reference voltage.
- Can be used in the voltage switching mode.
- Logic inputs which meet TTL voltage level specifications.
- Operates "STAND ALONE" (without up) if desired.
- Available in 20-pin small-outline or molded chip carrier package.

Pin Diagrams

Fig. 7.22 shows the pin diagram of DAC0830. The function of each pin is explained in Table 7.2.

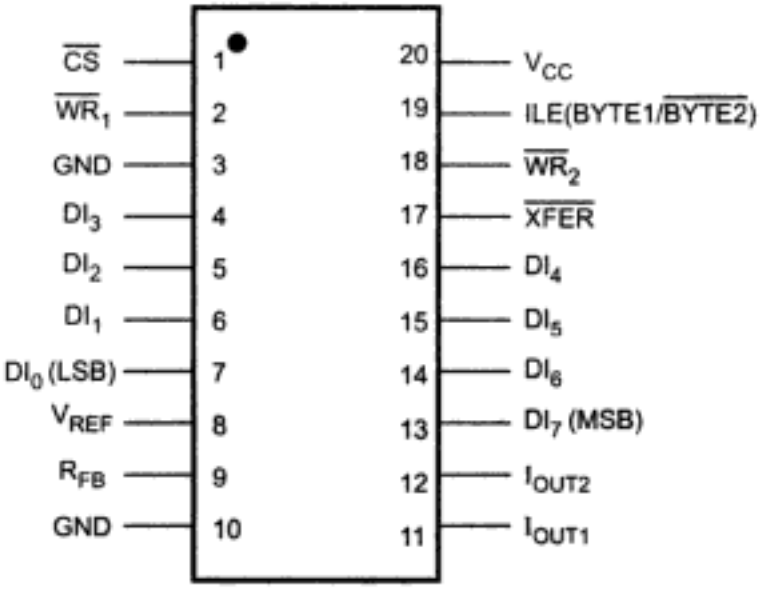


Fig. 7.22

Control Signals (All control signals level actuated)	
$\overline{CS}$:	Chip Select (active low). The $\overline{CS}$ in combination with ILE will enable $\overline{WR_1}$.
ILE :	Input Latch Enable (active high). The ILE in combination with $\overline{CS}$ enables $\overline{WR_1}$.
$\overline{WR_1}$:	Write 1. The active low $\overline{WR_1}$ is used to load the digital input data bits (DI) into the input latch. The data in the input latch is latched when $\overline{WR_1}$ is high. To update the input latch- $\overline{CS}$ and $\overline{WR_1}$ must be low while ILE is high.
$\overline{WR_2}$:	Write 2 (active low). This signal, in combination with $\overline{XFER}$, causes the 8-bit data which is available in the input latch to transfer to the DAC register.
$\overline{XFER}$:	Transfer control signal (active low). The $\overline{XFER}$ will enable $\overline{WR_2}$.

Table 7.2 Pin description

Other Pin Functions	
DI_0-DI_7 :	Digital Inputs. DI_0 is the least significant bit (LSB) and DI_7 is the most significant bit (MSB).
I_{OUT1} :	DAC Current Output 1. I_{OUT1} is a maximum for a digital code of all 1's in the DAC register, and is zero for all 0's in DAC register.
I_{OUT2} :	DAC Current Output 2. $I_{OUT1} + I_{OUT2}$ = constant (I full scale for a fixed reference voltage).
R_{fb} :	Feedback Resistor. The feedback resistor is provided on the IC chip for use as the shunt feedback resistor for the external op-amp which is used to provide an output voltage for the DAC. This on-chip resistor should always be used (not an external resistor) since it matches the resistors which are used in the on-chip R-2R ladder and tracks these resistors over temperature.
V_{REF} :	Reference Voltage Input. This input connects an external precision voltage source to the internal R-2R ladder. V_{REF} can be selected over the range of +10 V to –10 V. This is also the analog voltage input for a 4-quadrant multiplying DAC application.
V_{CC} :	Digital Supply Voltage. This is the power supply pin for the part. V_{CC} can be from +5 V_{DC} to +15 V_{DC}. Operation is optimum for +15 V_{DC}.
GND :	The pin 10 voltage must be at the same ground potential as I_{OUT1} and I_{OUT2} for current switching applications. Any difference of potential (V_{OS} pin 10) will result in a linearity change of $\frac{V_{OS}\ \text{pin}\ 10}{3\ V_{REF}}$ For example, if V_{REF} = 10 V and pin 10 is 9 mV offset from I_{OUT1} and I_{OUT2}, the linearity change will be 0.03%. Pin 3 can be offset ± 100 mV with no linearity change, but the logic input threshold will shift.

Functional Block Diagram

A most unique characteristic of this DAC is that the 8-bit digital input byte is double-buffered. This means that the data must transfer through two independently controlled 8-bit latching registers before being applied to the R-2R ladder network to change the analog output. The addition of a second register allows two useful control

Hidden page

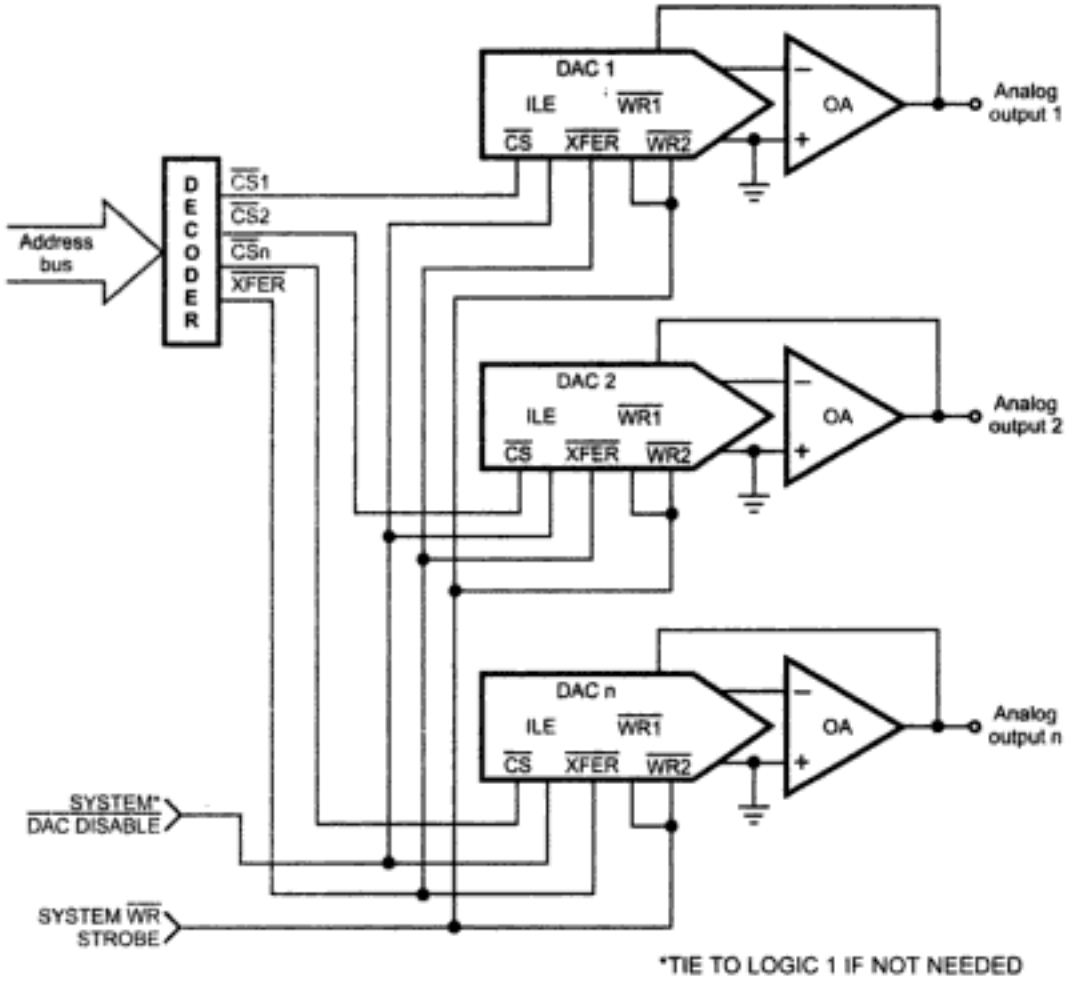


Fig. 7.24 Controlling multiple DACs

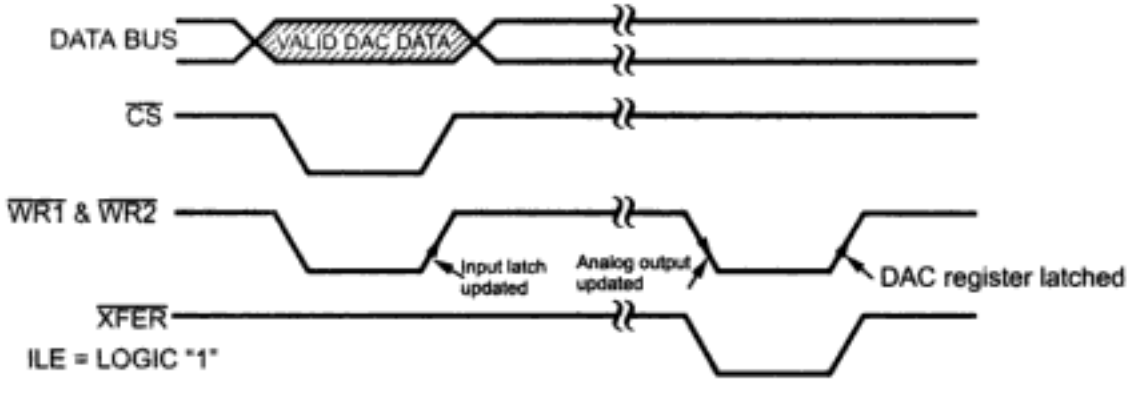


Fig. 7.25 Timing diagram

The ILE pin is an active high chip select which can be decoded from the address bus as a qualifier for the normal $\overline{CS}$ signal generated during a write operation. This can be used to provide a higher degree of decoding unique control signals for a particular DAC, and thereby create a more efficient addressing scheme.

Hidden page

Hidden page

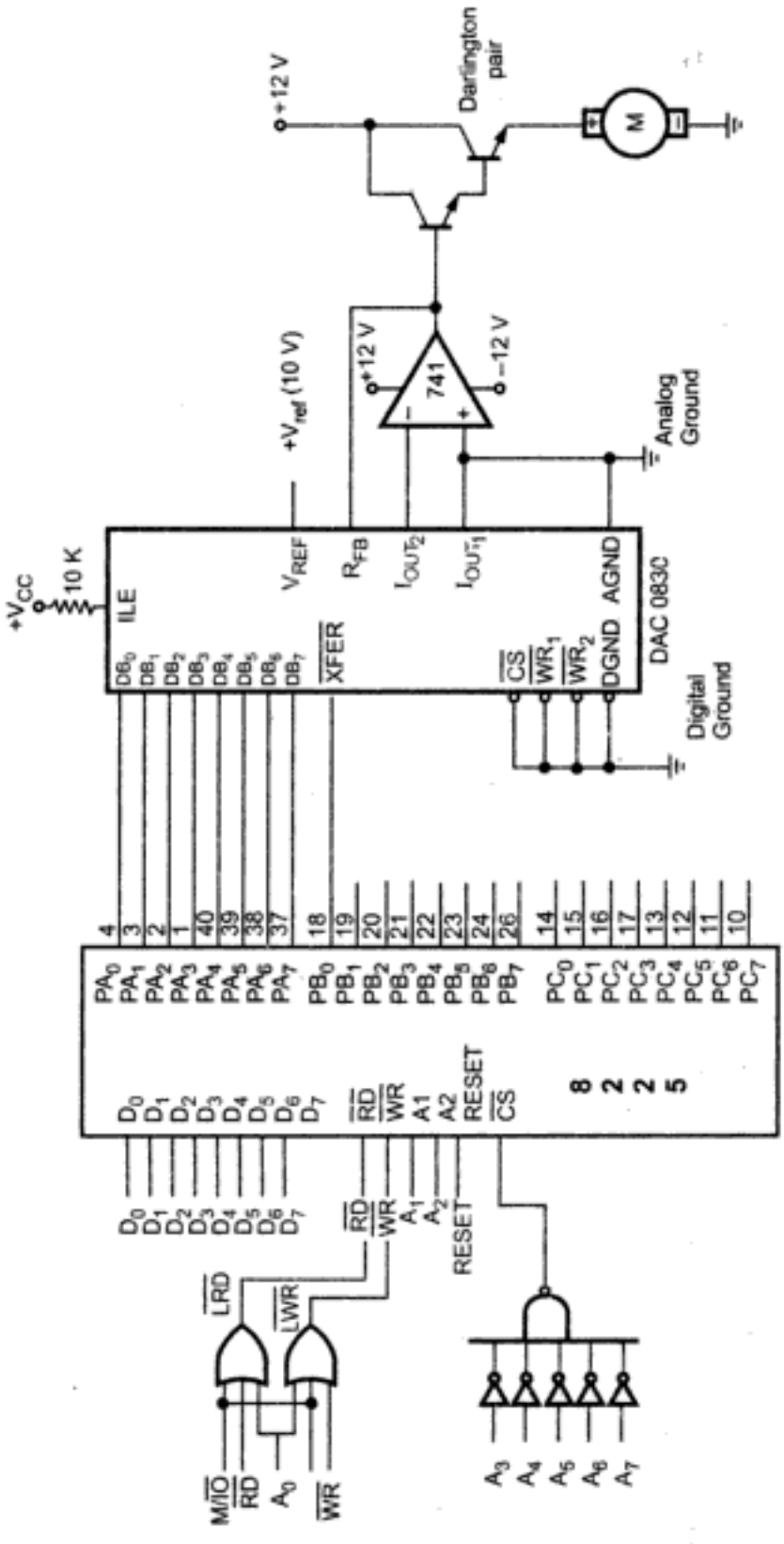


Fig. 7.28 Interfacing DAC0830 to 8086 microprocessor using 8255

Hidden page

Hidden page

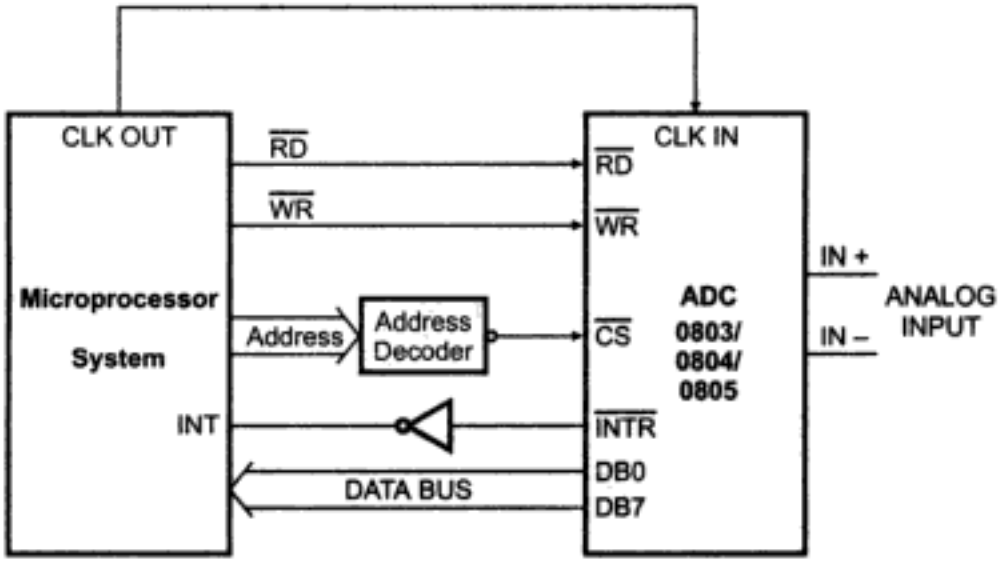


Fig. 7.32 Interfacing of ADC 0803/0804/0805 with microprocessor system

Interfacing the ADC 0803/0804/0805 to 8086 Microprocessor

Fig. 7.33 shows the interfacing of ADC 0803/0804/0805 to the 8086 microprocessor. Here, converted digital data is read through data bus. The address is decoded using I/O mapped I/O technique. As shown in the Fig. 7.33, address for ADC is 80H. The Fig. 7.34 shows the timing diagram of ADC operation. The conversion starts when $\overline{CS}$ and $\overline{WR}$ signals go low. The end of conversion is indicated by $\overline{INTR}$ output of the ADC. The $\overline{INTR}$ output goes low after conversion is over. Therefore, $\overline{INTR}$ signal is polled through data bus by enabling a buffer to detect the end of conversion. Once the conversion is over, the digital data is read by activating I/O read command. This is illustrated in the following procedure.

(See Fig. 7.33 and 7.34 on next page)

```
; Procedure to read data from ADC 0803/0804/0805

     READ      PROC NEAR
               OUT 80H,AL     ; Start conversion
AGAIN:         IN AL,82H      ; Read INTR
               AND AL,80H     ; Check INTR
               JNZ AGAIN      ; Repeat until INTR = 0
               IN AL,80H      ; Read digital data in AL
               RET
READ           ENDP
```

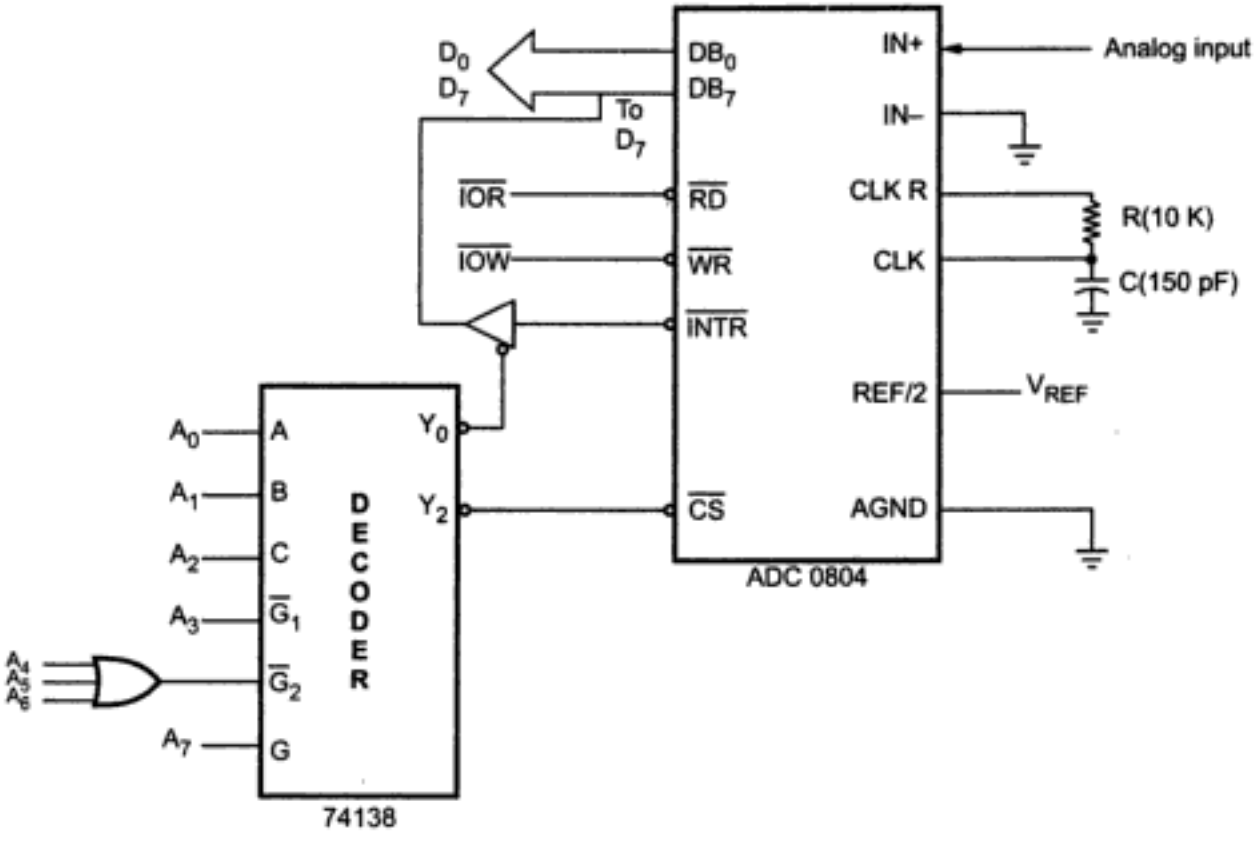


Fig. 7.33 Interfacing of ADC 0803/0804/0805 to 8086

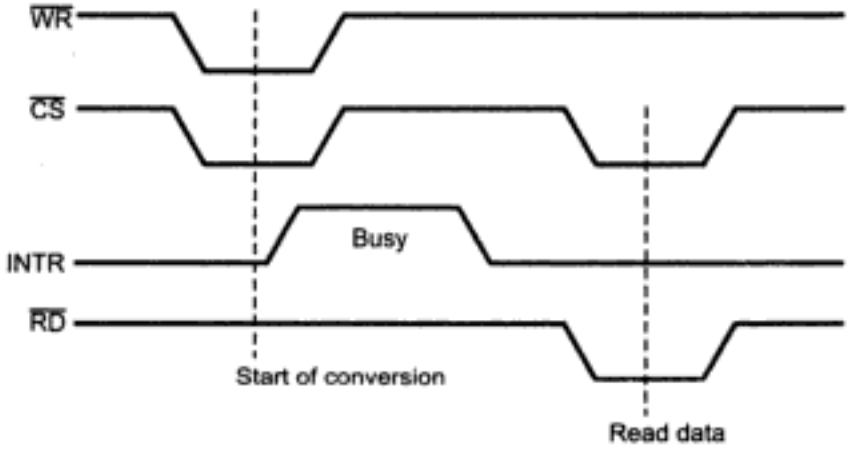


Fig. 7.34 Timing diagram

Interfacing ADC 0803/0804/0805 to 8086 using 8255

Fig. 7.35 shows the interfacing of ADC 0803/0804/0805 to 8086 using 8255. Here, port A of 8255 is used to read digital data from 8255. The start of conversion signal ($\overline{WR}$ and $\overline{CS}$ = low) generated using port B, PB_0 pin. The end of conversion is detected by polling $\overline{INTR}$ pin through PC_0. The procedure given below illustrates the operation.

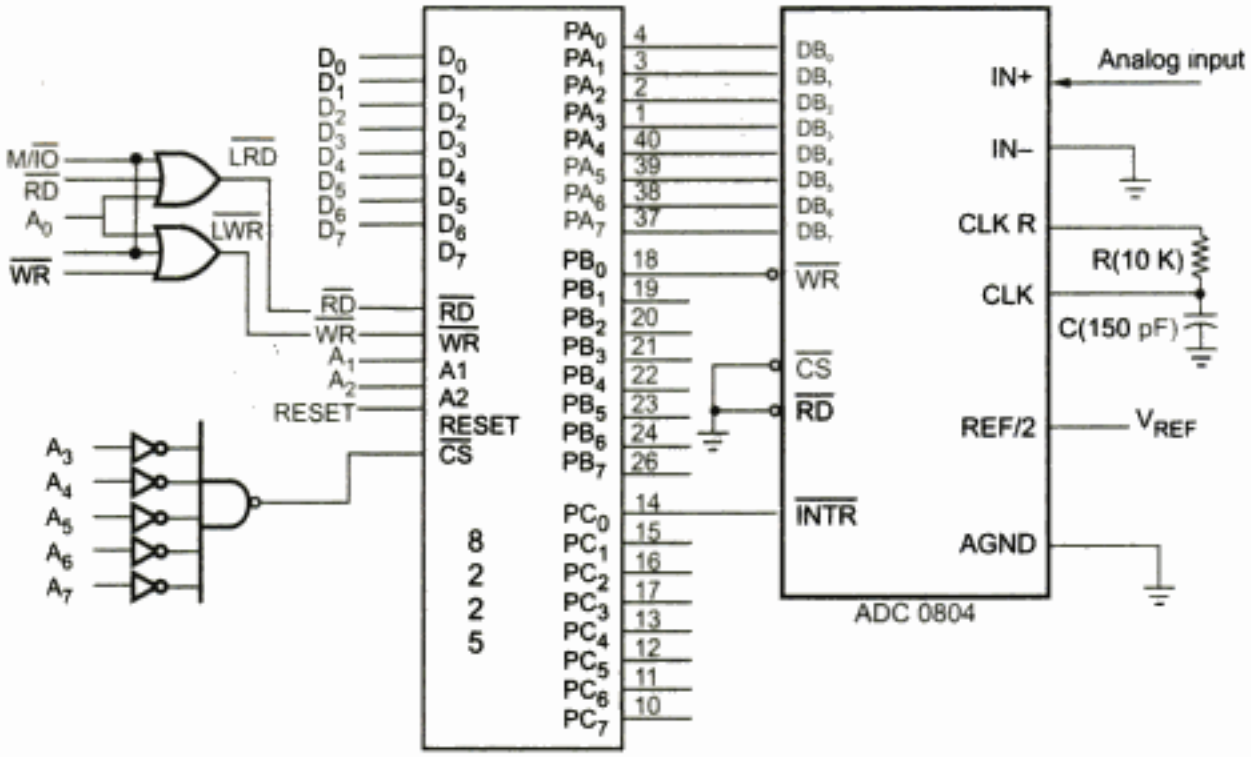


Fig. 7.35 Interfacing of ADC 0803/0804/0805 to 8086 using 8255

```
READ    PROC NEAR
        MOV AL,99H    ; [Initialize 8255 as port A and
        OUT 06H,AL    ; port C input and port B output]
        MOV AL,0FFH   ; [Make WR
        OUT 02H,AL    ; high]
        MOV AL,0FCH   ; [Send start of
        OUT 02H,AL    ;  conversion]
        NOP           ; wait
        MOV AL,0FFH   ; [Make WR
        OUT 02H,AL    ;  high]
BACK:   IN AL,04H     ; Read INTR
        AND AL,01H    ; Check INTR
        JNZ BACK      ; Repeat until INTR = 0
        IN AL,00H     ; Read digital data in al
        RET
READ    ENDP
```

Application

This section illustrates the application of ADC/DAC to store and reproduce audio signal or speech. Refer Fig. 7.36. Here, speech data is converted to digital data using ADC0804. This data is stored in the array at the sampling rate of 1/2048 of second. Then this sampled data is sent to DAC0830 with same rate to reduce the speech signal. This is illustrated in the following program.

Hidden page

```
.MODEL SMALL
.DATA
        SAMPLES DB 2048 DUP (?)    ; Space for speech samples
.CODE
START:  MOV AX,@DATA               ; [Initialize
        MOV DS,AX                  ; data segment]
        CALL READ                  ; Read speech
        CALL WRITE                 ; Reproduce speech
READ    PROC NEAR
        MOV CX,2048                ; Initialize counter
        MOV DI,OFFSET SAMPLES      ; Initialize pointer to array
AGAIN:  OUT 84H,AL                 ; Send start of conversion
BACK:   IN AL,80H                  ; Read INTR
        AND AL,80H                 ; Check INTR
        JNZ BACK                   ; Repeat until INTR = 0
        IN AL,84H                  ; Read sample
        MOV [DI],AL                ; Store sample in array
        INC DI                     ; Increment array pointer
        CALL DELAY                 ; Wait for 1/2048 seconds
        LOOP AGAIN                 ; Repeat 2048 times
        RET
READ    ENDP
WRITE   PROC NEAR
        MOV CX,2048                ; Initialize counter
        MOV DI,OFFSET SAMPLES      ; Initialize array pointer
BACK1:  MOV AL,[DI]                ; Read sample from array
        OUT 82H,AL                 ; Send it to DAC
        INC DI                     ; Increment array pointer
        CALL DELAY                 ; Wait for 1/2048 second
        LOOP BACK1                 ; Repeat 2048 times
        RET
WRITE   ENDP
DELAY   PROC NEAR
; This procedure generates approximately 1/2048 second delay
; assuming 8MHz Clock frequency of 8086.
        PUSH CX                    ; Save CX register
AGAIN1: MOV CX,0255                ; Initialize counter
        LOOP AGAIN1                ; Repeat until count = 0
        POP CX                     ; Restore CX register
        RET
DELAY   ENDP
        END START
```

7.10.2 ADC 0808/0809

The ADC 0808 and ADC 0809 are monolithic CMOS devices with an 8-channel multiplexer. These devices are also designed to operate from common microprocessor control buses, with tri-state output latches driving the data bus. The main features of these devices are :

Features

- 8-bit successive approximation ADC.
- 8-channel multiplexer with address logic.
- Conversion time 100 μs.
- It eliminates the need for external zero and full-scale adjustments.
- Easy to interface to all microprocessors.
- It operates on single 5 V power supply.
- Output meet TTL voltage level specifications.

Pin Diagram

Fig. 7.37 shows pin diagram of 0808/0809 ADC.

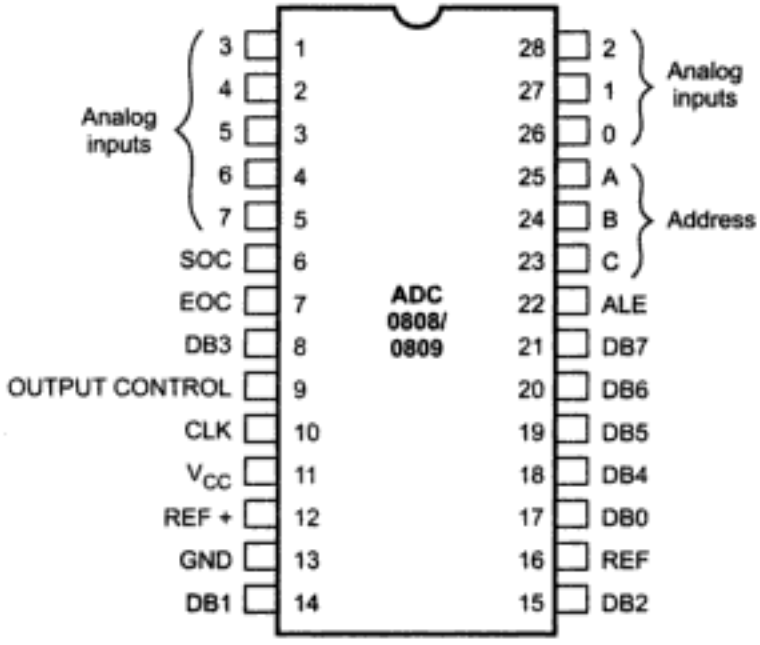


Fig. 7.37 Pin diagram of 0808/0809

Operation

ADC 0808/0809 has eight input channels, so to select desired input channel, it is necessary to send 3-bit address on A, B and C inputs. The address of the desired channel is sent to the multiplexer address inputs through port pins. After at least 50 ns, this address must be latched. This can be achieved by sending ALE signal. After another 2.5 μs, the start of conversion (SOC) signal must be sent high and then low to start the conversion process. To indicate end of conversion ADC 0808/0809 activates EOC signal. The microprocessor system can read converted digital word through data bus by enabling the output enable signal after EOC is activated. This is illustrated in Fig. 7.38.

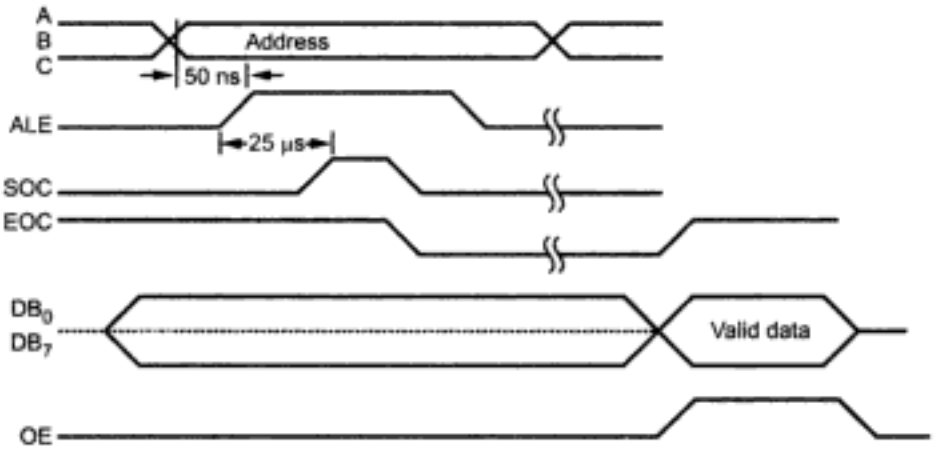


Fig. 7.38 Timing waveforms for ADC 0808

Interfacing

Fig. 7.39 shows typical interfacing circuit for ADC 0808 with microprocessor system.

The zener diode and LM 308 amplifier circuitry is used to produce a V_{CC} an $+V_{REF}$ of 5.12 V for the A/D converter. With this reference voltage the A/D converter will have 256 steps of 20 mV each.

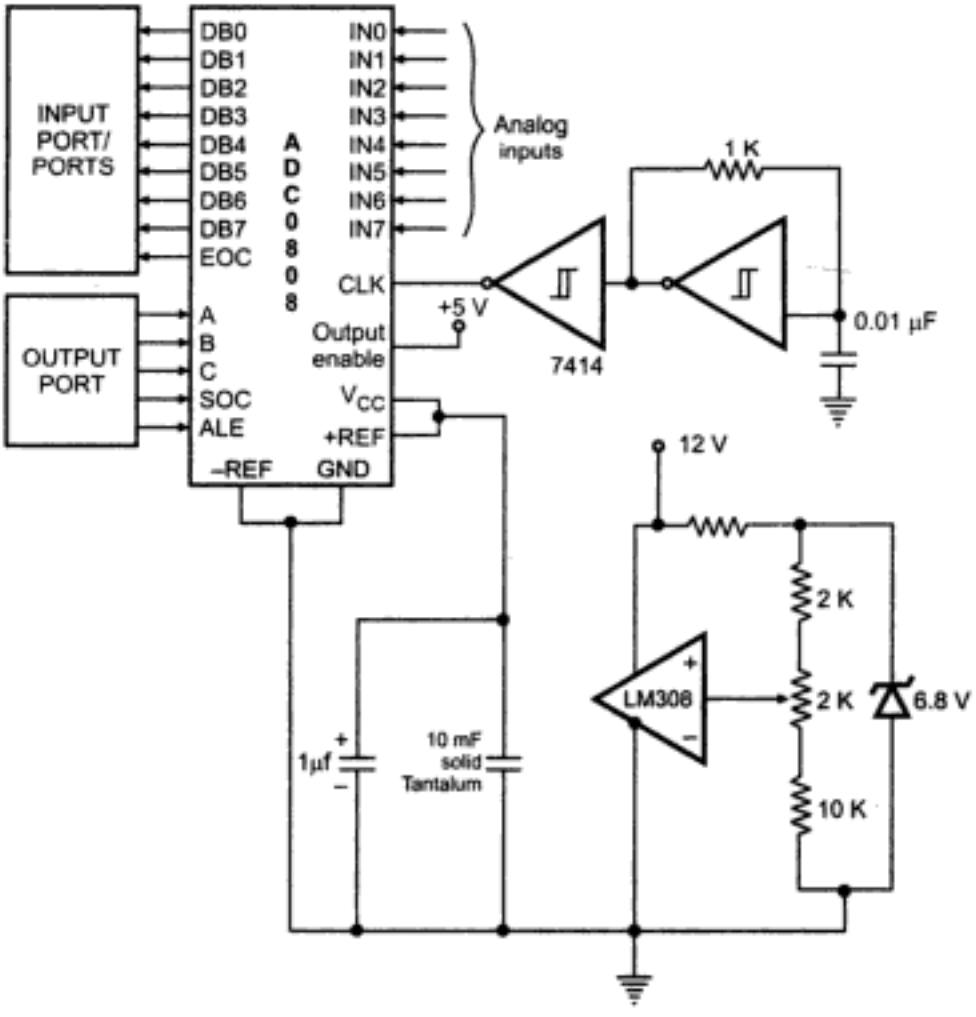


Fig. 7.39 Typical interface for 0808/0809

7.11 Stepper Motor Interfacing

A stepper motor is a digital motor. It can be driven by digital signal. Fig. 7.40 shows the typical 2 phase motor interfaced using 8255. Motor shown in the circuit has two phases, with center-tap winding. The center taps of these windings are connected to the 12 V supply. Due to this, motor can be excited by grounding four terminals of the two windings. Motor can be rotated in steps by giving proper excitation sequence to these windings. The lower nibble of port A of the 8255 is used to generate excitation signals in the proper sequence.

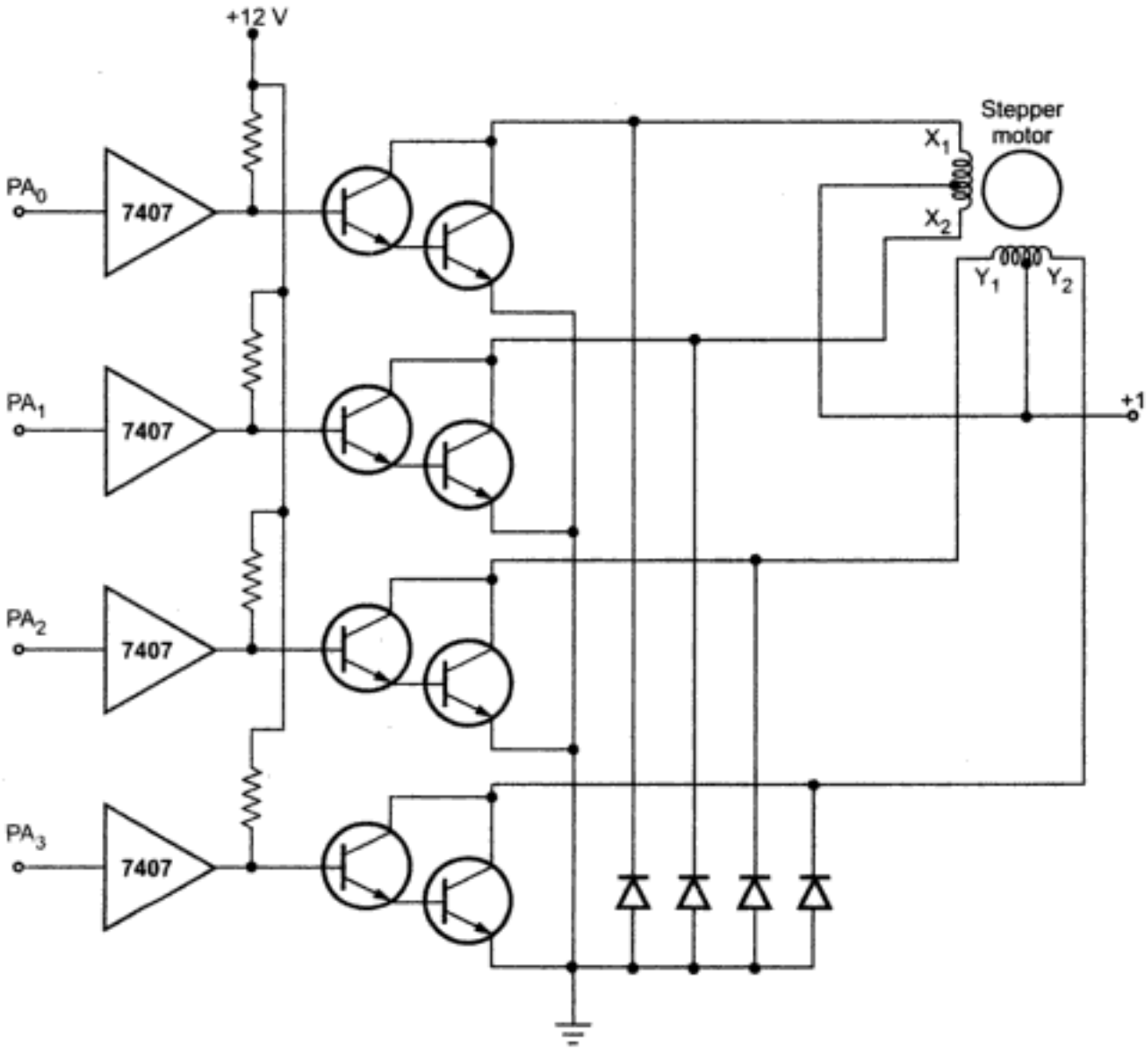


Fig. 7.40 Stepper motor interface

The Table 7.3 shows typical excitation sequence. The given excitation sequence rotates the motor in clockwise direction. To rotate motor in anticlockwise direction we have to excite motor in a reverse sequence. The excitation sequence for stepper motor may change due to change in winding connections. However, it is not desirable to excite both the ends

of the same winding simultaneously. This cancels the flux and motor winding may damage. To avoid this, digital locking system must be designed. Fig. 7.41 shows a simple digital locking system. Only one output is activated (made low) when properly excited; otherwise output is disabled (made high).

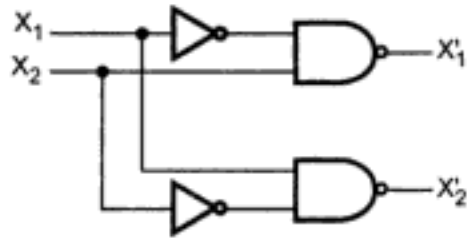


Fig. 7.41 Digital locking system

Step	X_1	X_2	Y_1	Y_2
1	0	1	0	1
2	1	0	0	1
3	1	0	1	0
4	0	1	1	0
1	0	1	0	1

Table 7.3 Full step excitation sequence

The excitation sequence given in Table 7.3 is called **full step sequence** in which excitation ends of the phase are changed in one step. The excitation sequence given in Table 7.4 takes two steps to change the excitation ends of the phase. Such a sequence is called **half step sequence** and in each step the motor is rotated by 0.9°.

Step	X_1	X_2	Y_1	Y_2
1	0	1	0	1
2	0	0	0	1
3	1	0	0	1
4	1	0	0	0
5	1	0	1	0
6	0	0	1	0
7	0	1	1	0
8	0	1	0	0
1	0	1	0	1

Table 7.4 Half step excitation sequence

We know that stepper motor is stepped from one position to the next by changing the currents through the fields in the motor. The winding inductance opposes the change in current and this puts limit on the stepping rate. For higher stepping rates and more torque, it is necessary to use a higher voltage source and current limiting resistors as shown in Fig. 7.42. By adding series resistance, we decrease L/R time constant, which allows the current to change more rapidly in the windings. There is a power loss across series resistor, but designer has to compromise between power and speed.

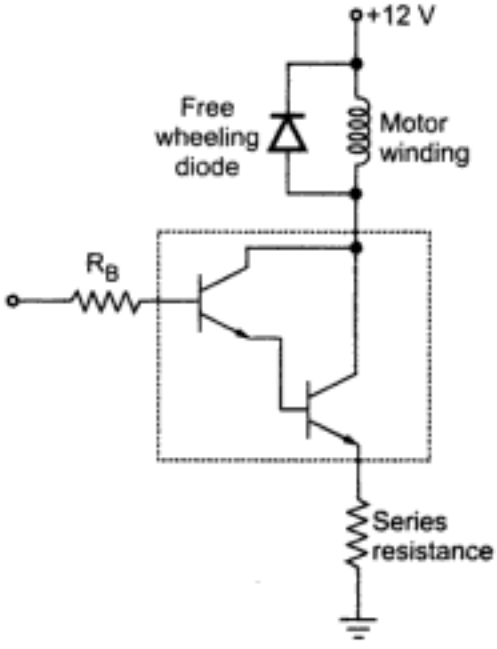


Fig. 7.42 Excitation circuit with series resistance

Example 3 : *Interface stepper motor to the 8086 microprocessor system and write an 8086 assembly language program to control the stepper motor.*

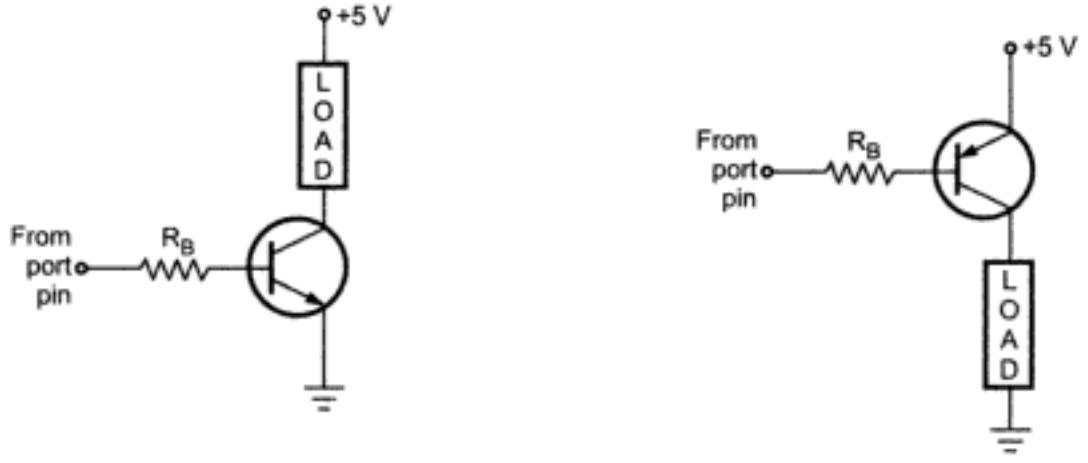
Solution : Hardware : Fig. 7.43 shows the typical 2 phase motor rated 12 V/.67A/ph interfaced with the 8086 microprocessor system using 8255. Motor shown in the circuit has two phases, with center-tap winding. The center taps of these windings are connected to the 12 V supply. Due to this, motor can be excited by grounding four terminals of the two windings. Motor can be rotated in steps by giving proper excitation sequence to these windings. The lower nibble of port A of the 8255 is used to generate excitation signals in the proper sequence. These excitation signals are buffered using driver transistors. The transistors are selected such that they can source rated current for the windings. Motor is rotated by 1.8^o per excitation.

Hidden page

Hidden page

7.12.2 Transistor Buffers

The Fig. 7.44 shows some buffer circuits using transistors. In these circuits, transistor is used as a switch. It can be switch ON or OFF with logic 0 or logic 1 on port pin depending on the application. To make transistor ON with logic 0 at port pin we have to use pnp transistor otherwise we have to use npn transistor. When transistor is ON, its collector current drives the load. To determine component values and transistor we have to check maximum collector current of the transistor (I_{Cmax}), maximum h_{fe} that transistor can provide (h_{femax}), maximum collector to emitter breakdown voltage (V_{BCEO})and a maximum power dissipation (p_{dmax}) allowed by the transistor.



(a) npn transistor (b) pnp transistor

Fig. 7.44 Transistor buffer circuits

Let us assume that the load current is 200 mA and maximum sourcing current of port pin is 1 mA. Then transistor should have

$$I_{Cmax} > 200 \text{ mA}$$

$$h_{fe\ min} > \frac{I_C}{I_{Bmax}}$$

$$> \frac{200 \text{ mA}}{1 \text{ mA}}$$

$$\therefore \quad h_{fe\ min} > 200$$

$$P_{dmax} > V_{CEsat} \times I_{Cmax}$$

$$> 0.2 \times 200 \text{ mA}$$

$$> 40 \text{ mW}$$

Assuming output high voltage of port pin equal to 4.5 V we have

$$R_B = \frac{4.5 - 0.8}{1 \text{ mA}}$$

$$= 3.7 \text{ k}\Omega$$

Hidden page

Hidden page

The mechanical relays or contactors, however, have several serious problems. When the contacts are opened and closed, arcing takes place between the contact, which causes the contacts to oxidize and pit. As the contacts are oxidized, they become higher resistance contact and may get hot enough to melt. Another disadvantage of mechanical relays is that when they switch ON or OFF at high-voltage point, they produce large amount of electrical noise, called **electromagnetic interference (EMI)**.

7.12.3.2 Solid State Relays

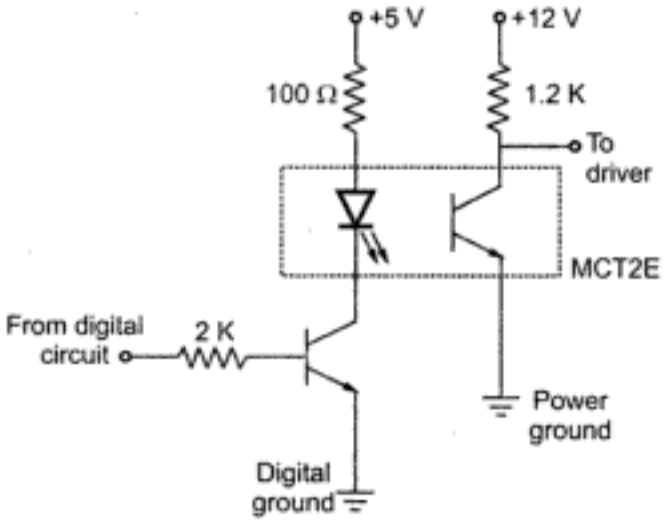


Fig. 7.48 Optoisolator circuitry

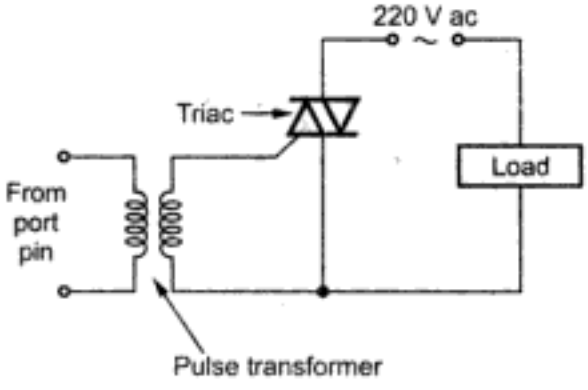


Fig. 7.49 Isolation using pulse transformer

To avoid the problems of mechanical relays, solid state relays are used. In this triac is used as a switching element and isolation is provided by optoisolators or pulse transformer. The Fig. 7.48 shows the typical optoisolator circuitry. It consists of LED and a phototransistor. LED glows when digital input is high, making phototransistor ON. Thus digital input controls the voltage at the collector of phototransistor without any physical connection between them, providing isolation. When digital input is low, LED and hence phototransistor is OFF.

The Fig. 7.49 shows the typical pulse transformer circuit. The pulse transformer magnetically couples the control and power circuitry avoiding electrical contact between them.

The optoisolator circuits give better performance at relatively low switching speed. Because it has switching speed limitations. On the other hand, at high switching speeds pulse transformer provides better performance. At low switching speed pulse transformer may get saturated to deteriorate its performance.

7.13 Keyboard and Display Interfacing

In this section we discuss the keyboard and display interfacing using 8255.

Example 4 : *Interface 4 × 4 keyboard with 8086 microprocessor.*

Solution : **Hardware** : Fig. 7.50 shows a matrix keyboard with 16 keys connected to the 8086 microprocessor using 8255. A matrix keyboard reduces the number of connections, thus the number of interfacing lines. In this example the keyboard with 16 keys, is

arranged in 4 × 4 (4 rows and 4 columns) matrix. This requires eight lines from the microprocessor to make all the connections instead of 16 lines if the keys are connected individually. The interfacing of matrix keyboard requires two ports : one input port and one output port. Rows are connected to the Input Port (return lines) and columns are connected to the Output Port (scan lines). When all keys are open row and column do not have any connection. When any key is pressed, it shorts corresponding row and column. If the output line of this column is low, it makes corresponding row line low; otherwise the status of row line is high. The key is identified by data sent on the output port and input code received from the input port. The following section explains the steps required to identify pressed key.

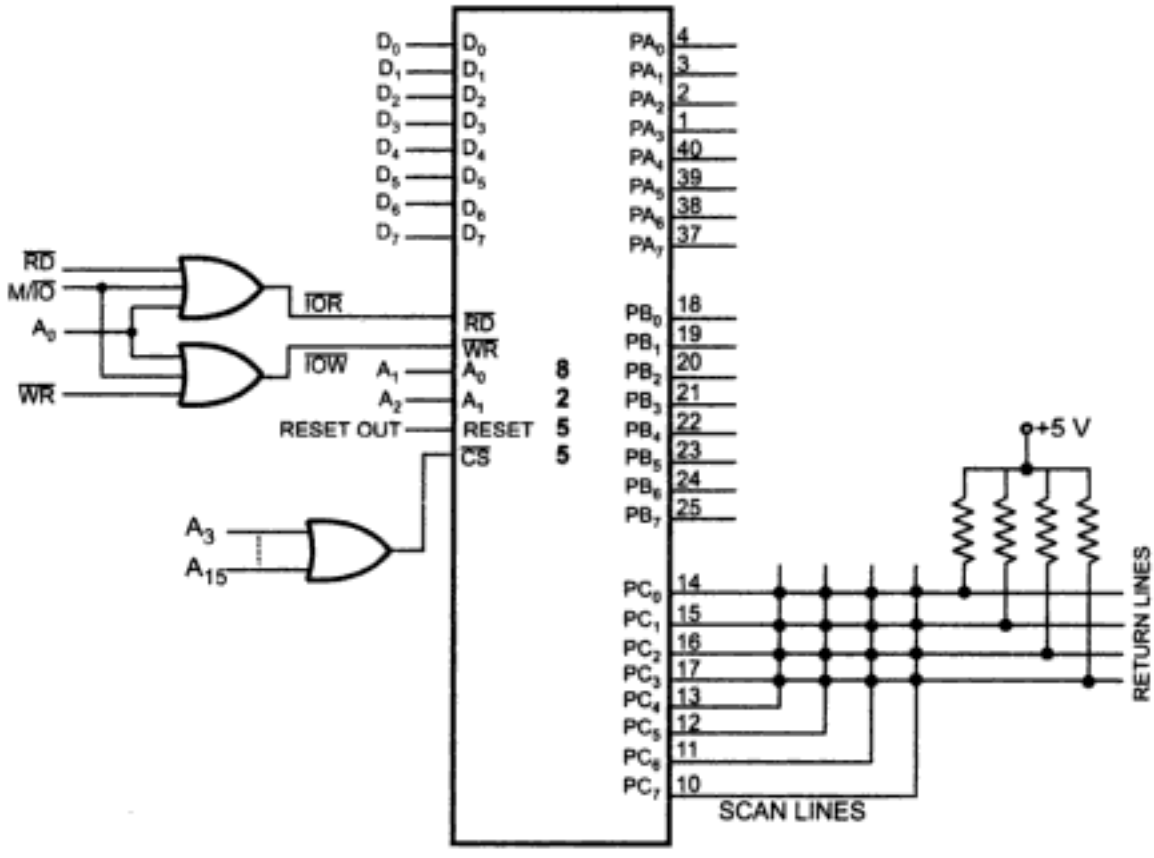


Fig. 7.50 Interfacing of 4 × 4 keyboard with 8086

Check 1 : Whether any key is pressed or not

1. Make all column lines zero by sending low on all output lines. This activates all keys in the keyboard matrix. (Note : When scan lines are logic high, the status on the return lines do not change, it will remain logic high.)
2. Read the status of return lines. If the status of all lines is logic high, key is not pressed; otherwise key is pressed.

Check 2 :

1. Activate keys from any one column by making any one column line zero.
2. Read the status of return lines. The zero on any return line indicates key is pressed from the corresponding row and selected column. If the status of all lines is logic high, key is not pressed from that column.
3. Activate the keys from the next column and repeat 2 and 3 for all columns.

In Fig. 7.50 the scan lines are connected to the port C_L of 8255 and returns lines are connected to the port C_U of 8255.

Flowchart

(See flowchart on next page).

Source program

```
              PORTA        EQU      0000
              PORTC        EQU      0004
              CR           EQU      0006
     PROC KEY NEAR
START:   MOV AL,81H       ; Initialize Port CL as input and Port CU
                          ; as output
         MOV DX,CR        ; [ Initialise
         OUT DX,AL        ;   8255 ]
         MOV AL,00H
         MOV DX,PORTC
         OUT DX,AL        ; Make all scan lines zero
BACK:    IN  AL,DX
         AND AL,0FH
         CMP AL,0FH       ; Check for key release
         JNZ BACK         ; If not, wait for key release
BACK1:   IN  AL,DX
         AND AL,0FH
         CMP AL,0FH       ; Check for key press
         JZ BACK1         ; If not, wait for key press
         CALL DELAY       ; Wait for key debounce
         MOV BL,00H       ; Initialize key counter
         MOV CL,04H
         MOV BH,FEH       ; Make one column low
NEXTCOL: MOV AL,BH
         OUT DX,AL
         MOV CH,04H       ; Initialize row counter
         MOV DX,PORTA
         IN AL,DX         ; Read return line status
```

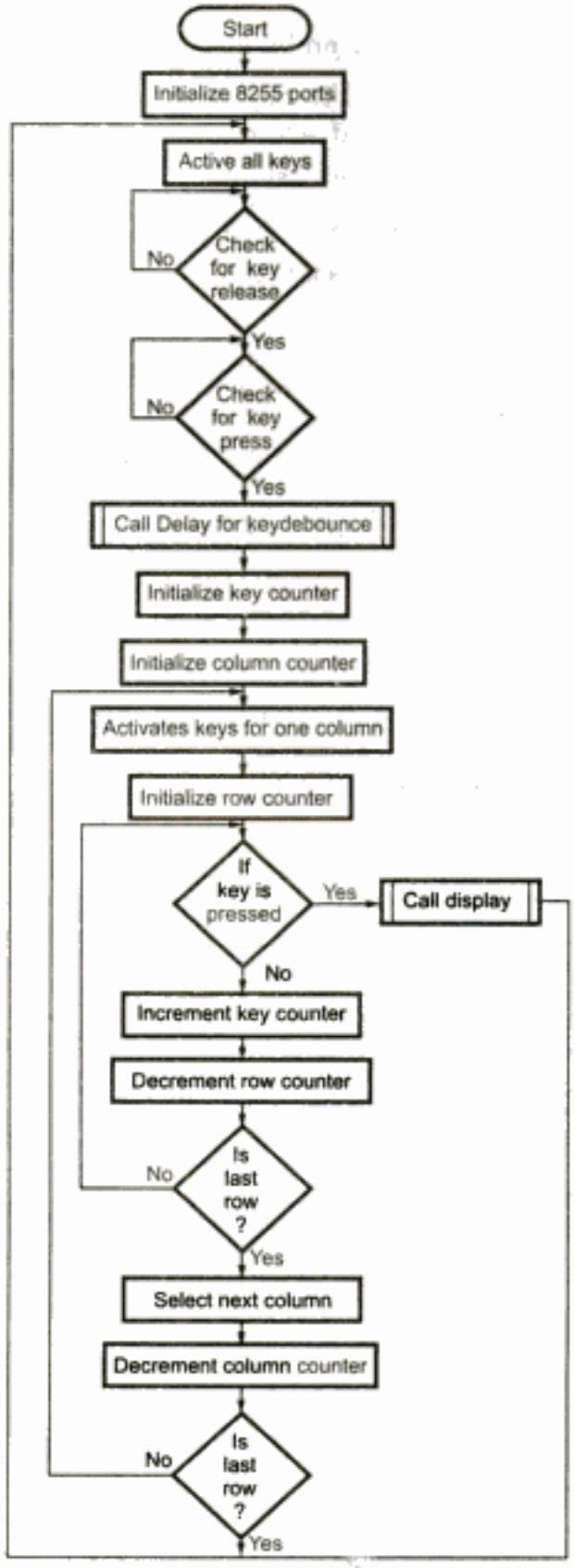


Fig. 7.51 Flowchart

```
NEXTROW: RCR AL,1       ; Check for one Row
         JNC DISPLAY    ; If zero, goto display
                        ; otherwise continue
         INC BL         ; Increment key counter
         DEC CH         ; Decrement row counter
         JNZ NEXTROW    ; Check for next row
         MOV AL,BH
         RCL AL,1       ; Select the next column
         MOV BH,AL
         DEC C          ; Decrement column count
         JNZ NEXTCOL    ; Check for last column if not repeat
         JMP START      ; Goto start
         RET
         KEY ENDP
         END START
```

Example 5 : *Interface an 8-digit 7 segment LED display using 8255 to the 8086 microprocessor system and write an 8086 assembly language routine to display message on the display.*

Solution : Hardware : Fig. 7.52 shows the multiplexed eight digit 7-segment display connected in the 8086 system using 8255. In this circuit port A and port B are used as simple latched output ports. Port A provides the segment data inputs to the display and port B provides a means of selecting a display position at a time for multiplexing the displays. The 8255 is addressed using direct addressing mode, so only A_0-A_7 lines are used to decode the addresses for 8255.

For this circuit different addresses are :

PA = 00H PC = 04H

PB = 02H CR = 06H

The register values are chosen in Fig. 7.52 so the segment current is 80 mA. This current is required to produce an average of 10 mA per segment as the displays are multiplexed. In this type of display system, only one of the eight display position is ON at any given instant. Only one digit is selected at a time by giving low signal on the corresponding control line. Maximum anode current is 560 mA (7-segments × 80 mA = 560 mA), but the average anode current is 70 mA.

Software : Before going to write the software we must know the control word to program 8255 according to hardware connections. For 8255 Port A and B are used as output ports.

Control word format for 8255

BSR	Mode A		PA	PC_U	Mode B	PB	PC_L	
1	0	0	0	X	0	0	X	= 80H

Hidden page

Program :

```
.MODEL SMALL
.DATA
     PA        EQU  80H
     PB        EQU       82H
     CR        EQU  86H
     MES  DB  41H,42H,43H,44H,45H,46H,47H,48H
.CODE
; Procedure to display message on multiplexed LED display
DISP      PROC NEAR
          MOV AX,@DATA              ; [ Initialise
          MOV DS,AX                 ;   data segment ]
          MOV AL,80H                ; Load control word in AL
          OUT CR,AL                 ; Load control word in CR
          PUSH F                    ; Save registers
          PUSH AX
          PUSH BX
          PUSH DX
          PUSH SI
; set up registers for display
          MOV BX,08H                ; load count
          MOV AH,7FH                ; load select pattern
          LEA SI,MES                ; starting address of message
; display message
DISP1:    MOV AL,AH                 ; select digit
          OUT PB,AL
          MOV AL,[BX+SI]            ; get data
          OUT PA,AL                 ; display data
          CALL DELAY                ; wait for some time
          ROR AH,01H                ; adjust selection pattern
          DEC BX                    ; adjust count
          JNZ DISP1                 ; repeat 8 times
          POP SI                    ; restore registers
          POP DX
          POP BX
          POP AX
          POPF
          RET
          DISP ENDP
```

Note : This procedure must be called continuously to display the 7-segment coded message in the memory.

7.15 Centronics Printer Interface

As explained earlier, handshaking signals are required to transfer data between two devices whose speeds are not same. This centronics protocol is a printer protocol, gives standards for printer interface.

It has 36 pins. The Fig. 7.53 shows the pin definitions for centronics interface. The ASCII characters are sent to the printer through eight data lines. Each data line has individual ground to reduce the change of picking up electrical noise in the lines.

Signal Pin No.	Return Pin No.	Signal	Direction	Description
1	19	$\overline{\text{STROBE}}$	IN	STROBE pulse to read data in. Pulse width must be more than 0.5 ms at receiving terminal. The signal level is normally "high"; read-in of data is performed at the "low" level of this signal.
2	20	DATA 1	IN	These signals represent 8-bit parallel data Each signal is at "high" level when data is logical "1" and "low" when logical "0".
3	21	DATA 2	IN	
4	22	DATA 3	IN	
5	23	DATA 4	IN	
6	24	DATA 5	IN	
7	25	DATA 6	IN	
8	26	DATA 7	IN	
9	27	DATA 8	IN	
10	28	$\overline{\text{ACKNLG}}$	OUT	Approximately 5 ms pulse; "low" indicates that data has been received and the printer is ready to accept other data.
11	29	BUSY	OUT	A "high" signal indicates that the printer cannot receive data. The signal becomes "high" in the following cases: 1. During data entry. 3. In "office" state. 2. During printing operation. 4. During printer error status.
12	30	PE	OUT	A "high" signal indicates that the printer is out of paper.
13	--	SLCT	OUT	This signal indicates that the printer is in the selected state.
14	--	$\overline{\text{AUTO FEED XT}}$	IN	When this signal being at "low" level, the paper is automatically fed one line after printing. (The signal level can be fixed to "low" with DIP SW pin 2-3 provided on the control circuit board).
15	--	NC		Not used.
16	--	OV		Logic GND level.
17	--	CHASIS-GND	--	Printer chasis GND. In the printer, the chasis GND and the logic GND are isolated from each other.
18	--	NC	--	Not used.
19-30	--	GND	--	"Twisted-Pair Return" signal; GND level.

31	--	$\overline{\text{INIT}}$	IN	When the level of this signal becomes "low" the printer controller is reset to its initial state and the print buffer is cleared. This signal is normally at "high" level and its pulse with must be more then 50 μs at the receiving terminal.
32	--	$\overline{\text{ERROR}}$	OUT	The level of this signal becomes "low" when the printer is in "Paper End" state. "Offline" state and "Error" state.
33	--	GND	--	Same as with pin numbers 19 to 30.
34	--	NC	--	Not used.
35				Pulled up to + 5 V_{dc} through 4.7 K-ohms resistance.
36	--	$\overline{\text{SLCT IN}}$	IN	Data entry to the printer is possible only when the level of this signal is "low".

Notes :

1. "Direction" refers to the direction off signal flow as viewed from the printer.
2. "Return" denotes "Twisted-Pair Return" and is to be connected at signal-ground level.

   When making the interface, be sure to use a twisted-pair cable for each signal and never fail to complete connection on the return side to prevent noise effectively, these cables should be shielded and connected to the chassis of the system unit.
3. All interface conditions are based on TTL level. Both the rise and fall times of each signal must be less than 0.2 μs.
4. Data transfer must not be carried out by ignoring the ACKNLG or BUSY signal. (Data transfer to this printer can be carried out only after confirming the ACKNLG signal or when the level of the BUSY signal is "low".

Fig. 7.53 Pin definitions for centronics interface

The other signals fall into two categories, signals sent to the printer to tell it which operation to do and signals from the printer that indicate its status. These signals are as follows :

Input signals for printer :

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

Status signals output from printer :

1. $\overline{\text{ACK}}$: 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. 7.54 shows the timing waveforms for transfer of data characters to an IBM printer using the basic handshake signals.

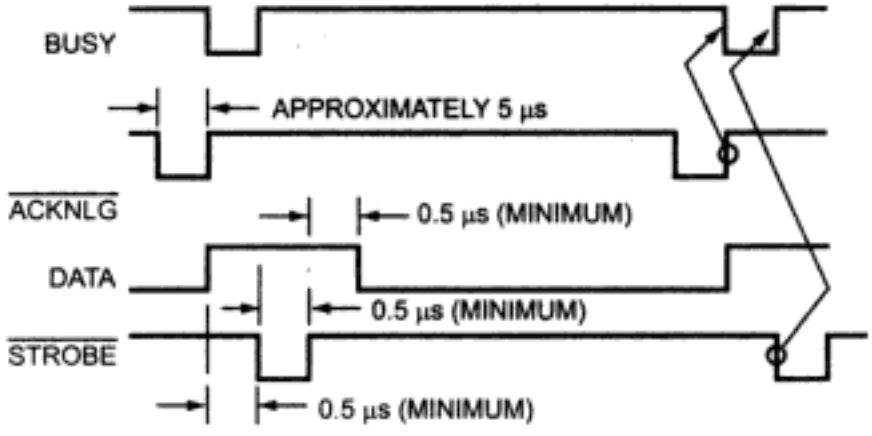


Fig. 7.54 Timing waveforms for transfer of data characters to an IBM printer

Communication between Computer and 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 at least 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.

Centronics Printer Interface using 8255

Fig. 7.55 shows the circuit for interfacing centronics type parallel input printer to 8255A. Port A is used to send 8-bit data to the printer. It is used in mode 1 so PC_6 ($\overline{OBF}$ signal) is used as $\overline{STB}$ signal to tell the printer that valid data is available on the data bus and PC_7 is used as an $\overline{ACK}$ signal. BUSY, PE and $\overline{ERROR}$ signals are connected to the PB_0 to PB_2 port lines.

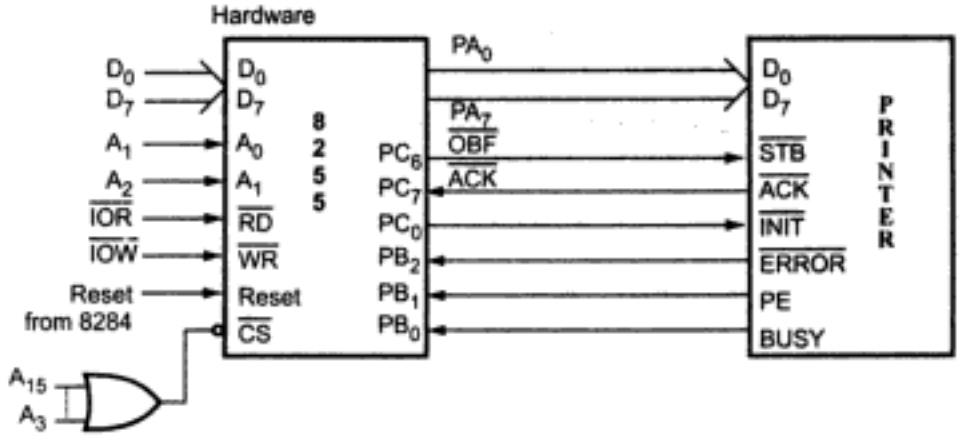


Fig. 7.55 Interfacing centronics printer to 8255A

In the next section we will see flowchart and program required to print a message.

Flowchart : Fig. 7.56 Flowchart for printer interface.

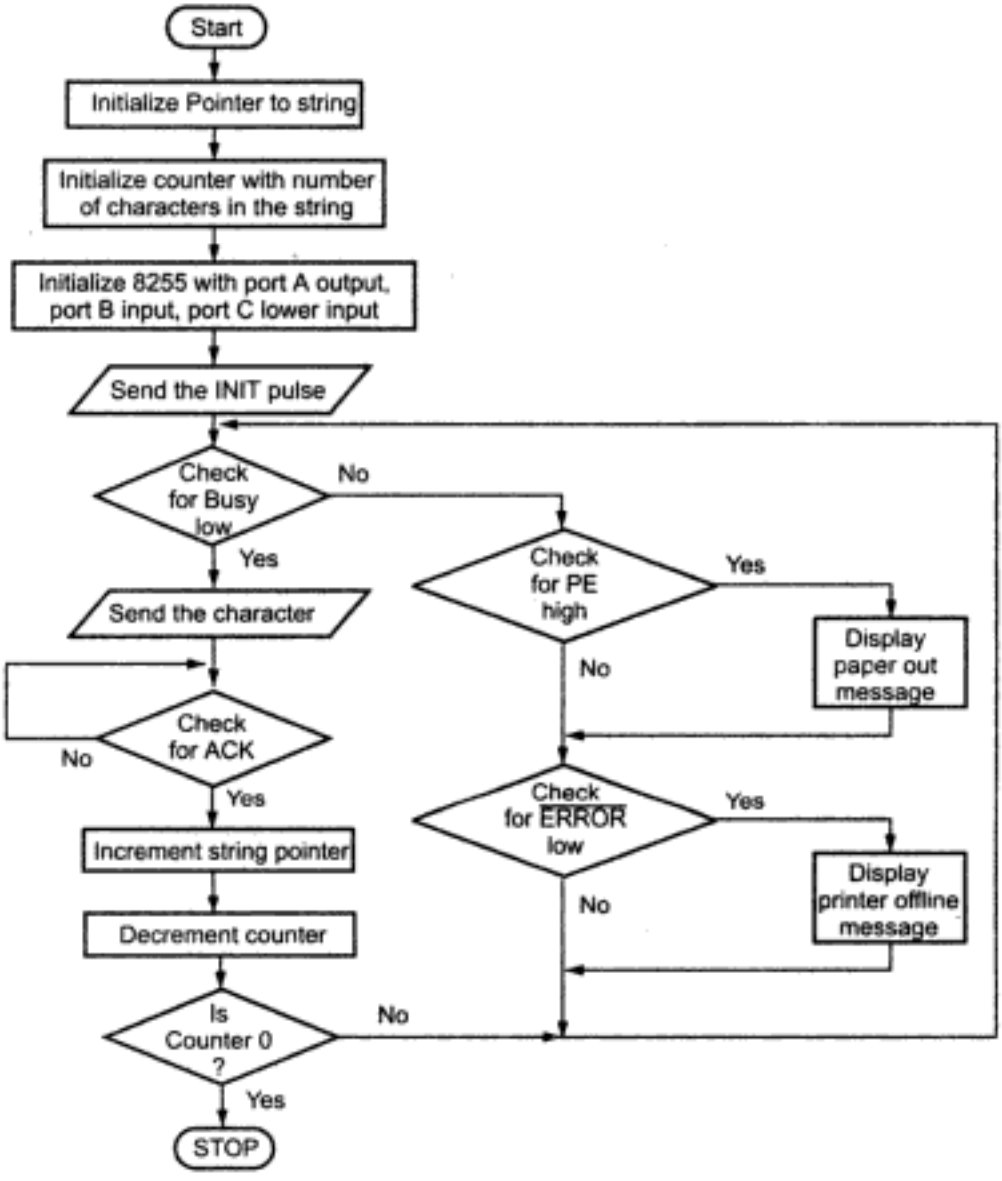


Fig. 7.56 Flowchart for printer interface

In the program it is necessary to initialize 8255 as follows

Port	Input/output	Mode
Port A	Output	1
Port B	Input	0
Port C Upper	—	—
Port C Lower	Output	—

Control word :

I/O	Mode A		PA	PC_U	Mode B	PB	PC_L
1	0	1	0	X	0	1	0

= A2H

I/O map :

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address	Port
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000 H	Port A
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0002 H	Port B
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0004 H	Port C
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0006 H	CR

Program :

```
        .MODEL SMALL
        PortA       EQU     0000
        PortB       EQU     0002
        PortC       EQU     0004
        CR          EQU     0006
        .DATA
            Mes1    DB 'Printer Paper Out', 10, 13, '$'
            Mes2    DB 'Printer Offline', 10, 13, '$'
            Mes3    DB 'Printing Over', '$'
            Mes4    DB 'This is to be print'
            COUNT   DB 15
        .CODE
START:  MOV     AX,@DATA        ; Initialize data segment
        MOV     DS,AX
        LEA     BX,MES4         ; Initialize pointer to string
        MOV     DX,CR           ; CR is control register address
        MOV     AL,0A2H         ; Load control word
        OUT     DX,AL
        MOV     AL,07           ; Make INTE_A high to enable INTR_A
```

```
        OUT     DX,AL
        MOV     AL,00
        OUT     DX,AL          ; Make PC0 low (BSR mode)
                               ; to give INIT low
        MOV     CX,0FFFH
BACK:   DEC     CX             ; Wait for more than 50 µs
        LOOP BACK
        MOV     AL,01
        OUT     DX,AL          ; Make INIT HIGH
NEXT:   MOV     DX,PortB
        IN      AL,DX
        MOV     AH,AL          ; Save status information
        AND     AL,01H
        JNZ     CHECK          ; Check for BUSY if high goto
                               ; check
        MOV     AL,[BX]
        MOV     DX,PortA
        OUT     DX,AL          ; Send the character
        MOV     DX,PortC
AGAIN:  IN      AL,DX          ; Check for ACK by
        AND     AL,08          ; checking INTRA line high
        JZ      AGAIN
        INC     BX             ; Increment string pointer
        MOV     AL,COUNT
        DEC     AL
        MOV     COUNT,AL       ; Decrement counter
        JNZ     NEXT           ; Check for counter = 0
        JMP     LAST
CHECK:  MOV     AL,AH
        AND     AL,02
        MOV     AL,AH          ; Save printer status
        JZ      CHECK1
        LEA     DX,MES1
        MOV     AH,09H         ; Call for DOS interrupt
        INT 21H                ; to display MES1
CHECK1: AND     AL,04
        JNZ     NEXT
        LEA     DX,MES2
        MOV     AH,09H         ; Call for DOS interrupt to
        INT     21H            ; display MES2
        JMP     NEXT
LAST :  LEA     DX,MES3        ; Call for DOS interrupt to
        MOV     AH,09H         ; display MES3
        INT     21H
        MOV     AH,4CH         ; Terminate program
        INT     21H
        END     START
        END
```

Hidden page

8

8086 Interrupts

8.1 Introduction

Sometimes it is necessary to have the computer automatically execute one of a collection of special routines whenever certain conditions exists within a program or in the microcomputer system. For example, it is necessary that microcomputer system should give response to devices such as keyboard, sensor and other components when they request for service.

The most common method of servicing such device is the **polled approach**. This is where the processor must test each device in sequence and in effect "ask" each one if it needs communication with the processor. It is easy to see that a large portion of the main program is looping through this continuous polling cycle. Such a method would have a serious and decremental effect on system throughput, thus limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices.

A more desirable method would be the one that allows the microprocessor to execute its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method, would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is completed, the processor would resume exactly where it left off. This method is called **interrupt method.** It is easy to see that system throughput would drastically increase, and thus enhance its cost effectiveness. Most microprocessors allow execution of special routines by interrupting normal program execution. When a microprocessor is interrupted, it stops executing its current program and calls a special routine which "services" the interrupt. The event that causes the interruption is called **interrupt** and the special routine executed to service the interrupt is called **interrupt service routine/procedure**. Normal program can be interrupted by three ways :

1. By external signal
2. By a special instruction in the program or
3. By the occurrence of some condition.

An interrupt caused by an external signal is referred as a **hardware interrupt**. Conditional interrupts or interrupts caused by special instructions are called **software interrupts**.

8.2 Interrupt Cycle of 8086/88

An 8086 interrupt can come from any one the three sources :

- External signal
- Special Instruction in the program
- Condition produced by instruction

8.2.1 External Signal (Hardware Interrupt)

An 8086 can get interrupt from an external signal applied to the nonmaskable interrupt (NMI) input pin, or the interrupt (INTR) input pin.

8.2.2 Special Instruction

8086 supports a special instruction, INT to execute special program. At the end of the interrupt service routine, execution is usually returned to the interrupted program.

8.2.3 Condition Produced by Instruction

An 8086 is interrupted by some condition produced in the 8086 by the execution of an instruction. For example divide by zero : Program execution will automatically be interrupted if you attempt to divide an operand by zero.

At the end of each instruction cycle 8086 checks to see if there is any interrupt request. If so, 8086 responds to the interrupt by performing series of actions (Refer Fig. 8.1).

1. It decrements stack pointer by 2 and pushes the flag register on the stack .
2. It disables the INTR interrupt input by clearing the interrupt flag in the flag register.
3. It resets the trap flag in the flag register.
4. It decrements stack pointer by 2 and pushes the current code segment register contents on the stack.
5. It decrements stack pointer by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect far jump at the start of the procedure by loading the CS and IP values for the start of the interrupt service routine (ISR).

An IRET instruction at the end of the interrupt service procedure returns execution to the main program.

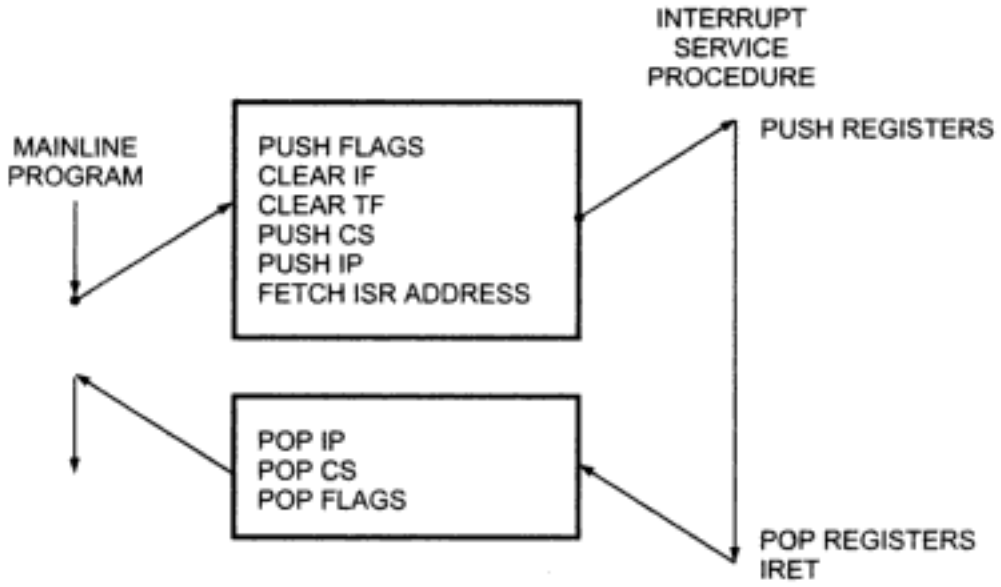


Fig. 8.1 8086 interrupt response

Now the question is "How to get the values of CS and IP register ?" The 8086 gets the new values of CS and IP register from four memory addresses. When it responds to an interrupt, the 8086 goes to memory locations to get the CS and IP values for the start of the interrupt service routine. In an 8086 system the first 1 Kbyte of memory from 00000H to 003FFH is reserved for storing the starting addresses of interrupt service routines. This block of memory is often called the **interrupt vector table** or the **interrupt pointer table**. Since 4 bytes are required to store the CS and IP values for each interrupt service procedure, the table can hold the starting addresses for 256 interrupt service routines. Fig. 8.2 shows how the 256 interrupt pointers are arranged in the memory table.

Each interrupt type is given a number between 0 to 255 and the address of each interrupt is found by multiplying the type by 4 e.g. for type 11, interrupt address is $11 \times 4 = 44_{10} =$ 0002CH

Only first five types have explicit definitions such as divide by zero and non maskable interrupt. The next 27 interrupt types, from 5 to 31, are reserved by Intel for use in future microprocessors. The upper 224 interrupt types, from 32 to 255, are available for user for hardware or software interrupts.

When the 8086 responds to an interrupt, it automatically goes to the specified location in the interrupt vector table to get the starting address of interrupt service routine. So user has to load these starting addresses for different routines at the start of the program

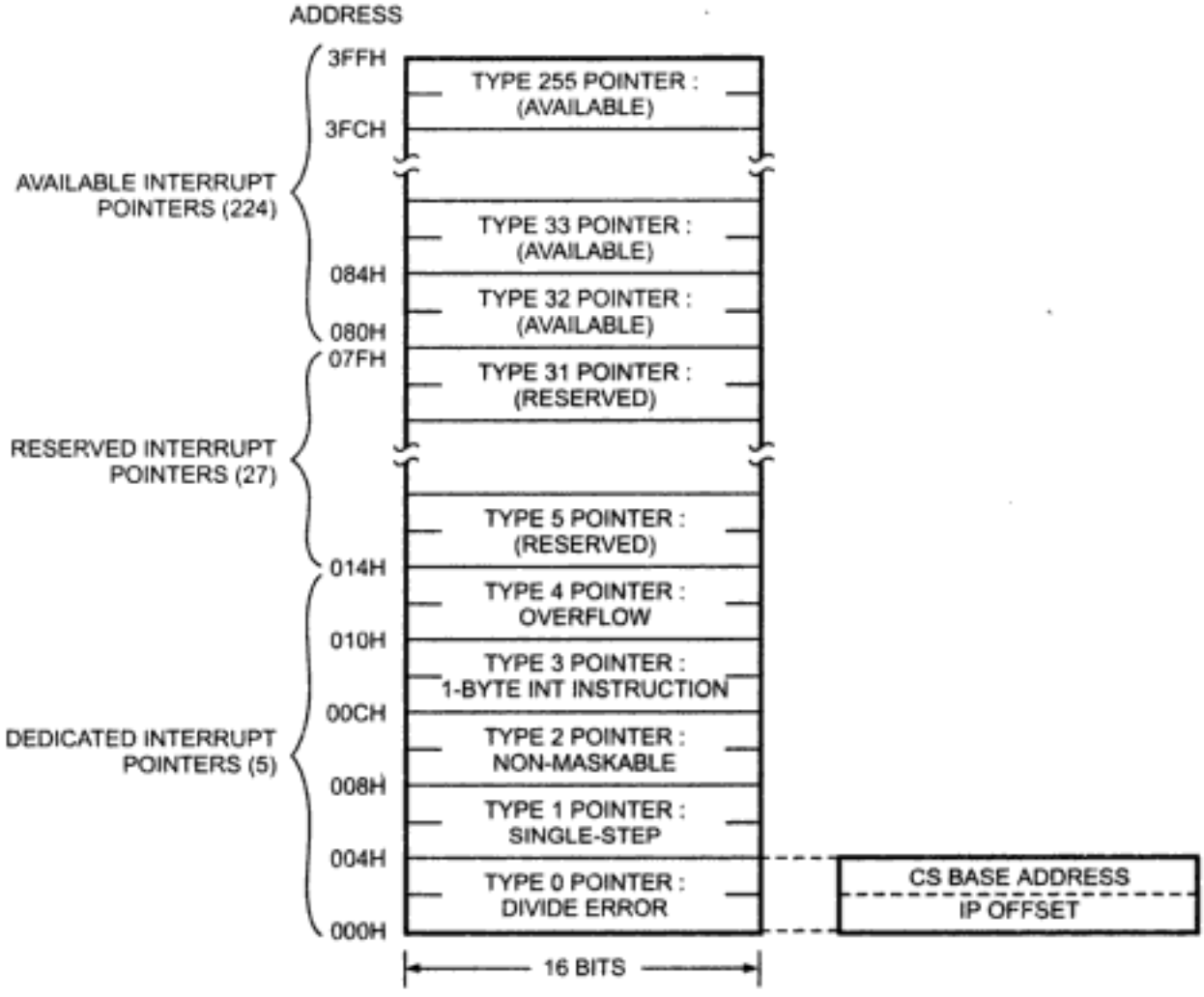


Fig. 8.2 8086 interrupt vector table

8.3 8086 Interrupt Types

8.3.1 Divide by Zero Interrupt (Type 0)

When the quotient from either a DIV or IDIV instruction is too large to fit in the result register; 8086 will automatically execute type 0 interrupt.

8.3.2 Single Step Interrupt (Type 1)

The type 1 interrupt is the single step trap. In the single step mode, system will execute one instruction and wait for further direction from user. Then user can examine the contents of registers and memory locations and if they are correct, user can tell the system to execute the next instruction. This feature is useful for debugging assembly language programs.

Hidden page

executes the INTO instruction, the instruction will simply function as ar NOP (no operation). However, if the overflow flag is set, indicating an overflow error, the 8086 will execute a type 4 interrupt after executing the INTO instruction.

Another way to detect and respond to an overflow error in a program is to put the jump if overflow instruction, (JO) immediately after the arithmetic instruction. If the overflow flag is set as a result of arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds in the way you want to the overflow.

8.3.6 Software Interrupts

Type 0 - 255 :

The 8086 INT instruction can be used to cause the 8086 to do one of the 256 possible interrupt types. The interrupt type is specified by the number as a part of the instruction. You can use an INT2 instruction to send execution to an NMI interrupt service routine. This allows you to test the NMI routine without needing to apply an external signal to the NMI input of the 8086.

With the software interrupts you can call the desired routines from many different programs in a system e.g. BIOS in IBM PC. The IBM PC has in its ROM collection of routines, each performing some specific function such as reading character from keyboard, writing character to CRT. This collection of routines referred to as **Basic Input Output System** or **BIOS**.

The BIOS routines are called with INT instructions. We will summarize interrupt response and how it is serviced by going through following steps.

1. 8086 pushes the flag register on the stack.
2. It disables the single step and the INTR input by clearing the trap flag and interrupt flag in the flag register.
3. It saves the current CS and IP register contents by pushing them on the stack.
4. It does an indirect far jump to the start of the routine by loading the new values of CS and IP register from the memory whose address calculated by multiplying 4 to the interrupt type, For example, if interrupt type is 4 then memory address is $4 \times 4 = 10_{10} = 10H$. So 8086 will read new value of IP from 00010H and CS from 00012H.
5. Once these values are loaded in the CS and IP, 8086 will fetch the instruction from the new address which is the starting address of interrupt service routine.
6. An IRET instruction at the end of the interrupt service routine gets the previous values of CS and IP by popping the CS and IP from the stack.
7. At the end the flag register contents are copied back into flag register by popping the flag register form stack.

8.3.7 Maskable Interrupt (INTR)

The 8086 INTR input can be used to interrupt a program execution. The 8086 is provided with a maskable handshake interrupt. This interrupt is implemented by using two pins - INTR and $\overline{INTA}$. This interrupt can be enabled or disabled by STI (IF=1) or CLI (IF=0), respectively. When the 8086 is reset, the interrupt flag is automatically cleared (IF=0). So after reset INTR is disabled. User has to execute STI instruction to enable INTR interrupt.

The 8086 responds to an INTR interrupt as follows :

1. The 8086 first does two interrupt acknowledge machine cycles as shown in the Fig. 8.3 to get the interrupt type from the external device. In the first interrupt acknowledge machine cycle the 8086 floats the data bus lines AD_0-AD_{15} and sends out an $\overline{INTA}$ pulse on its $\overline{INTA}$ output pin. This indicates an interrupt acknowledge cycle in progress and the system is ready to accept the interrupt type from the external device. During the second interrupt acknowledge machine cycle the 8086 sends out another pulse on its $\overline{INTA}$ output pin. In response to this second $\overline{INTA}$ pulse the external device puts the interrupt type on lower 8 bits of the data bus.

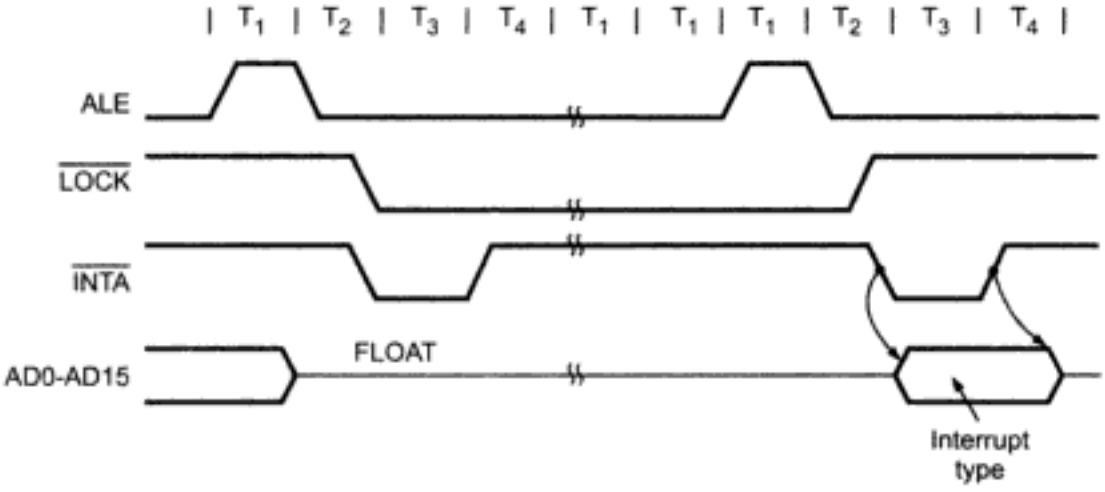


Fig. 8.3 Interrupt acknowledge machine cycle

2. Once the 8086 receives the interrupt type, it pushes the flag register on the stack, clears TF and IF, and pushes the CS and IP values of the next instruction on the stack.
3. The 8086 then gets the new value of IP from the memory address equal to 4 times the interrupt type (number), and CS value from memory address equal to 4 times the interrupt number plus 2.

8.4 Interrupt Priorities

As far as the 8086 interrupt priorities are concerned, software interrupts (All interrupts except single step, NMI and INTR interrupts) have the highest priority, followed by NMI followed by INTR. Single step has the least priority.

Interrupt	Priority
Divide Error, Int n, Int 0	HIGHEST
NMI	↓
INTR	↓
SINGLE - STEP	LOWEST

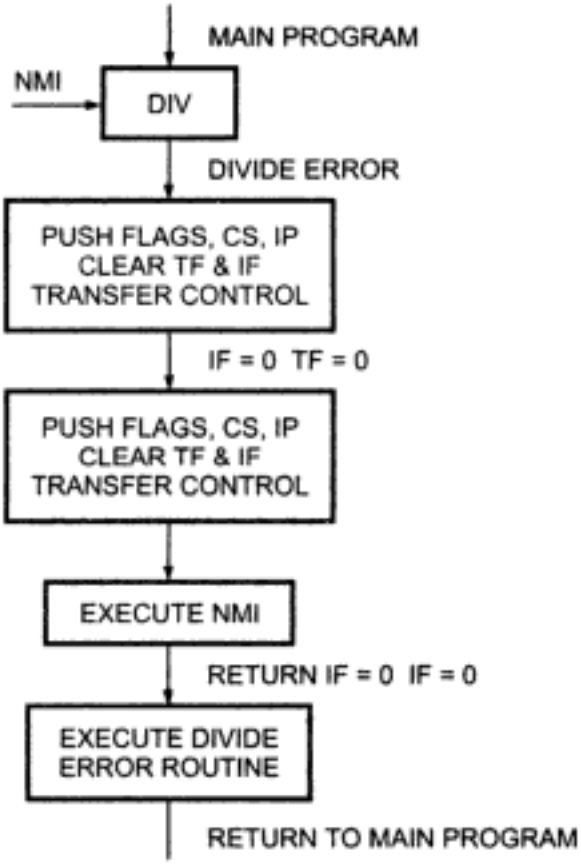


Fig. 8.4 Flow-chart for divide error routine

The interrupt flag is automatically cleared as part of the response of an 8086 to an interrupt. This prevents a signal on the INTR input from interrupting a higher priority interrupt service routine. The 8086 allows NMI input to interrupt higher priority interrupt, for example suppose that a rising edge signal arrives at the NMI input while the 8086 is executing a DIV instruction, and that the division operation produces a divide error. Since the 8086 checks for internal interrupts before it checks for an NMI interrupt, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go to the start of the divide error service routine. The 8086 will then do an NMI interrupt response and execute non-maskable interrupt service routine. After completion of NMI service routine an 8086 will return to the divide error routine. It will execute divide error routine and then it will return to the main program (Refer Fig. 8.4).

8.5 Expanding Interrupt Structure using PIC 8259

Interrupts can be used for a variety of applications. Each of these interrupt applications requires a separate interrupt input. If we are working with an 8086, we get only two interrupt inputs INTR and NMI. For applications where we have multiple interrupt sources, we use external device called a priority interrupt controller (PIC). Fig. 8.5 shows the connection between 8086 and 8259.

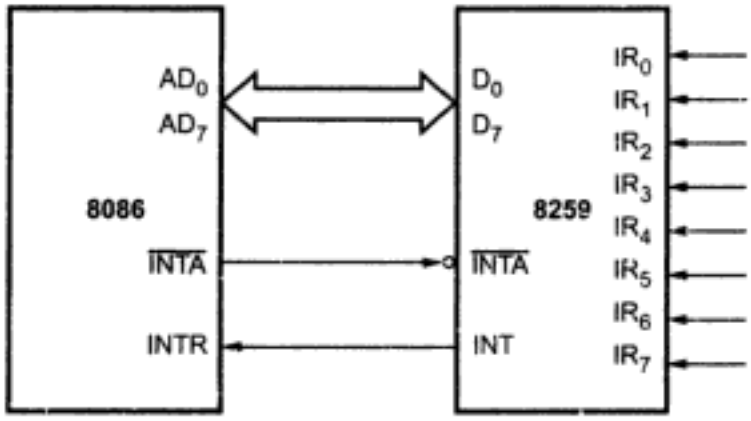


Fig. 8.5 Connection between 8086 and 8259

8.5.1 Features of 8259

1. It can manage eight priority interrupts. This is equivalent to provide eight interrupt pins on the processor in place of INTR pin.
2. It is possible to locate vector table for these additional interrupts any where in the memory map. However, all eight interrupts are spaced at the interval of either four or eight locations.
3. By cascading 8259s it is possible to get 64 priority interrupts.
4. Interrupt mask register makes it possible to mask individual interrupt request.
5. The 8259A can be programmed to accept either the level triggered or the edge triggered interrupt request.
6. With the help of 8259A user can get the information of pending interrupts, in-service interrupts and masked interrupts.
7. The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts.

8.5.2 Block Diagram of 8259A

Fig. 8.6 shows the internal block diagram of the 8259A. It includes eight blocks : data bus buffer, read/write logic, control logic, three registers (IRR, ISR and IMR), priority resolver, and cascade buffer.

Data Bus Buffer

The data bus allows the 8086 to send control words to the 8259A and read a status word from the 8259A and read a status word from the 8259A. The 8-bit data bus also allows the 8259A to send interrupt types to the 8086.

Hidden page

Priority Resolver

The priority resolver determines the priorities of the bits set in the IRR. The bit corresponding to the highest priority interrupt input is set in the ISR during the $\overline{INTA}$ input.

Cascade Buffer Comparator

This section generates control signals necessary for cascade operations. It also generates Buffer-Enable signals. As stated earlier, the 8259 can be cascaded with other 8259s in order to expand the interrupt handling capacity to sixty-four levels. In such a case, the former is called a **master**, and the latter are called **slaves**. The 8259 can be set up as a master or a slave by the $\overline{SP}$ / $\overline{EN}$ pin.

CAS 0 - 2

For a master 8259, the CAS_0-CAS_2 pins are outputs, and for slave 8259s, these are inputs. When the 8259 is a master (that is, when it accepts interrupt requests from other 8259s), the CALL opcode is generated by the Master in response to the first $\overline{INTA}$. The vectoring address must be released by the slave 8259. The master sends an identification code of three-bits (to select one out of the eight possible slave 8259s) on the CAS_0-CAS_2 lines. The slave 8259s accept these three signals as inputs (on their CAS_0-CAS_2 pins) and compare the code sent by the master with the codes assigned to them during initialisation. The slave thus selected (which had originally placed an interrupt request to the master 8259) then puts out the address of the interrupt service routine during the second and third $\overline{INTA}$ pulses from the CPU.

$\overline{SP}$ / $\overline{EN}$ (Slave Program /Enable Buffer)

The $\overline{SP}$ / $\overline{EN}$ signal is tied high for the master. However, it is grounded for the slave.

In large systems where buffers are used to drive the data bus, the data sent by the 8259 in response to $\overline{INTA}$ cannot be accessed by the CPU (due to the data bus buffer being disabled).

If an 8259 is used in the buffered mode (buffered or non-buffered modes of operation can be specified at the time of initialising the 8259), the $\overline{SP}$ / $\overline{EN}$ pin is used as an output which can be used to enable the system data bus buffer whenever the 8259's data bus outputs are enabled (when it is ready to send data).

Means, in non-buffered mode, the $\overline{SP}/\overline{EN}$ pin of an 8259 is used to specify whether the 8259 is to operate as a master or as a slave, and in the buffered mode, the $\overline{SP}/\overline{EN}$ pin is used as an output to enable the data bus buffer of the system.

8.5.3 Interrupt Sequence

The events occur as follows in an 8086 system :

1. One or more of the INTERRUPT REQUEST lines (IR0-IR7) are raised high, setting the corresponding IRR bit(s).

2. The priority resolver checks three registers : The IRR for interrupt requests, the IMR for masking bits, and the ISR for the interrupt request being served. It resolves the priority and sets the INT high when appropriate.
3. The CPU acknowledges the INT and responds with an $\overline{INTA}$ pulse.
4. Upon receiving an $\overline{INTA}$ from the CPU, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive data bus during this cycle.
5. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the 8259A can be configured to match his system requirements. The priority modes can be changed or reconfigured dynamically at any time during the main program. This means that the complete interrupt service structure can be defined as required, based on the total system environment.
6. The 8086 will initiate a second INTA pulse. During this pulse, the 8259A releases a 8-bit pointer (interrupt type) onto the Data Bus where it is read by the CPU.
7. This completes the interrupt cycle. In the AEOI mode the ISR bit is reset at the end of the second INTA pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt subroutine.

8.5.4 Priority Modes and Other Features

The various modes of operation of the 8259 are :

(a) Fully Nested Mode,

(b) Rotating Priority Mode,

(c) Special Masked Mode, and

(d) Polled Mode.

a) Fully Nested Mode :

After initialization, the 8259A operates in fully nested mode so it is called as default mode. The 8259 continues to operate in the Fully Nested Mode until the mode is changed through Operation Command Words. In this mode, IR0 has highest priority and IR7 has lowest priority. When the interrupt is acknowledged, it sets the corresponding bit in ISR. This bit will prevent all interrupts of the same or lower level, however it will accept higher priority interrupt requests. The vector address corresponding to this interrupt is then sent. The bit in the ISR will remain set until an EOI command is issued by the microprocessor at the end of interrupt service routine.

But if AEOI (Automatic End of Interrupt) bit is set, the bit in the ISR resets at the trailing edge of the last $\overline{INTA}$.

Hidden page

(i) Automatic Rotation

In this mode, a device, after being serviced, receives the lowest priority. Assuming that IR_3 has just been serviced, it will receive the seventh priority.

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
4	5	6	7	0	1	2	3

(ii) Specific Rotation

In the Automatic Rotation mode, the interrupt request last serviced is assigned the lowest priority, whereas in the Specific Rotation mode, the lowest priority can be assigned to any interrupt input (IR_0 to IR_7) thus fixes all other priorities.

For example if the lowest priority is assigned to IR_2, other priorities are as shown below.

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
5	6	7	0	1	2	3	4

d) Special Mask Mode :

If any interrupt is in service then the corresponding bit is set in ISR and the lower priority interrupts are inhibited. Some applications may require an interrupt service routine to dynamically alter the system priority structure during its execution under software control, for example, the routine may wish to inhibit lower priority requests for a portion of its execution but enable some of them for another portion. In these cases we have to go for special mask mode.

In the special mask mode it inhibits further interrupts at that level and enables interrupts from all other levels (lower as well as higher) that are not masked. Thus any interrupt may be selectively enabled by loading the mask register.

e) Poll Mode :

In this mode the INT output is not used. The microprocessor checks the status of interrupt requests by issuing poll command. The microprocessor reads contents of 8259A after issuing poll command. During this read operation the 8259A provides polled word and sets ISR bit of highest priority active interrupt request FORMAT.

I	X	X	X	X	W_2	W_1	W_0

$I = 1 \rightarrow$ One or more interrupt requests activated.

$I = 0 \rightarrow$ No interrupt request activated.

$W_2\ W_1\ W_0 \rightarrow$ Binary code of highest priority active interrupt request.

8.5.5 Programming the 8259A

The 8259A requires two types of command words. Initialization Command Words (ICWs) and Operational Command Words (OCWs).

The 8259A can be initialized with four ICWs; the first two are compulsory, and the other two are optional based on the modes being used. These words must be issued in a given sequence. After initialization, the 8259A can be set up to operate in various modes by using three different OCWs; however, they no longer need to be issued in a specific sequence.

Flow chart :

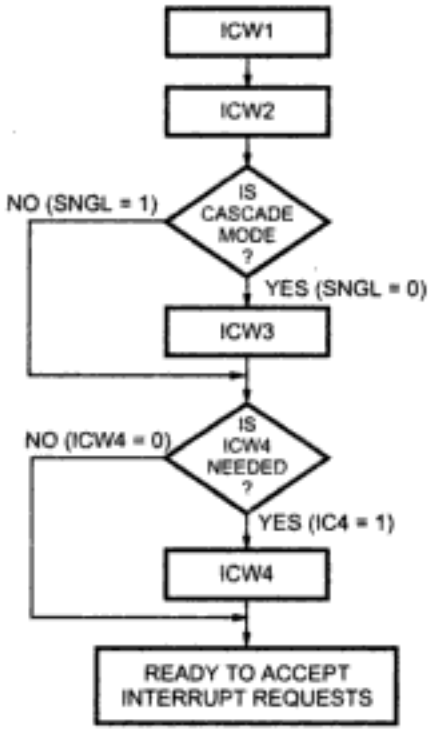


Fig. 8.7 8259 A initialization flowchart

Initialization Command Word 1 (ICW1)

Fig. 8.8 shows the Initialization Command Word 1 (ICW1).

A write command issued to the 8259 with $A_0 = 0$ and $D_4 = 1$ is interpreted as ICW1, which starts the initialization sequence.

It specifies

1. Single or multiple 8259As in the system.
2. 4 or 8 bit interval between the interrupt vector locations.
3. The address bits A_7 - A_5 of the CALL instruction.
4. Edge triggered or level triggered interrupts.
5. ICW4 is needed or not.

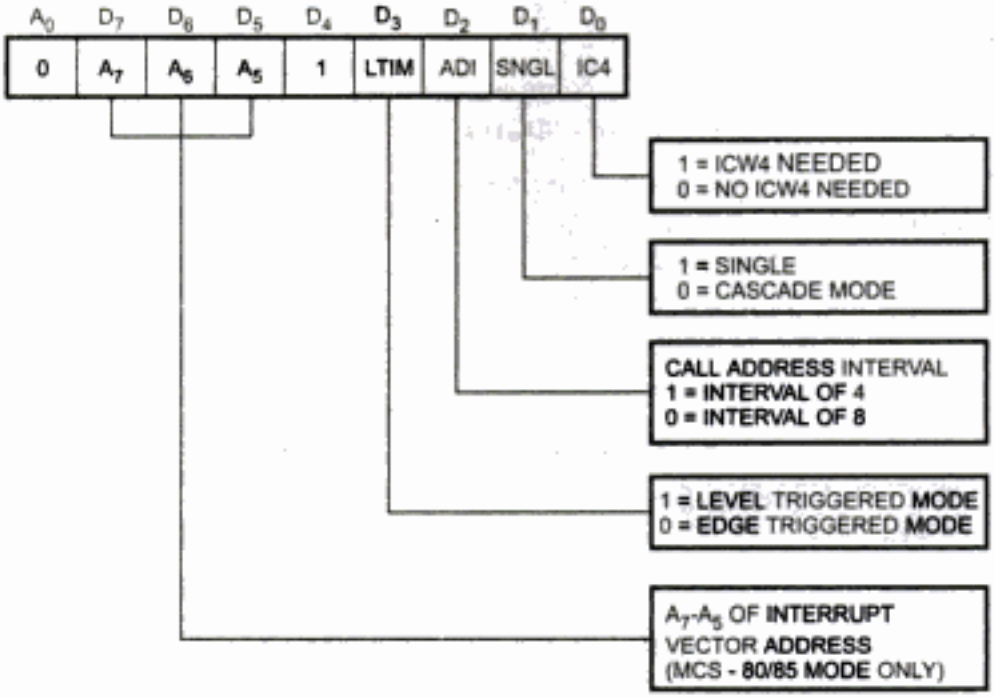


Fig. 8.8 Initialization command word 1 (ICW1)

Initialization Command Word 2 (ICW2)

Fig 8.9 shows the Initialization Command Word 2 (ICW2).

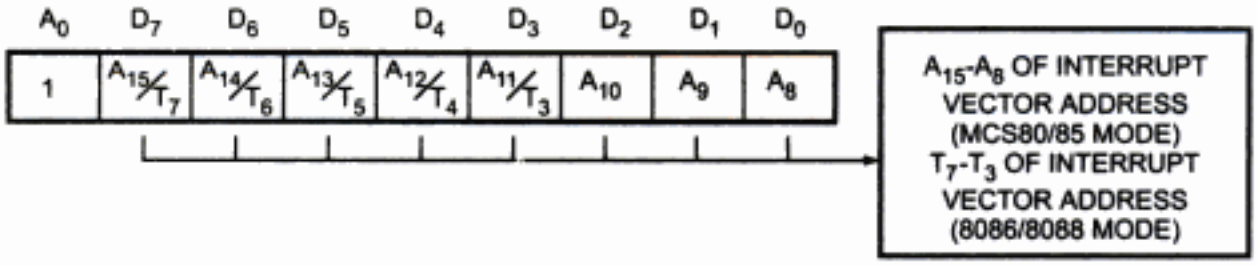


Fig. 8.9 Initialization command word 2 (ICW2)

A write command following ICW1, with A0 = 1 is interpreted as ICW2. This is used to load the high order byte of the interrupt vector address of all the interrupts.

Initialization Command Word 3 (ICW3)

ICW3 is required only if there is more than one 8259 in the system and if they are cascaded. An ICW3 operation loads a slave register in the 8259. The format of the byte to be loaded as an ICW3 for a master 8259 or a slave is shown in the Fig. 8.10. For master, each bit in ICW3 is used to specify whether it has a slave 8259 attached to it on its corresponding IR (Interrupt Request) input. For slave, bits D_0-D_2 of ICW3 are used to assign a slave identification code (slave ID) to the 8259.

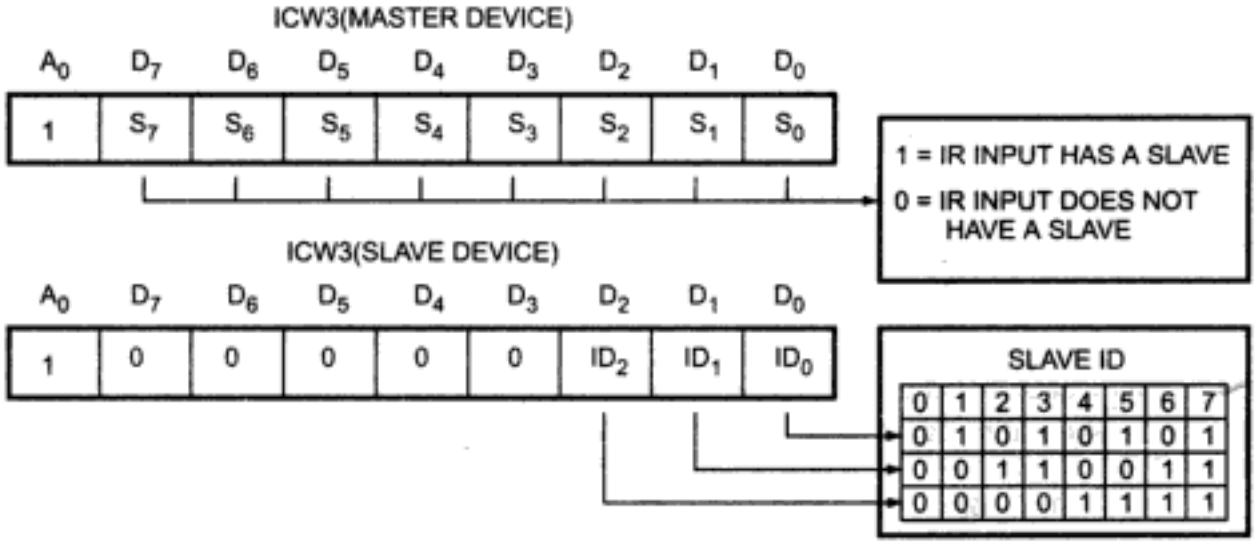


Fig. 8.10 Initialization command word 3 (ICW3)

Initialization Command Word 4 (ICW4)

It is loaded only if the D_0 bit of ICW1 (IC 4) is set. The format of ICW4 is shown in Fig. 8.11.

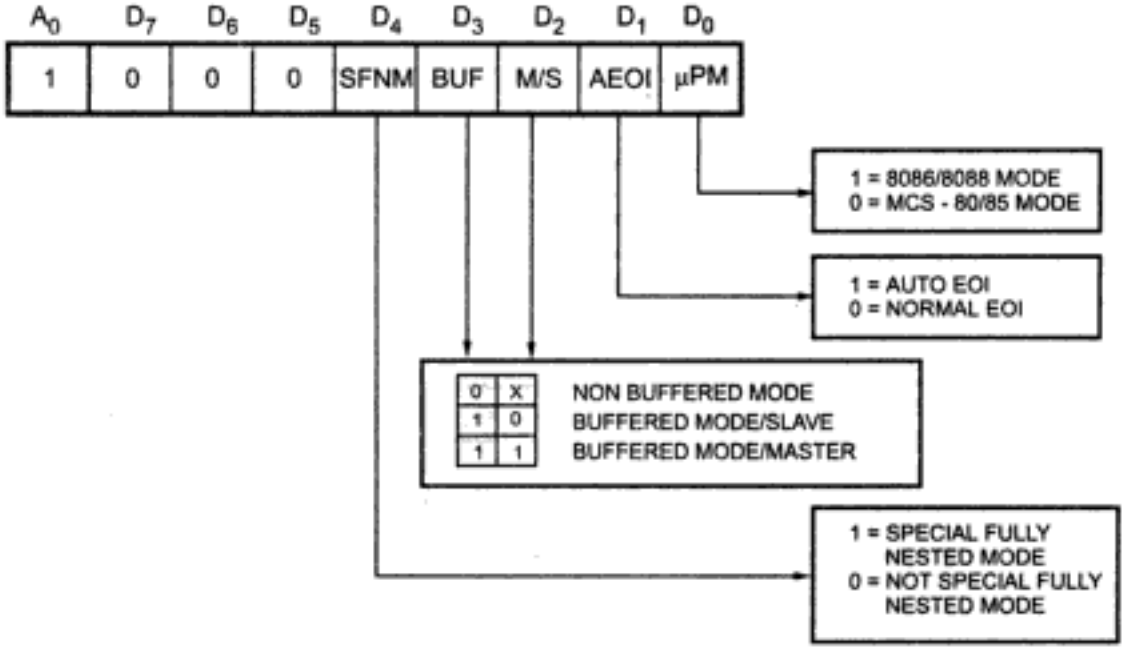


Fig. 8.11 Initialization command word 4 (ICW4)

It specifies.

1) Whether to use special fully nested mode or non special fully nested mode.
2) Whether to use buffered mode or non buffered mode.
3) Whether to use Automatic EOI or Normal EOI
4) CPU used, 8086/8088 or 80810.

After initialisation, the 8259 is ready to process interrupt requests. However, during operation, it might be necessary to change the mode of processing the interrupts. Operation Command Words (OCWs) are used for this purpose. They may be loaded anytime after the 8259's initialisation to dynamically alter the priority modes.

Operation Command Word 1 (OCW1)

A Write command to the 8259 with A_0 = 1 (after ICW2) is interpreted as OCW1. OCW1 is used for enabling or disabling the recognition of specific interrupt requests by programming the IMR.

M = 1 indicates that the interrupt is to be masked, and M = 0 indicates that it is to be unmasked as shown in Fig. 8.12.

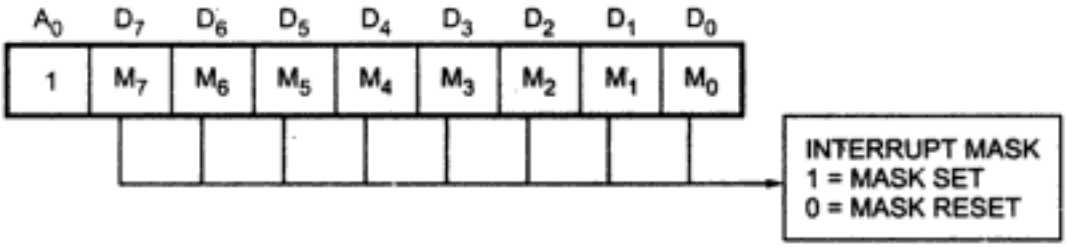


Fig. 8.12 Operation command word 1 (OCW1)

Operation Command Word 2 (OCW2)

A Write command with A_0 = 1 and D_4 D_3 = 00 is interpreted as OCW2. The R(Rotate), SL (Select-Level), EOI bits control the Rotate and End Of Interrupt Modes and combinations of the two. Fig. 8.13 shows the Operation Command Word format. $L_2 - L_0$ are used to specify the interrupt level to be acted upon when the SL bit is active.

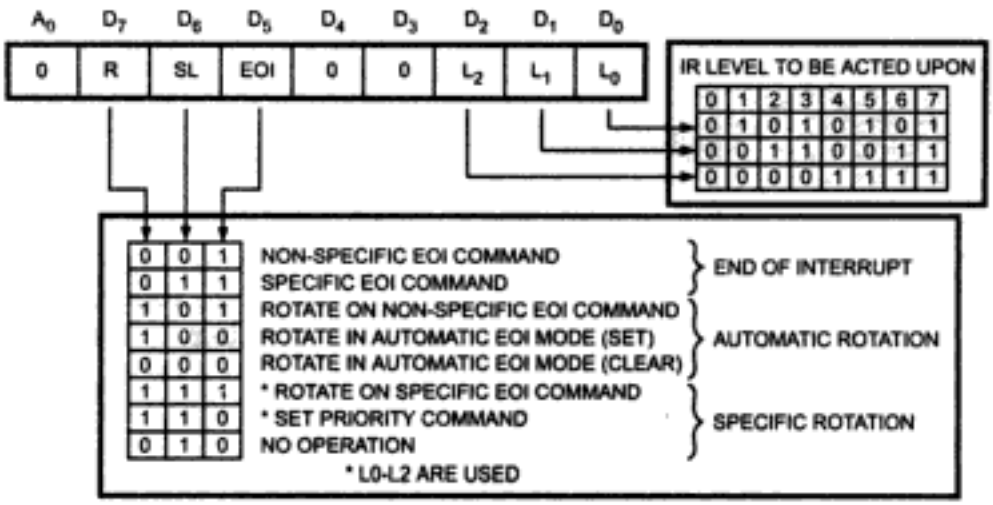


Fig. 8.13 Operation command word 2 (OCW2)

Hidden page

IMR Status Read

A Read command issued to the 8259 with $A_0 = 1$ (with $\overline{RD}$, $\overline{CS} = 0$) causes the 8259 to put out the contents of the Interrupt Mask Register. OCW3 is not required for a status read of the IMR.

As described earlier, the sequence shown in flowchart (Fig. 8.4) must be followed to initialize 8259A. According to this flow chart an ICW1 and an ICW2 must be sent to any 8259A in the system. If the system has any slave 8259As (cascade mode) then an ICW3 must be sent to the master, and a difference ICW3 must be sent to the slave. If the system is an 8086, or if you want to specify certain special conditions, then you have to send an ICW4 to the master and to each slave. To have better understanding the initiation sequences for different specification are given in the next section.

Note : It is assumed that A_1 of the system bus is connected to the A_0 of the 8259A. So the internal addresses correspond to 0 and 2. It is also assumed that the base address of the device is 40H. So the two system addresses for the 8259A are 40H and 42H.

Example 1 : *Write the initialization instructions for 8259A interrupt controller to meet the following specifications :*
a) Interrupt type 32. b) Edge triggered, single and ICW4 needed.
c) Mask interrupts IR1 and IR3.

Solution :

ICW1

A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC4
0	0	0	1	0	0	1	1

= 13H

Note : When used with an 8086, bit D_2, D_5, D_6 and D_7 are don't care, so we make them 0's for simplicity.

ICW2

In an 8086 system ICW2 is used to tell the 8259A the type number to send in response to an interrupt signal on the IR0 input.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	1	0	0	0	0	0

= 20H = 32 Decimal

ICW2 for sending interrupt type 32 to the 8086 in response to an IR0 interrupt is 20H

Note : For an IR1 input the 8259A will send 00100001 binary (33 decimal) and so on for the other IR inputs.

ICW3

Since we are not using a slave in our example, we don't need to send an ICW3.

ICW4

For our example, the only reason we need to send an ICW4 is to let the 8259A know that it is operating in an 8086 system. We do this by making bit D_0 of the ICW4 one.

OCW1

An OCW1 must be sent to an 8259A to unmask any IR inputs. For our example we want to mask IR1 and IR3, so we put 1's in these two bits and 0's in the rest of the bits.

M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0	
0	0	0	0	1	0	1	0	= 0AH

Program :

```
MOV AL,13H    ; edge triggered, single, ICW4 needed
OUT 40H,AL    ; Send ICW1
MOV AL,20H    ; type 32 is first 8359A type
OUT 41H,AL    ; send ICW2
MOV AL,01H    ; ICW4, 8086 mode.
OUT 41H,AL    ; send ICW4
MOV AL,0AH    ; OCW1 to mask IR1 and IR3
OUT 41H,AL    ; send OCW1
```

Example 2 : *Write the initialization instructions for master and slave configuration to meet the following specifications :*

1) The INTR of slave is routed through IR2 of the master 8259A to the 8086.
2) Master and slave are both level triggered.
3) First interrupt types for master and slave are 32 and 64 respectively.
4) Modes : automatic rotation and auto end of interrupt.
5) Addresses of the master are 40H and 41H and the slave are 80H and 81H.
6) Buffers are not used.
Initialization command words for Master ICW1 (Master)

ICW1 (master)

A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC4	
0	0	0	1	1	0	0	1	= 10H

ICW2 (master)

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
0	0	1	0	0	0	0	0	= 20H

ICW3 (master)

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0	
0	0	0	0	0	1	0	0	= 04H

ICW4 (master)

0	0	0	SFNM	BUF	M/S	AEOI	μPM	
0	0	0	0	0	0	1	1	= 03H

Program :

```
MOV AL, 19H  ; level triggered, cascaded, ICW4 needed
OUT 40H, AL  ; send ICW1 (master)
MOV AL, 20H  ; type 32 is first 8259A type
OUT 41H, AL  ; send ICW2 (master)
MOV AL, 04H  ; slave at IR2
OUT 42H, AL  ; send ICW3 (master)
MOV AL, 03H  ; ICW4, 8086 mode, and set AEOI
OUT 41H, AL  ; send ICW4 (master)
MOV AL, 19H  ; level triggered, cascaded, ICW4 needed
OUT 80H, AL  ; send ICW1 (slave)
MOV AL, 40H  ; type 64 is first 8259A type
OUT 81H, AL  ; send ICW3 (slave)
MOV AL, 02H  ; ID for slave connected to IR2
OUT 81H, AL  ; send ICW2 (slave)
MOV AL, 01H  ; ICW4, 8086 mode
OUT 81H, AL  ; send ICW4
MOV AL, 80H  ; OCW2 (rotate in auto EOI mode set command)
OUT 80H, AL  ; send OCW2 (slave)
```

8.5.6 8259A Interfacing

Fig. 8.15 shows that how an 8259A can be interfaced with the 8086 microprocessor system in minimum mode. In case of 8088 microprocessor same interfacing diagram can be used except M/$\overline{IO}$ signal. In 8088, M/$\overline{IO}$ signal is represented by IO/$\overline{M}$ signal, therefore this signal is connected to G (active high) signal of decoder to interface 8259A in I/O mapped I/O mode.

Addressing of 8259A :

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
1	1	1	1	1	1	1	1	1	1	1	1	0	0	X	0	FFF0H
F				F				F				0/2				FFF2H

The 74LS138 address decoder will assert the $\overline{CS}$ input of the 8259A when an I/O base address is FFF0H or FFF2H on the address bus. The A_0 input of the 8259A is used to select one of the two internal addresses in the device. A_0 of the 8259A is connected to system line A1. So the system addresses for the two internal addresses are FFF0H and FFF2H. The data lines of an 8259A are connected to the lower half of the system data bus, because the 8086 expects to receive interrupt types on these lower eight data lines. $\overline{RD}$ and $\overline{WR}$ signals are connected to the system $\overline{RD}$ and $\overline{WR}$ lines. The interrupt request signal

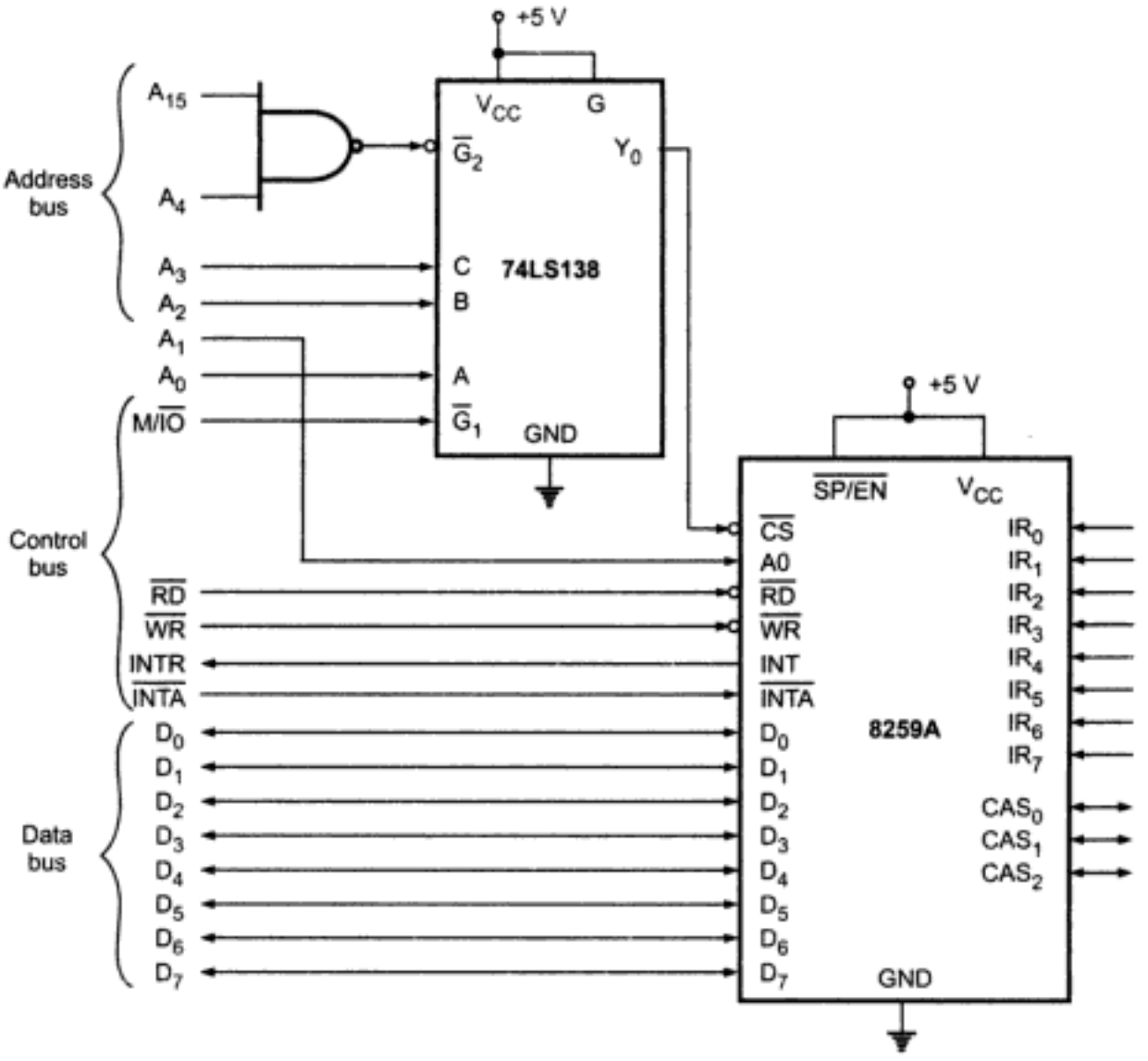


Fig. 8.15 8259A interface to 8086 system bus

INT from the 8259A is connected to the INTR input of the 8086 and $\overline{INTA}$ from the 8086 is connected to $\overline{INTA}$ on the 8259A. As we are using single 8259A in the system $\overline{SP}/\overline{EN}$ pin is tied high and CAS_0-CAS_2 lines are left open. The eight IR inputs are available for interrupt signals.

Note :

1. Unused IR inputs should be tied to ground so that a noise pulse cannot accidentally cause an interrupt.
2. In maximum mode RD and INTA signals of 8259A are connected to the IORC, IOWC and INTA lines of 8288 bus controller.

Cascading :

The 8259A can be easily interconnected to get multiple interrupts. Fig. 8.16 shows how 8259A can be connected in the cascade mode. In cascade mode one 8259A is configured in Master mode and other should be configured in the Slave mode. In this figure 8259A-1 is in the master mode and others are in slave mode. Each slave 8259A is identified by the number which is assigned as a part of its initialization. Since the 8086 has only one INTR input, only one of the 8259A INT pins is connected to the 8086 INTR pin. The 8259A connected directly into the 8086 INTR pin is referred as the master. The INT pins from other 8259A are connected to the IR inputs of the master 8259A. These cascaded 8259As are referred as slav :. The INTA signal is connected to both master and slave 8259A.

(See Fig. 8.16 on next page.)

The cascade pins CAS_0 to CAS_2 are connected from the master to the corresponding pins of the slave. For the master these pins function as outputs, and for the slave these pins function as inputs. The $\overline{SP}/\overline{EN}$ signal is tied high for the master. However it is grounded for the slave.

Each 8259A has its own addresses so that command words can be written to it and status bytes read from it.

Addresses for 8259As :

No	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
8259A-1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	X	0	FFF0H
	F				F				F				0/2				FFF2H
8259A-2	1	1	1	1	1	1	1	1	1	1	1	1	0	1	X	0	FFF4H
	F				F				F				4/6				FFF6H
8259A-3	1	1	1	1	1	1	1	1	1	1	1	1	1	0	X	0	FFF8H
	F				F				F				8/A				FFFAH

Master and slave operation :

When the slave receives an interrupt signal on one of its IR inputs, it checks mask condition and priority of the interrupt request. If the interrupt is unmasked and its priority is higher than any other interrupt level being serviced in the slave, then the slave will send an INT signal to the IR input of a master. If that IR input of the master is unmasked and if that input is a higher priority than any other IR inputs currently being serviced, then the master will send an INT signal to the 8086 INTR input. If the INTR interrupt is enabled, the 8086 will go through its INTR interrupt procedure and sends out two $\overline{INTA}$ pulses to both the master and the slave. The slave ignores the first interrupt acknowledge pulse but the master outputs a 3-bit slave identification number on the CAS_0-CAS_2 lines. Sending the 3-bit ID number enables the slave. When the slave receives the second $\overline{INTA}$ pulse from the 8086, the slave will send the desired type number to the 8086 on the eight data lines.

If an interrupt signal is applied directly to one of the IR inputs of the master, the master will send the desired interrupt type to the 8086 when it receives the second $\overline{INTA}$ pulse from the 8086.

Hidden page

8.6 Interrupt Example

There are several reasons for writing interrupt service routine. However, to write such a interrupt service routine we have set the address of our interrupt service routine in the interrupt vector table. To set an interrupt vector to a specified address, (starting address of interrupt service routine) there are two ways :

1. Using function 25H of INT 21H
2. Without using any DOS function.

1. INT21H, Function 25H : Set Interrupt Address

To set a new interrupt address, load the required interrupt number in the AL and the new address in the DX :

```
MOV AH, 25H        ; Request interrupt address
MOV AL, int #      ; Interrupt number
LEA DX, newaddr    ; New address for interrupt
INT 21H
```

The above program replaces the present address of the interrupt with the new address. In effect, then, when the specified interrupt occurs, processing links to resident program, rather than to the normal interrupt address.

2. Without using any DOS Function

The DOS function discussed above do nothing more than getting address of interrupt vector corresponding to an interrupt number and loading two words (segment address and offset address of the interrupt service routine) into it. The address of interrupt vector can be obtained by multiplying the interrupt number by 4. Once we get the address of the Interrupt vector table we have load the segment address and offset address of the interrupt service routine.

➠ **Example 3 :** *Generate a real time clock by generating a periodic interrupt request signal on the $\overline{NMI}$ input of 8086.*

Solution : Hardware : The Fig. 8.17 shows simple circuit that generates interrupt request after every 0.5 sec.

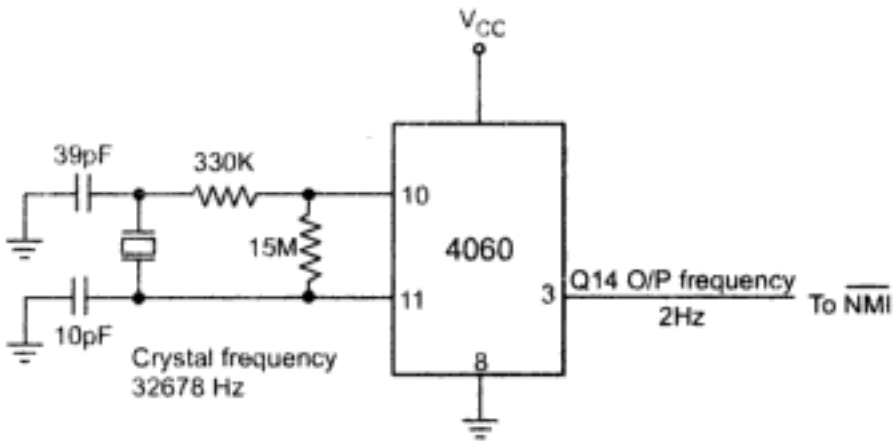


Fig. 8.17 Interrupt generation circuit

Hidden page

```
          JNZ  DONE
          MOV  HR,00H     ; Reset HR = 00
DONE:     POP  SI         ; restore registers
          MOV  AH,00
          IRET
          TIMES ENDP
          END  START
          END
```

Review Questions

1. *What do you mean by interrupt ?*
2. *What is interrupt service routine ?*
3. *What are the sources of interrupts in 8086 ?*
4. *What is interrupt vector table ?*
5. *Draw and explain the IVT for 8086.*
6. *Briefly describe the conditions which cause the 8086 to perform each of the following types of interrupts : Type 0, Type 1, Type 2, Type 3 and Type 4.*
7. *Explain interrupt structure of 8086.*
8. *What are software interrupt ? How 8086 responds to software interrupts ?*
9. *Draw and explain the interrupt acknowledge cycle of 8086.*
10. *Describe the response of 8086 to the interrupt coming on pin.*
11. *What do you mean by interrupt priorities ?*
12. *State the interrupt priorities for 8086 interrupts.*
13. *What are advantages of using 8259 ?*
14. *List the features of 8259.*
15. *Explain the operating modes of 8259.*
16. *Draw and explain the interfacing of 8259 with 8086.*
17. *Draw and explain the interfacing of cascaded 8259s with 8086.*
18. *Explain the procedure of interrupt programming.*

❑❑❑

9

Introduction to DOS and BIOS Interrupts

In IBM PC, part of the operating system is located in the permanent memory (ROM) and part is loaded during power up. The part located in ROM is referred to as **ROM-BIOS** (Basic Input/Output System). The other part which is loaded in RAM during power-up from harddisk or floppy disk is known as **DOS** (Disk Operating System).

BIOS is located in an 8K-byte ROM at the top of memory, the address range being from FE000H to FFFFFH. The programs within ROM-BIOS provide the most direct, lowest level interaction with the various devices in the system. The ROM-BIOS contains routines for

1. Power-on self test
2. System configuration analysis
3. Time-of-day
4. Print screen
5. Bootstrap loader
6. I/O support program for
    a. Asynchronous communication
    b. Keyboard
    c. Diskette
    d. Printer
    e. Display

Most of these programs are accessible to the assembly-language programmer through the software interrupt instruction (INT). The design goal for the ROM-BIOS programs is to provide a device-independent interface to the various physical devices in the system.

It is seen that ROM-BIOS provides basic low-level services. Using ROM-BIOS one can output characters to various physical devices like the printer or the display monitor, one can read characters from keyboard, one can read or write sectors of data to the diskette. But still few things we can't do with ROM-BIOS.

1. It is not possible to provide ability to load and execute programs directly.
2. It is not possible to store data on the diskette organized as logical files.
3. ROM-BIOS has no command-interpreter to allow us to copy files, print files, delete files.

It is DOS that provides these services. When we turn our computer ON, we expect to see a message or a prompt. We except to be able to look at the diskette directory to see what data files or programs the diskette contains. We expect to run a program by typing its name. We want to copy programs from one diskette to another, print programs, and delete programs. All these services are provided by group of programs called DOS. The services provided by DOS can be grouped into following categories.

1. Character Device I/O : This group includes routines that input or output characters to character oriented devices such as the printer, the display monitor, and the keyboard.

2. File Management : This group includes routines that manage logical files, allowing you to create, read, write and delete files.

3. Memory Management : This group includes routines that allow us to change, allocate, and deallocate memory.

4. Directory Management : This group includes routines that permit us to create, change search, and delete directories.

5. Executive Functions : This group includes routines that allow us to load and execute programs, to overlay programs, to retrieve error codes from completed programs, and to execute commands.

6. Command Interpreter : This routine is in action whenever a prompt is present on the screen. It interprets commands and executes DOS functions, utility programs, application programs, depending upon the command.

7. Utility Programs : These programs facility to copy, delete provides the DISKCOPY, DIR and many other DOS commands.

Hidden page

Int 21H Direct console I/O Function 06H

Used by program that need to read and write all possible characters and control codes without any interference from the operating system.

Reads a character from the standard input device or writes a character to the standard output device. I/O may be redirected.

Calling parameters

```
AH = 06H
DL = function  requested
     00H-FEH   if output request
     0FFH      if input request
```

Returns :

```
Nothing, if called with DL = 00H-0FEH
If called with DL = FFH and a character is ready returns
Zero flag = clear
     AL = 8-bit input data
If called with DL = FFH and no character is ready
Zero flag = set
```

Int 21H Unfiltered character input without echo Function 07H

Reads a character from the standard input device without echoing it to the standard output device. If no character is ready, waits until one is available.

Calling Parameter

```
AH = 07H
```

Returns

```
AL = 8-bit input data
```

Example : Read a character from the standard input without echoing it to the display, and store it in the variable char.

```
char    db  0          ; input character
        .
        .
        mov ah,7       ; function number
        int 21h        ; transfer to MS-DOS
        mov char,al    ; save character
        .
        .
```

Hidden page

If the buffer fills to one fewer than the maximum number of characters it can hold, subsequent input is ignored and the bell is sounded until a carriage return is detected.

Example : Read a string that is maximum of 80 characters long from the standard input device, placing it in the buffer named buffer

```
buffer   db   81                ; maximum length of input
         db   0                 ; actual length of input
         db   81 dup (0)        ; actual input placed here

         .
         .
         mov  ah,0ah            ; function number
         mov  dx,seg buffer     ; input buffer address
         mov  ds,dx
         mov  dx,offset buffer
         int  21h               ; transfer to MS-DOS
         .
         .
```

Int 21H — Check input status — Function 0BH (11)

Checks whether a character is available from the standard input device.

Calling Parameter

```
AH = 0BH
```

Returns

```
AL = 00H if no character is available
```

FFH if at least one character is available

Example : Test whether a character is available from the standard input.

```
.
.
mov  ah,0bh        ; function on number
int  21h           ; transfer to MS-DOS
or   al,al         ; character waiting?
jnz  avail         ; jump if char available
.
.
```

Int 21H **Flush input buffer and then input** **Function 0CH (12)**

Clears the standard input buffer and then invokes one of the character input functions. Input can be redirected.

Calling parameters

```
AH = 0CH
AL = number of input function to be invoked
     after resetting buffer (must be 01H, 06H,
     07H, 08H, or 0AH)
(if AL = 0AH)
DS:DX = segment:offset of input buffer
```

Returns : (if called with AL = 01H, 06H, 07H, or 08H)

```
AL = 8-bit input data
```

(if called with AL = 0AH)

Nothing (data placed in buffer)

9.2 Character Display Functions

Int 21H **Character output** **Function 02H**

Outputs the character to the standard output device.

Calling Parameters

```
AH = 02H
DL = 8-bit data for output
```

Returns : Nothing

Example: Send the character "*" to the standard output device.

```
.
.
mov ah,2          ; function number
mov dl,'*'        ; character to output
int 21h           ; transfer to MS-DOS
.
.
```

Int 21H **Display string** **Function 09H**

Sends a string of characters to the standard output device. End of string is indicated by character $ (24H).

Calling Parameters

```
AH = 09H
DS = segment:offset of string
```

Returns : Nothing

Example : Send the string, followed by a carriage return and line feed, to the standard output device.

```
cr   equ  0dh
lf   equ  0ah
msg  db   'MICROPROCESSOR',cr,lf,'$'
     .
     .
     mov  ah,09h        ; function number
     mov  dx,seg msg    ; address of string
     mov  ds,dx
     mov  dx,offset msg
     int  21h           ; transfer to MS-DOS
```

9.3 File Control Block Functions

Int 21H **Open file** **Function 0FH (15)**

Opens a file and makes it available for subsequent read/write operations.

Calling Parameters

```
   AH  =  0FH
DS:DX  =  segment:offset of file control block
```

Returns :

If function successful (file found)

```
AL  =  00H
```

and FCB filled in by MS-DOS as follows :

drive field (offset 00H) = 1 for drive A, 2 for drive B, etc.
current block field (offset 0CH) = 00H
record size field (offset 0EH) = 0080H
[2.0+] size field (offset 10H) = file size from directory
[2.0+] date field (offset 14H) = date stamp from directory
[2.0+] time field (offset 16H) = time stamp from directory

If function unsuccessful (file not found)

```
AL = FFH
```

Hidden page

```
        mov  ds,  dx
        mov  dx,  offset  myfcb
        int  21h                ;  transfer  to  MS-DOS
        or   al,  al            ;  check  status
        jnz  error              ;  jump  if  close  failed
          .
          .
```

Int 21H — Delete file — Function 13H (19)

Deletes all matching files from the current directory on the default or specified disk drive.

Calling parameters

```
        AH = 13H
    DS:DX = segment:offset of file control block
```

Returns :

If function successful (file or files deleted)

```
        AL = 00H
```

If function unsuccessful (no matching files were found, or at least one matching file was read-only)

```
        AL = FFH
```

Example :

Delete the file TEST.DAT from the current disk drive and directory.

```
myfcb   db   0                  ;  drive  =  default
        db   'TEST'             ;  filename,  8  characters
        db   'DAT'              ;  extension,  3  characters
        db   25  dup  (0)       ;  remainder  of  FCB
          .
          .
        mov  ah,13h             ;  function  number
        mov  dx,seg  myfcb      ;  address  of  FCB
        mov  ds,dx
        mov  dx,offset  myfcb
        int  21h                ;  transfer  to  MS-DOS
        or   al,al              ;  check  status
        jnz  error              ;  jump  if  close  failed
          .
          .
```

Int 21H Sequential read Function 14H (20)

Reads the next sequential block of data from a file, then increments the file pointer appropriately.

Calling parameters

```
AH = 14H
DS:DX = segment:offset of previously opened file
        control block
```

Returns

```
AL = 00H  if read successful
     01H  if end of file
     02H  if segment wrap
     03H  if partial record read at end of file
```

Example : Read 512 bytes of data from the file specified by the previously opened file control block myfcb.

```
myfcb   db   0                      ; drive = default
        db   'TEST'                 ; filename, 8
                                    ; characters
        db   'DAT'                  ; extension, 3
                                    ; characters
        db   25 dup (0)             ; remainder of FCB
          .
          .
        mov  ah,14h                 ; function number
        mov  dx,seg myfcb           ; address of FCB
        mov  ds,dx
        mov  dx,offset myfcb        ; set record size
        mov  word ptr myfcb+0eh,512
        int  21h                    ; transfer to MS-DOS
        or   al,al                  ; check status
        jnz  error                  ; jump if read failed
               .
               .
```

Int 21H Sequential write Function 15H (21)

Writes the next sequential block of data from a file, then increments the file pointer appropriately.

Hidden page

and FCB filled in by MS-DOS as follows :

drive field (offset 00H) = 1 for drive A, 2 for drive B, etc.
current block field (offset 0CH) = 00H
record size field (offset 0EH) = 0080H
[2.0+]size field (offset 10H) = file size from directory
[2.0+]date field (offset 14H) = date stamp from directory
[2.0+] time field (offset 16H) = time stamp from directory

If function unsuccessful (directory full)

```
AL = FFH
```

Example : Create a file in the current directory using the name in the file control block myfcb.

```
myfcb       db   0              ; drive = default
            db   'TEST'         ; filename, 8 characters
            db   'DAT'          ; extension, 3 characters
            db   25 dup (0)     ; remainder of FCB
              .
              .
            mov  ah,16h             ; function number
            mov  dx,seg myfcb       ; address of FCB
            mov  ds,dx
            mov  dx,offset myfcb
            int  21h                ; transfer to MS-DOS
            or   al, al             ; check status
            jnz  error              ; jump if create failed
              .
              .
```

Int 21H — Rename file — Function 17H (23)

Alters the name of all matching files in the current directory on the disk in the specified drive.

Calling parameters

```
AH = 17H
DS:DX = segment:offset of "special" file control
        block
```

Returns : If function successful (one or more files are renamed)

```
AL = 00H
```

If function unsuccessful (no matching files, or new filename matched an existing file)

```
AL = FFH
```

Example : Rename the file .DAT to NEWNAME.DAT.

```
myfcb   db   0              ; drive = default
        db   'OLDNAME'      ; old file name, 8 characters
        db   'DAT'          ; old extension, 3 characters
        db   6 dup (0)      ; reserved area
        db   'NEWNAME'      ; new file name, 8 characters
        db   'DAT'          ; new extension, 3 characters
        db   14 dup (0)     ; reserved area
          .
          .
        mov  ah,17h             ; function number
        mov  dx,seg myfcb       ; address of FCB
        mov  ds,dx
        mov  dx,offset myfcb
        int  21h                ; transfer to MS-DOS
        or  al,al               ; check status
        jnz  error              ; jump if close failed
          .
          .
```

Int 21H — Get file size — Function 23H (35)

Searches for a matching file in the current directory; if one is found, updates the FCB with the file's size in terms of number of records.

Calling Parameters :

```
AH = 23H
DS:DX = segment: offset of unopened file control
        block
```

Returns : If function successful (matching file found)

```
AL = 00H
```

and FCB relative-record field (offset 21H) set to the number of records in the file.

If function unsuccessful (no matching file found)

```
AL = FFH
```

Example : Determine the size in bytes of the file MICRO.DAT

```
myfcb   db    0              ; drive : default
        db    'MICRO'        ; filename, 8 chars
        db    'DAT'          ; extension, 3 chars
        db    25 dup (0)     ; remainder of FCB
```

Hidden page

```
If function failed
Carry flag = set
        AX = error code
```

Example : Create and open,or truncate to zero length and open, the file C:\MBS\PRO1.ASM and save the handle for subsequent access to the file.

```
fname       db      'C:\MBS\PRO1.ASM',0
fhandle     dw      ?
    .
    .
    mov  ah,3ch             ;   function number
    xor  cx,cx              ;   normal attribute
    mov  dx,seg fname       ;   address of path name
    mov  ds,dx
    mov  dx,offset fname
    int  21h                ;   transfer to MS-DOS
    jc   error              ;   jump if create failed
    mov  fhandle,ax         ;   save file handle
    .
    .
```

Int 21H — Open file — Function 3DH (61)

Opens the specified file in the designated or default directory on the designated or default disk drive. A handle is returned which can be used by the program for subsequent access to the file.

Calling Parameters

```
AH = 3DH
AL = access mode
  Bit(s)   Significance
  0-2           access mode
                000 = read access
                001 = write access
                010 = read/write access
  3         reserved (0)
  4-6       sharing mode (MS-DOS versions 3.0
            and later)
                000 = compatibility mode
                001 = deny all
                010 = deny write
```

```
                    011 = deny read
                    100 = deny none
          7       inheritance flag (MS-DOS versions 3.0
                  & later)
                    0 = child process inherits handle
                    1 = child does not inherit handle
    DS:DX = segment:offset of ASCII path name
```

Returns : If function successful

```
Carry flag = clear
     AX = handle
```

If function unsuccessful

```
Carry flag = set
     AX = error code
```

Example : Open the file C:\\PRO1.ASM for both reading and writing, and save the handle for subsequent access to the file.

```
fname       db   'C:\MBS\PRO1.ASM',0
fhandle     dw   ?

            .
            .
            mov  ah,3dh          ;   function number
            mov  al,02h          ;   mode - read/write
            mov  dx,seg  fname   ;   address of path name
            mov  ds,dx
            mov  dx,offset  fname
            int  21H             ;   transfer to MS-DOS
            jc   error           ;   jump if open failed
            mov  fhandle,ax      ;   save file handle
            .
            .
```

Int 21H — Close file — Function 3EH (62)

Given a handle that was obtained by a previous successful open or create operation, flushes all internal buffers associated with the file to disk, closes the file, and releases the handle for reuse. If the file was modified, the time and date stamp and file size are updated in the file's directory entry.

Calling Parameters

```
AH = 3EH
BX = handle
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
AX = error code
```

Example : Close the file whose handle is saved in the variable fhandle.

```
fhandle  dw   0
           .
           .
           mov  ah,3eh        ;   function number
           mov  bx,fhandle    ;   file handle
           int  21h           ;   transfer to MS-DOS
           jc   error         ;   jump if close failed
           .
           .
```

Int 21H — Read file or device — Function 3FH (63)

Given a valid file handle from a previous open or create operation, a buffer address, and a length in bytes, transfers data at the current file-pointer position from the file into the buffer and then updates the file pointer position.

Calling Parameters

```
AH = 3FH
BX = handle
CX = number of bytes to read
DS:DX = segment:offset of buffer
```

Returns : If function successful

```
Carry flag = clear
AX = bytes transferred
```

If function unsuccessful

```
Carry flag = set
AX = error code
```

Example : Using the file handle from a previous open or create operation, read 512 bytes at the current file pointer into the buffer named buff.

```
buff    db   512 dup (?) ; buffer for read
fhandle dw   ?           ; contains file handle
        .
        .
        .
        mov ah,3fh           ; function number
        mov dx, seg buff     ; buffer address
        mov ds, dx
        mov dx, offset buff
        mov bx, fhandle  ; file handle
        mov cx, 512          ; length to read
        int 21h              ; transfer to MS-DOS
        jc  error            ; jump, read failed
        cmp ax, cx           ; check length of read
        jl  done             ; jump, end of file
        .
        .
```

Int 21H — Write file or device — Function 40H (64)

Given a valid file handle from a previous open or create operation, a buffer address, and a length in bytes, transfers data from the buffer into the file and then updates the file pointer position.

Calling parameters

```
AH = 40H
BX = handle
CX = number of bytes to write
DS:DX = segment:offset of buffer
```

Returns : If function successful

```
Carry flag = clear
AX = bytes transferred
```

If function unsuccessful

```
Carry flag = set
AX = error code
```

Example : Using the handle from a previous open or create operation, write 512 bytes to disk at the current file pointer from the buffer named buff.

```
buff    db  512 dup (?)        ; buffer for write
fhandle dw  ?                  ; contains file handle
        .
        .
        mov ah,40h             ; function number
        mov dx, seg buff       ; buffer address
        mov ds, dx
        mov dx, offset buff
        mov bx, fhandle  ; file handle
        mov cx, 512            ; length to write
        int 21h                ; transfer to MS-DOS
        jc  error              ; jump, write failed
        cmp ax, 512            ; entire record written?
        jne error              ; no, jump
        .
```

Int 21H — Delete file — Function 41H (65)

Deletes a file from the specified or default disk and directory.

Calling Parameters

```
AH = 41H
DS:DX = segment:offset of ASCIIZ pathname
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
AX = error code
```

Example : Delete the file named MICRO.DAT from the directory \MYDIR on drive C.

```
fname   db  'C:\MYDIR\MICRO.DAT',0
        .
        .
mov ah,41h             ; function number
mov dx,seg fname       ; filename address
mov ds,dx
mov dx,offset fname
int 21h                ; transfer to MS-DOS
jc  error              ; jump if delete failed
.
.
.
```

INT 21H — Move file pointer — Function 42H (66)

DOS maintains a file pointer. The open file operation initialize file pointer to 0 and subsequent sequential reads and writes increment file pointer by record.

Calling parameters

```
AH = 42H
AL = method code
     00H absolute offset from start of file
     01H signed offset from current file pointer
     02H signed offset from end of file
BX = handle
CX = most significant half of offset
DX = least signficant half of offset
```

Returns : If function successful

```
Carry flag = clear
     DX = most significant half of resulting file
          pointer
     AX = least significant half of resulting file
          pointer
```

If function unsuccessful

```
Carry flag = set
     AX = error code
```

Int 21H — Rename file — Function 56H (86)

Renames a file and/or moves its directory entry to a different directory on the same disk. In MS-DOS version 3.0 and later, this function can also be used to rename directories.

Calling parameters

```
     AH = 56H
    DS: = segment:offset of current ASCIIZ pathname
  ES:DI = segment:offset of new   pathname
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
     AX = error code
```

Example : Change the name of the file MYFILE.DAT in the directory \MYDIR on drive C to MYTFXT.DAT. At the same time, move the file to the directory \SYSTEM on the same drive.

```
oldname  db   'C:\MYDIR\MYFILE.DAT',0   ; drive = default
newname  db   'C:\SYSTEM\MYTEXT.DAT',0
         .
         .
         mov  ah,  56h               ; function number
         mov  dx,  seg oldname  ; old filename address
         mov  ds,  dx
         mov  dx,  offset oldname
         mov  di,  seg newname       ; new filename address
         mov  es,  di
         mov  di,  offset newname
              int  21h               ; transfer to MS-DOS
              jc error               ; jump if rename
                                     ; failed
                .
                .
```

9.5 Memory Management Functions

Int 21H	Allocate memory block	Function 48H (72)

Allocates a block of memory and returns a pointer to the beginning of the allocated area.

Calling parameters :

```
AH = 48H
BX =  number of paragraphs of memory needed
```

Returns : If function successful

```
Carry flag = clear
        AX = base segment address of allocated block
```

If function unsuccessful

```
Carry flag = set
        AX = error code
           = size of largest available block
             (paragraphs)
```

Example : Request a 64 KB block of memory for use as a buffer.

```
bufseg   dw   ?              ; segment base of new block
```

```
        .
        .
        mov  ah,48h         ; function number
        mov  bx,1000h       ; block size (paragraphs)
        int  21h            ; transfer to MS-DOS
        jc   error          ; jump if allocation failed
        mov  bufseg,ax      ; save segment of new block
```

Int 21H — Release memory block — Function 49H (73)

Releases a memory block and makes it available for use by other programs.

Calling parameters :

```
AH = 49H
ES = segment of block to be released
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
AX = error code
```

Example : Release the memory block that was previously allocated in the example for 21H Function 48H.

```
bufseg  dw   ?              ; segment base of block
        .
        .
        mov  ah,49h         ; function number
        mov  es,bufseg      ; base segment of block
        int  21h            ; transfer to MS-DOS
        jc   error          ; jump if release failed
        .
        .
```

Int 21H — Resize memory block — Function 4AH (74)

Dynamically shrinks or extends a memory block, according to the needs of an application program.

Calling parameters :

```
AH = 4AH
   = desired new block size in paragraphs
ES = segment of block to be modified
```

Hidden page

Int 15H **Get extended memory size** **Function 88H(136)**

Returns the amount of extended memory installed in the system

Calling parameters :

```
AH = 88H
```

Returns :

```
AX = amount of extended memory (in KB)
```

9.6 Display Functions Provided by ROM BIOS

Int 10H **Set video mode** **Function 00H**

Selects the current video display mode. Also selects the active video controller, if more than one video controller is present.

Calling Parameters

```
AH = 00H
AL = video modes
```

Returns Nothing

Different Video Modes

Mode	Resolution	Colors	Text/graphics
00H	40-by-25 color burst off	16	text
01H	40-by-25	16	text
02H	80-by-25 color burst off	16	text
03H	80-by-25	16	text
04H	320-by-200	4	graphics
05H	320-by-200 color burst off	4	graphics
06H	640-by-200	2	graphics
07H	80-by-25	2^1	text
08H	160-by-200	16	graphics

09H	320-by-200	16	graphics
0AH	640-by-200	4	graphics
0BH	reserved		
0CH	reserved		
0DH	320-by-200	16	graphics
0EH	640-by-200	16	graphics
0FH	640-by-350	2^2	graphics
10H	640-by-350	4	graphics
10H	640-by-350	16	graphics
11H	640-by-480	2	graphics
12H	640-by480	16	graphics
13H	320-by-200	256	graphics

Int 10H — Set cursor type — Function 01H

Selects the starting and ending lines for the blinking hardware cursor in text display modes.

Calling Parameters :

```
        AH = 01H
CH bits 0-4 = starting line for cursor
CL bits 0-4 = ending line for cursor
```

Note : Cursor can be disabled by setting CH = 20H

Returns : Nothing

Int 10H — Set cursor position — Function 02H

Positions the cursor on the display, using text coordinates.

Calling Parameters :

```
AH = 02H
BH = page
DH = row (y coordinate)
DL = column (x coordinate)
```

Returns : Nothing

Int 10H **Get cursor position** **Function 03H**

Obtains the current position of the cursor on the display, in text coordinates.

Calling Parameters :

```
AH = 03H
BH = page
```

Returns :

```
CH = starting line for cursor
CL = ending line for cursor
DH = row (y coordinate)
DL = column (x coordinate)
```

Int 10H **Read character and attribute at cursor** **Function 08H**

Writes an ASCII character and its attribute to the display at the current cursor position.

Calling Parameters :

```
AH = 08h
AL = character
BH = page
BL = attribute (text modes) or color
     (graphics modes)
CX = count of characters to write
     (replication factor)
```

Returns : Nothing

Int 10H **Write character at cursor** **Function 0AH (10)**

Writes an ASCII character to the display at the current cursor position. The character receives the attribute of the previous character displayed at the same position.

Calling Parameters :

```
AH = 0AH
AL = character
BH = page
BL = color
CX = count of characters to write
     (replication factor)
```

Returns : Nothing

Hidden page

```
mov ah,5          ; function number
mov dl,'*'        ; character to output
int 21h           ; transfer to MS-DOS
```

Int 17H — Write character to printer — Function 00H

Sends a character to the specified parallel printer interface port and returns the current status of the port.

Calling parameters :

```
AH = 00H
AL = character
DX = printer number (0 = LPT1, 1 = LPT2,
     2 = LPT3)
```

Returns :

```
AH = status
```

Bit	Significance (if set)
0	printer timed-out
1	unused
2	unused
3	I/O error
4	printer selected
5	out of paper
6	printer acknowledge
7	printer not busy

Int 17H — Initialize printer port — Function 01H

Initializes the specified parallel printer interface port and returns its status.

Calling parameters :

```
AH = 01H
DX = printer number (0 = LPT1, 1 = LPT2,
     2 = LPT3)
```

Returns :

```
AH = status (see Int 17H Function 00H)
```

Int 17H **Get printer status** **Function 02H**

Returns the current status of the specified parallel printer interface port.

Calling parameters :

```
AH = 02H
DX = printer number (0 = LPT1, 1 = LPT2,
     2 = LPT3)
```

Returns :

```
AH = status (see Int 17H Function 00H)
```

❑❑❑

10 Serial Communication

Most of the microprocessors are designed for parallel communication. In parallel communication number of lines required to transfer data depend on the number of bits to be transferred. For example, to transfer a byte of data, 8 lines are required and all 8 bits are transferred simultaneously. Thus for transmitting data over a long distance, using parallel communication is impractical due to the increase in cost of cabling. Parallel communication is also not practical for devices such as cassette tapes or a CRT terminal. In such situations, serial communication is used. In serial communication one bit is transferred at a time over a single line.

To implement serial communication in the microcomputer system, it is necessary to understand the basic concepts of serial communication. The following section describes the basic concepts involved in serial communication.

Basic concepts :

1. Classification
2. Transmission formats
3. Data communication over telephone lines
4. Error detection
5. Interfacing requirements
6. Serial communication standards.

10.1 Classification

Serial data transmission can be classified on the basis of how transmission occurs.

1. Simplex
2. Half duplex
3. Full duplex

10.1.1 Simplex

In simplex, the hardware exists such that data transfer takes place only in one direction. There is no possibility of data transfer in the other direction. A typical example is transmission from a computer to the printer.

10.1.2 Half Duplex

The half duplex transmission allows the data transfer in both directions, but no simultaneously. A typical example is a walkie-talkie.

10.1.3 Full Duplex

The full duplex transmission allows the data transfer in both direction simultaneously The typical example is transmission through telephone lines.

10.2 Transmission Formats

The data in the serial communication may be sent in two formats :

a) Asynchronous b) Synchronous

10.2.1 Asynchronous

Fig. 10.1 shows the transmission format for asynchronous transmission. Asynchronous formats are character oriented. In this, the bits of a character or data word are sent at a constant rate, but characters can come at any rate (asynchronously) as long as they do no overlap. When no characters are being sent, a line stays high at logic 1 called **mark**, logic 0 is called **space**. The beginning of a character is indicated by a start bit which is always low. This is used to synchronize the transmitter and receiver. After the start bit, the data bits are sent with least significant bit first, followed by one or more stop bits (active high). The stop bits indicate the end of character. Different systems use 1, 1 1/2 or 2 stop bits. The combination of start bit, character and stop bits is known as **frame.** The start and stop bits carry no information, but are required because of the asynchronous nature of data. Fig. 10.2 illustrates how the data byte CAH would look when transmitted in the asynchronous serial format.

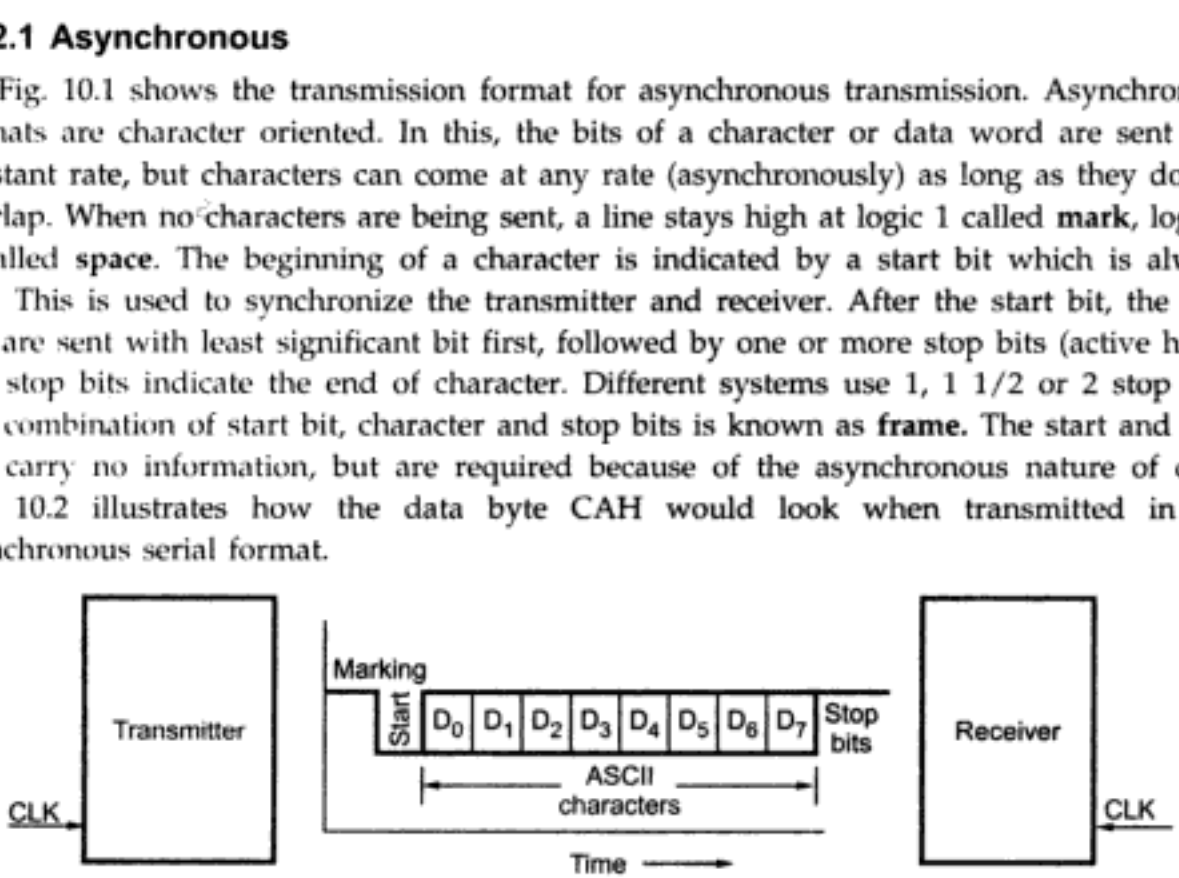


Fig. 10.1 Transmission format for asynchronous transmission

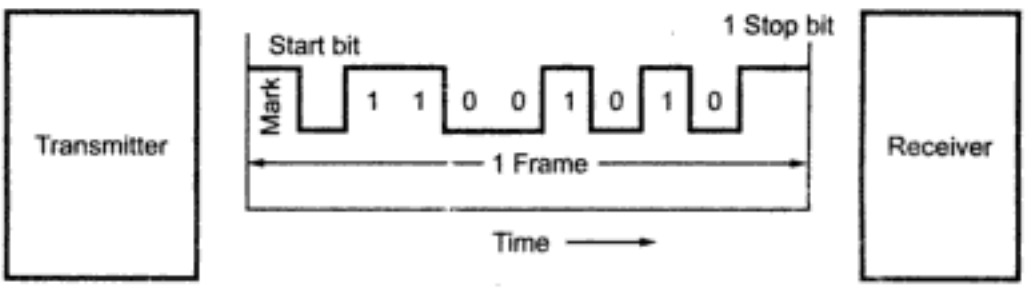


Fig. 10.2 Asynchronous format with data byte CAH

The data rate can be expressed as bits/sec. or characters/sec. The term bits/sec is also called the **baud rate**. The asynchronous format is generally used in low-speed transmission (less than 20 Kbits/sec).

10.2.2 Synchronous

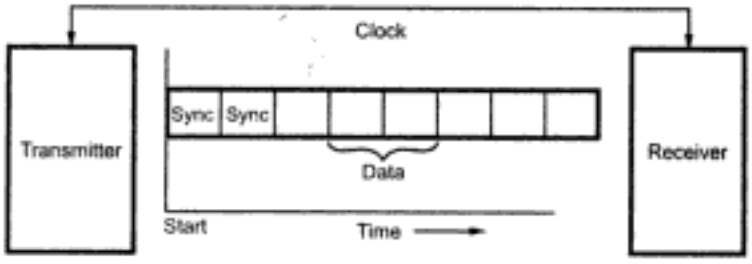


Fig. 10.3 Synchronous transmission format

The start and stop bits in each frame of asynchronous format represents wasted overhead bytes that reduce the overall character rate. These start and stop bits can be eliminated by synchronizing receiver and transmitter. They can be synchronized by having a common clock signal. Such a communication is called **synchronous serial communication**. The Fig. 10.3 shows the transmission format of synchronous serial communication. In this transmission synchronous bits are inserted instead of start and stop bits.

Sr. No.	Asynchronous Serial Communication	Synchronous Serial Communication
1.	Transmitters and receivers are not synchronized by clock.	Transmitter and receivers are synchronized by clock.
2.	Bits of data are transmitted at constant rate.	Data bits are transmitted with synchronisation of clock.
3.	Character may arrive at any rate at receiver.	Character is received at constant rate.
4.	Data transfer is character oriented.	Data transfer takes place in blocks.
5.	Start and stop bits are required to establish communication of each character.	Start and stop bits are not required to establish communication of each character; however, synchronisation bits are required to transfer the data block.
6.	Used in low-speed transmissions at about speed less than 20 kbits/sec.	Used in high-speed transmissions

Table 10.1 Comparison between asynchronous and synchronous serial data transfer

10.3 Interfacing Requirements

To implement serial communication in microprocessor system we need basically two devices :

1) Parallel to serial converter 2) Serial to parallel converter.

To transmit byte data it is necessary to convert byte into eight serial bits. This can be done by using the parallel to serial converter. Similarly at the reception these serial bits must be converted into parallel 8 bit data. The serial to parallel converter is used to convert serial data bits into the parallel data.

The devices are designed for his purpose are called universal asynchronous receiver-transmitter (UART). The devices which provides synchronous as well as asynchronous transmission and reception are called as universal synchronous asynchronous receiver-transmitter. A good example of UART is 8250 and USART is 8251. These devices are software programmable for number of data bits, parity and number of stop bits. In the next sections, we discuss the 8251 (USART).

10.4 USART 8251

To implement serial communication in microprocessor system we need basically two devices : 1) Parallel to serial converter

2) Serial to parallel converter.

To transmit byte data it is necessary to convert byte into eight serial bits. This can be done by using the parallel to serial converter. Similarly at the reception these serial bits must be converted into parallel 8 bit data. The serial to parallel converter is used to convert serial data bits into the parallel data.

The devices are designed for this purpose are called universal asynchronous receiver-transmitter (UART). The devices which provides synchronous as well as asynchronous transmission and reception are called universal synchronous asynchronous receiver-transmitter. A good example of UART is 8250 and USART is 8251. These devices are software programmable for number of data bits, parity and number of stop bits. In the next section we discuss IC 8251 (USART).

10.4.1 Features

1. The Intel 8251A is an universal synchronous and asynchronous communication controller.
2. It supports standard asynchronous protocol with :
   a) 5 to 8 Bit character format
   b) odd, even or no parity generation and detection
   c) Baud rate from DC to 19.2 Kbaud
   d) False start bit detection
   e) Automatic break detect and handling
   f) Break character generation.
3. It has built in baud rate generator.
4. It supports standard synchronous protocol with :
   a) 5 to 8 Bit character format
   b) Internal or external character synchronization
   c) Automatic sync insertion
   d) Baud rate from DC to 64 Kbaud

5. It allows full duplex transmission and reception.
6. It provides double buffering of data both in the transmission section and in the receiver section.
7. It provides error detection logic, which detects parity, overrun and framing errors.
8. It has Modem Control Logic, which supports basic data set control signals.
9. It provides separate clock inputs for receiver and transmitter sections, thus providing an option of fixing different baud rates for the transmitter and receiver section.
10. It is compatible with an extended range of Intel microprocessors.
11. It is fabricated in 28 pin DIP package and its all inputs and outputs are TTL compatible.
12. It is available in standard as well as extended temperature range.

10.4.2 Pin Diagram of 8251A

Fig. 10.4 shows the pin diagram of 8251A.

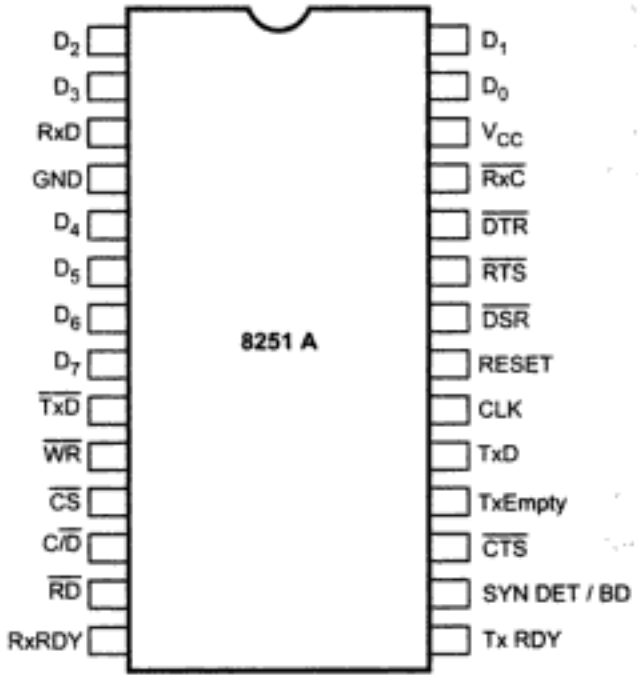


Fig. 10.4 Pin diagram of 8251A

Data Bus : Bi-directional, tri-state, 8-bit Data Bus. This pin allow transfer of bytes between the CPU and the 8251A.

$\overline{\text{RD}}$ (Read) : A low on this input allows the CPU to read data or status bytes from 8251A

$\overline{\text{WR}}$ (Write) : A low on this input allows the CPU to write data or command word to the 8251A.

Hidden page

TxC (Transmitter Clock) : This clock input controls the rate at which the character is to be transmitted.

Receiver Signals

RxD (Receiver Data) : This input receives a composite serial stream of data on the rising edge of $\overline{RxC}$.

RxRDY (Receiver Ready) : This output indicates that the 8251A contains a character that is ready to be input to the CPU.

$\overline{RxC}$ (Receiver Clock) : This clock input controls the rate at which the character is to be received.

SYNDET (Sync Detect)/ BRKDET (Break Detect)

This pin is used in synchronous mode for detection of synchronous characters and may be used as either input or output.

In asynchronous mode this pin goes high if receiver line stays low for more than 2 character times. It then indicates a break in the data stream.

When used as an input (external sync detect mode) a positive signal will cause the 8251A to start receiving data characters on the rising edge of the next $\overline{RXC}$.

10.4.3 Block Diagram

Fig. 10.5 shows the block diagram of IC 8251A. It includes : Data bus buffer, Read/Write control logic, modem control, Transmit buffer, Transmit Control, Receiver Buffer and Receiver control.

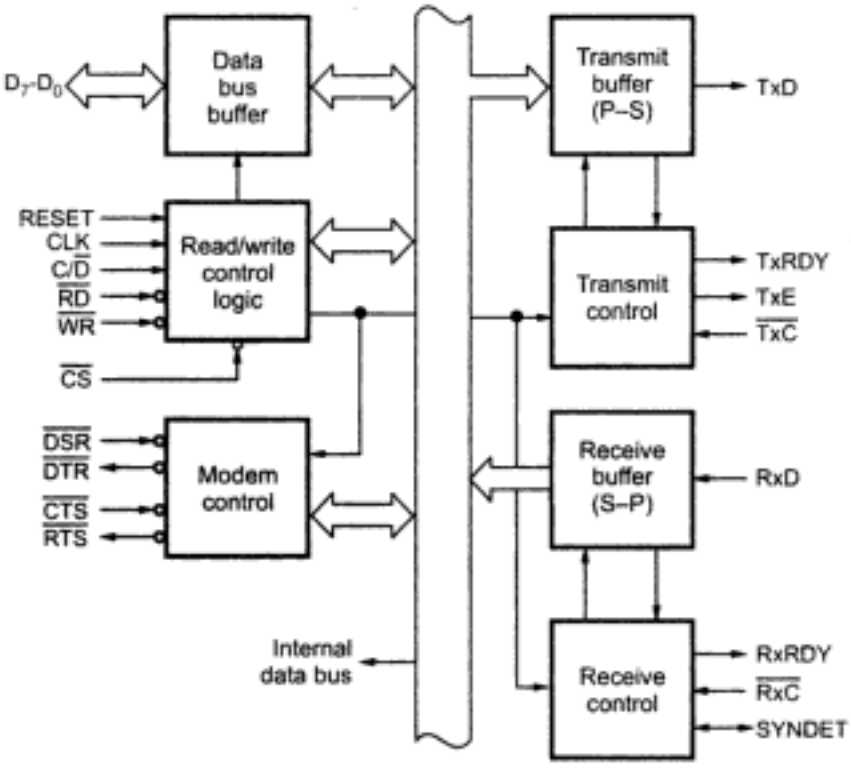


Fig. 10.5 Block diagram


image
not
available

Hidden page

1. Mode instruction 2. Command instruction

Mode Instruction : Fig. 10.6 shows the mode instruction format.

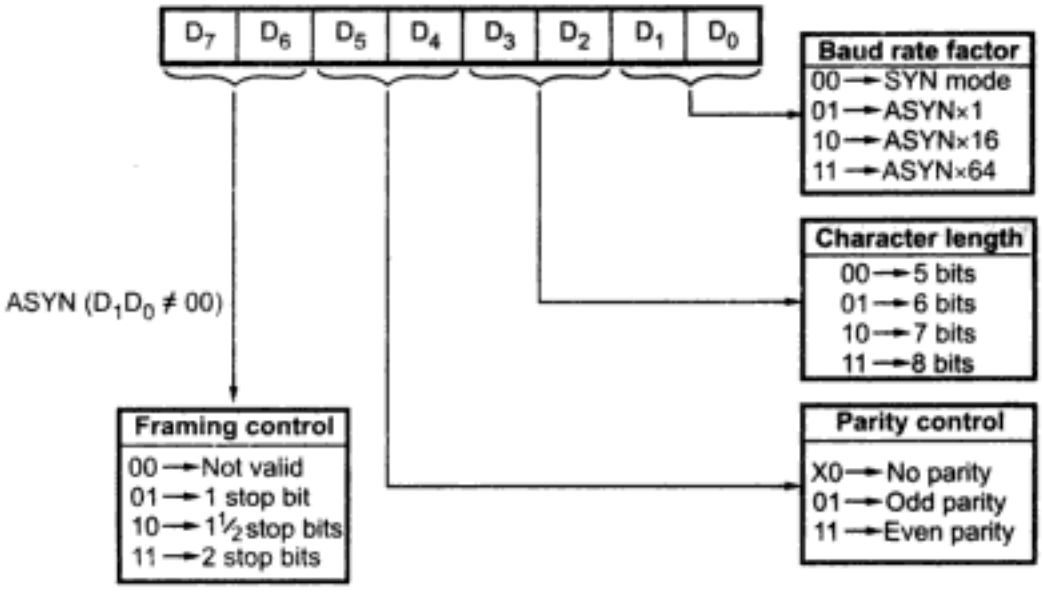


Fig. 10.6 Mode instruction format

The instruction can be considered as four 2-bit fields. The first 2-bit field (D_1-D_0) determines whether the USART is to operate in the synchronous (00) or asynchronous mode. In the asychronous mode, this field determines the division factor for clock to decide the baud rate. For example, if D_1 and D_0 are both ones, the $\overline{RxC}$ and $\overline{TxC}$ will be divided by 64 to establish the baud rate.

The second 2-bit field (D_3-D_2) determines number of data bits in one character. With this 2-bit field we can set character length from 5-bits to 8 bits.

The third 2-bit field, (D_5-D_4), controls the parity generation. The parity bit is added to the data bits only if parity is enabled.

The last field, (D_7-D_6), has two meanings depending on whether operation is to be in the synchronous or asynchronous mode. For asynchronous mode, (i.e. D_1 D_0 ≠ 00), it controls the number of STOP bits to be transmitted with the character. In synchronous mode, (i.e. D_1D_0) = 00) this field controls the synchronizing process. It decides whether to operate with external synchronization or internal synchronization and whether to transmit single synchronizing character or two synchronizing characters.

Command Instruction

After the mode instruction, command character should be issued to the USART. It controls the operation of the USART within the basic frame work established by the mode instruction. Fig. 10.7 shows command instruction format.

It does function such as : Enable Transmit/Receive, Error Reset and modem control.

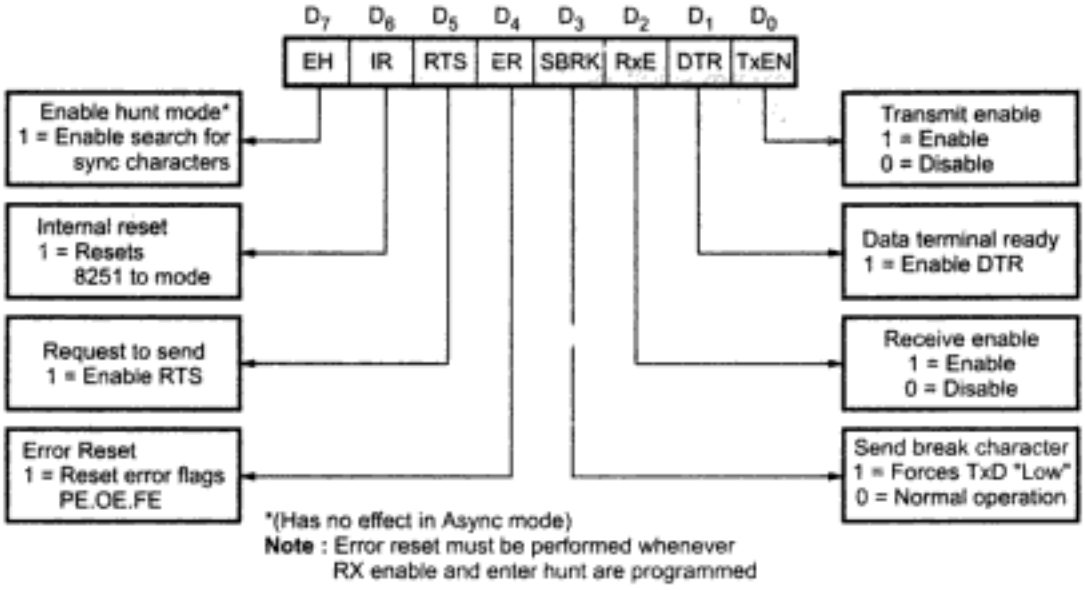


Fig. 10.7 Command instruction format

10.4.5 8251A Status Word

In the data communication systems it is often necessary to examine the "status" of the transmitter and receiver. It is also necessary for CPU to know if any error has occured during communication. The 8251A allow the programmer to read above mentioned information from the status register any time during the functional operation. Fig. 10.8 shows the format of status register.

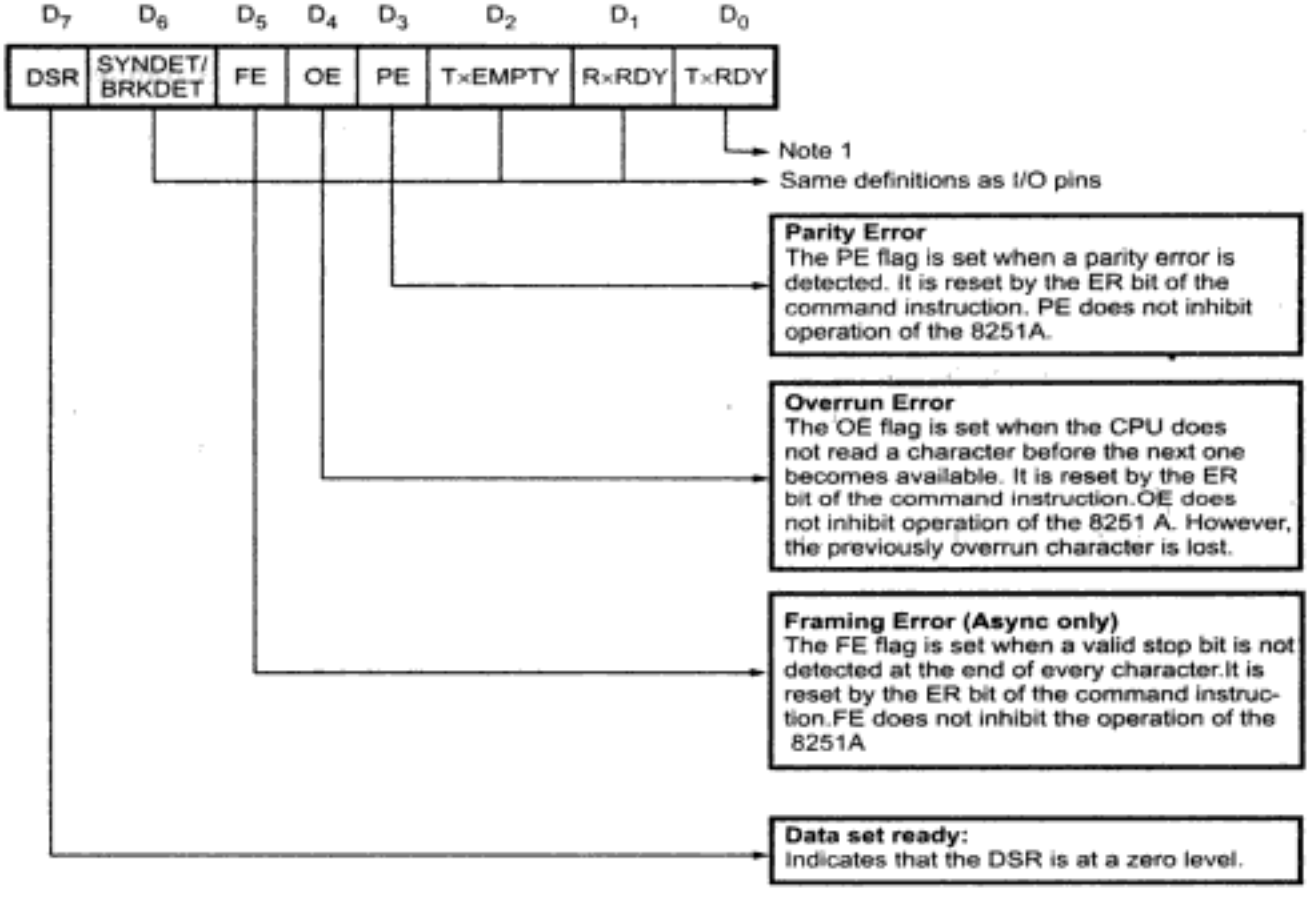


Fig. 10.8 Status register format

Error Definitions

Parity Error : At the time of transmission of data an even or odd parity bit is inserted in the data stream. At the receiver end, if parity of the character does not match with the pre-defined parity parity error occurs.

Overrun Error : In the receiver section received character is stored in the receiver buffer. The CPU is supposed to read this character before reception of the next character. But if CPU fails in reading the character loaded in the receiver buffer, the next the received character replaces the previous one and the OVERRRUN Error occurs.

Framing Error : If valid stop bit is not detected at the end each character framming error occurs.

All these errors, when occur, set the corrosponding bits in the status register. These error bits are reset by setting ER bit in the command instruction.

10.4.6 Data Communication Types

We know that, 8251A is Universal Synchronous, Asynchronous, Receiver, and Transmitter. Therefore communication can take place with four different ways.

1. Asynchronous transmission
2. Asynchronous reception
3. Synchronous transmission
4. Synchronous reception

These communication modes can be enabled by writing proper mode and command instructions. The mode instruction defines the baud rate (in case of asynchronous mode), character length, number of stop bit(s) and parity type. After writing proper mode instruction it is necessary to write appropriate command instruction depending on the communication type.

Asynchronous Transmission

Transmission can be enabled by setting transmission enable bit (bit 0) in the command instruction. When transmitter is enabled and $\overline{CTS}$ = 0 the transmitter is ready to transfer data on TxD line.

Operation : When transmitter is ready to transfer data on TxD line, CPU sends data character and it is loaded in the transmit buffer register. The 8251A then automatically adds a start bit (low level) followed by the data bits (least significant bit first), and the programmed number of STOP bit(s) to each character. It also adds parity information prior to STOP bit(s), as defined by the mode instruction. The character is then transmitted as a serial data stream on the TxD output at the falling edge of $\overline{TxC}$. The rate of transmission is equal to 1, $1/16$ or $1/64$ that of the $\overline{TxC}$, as defined by the mode instruction. Fig. 10.9 shows the transmitter output in the asynchronous mode.

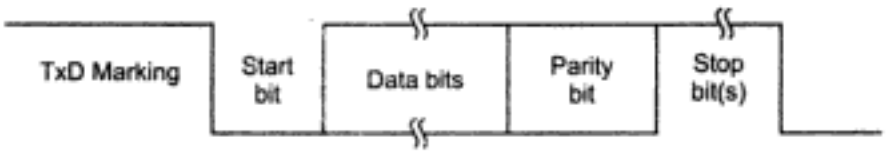


Fig. 10.9 Transmitter output in asynchonous mode

Asynchronous Reception

Reception can be enabled by setting receive enable bit (bit 2) in the command instruction.

Operation :

The RxD line is normally high. 8251A looks for a low level on the RxD line. When it receives the low level, it assumes that it is a START bit and enables an internal counter. At a count equivalent to one-half of a bit time, the RxD line is sampled again. If the line is still low, a valid START bit is detected and the 8251A proceeds to assemble the character. After succesful reception of a START bif the 8251A receives data, parity, and STOP bits and then transfers the data on the receiver input register. The data is then transferred into the receiver buffer register. Fig. 10.10 shows the receiver input in the asynchronous mode.

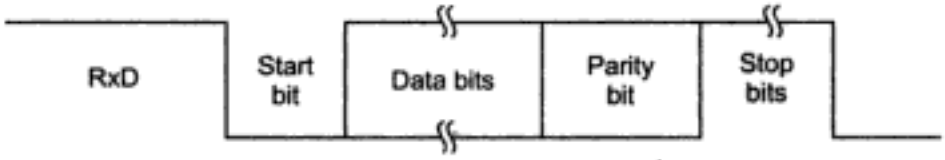


Fig. 10.10 Receiver input in asynchronous mode

Synchronous Transmission

Transmission can be enabled by setting transmission enable bit (bit 0) in the command instruction. When transmitter is enabled and $\overline{CTS}$ = 0, the transmitter is ready to transfer data on TxD line.

Operation : When transmitter is ready to transfer data on TxD line, 8251A transfers characters serially out on the TxD line at the falling edge of the $\overline{TxC}$. The first character usually is the SYNC character.

Once transmission has started, the data stream at the TxD output must continue at the $\overline{TxC}$ rate. If CPU does not provide 8251A with a data character before transmitter buffers become empty, the SYNC characters will be automatically inserted in the TxD data stream, as shown in the Fig. 10.11. In this case, the TxEMPTY pin is raised high to indicate CPU that transmitter buffers are empty. The TxEMPTY pin is internally reset when CPU writes data character in the transmitter buffer.

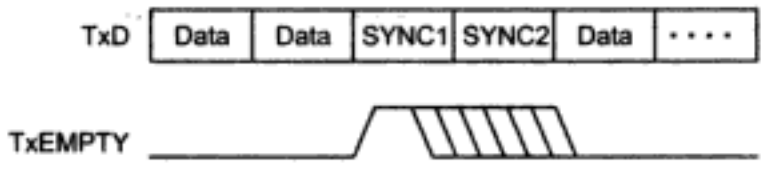


Fig. 10.11 Insertion of SYNC characters

Synchronous Reception : Reception can be enabled by setting receive enable bit (bit 2) in the command instruction.

Operation : In this mode character synchronization can be achieved internally or externally.

Internal SYNC To detect the SYNC character 8251A should be programmed in the 'Enter HUNT' mode by setting bit 7 in the command insturction. Once 8251A enters in the 'Enter HUNT' mode it starts sampling data on the RxD pin on the rising edge of the $\overline{RxC}$. The content of the receiver buffer is compared at every bit boundary with the first SYNC character until a match occurs. If the 8251A has been programmed for two SYNC characters, the subsequent SYNC characters are compared until the match occurs. Once 8251A detects SYNC character(s) it enters from 'HUNT' mode to character synchronization mode, and starts receiving the data characters on the rising edge of the next $\overline{RxC}$. To indicate that the synchronization is achieved 8251A sets the SYNDET pin high. It is reset automatically when CPU reads the status register.

External SYNC

In the external SYNC mode, synchronization is achieved by applying a high level on the SYNDET pin, thus forcing the 8251A out of the HUNT mode.

10.4.7 Interfacing 8251A to 8086 in I/O Mapped I/O Mode

Fig. 10.12 shows the interfacing of 8251A with 8086 in I/O mapped I/O technique. Here, $\overline{RD}$ and $\overline{WR}$ signals are activated when $M/\overline{IO}$ signal is low, indicating I/O bus cycle. Only lower data bus (D_0 - D_7) is used as 8251A is 8-bit device. Reset out signal from clock generator is connected to the reset signal of the 8251A.

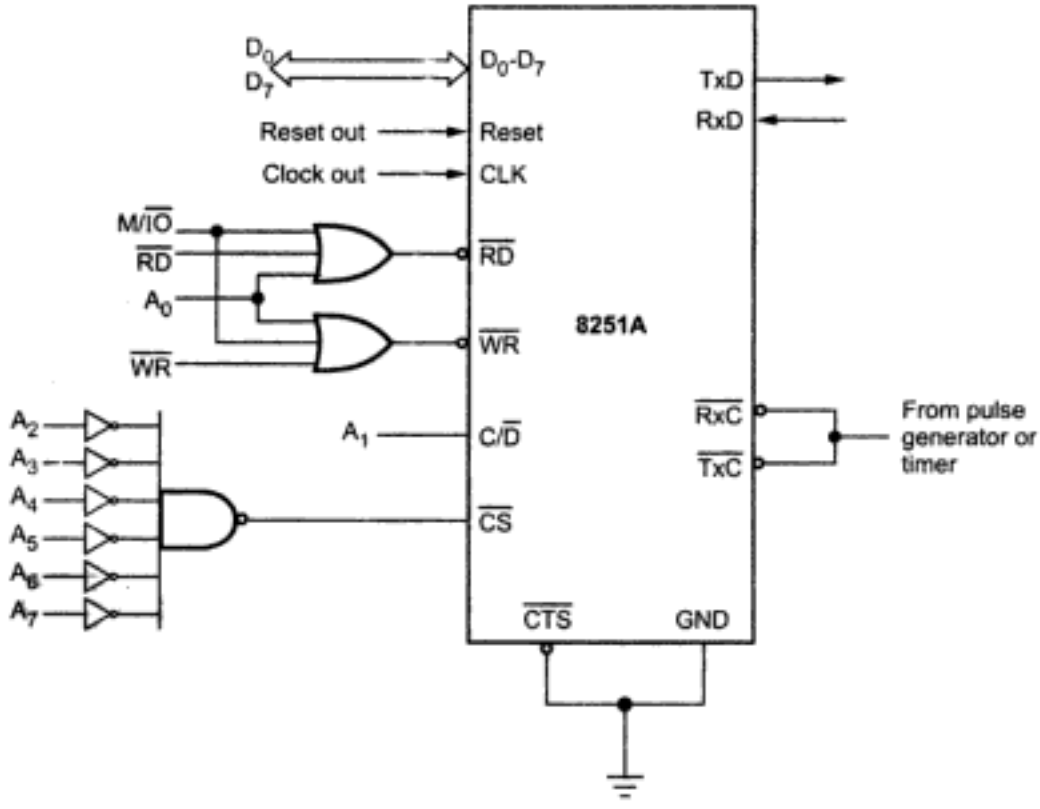


Fig. 10.12 Interfacing of 8251A with 8086 in I/O mapped I/O

I/O Map :

Register	Address lines								Address
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Data Register	0	0	0	0	0	0	0	0	00H
Control Register	0	0	0	0	0	0	1	0	02H

10.4.8 Interfacing 8251A to 8086 in Memory Mapped I/O

In this type of I/O interfacing, the 8086 uses 20 address lines to identify an I/O device; an I/O device is connected as if it is a memory register. The 8086 uses same control signals and instructions to access I/O as those of memory. Fig. 10.13 shows the interfacing of 8251A with 8086 in memory mapped I/O technique. Here, $\overline{RD}$ and $\overline{WR}$ signals are activated when M/$\overline{IO}$ signal is high, indicating memory bus cycle. Address line A_1 is used to select either data register or control register. The remaining address lines A_2-A_{19} are used to decoder the addresses for 8251A.

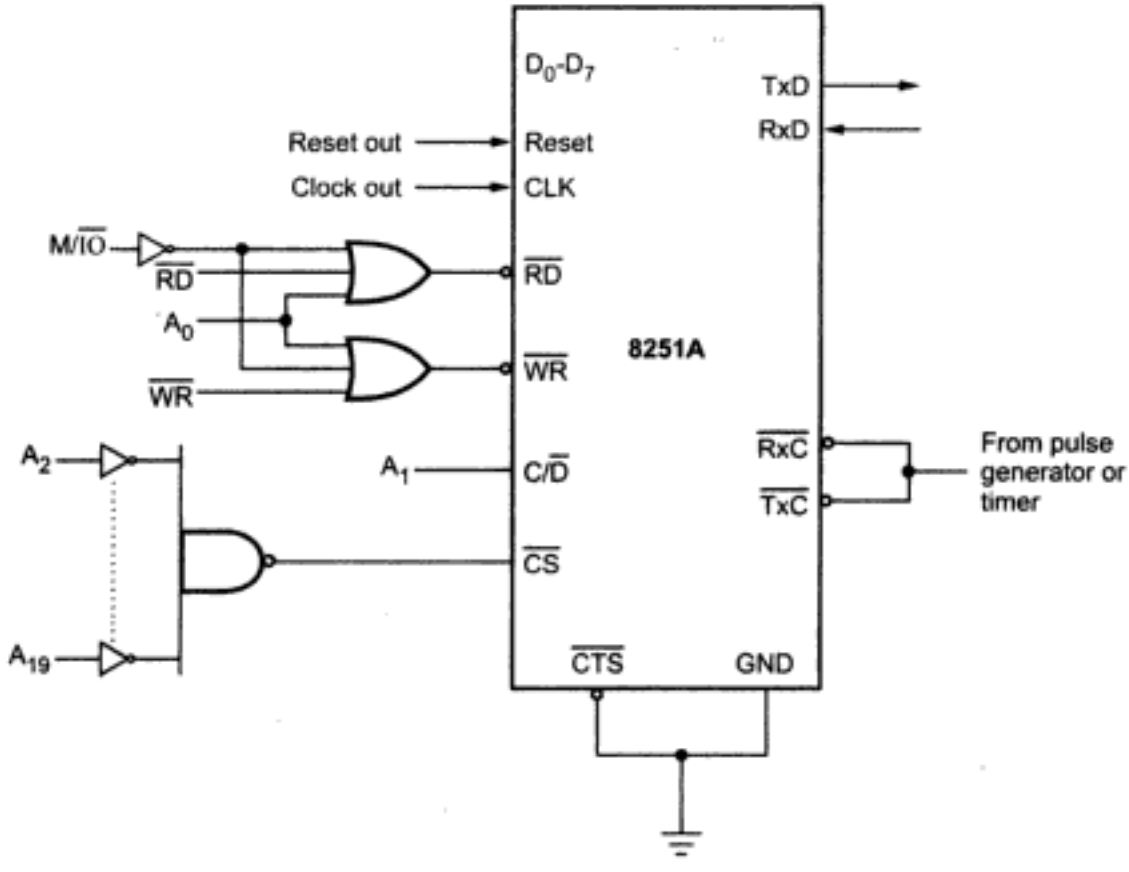


Fig. 10.13 Interfacing of 8251A with 8086 in memory mapped I/O

I/O Map :

Register	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
Data Register	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
Control Register	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	00002H

10.4.9 Programming Examples

To implement serial communication the CPU must inform the 8251A all details such as mode, baud rate (in case of asynchronous mode), stop bits, parity etc. Therefore, prior to data transfer, a set of control words must be loaded into the mode instruction and control instruction registers of 8251A.

Example 1 : Write the sequence of instructions required to initialize 8251A at address 80H and 81H for the configuration given below

i) Character length - 6 bits
ii) Parity even
iii) Stop bit 1
iv) Baud rate 64 X
v) DTR and RTS asserted
vi) Error flag reset
vii) Transmitter enable

Sol. : In the example, number of stop bits and baud rate is specified, therefore, it is necessary to initialize 8251A in the asychronous mode.

Mode word for given specification is as follows.

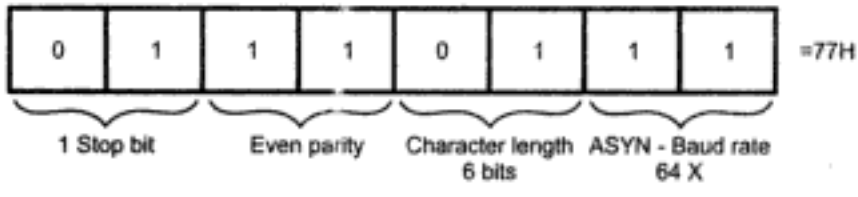


Fig. 10.14

Command word for given specification is as follows.

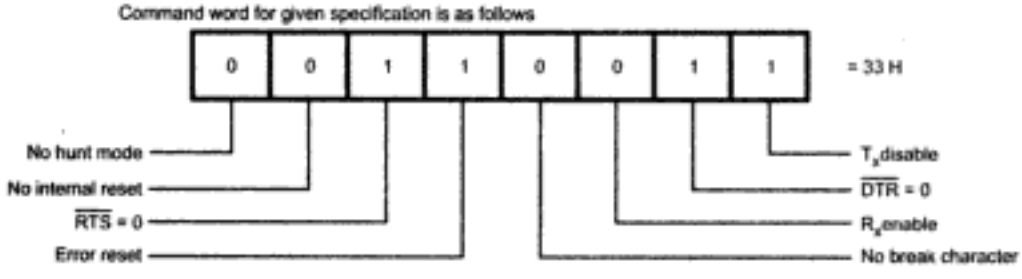


Fig. 10.15

Program :

```
MOV AL,  00H
OUT 81H, AL
OUT 81H, AL
OUT 81H, AL        ; Dummy mode word
MOV AL,  40H       ; Reset command word
OUT 81H, AL        ; Reset  8251A
MVI AL,  77H       ; Mode Word intilization
OUT 81H, AL
MOV AL,  33H       ; Command word initialization
OUT 81H, AL
```

Note : Before initialization of the 8251A, the dummy mode word and the reset command are sent to the control register. Initially control register may have any random word; therefore, it is a good practice to reset the 8251A. However, it expects the instruction as a mode word followed by the command word. Therefore, the reset command is sent after sending three dummy mode words, which are recommended to avoid problems when it is turned on.

10.5 Serial Communication Protocol (RS232C)

In response to the need for signals and handshake standards between DTE and DCE, the Electronic Industries Association (EIA) introduced EIA standard RS-232 in 1962. It was revised and named as RS-232C, in 1969 by EIA. It is widely accepted for single ended data transmission over short distances with low data rates.

This standard describes the functions of 25 signal and handshake pins for serial data transfer. It also describes the voltage levels, impedance levels, rise and fall times, maximum bit rate, and maximum capacitance for these signal lines. RS-232C specifies 25 signal pins and it specifies that the DTE connector should be male, and the DCE connector should be a female. The most commonly used connector should be a female. The most commonly used connector, DB-25P is shown in the Fig. 10.16.

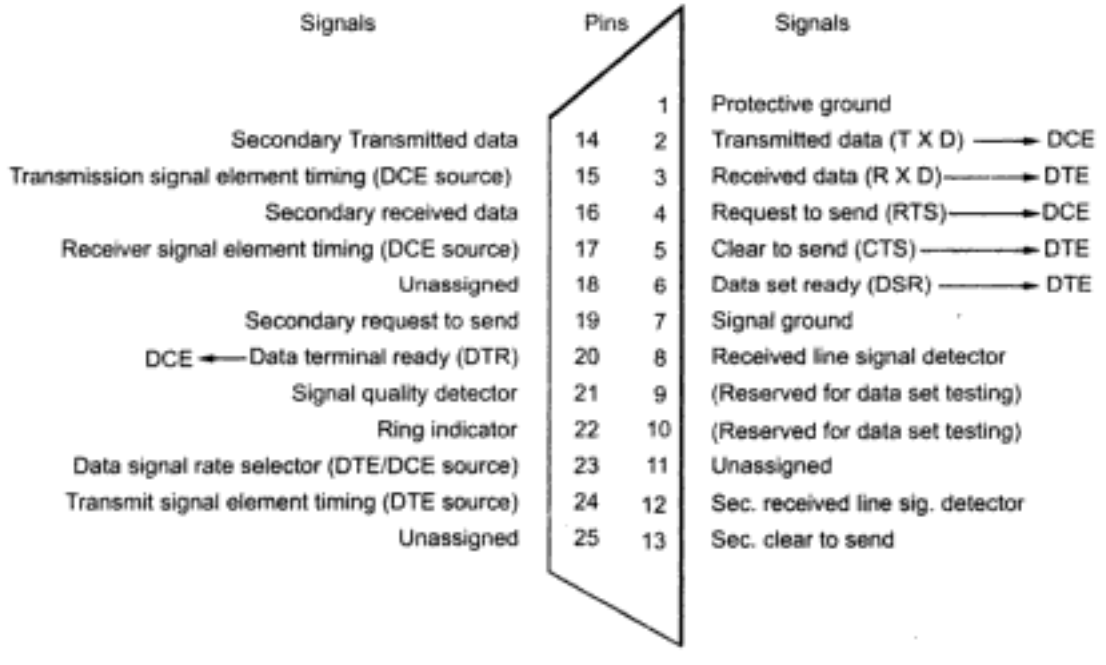


Fig. 10.16 RS 232C 25 pin connector

Pin Number	Common Name	RS-232C Name	Description	Signal Direction On DEC
1		AA	Protective ground	
2	TXD	BA	Transmitted data	IN
3	RXD	BB	Received data	OUT
4	$\overline{\text{RTS}}$	CA	Request to send	IN
5	$\overline{\text{CTS}}$	CB	Clear to send	OUT
6	$\overline{\text{DSR}}$	CC	Data set ready	OUT
7	GND	AB	Signal ground (common return)	
8	$\overline{\text{CD}}$	CF	Received line signal detector	OUT
9			(Reserved for data set testing)	
10			(Reserved for data set testing)	
11			Unsigned	
12		SCF	Secondary recd. line sig. detector	OUT
13		SCB	Secondary clear to send	OUT
14		SBA	Secondary transmitted data	IN
15		DB	Transmission signal element timing (DCE source)	OUT
16		SBB	Secondary received data	OUT
17		DD	Receiver signal element timing (DCE source)	OUT
18			Unassigned	
19		SCA	Secondary request to send	IN
20	$\overline{\text{DTR}}$	CD	Data terminal ready	IN
21		CG	Signal quality detector	OUT
22		CE	Ring indicator	OUT
23		CH/CI	Data signal rate selector (DTE/DCE source)	IN/OUT
24		DA	Transmit signal element timing (DTE source)	IN
25			Unassigned	

Table 10.2

The Table 10.2 shows pins and signals description for RS-232C for data lines, The voltage level + 3V to + 15V is defined as logic 0; from –3 V to –15 V is defined as logic 1. The control and timing signals are compatible with the TTL level. Because of the incompatibility of the data lines with the TTL logic, voltage translators, called line drivers and line receivers, are required to interface TTL logic with the RS-232 signals. Fig. 10.17 shows the interfacing between TTL and RS-232 signals. The line driver, MC1488, converts logic 1 into approximately 9 V. These levels at the receiving end are again converted by the line receiver, MC1489, into TTL-compatible logic.

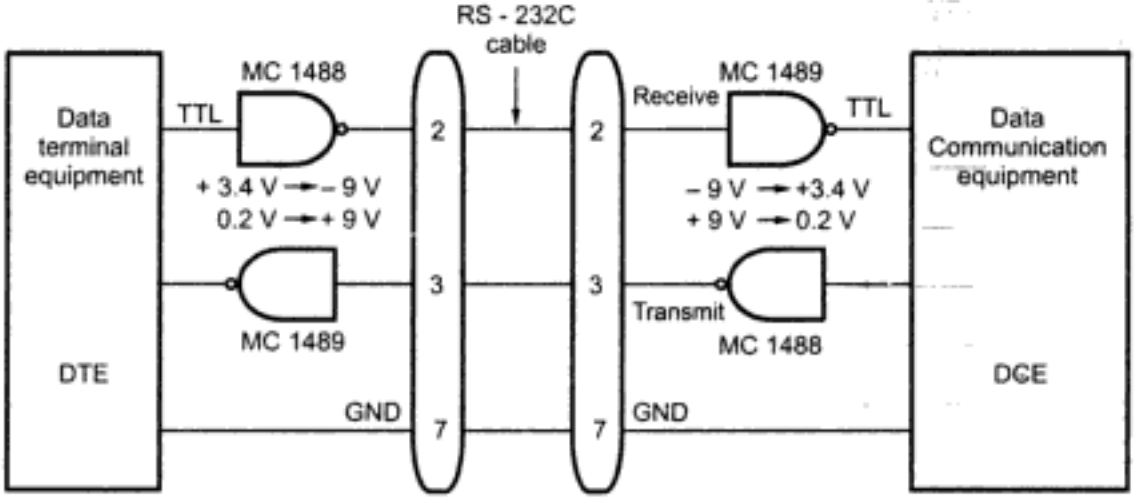


Fig. 10.17 Line drivers and receivers

10.6 Sample Programs of Serial Data Transfer

In this section, we are going to see three programs which transfer data serially through the COM port of one PC to another PC.

10.6.1 Program to Transmit One Character

```
CODE SEGMENT
ASSUME  CS : CODE, DS : CODE
ORG  0100H
START :  MOV  AH,00H    ; Initialize serial port COM2 with
         MOV  AL,03H    ; 8-data bits, 1 stop bit, parity
         MOV  DX,01H    ; none and baud rate 110 bps.
         INT  14H
         MOV  AH,08H    ; Read character from keyboard
         INT  21H

         MOV  AH,01H    ; Transmit character in AL to
         MOV  DX,01H    ; COM2 serial port
         INT  14H

         MOV  AH,4CH    ; Terminate program and
         INT  21H       ; return to DOS
CODE ENDS
END  START
```

Hidden page

```
                MOV AL, 02              ; the end and use
                MOV BX,FILE_HANDLE      ; end address to get
                MOV CX, 0               ; the file size
                MOV DX, 0
                INT 21H
                MOV SIZ,AX

                MOV AH,42H              ; SET file  POINTER AT
                MOV AL,00               ; STARTING POSITION
                MOV BX,FILE_HANDLE
                MOV CX,0
                MOV DX, 0
                INT 21H

BACK :          MOV AH, 3FH             ; Read file one chacacter
                MOV BX, FILE_HANDLE     ; at a time
                MOV CX, 1
                MOV DX,OFFSET BUFF
                INT 21H

                MOV SI,OFFSET BUFF

                MOV AH,00H              ; Initialize COM 1
                MOV AL,03H
                MOV DX,0
                INT 14H

                MOV AH,01H              ; Transmit character read
                MOV AL,[SI]             ; from file to COM1
                MOV DX,0                ; Display the same
                INT 14H                 ; Character on the monitor
                MOV AH,02H
                MOV DL,[SI]

                INT 21H

                DEC WORD PTR SIZ        ; Decrement size pointer
                CMP SIZ,0               ; Check if end of file
                JNZ BACK
                MOV AH, 4CH             ; Terminate program and
                INT 21 H                ; return to DOS
CODE            ENDS
END START
```

Program to Receive file

```
CODE SEGMENT
ASSUME CS:CODE, DS:CODE
ORG 0100H
START :  MOV AH,00              ; INITIALIZE  COM1 PORT
         MOV AL, 03
         MOV DX,00
         INT 14H
AGAIN :  MOV AH,03          ; Read status of COM1
         MOV DX,00
         INT 14H
```

```
            AND AH,01          ; Check COM1, if it is ready
                               ; to receive
            CMP AH,01          ; data
            JNE AGAIN          ; if not check status again

NEXT :      MOV AH,02          ; Receive data from serial
            MOV DX,00          ; Port COM1
            INT 14H

            CMP AL,1AH         ; Check for end of
            JE STOP            ; file character if yes  stop
            MOV DL,AL          ; Display the received
            MOV AH,02          ; character
            INT 21H
            JMP NEXT           ; Goto receive next character

STOP :      MOV AH,4CH         ; TERMINATION
            INT 21 H
CODE ENDS
END START
```

10.7 Introduction to High-Speed Serial Communication Standards, 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, key boards, 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.

Hidden page

Hidden page

reduce power consumption, this feature is especially useful on battery powered computers where every milliampere counts.

12. Flexibility

USB's four transfer types and two speed (3 with version 2.0) make it feasible for many types of peripherals. There are transfer types suited for exchanging large and small blocks of data, with and without time constraints. For data that cannot tolerate delays, USB can guarantee a transfer rate or maximum time between transfers.

Unlike other interfaces, the USB does not assign specific functions to signals or make other assumptions about how the interface will be use. For example, the status and control lines on the PC's parallel port were defined with the intention of communicating with line printers.

For communicating with common device types such as printers and modems, USB supports classes with defined device requirements and protocols. This saves developers from having to reinvent these for each peripheral.

13. 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.

14. Peripheral support

On the peripheral side, each USB device's hardware must include a controller chip that handles the details of USB communications. Some controllers are complete microcomputers that include a CPU and memory that stores the code that runs inside the peripheral. Others handle only USB-specific tasks, with a data bus that connects to another microcontroller that performs non USB related functions and communicates with the USB controller as needed.

The peripheral is responsible for responding to requests to send and receive configuration data, and for reading and writing other data when requested. In some chips, some of the functions are microcoded in hardware and don't need to be programmed.

Many USB controllers are based on popular architectures such as Intel's 8051, with added circuits and machine codes to support USB.

Most peripheral manufactures provide sample code for their chips.

10.7.2 Limitation of USB

All of USB's advantages mean that it's a good candidate for use with many peripherals. But one interface can't do it all.

From the user's perceptive, the downside to USB includes lack of support on order hardware and operating systems, speed and distance limits that make USB impractical for

Hidden page

The controller

Just about any new PC will have a USB controller and at least two port connectors. If a computer doesn't have a USB controller built into its motherboard, you can add one on an expansion card that plugs into a slot on the PCI bus.

The operating system

The other side of USB support is in the operating system. Windows 95 had some USB support, but the support was greatly improved and enhanced in Windows 98. Windows 95 and Windows 98 can't use the same device drivers. Windows NT 4 doesn't support USB. However, if you're developing a peripheral that needs to run under NT, there are third party products that you can use to create a device driver that enables the peripheral to be used under NT. DOS and Windows 3.x also have no USB support, though again, third party products may be available.

The components

The physical components of the Universal Serial Bus consist of the circuits, connectors, and cables between a host and one or more devices.

The host is a PC or other computer that contains two components; a host controller and a root hub. These work together to enable the operating system to communicate with the devices on the bus. The host controller formats data for transmitting on the bus and translates received data to a format that operating system components can understand. The host controller also performs other functions related to managing communications on the bus. The root hub has one or more connectors for attaching devices. The root hub detects the attachment and removal of devices, carries out requests from the host controller, and passes data between devices and the host controller.

The devices are the peripherals and additional hubs that connect to the bus. A hub has one or more ports for connecting devices. Each device must contain circuits and code that know how to communicate with the host.

10.7.4 USB "tiered star" Topology

As shown in Fig. 10.18 at the center of each star is a hub. Each point on a star is a device that connect to one of the hub's ports. The devices may be additional hubs or other peripherals. The number of points on each star can vary, with a typical hub having two, four, or seven ports. When there are multiple hub in series you can think of them as connecting in a tier, or series, one above the next. (Refer Fig. on next page.).

All of the devices on a bus share one data path to the host computer. Only one device can communicate with the host at a time. If you need more bandwidth , you can add a second data path to the host by installing an expansion card with another host controller and root hub.

Fig. 10.19 shows a few of the possible configurations for a PC with a root hub that has to USB connectors.

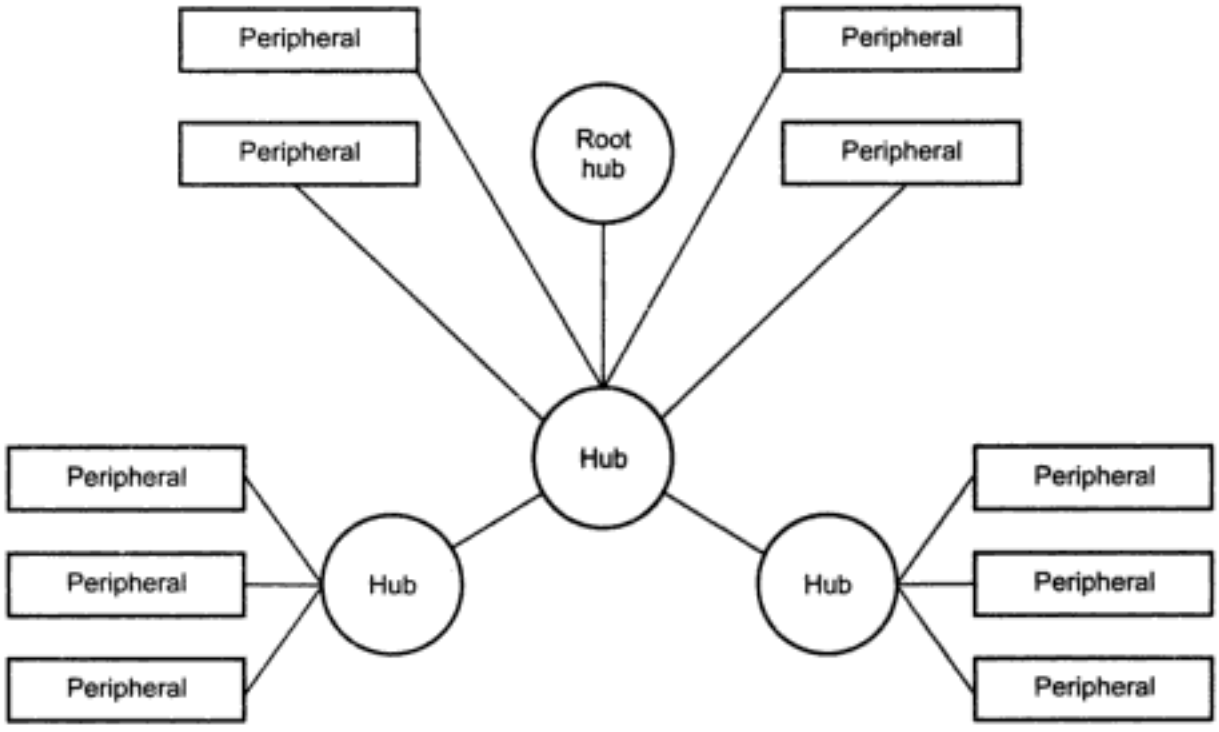


Fig. 10.18 USB tired star topology

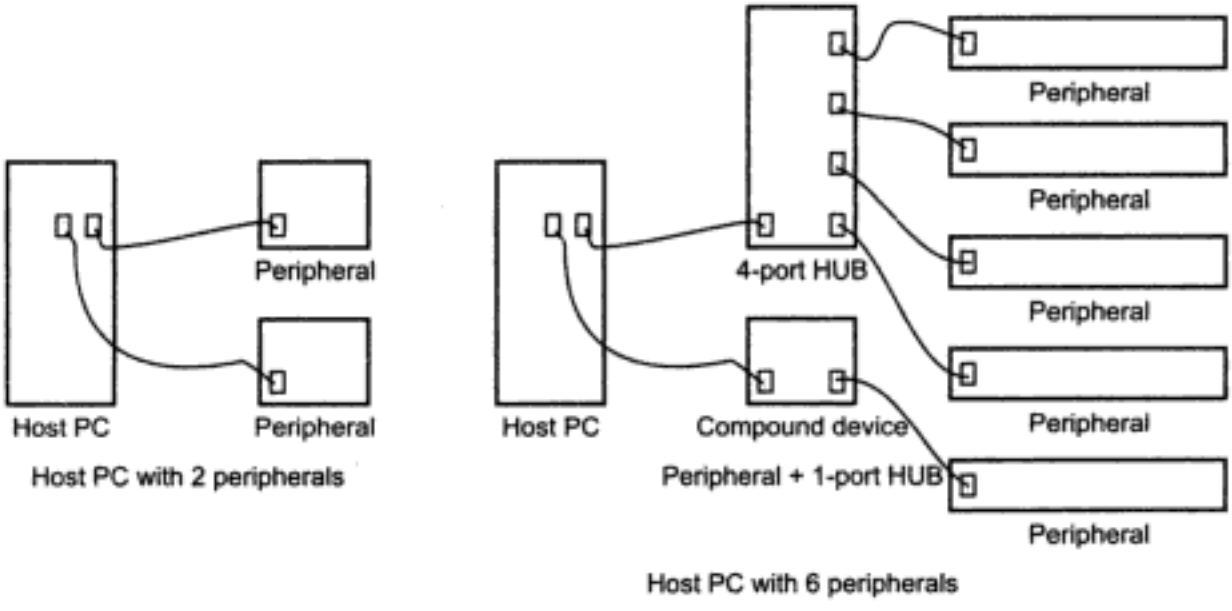


Fig. 10.19 Different configurations for connecting USB devices to a host PC

10.7.5 Terminology used in USB

Host

The host is a PC or other computer that contains two components : a host controller and a root hub.

Hidden page

Hidden page

Hidden page

connect to the bus detect the absence of bus activity for three milli seconds, they must enter the suspend state and limit the current they draw from the bus.

5. Exchange data with the host

After the device is configured, it must respond to request to send and receive data. The host may pole device at regular intervals or only when an application requests to communicate with it. The device must respond to each poll by sending an acknowledgment (ACK) that indicates that it received the data, or a negative acknowledgment (NAK) to indicate that it is busy to handle the data.

10.7.8 USB Communication

USB communication is divided into two types, depending on whether they're used in initial configuration or in applications. In configuration communications, the host learns about the device and prepares it for exchanging data. Most of these communications take place when the host enumerates the device on power up or attachment. Application communications occur when applications on the host exchange data with an enumerated device. These are the communications that carry out the device's purpose. For example, for a keyboard, the application communications are the sending of keypress data to the host, to tell an application to display a character or perform other actions.

1. Configuration communications

During enumeration, the device's firmware responds to a series of standard requests from the host. The device must identify each request, return the requested information, and take other actions specified by the requests.

On PCs, Windows performs the enumeration, so there's no user programming involved. However, to complete the enumeration, Windows must have two files available an INF file that identifies the filename and location of the device's driver, and the device driver itself.

2. Application communications

After the host has exchanged enumeration information with the device and a device driver has been assigned and loaded, the application phase can be fairly straightforward. At the host, applications can use standard Windows API functions to read and write to the device. At the device, transferring data typically requires placing data to send in the USB controller's transmit buffer, reading received data from the receive buffer when a hardware interrupt signals that data has arrived, and on completing a transfer, ensuring the device is ready for the next transfer. Most devices also require some additional support for handling errors and other events.

10.7.9 Elements of Transfer

1. Device endpoints

All transmissions travel to or from a device endpoint. The endpoint is a buffer that stores multiple bytes. Typically it's a block of data memory or a register in the controller

Hidden page

10.7.10 Data Transfer Types

The USB is designed to handle many types of peripherals with varying requirements for transfer rate, response time, and error correcting. There are four types of data transfers each handle different needs, and a peripheral can support the transfer types that are best suited for its purpose.

1. Control transfer

Control transfers are the only type with functions defined by the USB specification. These transfers enable the host to read and select configurations and other settings on the devices being enumerated. Control transfers may also send custom requests that send and receive blocks of data for any purpose. All USB devices must support control transfers.

This data transfer exchanges configuration, setup, and command information between the device and host. CRCs check the data and initiate retransmissions when needed to guarantee the correctness of these packets.

Control Transfers Use Message Pipes. In a message pipe, each transfer begins with a Setup transaction containing a request. To complete the transfer, the host and device may exchange data and status information, or the device may just send status information. There is always at least one transaction that sends information in each direction.

If the request is one that the device supports, it takes the requested action. A device may also respond with a code that indicates that it doesn't support the request.

2. Bulk transfer

Bulk transfers are intended for situations where the rate of transfer isn't critical, such as sending a file to a printer or receiving data from a scanner. In these cases quick transfers are nice, but the data can wait if necessary. If the bus is very busy with other transfers that have guaranteed transfer rates, bulk transfers must wait, but if the bus is idle, bulk transfers are very fast. Only full-speed devices can do bulk transfers. Devices aren't required to support bulk transfers, but a specific device class might require it.

This data transfer moves large amounts of data when timely delivery is not critical. Typical applications include printers and scanners. Bulk transfers are fillers, claiming unuse USB bandwidth when nothing more important is going on. CRCs protect these packets.

3. Interrupt transfer

Interrupt transfers are for devices that must receive the host's or device's attention quickly. Other than control transfers, interrupt transfers are the only way that low speed devices can transfer data. A keyboard or mouse can use interrupt transfers to send keypress or mouse movement data. Both full and low speed devices can do interrupt transfers. Devices aren't required to support interrupt transfers, but a specific device class might require it.

This data transfers, though not interrupt in the CPU diverting sense, poll devices to see if they need service. Peripherals exchanging small amounts of data that need

Hidden page

Hidden page

suspend state, error checking information, and other information about how the chip will be used and the current status of transmitted or received data.

5. USB port

A USB peripheral controller must of course have a USB port and supporting circuits.

6. USB buffers

A USB controller must have transmit and receive buffers for storing USB data.

Review Questions

1. *Compare parallel and serial type of data transfer.*
2. *Classify and explain serial communication systems.*
3. *Explain data communication formats in serial communication.*
4. *Differentiate between Synchronous and Asynchronous data transfer.*
5. *List the features of 8251A.*
6. *Discuss the organization and architecture of 8251A (USART) with a functional block diagram.*
7. *Draw and explain command and mode word formats of 8251A.*
8. *Draw and explain the status word format of 8251A.*
9. *With a neat diagram, explain how 8251 is interfaced with 8086 and used for serial communication.*
10. *Write a short note on RS232C protocol.*
11. *Explain the features of USB.*
12. *Give the details of USB connector with the help of diagram.*
13. *How USB data is generated ? Explain the encoding method used by the USB.*
14. *What is bit stuffing ?*
15. *Draw the flow chart explaining the process of generating USB data from the raw digital serial data.*
16. *Write a short note on USB commands.*
17. *What do you mean by stop and wait flow control ?*

❑❑❑

11

8051 Microcontroller

11.1 Introduction

To make a complete microcomputer system, only microprocessor is not sufficient. It is necessary to add other peripherals such as read only memory (ROM), read/write memory (RAM), decoders, drivers, number of input/output devices to make a complete microcomputer system. In addition, special purpose devices, such as interrupt controller, programmable timers, programmable I/O devices, DMA controllers may be added to improve the capability and performance and flexibility of a microcomputer system.

The key feature of microprocessor based computer system is that it is possible to design a system with a great flexibility. It is possible to configure a system as large system or small system by adding suitable peripherals.

On the other hand, the microcontroller incorporates all the features that are found in microprocessor. However, it has also added features to make a complete microcomputer system on its own. The microcontroller has built-in ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit.

The microcontroller has on-chip (built-in) peripheral devices. These on-chip peripherals make it possible to have single-chip microcomputer system. There are few more advantages of built-in peripherals :

- Built-in peripherals have smaller access times hence speed is more.
- Hardware reduces due to single chip microcomputer system.
- Less hardware, reduces PCB size and increases reliability of the system.

Comparison between Microprocessor and Microcontroller

We have discussed what is a microprocessor and a microcontroller. Let us see the points of differences between them.

No.	Microprocessor	Microcontroller
1.	Microprocessor contains ALU, control unit (clock and timing circuit), different register and interrupt circuit.	Microcontroller contains microprocessor, memory (ROM and RAM), I/O interfacing circuit and peripheral devices such as A/D converter, serial I/O, timer etc.
2.	It has many instructions to move data between memory and CPU.	It has one or two instructions to move data between memory and CPU.
3.	It has one or two bit handling instructions.	It has many bit handling instructions.
4.	Access times for memory and I/O devices are more.	Less access times for built-in memory and I/O devices.
5.	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing the reliability.
6.	Microprocessor based system is more flexible in design point of view.	Less flexible in design point of view.
7.	It has single memory map for data and code.	It has separate memory map for data and code.
8.	Less number of pins are multifunctioned.	More number pins are multifunctioned.

The 8051 is an 8-bit microcontroller designed by Intel. It was optimized for 8-bit math and single bit Boolean operations. Its family-MCS-51 includes 8031, 8051 and 8751 microcontrollers. The Table 11.1 gives the summary of MCS-51 microcontrollers.

Device	Internal Memory		Timer / Event Counters	Interrupts
	Program	Data		
8052AH	8 K × 8 ROM	256 × 8 RAM	3 × 16-Bit	6
8051AH	4 K × 8 ROM	128 × 8 RAM	2 × 16-Bit	5
8051	4 K × 8 ROM	128 × 8 RAM	2 × 16-Bit	5
8032AH	none	256 × 8 RAM	2 × 16-Bit	6
8031AH	none	128 × 8 RAM	2 × 16-Bit	5
8031	none	128 × 8 RAM	2 × 16-Bit	5
8751H	4 K × 8 EPROM	128 × 8 RAM	2 × 16-Bit	5
8751H-12	4 K × 8 EPROM	128 × 8 RAM	2 × 16-Bit	5

Table 11.1 MCS-51 family

In this chapter we are going to see features and the internal hardware details (architecture) of 8051 microcontroller.

11.2 Features of 8051

The features of the 8051 family are as follows :

1) 4096 bytes on - chip program memory.
2) 128 bytes on - chip data memory.
3) Four register banks.
4) 128 User-defined software flags.
5) 64 Kilobytes each program and external RAM addressability.
6) One microsecond instruction cycle with 12 MHz crystal.
7) 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
8) Multiple mode, high-speed programmable serial port.
9) Two multiple mode, 16-bit Timers/Counters.
10) Two-level prioritized interrupt structure.
11) Full depth stack for subroutine return linkage and data storage.
12) Direct Byte and Bit addressability.
13) Binary or Decimal arithmetic.
14) Signed-overflow detection and parity computation.
15) Hardware Multiple and Divide in 4 μsec.
16) Integrated Boolean Processor for control applications.
17) Upwardly compatible with existing 8084 software.

11.3 8051 Microcontroller Hardware

The Fig. 11.1 shows the internal block diagram of 8051. It consists of a CPU, two kinds of memory sections (data memory - RAM and program memory - EPROM/ROM), input/output ports, special function registers and control logic needed for a variety of peripheral functions. These elements communicate through an eight bit data bus which runs throughout the chip referred as internal data bus. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired.

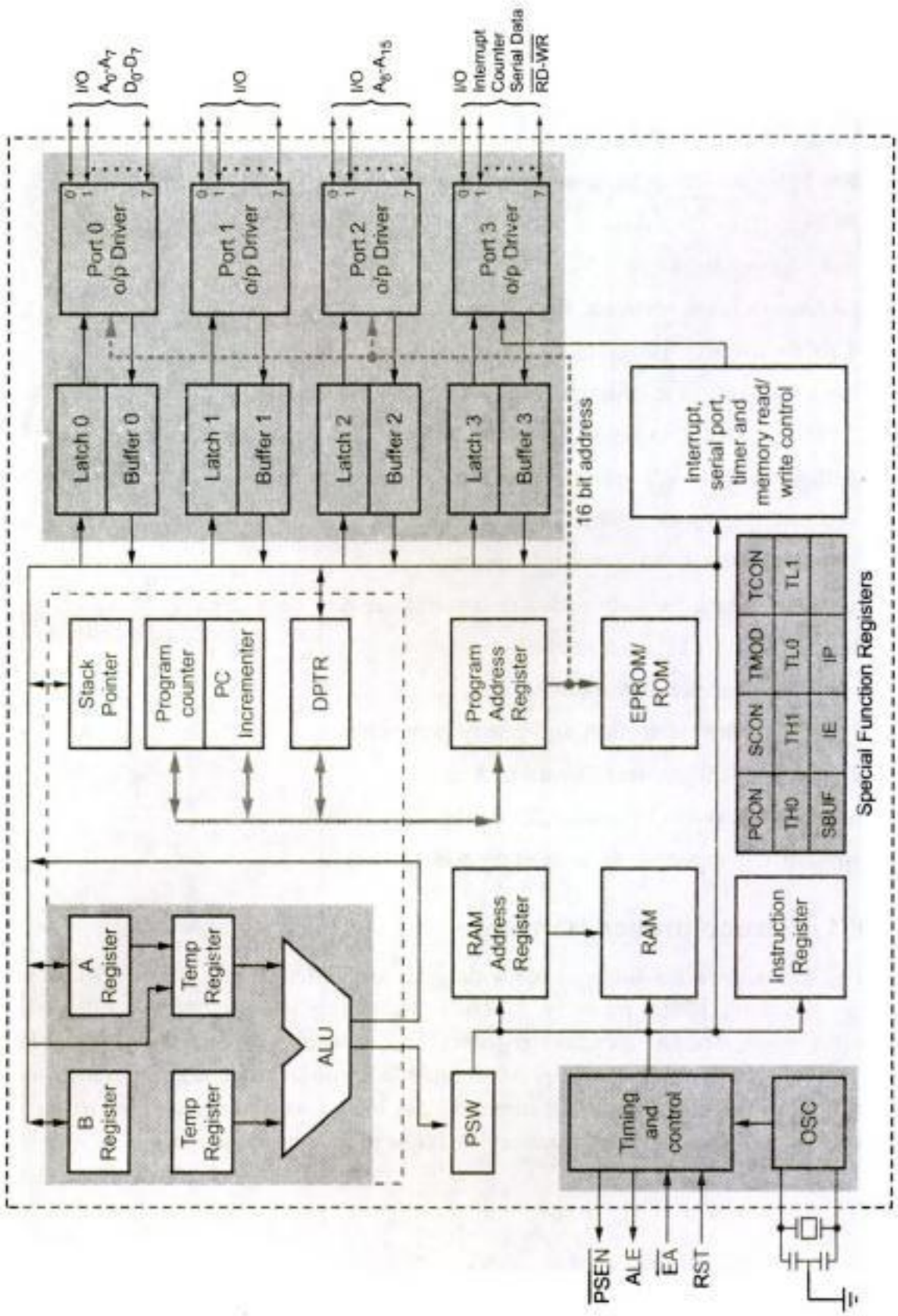


Fig.11.1 Block diagram of 8051

Hidden page

Port 0 (Pins 32 - 39)

Port 0 pins can be used as I/O pins. The ouput drives and input buffers of port 0 are used to access external memory. Port 0 outputs the low order byte of the external memory address, time multiplexed with the data being written or read. Thus, port 0 can be used as a multiplexed address/data bus.

Port 1 (Pins 1 - 8)

Port 1 pins can be used only as I/O pins.

Port 2 (Pins 21 - 28)

The output drives of port 2 are used to access external memory. Port 2 outputs the high order byte of the external memory address when the address is 16 bits wide. Otherwise, port 2 is used as an I/O port.

Port 3 (Pins 10 - 17)

All port pins of port 3 are multifunctional. They have special functions.

Power-supply pins V_{CC} (Pin 40) and V_{SS} (Pin 20)

8051 operates on d.c. power supply of +5 V with respect to ground. The +5 V is to be connected to pins V_{CC} and ground to pin V_{CC} with rated power supply current of 125 mA.

Oscillator pins XTAL2 (Pin 18) and XTAL 1 (Pin 19)

For generating an internal clock signal, the external oscillator is connected at these two pins.

ALE (Address Latch Enable, Pin 30)

AD_0 to AD_7 lines are multiplexed. To demultiplex these lines and for obtaining lower half of an address, an external latch and ALE signal of 8051 is used.

RST (Reset, Pin 9)

This pin is used to reset 8051. For proper reset operation, reset signal must be held high at least for two machine cycles, while oscillator is running.

$\overline{\text{PSEN}}$ (Program Store Enable, Pin 29)

It is the active low output control signal used to activate the enable signal of the external ROM/EPROM. It is activated every six oscillator periods while reading the external memory. Thus, this signal acts as the read strobe to external program memory.

$\overline{\text{EA}}$ (External Access, Pin 31)

When the $\overline{\text{EA}}$ pin is high (connected to V_{CC}), program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. When $\overline{\text{EA}}$ is low (gorunded), all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM.

11.3.2 Central Processing Unit (CPU)

The CPU of 8051 consists of eight-bit Arithmetic and Logic unit with associated registers like A, B, PSW, SP, the sixteen bit program counter and "Data pointer" (DPTR) registers. Alongwith these registers it has a set of special function registers. Along with these registers it has a set of special function registers.

The 8051's ALU can perform arithmetic and logic functions on eight bit variables. The arithmetic unit can perform addition, subtraction, multiplication and division. The logic unit can perform logical operations such as AND, OR, and Exclusive-OR, as well as rotate, clear, and complement. The ALU also looks after the branching decisions. An important and unique feature of the 8051 architecture is that the ALU can also manipulate one bit as well as eight-bit data types. Individual bits may be set, cleared, complemented, moved, tested, and used in logic computation.

11.3.3 Internal RAM

The 8051 has 128-byte internal RAM. It is accessed using RAM address register. The Fig. 11.3 shows the organisation of internal RAM. As shown in the Fig. 11.3, internal RAM of 8051 is organised into three distinct areas :

- Working registers
- Bit Addressable
- General Purpose

1. First thirty-two bytes from address 00H to 1FH of internal RAM constitute 32 working registers. They are organised into four banks of eight registers each. The four register banks are numbered 0 to 3 and are consists of eight registers named R_0 to R_7. Each register can be addressed by name or by its RAM address. Only one register bank is in use at a time. Bits RS_0 and RS_1 in the PSW determine which bank of registers is currently in use. Register banks when not selected can be used as general purpose RAM. On reset, the Bank 0 is selected.
2. The 8051 provides 16 bytes of a bit-addressable area. It occupies RAM byte addresses from 20H to 2FH, forming a total of 128 (16 × 8) addressable bits. An addressable bit may be specified by its bit address of 00H to 7FH, or 8 bits may form any byte address from 20H to 2FH. For example, bit address 4EH refers bit 6 of the byte address 29H.
3. The RAM area above bit addressable area from 30H to 7FH is called general purpose RAM. It is addressable as byte.

See Fig. 11.3 on next page.

11.3.4 Internal ROM

The 8051 has 4 Kbyte of internal ROM with address space from 0000H to 0FFFH. It is programmed by manufacturer when the chip is built. This part cannot be erased or altered after fabrication. This is used to store final version of the program.

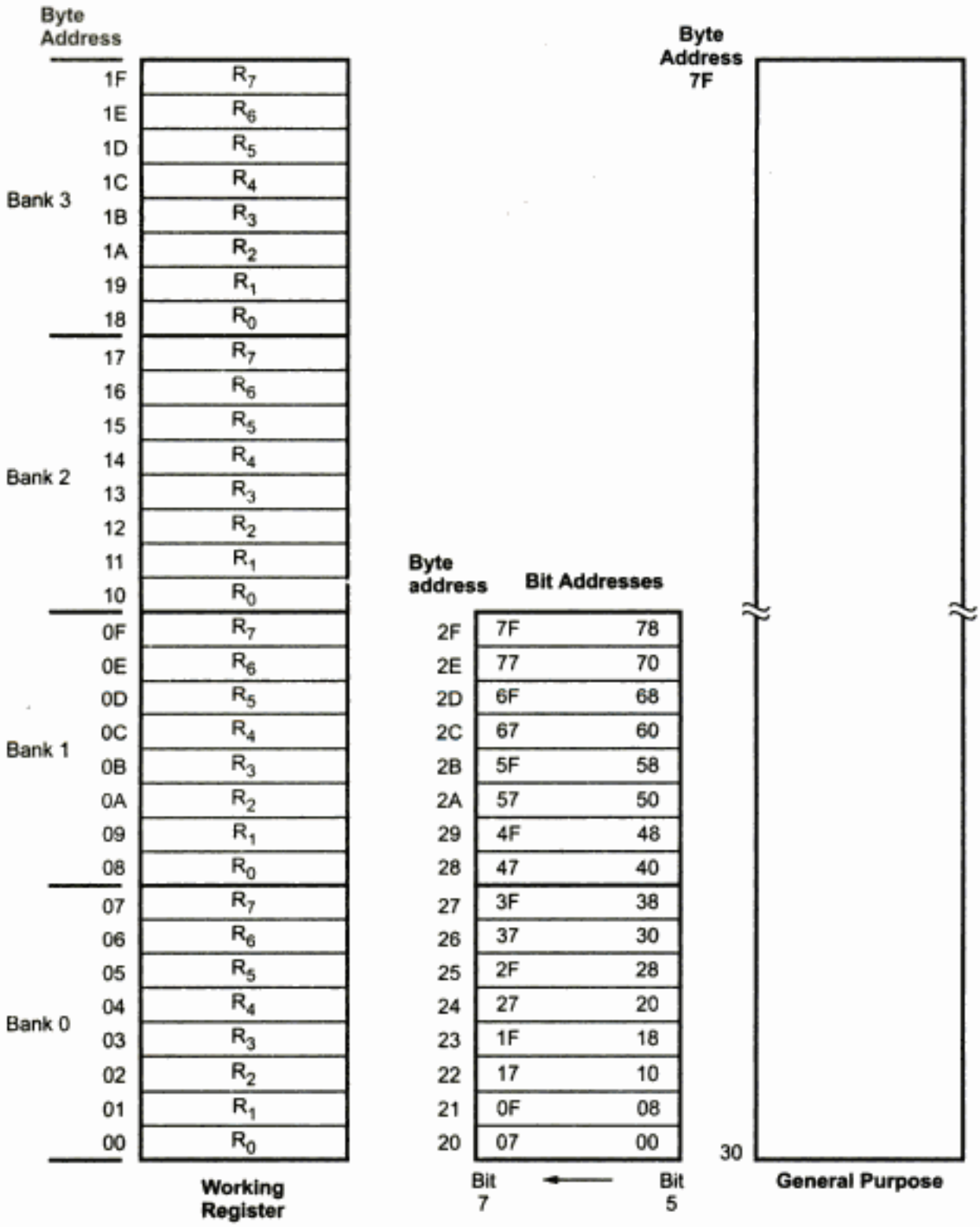
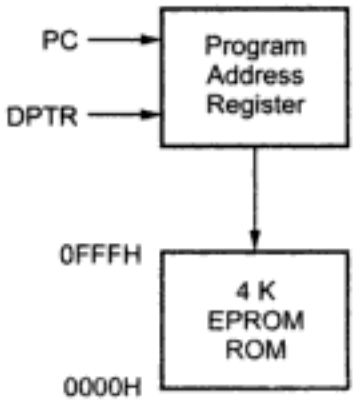


Fig. 11.3 Organisation of internal RAM of 8051

It is accessed using program address register. The program addresses higher than 0FFFH, which exceed the internal ROM capacity will cause the 8051 to automatically fetch code bytes from external program memory. However, code bytes can also be fetched exclusively from an external memory addresses 0000H to FFFFH, by connecting the external access pin (EA) to ground.



11.3.5 Input/Output Ports

The 8051 has 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2, and P3). All four ports are bidirectional, i.e. each pin will be configured as input or output (or both) under software control. Each port consists of a latch, an output driver, and an input buffer.

The output drives of Ports 0 and 2 and the input buffers of Port 0, are used to access external memory. As mentioned earlier, Port 0 outputs the low order byte of the external memory address, time multiplexed with the data being written or read, and Port 2 outputs the high order byte of the external memory address when the address is 16 bits wide. Otherwise Port 2 gives the contents of special function register P2.

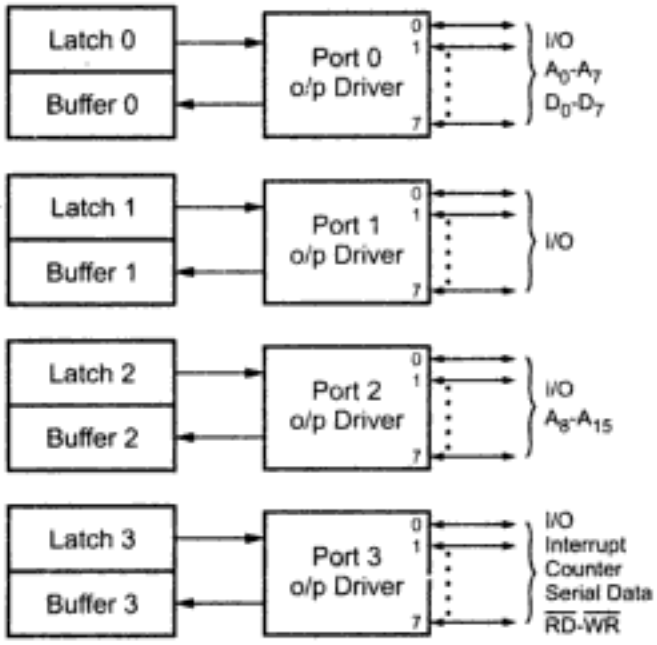


Fig. 11.4 I/O Ports

All port pins of Port 3 are multifunctional. They have special functions as shown below including two external interrupts, two counter inputs, two special data lines and two timing control strobes.

Symbol	Position	Name and Significance
$\overline{RD}$	P3.7	Read data control output. Active low pulse generated by hardware when external data memory is read.
$\overline{WR}$	P3.6	Write data control output. Active low pulse generated by hardware when external data memory is written.
T1	P3.5	Timer/counter 1 external input or test pin.
T0	P3.4	Timer/counter 0 external input or test pin.
$\overline{INT1}$	P3.3	Interrupt 1 input pin. Low-level or falling-edge triggered.
$\overline{INT0}$	P3.2	Interrupt 0 input pin. Low-level or falling-edge triggered.
TXD	P3.1	Transmit Data pin for serial port in UART mode. Clock output in shift register mode.
RXD	P3.0	Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode.

Table 11.2

11.3.6 Register Set of 8051

11.3.6.1 Register A (Accumulator)

It is an 8-bit register. It holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication, and division). The accumulator can be the source or destination for logical operations and a number of special data movement instructions, including look-up tables and external RAM expansion. Several functions apply exclusively to the accumulator : rotate, parity computation , testing for zero , and so on.

11.3.6.2 Register B

In addition to accumulator, an 8-bit B-register is available as a general purpose register when it is not being used for the hardware multiply/divide operation.

11.3.6.3 Program Status Word (Flag Register)

Many instructions implicitly or explicitly affect (or are affected by) several status flags, which are grouped together to form the Program Status Word. Fig. 11.5 shows the bit pattern of the program status word. It is an 8-bit word, containing the information as follows.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
CY	AC	F0	RS1	RS0	OV	-	P

CY Bit 7 - Carry flag

AC Bit 6 - Auxiliary carry flag for BCD operations

F0 Bit 5 - User defined flag (Flag zero)

RS1, RS0 Bit 4-3 - Select the working register banks as follows :

RS1	RS0	Bank Selection	
0	0	00H - 07H	Bank 0
0	1	08H - 0FH	Bank 1
1	0	10H - 17H	Bank 2
1	1	18H - 1FH	Bank 3

OV Bit 2 - Overflow flag

– Bit 1 - Reserved

P Bit 0 - Parity flag (1 = Even parity)

Fig. 11.5 Program status word

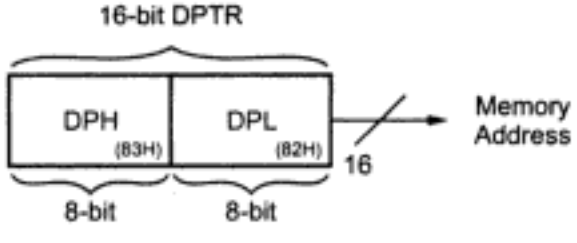
11.3.6.4 Stack and Stack Pointer

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes data to store and retrieve data quickly. The stack pointer register is used by the 8051 to hold an internal RAM address that is called **top of stack.** The stack pointer register is 8-bit wide. It is increased **before** data is stored during PUSH and CALL instructions and decremented **after** data is restored during POP and RET instructions. Thus stack array can reside anywhere in on-chip RAM. The stack pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H. The operation of stack and stack pointer is illustrated in Fig. 11.6.

Please refer Fig. 11.6 on next page.

11.3.6.5 Data Pointer (DPTR)

The data pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its function is to hold a 16 bit address. It may be manipulated as a 16 bit data register or as two independent 8 bit registers. It serves as a base register in indirect jumps, lookup table instructions and external data transfer. The DPTR does not have a single internal address; DPH (83H) and DPL (82H) have separate internal addresses.



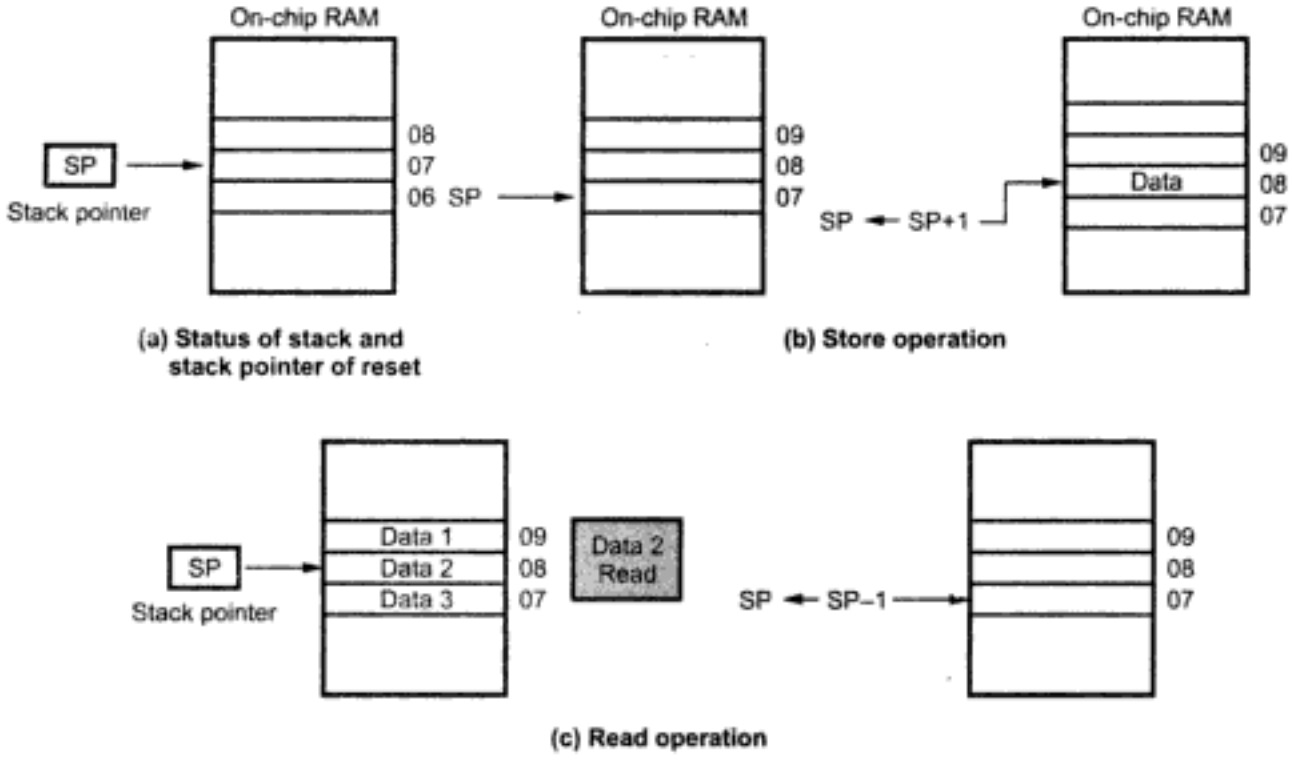


Fig. 11.6

11.3.6.6 Program Counter

The 8051 has a 16-bit program counter. It is used to hold the address of memory location from which the next instruction is to be fetched. Due to this the width of the program counter decides the maximum program length in bytes. For example, 8051 is 16-bit hence it can address upto 2^{16} bytes (64 K) of memory.

The PC is automatically incremented to point the next instruction in the program sequence after execution of the current instruction. It may also be altered by certain instructions. The PC is the only register that does not have an internal address.

11.3.6.7 Special Function Registers

Unlike other microprocessors in the Intel family, 8051 uses memory mapped I/O through a set of special function registers that are implemented in the address space immediately above the 128 bytes of RAM. Fig. 11.7 shows special function bit addresses. All access to the four I/O ports, the CPU registers, interrupt-control registers, the timer/counter, UART, and power control are performed through registers between 80H and FFH.

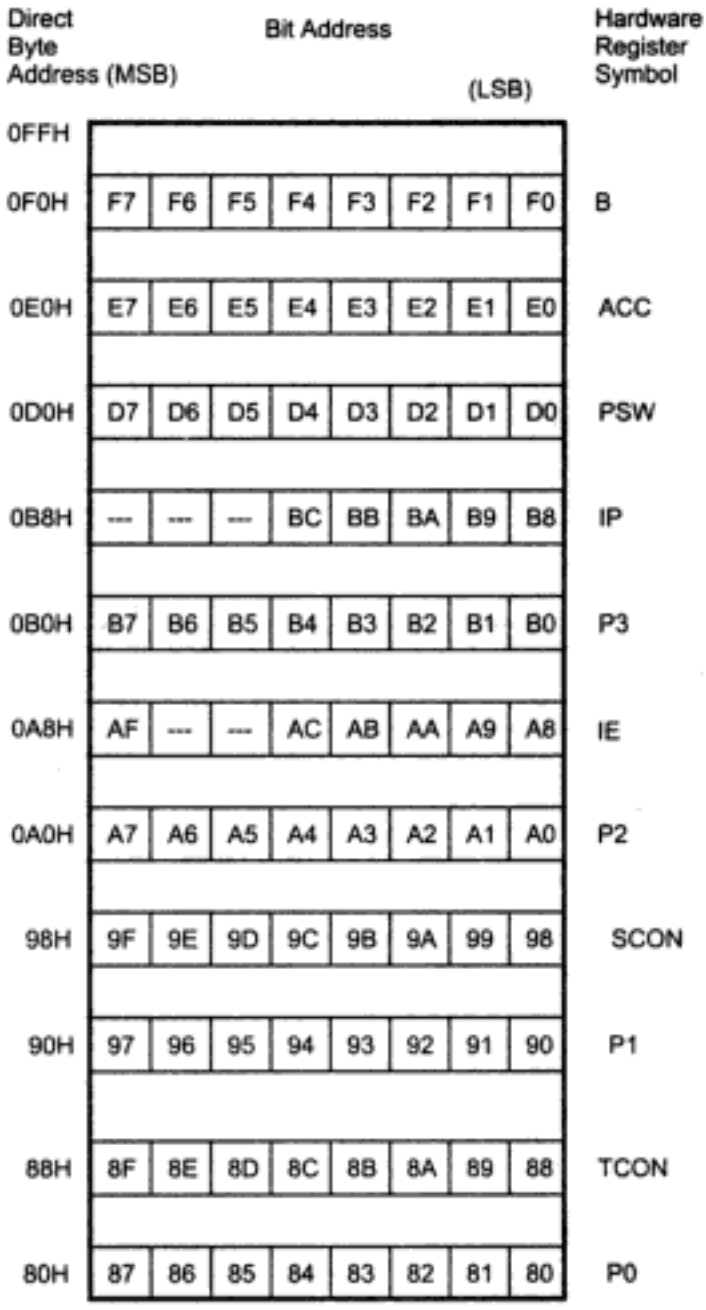


Fig. 11.7 SFR bit address

Table 11.3 contains a list of all the SFRs and their addresses and their value in binary. Comparing Table 11.3 and Table 11.4 shows that all of the SFRs that are byte and bit addressable are located on the first column of the Table 11.4.

Symbol	Name	Address	Value in Binary
*ACC	Accumulator	0E0H	0 0 0 0 0 0 0 0
*B	B Register	0F0H	0 0 0 0 0 0 0 0
*PSW	Program Status Word	0D0H	0 0 0 0 0 0 0 0
SP	Stack Pointer	81H	0 0 0 0 0 1 1 1
DPTR DPL DPH	Data Pointer 2 Bytes Low Byte High Byte	 82H 83H	 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
*P0	Port 0	80H	1 1 1 1 1 1 1 1
*P1	Port 1	90H	1 1 1 1 1 1 1 1
*P2	Port 2	0A0H	1 1 1 1 1 1 1 1
*P3	Port 3	0B0H	1 1 1 1 1 1 1 1
*IP	Interrupt Priority Control	0B8H	8051 X X X 0 0 0 0 0 8052 X X 0 0 0 0 0 0
*IE	Interrupt Enable Control	0A8H	8051 0 X X 0 0 0 0 0 8052 0 X 0 0 0 0 0 0
TMOD	Timer/Counter Mode Control	89H	0 0 0 0 0 0 0 0
*TCON	Timer/Counter Control	88H	0 0 0 0 0 0 0 0
* + T2CON	Timer/Counter 2 Control	0C8H	0 0 0 0 0 0 0 0
TH0	Timer/Counter 0 High Byte	8CH	0 0 0 0 0 0 0 0
TL0	Timer/Counter 0 Low Byte	8AH	0 0 0 0 0 0 0 0
TH1	Timer/Counter 1 High Byte	8DH	0 0 0 0 0 0 0 0
TL1	Timer/Counter 1 LowByte	8BH	0 0 0 0 0 0 0 0
+ TH2	Timer/Counter 2 High Byte	0CDH	0 0 0 0 0 0 0 0
+ TL2	Timer/Counter 2 Low Byte	0CCH	0 0 0 0 0 0 0 0
+ RCAP2H	T/C 2 Capture Reg. High Byte	0CBH	0 0 0 0 0 0 0 0
+ RCAP2L	T/C 2 Capture Reg. Low Byte	0CAH	0 0 0 0 0 0 0 0
* SCON	Serial Control	98H	0 0 0 0 0 0 0 0
SBUF	Serial Data Buffer	99H	Interminate
PCON	Power Control	87H	HMOS 0 X X X X X X X CHMOS 0 X X X 0 0 0 0

Table 11.3 List of all SFRs (* = Bit addressable, + = 8052 only)

	Bit Addressable	8 Bytes							
F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8	T2CON		RCAP2L	RCAP2H	TL2	TH2			CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PCON	87

Table 11.4 SFR memory map

11.4 Memory Organization in 8051

Fig. 11.8 shows the basic memory structure for 8051. It can access upto 64 K program memory and 64 K data memory. The 8051 has 4 Kbytes of internal program memory and 256 bytes of internal data memory.

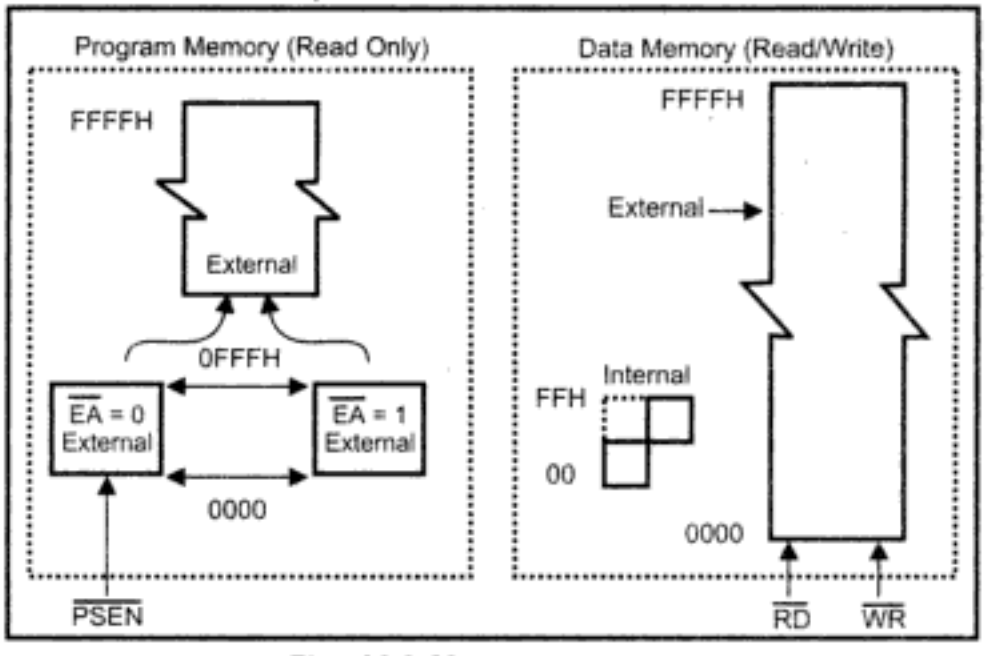


Fig. 11.8 Memory structure

Hidden page

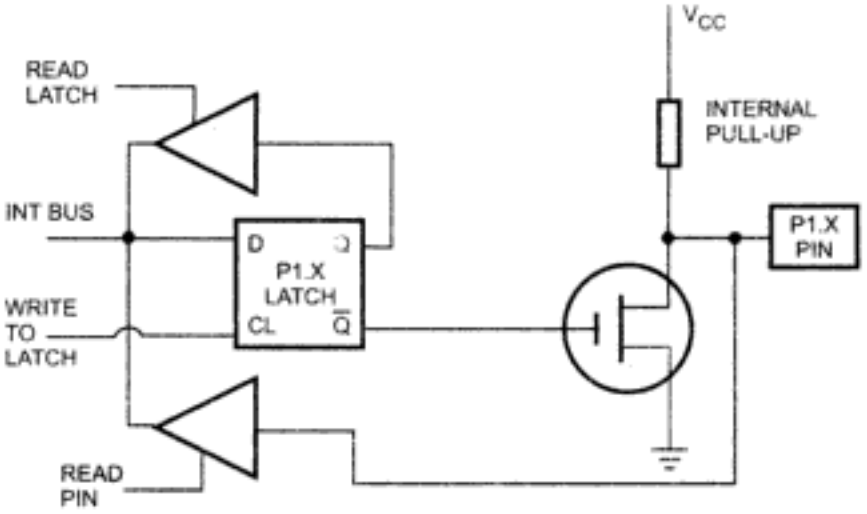


Fig. 11.9 (b) Port 1 bit

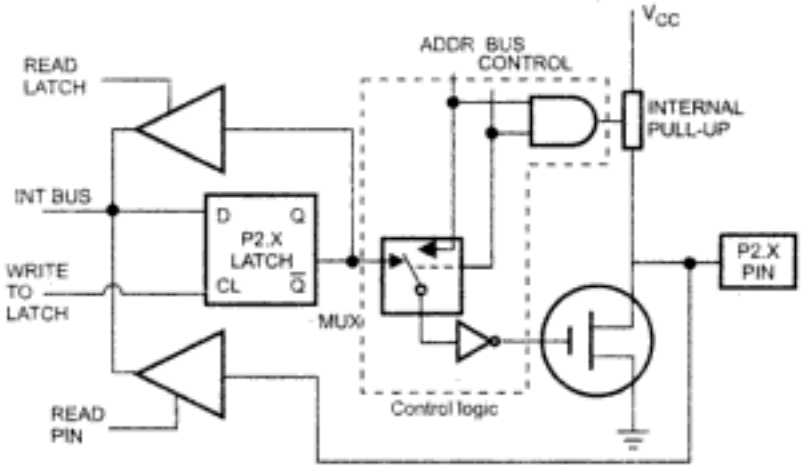


Fig. 11.9 (c) Port 2 bit

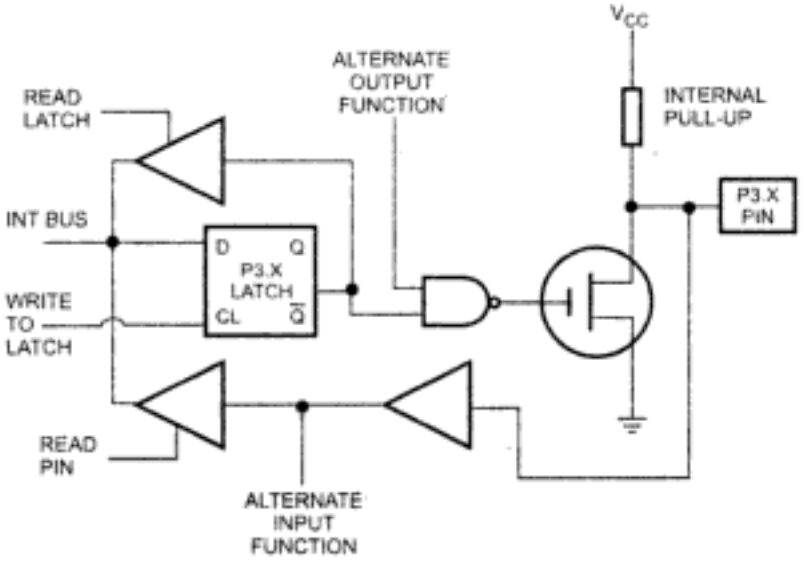


Fig. 11.9 (d) Port 3 bit

Fig. 11.9 8051 port bit latches and I/O buffers

As shown in the Fig. 11.9, for Port 0 and Port 2 drivers are switchable to internal ADDR/DATA and ADDR bus, respectively, by internal CONTROL signal. The switching is required to access external memory. During external memory accesses, the P2 SFR remains unchanged, but P0 SFR gets 1s written to it.

As mentioned earlier, Port 3 has multifunction pins. Therefore, each pin of Port 3 can be programmed to use as I/O or as one of the alternate function. This is achieved by the another control input, "alternate output function", as shown in the Fig. 11.9. When latch bit of Port 3 contains 1, the output level is controlled by control input, "alternate output function."

The port pin can be configured as an input by writing 1 in the latch bit of the corresponding pin. It turns OFF the output driver FET. Then for, Ports 1, 2 and 3, the pin is pulled high by the internal pull-up, but can be pulled low by an external source. There is no internal pull-up for port 0. Therefore, its output pin floats when 1 is written in the latch bit, and pin can be used as a high impedance input. The port 0 is said to be "**true bidirectional**", because when configured as an input it floats.

On the otherhand, the output of Ports 1, 2 and 3 are pulled high with pull-up registers, when configured as an input. Thus they are sometimes called "**quasi bidirectional**" ports.

The Table 11.5 summarizes the functions of four ports.

Port	Functions
Port 0	• Used as an I/O port • Used as a bi-directional low-order address and data bus for external memory.
Port 1	• Used as an input/output port.
Port 2	• Used as an input/output port. • Used as a higher-order address bus for external memory.
Port 3	• Used as an input/output port or used for alternate function as shown below. P3.0-RXD Serial data input P3.1-TXD Serial data output P3.2-$\overline{\text{INT0}}$ External interrupt 0 P3.3-$\overline{\text{INT1}}$ External interrupt 1 P3.4-T0 External timer 0 input P3.5-T1 External timer 1 input P3.6-$\overline{\text{WR}}$ External memory write signal P3.7-$\overline{\text{RD}}$ External memory read signal

Table 11.5 Port functions

11.6 External Data Memory and Program Memory

We have seen that 8051 has internal data and code memory with limited memory capacity. This memory capacity may not be sufficient for some applications. In such situations, we have to connect external ROM/EPROM and RAM to 8051 microcontroller to increase the memory capacity. We also know that ROM is used as a program memory and RAM is used as a data memory. Let us see how 8051 accesses these memories.

11.6.1 External Program Memory

Fig. 11.10 shows a map of the 8051 program memory.

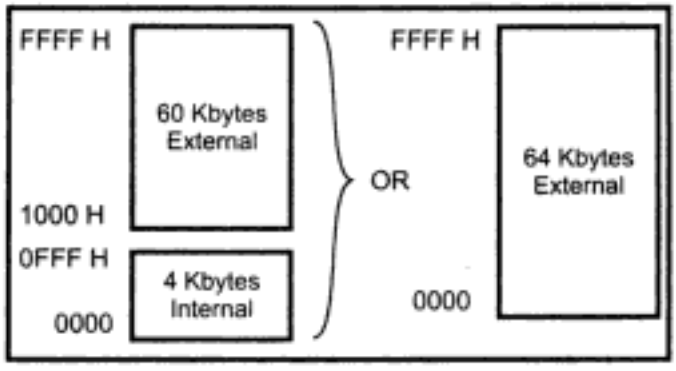


Fig. 11.10 The 8051 program memory

In 8051, when the $\overline{EA}$ pin is connected to V_{CC}, program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. On the other hand when $\overline{EA}$ pin is grounded, all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM. The $\overline{PSEN}$ signal is used to activate output enable signal of the external ROM/EPROM, as shown in the Fig. 11.11.

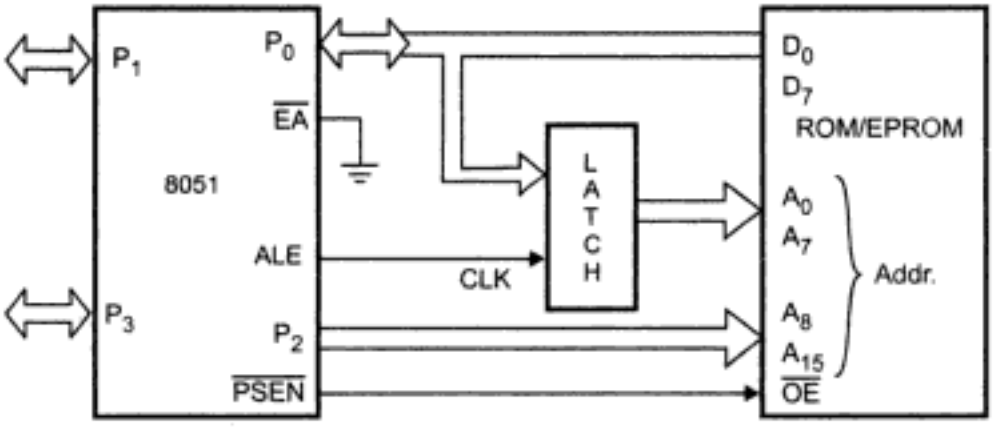


Fig. 11.11 Accessing external program memory

Hidden page

Instructions to Access External ROM / Program Memory

The table 11.6 explains the instructions to access external ROM/program memory.

Mnemonic	Operation
MOVC A, @ A + DPTR	Copy the contents of the external ROM address formed by adding A and the DPTR, to A.
MOVC A, @ A + PC	Copy the contents of the external ROM address formed by addding A and the PC, to A.

Table 11.6

11.6.2 External Data Memory

Fig. 11.14 shows a map of the 8051 data memory

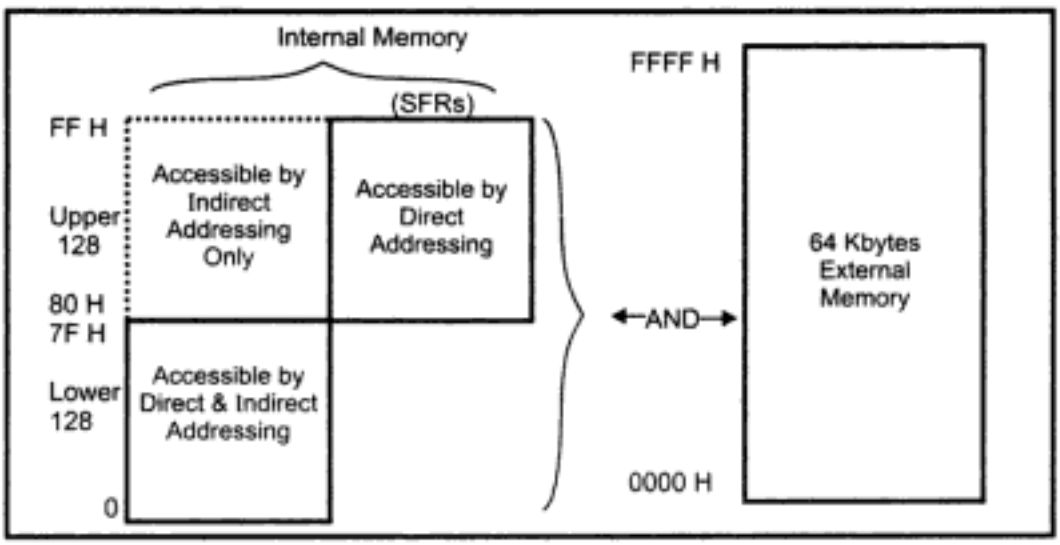


Fig. 11.14 A map of the 8051 data memory

The 8051 can address upto 64 Kbytes of external data memory. The "MOVX" instruction is used to access the external data memory. The internal data memory space for 8051 is divided into three blocks : Lower 128 bytes, Upper 128 bytes and SFRs. The upper addresses and SFRs occupy the same block of address space, 80H through FFH, although they are physically separate entities. As shown in the Fig. 11.14, the upper address space is accessible by indirect addressing only and SFRs are accessible by direct addressing only. On the other hand, lower address space can be accessed either by direct addressing or by indirect addressing.

Hidden page

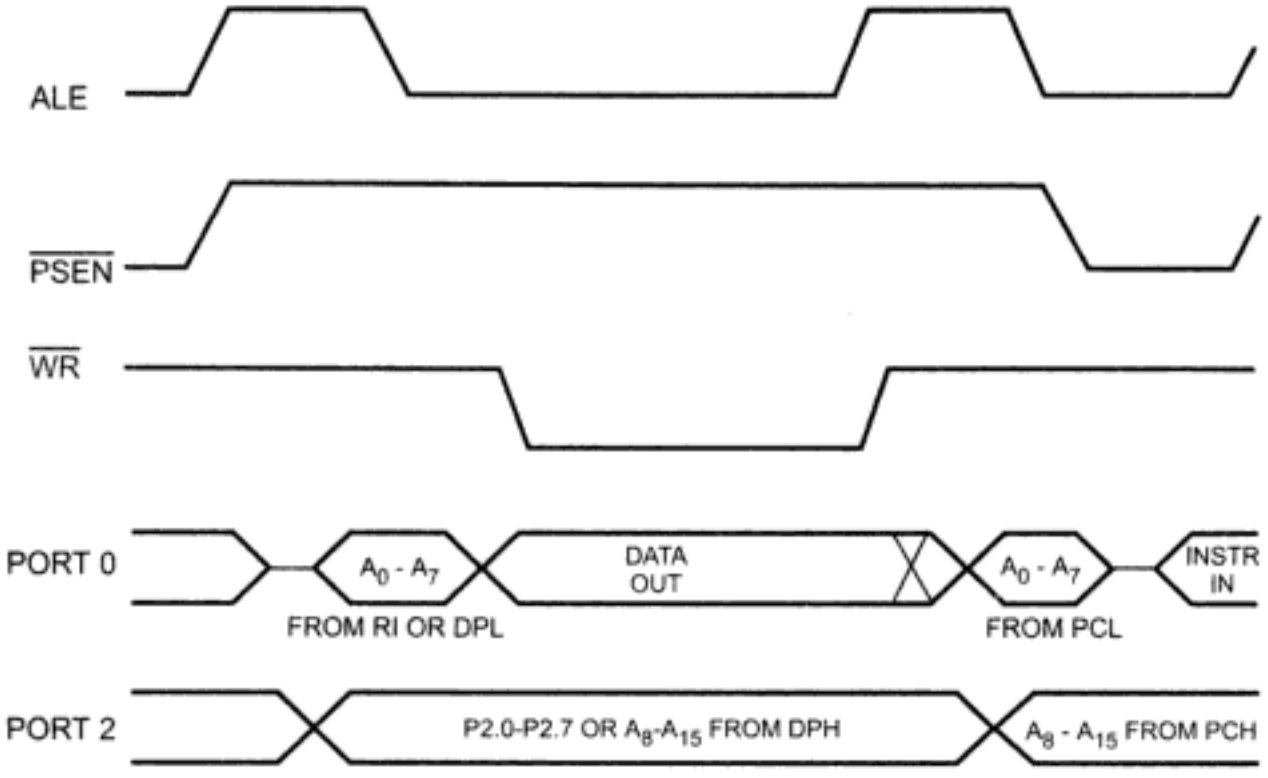


Fig. 11.16 (b) Timing waveforms for external data memory write cycle

Instructions to Access External Data Memory

The table 11.7 explains the instruction to access external data memory.

Mnemonic	Operation
MOVX A, @Rp	Copy the contents of the external address in Rp to A.
MOVX A, @DPTR	Copy the contents of the external address in DPTR to A.
MOVX @ Rp, A	Copy data from A to the external address in Rp.
MOVX @DPTR, A	Copy data from A to the external address in DPTR.

Table 11.7

11.6.3 Important Points to Remember in Accessing External Memory

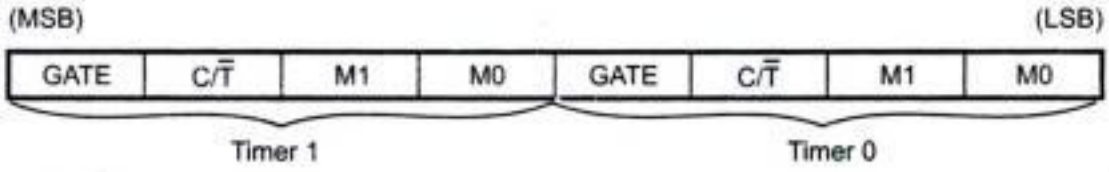
- All external data moves with external ROM or external RAM involve the A register.
- While accessing external memory, R_p can address 256 bytes and DPTR can address 64 Kbytes.
- MOVX instruction is used to access external RAM or I/O addresses.

When PC is used to access external ROM, it is incremented by 1 (to point to the next instruction) before it is added to A to form the physical address of external ROM.

Hidden page

11.7.2 Timer 0 and Timer 1

In this timers, "Timer" or Counter" mode is selected by control bits $C/\overline{T}$ in the Special Function Register TMOD (Fig. 11.18). These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in TMOD. Modes 0, 1 and 2 are same for both Timer/Counters. Mode 3 is different. The four operating modes are described as follows :



GATE Gating control when set. Timer/Counter "x" is enabled only while "INTx" pin is high and "TRx" control bit is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set.

$C/\overline{T}$ Timer or Counter selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M1	M0	Operating Mode
0	0	8-bit Timer/Counter "THx" with "TLx"s 5-bit prescaler.
0	1	16-bit Timer/Counter "THx" with "TLx" are cascaded; there is no prescaler.
1	0	8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped.

Fig. 11.18 TMOD : Timer/counter mode control register

MODE 0

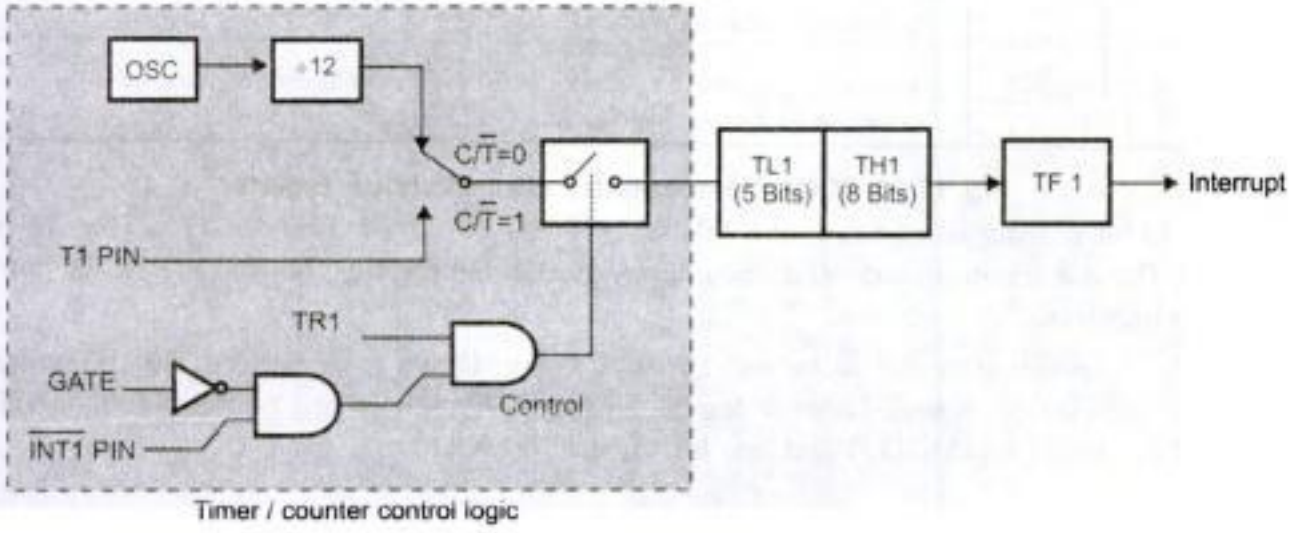
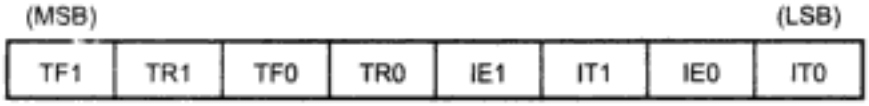


Fig. 11.19 Timer/counter 1 mode 0 : 13-bit counter

Both Timers in Mode 0 is an 8-bit Counter with a divide-by-32 prescaler. This 13-bit timer is MCS-48 compatible. Fig. 11.19 shows the Mode 0 operation as it applies to Timer 1. In this mode, the Timer register is configured as a 13-bit register. As the count rolls over from all 1s to all 0s, it sets the Timer interrupt flag TF1. The counted input is enabled to the Timer when TR1 = 1 and either GATE = 0 or $\overline{INT1}$ = 1. (Setting GATE = 1 allows the Timer to be controlled by external input $\overline{INT1}$, to facilitate pulse width measurements.) TR1 is a control bit in the Special Function Register TCON (Fig. 11.20) GATE is in TMOD.



Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.
TF0	TCON.5	Timer 0 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.
IE1	TCON.3	Interrupt 1 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT0	TCON.0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

Fig 11.20 TCON-timer/counter control/status register

The 13-bit register consists of all 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the registers.

Mode 0 operation is the same for Timer 0 as for Timer 1. Substitute TR0, TF0 and $\overline{INT0}$ for the corresponding Timer 1 signals in Fig. 11.19. There are two different GATE bits, one for Timer 1 (TMOD.7) and one for Timer 0 (TMOD.3).

Hidden page

MODE 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0. Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. The logic for Mode 3 on Timer 0 is shown in Fig. 11.23. TL0 uses the Timer 0 control bits : $C/\overline{T}$, GATE, TR0, $\overline{INT0}$, and TF0. TH0 is locked into a timer mode (counting machine cycles) and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the : Timer 1 interrupt.

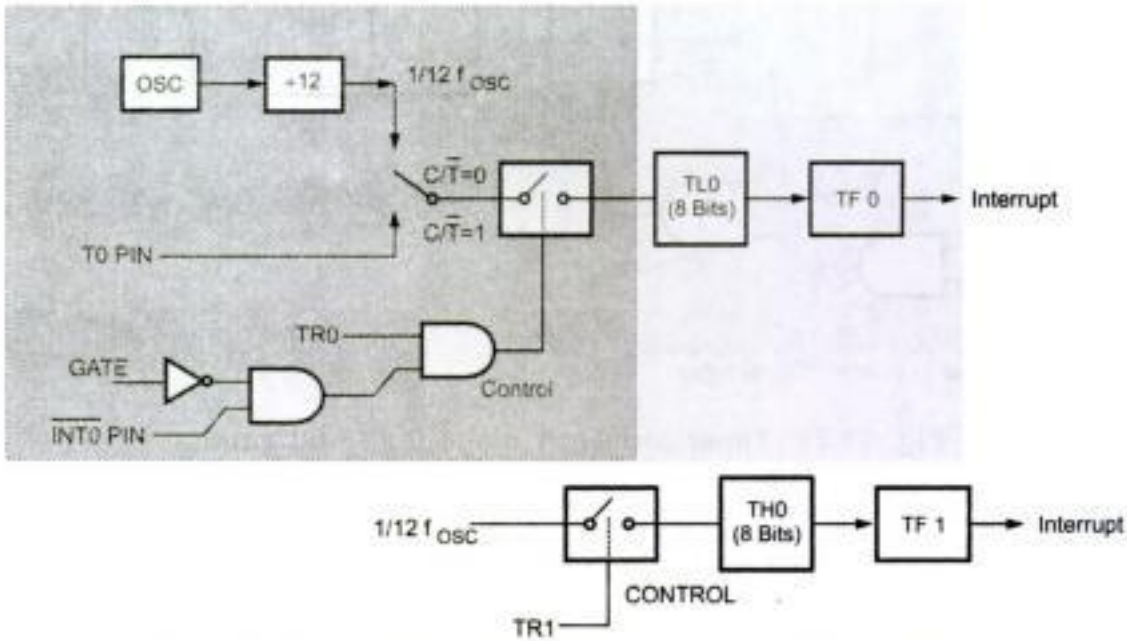


Fig. 11.23 Timer/counter 0 mode 3 : two 8-bit counters

Mode 3 is provided for applications requiring an extra 8-bit timer or counter. With Timer 0 in Mode 3, an 8051 can look like it has three Timer/Counters, and an 8052, like it has four. When timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

The Table 11.8 summarizes the modes of timers.

Mode	Brief Description
Mode 0	13-bit timer (TL-5 bits and TH-8 bits). Counter overflow is indicated by time interrupt flag.
Mode 1	16-bit timer (TL-8 bits and TH-8 bits). Rest is same as mode 0.
Mode 2	Automatic reload mode. 8-bit counter (TL-8 bit) overflow from TL not only sets TF, but also reloads TL with the contents of TH.
Mode 3	Establishes TL and TH as two separate counters.

Table 11.8 Summary of timer modes

Hidden page

RI	SCON.0	Receive Interrupt flag. Set by hardware when byte received. Cleared by software after servicing.
	Note : **Mode**	The state of (SM0, SM1) selects ; **SM0 SM1**
	0	0 0 - Shift register ; baud = f/12
	1	0 1 - 8-bit UART, variable data rate.
	2	1 0 - 9-bit UART, fixed data rate ; baud = f/32 or f/64
	3	1 1 - 9-bit UART, variable data rate.

Fig. 11.24 (a) SCON-serial port control/status register

(MSB) (LSB)

7	6	5	4	3	2	1	0
SMOD	–	–	–	GF1	GF0	PD	IDL

Symbol	Position	Name and Significance
SMOD	PCON.7	Serial baud rate modify bit. It is 0 at reset. It is set to 1 by program to double the baud rate.
–	PCON.6-4	Not defined
GF1	PCON.3	General purpose user flag bit 1. Set/cleared by program.
GF0	PCON.2	General purpose user flag bit 0. Set/cleared by program.
PD	PCON.1	Power down bit. It is set to 1 by program to enter power down configuration for CHMOS microcontrollers.
IDL	PCON.0	Idle mode bit. It is set to 1 by program to enter idle mode configuration for CHMOS microcontrollers.
Note : PCON is not bit addressable		

Fig. 11.24 (b) PCON register

11.8.1 Operating Modes for Serial Port

MODE 0

In this mode, serial data enters and exits through RXD. TXD outputs the shift clock. 8 bits are transmitted/received : 8 data bits (LSB first). The baud rate is fixed at 1/12 the oscillator frequency.

MODE 1

In this mode, 10 bits are transmitted (through TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SCON. The baud rate is variable.

MODE 2

In this mode, 11 bits are transmitted (thorugh TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8 in SCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8. On receive, the 9th data bit goes into RB8 in Special Function Regsiter SCON, while the stop bit is ignored. The baud rate is programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ the oscillator frequency.

MODE 3

In this mode, 11 bits are transmitted (through TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except the baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SBUF as a destination register. Reception is initiated in Mode 0 by the condition RI = 0 and REN = 1. Reception is initiated in the other modes by the incoming start bit if REN = 1.

The Table 11.9 summaries the four serial port modes provided by 8051.

Mode	Transmission Format	Baud Rate
0	8-data bits	$\frac{1}{12}$ oscillator frequency
1	10-bit (start bit + 8-data bits + stop bit)	Variable
2	11-bit (start bit + 8-data bits + programmable 9^{th} data bit + stop bit)	Programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ oscillator frequency
3	11-bit (start bit + 8 data bit + programmable 9^{th} data bit + stop bit)	Variable

Table 11.9 Summary of serial port modes

11.8.2 Serial Port Control Register

The serial port control and status register is the Special Function Register SCON, shown in Fig. 11.25. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial port interrupt bits (TI and RI).

11.8.3 Generating Baud Rates

Serial Port in Mode 0 :

Mode 0 has a fixed baud rate which is 1/12 of the oscillator frequency. To run the serial port in this mode none of the Timer/Counters need to be set up. Only the SCON register needs to be defined.

$$\text{Baud Rate} = \frac{\text{Osc Freq}}{12}$$

Serial port in Mode 1

Mode 1 has a variable baud rate. The baud rate can be generated by either Timer 1 or Timer 2 (8052 only).

Using Timer/Counter 1 to Generate Baud Rates

For this purpose, Timer 1 is used in mode 2 (Auto-Reload).

$$\text{Baud Rate} = \frac{k \times \text{Oscillator Freq.}}{32 \times 12 \times [256 - TH1)]}$$

If SMOD = 0, then k = 1.

If SMOD = 1, then k = 2. (SMOD is the PCON register)

Most of the time the user knows the baud rate and needs to know the reload value for TH1. Therefore, the equation to calculate TH1 can be written as :

$$TH1 = 256 - \frac{k \times \text{Osc Freq.}}{384 \times \text{baud rate}}$$

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate. In this case, the user may have to choose another crystal frequency.

Since the PCON register is not bit addressable, one way to set the bit is logical ORing the PCON register. (i.e. ORL PCON, #80H). The address of PCON is 87H.

Using Timer/Counter 2 to Generate Baud Rates

For this purpose, Timer 2 must be used in the baud rate generating mode. If Timer 2 is being clocked through pin T2 (P1.0) the baud rate is :

$$\text{Baud Rate} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

And if it is being clocked internally the baud rate is :

$$\text{Baud Rate} = \frac{\text{Osc Freq.}}{32 \times [65536 - (RCAP2H, RCAP2L)]}$$

To obtain the reload value for RCAP2H and RECAP2L the above equation can be rewritten as :

$$RCAP2H, RCAP2L = 65536 - \frac{\text{Osc Freq.}}{32 \times \text{baud rate}}$$

Serial Port in Mode 2

The baud rate is fixed in this mode and is $1/32$ or $1/64$ of the oscillator frequency depending on the value of the SMOD bit in the PCON register. In this mode none of the Timers are used and the clock comes from the internal phase 2 clock.

SMOD = 1, Baud Rate = $1/32$ Osc Freq.

SMOD = 0, Baud Rate = $\frac{1}{64}$ Osc Freq.

To set the SMOD bit : ORL PCON,#80H. The address of PCON is 87H.

Serial Port in Mode 3

The baud rate in mode 3 is variable and sets up exactly the same as in mode 1.

11.9 Interrupt Structure

The 8051 provides 5 interrupt sources. The 8052 provides 6. These are shown in Fig. 11.25. The external Interrupts $\overline{INT0}$ and $\overline{INT1}$ can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON. The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by the hardware when the service routine is vectored to only if the interrupt was transition-activated. If the interrupt was level-activated, then the external requesting source is what controls the request flag, rather that the on-chip hardware.

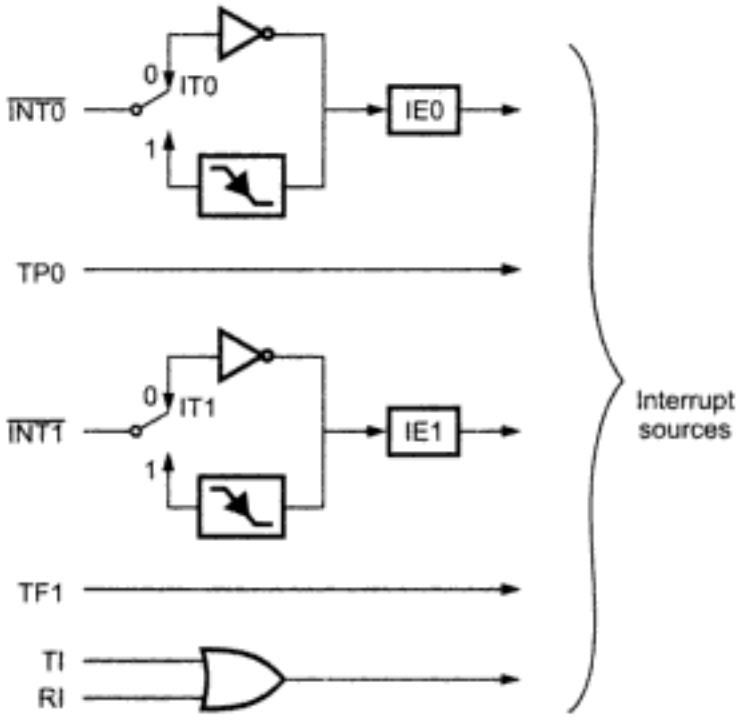


Fig. 11.25 MCS - 51 Interrupt structure

The Timer 0 and Timer 1 Interrupts are generated by TF0 and TF1, which are set by a rollover in their respective Timer/Counter registers (except see Timer 0 in Mode 3). The timer flag set upon generation of interrupt is cleared by the on-chip hardware when microcontroller starts execution of particular interrupt service routine.

The Serial port Interrupt is generated by the logical OR of RI and TI. Neither of these flags is cleared by hardware when the service routine is vectored to. In fact, the service

Hidden page

(MSB) (LSB)

-	-	-	PS	PT1	PX1	PT0	PX0

Symbol	Position	Name and Significance
-	IP.7	(Reserved)
-	IP.6	(Reserved)
-	IP.5	(Reserved)
PS	IP.4	Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port.
PT1	IP.3	Timer 1 Priority control bit Set/cleared by software to specify high/low priority interrupts for timer/counter 1.
PX1	IP.2	External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1.
PT0	IP.1	Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0.
PX0	IP.0	External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0.

Fig. 11.27 IP - Interrupt priority control register

If two requests of different priority levels are received simultaneously, the request of higher priority level is servided. If requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence, as follows :

No.	Source	Priority within Level
1.	IE0	(highest)
2.	TF0	
3.	IE1	
4.	TF1	
5.	RI + TI	(lowest)

Note that the "priority within level" structure is only used to resolve simultaneous requests of the same priority level.

The IP register contains a number of unimplemented bits. IP.7 and IP.6 are vacant in the 8052s, and in the 8051s these and IP.5 are vacant. User software should not write 1s to these bit positions, since they may be used in future MCS-51 products.

Hidden page

11.10 Interfacing 8255 for I/O Expansion

As seen earlier, for interfacing external memory to the 8051, port 0 and port 2 are used as multiplexed address/data bus and a higher order address bus respectively. If the circuit needs the on chip peripherals (e.g. serial I/O and Interrupts) then only 1 port is available for I/O. In such situations, I/O expansion is necessary and it is achieved by using 8255. The Fig. 11.28 shows the expanded I/O ports using 8255. Data bus of 8255 is connected to the Port 0. Address lines A0 and A1, after latches are connected to A0 and A1 of the 8255.

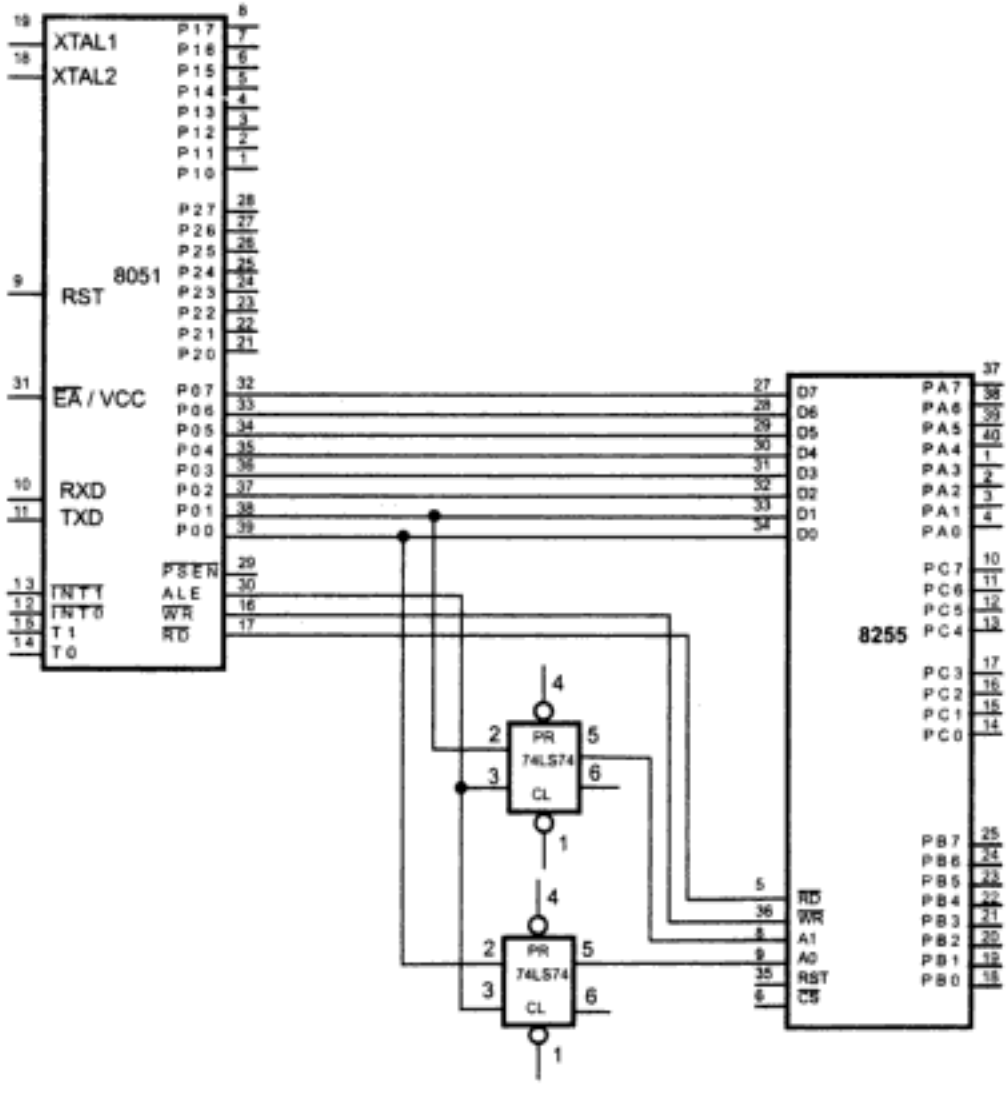


Fig. 11.28 I/O expansion using 8255

Hidden page

Hidden page

Hidden page

Mnemonic and Description	Instruction Code			
LOGIC	76543210	76543210	76543210	76543210
NOT = Invert	1111011w	mod 010 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	110100vw	mod 100 r/m		
SHR = Shift Logical Right	110100vw	mod 101 r/m		
SAR = Shift Arithmetic Right	110100vw	mod 111 r/m		
ROL = Rotate Left	110100vw	mod 000 r/m		
ROR = Rotate Right	110100vw	mod 001 r/m		
RCL = Rotate Through Carry Flag Left	110100vw	mod 010 r/m		
RCR = Rotate Through Carry Right	110100vw	mod 011 r/m		
AND = And:				
Reg./Memory with Register to Either	001000dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 100 r/m	data	data if w =1
Immediate to Accumulator	0010010w	data	data if w = 1	
TEST = And Function to Flags, No Result:				
Register/Memory and Register	1000010w	mod reg r/m		
Immediate Data and Register/Memory	1111011w	mod 000 r/m	data	data if w = 1
Immediate Data and Accumulator	1010100w	data	data if w = 1	
OR = Or:				
Reg./Memory and Register to Either	000010dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 001 r/m	data	data if w = 1
Immediate to Accumulator	0000110w	data	data if w = 1	
XOR = Exclusive or:				
Reg./Memory and Register to Either	001100dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 110 r/m	data	data if w = 1
Immediate to Accumulator	0011010w	data	data if w = 1	
STRING MANIPULATION				
REP = Repeat	1111001z			
MOVS = Move Byte/Word	1010010w			
CMPS = Compare Byte/Word	1010011w			
SCAS = Scan Byte/Word	1010111w			
LODS = Load Byte/Wd to AL/AX	1010110w			
STOS = Stor Byte/Wd from AL/A	1010101w			
CONTROL TRANSFER **CALL = Call:**				
Direct within Segment	11101000	disp-low	disp-high	
Indirect within Segment	11111111	mod 010 r/m		
Direct Intersegment	10011010	offset-low	offset-high	
		seg-low	seg-high	
Indirect intersegment	11111111	mod 011 r/m		

Mnemonic and Description	Instruction Code		
JMP = Unconditional Jump	76543210	76543210	76543210
Direct within Segment	11101001	disp-low	disp-high
Direct within Segment-Short	11101011	disp	
Indirect within Segment	11111111	mod 100 r/m	
Direct Intersegment	11101010	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	11111111	mod 101 r/m	
RET = Return from CALL:			
Within Segment	11000011		
Within Seg Adding Immed to SP	11000010	data-low	data-high
Intersegment	11001011		
Intersegment Adding Immediate to SP	11001010	data-low	data-high
JE/JZ = Jump on Equal/Zero	01110100	disp	
JL/JNGE = Jump on Less/Not Greater or Equal	01111100	disp	
JLE/JNG = Jump on Less or Equal/ Not Greater	01111110	disp	
JB/JNAE = Jump on Below/Not Above or Equal	01110010	disp	
JBE/JNA = Jump on Below or Equal/ Not Above	01110110	disp	
JP/JPE = Jump on Parity/Parity Even	01111010	disp	
JO = Jump on Overflow	01110000	disp	
JS = Jump on Sign	01111000	disp	
JNE/JNZ = Jump on Not Equal/Not Zero	01110101	disp	
JNL/JGE = Jump on Not Less/Greater or Equal	01111101	disp	
JNLE/JG = Jump on Not Less or Equal/ Greater	01111111	disp	
JNB/JAE = Jump on Not Below/Above or Equal	01110011	disp	
JNBE/JA = Jump on Not Below or Equal/Above	01110111	disp	
JNP/JPO = Jump on Not Par/Par Odd	01111011	disp	
JNO = Jump on Not Overflow	01110001	disp	
JNS = Jump on Not Sign	01111001	disp	
LOOP = Loop CX Times	11100010	disp	
LOOPZ/LOOPE = Loop While Zero/ Equal	11100001	disp	
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	11100000	disp	
JCXZ = Jump on CX Zero	11100011	disp	
INT = Interrupt			
Type Specified	11001101	type	
Type 3	11001100		
INTO = Interrupt on Overflow	11001110		
IRET = Interrupt Return	11001111		

Mnemonic and Description	Instruction Code	
PROCESSOR CONTROL	**76543210**	**76543210**
CLC = Clear Carry	11111000	
CMC = Complement Carry	11110101	
STC = Set Carry	11111001	
CLD = Clear Direction	11111100	
STD = Set Direction	11111101	
CLI = Clear Interrupt	11111010	
STI = Set Interrupt	11111011	
HLT = Halt	11110100	
WAIT = Wait	10011011	
ESC = Escape (to External Device)	11011xxx	mod x x x r/m
LOCK = Bus Lock Prefix	11110000	

❑❑❑

Hidden page

Hidden page

Mnemonic	Description	Clock cycles	Number of bytes	Page No.
DIV	Unsigned division			188
	8-bit register	80–90	2	
	16-bit register	144–162	2	
	8-bit memory	(86–96) + EA	2-4	
	16-bit memory	(150–168) + EA	2-4	
ESC	Escape			203
	Register	2	2	
	Memory	8 + EA	2-4	
HLT	Halt	2	1	202
IDIV	Integer division			189
	8-bit register	101–112	2	
	16-bit register	165–184	2	
	8-bit memory	(107–118) + EA	2-4	
	16-bit memory	(171–190) + EA	2-4	
IMUL	Integer multiplication			188
	8-bit register	80–98	2	
	16-bit register	128–154	2	
	8-bit memory	(86–104) + EA	2-4	
	16-bit memory	(134–160) + EA	2-4	
IN	Input from I/O port			180
	Fixed port	10	2	
	Variable port	8	1	
INC	Increment by 1			181
	16-bit register	2	1	
	8-bit register	3	2	
	Memory	15 + EA	2-4	
INT	Interrupt			204
	Type=3	52	1	
	Type≠3	51	2	
INTO	Interrupt if overflow		1	204
	Interrupt is taken	53		
	Interrupt is not taken	4		
IRET	Return from interrupt	24	1	204
JA/	Jump if above/	16/4	2	200
JNBE	Jump if not below or equal			200

Mnemonic	Description	Clock cycles	Number of bytes	Page No.
JAE/	Jump if above or equal/	16/4	2	200
JNB/	Jump if not below/			200
JNC	Jump if not carry			200
JB/	Jump if below/	16/4	2	200
JNAE/	Jump if not above or equal/			200
JC	Jump if carry			200
JBE/	Jump if below or equal/	16/4	2	200
JNA	Jump if not above			200
JCXZ	Jump if CX is zero	18/6	2	200
JE/	Jump if equal/	16/4	2	201
JZ	Jump if zero			200
JG/	Jump if greater/	16/4	2	200
JNLE	Jump if not less or equal			200
JGE/	Jump if greater or equal/	16/4	2	200
JNL	Jump if not less			200
JL/	Jump if less/	16/4	2	200
JNGE	Jump if not greater or equal			200
JLE/	Jump if less or equal/	16/4	2	200
JNG	Jump if not greater			200
JMP	Jump			199
	Intrasegment direct short	15	2	
	Intrasegment direct	15	3	
	Intersegment direct	15	5	
	Intrasegment indirect through memory	18 + EA	2-4	
	Intrasegment indirect through register	11	2	
	Intersegment indirect	24 + EA	2-4	
JNE/	Jump if not equal/	16/4	2	200
JNZ	Jump if not zero			200

Hidden page

Mnemonic	Description	Clock cycles	Number of bytes	Page No.
MOV	Move			**176**
	Accumulator to memory	10	3	
	Memory to accumulator	10	3	
	Register to register	2	2	
	Memory to register	8 + EA	2-4	
	Register to memory	9 + EA	2-4	
	Immediate to register	4	2-3	
	Immediate to memory	10 + EA	3-6	
	Register to SS, DS, or ES	2	2	
	Memory to SS, DS, or ES	8 + EA	2-4	
	Segment register to register	2	2	
	Segment register to memory	9 + EA	2-4	
MOVS/	Move string/		1	**205**
MOVSB/	Move byte string/			**205**
MOVSW	Move word string			**205**
	Not repeated	18		
	Repeated	9 + 17/rep		
MUL	Unsigned multiplication			**187**
	8-bit register	70-77	2	
	16-bit register	118-133	2	
	8-bit memory	(76-83) + EA	2-4	
	16-bit memory	(124-139) + EA	2-4	
NEG	Negate			**185**
	Register	3	2	
	Memory	16 + EA	2-4	
NOP	No operation	3	1	**204**
NOT	Logical NOT			**191**
	Register	3	2	
	Memory	16 + EA	2-4	
OR	Logical OR			**191**
	Register to register	3	2	
	Memory to register	9 + EA	2-4	
	Register to memory	16 + EA	2-4	
	Immediate to accumulator	4	2-3	
	Immediate to register	4	3-4	
	immediate to memory	17 + EA	3-6	

Hidden page

Hidden page

Hidden page

Mnemonic	Description	Clock cycles	Number of bytes	Page No.
XOR	Logical exclusive OR			**192**
	Register with register	3	2	
	Memory with register	9 + EA	2-4	
	Register with memory	16 + EA	2-4	
	Immediate with accumulator	4	2-3	
	Immediate with register	4	3-4	
	Immediate with memory	17 + EA	3-6	

Hidden page

If address is not specified, the location begins where the last D command left off, or at location DS:0 if the command us being typed for the first time. An address may consist of a segment-offset address or just an offset :

```
D   E000:0      (segment-offset)
D   ES:200      (segment register-offset)
D   200         (offset)
```

The default segment is DS, so it is not necessary to specify segment unless we want to dump an offset from another segment location.

A range may be given, telling DEBUG to dump all bytes within the range :

```
D   100  200  ; Dump DS:0100 through 0200
```

Other segment registers or absolute addresses may be used, as shown in the following examples.

Examples

1.	D	Dump 128 bytes from the last referenced location.
2.	D SS:0A	Dump the bytes at offsets 0-A from SS.
3.	D 300:0	Dump 128 bytes at offset zero from segment 0300h.
4.	D 0 200	Dump offset 0-200 from DS.
5.	D 100 L 20	Dump 20th bytes, starting at offset 100h from DS.

E (Enter)

The E command places individual bytes in memory. We must supply a starting memory location where the values will be stored. If only an offset value is entered, the offset is assumed to be from DS. Otherwise, a 32-bit address may be entered, or another segment register may be used. The syntax is :

E address	Enter new byte value at address
E address [list]	Replace the contents of memory starting at the specified address with the values contained in the list.

To begin entering hexadecimal or character data at DS:200, type E 200. Press the space bar to advance to the next byte, and press ENTER to stop. To enter a string into memory starting at location CS:200, type E CS:200 "This is a string".

F (Fill)

The F command fills a range of memory with a single value or a list of values. The range must be specified as two offset addresses or segment-offset addresses. The syntax is :

F range list

Examples

1. F 100 200, ` '	Fill with spaces.
2. F CS:0100 CS:0200, FF	Fill with hex 0FF.
3. F 200 L 30 'A'	Fill 30 hex bytes with the letter 'A', starting at location 200.

G (Execute)

The G command executes the program in the memory. We can specify a starting address and a breakpoint, causing the program to stop at a given address. The syntax is :

G [= startaddress] brkptaddress [brkptaddress...]

If no breakpoint is specified, the program runs until it stops by itself and returns to DEBUG. Up to 10 breakpoints may be specified.

Examples

1. G	Execute from IP to the end of the program.
2. G 100	Execute from the IP to CS:100h and stop.
3. G = 100 500	Begin execution at offset 100h and stop before the instruction at offset 500h

H (Hexarithmetic)

The H command performs addition and subtraction on two hexadecimal numbers, entered in the following syntax :

H value1 value2

Example

H 10 20	Hexadecimal values 10 and 20 are added and subtracted
0030 0010	← displayed by Debugger

Hidden page

Hidden page

T (Trace)

The T command executes one or more instructions from the current CS:IP location or an optional address, if specified. The contents of the registers are shown after each instruction is executed. The syntax is :

T [= address] [, value]

Examples

1. T — Trace one instruction from the current location
2. T 5 — Trace five instructions.
3. T = 5, 10 — Start tracing at CS:5, and trace the next 16 steps.

This command traces individual loop iterations, so we can use it to debug statements within a loop. Also, the T command traces into procedure calls, whereas the P command executes a procedure call in its entirely without tracing.

U (Unassemble)

The U command translates memory into assembly language mnemonics. This is called unassembling or disassembling memory. If we don't specify an address, debugger disassembles from the location where the last U command is left off. If the command is used for the first time after loading debugger, memory is disassembled from location CS : 100. The syntax is :

Syntax 1 : U [address]

Syntax 2 : U [range]

Examples

1. U — Unassemble the next 32 bytes from the current location.
2. U 0 — Unassemble 32 bytes from location 0.
3. U 100, 200 — Unassemble all bytes from offset 100h-200h.

W (Write)

The W command writes a block of memory to a file or to individual disk sectors. If we want to write to a file, it must first be initialized with the N command. (If the file was just loaded either on the DOS command line or with the Load command, we do not need to repeat the Name command.)

Place the number of bytes to be written in BX and CX (BX contains the high 16 bits, and CX contains the low 16 bits). If a file is 256 bytes long, for example, the BX and CX registers will contain the following :

BX = 0000 CX = 0100

Examples

1. N MYFILE.COM	Initialize the filename MYFILE.COM on the default drive.
2. R CX 20	Set the CX register to 20h, the length of the file.
3. W	Write 20h bytes to the file, starting at CS : 100.
4. W 0	Write from location CS : 0 to the file.

❑❑❑

Microprocessors & Interfacing

✎ Chapterwise University Questions with Answers

Hidden page

Hidden page

3 8086 Instruction Set and Assembly Language Programming

Q.1 *Discuss various branch instruction of 8086 microprocessor, that are useful for relocation.* **[April/May-2005, Set-1, 8 Marks; April/May-2006, Set-1, 8 Marks]**

Ans. : Refer section 3.7.

Q.2 *It is necessary to check whether the word stored in location 4000H: A000H is positive number of not. Show all possible ways of testing the above condition and store 00H if the condition is satisfied in location 3000 : 2002. Otherwise store 0FFH.*

[April/May-2005, Set-2, 16 Marks]

Ans. : Refer program 12 in chapter 4.

Q.3 *Distinguish between inter-segment and intra-segment CALL instructions and explain with examples how they are executed.* **[April/May-2005, Set-2, 8 Marks]**

Ans. : Refer section 3.7.1.

Q.4 *Give a neat flow chart and the corresponding 8086 assembly language program for performing bubble sort in an array of N elements of 4-digit Hex number.*

[April/May-2005, Set-2, Set-3, 8 Marks]

Ans. : Refer program 19 in chapter 4.

Q.5 *What is a procedure ? How is a procedure identified as near or far ?*

[April/May-2005, Set-4, 8 Marks]

Ans. : Refer section 3.17.

Q.6 *Discuss the importance of procedures in assembly language programming.*

[Nov./Dec.-2005, Set-1, 8 Marks]

Ans. : Refer section 3.17

Q.7 *What is a recursive procedure ? Write a recursive procedure to calulate the factorial of number N, where N is a two-digit Hex number.*

[Nov./Dec.-2005, Set-2, Set-4, 8 Marks]

Ans. : Refer program 5 of chapter 4.

Q.8 *What are the loop instructions of 8086 ? Explain the use of DF flag in the execution of string instructions.* **[Nov./Dec.-2005, Set-2, 8 Marks]**

Ans. : Refer sections 3.8 and 3.6.

Q. 9 *What instruction set support is provided in 8086 ?* **[Nov./Dec.-2005, Set-2, 4 Marks]**

Ans. : Refer section 3.3.

Q.10 *Develop a far procedure declared as PUBLIC to convert a 4-digit BCD number to its equivalent hex number.* **[Nov./Dec.-2005, Set-3, 8 Marks]**

Ans. : Refer program 13 of chapter 13.

Q.11 *Give the assembly language implementation of the following.*

i) DO WHILE ii) FOR **[Nov./Dec.-2005, Set-4, 8 Marks]**

Ans. : i) DO WHILE

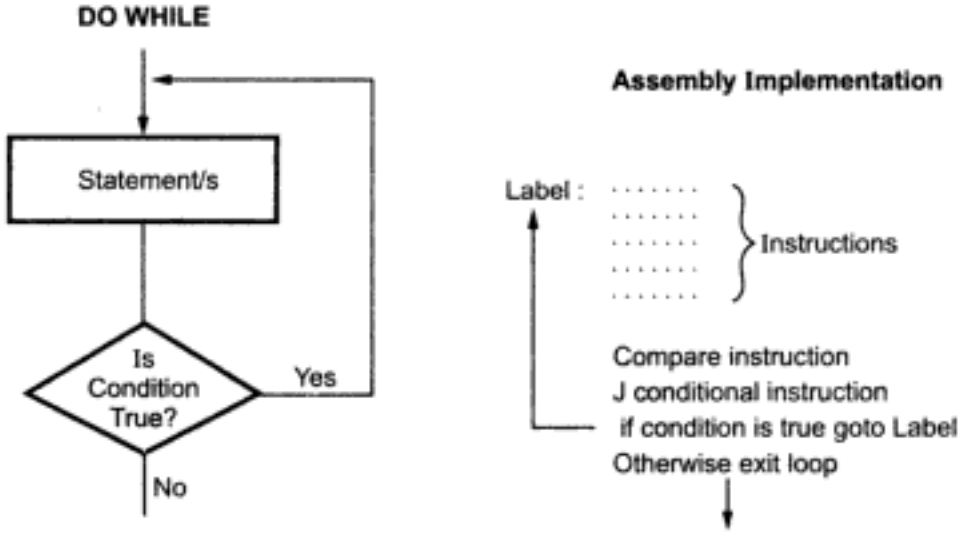


Fig. 1

ii) FOR

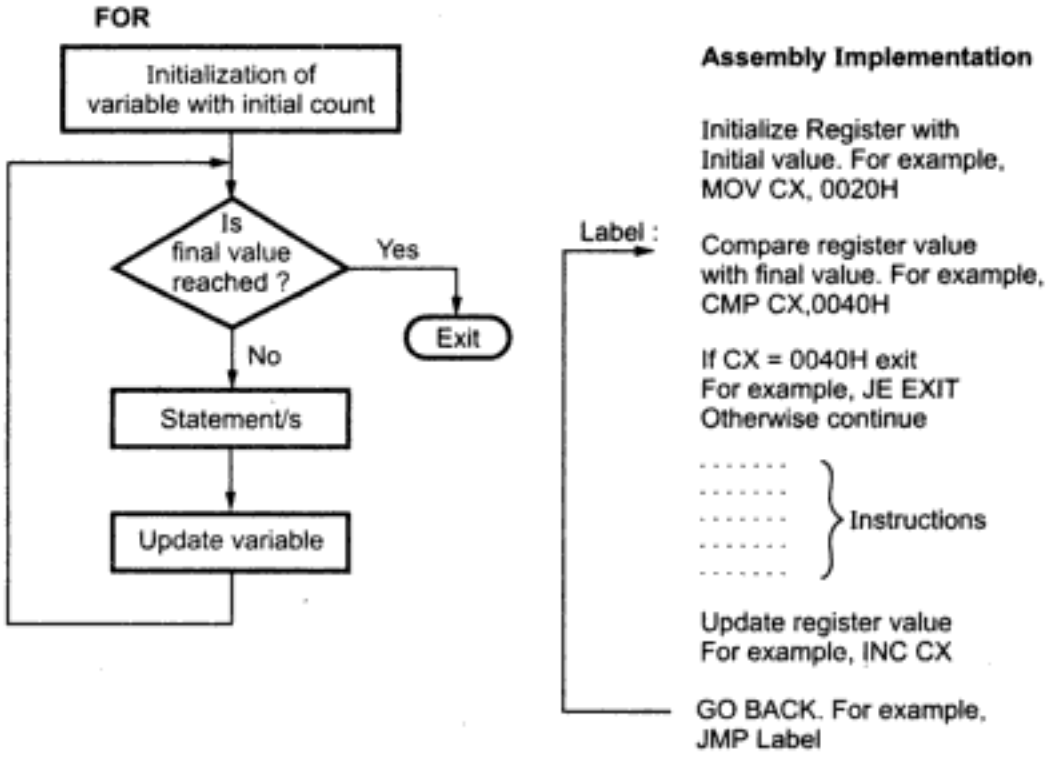


Fig. 2

Q.12 *Explain how IRET instruction is executed ?* **[Nov./Dec.-2005, Set-4, 8 Marks]**

Ans. : Refer section 3.11.

Q.13 *Using a do-while construct, develop a sequence of 8086 instructions that reads a character string from the keyboard and after pressing the enter key the character string is to be displayed again.* **[April/May-2006, Set-1, 8 Marks]**

Ans. :

- `model small`
- `stack 100`
- `data`

```
msl db 10, 13, 'Enter the string $'
buff db 80 dup($)
```

- `code`

```
start : mov ax, @data ; [loads the address of
        mov ds, ax    ; data segment in DS]
        lea bx, buff  ; get the address of buffer
        lea dx, msl   ; display the message
```

```
        mov ah, 09H
        int 21H
back :  mov ah, 01   ; Read character
        int 21H
        mov [BX],AL ; Save the character
        inc BX       ; Increment pointer
        cmp AL, 13   ; Check if it is enter
        jnz back     ; If not read next character
        lea dx, buff ; Display the string
        mov ah, 09H
        int 21h
        mov ah, 4ch;[Exit to
        int 21h     ; DOS]
end start
end
```

Q.14 *Discuss the addressing modes provided by 8086 and explain with examples.*

[April/May-2006, Set-2, 8 Marks]

Ans. : Refer section 3.2.

Q.15 *It is necessary to define a block of data in 8086 assemble language program. The length of the block is 80,000 Bytes. Give the initialization of data segment for the above data. It is necessary to exchange second element and 70000th element in the above. Give the sequence of instructions to perform the above operation.*

[April/May-2006, Set-2, 16 Marks]

Ans. :

```
• model large
data segment
buff db 65535 dup()
data ends

data1 segment
buff1 db 14465 dup()
data1 ends

code segment
    assume cs : code, ds : data
```

```
Start : mov ax, data          ;  [Initialize
        mov ds, ax            ;  data segment]
        lea bx, buff          ;  get the address of buff
        mov CL, [bx + 1]      ;  get data

        mov ax, data1         ;  [Initialize
        mov ds, ax            ;   data1 segment]
        lea bx, buff1         ;  get the address of buff1
        mov si, 4465          ;  load offset
        mov ch, [bx+si]       ;  get data
        mov [bx + si], cl     ;  store data
        mov ax, data          ;  [Initialize
        mov ds, ax            ;   data segment]
        lea bx, buff          ;  get the address of buff
        mov [bx+1], ch        ;  store data

        mov ah, 4CH           ;  [Terminate to
        int 21 H              ;   DOS]
code ends
end start
```

Q.16 *Explain the use of addressing mode. It is necessary to move a byte from location 4000H : 2000H to 4000H : 2005H. Give all possible methods using 8086 addressing modes.* **[April/May-2006, Set-3, 8 Marks]**

Ans. : Refer section 3.2.

Q.17 *Explain in detail the coding template for 8086 MOV instruction.*

[April/May-2006, Set-3, 8 Marks]

Ans. : Refer section 3.19.

Q.18 *It is necessary to declare a program as a public procedure to be accessible by other programs. Give the sequence of assembly language statements. An external program called "fact" is to used in this program. Show the required statements.*

[April/May-2006, Set-3, 8 Marks]

Ans. : Refer section 3.12.3.

Q.19 *It is necessary to check whether the word stored in location 3000H : 2000H is zero or not. Show all possible ways of testing the above condition using different addressing modes and store 0FFH if the condition is satisfied in location 3000H : 2002H. Otherwise store 00H.* **[April/May-2006, Set-4, 16 Marks]**

Ans. : Refer section 3.2.

❑❑❑

4 Assembly Language Programs

Q.1 *Develop an 8086 assembly language program that will determine if a given sub-string is present or not in a main string of characters. Place the result as 'P' if present else place 'N' in memory location 'result'.* **[April/May-2005, Set-4, 8 Marks]**

Ans. : Refer program 18.

Q.2 *Using DF flag and string instructions, write an assembly language program to move a block of data of length N from source to destination. Assume all possible conditions.*

[Nov./Dec.-2005, Set-1, 8 Marks]

Ans. : Refer program 11.

Q.3 *Develop a near procedure to find the GCD of two numbers of 2-digit Hex. Use this procedure to find the GCD of three numbers.* **[Nov./Dec.-2005, Set-3, 8 Marks]**

Ans. : Refer program 22.

❑❑❑

Hidden page

I/O Map :

$\overline{BHE}$	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	HEX address	I/O Device
1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0F00H	1-data
1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0F02H	1-status
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	0F01H	2-data
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	0F03H	2-status
1	0	0	0	0	1	1	1	1	0	0	0	0	0	1	0	0	0F04H	3-data
1	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	0	0F06H	3-status
0	0	0	0	0	1	1	1	1	0	0	0	0	0	1	0	1	0F05H	4-data
0	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1	0F07H	4-status
1	0	0	0	0	1	1	1	1	0	0	0	0	1	0	0	0	0F08H	5-data
1	0	0	0	0	1	1	1	1	0	0	0	0	1	0	1	0	0F0AH	5-status

Instruction sequence to read and store status of each I/O device

```
MOV DX, 0F02H
IN AL, DX              ;   Read status 1
MOV LOC1, AL           ;   Store it
MOV DX, 0F03H
IN AL, DX              ;   Read status 2
MOV LOC2, AL           ;   Store it
MOV DX, 0F06H
IN AL, DX              ;   Read status 3
MOV LOC3, AL           ;   Store it
MOV DX, 0F07H
IN AL, DX              ;   Read status 4
MOV LOC4, AL           ;   Store it
MOV DX, 0F0AH
IN AL, DX              ;   Read status 5
MOV LOC5, AL           ;   Store it
```

Q. 3 *In a home PC with 8088 processor, SRAM is provided from 00000H and EPROM ends with the address of FFFFFH. The capacity of SRAM is 256 KB and that of EPROM is 32 KB. All the chips are of size 32 KB. Give the address map for individual chip and design the complete memory interface.*

[April/May-2005, Set-2, 16 Marks]

Ans. : Refer section 5.10.

Q. 4 *Describe the function of the following pins and their use in 8086 based system.*

a) NMI b) $\overline{LOCK}$ c) $\overline{TEST}$ d) RESET. **[April/May-2005, Set-4, 16 Marks]**

Ans. : Refer section 5.2.

Q. 5 *Describe the function of the following pins in 8086 maximum mode of operation.*

(a) $\overline{TEST}$

(b) RQ/GT_0 and RQ/GT_1

(c) QS_0 and QS_1

(d) S_0, S_1, S_2 **[Nov./Dec.-2005, Set-1, 16 Marks]**

Ans. : Refer section 5.2.

Q. 6 *With a sketch explain 74LS138 decoder and its use.* **[Nov./Dec.-2005, Set-1, 8 Marks]**

Ans. :

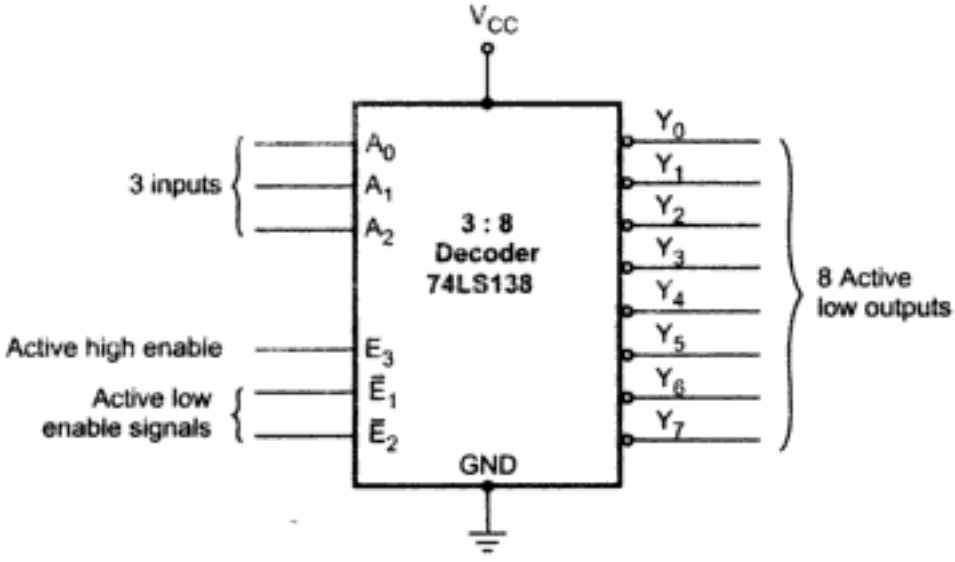


Fig. 2

The IC 74LS138 is a 3 : 8 decoder. It has 3 input lines (select lines), 8 output lines (Active low) and three enable signals : $\overline{E}_1, \overline{E}_2$ and E_3. To make decoder active E_3 should be high, and $\overline{E}_1$ and $\overline{E}_2$ should be low. Once 74LS138 is enabled, according to the inputs A_0, A_1 and A_2 one of the output pin is activated.

Function Table of 74LS138 :

INPUTS						OUTPUTS							
$\overline{E}_1$	$\overline{E}_2$	E_3	A_0	A_1	A_3	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$
H	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	L	X	X	X	H	H	H	H	H	H	H	H
L	L	H	L	L	L	L	H	H	H	H	H	H	H
L	L	H	H	L	L	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	H	H	L	H	H	H	L	H	H	H	H
L	L	H	L	L	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	L	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

The 74LS138 decoder is used for generating chip select signals by decoding the address.

Q. 7 *Generate chip select signals with the help of 74LS138 to six memory chips of size 16 kB, with the address map from 00000H to 17FFFH.*

[Nov./Dec.-2005, Set-1, 8 Marks]

Ans. : The 16 kB memory requires 2^{14} address lines, i.e. A_0 - A_{13}. The remaining address lines are used as a decoder input.

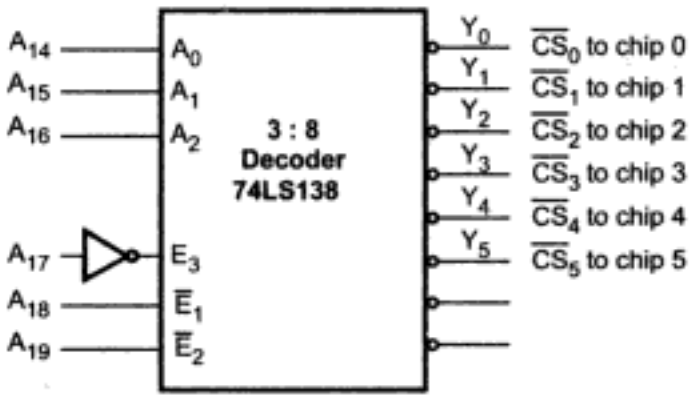


Fig. 3

Hidden page

Q. 10 *A target system based on 8086 processor uses less amount of SRAM. The programs are stored in EPROM that starts from F0000H ends with the address of FFFFFH. The capacity of SRAM is 8 KB interfaced address 00000H. The chip size is 8 KB for EPROM and SRAM. Show the compiete memory interface.*

[Nov./Dec.-2005, Set-2, 16 Marks]

Ans. : Refer section 5.10.

Q. 11 *What is the purpose of ALE, BHE, DT/$\overline{R}$ and $\overline{DEN}$ pins of 8086 ? Show their timing in the system bus cycle of 8086 .* **[Nov./Dec.-2005, Set-3, 8 Marks]**

Ans. : Refer section 5.2.

Q. 12 *Why 8086 memory is mapped into 2 byte wide banks ? What logic levels are found with BHE and A0 when 8086 reads a word from the address 0A0AH ?*

[Nov./Dec.-2005, Set-3, 8 Marks ; April/May-2006, Set-1, Set-4, 8 Marks]

Ans. : Refer section 5.3.

Q. 13 *Distinguish between a memory read and write machine cycle. Draw the timing diagrams in minimum and maximum modes of operation.*

[Nov./Dec.-2005, Set-4, 16 Marks]

Ans. : Refer section 5.6.3.

Q. 14 *In an SDK-86 kit 128 KB SRAM and 64 KB EPROM is provided on system and provision for expansion of another 128 KB SRAM is given. The on system SRAM address starts from 00000H and that of EPROM ends with FFFFFH. The expansion slot address map is from 80000H to 9FFFFH. The size of SRAM chip is 64 KB. EPROM chip size is 16 KB. Give the complete memory interface and also the address map for individual chips.* **[Nov./Dec.-2005, Set-4, 16 Marks]**

Ans. : Please refer Fig. 5 on next page.

Memory map :

$\overline{BHE}$	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Hex Address	Memory
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F8000H	Even
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFEH	EPROM1
0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	F8001H	Odd
0	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	F8FFFH	EPROM2
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H	Even
1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0FFFEH	RAM1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001H	Odd
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0FFFFH	RAM2
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80000H	Even
1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	8FFFEH	RAM2
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	80001H	Odd
0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	8FFFFH	RAM2

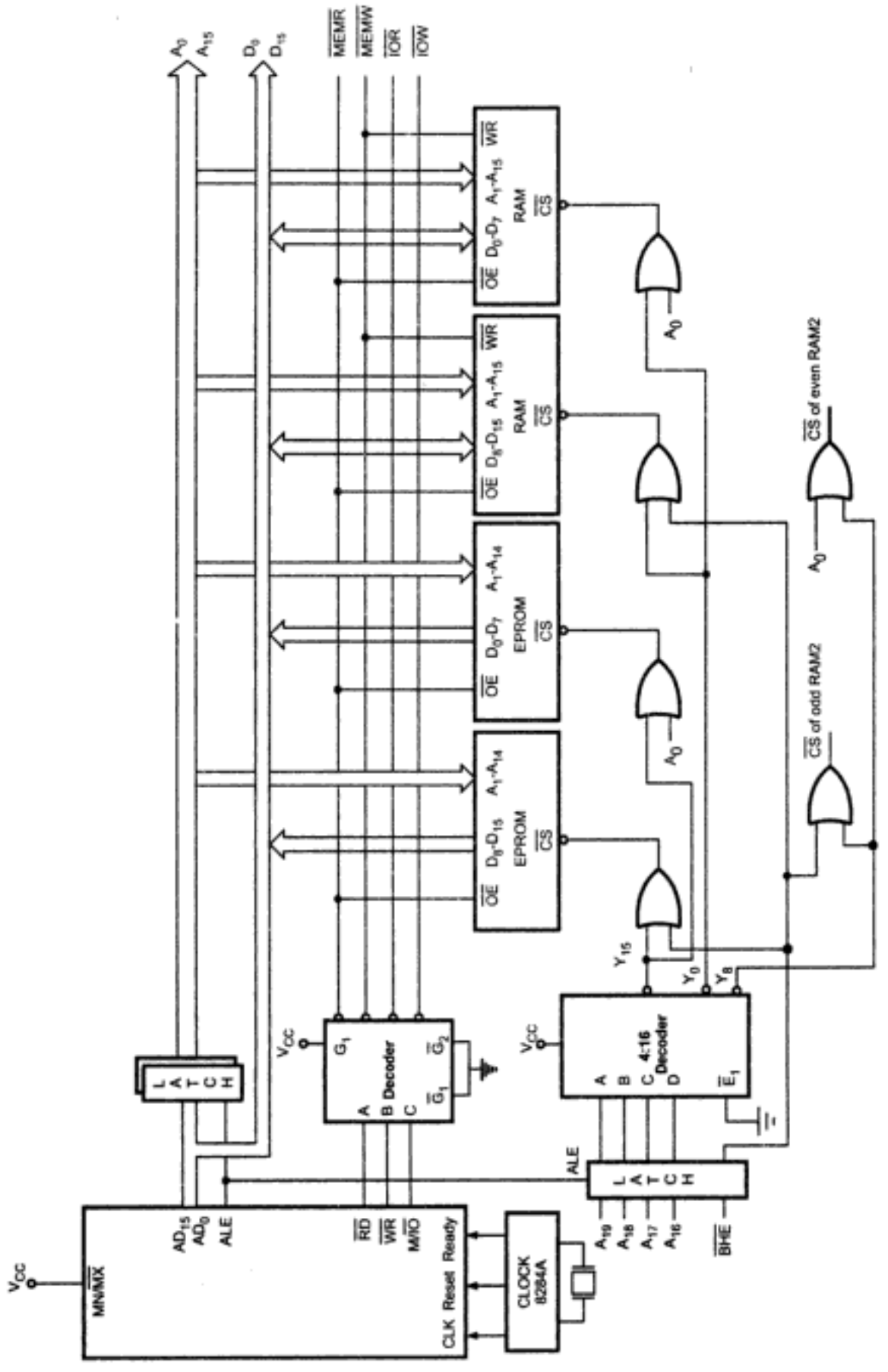


Fig. 5

Hidden page

6 Direct Memory Access (DMA) - 8237/8257

Q. 1 *Explain demand transfer mode and block transfer mode of 8237.*

[Nov./Dec.-2005, Set-1, Set-3, Set-4, 6 Marks]

Ans. : Refer section 6.9.

Q. 2 *Show how 8237's are cascaded to provide more number of DRQ's and explain the operation.* **[Nov./Dec.-2005, Set-1, Set-3,Set-4, 6 Marks]**

Ans. : Refer section 6.9.

Q. 3 *Explain how memory to memory transfer is performed with 8237.*

[Nov./Dec.-2005, Set-1, Set-3, Set-4, 4 Marks]

Ans. : Refer section 6.10.

Q. 4 *Explain with a neat sketch all registers of 8237 and their use in DMA transfer.*

[Nov./Dec.-2005, Set-2, 16 Marks]

Ans. : Refer section 6.12.

Q. 5 *Explain single transfer mode and block transfer mode of 8237.*

[April/May-2006, Set-2, 16 Marks]

Ans. : Refer section 6.9.

Q. 6 *With a neat sketch explain 8237 DMA controller and its application.*

[April/May-2006, Set-4, 8 Marks]

Ans. : Refer sections 6.9 and 6.1.

❑❑❑

7

8255 PPI (Programmable Peripheral Interface)

Q. 1 *Explain how to interface a stepper motor with 4 step input sequence to 8086 based system with the help of hardware design. Write the instruction sequence to move the stepper motor 10 steps in clockwise and 12 steps in anti-clockwise direction.*

[April/May-2005, Set-1, 16 Marks]

And. : Refer section 7.12.

Q. 2 *Explain why 8255 ports are divided into two groups. Discuss how these groups are controlled in different modes of operation. Explain different control signals and their associated pins for bi-directional I/O mode of operation.*

[April/May-2005, Set-2, 16 Marks; Nov./Dec.-2005, Set-3, 16 Marks ; April/May-2006, Set-1, Set-2, 8 Marks]

And. : Refer section 7.4.

Q. 3 *Interface a 12-bit DAC to 8255 with an address map of 0C00H to 0C03H. The DAC provides output in the range of + 5 V to – 5 V. Write the instruction sequence.*

(a) For generating a square wave with a peak to peak voltage of 4V and the frequency will be selected from memory location 'F'.

(b) For generating a triangular wave with a maximum voltage of + 3V and a minimum of – 2 V. **[Nov./Dec. - 2005, Set-1, 16 Marks]**

Ans. :

Interface :

Please refer Fig. 1 on next page.

$$\text{Resolution} = \frac{10\text{ V}}{2^{12} - 1}$$

$$= 2.442\text{ mV}$$

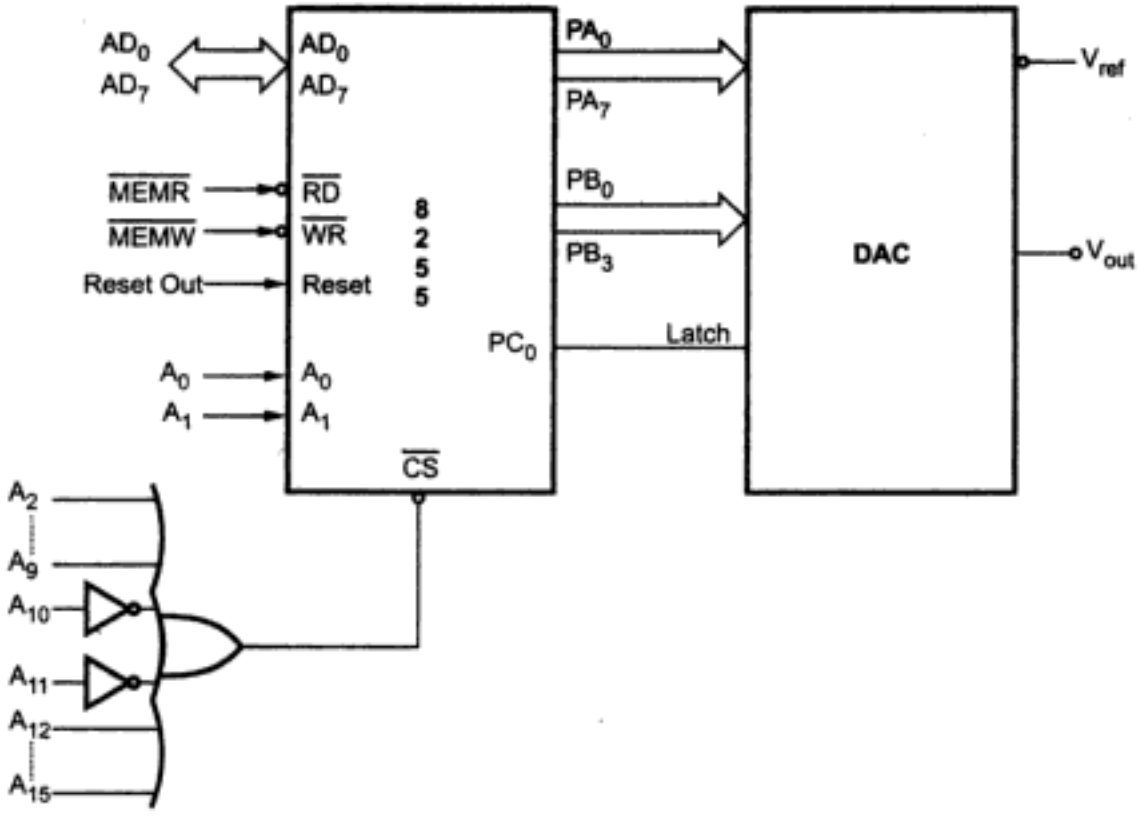


Fig. 1

Digital data	Equivalent Analog Output Voltage
000H	– 5 V
FFFH	+ 5 V
7FFH	0 V
4CCH	– 2 V
CCCH	3 V
E65H	4 V
19AH	– 4 V

a) Generating square wave :

```
        LXI SP,27FFH  ;  Initialize stack pointer
        MVI A, 80H    ;  Initialize 8255 to configure
        STA 0C03H     ;  PA, PB and PC as output

Loop :  MVI A, 9AH    ;  Load and send digital data
        STA 0C00H     ;  corresponds to -4V
```

```
        MVI A, 01H
        STA 0C01H
        MVI A, 01H      ;  Enable latch signal
        STA 0C02H
        NOP
        MVIA, 00H
        STA 0C02H
        CALL Delay      ;  Wait for OFF period
        MVI A, 65 H     ;  Load and send digital data
        STA 0C00H       ;  corresponds to + 4 V
        MVI A, 0EH
        STA 0C01H
        MVI A, 01H      ;  Enable latch signal
        STA 0C02H
        NOP
        MVI A, 00H
        STA 0C02H
        CALL Delay      ;  Wait for ON period
        JMP LOOP        ;  Repeat

Delay : LDA 'F'         ;  Read the delay count which is
                        ;  inversely proportional to frequency
BACK :  DCR A           ;  Decrement count
        JNZ BACK        ;  Check if count = 0; otherwise repeat
        RET             ;  Return to main program
```

b) Generating Triangular Wave :

```
        LXI SP, 27FFH     ;  Initialize stack pointer
        MVI A, 80 H       ;  Initialize 8255 to configure
        STA 0C03H         ;  PA, PB and PC as output

BACK :  LXI H, 04CCH      ;  Load and send digital data
LOOP1 : SHLD 0C00H        ;  correspond to -2V
        CALL LATCH
        INX H             ;  Increment digital data
```

```
        MOV A, L          ;  Check digital data for positive
        CPI CCH           ;  peak output (+3)
        JNZ LOOP1
        MOV A, H
        CPI 0CH
        JNZ LOOP1
LOOP2 : SHLD 0C00H        ;  Send digital data
        CALL LATCH
        DCX H             ;  Decrement digital data
        MOV A, L          ;  Check digital data for negative
        CPI CCH           ;  peak output (-2)
        JNZ LOOP2
        MOV A, H
        CPI 04H
        JNZ LOOP2
        JMP BACK          ;  Repeat
LATCH : MVI A, 01H        ;  Enable latch signal
        STA 0C02H
        NOP
        MVI A, 00H
        STA 0C02H
```

Q. 4 *Write the necessary instruction sequence to initialize 8255 with address 0C00H to 0C03H for the following combinations.*

a) Port A as input port in mode 1 and port B as input port in mode 1 without the interrupt driven I/O.

b) Port A in mode 2 as output port and port B as input port in mode 0 with interrupt driven I/O.

c) Port A in mode 0, port C upper half as input ports and port B as input port in mode 1 with interrupt driven I/O.

d) Port A as output port in mode 1 with active interrupt, port B as input port in mode 0 and port C lower half as output port in mode 0.

[Nov./Dec.-2005, Set-4, 16 Marks]

Ans. : Refer section 7.5.

Q. 5 *It is necessary to initialize interrupt for mode 1 operation of port -A as input and port-B as output in the same mode with the 8255 address map of 0400H to 0700H. Give the complete hardware design to interface 8255 to 8086 processor with this address map. Write the instruction sequence for the initialization of 8255 in the above modes. Give the instruction sequence to change the operation modes of port A, port B, port C lower-half and port B to mode 0 input ports.*

[April/May-2006, Set-4, 16 Marks]

Ans. : Refer sections 7.7 and 7.5.

❑❑❑

Hidden page

Q. 7 *Address 000E0H in the interrupt vector table contains 4132H and address 000E2H contains 0040H.*

i) To what interrupt type do these locations correspond ?

ii) What is the starting address for the interrupt service routine ?

[April/May - 2005, Set-3, Set-4, 4 Marks]

Ans. : Interrupt vector Table :

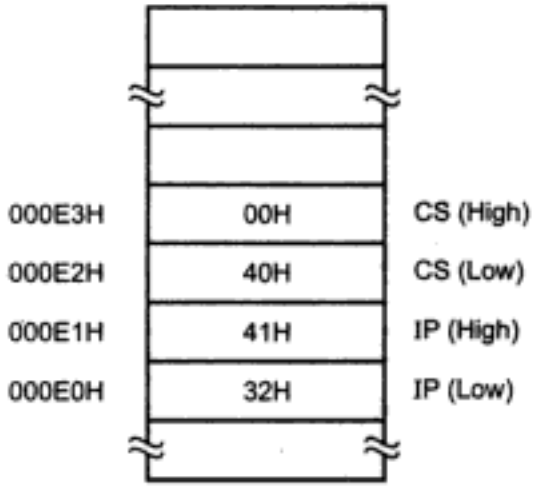


Fig. 1

i) Interrupt type = E 0 H/4 = 224/4 = 56 in decimal

ii) CS = 0040H IP = 4132 H

$\therefore$ 0 0 4 0 0

\+ 4 1 3 2

Starting address of ISR = 0 4 5 3 2 H

Q. 8 *What is the purpose of operational command words of 8259 ? Explain their format and the use.* **[Nov./Dec.-2005, Set-1, 8 Marks]**

Ans. : Refer section 8.5.5.

Q. 9 *What detailed hardware and the associated algorithm, explain how a real time clock will be implemented in an 8086 based system ?* **[Nov./Dec.-2005, Set-2, 16 Marks]**

Ans. : Please refer Fig. 2 on next page.

Algorithm (Initialization) :

1. Initialize clock i.e. Hours, Minutes and Seconds.
2. Load the address of ISR in interrupt vector table at 0008H.
3. Wait for interrupt.

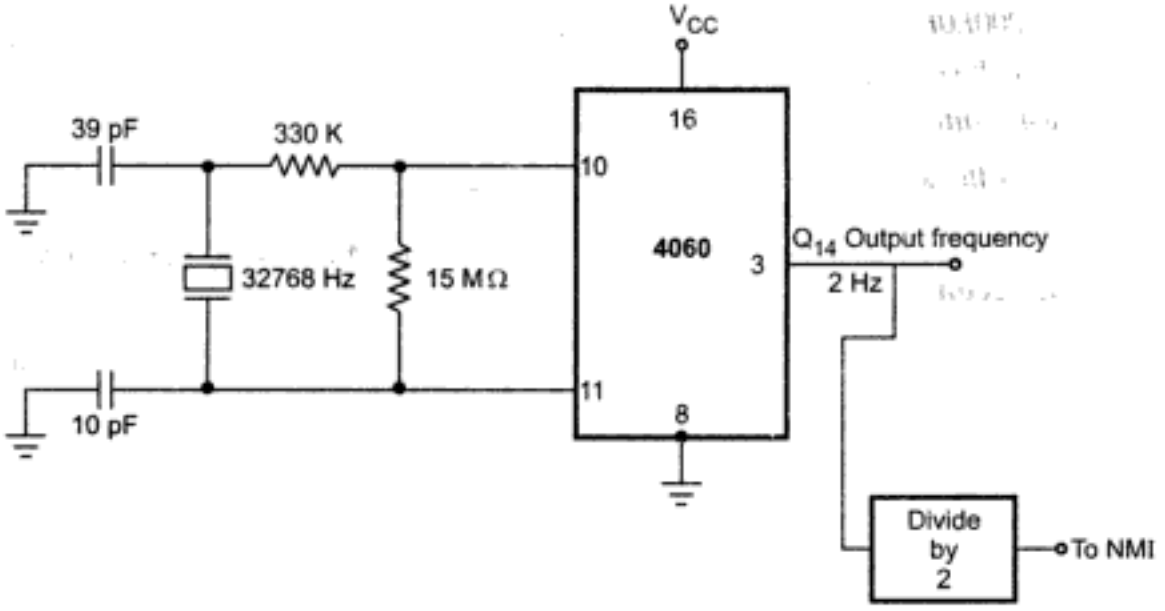


Fig. 2

Algorithm (Interrupt service routine) :

1. Save registers.
2. Increment seconds.
3. If seconds = 60, make seconds = 0 and increment minutes.
4. If minutes = 60, make minutes = 0 and increment hours.
5. If hours = 13, make hours = 1.
6. Return to main program.

Q. 10 *Write an initialization sequence for an 8259 that is the only 8259 in an 8086 based system, with an even address of 0F0H that will cause.*

a) Request to the edge triggered mode

b) IR_0 request to an interrupt type 30

c) SP/EN to output a disable signal to the data-bus transceivers

d) The IMR to be cleared

e) The highest priority interrupt will be IR_6. **[Nov./Dec.-2005, Set-3, 16 Marks]**

Ans. : Refer section 8.5.5.

Q. 11 *Draw the block diagram of 8259 and explain each block ?*

[Nov./Dec.-2005, Set-4, 8 Marks]

Ans. : Refer section 8.5.2.

Q. 12 *Under what conditions type 0 interrupt is initiated ? List out the instructions that may cause type 0 interrupt.* **[April/May - 2006, Set-1, Set-2,Set-3, 6 Marks]**

Ans. : Refer section 8.3.1.

Q. 13 *Explain the following terms with reference to 8259.*

(a) END of interrupt

(b) Automatic rotation

(c) Poll command

(d) Read resister command **[April/May-2006, Set-4, 16 Marks]**

Ans. : Refer section 8.5.4.

❑❑❑

10 Serial Communication

Q. 1 *Discuss the Command instruction and Status register format of 8251.*

[April/May-2005, Set-1, Set-2, 8 Marks]

Ans. : Refer sections 10.4.4 and 10.4.5.

Q. 2 *Draw the block diagram of 8251 and explain each block.*

[April/May-2005, Set-3, 8 Marks]

Ans. : Refer section 10.4.3.

Q. 3 *Draw the flowchart showing how synchronous serial data can be sent from a port line using software routine.* **[April/May-2005, Set-3, 8 Marks]**

Ans. : Refer section 10.4.6.

Q. 4 *Discuss the serial data transmission standards and their specifications.*

[April/May-2005, Set-4, 8 Marks]

Ans. : Refer section 10.5.

Q. 5 *A terminal is transmitting asynchronous serial data at 2400 bd. What is the bit time? Assuming 7 data bits, a parity bit and 1 stop bit how long does it take to transmit one character ?* **[April/May-2005, Set-4, 8 Marks]**

Ans. : Refer section 10.5.

Q. 6 *Write an initialization sequence to operate 8251 in asynchronous mode with 8-bit character size, baud rate factor 64, two stop bits and odd parity enable. The 8251 is interfaced with 8086 at address 082H.* **[April/May-2006, Set-1, 8 Marks]**

Ans. : Mode word for given specification is as follows.

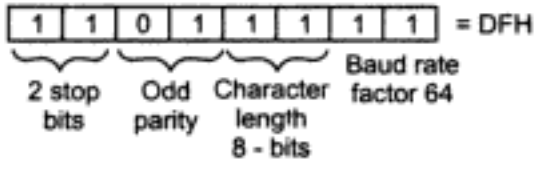


Fig. 1

Hidden page

Q. 10 *How do we connect RS-232C equipment*

i) To data terminal type devices ?

ii) To serial port of SDK – 86, RS-232C connection ?

[April/May-2006, Set-3, 10 Marks, Set-4, 8 Marks]

Ans. : Refer section 10.5.

Q. 11 *Give the specifications of RS-232C.* **[April/May-2006, Set-3, 6 Marks]**

Ans. : Refer section 10.5.

❑❑❑

11

8051 Microcontroller

Q. 1 *Discuss the following signal descriptions*

i) $\overline{INT}_0/\overline{INT}_1$ *ii) TXD*

ii) T_0 *and* T_1 *iii)* $\overline{RD}$ **[April/May-2005, Set-1, 8 Marks]**

Ans. : Refer section 11.3.1.

Q. 2 *Draw and discuss the formats and bit definitions of the following SFRs in 8051 microcontroller.*

i) TMOD ii) PSW **[April/May-2005, Set-1, 8 Marks]**

Ans. : Refer sections 11.7.2 and 11.3.6.3.

Q. 3 *Draw and discuss the formats and bit definitions of the following SFR's in 8051 microcontroller.*

a) PSW b) IE c) SCON d) TMOD

[April/May-2005, Set-2, 8 Marks ; April/May-2006, Set-3, 8 Marks]

Ans. : Refer sections 11.3.6.3, 11.9, 11.8 and 11.7.2.

Q. 4 *Discuss the interrupt structure of 8051. Mention the priority. Explain how least priority is made as highest priority.* **[April/May-2005, Set-3, 8 Marks ; Nov./Dec.-2005, Set-1, 8 Marks, April/May-2006, Set-2, 8 Marks]**

Ans. : Refer section 11.9.

Q. 5 *Draw and discuss the formats and bit definitions of the following SFR's in 8051 microcontroller.*

a) IP b) TMOD c) TCON d) SCON

[April/May-2005, Set-4, 16 Marks ; Nov./Dec.-2005, Set-2, 8 Marks]

Ans. : Refer sections 11.9.1, 11.7.2 and 11.8.

Q. 6 *Explain the alternate functions of Port 0, Port 2 and Port 3.*

[Nov./Dec.-2005, Set-1, 8 Marks ; Nov./Dec.-2005, Set-3, 8 Marks ; April/May-2006, Set-2, 8 Marks]

Ans. : Refer section 11.3.1.

Q. 7 *Discuss the following signal descriptions.*

i) ALE/PROG ii) $\overline{EA}/V_{PP}$ iii) $\overline{PSEN}$ iv) RXD **[Nov./Dec.-2005, Set-2, 8 Marks]**

Ans. : Refer section 11.3.1.

Q. 8 *Discuss the register set of MCS-51 family of microcontrollers.*

[Nov./Dec.-2005, Set-3, 8 Marks]

Ans. : Refer section 11.3.6.

Q. 9 *Discuss the following signal descriptions.*

(a) ALE/PROG (b) $\overline{EA}/V_{PP}$ (c) $\overline{PSEN}$ (d) RXD
(e) $\overline{INT_0}/\overline{INT_1}$ (f) TXD (g) T_0 and T_1 (h) $\overline{RD}$

[April/May-2006, Set-1, 16 Marks]

Ans. : Refer section 11.3.1.

❑❑❑

Hidden page

Hidden page

Contents

- An overview of 8085, Architecture of 8086, Microprocessor, Special functions of general purpose registers, 8086 flag register and function of 8086 flags.
- Addressing modes of 8086, Instruction set of 8086, Assembler directives simple programs, Procedures, and Macros.
- Assembly language programs involving logical, Branch and Call instructions, Sorting, Evaluation of arithmetic expressions, String manipulation.
- Pin diagram of 8086-Minimum mode and maximum mode of operation, Timing diagram, Memory interfacing to 8086 (Static RAM and EPROM), Need for DMA, DMA data transfer method, Interfacing with 8237/8257.
- 8255 PPI-Various modes of operation and interfacing to 8086, Interfacing keyboard, Displays, Stepper motor and actuators, D/A and A/D converter interfacing.
- Interrupt structure of 8086, Vector interrupt table, Interrupt service routines, Introduction to DOS and BIOS interrupts, 8259 PIC architecture and interfacing cascading of interrupt controller and its importance.
- Serial data transfer schemes, Asynchronous and synchronous data transfer schemes, 8251 USART architecture and interfacing, TTL to RS 232C and RS232C to TTL conversion, Sample program of serial data transfer, Introduction to High-speed serial communications standards, USB.
- 8051 Microcontroller architecture, Register set of 8051, Modes of timer operation, Serial port operation, Interrupt structure of 8051, Memory and I/O interfacing 8051.

First Edition : 2009

Price INR 285/-

10 Digit: ISBN 81-8431-125-7

13 Digit: ISBN 978-81-8431-125-9

Technical Publications Pune

 # 1, Amit Residency, 412 Shaniwar Peth, Pune - 411030, M.S., India.

Telefax : +91 (020) 24495496/97, Email : technical@vtubooks.com

Visit us at : **www.vtubooks.com**